Test-and-Set in Optimal Space

George Giakkoupis
INRIA Rennes
george.giakkoupis@inria.fr

Maryam Helmi
University of Calgary
mhelmikh@ucalgary.ca

Lisa Higham
University of Calgary
higham@ucalgary.ca

Philipp Woelfel
University of Calgary
woelfel@ucalgary.ca

November 4, 2014

Abstract

The test-and-set object is a fundamental synchronization primitive for shared memory systems. It is essentially equivalent to (weak) leader election, where losing processes don't learn the name of the leader. In 1989 Styer and Peterson [25] showed that the number of registers required for a deadlock-free implementation of a test-and-set object is $\lceil \log n \rceil + 1$, in the standard asynchronous shared memory model with n processes, supporting atomic reads and writes. It remained open whether the lower bound is asymptotically tight, even for stronger progress conditions, and in particular for obstruction-freedom. In this paper we answer this long-standing open question in the affirmative. We present a deterministic obstruction-free implementation of a test-and-set object from $O(\log n)$ registers of size $O(\log n)$ bits. This improves on the previously best bound of $O(\sqrt{n})$ registers [14]. Combining our obstruction-free algorithm with techniques from previous research [14, 15], we also obtain a randomized wait-free test-and-set algorithm from $O(\log n)$ registers, with expected step-complexity $O(\log^* n)$ against the oblivious adversary.

1 Introduction

A *test-and-set* (*TAS*) object is perhaps the simplest standard shared memory primitive that has no wait-free deterministic implementation from registers. It stores one bit, which is initially 0, and supports a `test&set()` operation that sets the bit's value to 1 and returns its previous value. A TAS object trivially provides a solution to another famous problem, namely that of (weak) *leader election*: Each participating process has to decide on one value, `win` or `lose`, such that exactly one process (the leader) wins. (The term leader election is ambiguous, as it is sometimes used to denote the *name consensus* problem, where the losing processes need to output the ID of the winner.) In fact, weak leader election and test-and-set are equally hard problems, because a very simple algorithm using a leader election protocol and one additional binary register can be used to implement a linearizable TAS object, such that a `test&set()` operation requires only a constant number of read and write operations in addition to the leader election protocol [17].

TAS objects have consensus number two, that is, they can be used together with registers to solve deterministic wait-free consensus only in systems with two processes. Despite that, TAS is a standard building block for shared memory algorithms, that has been used to solve many classical problems, such as mutual exclusion and renaming [5–7, 9, 11, 23, 24]. Since TAS objects are among the simplest synchronization primitives, they are well suited to research the difficulties arising in synchronization problems. Algorithms or impossibility results for TAS can yield deep insights into the complexity of other shared memory problems, and can help devising solutions for them.

We consider a standard shared memory system in which n processes communicate through atomic read and write operations on shared registers. A common assumption is that each registers can store $\Theta(\log n)$ bits, although in some settings registers of unbounded size are assumed. The strongest reasonable progress condition is wait-freedom, which guarantees that every operation finishes in a finite number of the calling process' steps, independent of other processes. Since TAS has consensus number two, no deterministic wait-free TAS implementation from registers exists, even for $n = 2$ processes. A weaker progress condition, and the one most frequently used for analyzing space complexity, is *obstruction-freedom* [21]. It guarantees that any process will finish its operation, provided the process takes sufficiently many steps on its own, without interference from other processes (i.e., if it runs solo). Any shared memory object has an obstruction-free implementation from n registers.

The randomized step complexity of TAS has been thoroughly investigated, with significant progress being made in recent years [2, 4, 7, 16, 26]. In contrast, little was known about the space complexity of obstruction-free or randomized wait-free TAS. In 1989 Styer and Peterson [25] showed that any deadlock-free TAS implementation requires at least $\lceil \log n \rceil + 1$ registers, and complemented this result by an algorithm matching the bound. Styer and Peterson's work at the time was motivated by interest in the space complexity of mutual exclusion. A *long-lived* version of a TAS object, which supports a `reset()` operation in addition to `test&set()`, immediately yields a general mutual exclusion algorithm, but cannot be implemented from fewer than n registers [10]. A *one-shot* TAS, as we consider it in this work, allows exactly one process to enter the critical section.

Deadlock-freedom is a natural progress property for mutual exclusion related problems, where waiting for other processes is inherent in the problem specification. But it is too weak for other problems, as it does not preclude that a single slow or failing process prevents all other processes from making progress. Stronger progress properties, such as obstruction-freedom, lock-freedom, or randomized wait-freedom, are more desirable for problems that do not inherently require processes to wait for each other. Research on the space complexity of shared memory problems has focused on the obstruction-free progress property [13, 19, 21, 22].

Clearly, Styer and Peterson's lower bound implies that no TAS implementation from $\lceil \log n \rceil$

or fewer registers exists for any progress property that is stronger than deadlock-freedom, such as the above mentioned ones. However, despite significant research on TAS, until recently no TAS implementation with sub-linear space complexity and stronger than deadlock-free progress guarantees was known. In 2013 we devised a deterministic obstruction-free TAS algorithm using just $\Theta(\sqrt{n})$ registers, where the longest solo-run for any process to finish a `test&set()` method call comprises $\Theta(\sqrt{n})$ steps [14]. We now present a significantly improved result.

Theorem 1. *There is a deterministic obstruction-free implementation of a TAS object from $\Theta(\log n)$ registers of size $O(\log n)$ bits, where any process finishes its* `test&set()` *method call in $O(\log n)$ solo steps.*

This space upper bound is asymptotically optimal, and proves that Styer and Peterson's space lower bound of $\Omega(\log n)$ registers is tight even for the obstruction-free progress property. Note that, in general, short longest solo runs have performance benefits, as processes have a better chance of completing their TAS before they get interrupted. Our algorithm satisfies an even stronger property: A process needs only execute a constant number of solo steps $O(\log n)$ times, to finish its `test&set()` method call.

