Concurrent use of write-once memory

James Aspnes[*] Keren Censor-Hillel[†] Eitan Yaakobi[‡]

June 23, 2016

Abstract

We consider the problem of implementing general shared-memory objects on top of write-once bits, which can be changed from 0 to 1 but not back again. In a sequential setting, write-once memory (WOM) codes have been developed that allow simulating memory that support multiple writes, even of large values, setting an average of $1 + o(1)$ write-once bits per write. We show that similar space efficiencies can be obtained in a concurrent setting, though at the cost of high time complexity and fixed bound on the number of write operations. As an alternative, we give an implementation that permits unboundedly many writes and has much better amortized time complexity, but at the cost of unbounded space complexity. Whether one can obtain both low time complexity and low space complexity in the same implementation remains open.

[*]Yale University, Department of Computer Science, aspnes@cs.yale.edu.

[†]Technion, Department of Computer Science, ckeren@cs.technion.ac.il. Supported in part by the Israel Science Foundation (grant 1696/14).

[‡]Technion, Department of Computer Science, yaakobi@cs.technion.ac.il.

1 Introduction

Write-once memory (WOM) is a storage medium with memory elements, called cells, that can only *increase* their value. These media can be represented as a collection of binary cells, each of which initially represents a bit value 0 that can be *irreversibly* overwritten with a bit value 1. WOM codes, first introduced by Rivest and Shamir [33], enable to record data multiple times without violating the asymmetry writing constraint in a WOM. The goal in the design of a WOM code is to maximize the total number of bits that can be written to the memory in t writes, while preserving the property that cells can only increase their level.

These codes were first motivated by storage media such as punch cards and optical storage. However, in the last decade, a wide study of these codes re-emerged due to their connection to *Flash memories*. Flash memories contain floating gate cells which are electrically charged with electrons to represent the cell level. While it is fast and simple to increase a cell level, reducing its level requires a long and cumbersome operation of first erasing its entire containing block and only then programming relevant cells. Applying a WOM code enables additional writes before having to physically erase the entire block.

This paper provides the first study of concurrency in write-once shared memory. We investigate concurrent write-once memory from a theoretical viewpoint, which, in particular, means that we consider the memory impossible to erase (as opposed to considering it to be expensive). We show that any problem that can be solved in a standard shared-memory model can be solved in a write-once memory model, at the cost of some overhead. Our goal is to provide an analysis of this cost, both in terms of step complexity and space complexity.

Motivation: In addition to our interest in WOM as a computing model, our study is motivated by two observations. First, WOM is not subject to the **ABA problem**, in which memory can change back and forth going unnoticed, which is proven to be hard to overcome [1].

The second reason is that several known concurrent algorithms are already implemented using write-once bits. In other words, for some specific problems, the overhead of using WOM can be reduced compared to the general case. Examples of such implementations are the **sifters** constructed by Alistarh and Aspnes [2] and by Giakkoupis and Woelfel [17], and some variants of the **conflict detectors**, of Aspnes and Ellen [5].[1] A **max register** [3] is another example of an object that can be implemented using write-once bits (see overview in Section 5). Interestingly, the covering arguments used to prove lower bounds on max registers [3] imply that no historyless primitive can give a better implementation than write-once bits.

Yet these specific solutions do not immediately give a general implementation of arbitrary shared-memory objects, and the question arises whether the space efficiencies obtained by WOM codes in a sequential setting can transfer to a concurrent setting as well.

The challenge: To give a flavor of the challenge in adopting known WOM codes to concurrent use, we explain a simple example in Table 1, introduced by Rivest and Shamir [33], which enables the recording of two bits of information in three cells twice. It is possible to verify that after the first 2-bit data vector is encoded into a 3-bit codeword, if the second 2-bit data vector is different from the first, the 3-bit codeword into which it is encoded does not change any code bit 1 into a code bit 0, ensuring that it can be recorded in the write-once medium.

[1] This does not include the $\Theta(\log m/\log\log m)$-step m-valued conflict detector that appears in [5], but does include a simpler $\Theta(\log m)$-step conflict detector in which a write of a value whose bits are $x_{k-1},\ldots,x_0$ is done by setting to 1 the corresponding bits $A[i][x_i]$ in a $k \times 2$ array A.

Data bits	First write	Second write
00	000	111
10	100	011
01	010	101
11	001	110

Table 1: A WOM code Example

Suppose now that the above code is used in a concurrent WOM system, and that two processes p_1 and p_2 invoke write operations with input data bits 10 and 01, respectively. This means that p_1 needs to write 100 into the memory, and p_2 needs to write 010 to it. In other words, p_1 needs to set the first of the three bits to 1, while p_2 needs to set the second. Consider a schedule in which p_1, p_2 set their respective bit in some order, and afterwards another process p_3 reads the shared memory. The bits that it sees are 110, but these correspond to the input 11, which was never written into the memory, violating the specification of the memory.

The difficulty above is amplified by the fact that since more than a single process is writing and reading the content of the memory, it is not known what the value of t is, that is, how many writes have occurred so far. This is needed in the above example for both writing and reading.

