Sub-Logarithmic Test-and-Set Against a Weak Adversary*

Dan Alistarh[1] and James Aspnes[2]

[1] EPFL
[2] Yale University

Abstract. A randomized implementation is given of a test-and-set register with $O(\log \log n)$ individual step complexity and $O(n)$ total step complexity against an oblivious adversary. The implementation is linearizable and multi-shot, and shows an exponential complexity improvement over previous solutions designed to work against a strong adversary.

1 Introduction

A *test-and-set* object supports an atomic test-and-set operation, which returns 0 to the first process that executes it and 1 to all subsequent processes. Test-and-set is a classic synchronization primitive, often used in multiprocessing computer systems as a tool for implementing mutual exclusion. It also has close connections to the traditional distributed computing problems of *consensus* [11] and *renaming* [2]. For two processes, test-and-set can be used to solve consensus and vice versa [11]; this implies that test-and-set has no deterministic wait-free implementation from atomic registers. Nonetheless, randomized implementations can solve test-and-set efficiently.

The randomized test-and-set object of Afek *et al.* [1] requires $O(\log n)$ steps on average, where n is the number of processes. It is built from a tree of 2-process test-and-set objects that are in turn built from 2-process randomized consensus protocols. The performance bounds hold even when scheduling is under the control of an *adaptive adversary*, which chooses at each step which process executes the next low-level operation based on complete knowledge of the system, including the internal states of processes.

In the case of consensus, it is known that replacing an adaptive adversary with an *oblivious adversary*, that fixes the entire schedule in advance, improves performance exponentially, from an $\Omega(n)$ lower bound on the expected number of steps performed by any one process [3] to an $O(\log n)$ upper bound [6]. Thus, a natural question is whether an algorithm with step complexity lower than $\Theta(\log n)$ is possible for test-and-set against a weak adversary.

In this paper, we answer this question in the affirmative. We show that, even though test-and-set has a fast $O(\log n)$ implementation against an adaptive adversary, this same exponential improvement holds: by exploiting the limitations of the oblivious adversary, we can reduce the complexity of test-and-set from $O(\log n)$ to $O(\log \log n)$.

* The work of Dan Alistarh was supported by the NCCR MICS Project. The work of James Aspnes was supported in part by NSF grant CCF-0916389.

The essential idea of our algorithm is to rapidly eliminate most processes from consideration using a sequence of *sifting* rounds, each of which reduces the number of survivors to roughly the square root of the number of processes that enter the round, with high probability; in particular, this reduces the number of survivors to polylogarithmic in $O(\log \log n)$ rounds.

The intuition behind the sifting technique is quite simple: each process either writes or reads a register in each round with a carefully-tuned probability. The process continues to the next round only if it chooses to write, or if it reads the value $\perp$, indicating that its read preceded any writes in that round. Because an oblivious adversary cannot predict which process will read and which will write, it effectively plays a game where the processes access the register one at a time, with only writers surviving after the first write; the probabilities are chosen so that the sum of the expected number of initial readers and the expected number of writers is minimized. At the same time, this scheme ensures that *at least one* process survives each round of sifting, because either all writers survive, or, if there are no writers, all readers survive. This technique works despite asynchrony or process crashes.

After $\Theta(\log \log n)$ rounds of sifting, the number of remaining candidates is small enough that the high-probability bounds used to limit the number of survivors in each round stop working. On the other hand, we notice that in this case we are left with $O(\text{polylog}\, n)$ survivors, and we can feed these survivors into a second stage consisting of the adaptive test-and-set implementation of [2], that has step complexity $O(\log k)$ for k participating processes. Thus, the running time of this second stage is also $O(\log \log n)$, which yields the step complexity upper bound of $O(\log \log n)$. A similar analysis shows that the total number of steps that all processes take during an execution of the algorithm is $O(n)$, which is clearly optimal.

It is worth noting that in the presence of an adaptive adversary, though the initial sifting phase fails badly (the adversary orders all readers before all writers, so all processes survive each round), the adaptive test-and-set still runs in $O(\log n)$ time, and the $O(\log \log n)$ overhead of the initial stage disappears into the constant. So our algorithm has the desirable property of degrading gracefully even when our assumption of an oblivious adversary is too strong.

