A Time Complexity Lower Bound for Randomized Implementations of Some Shared Objects

(PRELIMINARY VERSION)

Prasad Jayanti
Department of Computer Science
Dartmouth College
Hanover, NH 03755

prasad@cs.dartmouth.edu

Abstract

Many recent wait-free implementations are based on a shared-memory that supports a pair of synchronization operations, known as LL and SC. In this paper, we establish an intrinsic performance limitation of these operations: even the simple wakeup problem [16], which requires some process to detect that all n processes are up, cannot be solved unless some process performs \( \Omega(\log n) \) shared-memory operations. Using this basic result, we derive a \( \Omega(\log n) \) lower bound on the worst-case shared-access time complexity of n-process implementations of several types of objects, including fetch&increment, fetch&multiply, fetch&and, queue, and stack. (The worst-case shared-access time complexity of an implementation is the number of shared-memory operations that a process performs, in the worst-case, in order to complete a single operation on the implementation.)

Our lower bound is strong in several ways: it holds even if (1) shared-memory has an infinite number of words, each of unbounded size, (2) shared-memory supports move and swap operations in addition to LL, SC, and validate, (3) implementation employs randomization (in this case, the worst-case expected shared-access time complexity is \( \Omega(\log n) \)), and (4) each process applies only one operation on the implemented object. Finally, the lower bound is tight: if the size of shared registers is not restricted, the universal construction of Afek, Dauber, and Touitou [1] (after two minor modifications) has \( O(\log n) \) worst-case shared-access time complexity.

An n-process universal construction can be instantiated with the sequential specification of any type \( T \) to obtain a wait-free implementation of an atomic object of type \( T \) that \( n \) concurrent processes can share. A universal construction is oblivious if it does not exploit the semantics of the type that it is instantiated with. Our lower bound implies that for any shared object \( O \) implemented using any oblivious universal construction, no matter what \( O \)'s type is, in the worst-case a process performs \( \Omega(\log n) \) operations on shared-memory in order to complete a single operation on \( O \). Thus, if our goal is to implement shared objects whose operations run in sublogarithmic time (preferably constant time), oblivious universal constructions cannot be useful; the design of sublogarithmic time implementations must necessarily exploit the semantics of the type being implemented.

1 Introduction

In asynchronous multiprocess systems, software implementations of shared objects are typically based on locking. Since locking has several drawbacks, such as convoying, priority inversion, and deadlock, recent research has focussed on lock-free implementations. Specifically, a wait-free implementation guarantees that if a process invokes an operation on the implementation and repeatedly takes steps, its operation will eventually complete, regardless of the speeds of other processes. Many recent wait-free implementations are based on a shared-memory that supports a pair of synchronization operations, known as LL and SC [33, 9, 19, 22, 1, 31, 4, 3, 7, 2, 30]. These operations work as follows. LL\((a)\) returns the value at location \( a \). SC\((a,v)\) either changes the value at location \( a \) to \( v \) and returns true, or has no effect on location \( a \) and returns false. Correspondingly, we say the SC is successful or unsuccessful. Specifically, if process \( P \) applies LL\((a)\), a subsequent SC\((a,v)\) by \( P \) succeeds if no other process performs a successful SC operation at location \( a \) in the interim. The trend towards the use of LL and SC in recent research is due to their availability on some existing architectures [26, 32, 21], and the fact that they are easy to use and often yield efficient shared-object implementations.

In this paper, we establish an intrinsic performance limitation of implementations that are based on LL and SC by deriving a lower bound on their worst-case shared-access time complexity—the number of operations that a process performs on shared-memory, in the worst-case, in order to complete a single operation on the implemented shared object. Since an access to shared-memory takes one or two orders of magnitude longer than a local computation step [10], the wall-clock time to execute an operation on a shared-object implementation is mostly dictated by the shared-access time complexity of the implementation. This makes a lower bound on this complexity measure useful.

1.1 The lower bound

Consider the wakeup problem for \( n \) processes [16], specified as follows: (1) Every process terminates in a finite number of
its steps, returning either 0 or 1. (2) In every run in which all processes terminate, at least one process returns 1, and (3) In every run in which one or more processes return 1, every process takes at least one step before any process returns 1. Intuitively, the problem requires the process that wakes up last to detect that every other process is up.

We prove the following basic result: For all algorithms to the wakeup problem in which processes communicate by applying LL, SC, and validate operations on shared-memory, there is a run in which some process performs Ω(log n) shared-memory operations.

The above result is then applied to derive a Ω(log n) lower bound on the worst-case shared-access time complexity of n-process implementations of several types of objects, including log n-bit fetch&increment object, n-bit fetch&multiply object, n-bit fetch&and object, and a queue or a stack that can contain at least n items. Our lower bound is strong in several ways, as described below.

1. The lower bound holds even if shared-memory has an infinite number of words and each word has an unbounded size.

2. The lower bound holds even if shared-memory supports move and (register to memory) swap operations, in addition to LL, SC, and validate. This is significant since move operation, by itself, is universal [18], and swap is strictly stronger than the standard write operation. (All operations are defined in Section 3.)

3. The lower bound holds even for randomized universal constructions. In this case, the worst-case expected shared-access time complexity is Ω(log n).

4. The lower bound is proved for single-use implementations, where each process applies at most one operation on the implemented object.

5. Finally, the lower bound is tight: with two minor modifications, the Group-Update universal construction of Afek, Dauber, and Touitou [1] becomes a universal construction (based on LL/SC), then no matter what the object’s type is, in the worst-case a process performs Ω(log n) shared-memory operations in order to complete a single operation on the implementation.

Some LL/SC based implementations have achieved a worst-case time complexity that does not depend on n. Examples include implementations of compare&swap [4], W-word Weak LL/SC [4, 29], and generalized snapshots and buffer [24]. Our lower bound implies that any such constant time implementations must necessarily exploit the semantics of the type of object being implemented. Specifically, such implementations cannot be obtained from any oblivious universal construction.

2 Related work

The wakeup problem was defined by Fischer, Moran, Rudich, and Taubenfeld and solved in a different context—using an arbitrarily initialized shared-memory that supports the read-modify-write operation [16]. Some ideas in our lower bound proof are similar to the ones employed by Attiya, Lynch, and Shavit to prove a lower bound of Ω(log n) on the round complexity of any algorithm for approximate agreement from single-writer read/write registers [8].