We emphasize that there is a significant amount of fundamental simulations of different types of registers in the literature of distributed computing (see, e.g., [7, Chpater 10] and [21, Chapter 4]). The above WOM example satisfies the definition of a **single-writer-multi-reader (SWMR) safe register** [28,29], in which a read that is not concurrent with a write returns a correct value. Known simulations can use this object to construct **multi-writer-multi-reader (MWMR) atomic registers**. However, these simulations do not comply with the restrictions that arise from WOM, and hence different solutions must be sought.

1.1 Our contribution

We first show that with one additional bit that indicates to read operations that a write operation has been completed, we can easily implement a write-once m-bit register. Then, we show how to support t writes, still for a single writer, within a space complexity of $2m + t$ bits. After these toy examples, our goal is to get closer to the $t(1 + o(1))$-space WOM code constructions for the non-concurrent setting. Carefully adapting the tabular code of [33] to our concurrent setting, allows us to obtain a SWMR m-bit register that supports t writes, with the following properties.

Theorem 4.1 *There is an algorithm that implements an n-process SWMR m-bit register supporting up to t writes, using space complexity of $(1+o(1))t$ when $t = \omega(m2^m)$, and with amortized step complexity $O(n2^m)$ for a write and $O(2^m)$ for a read.*

We then extend our tabular construction to support multiple writers, with the aid of a reduction from MWMR registers to SWMR registers due to [24], and with incorporating safe-agreement objects [10] in order to efficiently share space. Our result is summarized as follows.

Theorem 4.2 *There is an algorithm that implements an n-process MWMR m-bit register that supports up to t writes, using space complexity of $(2 + o(1))t$ when $t = \omega((m + \log n)n^6 2^m)$, and with amortized step complexity $O(n^2 2^m)$ for both write and read operations.*

The drawback of the above implementation is its large step complexity. At the cost of increased space complexity, we show how to build a WOM code on top of a max register, which allows drastically reduced step complexities, as stated next. Here, a unique timestamp t is guaranteed to be associated by the algorithm with each write operation.

Theorem 5.1 *There is an algorithm that implements an n-process MWMR register of m bits with unbounded space, where the* amortized *step complexity of a write operation that gets associated with a timestamp t is $O(\log t + m + \log n)$ and the step complexity of a read operation that reads a value associated with a timestamp t is $O(\log t + m + \log n)$.*

Whether it is possible to obtain both low time complexity and low space complexity in the same implementation remains an intriguing open question.

1.2 Additional related work

In their pioneering work, Rivest and Shamir also reported on more WOM code constructions, including tabular WOM codes and linear WOM codes. Since then, several more constructions were studied in the 1980's and 1990's [13, 15, 18], and more interest to these codes was given in the past seven years; see e.g. [9, 11, 12, 14, 34, 35, 37–40].

The capacity of a WOM was also rigorously investigated. The maximum sum-rate as well as the capacity regions were studied in [19, 33, 36] with extensions to the non-binary case in [16]. The implementation of WOM codes in several applications such flash memories and phase-change memories was recently explored in [26, 30, 31, 41, 42]. These works were motivated by the system implementation on WOM codes in these memories, while taking into account the hardware and architecture limitations when implementing these codes into the system.

Write-once memory should not be confused with **sticky registers** as defined by Plotkin [32], which in some recent systems literature (e.g. [8]) have been described as registers with **write-once semantics**. Sticky registers initially hold a default "empty" value, and any write after the first has no effect. Such registers are equivalent to consensus objects, and thus significantly more powerful than standard shared memory. In contrast, write-once memory as considered here and in the WOM code literature is weaker than standard shared memory.

1.3 Model

We use a standard asynchronous shared memory system, restricted by the assumption that registers hold only a single bit and `write` operations can only write 1. We assume that all registers are initially 0, as any register initialized to 1 conveys no information and can safely be omitted. Asynchrony is modeled by interleaving according to a schedule chosen by an adversary. As we consider only deterministic algorithms, it is reasonable to assume that the adversary has unrestricted knowledge of the state of the system at all times, and can choose the schedule to make things as difficult for the algorithm as possible. The computational power of the adversary is unlimited; indeed, the adversary is essentially just a personification of the universal quantifier applied to schedules.

When implementing an object, our goal is **linearizability** [22]; given an execution of S of the implemented object, there should exist a sequential execution S' of the same object with the same operations, such that whenever some operation π finishes in S before another operation π' starts, π precedes π' in S'.

Time complexity: We use the standard notion of **step complexity**. The worst-case **individual step complexity** of an operation is the maximum number of **steps** (read or write operations applied to a write-once bit) carried out by the process executing the operation between when it starts and finishes. The **total step complexity** of a collection of operations is the maximum number of steps taken by all processes in any execution involving these operations.

We say that an operation π has **amortized step complexity** $C(\pi)$ if, in any execution, the total step complexity is bounded by the sum of the amortized step complexities of all operations in the execution. Note that the amortized step complexity of an operation is not uniquely determined, and there may be more than one way to trade off the amortized step complexities of operations.

Space complexity and storage: The straightforward measure of **space complexity** for write-once memory is the number of objects (in our case, write-once bits) that can be accessed during the execution of an algorithm or implementation. In traditional shared-memory models, this quantity is fixed throughout the execution of the algorithm.