While it is difficult to imagine an algorithm with significantly less than $O(\log \log n)$ step complexity, the question of whether a better algorithm is possible remains open. This is also the case for the adaptive adversary model, where there is no lower bound on expected step complexity to complement the $O(\log n)$ upper bounds of [1, 2].

The step complexity of our algorithm is $O(\log \log n)$ in expectation, and $O(\log n)$, with high probability. We notice that, even against a weak adversary, any step complexity upper bound on test-and-set that holds with high probability has to be at least $\Omega(\log n)$. This result follows from a lower bound of Attiya and Censor-Hillel on the complexity of randomized consensus [4].

Also of note, our algorithm suggests that non-determinism can be used to avoid some of the cost of expensive synchronization in shared memory, if the scheduler is oblivious. More precisely, Attiya *et al.* [5] recently showed that deterministic implementations of many shared objects, including test-and-set, queues, or sets, have worst-case executions which incur expensive read-after-write (RAW) or atomic-write-after-

read (AWAR) operation patterns. In particular, ensuring RAW order in shared memory requires introducing memory fences or barriers, which are considerably slower than regular instructions.

First, we notice that their technique also applies to randomized read-write algorithms against an adaptive adversary, yielding an adversarial strategy that forces *each* process to perform an expensive RAW operation pattern with probability 1. On the other hand, the sifting procedure of our algorithm bounds the number of processes that may perform RAW patterns in an execution to $O(\sqrt{n})$, with high probability. This shows that randomized algorithms can avoid part of the synchronization cost implied by the lower bound of [5], as long as the scheduler is oblivious.

Roadmap. We review the related work in Section 2, and precisely define the model of computation, problem statement, and complexity measures in Section 3. We then present our algorithm and prove it correct in Section 4. In Section 4.4, we present a simple technique for turning the single-shot test-and-set implementation into a multi-shot one, and derive lower bounds in Section 5. We summarize our results and give an overview of directions for future work in Section 6.

2 Related Work

The test-and-set instruction has been present in hardware for several decades, as a simple means of implementing mutual exclusion. Herlihy [11] showed that this object has consensus number 2.

Several references studied wait-free randomized implementations of test-and-set. References [10, 14] presented implementations with super-linear step complexity. (Randomized consensus algorithms also implement test-and-set, however their step complexity is at least linear [3].) The first randomized implementation with logarithmic step complexity was by Afek *et al.* [1], who extended the tournament tree idea of Peterson and Fischer [13], where the tree nodes are two-process test-and-set (consensus) implementations as presented by Tromp and Vitanyi [15]. Their construction has expected step complexity $O(\log n)$. This technique was made adaptive by the RatRace protocol of [2], whose step complexity is $O(\log^2 k)$ with probability $1 - 1/k^c$, for c constant, where k is the actual number of processes that participate in the execution. We use the RatRace protocol as the final part of our test-and-set construction. Note that these previous constructions assume a strong adaptive adversary. The approaches listed above for sublinear randomized test-and-set incur cost at least logarithmic in terms of expected time complexity, even if the adversary is oblivious, since they build on the tournament tree technique. References [7, 8] give deterministic test-and-set and compare-and-swap implementations with constant complexity in terms of *remote memory references* (RMRs), in an asynchronous shared-memory model with no process failures (by contrast, our implementation is wait-free). The general strategy behind their test-and-set implementation is similar to that of this paper and that of Afek *et al.* [1]: the algorithm runs a procedure to elect a leader process, i.e. the winner of the test-and-set, and then uses a separate flag register to ensure linearizability.

3 Preliminaries

Model. We assume an asynchronous shared memory model in which at most n processes may participate in any execution, $t < n$ of which may fail by crashing. We assume that processes know n (or a rough upper bound on n). Processes communicate through multiple-writer-multiple-reader atomic registers. Our algorithms are randomized, in that the processes' steps may depend on random local coin flips. Process crashes and scheduling are controlled by a weak *oblivious* adversary, i.e. an adversary that cannot observe the results of the random coin flips of the processes, and hence has to fix its schedule and failure pattern before the execution. On the other hand, we assume that the adversary knows the structure of the algorithm. By contrast, a strong *adaptive* adversary (as considered in Lemma 5) knows the results of the coin flips by the processes at any point during the algorithm, and may adjust the scheduling and failure pattern accordingly.