The lower bounds in this paper are in the same spirit as some existing lower bounds. Cypher’s result [14], together with either Mellor-Crummey and Scott’s result [27] or Craig’s result [12, 13], implies that there is no constant time implementation of a swap object from LL/SC or compare&swap. Attiya and Dagan prove that any implementation of binary LL/SC operations from LL/SC operations takes Ω(log log * n) time [7]. Moir points out that the results by Anderson [5] and Cypher [14] imply that there is no constant time implementation of fetch&add from compare&swap or LL and SC [28].

The first universal construction is due to Herlihy [17, 18]. The first lower bound on the complexity of randomized wait-free implementations is due to Fich, Herlihy, and Shavit [15], where they prove that Ω(√n) registers are needed in any randomized implementation of consensus. Aspnes proves the first lower bound on the time complexity of randomized wait-free implementations—Ω(n^2/ log^2 n)—for imple-
menting consensus from registers [6]. Jayanti, Tan, and Towseg prove that if oblivious universal constructions are based on consensus objects (as opposed to compare-\&-swap or LL and SC), their shared-access time complexity is $\Omega(n)$ [25]. This bound holds only if the universal construction is deterministic and implements long-lived objects (as opposed to single-use objects).

Chandra, Jayanti, and Tan present a $O(\log^2 n)$ wait-free implementation of any closed object from LL/SC [11]. The class of closed objects includes some objects (e.g., fetch\&increment) for which $\Omega(\log n)$ lower bound is proved in the present paper.

## 3 Model of computation

We adopt the standard asynchronous shared-memory model. In the following, we describe and define only those features that are specific to our model. Our system consists of $n$ processes, $p_0, \ldots, p_{n-1}$, and shared-memory—an infinite number of shared registers, $R_0, R_1, \ldots$, each of an unbounded size. If $p_i$ is in a termination state, $p_i$ has no next step. Otherwise, two types of steps are available to $p_i$: (1) $p_i$ tosses a coin locally, obtains one of the elements of COIN-RANGE as the outcome, where COIN-RANGE is an arbitrary set, and changes state, or (2) $p_i$ performs an operation on shared-memory, receives a response, and changes state.

Five types of operations—LL, SC, validate, swap, and move—are supported by the shared-memory. The state of each shared register $R$ is described by two quantities, $\text{value}(R)$ and $\text{Pset}(R)$ ($\text{Pset}$ stands for process set). The effect of each type of operation is as follows (below, let $A$ be $\text{value}(R)$ and $\text{Pset}(R)$, respectively, in $R_i$’s current state):

- **LL($R_i$)** by $p_i$ changes $\text{Pset}(R_i)$ to $A \cup \{p_i\}$ and returns $u$. If $p_i \in A$, $\text{SC($R_i,v$)}$ by $p_i$ changes $\text{value}(R_i)$ to $v$, $\text{Pset}(R_i)$ to $\emptyset$, and returns $\text{(true, u)}$ (we say SC is successful). If $p_i \notin A$, $\text{SC($R_i,v$)}$ by $p_i$ returns $\text{(false, u)}$ (we say SC is unsuccessful).

- **validate($R_i$)** by $p_i$ returns $\text{(true, u)}$. Otherwise, **validate($R_i$)** by $p_i$ returns $\text{(false, u)}$.

- **swap($R_i,v$)** by $p_i$ changes $\text{value}(R_i)$ to $v$, $\text{Pset}(R_i)$ to $\emptyset$, and returns $u$. (Because it returns the previous value, the swap operation is more powerful than the standard write operation.)

- **move($R_i,R_k$)** by $p_i$ changes $\text{value}(R_k)$ to $u$, $\text{Pset}(R_k)$ to $\emptyset$, and returns $\text{ack}$ (The state of $R_j$ remains unchanged.)

In the standard definition, the response of an SC or a validate operation includes only the boolean, not the previous value. Thus, the above definitions of these operations are stronger. Also, there is no need to include a read operation in the above set of operations: to read $R_j$ (without affecting $R_j$’s state), a process can perform validate($R_j$).

A run is a finite or infinite alternating sequence of configurations and events, starting from the initial configuration. We give the scheduler the “standard” power: in determining which process takes the next step, the scheduler can look at the run up to that point, but it cannot influence or predict the outcomes of future coin tosses. Formally, a scheduler is a function $\psi$ that maps each finite run to a process.

A run $R$ is a terminating run if it is a finite run and, in the final configuration of $R$, every process is in a terminating state. A run $R$ is unextendable if $R$ is either an infinite run or a terminating run. For a run $R$, let $t(p_i,R)$ be the number of $p_i$’s shared-memory steps in $R$ and $t(R)$ be $\max\{t(p_i,R) \mid 0 \leq i \leq n-1\}$. The expected shared-access time complexity of algorithm $\text{alg}$ against scheduler $\psi$, denoted $t(\text{alg},\psi)$, is $\sum t(R) \Pr(R)$, where the summation is over all unextendable runs $R$ of $\text{alg}$ permitted by the scheduler $\psi$. The worst-case expected shared-access time complexity of $\text{alg}$ is the maximum, over all schedulers $\psi$, of $t(\text{alg},\psi)$.

### Lemma 3.1
Let $\text{alg}$ be an algorithm that terminates with probability $c$. Suppose there is a scheduler $\psi$ such that in all terminating runs of $\text{alg}$ permitted by $\psi$, some process performs at least $k$ operations on shared-memory. Then, the worst-case expected shared-access time complexity of $\text{alg}$ is at least $ck$.

## 4 Limiting the influence of move

Suppose that each $p_i \in \{p_0,\ldots,p_{n-1}\}$ has move($R_i,R_{i+1}$) to perform. If $p_0,\ldots,p_{n-1}$ take steps, in that order, $R_0$’s original value will end up in $R_n$. If a process later reads $R_n$, it can infer that that all of $p_0,\ldots,p_{n-1}$ took a step in the past. Thus, the above schedule reveals too much information to processes and thus not helpful when proving a lower bound. Consider an alternative schedule in which all even numbered processes ($p_0,p_2,\ldots$) take their step before the odd numbered processes. This causes $R_i$ ($1 \leq i \leq n$) to get either the original value of $R_{i-1}$ (if $i$ is odd) or the original value of $R_{i-2}$ (if $i$ is even). Most importantly, for any $R_i$, there are only one or two processes (precisely, $p_{i-1}$ if $i$ is odd, and $\{p_{i-2},p_{i-1}\}$ if $i$ is even) that are responsible for what is moved into $R_i$. In particular, it is possible to schedule only $p_{i-2}$ and $p_{i-1}$ and still guarantee that $R_i$ will have the same final value as when the moves of all processes are scheduled. Thus, if a process later reads any one register $R_i$, the most it can infer is that $p_{i-2}$ and $p_{i-1}$ took their steps. Thus, in the alternative schedule, the move operations reveal much less information to processes. In this section, we show that, regardless of what move operations processes have to perform, it is possible to schedule them in such a way that they do not reveal much information to processes.