Giakkoupis and Woelfel [16] presented a randomized TAS algorithm that has $O(\log^* k)$ expected step complexity against the oblivious adversary, where k is the contention. The algorithm requires $\Theta(n)$ registers, but with high probability (*w.h.p.*) processes access only the first logarithmic number of them. In [14] we also showed that any obstruction-free algorithm can be made randomized wait-free against the oblivious adversary, with a polynomial expected step complexity. A combination of the algorithms in [16] and the one in [14] led to a randomized-wait free TAS implementation from $\Theta(\sqrt{n})$ registers, which has expected step complexity $O(\log^* n)$. This approach with some additional work can be applied to our $O(\log n)$ space algorithm.

Theorem 2. *There is a randomized TAS implementation from $\Theta(\log n)$ registers of size $O(\log n)$ bits, such that for any fixed schedule (determined by an oblivious adversary), the maximum number of steps executed by any process is $O(\log^* n)$ in expectation and $O(\log n)$ with high probability.*

Recently, Aghazdeh and Woelfel [3] gave a general transformation of any (one-shot) TAS object implemented from m b-bit registers into a long-lived one using $O(m \cdot n)$ registers of size $\max\{b, \log(n+m)\} + O(1)$ bits. A `reset()` operation takes only constant time in the worst-case, and the step complexity of a `test&set()` operation of the long-lived object is the same (up to a constant additive term) as the one of the one-shot TAS. Applying this to the construction described in Theorem 2, we obtain a long-lived TAS object from $O(n \log n)$ registers, where the expected step complexity of the `test&set()` operation is $O(\log^* n)$ and worst-case step complexity of the `reset()` operation is $O(1)$. Previously, the best space bound for achieving similar step complexities for `test&set()` and `reset()` was $O(n^{3/2})$ registers, achieved by, again, combining previous results [3, 14]. Note that the space lower bound for mutual exclusion [10] implies that any long-lived TAS implementation requires at least n registers.

2 Model

Our model of computation is the standard asynchronous shared memory model where a set of n processes $\mathcal{P}$ with distinct identifiers communicate through a set of shared multi-reader multi-writer registers. Each register supports two atomic operations, read and write. Each process is an assignment of a program that can access its local registers as well as shared registers. An *execution* is a sequence of operations taken by processes according to their programs.

The algorithm is *deterministic* if each process's program is deterministic. A deterministic implementation of a method is *wait-free* if from any point of an execution and for any process, the process completes its method call in a finite number of its own steps, regardless of the steps taken by other processes. A deterministic implementation of a method is *obstruction-free* if from any point of an execution, and for any process p, in a solo execution by p starting from this point, p completes its method call in a finite number of steps.

The algorithm is *randomized* if some process's program is randomized. An implementation of a method is *randomized wait-free* if from any point of an execution and for any process p, the number of steps by p required for p to complete its method call regardless of the steps taken by other processes, is finite in expectation [20].

In this paper we assume w.l.o.g. that the system supports atomic scans [1, 8, 12]. Operation `scan(`A`)` for some set A of registers, returns a snapshot of the values of all registers in A. An obstruction-free implementation of a `scan()` operation can be obtained by using one reserved register in addition to those that can be scanned [18]. In this implementation, the longest solo-run is bounded by the total number of scannable registers.

3 Efficient Sifters

An $f(k)$-*sifter*, where f is a function satisfying $1 \leq f(k) \leq \max\{k-1, 1\}$, supports only one method call, `Compete`. Each `Compete` method call returns either `win` or `lose`. In any execution, where k processes call `Compete`, at most $f(k)$ of them return `win`, and at most $k-1$ return `lose`. A 1-sifter is a (weak) leader election protocol, and thus yields a TAS object using one additional register.

Recent randomized TAS constructions [4,16] are based on randomized sifters, where the number of winning processes is at most $f(k)$ in *expectation*. Here, however, we use deterministic sifters, where $f(k)$ is a worst-case bound. Two such $f(k)$-sifters can be combined to obtain an $f(f(k))$ sifter, by letting the losers of the first sifter lose, and the winners of the first sifter call `Compete` on the second sifter.

In this section we present an obstruction-free $f(k)$ sifter implementation from 8 registers, where $f(k)$ is approximately $(2/3)k$.

Theorem 3. *There is an obstruction-free implementation of* $(\lfloor \frac{2k+1}{3} \rfloor)$*-Sifter using 8 multi-reader multi-writer* $O(\log n)$*-bit registers.*

Combining $O(\log n)$ such sifter objects yields a 1-sifter, and thus ultimately a TAS object, from $O(\log n)$ registers. Hence, Theorem 1 follows immediately from Theorem 3. The remainder of this section is devoted of the proof of this theorem.