Problem Statement. The multi-use *test-and-set* bit has a test-and-set operation which atomically reads the bit and sets its value to 1, and a reset operation which sets the bit back to 0. We say that a process *wins* a test-and-set object if it reads 0 from the object; otherwise, if it reads 1, the process *loses* the test-and-set. By the sequential specification, each correct process eventually returns an indication (*termination*), and only one process may return winner from a single instance of test-and-set (the *unique winner* property). Also, no process may return loser before the winner started the execution, and only the winner of an instance may successfully reset the instance.

Complexity Measures. We measure complexity in terms of process steps: each shared-memory operation and (local) coin flip is counted as a step. The *total step complexity* counts the total number of process steps in an execution, while the *individual step complexity* (or simply *step complexity*) is the number of steps a single process may perform during an execution.

As a second measure, we also consider the number of read-after-write (RAW) patterns that our algorithm incurs. This metric has been recently analyzed by Attiya et al. [5] in conjunction with atomic write-after-read (AWAR) operations (we consider read-write algorithms, which cannot employ AWAR patterns). In brief, the RAW pattern consists of a process writing to a shared variable A, followed by the same process reading from a different shared variable B, without writing to B in between. Enforcing RAW order on modern architectures requires introducing memory fences or barriers, which are substantially slower than regular instructions. For a complete description of RAW/AWAR patterns, please see [5].

4 Test-and-Set Algorithm

In this section, we present and prove correct a randomized test-and-set algorithm, called Sift, with expected (individual) step complexity $O(\log \log n)$ and total step complexity $O(n)$. The algorithm is structured in two phases: a sifting phase, which eliminates a large fraction of the processes from contention, and a competition phase, in which the survivors compete in an adaptive test-and-set instance to assign a winner.

```
1  Shared:
2  Reg, a vector of atomic registers, of size n, initially ⊥
3  Resolved, an atomic register
4  procedure Test-and-Set()
5    if Resolved = true then
6      return loser
     /* sifting phase */
7    for round r from 0 to ⌈(5/2) ln ln n⌉ do
8      π_r ← n^{-(2/3)^r/2}
9      flip ← 1 with probability π_r, 0 otherwise
10     if flip = 1 then
11       Reg[r] ← p_i
12     else
13       val ← Reg[r]
14       if val ≠ ⊥ then
15         Resolved ← true
16         return loser
     /* competition phase */
17   result ← RatRace(p_i)
18   return result
```

Fig. 1. The Sift test-and-set algorithm.

4.1 Description

The pseudocode of the algorithm is presented in Figure 1. Processes share a vector Reg of atomic registers, initially $\perp$, and an atomic register $Resolved$, initially false. The algorithm proceeds in two phases.

The first, called the *sifting* phase, is a succession of rounds $r \geq 0$, with the property that in each round a fraction of the processes are eliminated. More precisely, in round r, each process flips a binary biased coin which is 1 with probability $\pi_r = n^{-(2/3)^r/2}$ and 0 otherwise (line 9). If the coin is 1, then the process writes its identifier p_i to the register $Reg[r]$ corresponding to this round, which is initially $\perp$. Every process that flipped 0 then reads the value of $Reg[r]$ in round r. If this value is $\perp$, then the process continues to the next round. Otherwise, the process returns loser, but first marks the *Resolved* bit to true, to ensure that no incoming process may win after a process has lost (lines 13-16). We will prove that by the end of the $\lceil \frac{5}{2} \ln \ln n \rceil + 1$ rounds in this phase, the number of processes that have not yet returned loser is $O(\log^7 n)$, with high probability.

In the *competition* phase, we run an instance of the RatRace [2] adaptive test-and-set protocol, to determine the one winner among the remaining processes. In brief, in RatRace, each process will first acquire a temporary name, and then compete in a series of two-process test-and-set instances to decide the winner. Since the number of processes that participate in this last phase is polylogarithmic, we will obtain that the number of steps a process takes in this phase is $O(\log \log n)$ in expectation.

4.2 Proof of Correctness

We first show that the algorithm is a correct test-and-set implementation.