We use the following notation throughout this section. $S \subseteq \{p_0,\ldots,p_{n-1}\}$ is the set of all processes that have a move operation to perform. The function $f$ specifies the exact operation of each process in $S$: for all $p \in S$, if $f(p) = (R_i,R_j)$, then $p$’s operation is move($R_i,R_j$). In particular, it is possible to schedule only $p_{i-2}$ and $p_{i-1}$ and still guarantee that $R_i$ will have the same final value as when the moves of all processes are scheduled. Thus, if a process later reads any one register $R_i$, the most it can infer is that $p_{i-2}$ and $p_{i-1}$ took their steps. Thus, in the alternative schedule, the move operations reveal much less information to processes. In this section, we show that, regardless of what move operations processes have to perform, it is possible to schedule them in such a way that they do not reveal much information to processes.

We define source($R_i,\sigma,\langle S,f \rangle$) and movers($R_i,\sigma,\langle S,f \rangle$), inductively. Informally, if source($R_i,\sigma,\langle S,f \rangle$) = $R_j$ and movers($R_i,\sigma,\langle S,f \rangle$) = $q_1,q_2,\ldots,q_m$, it means that, after applying the schedule $\sigma$, $R_j$’s original value moves into $R_i$, and the operations of $q_1,q_2,\ldots,q_m$, in that order, are responsible for this. A precise inductive definition is as follows:

1. $\sigma = \lambda$: for all $R_i$, source($R_i,\lambda,\langle S,f \rangle$) = $R_i$, and movers($R_i,\lambda,\langle S,f \rangle$) = $\lambda$.
2. $\sigma = \sigma' \cdot p$: if $f(p) = (R_j,R_k)$,
   
   \[\text{source}(R_i,\sigma,\langle S,f \rangle) = \text{source}(R_j,\sigma',\langle S,f \rangle),\]
   
   \[\text{movers}(R_i,\sigma,\langle S,f \rangle) = \text{movers}(R_j,\sigma',\langle S,f \rangle) \cdot p\]
   
   And for all $k \neq i$, source($R_k,\sigma,\langle S,f \rangle$) = source($R_k,\sigma',\langle S,f \rangle$), and movers($R_k,\sigma,\langle S,f \rangle$) = movers($R_k,\sigma',\langle S,f \rangle$).
While there is \( p \in S' \) such that, for some \( R_j \) and \( R_i \), \( p \)'s operation is \( \text{move}(R_i, R_j) \) and \( \text{movers}(R_j, \sigma, (S, f)) = \lambda \).

- Let \( A \) be the set of all processes in \( S' \) whose destination register is \( R_i \). Clearly, \( p \in A \).
- Let \( \sigma' \) be any sequence such that every element of \( \sigma' \) is in \( A \), every process in \( A \) appears exactly once in \( \sigma' \), and \( p \) is the last element of \( \sigma' \).
- Let \( \sigma = \sigma \cdot \sigma' \) and \( S' = S' - A \).

Figure 1: First stage in the construction of a secretive complete schedule

\[ \sigma \text{ is a complete schedule with respect to } (S, f) \text{ if each process in } S \text{ appears exactly once in } \sigma. \]

\( \sigma \) is a secretive complete schedule with respect to \((S, f)\) if \( \sigma \) is complete with respect to \((S, f)\) and for all \( R_i \), \( \text{movers}(R_i, \sigma, (S, f)) \) has at most two processes.

**Lemma 4.1** For all \((S, f)\), there is a secretive complete schedule with respect to \((S, f)\).

**Proof** We construct the desired schedule in two stages. In the following, \( \sigma \) is the schedule constructed so far and \( S' \) is the set of processes yet to be included in \( \sigma \). Initially, \( \sigma = \lambda \) and \( S' = S \). The first stage is described in Figure 1. Informally, we identify a yet-to-be-scheduled process whose source register is still fresh. If \( p \)'s destination register is \( R_i \), we schedule all processes whose destination register is \( R_i \), ensuring that among these \( p \) is scheduled last. As a result, just after \( p \) is scheduled, we have: (i) \( \text{movers}(R_i, \sigma, (S, f)) = p \), and (ii) in any extension \( \sigma' \) of \( \sigma \), there will be no further moves into \( R_i \) and, therefore, \( \text{movers}(R_i, \sigma', (S, f)) = p \). This, together with the loop condition in Figure 1, implies the following claim.

**Claim 4.1** At the end of the first stage, the following statements are true:

1. For all \( R_i \), \( \text{movers}(R_i, \sigma, (S, f)) \) either has one process or is empty.
2. For all \( R_i \), if \( \text{movers}(R_i, \sigma, (S, f)) \) has one process, then in any extension \( \sigma' \) of \( \sigma \), \( \text{movers}(R_i, \sigma', (S, f)) = \text{movers}(R_i, \sigma, (S, f)) \).
3. The source register of any unscheduled move operation is stable: more precisely, if \( p \in S' \) and \( f(p) = \{R_i, R_j\} \), then \( \text{movers}(R_j, \sigma, (S, f)) \) has one process.

The second stage is as follows. Let \( \sigma' \) be any sequence in which each process in \( S' \) appears exactly once. Let \( \sigma = \sigma \cdot \sigma' \). Clearly, \( \sigma \) is a complete schedule. Further, using the above claim, it is easy to verify that, for all \( R_i \), the length of \( \text{movers}(R_i, \sigma, (S, f)) \) is at most 2. \( \square \)

If \( A \) is a set of processes, \( \sigma|A \) denotes the subsequence of \( \sigma \) that includes exactly the processes in \( A \). For example, if \( \sigma = p4, p1, p3, p2 \) and \( A = \{p2, p1\} \), then \( \sigma|A = p1, p2 \).

The following lemma justifies the adjective \"secretive\" in the definition of a secretive complete schedule. It states that it is possible to schedule only a small set of processes—any set that includes just the movers for \( R_i \)—and still have the same value moved into \( R_i \) as in the secretive complete schedule.