Intuition and Informal Description of the Sifter Procedure

To aid intuition for our sifter procedure, we first consider a very simple obstruction-free sifter object, D. Object D consists of an array of three registers, each of which can hold one process identifier, and supports two operations: 1) scan of all three registers, which returns a triple of process identifiers, called a *signature* and 2) write to a specified register. Each process p alternates between writing and scanning. When p writes, it writes its own identifier to a register of D that did not contain p in its preceding scan. The goal of any process, p, is to achieve a *clean-sweep* meaning that its scan returns signature (p, p, p). In this case, p terminates with win. If, however, while trying for a clean-sweep, p's scan returns a signature that contains more copies of a different identifier than it has copies of p, then p terminates with lose. Any process that runs alone for

Shared:
$A[0 \dots 2]$: array of registers; each register stores a value from $\mathcal{P} \cup \{\perp\}$ and is initially $\perp$
$B[0 \dots 3]$: array of registers; each register stores a pair (id, sig), where $\text{id} \in \mathcal{P} \cup \{\perp\}$, and sig is a triple from the set $(\mathcal{P} \cup \{\perp\})^3$; initially $\text{id} = \perp$ and $\text{sig} = (\perp, \perp, \perp)$.

`Compete()`:

```
1  pos := 0
2  while true do
3      A[pos].write(p)
4      a := scan(A)
5      if ∀ i ∈ {0,1,2}, a[i] = p then return win
6      if ∃ q ∈ P, num(p,a) < num(q,a) then return lose
7      if num(p,a) = 1 then
8          if Knockout(a) then return lose
9      let pos ∈ {0,1,2}: a[(pos − 1) mod 3] = p and a[pos] ≠ p
```

`Knockout(sig)`:

```
10 index := 0
11 while true do
12     B[index].write((p, sig))
13     (a, b) := scan(A, B)
14     if a ≠ sig then return true
15     if num((p,sig), b) ≤ 1 and ∃ q ∈ P, num((q,sig), b) ≥ 2 then return true
16     if ∀ i ∈ {0,...,3}, b[i] = (p, sig) then return false
17     let index ∈ {0,...,3}: b[index] ≠ (p, sig)
```

Figure 1: Implementation of Sifter for process $p \in \mathcal{P}$

five steps without losing, will win, and not all processes can lose, so D is an implementation of an obstruction-free sifter object.

Object D, however, is not a very efficient sifter object. Suppose that while a clean-sweep is being achieved by one process, two other processes cover two different register of D. Then these covering processes can over-write the clean-sweep, and be made to cover again, allowing another clean-sweep to proceed. By repeating this kind of scenario, executions are easily created where all but one process wins. Also, notice that to create another winner after a clean-sweep, such an obliteration of the clean-sweep by two (or three) over-writes is also necessary.

To reduce the number of processes that can win to at most a constant fraction of those that compete, the core idea is to prevent processes that participate in over-writing a clean-sweep, from covering again, without some process losing. This is achieved, in our algorithm, by enhancing D to become component A, and by adding a second similar (but not identical) component, B. Component B is an array of four registers instead of three, each of which can hold a pair consisting of a process identifier and a signature. Each scan now returns the content of both A and B, and a write by p can be either a write of p to a register of A, or a write of (p, s_p) to a register of B, where s_p is a signature. Processes begin by competing on component A and still strictly alternate between writing and scanning.

If process p, competing on component A, gets a scan with signature of A equal to s, and s contains 3 distinct identifiers, one of which is p, then p leaves A to compete on component B while remembering s. If p does not get such a scan and it does not lose immediately, then p is in at least two positions in s, so its last write could not have been part of an over-write of a clean-sweep. By writing the pair (p, s) to registers of B, p tries to achieve a clean-sweep of B (meaning a scan by p shows each of the 4 registers of B contains (p, s)). Process p competes on B only against other processes writing the same signature, s. That is, p loses if its scan shows that for some other process q, (q, s) occupies at least 2 positions of B while not more than one position holds (p, s). (The losing rule on B differs from that of A to protect against errors from writes by processes writing out-of-date signatures.) Suppose, however, that while trying to achieve a clean-sweep of B, one of p's scans shows a signature of A different from s. Then p can safely lose because the competition in A has continued in the meantime with other processes. That is, p only returns to compete in component A if it achieves a clean-sweep of B while its scans continue to show the signature of A is s.

Our proof will establish that not all processes can return lose, and at most $2/3$ of the processes can return win. For the intuition of our proof, assume, for a moment, that it is not possible for A's signature to be some value s, then change, and then subsequently changed back to s. This is enough to show that not all processes lose. Essentially, each process that loses, loses to another in such a way that there are no cycles in the "lose-to" relation. Now consider the intervals in an execution between the final scans of processes that win (achieve a clean-sweep of A). We associate a losing process with each of these intervals as follows. (1) For any process, p, that loses in A because its scan shows a signature of A in which some other process occupies more positions in A than itself, associate p with the interval that contains the last write by p. Similarly, (2) for any process q that loses in B after its scan shows a signature in A that differs from the one it is writing to B, associate q with the interval that contains its last write.

We will prove that each of the remaining intervals contain at least two processes that moved from A to B and finished competing on B using a signature, say s, while the signature of A remained s. Since B has 4 registers, after a first clean-sweep on B, a subsequent clean-sweep on B requires that the previous clean-sweep is over-written. But such an over-write requires participation from processes with signatures different from s, and any such process must necessarily lose on its next scan because its signature is not equal to that of A. So, in this case, the interval already has an

associated loser from (2). Otherwise, (3) we assign to this interval a process with signature s that loses to the process that achieved the clean-sweep. Notice, however, that this losing process could withhold its last write in order to be assigned again in (2). Therefore, using these three rules of assignment, we associate a losing process to every interval, and no process is associated with more than two of these intervals.

The flaw in this intuition is the assumption that a signature of A cannot change and later change back. Indeed, this is possible. Suppose a write w to A leaves A with the signature, s, and that s had also been an earlier signature of A. Our proof will use a closer analysis to show that if any register of B contains signature s after w, then that register of B was actually written with signature s after w. That is, earlier copies of signature s in B will be overwritten with signatures different from s, before the new occurrence of s as a signature of A. We will see that this is enough to complete the proof, even though signatures in A can repeat.

