Test-and-Set in Optimal Space

George Giakkoupis, Maryam Helmi, Lisa Higham, Philipp Woelfel

Test-and-Set in Optimal Space

George Giakkoupis
INRIA Rennes
Rennes, France
george.giakkoupis@inria.fr

Maryam Helmi
University of Calgary
Calgary, Canada
mhelmikh@ucalgary.ca

Lisa Higham
University of Calgary
Calgary, Canada
higham@ucalgary.ca

Philipp Woelfel
University of Calgary
Calgary, Canada
woelfel@ucalgary.ca

ABSTRACT

The test-and-set object is a fundamental synchronization primitive for shared memory systems. This paper addresses the number of registers (supporting atomic reads and writes) required to implement a one-shot test-and-set object in the standard asynchronous shared memory model with n processes. The best lower bound is $\log n - 1$ [12, 21] for obstruction-free and deadlock-free implementations, and recently a deterministic obstruction-free implementation using $O(\sqrt{n})$ registers was presented [11].

This paper closes the gap between these existing upper and lower bounds by presenting a deterministic obstruction-free implementation of a one-shot test-and-set object from $\Theta(\log n)$ registers of size $\Theta(\log n)$ bits. Combining our obstruction-free algorithm with techniques from previous research [11, 12], we also obtain a randomized wait-free test-and-set algorithm from $\Theta(\log n)$ registers, with expected step-complexity $\Theta(\log^* n)$ against the oblivious adversary. The core tool in our algorithm is the implementation of a deterministic obstruction-free *sifter* object, using only 6 registers. If k processes access a sifter, then when they have terminated, at least one and at most $\lfloor (2k+1)/3 \rfloor$ processes return "win" and all others return "lose".

Categories and Subject Descriptors

E.1 [**Data Structures**]: Distributed data structures; D.1.3 [**Concurrent Programming**]: [Distributed Programming]; F.1.2 [**Modes of Computation**]: [Parallelism and Concurrency]; F.2 [**Analysis of Algorithms and Problem Complexity**]: General

Keywords

Shared Memory Model, Test-and-Set Algorithm, Space Complexity, Obstruction-free

STOC'15, June 14–17, 2015, Portland, Oregon, USA.

ACM 978-1-4503-3536-2/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2746539.2746627.

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. A one-shot test-and-set is a one-bit register, which is initially 0, and supports one operation, namely `test&set()`. A `test&set()` sets the bit's value to 1 and returns its previous value. The TAS problem is related to *weak leader election*, where participating processes elect a leader among themselves. The leader returns `win`, and all other participating processes return `lose`. (The term leader election is ambiguous. It is also used to denote the *name consensus* problem, where the losing processes need to output the ID of the winner. We add the qualifier "weak" in order to distinguish the two variants.)

Leader election and TAS are equally hard problems: A single `test&set()` call solves leader election (by replacing the return values 0 and 1 with `win` and `lose`, respectively). On the other hand, there is a wait-free linearizable `test&set()` implementation, where in addition to participating in a leader election, a process needs to perform only two steps on a single register [14].

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 solve many classical problems, such as mutual exclusion and renaming [4–7, 9, 19, 20]. Since TAS objects are among the simplest synchronization primitives, they are well suited to investigate the difficulties arising in synchronization problems. Algorithms or impossibility results for TAS can provide insights into the complexity of other shared memory problems, and can contribute to 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 register 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's steps, independent of other processes. Since TAS has consensus number two, no deterministic wait-free TAS implementation from registers exists, even for 2 processes. A weaker progress condition, and the one most frequently used for analyzing space complexity, is *obstruction-freedom* [17]. 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 [16].

The randomized step complexity of TAS has been thoroughly investigated, with significant progress being made in recent years [1, 3, 6, 13, 22]. In contrast, little was known about the space complexity of obstruction-free or randomized wait-free TAS. In 1989, Styer and Peterson [21] studied the space complexity of the related mutual exclusion problem, under the deadlock-free progress requirement. As a special case they also considered a variant of deadlock-free weak leader election, where only the leader has to terminate. It suffices to add a single one-bit register to transform any such deadlock-free leader election protocol into a linearizable deadlock-free TAS. Styer and Peterson proved a space lower bound of $\lceil \log n \rceil + 1$ registers, and provided an algorithm that established that this bound is tight. Hence, in the case of deadlock-freedom, Styer and Peterson's results answer the question of the space complexity of one-shot TAS precisely up to a single register.