**Lemma 4.2** Let \( \sigma \) be a secretive complete schedule with respect to \((S, f)\) and \( S' \) be any subset of \( S \) that includes every process in \( \text{movers}(R_i, \sigma, (S, f)) \). Then, \( \text{source}(R_i, \sigma, (S, f)) = \text{source}(R_i, \sigma|S', (S, f)) \).

5 **Indistinguishability lemma**

5.1 The intuition

Recall the wakeup problem from Section 1.1. The main result of this section is as follows: For all algorithms to the wakeup problem in which processes communicate by applying LL, SC, validate, swap, and move operations on shared-memory, there is a scheduler that forces a process to perform \( \Omega(\log n) \) shared-memory operations in every terminating run. In the next paragraph, we describe the intuition behind the proof. To not obscure the main ideas, in the next paragraph we consider only deterministic algorithms and a shared-memory that supports LL and SC, but not validate, move, or swap operations. The ideas are similar to the ones introduced by Attiya, Lynch, and Shavit in the context of a time complexity lower bound for approximate agreement from single-writer read/write registers [8].

The scheduler forces processes to proceed in rounds. In each round, every process takes one step, which is either an LL or an SC on a shared register. For any process \( p_i \), let \( UP(p_i, r) \) denote the set of processes for which \( p_i \) could have evidence as being up (i.e., as having taken at least one step) by the end of round \( r \). For any shared register \( R \), let \( UP(R, r) \) denote the set of processes that could have been recorded in \( R \) as being up by the end of round \( r \). Clearly, \( UP(p_i, 0) = \{p_i\} \) and \( UP(R, 0) = \emptyset \). In each round \( r \), for each shared register \( R \), there are two possibilities: either exactly one process, say \( p_i \), performs a successful SC on \( R \) or no process performs a successful SC on \( R \). In the former case, we have \( UP(R, r) = UP(p_i, r - 1) \), and in the latter case we have \( UP(R, r) = UP(R, r - 1) \).

In the first case, if \( p_i \) performs \( LL \) on \( R \) before any process successfully SC's \( R \), we have \( UP(p_i, r) = UP(p_i, r - 1) \cup UP(R, r - 1) \). If \( p_i \) performs \( LL \) on \( R \) after some process successfully SC's \( R \), we have \( UP(p_i, r) = UP(p_i, r - 1) \cup UP(R, r) \). To be conservative, our update rule is that if \( p_i \) performs \( LL \) on \( R \) in round \( r \), then \( UP(p_i, r) = UP(p_i, r - 1) \cup UP(R, r) \).

5.2 Definition of \((\text{All}, \mathcal{A})\)-run

For any algorithm, the adversary scheduler is specified in Figure 2. In each run permitted by the scheduler, round
for $r := 1$ to $\infty$ do

1. Each $p_i \in \{p_0, \ldots, p_{n-1}\}$ performs (zero or more) coin tosses until either it is in a termination state or its next step is a shared-memory step.

   - Partition processes that have not terminated into $G_{1,r}$, $G_{2,r}$, $G_{3,r}$, and $G_{4,r}$ such that the next operation of each process in $G_{1,r}$ is an LL or a validate, the next operation of each process in $G_{2,r}$ is a move, the next operation of each process in $G_{3,r}$ is a swap, and the next operation of each process in $G_{4,r}$ is an SC.

2. Processes in $G_{1,r}$ perform one operation each, in the order of their IDs.

3. Processes in $G_{2,r}$ perform one operation each in the order in which they appear in $\sigma_r$.

4. Processes in $G_{3,r}$ perform one operation each, in the order of their IDs.

5. Processes in $G_{4,r}$ perform one operation each, in the order of their IDs.

$r$ refers to the segment of the run that contains the steps taken in the $r$th iteration of the for-loop. Each round has five phases. In the first phase, each of $p_0, \ldots, p_{n-1}$ performs coin tosses until either it has terminated or it is about to perform an operation on shared-memory. Processes that have not terminated are then partitioned into four groups, according to whether their next operation on shared-memory is LL (or validate), move, swap, or SC. The groups are scheduled one after another, in the remaining four phases. Processes within the LL-group, swap-group, and SC-group are scheduled in the order of their IDs, while processes in the move-group are scheduled according to a secretive complete schedule, defined in Section 4.

A toss assignment is any function $A : \{p_0, \ldots, p_{n-1}\} \times N' \rightarrow \text{COIN-RANGE}$, where $N'$ is the set of natural numbers and coin-range is the (arbitrary) set of outcomes of a coin toss. For any toss assignment $A$, define $(All, A)$-run as the (unique) unextendable run permitted by the adversary scheduler in which the outcome of $j$th coin toss by $p_i$ is $A(p_i, j)$.

Throughout this section, we refer to quantities, such as the end of round $k$ of $(All, A)$-run. Such a reference is of course meaningful only if the run has $k$ complete rounds. Informally, $(All, A)$-run has $k$ complete rounds if the first $k$ iterations of the for-loop in Figure 2 complete and the loop-index $r$ assumes the value $k + 1$ at some point. More precisely, $(All, A)$-run has infinitely many rounds if it is not the case that some process performs an infinite number of steps in Phase 1 of some round in $(All, A)$-run. According to this definition, if $(All, A)$-run is a terminating run (recall that a terminating run is a run in which all processes terminate), then it has infinitely many rounds; of course, if the last process terminates in round $k$, then all of rounds $k + 1, k + 2, \ldots$ are empty.

5.3 Update rules

For each process $p$, shared register $R$, and $r \geq 0$, we define the sets $UP(p, r, A)$ and $UP(R, r, A)$. Informally, $UP(p, r, A)$ is the set of processes that $p$ might know, by the end of round $r$ of $(All, A)$-run, as being up. And $UP(R, r, A)$ is the set of processes that can be inferred to be up from $R$'s value at the end of round $r$ of $(All, A)$-run. These sets are defined inductively, using the following rules, henceforth called the update rules. The definition assumes that $(All, A)$-run has infinitely many rounds. Since $A$ is the only toss assignment considered in this section, for brevity, we will denote the sets $UP(p, r, A)$ and $UP(R, r, A)$ as $UP(p, r)$ and $UP(R, r)$, respectively.

- $r = 0$: $UP(p, 0) = \{p\}$ and $UP(R, 0) = \emptyset$, for all processes $p$ and shared registers $R$.