However, for one of our implementations, we will assume that the space is unbounded, in order to exemplify its property of obtaining good step complexity despite supporting an unbounded number of writes. A simple argument (see also Section 6) shows that infinite space is inherent for supporting an unbounded number of writes in a write-once medium.

2 From write-once bits to write-once registers

We begin with two basic constructions of write-once single-writer registers from write-once bits.

2.1 A write-once single-writer register

A write-once single-writer register allows a single fixed writer process to write a value at most once during an execution. Initially, it holds a special empty state $\perp$, which is replaced by a value on the first write. The **width** m of the register is the size of the value written in bits. The effect of multiple writes is unspecified, although the writer process can simply choose to discard any writes after the first.

To implement a width-m write-once single-writer register for larger m, we use m write-once bits $r[0] \ldots r[m-1]$ to hold the contents of the register, plus an extra Done bit to mark the register as written. The writer fills in the content bits first, then sets Done; a reader looks at Done first, and returns $\perp$ if it is 0; if not, it reads and returns the values of the contents bits. This gives a linearizable implementation, where the high-level operations are linearized according to the order in which they access the Done bit; each `read` operation then returns either $\perp$ or the value of the unique `write` depending on whether it reads Done before or after the `write` sets it.

The reason we must have the Done bit is because otherwise the reader might observe an intermediate value which is not the input of the writer but rather only has a common prefix with it.

Note that if `write` is called more than once then the behavior of the register is unspecified.

Theorem 2.1. *The register implemented in Algorithm 1 is linearizable in any execution where* `write` *is called at most once.*

Proof. Given an execution, linearize all operations in the order in which they access the Done bit; this is trivially consistent with the observable execution order. If there is no `write` operation, all `read` operations return $\perp$, consistent with a sequential execution. If there is a `write` operation, then any `read` operation that observes 0 in Done linearizes before the `write` and returns $\perp$, while any `read` operation that observes 1 in Done linearizes after the `write` and returns the contents of r, which will be equal to the value left in r by the writer, because no operation updates r after the Done bit is set. So these return values are also consistent with the sequential execution. ☐

```
procedure write(r, x)
    for i ← 0 ... m − 1 do
        r[i] ← x[i] ;
    Done ← 1 ;
procedure read(r)
    if Done = 1 then
        for i ← 0 ... m − 1 do
            x[i] ← r[i] ;
        return x ;
    else
        return ⊥;
```

Algorithm 1: Implementation of an m-bit single-writer write-once register

2.2 An erasable write-once single-writer register

It is easy to extend the construction of §2.1 to a write-once single-writer register that is also *erasable*. This means that the register can be in one of three states: its initial state $\bot$, a state in which it holds a m-bit value, and an erased state $\top$. The register may be in the erased state after it was in a state in which it holds a m-bit value, but it cannot change its state after entering the erased state.

Notice that having an erased state does not allow us to re-use the register because otherwise, the reader may start reading the first bits of one m-bit value written and continue reading the last bits of a different m-bit written value, obtaining an invalid return value that was never written.

To implement the erasable register, we add the procedure `erase` which simply sets another shared bit Erased, and we modify the `read` procedure to check the Erased bit, as shown in Algorithm 2.

```
procedure erase(r)
    Erased ← 1 ;
procedure read(r)
    if Erased = 1 then
        return ⊤;
    if Done = 1 then
        for i ← 0 ... m − 1 do
            x[i] ← r[i] ;
        return x ;
    else
        return ⊥;
```

Algorithm 2: Implementation of an erasable m-bit single-writer write-once register

Theorem 2.2. *The register implemented in Algorithm 2 is linearizable in any execution where* `write` *is called at most once and not after* `erase` *is called.*

Proof. Given an execution, we linearize any `write` operation and all `read` operations that read 0 from `erase` before any `erase` operation and any `read` operation that reads 1 from `erase`.

Within the first set of operations, we linearize all operations in the order in which they access the Done bit, which by the proof of Theorem 2.1 is consistent with the observable execution order, as well as with a sequential execution.

Within the second set of operations, we linearize all `read` operations after the `erase` operation, where the latter must be invoked in order for any `read` operation to read 1 from Erased. This is trivially consistent with both the observable execution order and the sequential execution, because all such `read` operations return $\top$. □

3 Atomic multi-bit writes

The constructions of the preceding section allow only a single write operation. WOM codes generally support up to t write operations; however, because write-once bits are written one at a time, a reader might be confused by seeing an incomplete code word. In Algorithm 3, we show how to implement an atomic multi-bit write operation that can be used up to t times on an m-bit write-once memory by a single writer process, at a cost of increasing the memory size to $2m + t$. This allows any WOM code to be used in a SWMR register if we are willing to accept doubling the space requirement.

Each bit of the original register is implemented by two bits. A 00 pattern represents a 0 bit. The 01 and 10 patterns are used to represent 1 bits for writes that are in progress. These are eventually converted into 11 pattern to represent a 1 bit for a completed write.

A second array Done of t bits counts, in unary, the number of completed writes. Any 01 patterns are interpreted as 1 only when the number of 1 bits in Done is odd; similarly, any 10 patterns are interpreted as 1 only when the number of 1 bits in Done is even. This allows all new 1 bits corresponding to a write in progress to switch from being ignored by the reader to being included atomically when the next bit in Done is set. To allow subsequent write operations to do the same thing, the writer cleans up these partial bits by rewriting them as 11 on its way out.