Deadlock-freedom is a natural progress property for mutual exclusion related problems, where waiting for other processes is inherent in the problem specification. But for other problems, it is inappropriate because it does not preclude a single slow or failing process preventing all other processes from making progress. Alternative progress properties, such as obstruction-freedom, lock-freedom, or randomized wait-freedom, are more desirable for such problems. Research on the space complexity of shared memory problems has focused on the obstruction-free progress property [10, 15, 17, 18]. However, despite significant research on TAS, until recently the space complexity of obstruction-free TAS implementations remained an unanswered question.

In 2012, Giakkoupis and Woelfel [13] used the same lower bound technique as that of Styer and Peterson to conclude that obstruction-free one-shot TAS requires $\lceil \log n \rceil - 1$ registers. 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 [11]. We now present an asymptotically tight result.

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

In general, there are performance benefits if the solo run that is required for termination is short, because processes have a better chance of completing their TAS before they get interrupted. Our algorithm satisfies an even stronger property: A process needs to execute only a constant number of solo steps $\Theta(\log n)$ times, to finish its `test&set()` method call.

Giakkoupis and Woelfel [13] 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 [11] 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 two previous algorithms [11, 13] 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 $\Theta(\log n)$ space algorithm.

THEOREM 2. *There is a randomized TAS implementation from $\Theta(\log n)$ registers of size $\Theta(\log n)$ 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, Aghazadeh and Woelfel [2] 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 the proof of Theorem 2, yields a long-lived TAS object implemented 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 [2, 11]. The space lower bound for mutual exclusion [8] implies that any long-lived TAS implementation requires at least n registers.

2. MODEL AND TOOLS

Our model of computation is the standard asynchronous shared memory model where a set $\mathcal{P}$ of n processes with distinct identifiers communicate through a set of shared multi-reader multi-writer registers. Each register supports two atomic operations, read and write. To simplify our algorithm and proofs, we use a modified model that uses writes and scans where each scan returns the content of all registers instantaneously. This is not a loss of generality because, as proved in [11], with one additional register, any system that supports scans and writes can be replaced with atomic multi-reader multi-writer registers.

THEOREM 3. *[11] An obstruction-free implementation of a* `scan()` *operation can be obtained by using one reserved register in addition to those that can be scanned.*

Each process runs a program that can access the process's local registers as well as shared registers. A *schedule* is a sequence of process identifiers. A schedule, σ, gives rise to a sequence of operations, called an *execution* as follows. The i^{th} operation in the execution is the next operation in the program of the i^{th} process in σ. In our proofs, it suffices to consider just the shared memory 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.

An 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 intervening 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, p completes its method call in a finite number of its own steps, provided there are no intervening steps taken by other processes. In such an execution, we say that p runs *solo* during these uninterrupted steps by p.

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 is finite in expectation, regardless of the intervening steps taken by other processes [16].

A *test-and-set* (*TAS*) object 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 *weak leader election*, where each participating process has to decide on one value, `win` or `lose`. Among all processes that finish their leader election protocol, at most one process is allowed to win, and not all processes may lose. Hence, if all processes finish, then exactly one process, the leader, wins. Weak leader election and test-and-set are equally hard problems: Replacing the return values 0 and 1 of a `test&set()` operation with `win` and `lose`, respectively, yields a weak leader election protocol. On the other hand, Golab, Hendler and Woelfel gave an implementation of a TAS object using weak leader election and one additional register [14].

THEOREM 4. *[14] A linearizable TAS object can be implemented using a weak leader election protocol and one additional multi-reader/multi-writer binary register, such that a* `test&set()` *operation requires only a constant number of read and write operations in addition to the leader election protocol.*

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`. In this section we prove that:

THEOREM 5. *There is an obstruction-free implementation of an* $(\lfloor \frac{2k+1}{3} \rfloor)$*-sifter using 6 registers, each of size* $\Theta(\log n)$*-bits, where the atomic operations are:* scan *all registers and* write *to a designated register, and any process finishes its* `compete()` *method call in a constant number of solo steps.*

Recent randomized TAS constructions [3,13] 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. Hence, combining $O(\log n)$ $(\lfloor \frac{2k+1}{3} \rfloor)$-sifter objects yields a 1-sifter. A 1-sifter is a weak leader election protocol.