- $r \geq 1$, for a shared register $R$:

1. In round $r$ of $(All, A)$-run, if some process $p$ performs a successful SC on $R$, then $UP(p, r, r) = UP(p, r - 1)$.

   This rule is informally justified as follows: since $p$ is the last process to write a value in $R$ in round $r$, $R$'s value at the end of round $r$ reflects $p$'s knowledge at the end of round $r - 1$.

2. In round $r$ of $(All, A)$-run, if one or more processes perform swap on $R$, then $UP(R, r) = UP(p, r - 1)$, where $p$ is the last process to perform swap on $R$ in round $r$ of $(All, A)$-run. (Notice that in this case no process can have a successful SC on $R$ in round $r$.)

   The informal justification for this rule is the same as for the previous rule.

3. In round $r$ of $(All, A)$-run, if no process performs a swap on $R$ and some process performs a move into $R$, then $UP(R, r) = UP\text{-of-source} \cup UP\text{-of-movers}$, where $UP\text{-of-source} = UP\text{(source}(R, \sigma_r, (G_{2,r}, f_r)), r - 1)$ and $UP\text{-of-movers} = \{p \mid p \in UP(q, r - 1), \ q \in \text{movers}(R, \sigma_r, (G_{2,r}, f_r))\}$. $(\sigma_r, G_{2,r}, f_r)$ are defined in $(All, A)$-run.

   For an informal justification of this rule, suppose $source(R, \sigma_r, (G_{2,r}, f_r)) = R'$, and $(q, q') = \text{movers}(R, \sigma_r, (G_{2,r}, f_r))$. This implies that the value of $R$ at the end of round $r$ is the same as the value of $R'$ at the end of round $r - 1$. This movement of $R'$ into $R$ has become possible because of the move operations of $q$ and $q'$ in round $r$. These move operations are, in turn, possible only because $q$ and $q'$ were in certain specific states at the end of round $r - 1$. Thus, from $R'$'s value at the end of round $r$, one can potentially infer not only the state of $R'$ at the end of round $r - 1$,
but also the states of $q$ and $q'$ at the end of round $r - 1$. Hence the rule.

4. In round $r$ of (All, $A$)-run, if no process performs a successful $SC$ on $R$ or a swap on $R$ or a move into $R$, then $UP(p, r) = UP(R, r - 1)$.

- $r \geq 1$, for a process $p$:
  1. In round $r$ of (All, $A$)-run, if $p$ performs LL or validate on $R$, then $UP(p, r) = UP(p, r - 1) \cup UP(R, r - 1)$. This is justified because validate, by our definition, returns the current value of the register.
  2. In round $r$ of (All, $A$)-run, if $p$ performs a move operation, then $UP(p, r) = UP(p, r - 1)$. This is justified because, since the move operation returns only an acknowledgement, $p$ does not gain any new information.
  3. In round $r$ of (All, $A$)-run, if $p$ is the first process to perform swap on $R$ and no process performs a move into $R$, then $UP(p, r) = UP(p, r - 1) \cup UP(R, r - 1)$. This is justified because $p$’s swap returns the value of $R$ at the end of round $r - 1$.
  4. In round $r$ of (All, $A$)-run, if $p$ is the first process to perform swap on $R$ and at least one process performs a move into $R$, then $UP(p, r) = UP(p, r - 1) \cup UP$-of-source $\cup$ UPs-of-movers, where UP-of-source $= UP(source(R, \sigma_r, (G_{2,r}, f_r)), r - 1)$ and UPs-of-movers $= \{ p \mid p \in UP(q, r - 1), q \in movers(R, \sigma_r, (G_{2,r}, f_r)) \}$. For a justification, observe that $p$’s swap returns $v$ if $v$ is the value of $R$ immediately after all move operations of round $r$ are scheduled. Suppose $v$ was moved into $R$ from $R'$ by some processes (say, $q$ and $q'$). As argued before, from the fact that $R'$’s state is $v$, $p$ can potentially infer not only the value of $R'\approx$ at the end of round $r - 1$, but also the states of $q$ and $q'$ at the end of round $r - 1$. Hence the rule.
  5. In round $r$ of (All, $A$)-run, if $p$ performs swap on $R$ immediately after $q$ performs swap on $R$, then $UP(p, r) = UP(p, r - 1) \cup UP(q, r - 1)$. This is justified because $p$’s swap returns what $q$’s swap writes in $R$.
  6. In round $r$ of (All, $A$)-run, if $p$ performs a successful $SC$ on $R$, then $UP(p, r) = UP(p, r - 1) \cup UP(R, r - 1)$. This is justified because $SC$, by our definition, returns the value of $R$ (before $p$’s successful $SC$, which must be $R$’s value at the end of round $r - 1$).
  7. In round $r$ of (All, $A$)-run, if $p$ performs an unsuccessful $SC$ on $R$, then $UP(p, r) = UP(p, r - 1) \cup UP(R, r)$. This is justified because $SC$, by our definition, returns the value of $R$ and this could be $R$’s value at the end of round $r$.
  8. In round $r$ of (All, $A$)-run, if $p$ does not perform any operation on shared-memory, then $UP(p, r) = UP(p, r - 1)$.

**Lemma 5.1** Assume that (All, $A$)-run has infinitely many rounds. For every process or shared register $X$ and $r \geq 0$, $|UP(X, r)| \leq 4^r$.

**Proof** Since $\sigma_r$ is a secretive complete schedule (with respect to $(G_{2,r}, f_r)$), it follows that $movers(R, \sigma_r, (G_{2,r}, f_r))$ has at most two processes. Using this fact, the lemma is proved by an induction on $r$. □

### 5.4 Definition of $(S, A)$-run

If $UP(p, r) = S$ (more generally, if $UP(p, r) \subseteq S$), intuitively, $p$ has no evidence, by the end of round $r$ of (All, $A$)-run, that processes outside of $S$ are up. This suggests that there might be a run in which only processes in $S$ take steps, and $p$ cannot distinguish this run from (All, $A$)-run in the first $r$ rounds. Figure 3 presents the construction of such a run, that we call (S, $A$)-run. $(S, A)$-run is also structured in rounds. An important aspect of this construction is that not every process in $S$ takes a step in each round. Specifically, in round $r$ of $(S, A)$-run, the only processes to take steps are the ones in $S_i$—those that have not witnessed a process outside of $S$ in the first $r - 1$ rounds of (All, $A$)-run. Each round has five phases, which are similar to the five phases of a round in (All, $A$)-run. Notice that in Phase 1 the $j$th coin toss by $p_i$ (counting from the beginning of the run) is supplied with the outcome $A(p_i, j)$, which is also the outcome of $p_i$’s $j$th process in (All, $A$)-run. Notice also that processes in the move-group $(S_{2,r})$ are scheduled in the order in which they appear in the secretive complete schedule $\sigma_r$, defined for (All, $A$)-run (Claim A.3 in the appendix proves that every process in $S_{2,r}$ is in $\sigma_r$; therefore, this phase is well defined).