It is still important for the reader to get a consistent snapshot of both arrays. We do this using a standard double collect snapshot that returns only when two consecutive reads of the memory return the same values. Because the values in the write-once memory increase monotonically, no additional tagging is needed to avoid ABA issues.

Theorem 3.1. *Assuming only one process ever invokes* `write` *operations, Algorithm 3 implements an atomic multi-bit write.*

Proof. Because there is only one writer, we can characterize the state of the A and Done arrays exactly throughout the execution of the algorithm. Our induction hypothesis is that at the start of the r-th iteration, every entry in A is either 00 or 11, and Done contains $r - 1$ ones in positions 1 through $r - 1$. During the execution of `writeLocations` in round r, the writer will add 01 or 10 into the given locations, and only set the Done[r] bit when all these values are written. The second loop, converting all the intermediate values to 11, ensures that the induction hypothesis continues to hold at the end of this write. We linearize the `writeLocations` operation at the time it writes Done[r].

The double-collect snapshot ensures that each reader's copies MyA and MyDone of A and Done are equal to the value of A and Done at some time during its execution. We use this time as the read operation's linearization point. There are now several cases to consider for how this read interacts with a possible concurrent write:

1. If there is no concurrent write, then output[i] $= 1$ for precisely those locations that hold 11 and 0 for those that hold 00 (there are no other possible values). These correspond to the value provided in the arguments to preceding `writeLocations` operations.

```
procedure write(v)
    Let i_1, ..., i_k be the locations of the original WOM code that need to be set when
    writing the value v;
    writeLocations(i_1, ..., i_k);
procedure writeLocations(i_1, ..., i_k)
    r ← r + 1;
    for j ← 1 ... k do
        if r is odd then
            A[i_j] ← 01;
        else
            A[i_j] ← 10;
    Done[r] ← 1;
    for j ← 1 ... k do
        A[i_j] = 11;
procedure read()
    repeat
        MyA ← A;
        MyDone ← Done;
    until two consecutive iterations produce the same values;
    Let r be the largest index in MyDone such that MyDone[r] = 1;
    foreach i in the index set of MyA do
        if MyA[i] = 11 then
            output[i] ← 1;
        else if MyA[i] = 01 and r is odd then
            output[i] ← 1;
        else if MyA[i] = 10 and r is even then
            output[i] ← 1;
        else
            output[i] ← 0;
    return output;
```

Algorithm 3: Implementing an arbitrary WOM code using an atomic multi-bit write

2. If there is a concurrent write, and it has not yet set Done$[r]$ at the time the `read` operation reads it, then the `read` will set its own r to $r-1$. This will have the opposite parity of the 01 or 10 values written by the concurrent write, so these values will be treated as 0. It follows that the read will return whatever value was present at the start of the concurrent write, which is consistent with the linearization ordering.

3. If there is a concurrent write, and it has set Done$[r]$ before the `read` operation reads it, then any 01 or 10 values that have not yet be cleaned up will be interpreted by the reader as ones. In this case the reader will return the value present following the concurrent write, which is again consistent with the linearization ordering.

□

4 Registers based on the tabular WOM code

Here we give a family of register implementations based on the tabular WOM code of Rivest *et al.* [33]. These allow up to t writes of m-bit values. For the single-writer case (see §4.2), the construction requires only $(1 + o(1))t$ write-once bits provided $t = \omega(m2^m)$, for an average of $1 + o(1)$ bits per write. For the multi-writer case (§4.3), it requires $(2 + o(2))t$ bits under the same conditions on t. In both cases the amortized time complexity of each operation is polynomial in n and 2^m, even for very large tables. An alternative implementation that sacrifices space for speed will be given later in §5.

4.1 The tabular WOM code

The tabular WOM code represents 2^m distinct values as an array of k rows of $m+\ell$ bits each, where k and ℓ are parameters selected to maximize the efficiency of the code. Each row $A[i]$ consists of an m-bit increment field $A[i].\mathsf{increment}$, interpreted as an element of $\mathbb{Z}_{2^m}$, together with an ℓ-bit unary counter $A[i].\mathsf{count}$. A row is **unused** if all bits in the counter are 0, and **full** if all are 1. A row that is neither unused nor full is **active**. The value stored in the array is given by

$$\left(\sum_{i=1}^{k} A[i].\mathsf{increment} \cdot A[i].\mathsf{count}\right) \bmod 2^m. \tag{1}$$

To change the current value in A from x to y, the writer first checks for a used, non-full row that already has an increment value equal to $(y - x) \bmod 2^m$, and if so increments the counter in that row by one by writing an additional 1 bit. If there is no such row, the writer selects an unused row, writes $(y - x) \bmod 2^m$ to its increment field, and sets the count to 1 by writing a single one bit to the counter field. This process continues until the writer can no longer find an unused row when trying to write an increment that cannot be stored otherwise.