Lemma 1 (Correctness). *The* Sift *algorithm is a linearizable test-and-set implementation.*

Proof. The *termination* property follows by the structure of the sifting phase, and by the correctness of the RatRace protocol [2]. The *unique winner* property also follows from the properties of RatRace. Also, notice that, by the structure of the protocol, any process that performs alone in an execution returns winner.

To prove linearizability, we first show that the algorithm successfully elects a leader, i.e., given an execution $\mathcal{E}$ of the protocol and a process p_ℓ that returns loser in $\mathcal{E}$, there exists a (unique) process $p_w \neq p_\ell$ such that p_w either returns winner in $\mathcal{E}$, or crashes in $\mathcal{E}$. Second, we show that, using the $Resolved$ bit, the test-and-set operation by process p_w can be linearized *before* p_ℓ's test-and-set operation.

For the first part, we start by considering the line in the algorithm where process p_ℓ returned loser. If p_ℓ returned on line 18, then the above claim follows by the linearizability of the RatRace test-and-set implementation [2]. On the other hand, if p_ℓ returned on line 16, it follows that p_ℓ has read the identifier of another process from a shared register $Reg[r]$ in some round $r \geq 1$. Denote this process by q_1. We now analyze the return value of process q_1 in execution $\mathcal{E}$. If q_1 crashed or returned winner in execution $\mathcal{E}$, then the claim holds, since we can linearize q_1's operation or crash before p_ℓ's operation. If q_1 returns loser from RatRace, then again we are done, by the linearizability of RatRace. Therefore, we are left with the case where q_1 returned loser on line 16, after reading the identifier of another process q_2 from a register $Reg[r']$ with $r' \geq r$. Notice that q_2 and p_ℓ are distinct, since the write operations are atomic.

Next, we can proceed inductively to obtain a maximal chain of *distinct* processes q_1, q_2, up to q_k for some $k \geq 1$ such that for any $1 \leq i \leq k-1$, process q_i read process q_{i+1}'s identifier and returned loser in line 16. This chain is of length at most $n-1$. Considering the last process q_k, since the chain is maximal, process q_k could not have read any other process's identifier in line 13 during the sifting phase. Therefore, process q_k either obtains a decision value from RatRace in line 18, or crashes in $\mathcal{E}$ after reading $Resolved =$ false in line 6 of the protocol. Notice that, since the $Resolved$ bit is atomic, q_k's test-and-set operation could not have started after p_ℓ's operation ended. Therefore, if q_k decides winner or crashes, then we can linearize its operation before p_ℓ's operation and we are done. Otherwise, if q_k decides loser from RatRace, then there exists another process p_w that either returns winner or crashes during RatRace, and whose RatRace(p_w) operation can be linearized before q_k's. Therefore, we can linearize p_w's test-and-set operation before p_ℓ's to finish the proof of this claim.

Based on this claim, we can linearize any execution in which some process returns loser as follows. We consider the losing processes in the order of their read operations on register $Resolved$. Let p_ℓ be the first losing process in this order. We then apply the claim above to obtain that there exists a process p_w that either crashes or returns winner, whose test-and-set operation can be linearized before that of p_ℓ. This defines a valid linearization order on the operations that return in execution $\mathcal{E}$. The other operations (by processes that crash, except p_w) may be linearized in the order or non-overlapping

operations. The remaining executions can be linearized trivially. We note that, since we use the $Resolved$ bit to ensure linearization, we avoid the linearizability issues recently pointed out by Golab et al. [9] for randomized implementations.

4.3 Proof of Performance

We now show that the algorithm has expected step complexity $O(\log \log n)$. The intuition behind the proof is that, for each round r, the number of processes that keep taking steps *after* round $r+1$ of the sifting phase is roughly $\sqrt{n_r}$ (in expectation), where n_r is the number of processes that take steps in round $r+1$. Iterating this for $\lceil (5/2) \ln \ln n \rceil$ rounds leaves at most $\operatorname{polylog} n$ active processes at the end of the sifting phase, with high probability. We begin the proof by showing that the sifting phase reduces the set of competitors as claimed.