### 5.5 Definition of indistinguishability and the main lemma

Let $\Sigma$ be (All, $A$)-run or (S, $A$)-run that has infinitely many rounds. For each process $p$ and shared register $R$, state$(p, r, \Sigma)$, val$(R, r, \Sigma)$, and Pset$(R, r, \Sigma)$ denote the state of $p$, the value and the Pset of $R$, respectively, at the end of round $r$ of $\Sigma$. Val$\approx$(p, r, $\Sigma$) denotes the number of coin tosses performed by $p$ by the end of round $r$ of $\Sigma$. The indistinguishability relation, for processes and shared registers, is defined as follows.

(All, $A$)-run and (S, $A$)-run are indistinguishable to process $p$ up to the end of round $r$, denoted $(All, A)$-run $\approx_p^p (S, A)$-run, if (1) state$(p, r, (All, A)$-run) = state$(p, r, (S, A)$-run), and (2) numtosses$(p, r, (All, A)$-run) = numtosses$(p, r, (S, A)$-run).

(All, $A$)-run and (S, $A$)-run are indistinguishable to shared register $R$ up to the end of round $r$, denoted $(All, A)$-run $\approx_R^p (S, A)$-run, if the following two conditions hold:

(1) val$(R, r, (All, A)$-run) = val$(R, r, (S, A)$-run), and
(2) For all $p$ such that $UP(p, r) \subseteq S, p \in Pset(R, r, (All, A)$-run) if and only if $p \in Pset(R, r, (S, A)$-run).

Next we state the main lemma, which is proved by induction on $r$. The proof is very long and its outline is presented in the appendix.

**Lemma 5.2** (Indistinguishability Lemma) Assume that (All, $A$)-run has infinitely many rounds. For every $S \subseteq \{p_0, p_1, \ldots, p_{n-1}\}$, process or shared register $X$, and round $r \geq 0$, if $UP(X, r) \subseteq S$, then $(All, A)$-run $\approx_S^p (S, A)$-run.

### 6 Application of the indistinguishability lemma to lower bounds

In this section, we apply the indistinguishability lemma to obtain a lower bound of $\Omega(\log n)$ on the time complexity of any solution to the $n$-process wakeup problem (defined in Section 1.1), if the solution is based on a shared-memory
for $r := 1$ to $\infty$ do
- Let $S_r = \{ p \mid UP(p, r) \subseteq S \}$.
- Each $p_i \in S_r$ performs (zero or more) coin tosses until either it is in a termination state or its next step is a shared-memory step.
- The $j$th coin toss by $p_i$ is supplied with the outcome $A(p_i, j)$.
- Partition processes in $S_r$ that have not terminated into $S_1, S_2, S_3, S_4$ such that the next operation of each process in $S_1$ is an LL or a validate, the next operation of each process in $S_2$ is a move, the next operation of each process in $S_3$ is a swap, and the next operation of each process in $S_4$ is an SC.
- Processes in $S_1$ perform one operation each, in the order of their IDs.
- Processes in $S_2$ perform one operation each, in the order in which they appear in $\sigma_r$.
- Processes in $S_3$ perform one operation each, in the order of their IDs.
- Processes in $S_4$ perform one operation each, in the order of their IDs.

Figure 3: Definition of $(S, A)$-run

that supports only LL, SC, validate, move, and swap operations. Therefore, if processes can solve wakeup by applying a constant number of operations on linearizable objects of type $T$, it follows that the time complexity of any $n$-process linearizable implementation of a type $T$ object (from shared-memory supporting LL, SC, validate, move, and swap operations) is $\Omega(\log n)$ (see [20] for a definition of linearizability, also known as atomicity). This observation yields a $\Omega(\log n)$ time complexity lower bound for implementations of several types of objects.

**Theorem 6.1** Consider any algorithm for the $n$-process wakeup problem in which LL, SC, validate, move, and swap are all the operations that processes may apply on shared-memory. If the algorithm terminates with probability $c$, then its worst-case expected shared-access time complexity is at least $\frac{1}{c} \log n$.

**Proof** Consider any algorithm for the wakeup problem. Let $A$ be a toss assignment such that $(All, A)$-run is a terminating run of the algorithm (it follows that $(All, A)$-run has infinitely many rounds). The specification of wakeup requires that some process returns 1. We claim that the process that returns 1 performs at least $\log n$ shared-memory operations in $(All, A)$-run. This claim, together with Lemma 3.1, implies the theorem. The proof of the claim is as follows.

Let $p_i$ be the process that returns 1 in $(All, A)$-run and let $r$ be the number of shared-memory operations performed by $p_i$ in $(All, A)$-run. It follows that $p_i$ returns either after its shared-memory operation in round $r$ or in Phase 1 of round $r + 1$. Let $S = UP(p_i, r)$. By Lemma 5.1, $|S| \leq 4^r$. If the claim is false, $r < \log_4 n$. It follows that $|S| < n$. By Lemma 5.2, $(All, A)$-run $\approx p_i (S, A)$-run.

Suppose that $p_i$ returns 1 after its shared-memory operation in round $r$ of $(All, A)$-run. Since $(All, A)$-run $\approx p_i (S, A)$-run, it follows that $p_i$ returns 1 after its shared-memory operation in round $r$ of $(S, A)$-run.

Suppose that $p_i$ returns 1 in Phase 1 of round $r + 1$ of $(All, A)$-run. Since $(All, A)$-run $\approx p_i (S, A)$-run, it follows that $p_i$’s state and the number of coin tosses performed by $p_i$ by the end of round $r$ is the same in both $(All, A)$-run and $(S, A)$-run. It follows from these facts that $p_i$ returns 1 in Phase 1 of round $r + 1$ of $(S, A)$-run.

Since fewer than $n$ processes take steps in $(S, A)$-run, the return of 1 by $p_i$ violates the specification of the wakeup problem. This completes the proof of the claim. Hence the theorem.