We would like to get the space needed for t write operations as close to t as possible. There are two sources of space overhead that prevent this in the tabular WOM code. The first is that each $\mathsf{increment}$ field adds m bits that must be amortized over the ℓ write operations handled by that row; this gives $1 + o(1)$ overhead provided $\ell = \omega(m)$. The second is that up to $2^m - 1$ rows may be only partially used (if more than this are unused, we have rows available for all possible increments and can perform any new write operation). This overhead also becomes $1 + o(1)$ provided $k = \omega(2^m)$. Setting both $\ell = \omega(m)$ and $k = \omega(2^m)$ gives $t = \omega(m2^m)$ and a space complexity of $(1 + o(1))t$.

4.2 Single-writer implementation

The tabular WOM code has the useful property that as long as the writer writes $A[i].\mathsf{increment}$ in a new row before setting any of the bits in $A[i].\mathsf{count}$, the value stored in A changes atomically at the moment that the writer sets a bit in $A[i].\mathsf{count}$. This means that with a single writer, no special effort is needed to ensure linearizability, and we can treat the linearization point of a write operation as the moment it sets a bit in some count row.

On the other side, a read operation needs to obtain an atomic snapshot of the entire array to be able to compute the sum of the entries as given in (1). This can be done in a straightforward way using a double-collect snapshot, with some further optimizations possible by taking advantage of predicting which bits could be written next. Note that even with a snapshot, it is possible that a reader may observe an incomplete write of $A[i].\mathsf{increment}$ for some i. However, this can only occur if the corresponding $A[i].\mathsf{count}$ is still 0. So a read operation always returns the sum of the increments of all writes that linearize before it, giving correctness.

For $t = \omega(m2^m)$, the average time complexity of a write is $1 + o(1)$, though the cost of a specific write may range from 1 to $1 + m$, depending on whether it needs to set an increment field.

For read operations the cost may be much higher. Unlike the writer, a reader may need to read the same bit more than once to see if it has changed. Indeed, a naive implementation of the double-collect snapshot would force a reader to read all $k\,(m + \ell)$ bits at least twice during any read operation, and again for each write that occurs during the read. We can reduce the amortized cost by observing that the reader never needs to re-read a bit that is already 1, and by enforcing that the writer use new rows and write count bits in a specified order. This means that each new write might write to at most 2^m distinct locations in the count fields: one for each active row, plus at most one bit at the start of an unused row if the active rows do not span all 2^m possible increments. This reduces the cost imposed on each reader by a new write to at most $2^m + m$ operations (2^m count bits plus at most one increment field). If we multiply this by n potential readers, this raises the amortized cost of a write to $O(n2^m)$ bit operations, which is large but still independent of the table size. Shifting costs to the writers in this way still leaves the reader with an amortized cost of $O(2^m)$ to re-read zero bits to confirm that no new writes have occurred.

Pseudocode for an implementation that applies these optimizations is given in Algorithm 4. The above discussion essentially proves the following.

Theorem 4.1. *There is an algorithm that implements an n-process SWMR m-bit register supporting up to t writes, using space complexity of $(1 + o(1))t$ when $t = \omega(m2^m)$, and with amortized step complexity $O(n2^m)$ for a write and $O(2^m)$ for a read.*

4.3 Multi-writer extension

We can extend the single-writer construction to multiple writers using a construction of Israeli and Shaham [25, §4]. This construction implements a multi-writer multi-reader (MWMR) register from n single-writer multi-reader (SWMR) registers, one for each writer. Each MWMR write operation requires $O(n)$ SWMR read operations and 2 SWMR write operations. MWMR read operations require only $O(n)$ SWMR read operations. Each SWMR register must be large enough to store the contents of the MWMR register, plus an addition $6 \lg n + O(1)$ bits for pointers used to determine the linearization order.

By implementing each SWMR register as in the preceding section, for sufficiently large t, each writer process can carry out up to t writes at an amortized space complexity of $2 + o(1)$ bits per write. However, both the bound on t to obtain this space complexity and the time complexity of both read and write operations becomes quite large: t must be $\omega((m + \log n)n^6 2^m)$ and the amortized cost of both read and write operations rises to $O(n^2 2^m)$. Whether one can retain low per-write space complexity while getting low time complexity in a MWMR setting remains open.

A further annoyance is that the low amortized space complexity applies only when each writer individually uses up its allotment of $t = \omega((m+\log n)n^6 2^m)$ writes. While this might be a reasonable assumption for some applications, in the worst case we can imagine a single writer using up its allotment while the other writers do nothing, giving a per-write space complexity of $\Theta(n)$.

4.4 Allocating table rows from a common pool

We solve this problem by allocating table rows from a common pool. In this section we describe a simple storage allocator, based on the safe-agreement objects of Borowsky *et al.* [10]. Our storage allocator guarantees that all but $n - 1$ rows in a k-row array are assigned to some writer.