Lemma 2. *With probability $1 - o(n^{-c})$, for any fixed c, at most $\ln^7 n$ processes leave the sifting phase without returning* loser.

Proof. Fix some c, and let $c' = c + 1$.

Let

$$\begin{aligned}\kappa &= -\frac{1}{\ln(2/3)} \left(1 - \frac{\ln 7 + \ln \ln \ln n}{\ln \ln n}\right). \\ &\leq -\frac{1}{\ln(2/3)} \\ &< \frac{5}{2}.\end{aligned}$$

We will show that, with high probability, it holds that for all $0 \leq r \leq \kappa \ln \ln n$, at most $n_r = n^{(2/3)^r}$ processes continue after r rounds of the sifting phase. The value of κ is chosen so that

$$\begin{aligned}n_{\kappa \ln \ln n} &= n^{(2/3)^{\kappa \ln \ln n}} \\ &= \exp\left(\ln n \cdot (2/3)^{\kappa \ln \ln n}\right) \\ &= \exp\left(\exp\left(\ln \ln n + \ln(2/3) \cdot \kappa \ln \ln n\right)\right) \\ &= \exp\left(\exp\left(\ln \ln n - \left(1 - \frac{\ln 7 + \ln \ln \ln n}{\ln \ln n}\right) \ln \ln n\right)\right) \\ &= \exp\left(\exp\left(\ln \ln n - \ln \ln n + \ln 7 + \ln \ln \ln n\right)\right) \\ &= \exp\left(7 \ln \ln n\right) \\ &= \ln^7 n.\end{aligned}$$

This bound is a compromise between wanting the number of processes leaving the sifting phase to be as small as possible, and needing the number of survivors at each stage to be large enough that we can characterize what happens to them using standard concentration bounds. Because $\kappa < \frac{5}{2}$, and extra rounds of sifting cannot increase the number of surviving processes, if there are at most $\ln^7 n$ survivors after $\kappa \ln \ln n + 1$

rounds, there will not be any more than this number of survivors when the sifting phase ends after $\left\lceil \frac{5}{2} \ln \ln n \right\rceil + 1$ rounds, establishing the Lemma.

We now turn to the proof of the n_r bound, which proceeds by induction on r: we prove that if fewer than n_r processes enter round $r+1$, it is likely that at most $n_{r+1} = n^{(2/3)^{r+1}}$ processes leave it. The base case is that at most $n_0 = n^{(2/3)^0} = n$ processes enter round 1.

Note that π_r, the probability of writing the register in round r, is chosen so that $\pi_r = n_r^{-1/2}$.

From examination of the code, there are two ways for a process to continue the execution after round $r+1$: by writing the register (with probability $\pi_r = n^{-(2/3)^r/2}$), or by reading $\perp$ from the register before any other process writes it. Suppose at most n_r processes enter round r. Then the expected number of writers is at most $n_r \pi_r = n_r^{1/2}$. On the other hand, since the adversary fixes the schedule in advance, the expected number of processes that read $\perp$ is given by a geometric distribution, and is at most $1/\pi_r = n_r^{1/2}$, giving an expected total of at most $2n_r^{1/2}$ processes entering round $r+1$. (The symmetry between the two cases explains the choice of π_r given n_r.) We now compute a high-probability bound for the number of surviving processes.

Let R count the number of processes that read $\perp$. For R to exceed some value m, the first m processes to access the register in round r must choose to read it, which occurs with probability $(1-\pi_r)^m \le e^{-m\pi_r}$. It follows that $\Pr\left[R \ge (c' \ln n) n_r^{1/2}\right] \le e^{-c' \ln n} = n^{-c'}$.

Let W be the number of processes that write the register in round $r+1$. Using standard Chernoff bounds (e.g., [12, Theorem 4.4]), we have $\Pr\left[W \ge (c \ln n) n_r^{1/2}\right] \le 2^{-(c' \ln n) n_r^{1/2}} \le n^{-c'}$, provided $c' \ln n \ge 6$, which holds easily for sufficiently large n.