Sifter Procedure

Our sifter implementation is presented in Figure 1. Our procedure uses the following definition and notation: For any array a and any value v, define $\text{num}(v, a) = |\{i : a[i] = v\}|$.

4 Proof of Correctness of the Sifter Implementation

Notation and terminology A *losing scan* is a scan by a process such that this process will return `lose` in its next step, without doing any further shared memory operation. A *winning scan* is a scan by a process such that this process will return `win` in its next step, without doing any further shared memory operation. For each winning scan there exists a last write by the process that performs this scan. We call this write a *winning write*. For an execution E, let $s_1, s_2, \ldots, s_k$ be the sequence of winning scans in E and let $q_1, q_2, \ldots, q_k$ denote the corresponding sequence of processes that performed these scans. Observe that $\forall i$, $1 \leq i \leq k$, s_i is preceded by a winning write w_i performed by q_i. Furthermore, $\forall i$, $1 \leq i \leq k$, s_i must happen before w_{i+1} because at s_i all registers in A contain q_i's id however, at w_{i+1}, q_{i+1} has written its own id everywhere in A. Hence winning scans and winning writes strictly interleave. That is, the order of winning scans and writes in E is $w_1, s_1, w_2, s_2, \ldots, w_k, s_k$.

Given an execution $E = \text{op}_1, \text{op}_2 \ldots$, we denote the contiguous subsequence of E starting at op_i and ending at the operation immediately before op_j by $E[\text{op}_i : \text{op}_j)$. In our analysis we are interested in intervals between any winning scan and the next winning write. A *sifting interval* $E[s^* : w^*)$ is a subsequence of an execution that starts at some winning scan s^* and ends at the operation immediately before w^*, which is the first winning write in E after s^*. Observe that all sifting intervals are disjoint. Given sifting interval $I = E[s^* : w^*)$, let scanner(I) denote the process that executes s^* and writer(I) denote the process that executes w^*. If an execution contains k winning scans, then it has $k - 1$ disjoint sifting intervals.

A *signature* is an ordered triple of identifiers. A signature (p_0, p_1, p_2) is *full* if for any $i, j \in \{0, 1, 2\}, i \neq j$ implies $p_i \neq p_j$. The following lemmas concern properties of executions. Terms such as before, after, next, previous, precedes, and follows are all with respect to the order of operations in an execution. In an execution, process p is *poised to write* to register r, if it writes to register r in its next step. Equivalently, we say that p *covers* register r. A local variable x in the algorithm is denoted by x_p when it is used in the method call invoked by process p.

Lemma 4. *Suppose at scan s, $A = (p_0, p_1, p_2)$ is a full signature. For any $i \in \{0, 1, 2\}$, if p_i writes to A after s, then its first write after s into A is not to $A[i]$.*

Proof. Let w_i be the first write by p_i to A after s. Let s_i be the scan by p_i preceding w_i. If s_i happens before s, then there is no write to A by p_i in the interval $E[s_i : s)$. At s, $A[i] = p_i$, hence at s_i, $A[i] = p_i$. Suppose s_i happens after s. At s, $A[i]$ is the only location that contains p_i, and there is no write to A by p_i in the interval $E[s : s_i)$ and s_i is not a losing scan. Therefore, at s_i, $A[i] = p_i$. In either case, by Line 9, w_i is a write to $A[\text{pos}]$ where $\text{pos} \neq i$. □

Lemma 5. *Suppose at scan s, $A = (p_0, p_1, p_2)$ is a full signature. Let w be the first write to A after s. Then w changes the signature of A.*

Proof. Let q be the writer of w. If $q \notin \{p_0, p_1, p_2\}$, then since q writes its own id, it changes the signature of A. If $q \in \{p_0, p_1, p_2\}$, then by Lemma 4, a different location from $A[i]$ is written by q. Hence w changes the signature of A. □

Lemma 6. *Suppose at scan s_1, $A = (p_0, p_1, p_2)$ is a full signature. Let w be the first write to A after s_1. Let s_2 be any scan after w such that at s_2, $A = (p_0, p_1, p_2)$. Then:*

1. *for some i, $i \in \{0, 1, 2\}$, p_i calls* `Knockout(`σ`)` *and returns* `false` *in the interval $E[w, s_2)$, where $\sigma \neq (p_0, p_1, p_2)$, and*
2. *for all i, $i \in \{0, 1, 2\}$, p_i performs at least one write to A in the interval $E[w : s_2)$.*

Proof. By Lemma 5, w changes the value of A by writing to $A[i]$ for some $i \in \{0, 1, 2\}$. Since at s_2, $A[i] = p_i$, p_i must perform a write to $A[i]$ in the interval $E[w : s_2)$. By Lemma 4, the first write by p_i, say, w_1^i to A in the interval $E[w : s_2)$ is to $A[j]$ where $j \neq i$. Thus p_i must perform at least two writes to A in the interval $E[w : s_2)$. Let s_2^i and w_2^i denote the second and third shared memory steps by p_i in `Compete` method call during $E[w : s_2)$ and let σ be the signature of A at s_2^i. Immediately after w, no location in A contains p_i. In the interval $E[w : s_2^i)$, p_i writes only once and s_2^i is not a losing scan. Hence at s_2^i, p_i appears in $A[j]$, and this is p_i's only location in A. Therefore, σ is full and $\sigma \neq (p_0, p_1, p_2)$. This implies p_i calls `Knockout(`σ`)` after s_2^i. Finally, because w_2^i is a write to A after s_2^i, p_i must return `false` from this `Knockout` call, proving 1.