Hence, Theorem 1 follows from Theorems 5, 4 and 3. The remainder of this section is devoted of the proof of Theorem 5.

Sifter implementation

To aid intuition for our sifter implementation, we first consider a very simple obstruction-free sifter object, A. Object A 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 A 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 six steps without losing, will return `win`. Furthermore, not all processes can return lose. To see this, let w be the last write to A and let p be the writer of w. If process p returns `lose`, then there is a process q that occupies two positions in p's last scan, so q cannot return `lose`. Therefore, A is an implementation of an obstruction-free sifter object.

Object A, however, is not a very efficient sifter. Suppose that while a clean-sweep is being achieved by one process, two other processes cover two distinct registers of A. Then these covering processes can over-write the clean-sweep, and be made to again cover two distinct registers. Now a new process can run under the cover and achieve a clean sweep. By repeating this scenario, executions are easily created where all but one process returns `win`. 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 return `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 adding to A a second similar (but not identical) component, B. Component B is an array of three registers, 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. Each process begins by competing on component A and still strictly alternates between writing and scanning.

If process p, competing on component A, gets a scan with signature of A equal to s, where the identifiers in s are all distinct and one of them is p, then p leaves A to compete on component B while remembering s. (Notice that, if p does not get such a scan and it does not immediately return `lose`, then p is in at least two positions in s. Therefore, 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 3 registers of B contains (p, s)). There are two ways that process p can lose while playing on component B. First, p loses if, while trying to achieve a clean-sweep of B, one of p's scans shows a signature of A different from s. Second, p loses if its scan shows that for some other process q, (q, s) occupies at least 2 positions of B. That is, p only returns to continue competing on component A if it achieves a clean-sweep of B while each of its scans satisfies 1) the signature

of A is s, and 2) no other process with signature s occupies more than one register of B.

Our sifter implementation is presented in Figure 1.

Intuition for correctness proof

Our proof will establish that not all processes can return `lose`, and at most one more than 2/3 of the processes that participate can return `win`. While the proof has to attend to several subtleties and substantial detail, there are several insights that aid our intuition. Consider the three ways that a process can return `lose`. Let us say p *loses on* A if process p loses while playing on component A because the signature of A in its last scan contained more occurrences of some other process than occurrences of p. We say p *signature-loses on* B if process p with signature s, loses while trying to achieve a clean-sweep of B, because one of p's scans shows a signature of A different from s. We say p *process-loses on* B if process p loses because its scan shows that for some other process q, (q, s) occupies at least 2 positions of B.

For the intuition that not all processes return `lose`, consider the last write, say w, to A, and let p be the process that executes w. Process p cannot lose on A because if it did, then in p's last scan there is some process, $q \neq p$, that occupies 2 positions on A, and that process cannot return `lose` without first writing to A after w. Similarly, p cannot signature-lose on B because, again, that would imply a write to A after w. So suppose p process-loses on B. Then we show that there is some other process, say q, that has the same signature as p and is competing with p, and q cannot process-lose on B. Process q also cannot signature-lose on B or lose on A without a write to A happening after w.

Now consider the intervals in an execution between the final scans of processes that return `win` (achieve a clean-sweep of A). If ℓ processes return `win`, there are $\ell - 1$ such intervals. We associate a losing process with each of these intervals as follows. (1) For any process, p, if p loses on A, then associate p with the interval that contains the last write by p. Similarly, (2) for any process q that signature-loses on B, associate q with the interval that contains its last write.

Let I be an interval without an associated losing process via either (1) or (2). We will prove that there is a subinterval I' of I and either two or three processes that, during I', move from A to B and finish competing on B using some signature, say s, while the signature of A remains s. Now we focus on the execution during I'. Since B has three registers, after any clean-sweep on B, a subsequent clean-sweep on B requires (at least) two processes to over-write the previous clean-sweep. These over-writers must have signature s, because, otherwise, an over-writer has a signature different from that of A and would signature-lose on B, implying that I already has an associated losing process via (2). If there are two processes with signature s then I' can have at most one clean-sweep, and if there are three processes then I' can have at most two clean-sweeps. Therefore, at least one of the two or three processes competing on B with signature s cannot return `win`, and (3) one such process is associated with I. Notice, however, that this process could withhold its last write in order to be assigned to a later interval via (2).