Combining these bounds, with probability at least $1 - 2n^{-c}$ it holds that the number of survivors

$$\begin{aligned} W + R &\le 2(c' \ln n) n_r^{1/2} \\ &\le 2(c' \ln n) n^{(1/2)\cdot(2/3)^r} \\ &= \frac{2c' \ln n}{n^{(1/6)\cdot(2/3)^r}} \cdot n^{(2/3)^{r+1}} \\ &= \frac{2c' \ln n}{n_r^{1/6}} \cdot n_{r+1} \\ &\le \frac{2c' \ln n}{\ln^{7/6} n} \cdot n_{r+1} \\ &\le n_{r+1}, \end{aligned}$$

provided $\ln^{1/6} n \ge 2c'$, which holds for sufficiently large n.

The probability that the induction fails is bounded by $2n^{-c'} = 2n^{-c-1} = \frac{2}{n} n^{-c}$ per round; taking the union bound over $O(\log \log n)$ rounds gives the claimed probability $o(n^{-c})$.

To complete the proof of performance, first observe that the cost of the sifting phase is $O(\log \log n)$, by the above Lemma. The step complexity cost of a call to

RatRace [2] with $O(\log^7 n)$[3] other participants is $O(\log \log n)$, with probability at least $1 - (\log n)^{-c}$, and $O(\log n)$ otherwise. Choosing $c \geq 2$ gives an expected extra cost of the bad case of $o(1)$. Summing all these costs, we obtain a bound of $O(\log \log n)$ on the expected step complexity of Sift.

Since, with high probability, at most $\log^7 n$ processes participate in the RatRace instance, by the properties of RatRace, we obtain that the step complexity of the Sift algorithm is $O(\log n)$, with high probability.
We have therefore obtained the following bounds on the step complexity of the algorithm.

Lemma 3 (Step Complexity). *The* Sift *algorithm in Figure 1 runs in expected $O(\log \log n)$ steps, and in $O(\log n)$ steps, with high probability.*

We can extend this argument to obtain an $O(n)$ bound on the total number of steps that processes may take during a run of the algorithm.

Corollary 1 (Total Complexity). *The* Sift *algorithm has total step complexity $O(n)$, with high probability.*

Proof. The proof of Lemma 2 states that, with high probability, at most $n^{(2/3)^r}$ processes continue after sifting round r, for any $1 \leq r \leq \kappa \ln \ln n$. Let $\beta = \lfloor \kappa \ln \ln n \rfloor$.

It then follows that the total number of shared-memory operations performed by processes in the sifting phase is at most $\sum_{i=0}^{\beta} n^{(2/3)^r} + \sum_{i=\beta+1}^{\lceil (5/2) \ln \ln n \rceil} n^{(2/3)^\beta} \leq n + \sum_{i=1}^{(5/2) \ln \ln n} n^{2/3} \leq 2n$, with high probability. Since the total number of operations that $\ln^7 n$ participants may perform in RatRace is $O(\text{polylog } n)$, with high probability, the claim follows.

We notice that the processes that return loser without writing to any registers during an execution do not incur any read-after-write (RAW) cost in the execution. The above argument provides upper bounds for the number of such processes.

Corollary 2 (RAW Cost). *The* Sift *algorithm incurs $O(\sqrt{n})$ expected total RAW cost. With high probability, the algorithm incurs $O(n^{2/3} \log n)$ total RAW cost.*

Finally, we notice that the algorithm is correct even if the adversary is strong (the sifting phase may not eliminate any processes). Its expected step complexity in this case is the same as that of RatRace, i.e. $O(\log n)$.

Corollary 3 (Strong Adversary). *Against a strong adversary, the* Sift *algorithm is correct, and has expected step complexity $O(\log n)$.*

4.4 Multi-use Test-and-Set

We now present a transformation from single-use to multi-use test-and-set implementations. This scheme simplifies the technique presented in [1] for single-writer registers, in a setting where multi-writer registers are available. See Figure 2 for the pseudocode.

[3] Reference [2] presents an upper bound of $O(\log^2 k)$ on the step complexity of RatRace with k participants, with high probability. A straightforward refinement of the analysis improves this bound to $O(\log k)$, with high probability.

```
1  Shared:
2   T, a vector of linearizable single-use test-and-set objects
3   Index, an atomic register, initially 0
4  Local:
5   crtWinner, a local register, initially false
6  procedure Test-and-Set() /* at process p_i */
7     v ← Index.read()
8     res ← T[v].test-and-set()
9     if res ← winner then
10        crtWinner_i ← true
11    return res
12 procedure ReSet()
13    if crtWinner_i = true then
14        Index.write(Index.read() + 1)
15        crtWinner_i ← false
```

Fig. 2. The multi-use test-and-set algorithm.