By way of contradiction assume there is a $k \in \{0, 1, 2\}$, p_k does not perform any write to A in the interval $E[w : s_2)$. Because at s_2, $A[k] = p_k$, no process writes to $A[k]$ in the interval $E[w : s_2)$. In particular, since p_i wrote to $A[i]$ and to $A[j]$ in $E[w : s_2)$, $k \notin \{i, j\}$. Furthermore, to return $A[j]$ to p_j, p_j must over-write w_1^i in the interval $E[w_1^i : s_2)$. By Lemma 4, the first write by p_j, say w_1^j, to A in the interval $E[w : s_2)$ is to a location different from $A[j]$, and by assumption it cannot be to $A[k]$. So w_1^j is to $A[i]$. Therefore, p_j must do a later write, w_2^j, (not necessarily its next write) to $A[j]$ in this interval. Let s_2^j be the scan by p_j preceding w_2^j. At s_2^j, $p_j \in A$. However, $p_j \neq A[k]$ because, by assumption, $A[k]$ is not over-written, and $p_j \neq A[j]$ because p_j's next write is to $A[j]$. Therefore $p_j = A[i]$, implying by Line 9 that $(j - 1) \bmod 3 = i$. Recall that at s_2^i, $p_i = A[j]$, and p_i's next write is to $A[i]$. Hence, by Line 9, $(i - 1 \bmod 3) = j$. However, $(i - 1 \bmod 3) = j$ and $(j - 1 \bmod 3) = i$ is impossible. Hence p_k must perform at least one write to A during $E[w : s_2)$, proving 2. □

Lemma 7. *Suppose at scan s, $A = \sigma$ is a full signature, and there is an $i \in \{0, \ldots, 3\}$ such that $B[i] = (p, \sigma)$. Let w (respectively, w_i) be the last write to A (respectively, $B[i]$) preceding s. Then, w precedes w_i.*

Proof. By way of contradiction suppose w_i precedes w. Then at p's scan that precedes w_i, A must have signature σ. Let $\hat{s}$ be the most recent preceding scan before w in which the signature of A is σ. Let $\hat{w}$ be the first write to A in $E[\hat{s} : s)$. This write exists because $w \in E[\hat{s} : s)$. The write w_i does

not precede $\hat{w}$ because, by Lemma 6 item 1, there is process that executes a complete `Knockout(`σ'`)` in $E[\hat{w} : s)$, where $\sigma \neq \sigma'$, and in this execution that process over-writes every register in B.

We now show that $w_i \notin E[\hat{w} : w)$. By Lemma 6 item 2, p must write to A in the interval $E[\hat{w} : s)$. Since p did no operations between $\hat{w}$ and w_i, this required write would be after w_i and before w. This is impossible because p does a scan after w_i, before this write. Since the signature of A between w_i and w is not σ, this scan would be a losing scan. □

Lemma 8. *There is no execution in which all processes return* `lose`*.*

Proof. By way of contradiction, assume that there is an execution in which all processes return `lose`. Let u be the process that performs the last write to A, let w_u^A be that write, and let σ be the signature of A after w_u^A. Let s_u be the last scan by u. Then u returns `lose` in Line 6 or 8.

First consider the case in which u returns `lose` in Line 6. At s_u, $\text{num}(u, a_u)$ is not equal to 0 because the last write to A is performed by u and s_u happens after w_u^A. Therefore, $\text{num}(u, a_u)= 1$ and there is a process y such that in s_u, $\text{num}(y, a_u) = 2$. Let w_y^A be the last write by y to A. Since u performs the last write to A, w_y^A precedes w_u^A. Because no process writes y to A after w_y^A and no process writes to A after w_u^A and, later, at s_u, $\text{num}(y, a_u) = 2$, it follow that $\text{num}(y, A) \geq 2$ for the entire execution after w_y^A. Therefore any scan by y after w_y^A must satisfy $\text{num}(y, a_y)\geq 2$. This implies y cannot return `lose`, contradicting the assumption.

Next consider the case in which u returns `lose` in Line 8. This implies u calls `Knockout(`σ`)` after w_u^A from which it returns `true` in Line 14 or in Line 15. But u cannot return `true` in Line 14 because the value of array A remains σ after w_u^A. Therefore u returns `true` in Line 15.

Let $S = \{q \mid \exists i, B[i] = (q, \sigma) \text{ at some point after } w_u^A\}$. By Lemma 7, for each $q \in S$, q performs a write (q, σ) to B after w_u^A and, by assumption, subsequently does a losing scan. Because u returns `true` in Line 15, S is not empty. For each $q \in S$, q cannot return `lose` at Line 6 because this would imply q writes to A after w_u^A. Therefore q returns `lose` at Line 8 implying that q returns `true` at Line 14 or Line 15. It does not return `true` in Line 14 because $\text{sig}_q = \sigma$ and the value of A remains σ after w_u^A. Therefore for each $q \in S$, q returns `true` in Line 15, following a losing scan that is after w_u^A. Let z be the last process in S to do its losing scan, s_z. At s_z two registers in B contain (z', σ), where $z' \neq z$. Thus, between the last write by z' and the last scan by z', these two registers in B contain (z', σ). So z' cannot satisfy $\text{num}((z', \sigma), b) \leq 1$ after its last scan, contradicting the assumption that the last scan of z' is a losing scan. □

Lemma 9. *Suppose an interval between a write and the next scan by the same process, say p, contains a winning write. Then the scan by p is a losing scan.*