In the following, we refer to an implementation of a shared object as a $k$-use implementation if the implementation guarantees correctness only in the case when each process applies at most $k$ operations on the implemented object. A $k$-use implementation terminates with probability at least $c$ if the following is true: If each process terminates after applying at most $k$ operations on the implementation and each non-terminating process repeatedly takes steps, then the probability that all processes eventually terminate is at least $c$.

The next corollary states that if wakeup can be solved in constant time using a type $T$ object, then in any implementation of a type $T$ object, it takes $\Omega(\log n)$ steps to complete some operation on the implementation.

**Corollary 6.1** Suppose the following statements are true:

1. $T$ is any object type with the following property: there is a deterministic algorithm to solve the $n$-process wakeup problem in which processes communicate via a single linearizable object $O$ of type $T$ and each process terminates after applying at most $k$ operations on $O$ (for some constant $k$).

2. $T$ is any randomized linearizable $n$-process $k$-use implementation of an object of type $T$, and satisfies the following two properties:
   
   (a) The implementation is based on shared-memory that supports only LL, SC, validate, move, and swap operations.
   
   (b) The implementation terminates with probability at least $c$.

Then, the worst-case expected number of shared-memory operations required to complete some operation on the implementation $I$ is at least $\frac{1}{c} \log n$.

**Proof** Follows trivially from Theorem 6.1.
• \text{fetch\&and}(v)$, for a $k$-bit argument $v$, changes the state to the bitwise ANDing of $s$ and $v$, and returns $s$.

• \text{fetch\&or}(v)$, for a $k$-bit argument $v$, changes the state to the bitwise ORing of $s$ and $v$, and returns $s$.

• \text{fetch\&multiply}(v)$ changes the state to $(s\cdot v) \mod 2^k$, and returns $s$.

• \text{fetch\&complement}(i)$, for some $1 \leq i \leq k$, complements the $i$th bit of the state and returns the previous value of the state.

Theorem 6.2 Consider any randomized linearizable $n$-process implementation of any of the following objects from shared-memory that supports only LL, SC, validate, move, and swap operations:

1. $k$-bit object supporting the \text{fetch\&increment} operation, for any $k \geq \log n$

2. $k$-bit object supporting any one of \text{fetch\&and}, \text{fetch\&or}, \text{fetch\&complement}, \text{fetch\&multiply} operations, for any $k \geq n$

3. queue or stack that may initially contain $n$ or more items

4. $k$-bit object supporting read and increment operations, for any $k \geq \log n$ (the increment operations adds 1 to the state and returns just an acknowledgement)

The worst-case expected number of shared-memory operations required to complete an operation on the implementation is at least $\frac{1}{2} \log n$ in the first three cases, and $\frac{3}{4} \log n$ in the fourth case.

Proof For any object $O$ mentioned in the first three items of the theorem, we present a simple wakeup algorithm in which each process terminates after performing one operation on $O$. If $O$ is an object that supports read and increment operations, we present a wakeup algorithm in which each process terminates after performing two operations on $O$. These, together with Corollary 6.1, imply the theorem. The different $n$-process wakeup algorithms are as follows:

• Let $O$ be a $k$-bit object supporting \text{fetch\&increment} operation, for some $k \geq \log n$. Initialize $O$ to $0$. Process $p_i$’s algorithm is as follows: (i) apply \text{fetch\&increment} on $O$, (ii) if $O$’s response is $n - 1$, return 1 and terminate; otherwise return 0 and terminate.

• Let $O$ be a $k$-bit object supporting \text{fetch\&and} operation, for some $k \geq n$. Initialize $O$ so that each of its bits has 1. Process $p_i$’s algorithm is as follows: (i) apply \text{fetch\&and}$(v_i)$ on $O$, where $v_i$ has 0 in the $i$th bit and 1 in every other bit position, (ii) if $O$’s response has 0’s in the first $n$ bits except in the $i$th bit, return 1 and terminate; otherwise return 0 and terminate. The algorithm is similar if $O$ supports \text{fetch\&or} or \text{fetch\&complement}.

• Let $O$ be a $k$-bit object supporting \text{fetch\&multiply} operation, for some $k \geq n$. Initialize $O$ to 1. Process $p_i$’s algorithm is as follows: (i) apply \text{fetch\&multiply}(2) on $O$, (ii) if $O$’s response is 0, return 1 and terminate; otherwise return 0 and terminate.

• Let $O$ be a queue that initially contains $n$ items—1, 2, \ldots, $n$—with $n$ at the rear. Process $p_i$’s wakeup algorithm is as follows: (i) apply dequeue on $O$, (ii) if $O$’s response is $n$, return 1 and terminate; otherwise return 0 and terminate.

• Let $O$ be a $k$-bit object supporting read and increment operations, for some $k \geq n$. Initialize $O$ to 0. Process $p_i$’s algorithm is as follows: (i) increment $O$, (ii) read $O$, (iii) if the value read is $n$, return 1 and terminate; otherwise return 0 and terminate.

This completes the proof of the theorem. □

7 Open problems

Consider the \text{RMW}(R,f)$ operation which takes any computable function $f$ as an argument, changes the state of shared register $R$ from its current value of $v$ to $f(v)$, and returns $v$. If shared-memory supports such an operation and has registers of unbounded size, it is easy to see that every object has a wait-free implementation of unit worst-case shared-access time complexity. But suppose we restrict ourselves to a shared-memory that supports only “reasonable” operations (for some acceptable definition of “reasonable”). Is there a technique to prove a $\log n$ (or any non-constant) lower bound on the worst-case shared-access time complexity of oblivious universal constructions that are based on such a shared-memory? For instance, if shared-memory supports all of read, write, LL, SC, swap, compare\&swap, move, fetch\&add, fetch\&multiply, would the $\Omega(\log n)$ lower bound still hold? For the local time complexity of oblivious universal constructions, we are able to prove a $\Omega(n)$ lower bound that holds irrespective of the operations supported by the underlying shared-memory [23]. It is open if such a general result can be proved also for the shared-access time complexity.

If we rule out constructions that make impractical assumptions on the size of registers, $O(n)$ is the best known upper bound on the worst-case shared-access time complexity of universal constructions [19, 1]. The large gap between this upper bound and our lower bound of $\log n$ can be reduced by either designing faster universal constructions or by strengthening the lower bound. We now discuss the issues that concern the latter. The lower bound, as explained in the introduction, cannot be strengthened unless either the number or the size of shared registers is restricted. Restricting the number seems unnatural. As for the size, it does not appear easy to come up with a restriction that does not inadvertently rule out practical implementations. For instance, the assumption that register size is a constant (independent of $n$) is not satisfactory because the size will then be too small to store even a processID (which has $\log n$ bits). Similarly, any restriction on the size will rule out (practically acceptable) constructions that employ unbounded sequence numbers. One possibility to overcome these difficulties is to assume that each register has $O(\log n)$ bits and attempt to obtain a stronger lower bound (than the $\Omega(\log n)$ proved in this paper) for implementations of single-use objects.