Description. Processes share a list T of linearizable single-use test-and-set objects, and an atomic register $Index$. Each process maintains a local flag $crtWinner$ indicating whether it is the winner of of the current test-and-set instance. For the test-and-set operation, each process reads in variable v the $Index$ register and calls the single-use test-and-set instance $T[v]$. The process sets the variable $crtWinner$ if it is the current winner of $T[v]$, and returns the value received from $T[v]$. For the reset operation, the current winner increments the value of $Index$ and resets its $crtWinner$ local variable. (Recall that, by the specification of test-and-set [1], only the winner of an instance may reset it.)

The proof of correctness for the transformation is immediate. The complexity bounds are the same as those of the single-use implementation.

Improvements. The above scheme has the disadvantage that it may allocate an infinite number of single-use test-and-set instances. This can be avoided by allowing the current winner to de-allocate the current instance in the reset procedure, and adding version numbers to shared variables, so that processes executing a de-allocated instance automatically return loser.

5 Lower Bound

We first notice that any test-and-set algorithm against an oblivious adversary has executions with step complexity $\Omega(\log n)$, with probability at least $1/n^c$, for a small constant c. In essence, this implies that any bound $O(C)$ on the step complexity of test-and-set that holds with high probability has to have $C \in \Omega(\log n)$. This bound is a corollary of a result by Attiya and Censor-Hillel [4] on the complexity of randomized consensus against a weak adversary, and is matched by our algorithm.

Lemma 4. *For any read-write test-and-set algorithm A that ensures agreement, there exists a weak adversary such that the probability that A does not terminate after $\log n$ steps is at least $1/n^{\gamma}$, for a small constant γ.*

Proof. Consider a test-and-set algorithm A. Then, in every execution in which only two processes participate, the algorithm can be used to solve consensus. We now employ Theorem 5.3 of [4], which states that, for any consensus algorithm for n processes and f failures, there is a weak adversary and an initial configuration, such that the probability that A does not terminate after $k(n-f)$ steps is at least $(1-c^k\epsilon_k)/c^k$, where c is a constant, and ϵ_k is a bound on the probability for disagreement.

We fix parameters $k = \log n$, $\epsilon_k = 0$ (since the test-and-set algorithm guarantees agreement), $n = 2$, and $f = 1$. The claim follows, with the parameter $\gamma = \log c$.

Strong Adversary RAW Bound. We now apply the lower bound of Attiya *et al.* [5] on the necessity of expensive read-after-write (RAW) patterns in deterministic object implementations, to the case of randomized read-write algorithms against a strong adversary. We state our lower bound in terms of total RAW cost for a test-and-set object; since test-and-set can be trivially implemented using stronger objects such as consensus, adaptive strong renaming, fetch-and-increment, initialized queues and stacks, this bound applies to randomized implementations of these objects as well.

Lemma 5 (Strong Adversary RAW Bound). *Any read-write test-and-set algorithm that ensures safety and terminates with probability α against a* strong adaptive *adversary has worst-case expected total RAW complexity $\Omega(\alpha n)$.*

Proof (Sketch). Let A be a read-write test-and-set algorithm. We describe an adversarial strategy $S(A)$ that forces each process to perform a read-after-write (RAW) operation in all terminating executions. The adversary schedules process p_1 until it has its first write operation enabled. (Each process has to eventually write in a solo execution, since otherwise we can construct an execution with two winners.) Similarly, it proceeds to schedule each process $p_2, \ldots, p_n$ until each has its first write operation enabled (note that no process has read any register that another process wrote at this point). Let $R_1, R_2, \ldots, R_n$ be the registers that $p_1, p_2, \ldots, p_n$ write to, respectively (these registers are not necessarily distinct).