Proof. At the winning write all registers in A contain the id of the process that performs this winning write. In the sub-interval from the winning write to the scan by p there is no write by p. Since only p writes its id, at p's scan, $\text{num}(p, a_p) = 0$. Hence p returns `lose` after this scan. □

Observation 10. *Every sifting interval contains at least two writes to A.*

Proof. Consider the sifting interval $I = E[s^* : w^*)$. Since w^* is a winning write, $\text{writer}(I)$ must have performed at least two writes to A before w^*, and these two writes must be after the previous winning scan, which is s^*. □

A sifting interval that does not contain a write to A by a process whose next scan is a losing scan is called a *slow sifting interval.*

Lemma 11. *For any slow sifting interval I there exist two distinct processes q_0 and q_1 and a signature σ satisfying: for each i in $\{0,1\}$, during I, q_i performs a write and then a scan in* `Compete` *and then invokes* `Knockout(`σ`)` *and becomes poised to write (q_i, σ) to B. Furthermore, there is no write to A between these two scans.*

Proof. Let I be $E[s^* : w^*)$. Suppose that $w_1^A, w_2^A, \ldots, w_k^A$ is the sequence of all writes to A during I. By Observation 10, $k \geq 2$. For each i, $1 \leq i \leq k$, let s_i denote the next scan by the process that executes w_i^A. Each s_i is at Line 4 following w_i at Line 3 of `Compete`. Let S denote the set of all these scans. Let $\hat{I}$ denote the interval $E[w_2^A : w^*)$.

By Lemma 9, if s_i happens after w^* then s_i is a losing scan and hence $E[s^* : w^*)$ is not a slow sifting interval. Therefore $\forall i$, $1 \leq i \leq k$, s_i occurs in $E[w_i^A : w^*)$. Let q be the process that performs s_1. At s^*, num(scanner$(I), A) = 3$. Because only one write happens to A during $E[s^* : w_2^A)$, q would return `lose` at Line 6 if s_1 precedes w_2^A implying $E[s^* : w^*)$ is not a slow sifting interval. Hence $\forall i$, $1 \leq i \leq k$, s_i must happen in $\hat{I}$. Interval $\hat{I}$ consists of the $k-1$ disjoint sub-intervals $E[w_2^A : w_3^A), E[w_3^A : w_4^A), \ldots, E[w_k^A : w^* = w_{k+1}^A)$. Since k scans happen in these $k-1$ intervals, by the pigeonhole principal, there is a j, $2 \leq j \leq k$ such that (at least) two scans, say $s', s'' \in S$ occur in $E[w_j^A : w_{j+1}^A)$. Because no process performs two scans in `Compete` without writing to A in between, s' and s'' are performed by two distinct processes say q_0 and q_1. Because $E[s^* : w^*)$ is a slow sifting interval, neither q_0 nor q_1 return `lose` at Line 6. Since no write happens to A during $E[w_j^A, w_{j+1}^A)$, the scans by q_0 and q_1 in `Compete` return the same signature for A, say, σ. Therefore q_0 and q_1 both invoke `Knockout(`σ`)`. □

Lemma 12. *Let s be the scan by p immediately before p invokes* `Knockout(`σ`)` *and s' be any scan by p in this invocation. Let w be the first write to A after s. If w precedes s', then s' is a losing scan.*

Proof. Since p invokes `Knockout(`σ`)`, σ is a full signature and p is in σ. By way of contradiction suppose s' is not a losing scan. Hence at s', the signature of A is σ. Therefore, by Lemma 6, p writes to A in the interval $E[w : s')$. This is a contradiction because p is performing `Knockout` method call in this entire interval. □

Lemma 13. *For every slow sifting interval I there is a process p that performs a write during I and either the first or the second scan by p following this write is a losing* `scan`*.*

Proof. Let $I = E[s^* : w^*)$ be a slow sifting interval. By Lemma 11, there exist two processes q_0, q_1 and signature σ satisfying: for each i in $\{0,1\}$, during I, q_i performs a write to A and a scan in `Compete` and calls `Knockout(`σ`)` and becomes poised, at Line 12, to write (q_i, σ) to $B[0]$. Furthermore, there is no write to A between these two scans. Let $\hat{s}$ be the earliest of these two scans. Since both processes proceed to `Knockout`, the ids q_0 and q_1 each occur in exactly one location in σ. At $\hat{s}$, the signature in A is full and at w^* the same id is in all locations of A. Hence there is at least one write to A in $E[\hat{s} : w^*)$. Let w be the first write to A in $E[\hat{s} : w^*)$.

If there is $p \in \{q_0, q_1\}$, such that p performs a scan, say s, in Line 13 of its current call to `Knockout` after w, then by Lemma 12, s is a losing scan. Since p writes at least once in $E[s^* : w^*)$ and at most once after w, it follows that p performs its last or second last write during $E[s^* : w^*)$, and so s is either p's first or second scan following this write, and the lemma holds.

Otherwise, both q_0 and q_1 execute their last scan of their current call to `Knockout` before w. We partition this case into three subcases.

Case 1: There is $p \in \{q_0, q_1\}$ such that p's current `Knockout` call returns `true` (Line 14 or 15). Then p's last scan before returning `true` is a losing scan, and the lemma follows.

Case 2: The current `Knockout` calls of both q_0 and q_1 return `false` and there is a process $p \notin \{q_0, q_1\}$ that performs a write w_p to B with value (p, σ') in the interval $E[\hat{s} : w)$ where $\sigma' \neq \sigma$. When p did its scan in `Compete` just before invoking `Knockout(`σ'`)`, the signature of A was σ'. At w_p, the signature of A is $\sigma \neq \sigma'$, so there is a write to A between this scan by p and w_p. Hence, by Lemma 12, p's next scan after w_p is a losing scan, and again the lemma follows.