Therefore, using these three rules of association, we assign a losing process to every interval, and no process is assigned to more than two of these intervals. Thus there are at least $(\ell - 1)/2$ processes that cannot return `win`.

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 $E = \mathrm{op}_1, \mathrm{op}_2 \ldots$, we denote the contiguous subsequence of E starting at op_i and ending at the operation immediately before op_j by $E[\mathrm{op}_i : \mathrm{op}_j)$. In our analysis we are interested in intervals between any winning scan and the next winning write. A *sifting interval* is a subsequence of an execution that starts at some winning scan and ends at the operation immediately before the next winning write. Observe that all sifting intervals are disjoint. Also because there has been a preceding winning scan, no register of A contains $\perp$ in any sifting interval. 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. A local variable x in the algorithm is denoted by x_p when it is used in the method call invoked by process p.

Proof of correctness

LEMMA 6. *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 11, w_i is a write to $A[\mathrm{pos}]$ where $\mathrm{pos} \neq i$. □

LEMMA 7. *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 6, a different location from $A[i]$ is written by q. Hence w changes the signature of A. □

Shared Objects:

- $A[0,1,2]$ is an array of registers. Each array entry stores a value from $\mathcal{P} \cup \{\perp\}$ and is initially $\perp$.
- $B[0,1,2]$ is an array of registers. Each array entry 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 $\text{sig} = (\perp, \perp, \perp)$.

Notation: for any array X and value v, we define $\text{num}(v, X) := |\{i : X[i] = v\}|$.

Algorithm 1: 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
6          return win
7      if ∃ q ∈ P, num(p, a) < num(q, a) then
8          return lose
9      if num(p, a) = 1 then
10         if knockout(a) then return lose
11     Let pos ∈ {0,1,2} s.t. a[(pos − 1) mod 3] = p and a[pos] ≠ p
```

Algorithm 2: knockout(sig)