Another research direction is to explore the meaning and the possibility of non-oblivious universal constructions.

Acknowledgement

I am grateful to PODC referees for many comments that helped improve the presentation.
References


[23] Jayanti, P. A lower bound on the local time complexity of universal constructions. In these proceedings.


A Appendix: An outline of the proof of the Indistinguishability Lemma (Lemma 5.2)

**Lemma 5.2 (Indistinguishability Lemma)** Assume that $(All, A)$-run has infinitely many rounds. For every $S \subseteq \{p_0, p_1, \ldots, p_{n-1}\}$, process or shared register $X$, and round $r \geq 0$, if $UP(X, r) \subseteq S$, then $(All, A)$-run $\approx_X (S, A)$-run.

**Proof** The proof is by induction on $r$. The base case, for $r = 0$, is easy. Assume that $r \geq 1$ and the lemma holds for $r - 1$. We establish a series of claims to prove the induction step. In the statements of all of the following claims, $p$ and $q$ are arbitrary elements of $\{p_0, \ldots, p_{n-1}\}$ and $R$ is an arbitrary shared register. (The proofs of the claims are long and therefore omitted. A technical report from Dartmouth College Computer Science department will contain the full version.)

**Claim A.1** If $UP(p, r - 1) \subseteq S$, then:
1. $\text{numtosses}(p, r, (All, A)$-run) $\subseteq \text{numtosses}(p, r, (S, A)$-run).
2. $p$'s state at the end of Phase 1 in round $r$ of $(All, A)$-run is the same as $p$'s state at the end of Phase 1 in round $r$ of $(S, A)$-run.

**Claim A.2**
1. If $UP(p, r - 1) \not\subseteq S$, then $p$ does not perform a shared-memory step in round $r$ of $(S, A)$-run.
2. If $UP(p, r - 1) \subseteq S$ and $p$ does not perform a shared-memory step in round $r$ of $(All, A)$-run, then $p$ does not perform a shared-memory step in round $r$ of $(S, A)$-run.
3. If $UP(p, r - 1) \subseteq S$ and $p$ performs an operation on shared-memory in round $r$ of $(All, A)$-run, then $p$ performs the same operation on shared-memory in round $r$ of $(S, A)$-run.

The following claim shows that the scheduling of processes in $S_{2,r}$, in round $r$ of $(S, A)$-run, is well-defined.

**Claim A.3** $S_{2,r} \subseteq G_{2,r}$ (recall that $S_{2,r}$ and $G_{2,r}$ are the sets of processes that perform a move operation in round $r$ of $(S, A)$-run and $(All, A)$-run, respectively).

**Claim A.4** If some process performs a successful SC on $R$ in round $r$ of $(All, A)$-run, then $UP(R, r - 1) \subseteq UP(R, r)$.

**Claim A.5** If $UP(p, r) \subseteq S$ and $p$ performs SC on $R$ in round $r$ of $(All, A)$-run, then $UP(R, r) \subseteq S$.

**Claim A.6** If $UP(R, r) \subseteq S$ and some process $q$ performs SC($R, v$) successfully in round $r$ of $(All, A)$-run, then $q$ performs SC($R, v$) successfully in round $r$ of $(S, A)$-run.

**Claim A.7** Suppose that $UP(R, r) \subseteq S$ and some process performs a swap on $R$ in round $r$ of $(All, A)$-run. Then, (1) the value of $R$ immediately after all swap operations are performed in round $r$ of $(All, A)$-run is the same as the value of $R$ immediately after all swap operations are performed in round $r$ of $(S, A)$-run, and (2) $Pset$ of $R$ is empty immediately after all swap operations are performed in round $r$, both in $(All, A)$-run and in $(S, A)$-run.

**Claim A.8** Suppose that one of the following two statements is true:
1. $UP(R, r) \subseteq S$ and, in round $r$ of $(All, A)$-run, no process performs a swap on $R$, but some process performs a move into $R$.
2. $UP(p, r) \subseteq S$ and, in round $r$ of $(All, A)$-run, $p$ is the first process to perform a swap on $R$ and some process performs a move into $R$.

Then, (i) the value of $R$ immediately after all move operations are performed in round $r$ of $(All, A)$-run is the same as the value of $R$ immediately after all move operations are performed in round $r$ of $(S, A)$-run, and (ii) $Pset$ of $R$ is empty immediately after all move operations are performed in round $r$, both in $(All, A)$-run and in $(S, A)$-run.

**Claim A.9** If $UP(R, r) \subseteq S$ and no process performs a successful SC on $R$ in round $r$ of $(All, A)$-run, then no process performs a successful SC on $R$ in round $r$ of $(S, A)$-run.

**Claim A.10** Suppose that $UP(R, r) \subseteq S$ and, in round $r$ of $(All, A)$-run, no process performs a successful SC on $R$, or a swap on $R$, or a move into $R$. Then:
1. If some process $p$ performs SC on $R$ in round $r$ of $(S, A)$-run, $p$ receives the same response from $R$ in round $r$ of $(S, A)$-run as in round $r$ of $(All, A)$-run.
2. $(All, A)$-run $\approx_R (S, A)$-run.

The following claim proves the induction step for an arbitrary process $p$.

**Claim A.11** If $UP(p, r) \subseteq S$, then:
1. $\text{state}(p, r, (All, A)$-run) $\subseteq \text{state}(p, r, (S, A)$-run)
2. $\text{numtosses}(p, r, (All, A)$-run) $\subseteq \text{numtosses}(p, r, (S, A)$-run)

The following claim proves the induction step for an arbitrary shared register $R$.

**Claim A.12** If $UP(R, r) \subseteq S$, then:
1. $\text{val}(R, r, (All, A)$-run) $\subseteq \text{val}(R, r, (S, A)$-run)
2. For all $p$ such that $UP(p, r) \subseteq S$, $p \in \text{Pset}(R, r)$ and $(All, A)$-run if and only if $p \in \text{Pset}(R, r)$ and $(S, A)$-run.