A **safe-agreement object** provides a weak version of consensus that guarantees agreement and validity but not termination. Any process that accesses a safe-agreement object is guaranteed

```
shared data: Array A[0..r − 1] of rows, where each row A[i] has fields A[i].increment of m
             write-once bits and A[i].count[0..ℓ − 1] of ℓ write-once bits;
local data: Array next[0..2^m − 1] where each entry holds either ⊥ or an ⟨index, position⟩
            where index is an index into A and position is an index into A[index].count;
current, equal to the most recently computed value of the register;
Array MyA[0..r − 1] of rows, where each row MyA[i] has fields MyA[i].increment of m
write-once bits and MyA[i].count[0..ℓ − 1] of ℓ write-once bits;
procedure write(v)
    Let i = v − current (mod 2^m);
    if next[i] = ⊥ then
        next[i] ← ⟨r, 0⟩ where r is a newly-allocated row;
        A[next[i].index].increment ← i;
    A[next[i].index].count[next[i].position] ← 1;
    if next[i].position = ℓ − 1 then
        next[i] ← ⊥;
    else
        next[i].position = next[i].position + 1;
    current ← v;
procedure read()
    repeat
        foreach i such that MyA[i].count is not all 0 or all 1 do
            copy(MyA[i].count, A[i].count);
        Let i be the smallest index such that MyA[i].count is all 0;
        if A[i].count[0] ≠ 0 then
            MyA[i].increment ← A[i].increment;
            copy(MyA[i].count, A[i].count);
    until MyA is unchanged throughout an iteration;
    return Σ_{i=0}^{r−1} (MyA[i].increment · Σ_{j=0}^{ℓ−1} MyA[i].count[j]) (mod 2^m) ;
// Helper procedure for read
// Copies bits to X from Y assuming Y contains no 0 to the left of a 1
procedure copy(X, Y)
    Let j be the smallest index such that X[j] = 0;
    while j < ℓ ∧ Y[j] = 1 do
        X[j] ← 1;
        j ← j + 1;
```

Algorithm 4: Single-writer register implemented using a tabular WOM code

to obtain the id of a unique winner among the users of the object, provided no process halts during a special *unsafe* segment of its execution; if some process does halt, the object never returns. This means that, if we assign a safe-agreement object to control ownership of each of the k rows in our pool, at most $n-1$ rows will never be allocated, assuming at least one process continues to run.

Algorithm 5 shows how to implement a safe-agreement object using WOM. The mechanism is essentially the same as in the original Borowksy *et al.* algorithm, except that we encode the values 0 as 000 when it represents the initial value and 011 when it represents the result of a back-off, the value 1 as 001, and the value 2 as 101. The intuition is that a process first advances to level 1

```
// propose_i(v)
A[i] ← 001;
if snapshot(A) contains 101 for some j ≠ i then
    // Back off
    A[i] ← 011;
else
    // Advance
    A[i] ← 101;
// safe_i
repeat
    s ← snapshot(A);
until s[j] does not equal 001 for any j;
// agree_i
return the smallest index j with s[j] = 101;
```

Algorithm 5: Safe agreement (adapted from [10])

(001), then backs off if it detects another process already at level 2 (101). If a snapshot includes no processes at level 1, it is safe for any process that sees that snapshot to agree on the smallest process at level 2, because any later process will back off before reaching level 2. Termination is also guaranteed as long as no process stays at level 1 forever.

To implement the storage allocator, we add a safe-agreement object to each row; this increases the size of each row by $3n$ bits. We also include a $\lceil \lg n \rceil$-bit field to allow a reader to quickly identify the owner of a row. Despite these additions, we still get $1 + o(1)$ amortized bits per write by making $\ell = \omega(m + n)$.

To allocate a new row, a writer interleaves attempts to win the safe-agreement objects for the next n rows for which it has not yet determined a winner. At least one of these safe-agreement objects will eventually return a value. If this is the id of the writer, it can claim the row by writing its id to the id field and proceed as in the single-writer construction. If not, it continues to attempt to acquire a row from the set obtained by throwing in the next row that it has not previously attempted to acquire. In either case the writer eventually acquires a row or reaches a state where all but $n - 1$ rows have been allocated.

The reader's task is largely unchanged from the basic MWMR construction: for each of the n SWMR registers, there are at most 2^m active rows it must check for updates, plus up to n additional rows it must check for new activity. This again gives an amortized cost from the readers of $O(n2^m)$ steps per write operation. In addition, each write operation may impose a cost of $O(n)$ bit operations from extra collects in the snapshot on each other writer, for a total of $O(n^2)$ bit operations, for each row it attempts to allocate. This gives a total cost over all writes of $O(kn^3)$ for an amortized cost of $O(n^3/\ell) = O(n^2)$ per write. So the total amortized cost per write is $O(n(n + 2^m))$. This gives:

Theorem 4.2. *There is an algorithm that implements an n-process MWMR m-bit register that supports up to t writes, using space complexity of $(2 + o(1))t$ when $t = \omega((m + \log n)n^6 2^m)$, and with amortized step complexity $O(n^2 2^m)$ for both write and read operations.*

5 An unrestricted MWMR implemention based on max registers

The tabular WOM code constructions have two deficiencies: they have huge time complexity, and they are limited-use, permitting only a fixed maximum number t of write operations. In this section, we give a different construction (using unbounded space) that implements a wait-free m-bit MWMR register on top of a max register [3]. A max register provides `WriteMax` and `ReadMax` operations, where `ReadMax` returns the *largest* value written by any preceding `WriteMax`.

There are several known constructions of max registers [3, 4, 20], each of which has different goals. The basic structure we use here follows the tree implementation of [3], described in §5.1 for completeness. In §5.2 we construct our full MWMR m-bit register and prove its properties.