```
12 index := 0;
13 while true do
14     B[index].write((p, sig));
15     (a, b) := scan(A, B);
16     if a ≠ sig then return true;
17     if ∃ q ∈ P, q ≠ p, num((q, sig), b) ≥ 2 then
18         return true
19     if ∀ i ∈ {0,1,2}, b[i] = (p, sig) then
20         return false
21     Let index ∈ {0,1,2} s.t. b[index] ≠ (p, sig);
```

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

LEMMA 8. *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 7, 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 6, 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 6, 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 11 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 11, $(i - 1 \bmod 3) = j$. 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 9. *Suppose at scan s, $A = \sigma$ is a full signature, and there is an $i \in \{0,1,2\}$ 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 8 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 8 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 10. *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 8 or 10.

First consider the case in which u returns `lose` in Line 8. At s_u, 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, num(u, a_u)= 1 and there is a process y such that in s_u, 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, num$(y, a_u) = 2$, it follow that num$(y, A) \geq 2$ for the entire execution after w_y^A. Therefore any scan by y after w_y^A must satisfy 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 10. This implies u calls `knockout(`σ`)` after w_u^A from which it returns `true` in Line 16 or in Line 18. But u cannot return `true` in Line 16 because the value of array A remains σ after w_u^A. Therefore u returns `true` in Line 18.

Let $S = \{q \mid \exists i, B[i] = (q, \sigma)$ at some point after $w_u^A\}$. By Lemma 9, 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 18, S is not empty. For each $q \in S$, q cannot return `lose` at Line 8 because this would imply q writes to A after w_u^A. Therefore q returns `lose` at Line 10 implying that q returns `true` at Line 16 or Line 18. It does not return `true` in Line 16 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 18, 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 after z''s last scan at most one register of B contains an id not equal to z'. Therefore z' cannot return `true` in Line 18, contradicting the assumption that the last scan of z' is a losing scan. □

LEMMA 11. *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, num$(p, a_p) = 0$. Hence p returns `lose` after this scan. □

OBSERVATION 12. *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, 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 13. *For any slow sifting interval I, there exists a signature $\sigma = (q_0, q_1, q_2)$ and a set $Z \subseteq \{0, 1, 2\}$ satisfying: $|Z| = 2$ and for each $z \in Z$ during I, q_z performs a write and then a scan in* `compete()` *and then invokes* `knockout(`σ`)` *and becomes poised to write (q_z, σ) to B. Furthermore, there is no write to A between these two scans.*

PROOF. Let I be $E[s^* : w^*)$. Suppose that $w_1^A, w_2^A, \ldots, w_\ell^A$ is the sequence of all writes to A during I. By Observation 12, $\ell \geq 2$. For each i, $1 \leq i \leq \ell$, 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 11, 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 $\forall i$, $1 \leq i \leq \ell$, 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 8 if s_1 precedes w_2^A implying $E[s^* : w^*)$ is not a slow sifting interval. Hence $\forall i$, $1 \leq i \leq \ell$, s_i must happen in $\hat{I}$. Interval $\hat{I}$ consists of the $\ell - 1$ disjoint sub-intervals $E[w_2^A : w_3^A), E[w_3^A : w_4^A), \ldots, E[w_\ell^A : w^* = w_{\ell+1}^A)$. Since ℓ scans happen in these $\ell - 1$ intervals, by the pigeonhole principal, there is a j, $2 \leq j \leq \ell$ such that (at least) two scans in S, say s' and 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_z and $q_{z'}$. Because $E[s^* : w^*)$ is a slow sifting interval, neither q_z nor $q_{z'}$ return `lose` at Line 10. Since no write happens to A during $E[w_j^A, w_{j+1}^A)$, the scans by q_z and $q_{z'}$ in `compete()` return the same signature for A, say, σ where σ contains q_z and $q_{z'}$. Therefore q_z and $q_{z'}$ both invoke `knockout(`σ`)`. □

LEMMA 14. *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 8(point 2), p writes to A in the interval $E[w : s')$. This is a contradiction because p is performing `knockout(`σ`)` in this entire interval and there are no writes to A during the `knockout` method call. □

LEMMA 15. *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 13, there exists a full signature $\sigma = (q_0, q_1, q_2)$, a set $Q \subseteq \{q_0, q_1, q_2\}$ satisfying $2 \leq |Q| \leq 3$ and for each $q \in Q$ during I, q performs a write to A and a scan in `compete()` and calls `knockout(`σ`)` and becomes poised, at Line 14, to write (q, σ) to $B[0]$. Furthermore, there is no write to A between these scans. Let $\hat{s}$ be the earliest of these scans. 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 $E[\hat{s} : w^*)$.

Suppose there is $q \in Q$, such that q performs a scan, say s, in Line 15 of its current call to `knockout` after w. Then by Lemma 14, s is a losing scan. Since q writes at least once in $E[s^* : w^*)$ and at most once after w, it follows that q performs its last or second last write during $E[s^* : w^*)$, and so s is either q's first or second scan following this write, and the lemma holds.

Otherwise, all processes in Q execute at least one write and perform their last scan of their current call to `knockout` before w. We partition this case into three subcases.

Case 1: There is $q \in Q$ such that q calls `knockout(`σ`)` and returns `true` (Line 16 or 18). Then q's last scan before returning `true` is a losing scan, and the lemma follows.

Case 2: For each process $q \in Q$, q's current `knockout` call returns `false` and there is a process $p \notin Q$ 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 14, p's next scan after w_p is a losing scan, and again the lemma follows.