Case 3: The current `Knockout` calls of both q_0 and q_1 return `false` and there is no write to B in $E[\hat{s} : w)$ that contains a signature different from σ. Let $\{q', q''\} = \{q_0, q_1\}$ and let s' and s'' be the last scans by q' and q'', respectively, chosen such that s' precedes s''. Since q' returns `false`, all four registers in B contain σ at s'. Process q' does not participate after s', so there are at most two other processes including q'' that can write the signature σ after s'. Therefore, neither can do more than one write to B in $E[s' : w)$, because its next scan after such a write would be a losing scan, contradicting that q'' returns `false`. Therefore, this case is impossible. □

Lemma 14. *For every sifting interval, there is a process p and a write w by p satisfying: either the first operation by p or the third operation by p that follows w is a losing scan.*

Proof. For any sifting interval that is not slow, the lemma holds by definition. For any slow sifting interval, the lemma follows from Lemma 13, because each process alternates between writes and scans. □

Lemma 15. *If k processes invoke the* `Compete` *method, then at most $\lfloor \frac{2k+1}{3} \rfloor$ processes return* `win`*.*

Proof. If k' processes return `win`, then by definition, there are $k'-1$ sifting intervals. By Lemma 13, for each sifting interval there is at least a process that performs its last or second last write and it does not return `win`. Hence there are at least $\lceil \frac{k'-1}{2} \rceil$ extra processes which have invoked `Compete` and cannot return `win`. Since $\lceil \frac{k'-1}{2} \rceil + k' \leq k$, k' is at most $\lfloor \frac{2k+1}{3} \rfloor$. □

Lemma 16. *The Sifter implementation is obstruction-free where each process terminates in $O(1)$ solo steps.*

Proof. Suppose a process, p, begins a solo run while it is executing `Knockout`. If it returns `true` in either Line 14 or Line 15, then it terminates due to Line 8. Otherwise in each iteration of the while loop, it writes a new location in B, and no other process is writing. Therefore after four iterations, all locations in B contain (p, sig_p), and p returns `false` in Line 16. The value of A is equal to sig_p and it does not change when p executes `Knockout`. In sig_p, exactly one location in A contains p and no other process writes to A after it returns from its `Knockout` call. Hence p writes two more times to A and, by Line 5 returns `win`.

Suppose p starts its solo run in a `Compete` call. After at most one write it performs a scan. Then it either returns `win` due to Line 5 or returns `lose` in Lines 6, or it invokes a `Knockout` call. If it calls `Knockout`, then by the argument above it terminates. □

By Lemmas 15 and 16, the algorithm in Figure 1 implements an obstruction-free ($\lfloor \frac{2k+1}{3} \rfloor$)-Sifter using 7 $O(\log n)$-bit registers, if a linearizable `scan()` is available. As explained in Section 2, we need one additional register to realize such a `scan()` operation. Hence, we obtain an ($\lfloor \frac{2k+1}{3} \rfloor$)-Sifter using 8 registers. This completes the proof of Theorem 3, and thus Theorem 1 follows.

5 Randomized Test-and-Set

We can combine our algorithm with existing algorithms and techniques from [14, 15], to obtain a randomized test-and-set implementation from registers that uses logarithmic space, and has almost

constant, $O(\log^* n)$, expected step complexity against an oblivious adversary. Next we present such an implementation that has the properties stated in Theorem 2.

In [14, Theorem 2] it was shown that any obstruction-free test-and-set algorithm, in which any process finishes after at most b steps if it runs solo, can be transformed into a randomized one with the same space complexity, where each process finishes after at most $O\big(b(n+b)\log(n/\delta)\big)$ steps with probability at least $1-\delta$, for any $0 < \delta < 1$. Since $b = O(\log n)$ for our obstruction-free algorithm (see Theorem 1), it follows that it can be transformed into a randomized algorithm without increasing its logarithmic space, and with each process finishing in at most $O(n \log^2 n)$ steps, both in expectation and w.h.p.

To achieve the $O(\log^* n)$ step complexity, we combine the above algorithm with a randomized test-and-set construction proposed in [15]. The construction in [15] uses a chain of n sifter objects $F_1, \ldots, F_n$ alternating with n deterministic splitters $S_1, \ldots, S_n$, and a chain of n 2-process consensus objects $C_1, \ldots, C_n$. (A splitter returns `win`, `lose`, or `continue`, such that at most one process wins, not all processes lose, and not all continue.) A process p proceeds by accessing the sifters in increasing index order. If p loses in some sifter F_i then it loses immediately in the implemented test-and-set; if it wins, it tries to win splitter S_i. If p loses S_i, it loses also the test-and-set; if S_i returns `continue`, p continues to the next sifter, F_{i+1}; and if p wins S_i, it switches to the chain of 2-process consensus objects. In the last case, p then tries to win $C_i, C_{i-1}, \ldots, C_1$ (in this order). If it succeeds, it wins the test-and-set, otherwise it loses.

In [15], a randomized sifter object from s single-bit registers was proposed, with the properties that each process makes $O(1)$ steps, and if at most 2^s processes access the object then at most $O(s)$ of them win in expectation. Another property of this sifter we will need is that, for $s = \log n$, we have that at most $O(\log^2 n)$ processes win w.h.p.[1] Finally, we have that if $s = 2$ and k processes access the sifter, then at most $k/2 + 1$ processes win in expectation. In the following, we will refer to the above sifter object as *GW-sifter of size s*.