5.1 Tree-based max register

The standard tree-based max register is any binary tree whose leaves correspond to the possible values of the max register. Each node represents a single-bit register that can hold a value in $\{0, 1\}$. The aim is to have the current value of the tree be the rightmost leaf that is set to 1. To implement this, a `ReadMax` operation travels down the tree starting from the root node, going to the left child of a node if it reads 0 and going to the right child if it reads 1. A `WriteMax`(v) starts from the leaf that corresponds to the value v, and travels up the tree to the root, setting to 1 all bits to which it arrives from the right. An important technicality is that in order to make the above linearizable, before a `WriteMax` makes any change to a left subtree of a node, it checks that the bit at this node is 0. This allows, for example, implementing a b-bounded max register (supporting values in values in $\{0, ..., b-1\}$) using a balanced binary tree of depth $O(\log b)$.

However, we can also use an unbalanced binary tree with the property that each leaf v is at depth $O(\log v)$. Since the step complexity of any operation is proportional to the depth of the leaf it writes or returns, the latter gives an implementation with a step complexity of $O(\log v)$. This implementation also has the nice property that it can be extended to support an unbounded number of values. This is done by having a leaf at depth $O(n)$ point to a multi-writer snapshot object. This way the step complexity does not increase with the value v beyond limit, but is rather bounded by $O(\min\{\log v, n\})$, since there are linear-time implementations of snapshot objects [6, 23]. The problem with having the step complexity increase beyond limit is not only a complexity problem—it is also a computational problem in the sense that the implementation is not wait-free if we keep the tree infinite, since a `ReadMax` operation can always be pushed farther down to the right side of the tree by a new `WriteMax` operation with a larger value.

Using WOM, the tree-based max register implementation has the nice property that only single-bit registers are used and their value can only be changed from 0 to 1. However, we cannot use the snapshot object that truncates the tree at depth $O(n)$, because its known implementations do not translate into the write-once model. Another approach that avoids the usage of the snapshot object is the randomized helping mechanism used in [4]. But this also does not translate to WOM, and hence we seek a different helping solution.

5.2 Adding the helping mechanism

For the sake of presentation, we start with describing an attempt for building a standard register out of a tree-based max register. This most basic approach only gives a non-blocking SWMR register. Then, we add a helping mechanism to obtain wait-freedom. This still only works for the case of a single-writer-single-reader (SWSR) implementation. We then explain the challenges in extending this to the multi-writer-multi-reader (MWMR) case. We keep the descriptions of the non-blocking and wait-free SWMR registers informal for clarity, and leave the pseudocode and formal proof for the presentation of our full MWMR construction with a more involved helping mechanism for all processes.

5.2.1 A non-blocking SWSR write-once register

Suppose we have a single writing process p_W, and a single reading process p_R. We first describe an implementation of a SWSR register that is non-blocking but not wait-free, in order to give intuition for our framework.[2] We maintain an infinite unbalanced tree-based unbounded max register Max, and associate an m-bit register value(t) with each leaf t. The values of Max represent timestamps and value(t) represents the value written in the t-th operation, as follows. On its t-th `write` operation, p_W writes its input value into value(t) and then executes a `WriteMax`(t) operation on Max. Upon its `read` operation, p_R performs `ReadMax` on Max and then reads and returns value(t), where t is the timestamp returned from the `ReadMax` operation. The problem with this implementation is that operations of p_R are not wait-free because `ReadMax` may never return if p_W keeps invoking `WriteMax` operations and thus constantly pushes p_R down the rightmost infinite path of the tree.

5.2.2 A wait-free SWSR register

To make this implementation wait-free, we employ the following simple helping mechanism, which consists of an infinite array of bits HelpReq and an infinite array HelpData where each location has a 1-bit flag field and an unbounded register TS. When p_R starts its `read` operation, it first starts performing a `ReadMax` operation up to the first time at which it either returns the last value t it saw in previous invocations of `read` (or 0 if this is its first), or it discovers that a larger value was written. If t has not changed then p_R returns the same value value(t) that it returned for its previous `read` operation. Otherwise, p_R writes 1 into HelpReq$[k]$, where k is an integer that increases by 1 every time that p_R accesses HelpReq. Then, p_R alternates between taking another step in its `ReadMax` operation and reading HelpData$[k]$.flag. The operation completes either when p_R reads 1 from HelpData$[k]$.flag, in which case it reads t' from HelpData$[k]$.TS and returns value(t'), or when the `ReadMax` operation finishes and returns t', in which case p_R reads and returns value(t').

When p_W performs its t-th `write` operation, it firsts writes its input v to value(t) and then executes a `WriteMax`(t) operation on Max. Then, it checks whether p_R needs help by reading HelpReq$[k]$, where k is greater by 1 compared with the last index at which p_W accessed HelpReq, and 0 if this is its first access. If HelpReq$[k]$ is 1 then p_W writes t into HelpData$[k]$.TS and 1 into HelpData$[k]$.flag and returns.

The correctness of the wait-free SWSR register implementation described above is shown roughly as follows. If a `read` operation by p_R returns a value obtained by its embedded `ReadMax` operation, then it is the last value written by a `write` operation of p_W because of the correctness of Max. If instead a `read` operation by p_R returns a value obtained by finding a timestamp t in the corresponding HelpData$[k]$.TS array, then we can linearize it immediately after the `write` operation that wrote this timestamp because any `write` operation that accesses a location HelpData$[k']$.TS with $k' > k$ does so with a timestamp $t' > t$ because the timestamps are strictly monotone.