Case 3: For each process $q \in Q$, q's current `knockout` call returns `false` and there is no write to B in $E[\hat{s} : w)$ that contains a signature different from σ. We show that this case is impossible. Let S be the set of last scans of `knockout` calls by processes in Q. Let s'' be the last scan and s' be the second last scan in set S. Let q' and q'' be the processes performing s' and s'' respectively. Since q' returns `false`, all three registers in B contain (q', σ) at s'. After s', there can be at most one write to B by q''. Because q'''s next scan after such a write would be a losing scan, contradicting that q'' returns `false`. □

LEMMA 16. *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 15, because each process alternates between writes and scans. □

LEMMA 17. *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 15, for each sifting interval there is a process that performs its last or second last write and it cannot return `win`. Hence there are at least $\left\lceil \frac{k'-1}{2} \right\rceil$ processes which have invoked `compete()` and cannot return `win`. Since $\left\lceil \frac{k'-1}{2} \right\rceil + k' \leq k$, k' is at most $\lfloor \frac{2k+1}{3} \rfloor$. □

LEMMA 18. *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 16 or Line 18, then it terminates due to Line 10. Otherwise in each iteration of the while loop, it writes a new location in B. Therefore after three iterations, all locations in B contain (p, sig_p), and p returns `false` in Line 20. When p executes `knockout` during its solo run, the value of A is equal to sig_p because otherwise p returns `true` from its `knockout` call. 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 6 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 Line 8, or it invokes a `knockout` call. If it calls `knockout`, then by the argument above it terminates. □

By Lemmas 17 and 18, the algorithm in Figure 1 implements an obstruction-free $(\lfloor \frac{2k+1}{3} \rfloor)$-Sifter using 6 $O(\log n)$-bit registers, if a linearizable `scan()` is available.

Optimizing the sifter for TAS

We presented the sifter object with the smallest possible sizes for components A and B. Both consist of 3 registers. However a natural question is whether or not the space can be reduced by increasing the size of B. In fact, if we use a B component of size 4 instead of 3, for sufficiently large n, the space result for TAS is slightly smaller because fewer sifters are required. Here is the intuition. Suppose that B has size b. For every sifting interval that is not slow we associate a process that performs its last write to A and returns `lose` subsequently. For every slow sifting interval there exists at least two processes that invoke `knockout` with the same signature σ. One possibility in this case is that, from these two processes, one gets knocked out in B and performs its second last write in this interval and returns `lose` subsequently. The only way to have both of these processes returned `false` from `knockout`(σ), is by obliterating the trace of the first process that returns `false` so the other one can also return `false`. This obliteration can happen only by accumulating at least $b-2$ processes that have performed their second last writes in previous sifting intervals and are poised to perform their last writes. Therefore, the worst case of our analysis is when there are $b-2$ sifting intervals such that in each, one process performs its second last write, followed by a single sifting interval, I, in which these $b-2$ processes perform their last writes and return `lose`. These last writes manage to prevent a new process from being knocked out in B during interval I. Hence, for every $b-1$ sifting intervals there are at least $b-2$ losers. Suppose k processes invoke `compete()`. If k' processes return `win`, then by definition, there are $k'-1$ sifting intervals. Therefore there are at least $\left\lceil \frac{b-2}{b-1}(k'-1) \right\rceil$ processes which have invoked `compete()` and cannot return `win`. Since $\frac{b-2}{b-1}(k'-1)+k' \leq \left\lceil \frac{b-2}{b-1}(k'-1) \right\rceil + k' \leq k$, we have $k' \leq (\frac{b-1}{2b-3})k + \frac{b-2}{2b-3}$. Thus, we need at most $\log_{\frac{2b-3}{b-1}} n + 2$ sifters each of size $3+b$. Therefore the space complexity of our test-and-set object is at most $(3+b)(\log_{\frac{2b-3}{b-1}} n + 2)$. For $b = 4$ this function is less than $9.5 \log n + 14$. For $b = 3$, the space used in the worst case by our algorithm is at least $6 \log_{\frac{3}{2}} n \geq 10.25 \log n$. So we can conclude that for $n \geq 2^{18.46}$, our algorithm requires less space when component B has 4 registers instead of 3.

4. RANDOMIZED TEST-AND-SET

We can combine our TAS algorithm with existing algorithms and techniques from [11, 12], to obtain a randomized TAS 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 [11, Theorem 2] it was shown that any obstruction-free TAS 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 pro-

cess 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 randomized algorithm with another randomized TAS construction proposed in [12]. The construction in [12] 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 TAS; if it wins sifter F_i, it tries then to win splitter S_i. If p loses S_i, it loses also the TAS; 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 TAS, otherwise it loses.

In [12], a randomized sifter implementation from $s \geq 1$ single-bit registers is proposed. In this implementation each process takes $O(1)$ steps, and if at most 2^s processes access the sifter then at most $O(s)$ of them win in expectation. Further, for $s = \log n$, we have that w.h.p. at most $O(\log^2 n)$ processes win.[1] Finally, 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 TAS implementation in [12] 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 TAS 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 [12] 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 $O(\log^* n) + O(1/\log n) \cdot 3 \log n + O(1/n^2) \cdot O(n \log^2 n) = O(\log^* n)$.

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.

[1]This property is not shown in [12], but follows from standard probabilistic arguments.

5. REFERENCES

[1] 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.

[2] 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)*, pages 404–419, 2014.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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