The test-and-set implementation in [15] uses n GW-sifters of size $\log n$ each, and thus requires $\Theta(n \log n)$ registers in total. We modify this implementation as follows: The first sifter object, F_1, is a GW-sifters of size $\log n$, as before. The next $\ell = \sqrt{\log n}$ sifters, $F_2, \ldots, F_{\ell+1}$, are GW-sifters of size $\sqrt{\log n}$. Then, the next $m = 3 \log n$ sifters, $F_{\ell+2}, \ldots, F_{\ell+m+1}$ are GW-sifters of size 2. Finally, sifter $F_{\ell+m+2}$ is the randomized variant of our *test-and-set* algorithm described earlier. All objects F_i, S_i, and C_i for $i > \ell + m + 2$ are dismissed.

It is straightforward to verify that this implementation uses $\Theta(\log n)$ registers (note that $O(1)$ registers are needed to implement each of the objects S_i and C_i). The reason we can use sifters of size only $\sqrt{\log n}$ instead of $\log n$ starting from the second sifter, is that, w.h.p. at most $O(\log^2 n)$ processes win the first sifter (of size $\log n$), as we mentioned earlier. (So, in fact, it would suffice that those sifters be of size $\Theta(\log \log n) < \sqrt{\log n}$.) Then from the analysis in [15] we have that in expectation only the first $O(\log^* n)$ sifter objects are used, and with probability $1 - O(1/\log n)$, at most the first $O(\log \log n) < \ell$ sifters are used. Thus, only with probability $O(1/\log n)$ the constant-size sifters will be used. Since in each of these sifters half of the processes lose in expectation, and we have $m = 3 \log n$ such sifters, it follows that with probability $1 - O(1/n^2)$ no process will access the last sifter, $F_{\ell+m+2}$. From the above, it follows that the expected value of the maximum number of steps by a process is $O(\log^* n) + O(1/\log n) \cdot 3 \log n + O(1/n^2) \cdot O(n \log^2 n) = O(\log^* n)$.

[1] This property is not stated in [15], but can be proved using standard probabilistic arguments.

Acknowledgements

This research was undertaken, in part, thanks to funding from the Canada Research Chairs program, the Discovery Grants program of NSERC, and the INRIA Associate Team RADCON.

References

[1] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapsots of shared memory. *Journal of the ACM*, 40(4):873–890, 1993.

[2] Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG)*, pages 85–94, 1992.

[3] Zahra Aghazadeh and Philipp Woelfel. Space- and time-efficient long-lived test-and-set. In *Proceedings of the 18th International Conference on Principles of Distributed Computing (OPODIS)*, 2014. To appear.

[4] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Proceedings of the 25th International Symposium on Distributed Computing (DISC)*, pages 97–109, 2011.

[5] Dan Alistarh, James Aspnes, Keren Censor-Hillel, Seth Gilbert, and Morteza Zadimoghaddam. Optimal-time adaptive strong renaming, with applications to counting. In *Proceedings of the 30th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, 2011.

[6] Dan Alistarh, James Aspnes, Seth Gilbert, and Rachid Guerraoui. The complexity of renaming. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 718–727, 2011.

[7] Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th International Symposium on Distributed Computing (DISC)*, pages 94–108, 2010.

[8] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, April 1993.

[9] Harry Buhrman, Alessandro Panconesi, Riccardo Silvestri, and Paul Vitányi. On the importance of having an identity or, is consensus really universal? *Distributed Computing*, 18(3):167–176, 2006.

[10] James Burns and Nancy Lynch. Bounds on shared memory for mutual exclusion. *Information and Computation*, 107(2):171–184, 1993.

[11] Wayne Eberly, Lisa Higham, and Jolanta Warpechowska-Gruca. Long-lived, fast, waitfree renaming with optimal name space and high throughput. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, pages 149–160, 1998.

[12] Faith Ellen, Panagiota Fatourou, and Eric Ruppert. Time lower bounds for implementations of multi-writer snapshots. *Journal of the ACM*, 54(6), 2007.

[13] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free algorithms can be practically wait-free. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC)*, pages 78–92, 2005.

[14] George Giakkoupis, Maryam Helmi, Lisa Higham, and Philipp Woelfel. An $O(\sqrt{n})$ space bound for obstruction-free leader election. In *Proceedings of the 27th International Symposium on Distributed Computing (DISC)*, pages 46–60, 2013.

[15] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *Proceedings of the 31st SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 19–28, 2012.

[16] George Giakkoupis and Philipp Woelfel. A tight RMR lower bound for randomized mutual exclusion. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pages 983–1002, 2012.

[17] Wojciech Golab, Danny Hendler, and Philipp Woelfel. An $O(1)$ RMRs leader election algorithm. *SIAM Journal on Computing*, 39(7):2726–2760, 2010.

[18] Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. RMR-efficient implementations of comparison primitives using read and write operations. *Distributed Computing*, 25(2):109–162, 2012.

[19] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In *Proceedings of the 24th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 258–264, 2005.

[20] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991.

[21] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, pages 522–529, 2003.

[22] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[23] Clyde Kruskal, Larry Rudolph, and Marc Snir. Efficient synchronization on multiprocessors with shared memory. *ACM Transactions on Programming Languages and Systems*, 10(4):579–601, 1988.

[24] Alessandro Panconesi, Marina Papatriantafilou, Philippas Tsigas, and Paul M. B. Vitányi. Randomized naming using wait-free shared variables. *Distributed Computing*, 11(3):113–124, 1998.

[25] Eugene Styer and Gary Peterson. Tight bounds for shared memory symmetric mutual exclusion problems. In *Proceedings of the 8th SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 177–191, 1989.

[26] John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distributed Computing*, 15(3):127–135, 2002.