The adversary then schedules process p_n to write to register R_n. It then schedules process p_n until p_n reads from a register that it did not last write to. This has to occur, since otherwise there exists an execution $\mathcal{E}'$ in which process p_n takes the same steps, and a process $q \neq p_n$ is scheduled solo until completion, right before p_n writes to R_n. Process q has to decide winner in this parallel execution. On the other hand, process p_n cannot distinguish execution $\mathcal{E}'$ from a solo execution, therefore decides winner in $\mathcal{E}'$, violating the *unique winner* property. Since the algorithm guarantees safety in all executions, process p_n has to read from a register it did not last write to, and therefore incurs a RAW in execution $\mathcal{E}$.

Similarly, after process p_n performs its RAW, the adversary schedules process p_{n-1} to perform its write operation. By a similar argument to the one above, if the adversary schedules p_{n-1} until completion, p_{n-1} has to read from a register that it did not write to,

and incur a RAW. We proceed identically for processes $p_{n-2}, \ldots, p_1$ to obtain that each process incurred a RAW in execution $\mathcal{E}$ dictated by the strong adversarial scheduler, which concludes this sketch of proof.

Discussion. This linear bound applies to randomized algorithms against a strong adversary. Since, by Corollary 2, the Sift test-and-set algorithm has expected RAW cost $O(\sqrt{n})$ against a weak adversary, these two results suggest that randomization can help reduce some of the inherent RAW cost of implementing shared objects, if the scheduler is assumed to be oblivious.

6 Summary and Future Work

In this paper, we present a linearizable implementation of randomized test-and-set, with $O(\log \log n)$ individual step complexity and $O(n)$ total step complexity, against a weak oblivious adversary. Our algorithm shows an exponential improvement over previous solutions, that considered a strong adaptive adversary. Also, it has the interesting property that its performance degrades gracefully if the adversary is adaptive, as the algorithm is still correct in this case, and has step complexity is $O(\log n)$.

Lower bounds represent one immediate direction of future work. In particular, it is not clear whether better algorithms may exist, either for the oblivious adversary, or for the adaptive one. Also, another direction would be to see whether our sifting technique may be applied in the context of other distributed problems, such as mutual exclusion or cooperative collect.

Acknowledgements. The authors would like to thank Keren Censor-Hillel for useful discussions, and the anonymous reviewers for their feedback.

References

1. Yehuda Afek, Eli Gafni, John Tromp, and Paul M. B. Vitányi. Wait-free test-and-set (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 85–94, 1992.
2. Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast randomized test-and-set and renaming. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 94–108, Berlin, Heidelberg, 2010. Springer-Verlag.
3. Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):1–26, 2008.
4. Hagit Attiya and Keren Censor-Hillel. Lower bounds for randomized consensus under a weak adversary. *SIAM J. Comput.*, 39(8):3885–3904, 2010.
5. Hagit Attiya, Rachid Guerraoui, Danny Hendler, Petr Kuznetsov, Maged M. Michael, and Martin T. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *POPL*, pages 487–498, 2011.
6. Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *PODC '97: Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, New York, NY, USA, 1997. ACM.

7. Wojciech M. Golab, Vassos Hadzilacos, Danny Hendler, and Philipp Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In *PODC*, pages 3–12, 2007.
8. Wojciech M. Golab, Danny Hendler, and Philipp Woelfel. An o(1) rmrs leader election algorithm. *SIAM J. Comput.*, 39(7):2726–2760, 2010.
9. Wojciech M. Golab, Lisa Higham, and Philipp Woelfel. Linearizable implementations do not suffice for randomized distributed computation. In *STOC*, pages 373–382, 2011.
10. Maurice Herlihy. Randomized wait-free concurrent objects (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, PODC '91, pages 11–21, New York, NY, USA, 1991. ACM.
11. Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.
12. Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.
13. Gary L. Peterson and Michael J. Fischer. Economical solutions for the critical section problem in a distributed system (extended abstract). In *Proceedings of the ninth annual ACM symposium on Theory of computing*, STOC '77, pages 91–97, New York, NY, USA, 1977. ACM.
14. Serge Plotkin. Chapter 4: Sticky bits and universality of consensus. *Ph.D. Thesis, MIT*, 1998.
15. John Tromp and Paul Vitányi. Randomized two-process wait-free test-and-set. *Distrib. Comput.*, 15(3):127–135, 2002.