The step complexity of the t-th `write` operation is $O(\log t + m)$, because p_W needs $O(\log t)$ steps for `WriteMax`(t) and for writing t into some location of HelpData, m steps for writing value(t), and a single step for checking the current location in HelpReq. The latter is because if p_R accesses HelpReq then it must be because the value of Max changed since its last `read` operation, which means that if p_W executes a `write` operation then it only needs to check a single location in HelpReq. That is, it cannot be that p_R fills in many locations in HelpReq without p_W finding them, because every

[2] For the purpose of obtaining only a non-blocking SWSR write-once register, it is sufficient to construct an infinite array of m-bit locations to which the writer writes in increasing order and the reader searches for the last written location. However, we use here a max-register based implementation in order to build upon it when constructing our following wait-free SWSR and MWMR implementations.

accessed location by p_R implies the invocation of another `write` operation. Notice that this does not work if there are multiple readers because then p_W has to access a different pair of HelpReq and HelpData arrays for each reader who might need help.

The step complexity of a `read` operation that returns a value associated with timestamp t is also $O(\log t + m)$, because p_R needs $O(\log t)$ steps for the `ReadMax` operation that returns the previously seen timestamp $t' < t$ or finds that it increased, a single step for updating the current location HelpReq, $O(\log t)$ steps for reading HelpData, and m steps for reading value(t). The reason that only $O(\log t)$ steps are needed for reading HelpData is because the TS field is read only once, and the flag field is read at most a number of times as the number of steps of the `ReadMax` operation. Notice that this does not work if there are multiple writers because then p_R has to access a different pair of HelpReq and HelpData arrays for each writer who might be the only one who can provide help.

As explained above, this implementation does not extend to multiple writers or multiple readers. Below, we use the preceding framework along with a slightly more involved helping mechanism in order to allow for multiple writers *and* multiple readers.

5.2.3 A MWMR write-once register

In this section, we give the full MWMR register implementation. The main issues we have to handle are the following. First, a writer needs to choose a new timestamp for its next operation, which now depends on operations by other processes and cannot be generated locally. This is addressed by having the writer do a `read` operation and increment the returned timestamp in order to get its new timestamp. Notice that a `ReadMax` alone is insufficient because the writer may also need to use the helping mechanism.

Second, a `read` operation (including the embedded `read` operations!) may need to get help from one of the `write` operations, if its `ReadMax` operation does not finish. This is handled by interleaving the `ReadMax` operation with the reader side of the helping mechanism, which consists of signaling to all processes that help is needed and then cycling over an array according to the possible processes and waiting for a current timestamp to be written there. Here, a good timestamp t is one which the reader can safely adopt and return the value associated with it.

Third, a `write` operation may need to provide help for one of the `read` operations. To this end, the `WriteMax` operation is interleaved with writing to the helping mechanism. This is done by cycling over the array of all processes and checking whether any of them signaled for help. If the operation sees a signal for help at the currently checked location, it does another `ReadMax` and writes its timestamp as its helping data. Notice that the size of timestamp is unbounded, and in particular it may be greater than the timestamp that the `WriteMax` used. This means that we might not be able to afford writing the new timestamp without having a complexity that depends on the number of operations also for a `write` operation. To solve this, it may be the case that a `write` operation begins writing a certain timestamp as its helping data, but writing this value continues only in the next `write` operation invoked by this process. In fact, it may be that writing a timestamp for helping is spanned over multiple `write` operations of the process.

Below, we give the pseudocode of our algorithm and its analysis. The main max register we use is similar to the one from [4], but is slightly modified as follows. It is built as a binary tree of switch bits, which consists of an infinite spine forming the rightmost path through the tree, each node of which has a balanced m_ℓ-valued max register (of depth $O(\log m_\ell)$), rooted at its left child. Here we take m_ℓ to be 2^ℓ. We denote the m_ℓ-valued max registers as M_ℓ, where $\ell = 0, 1, \ldots$ is an increasing integer starting from the root.

When a `write` operation needs to write a value associated with timestamp t, it sets the relevant bits of the max register M_ℓ from leaf t up to the spine, and then from the spine to the root only until the previously set bit there. The fact that we do not have to go all the way up to the root is because there is no need to set bits that are already set, and hence this modification does not affect correctness. It will, however, allow us to save time, as in [4].

Shared data:
Max: an unbounded max register.
value(t): an m-bit register for every integer t.
HelpReq$[i, j, t]$: an infinite array of bits for each $i, j \in [n]$.
HelpData$[i, j, t]$: an infinite array for each $i, j \in [n]$.
HelpData$[i, j, t]$.flag is a single bit.
HelpData$[i, j, t]$.TS is an unbounded register.

Local data:
$helpFrom$: a process ID.
$helpTo$: a process ID.
$loc(helpTo)$: an array of n integers.

procedure `write`(v)
    Alternate between steps of `WriteMain`(v) and `WriteHelp` until the former finishes.

procedure `WriteMain`(v)
    $(t', -) \leftarrow$ `read`
    $t \leftarrow ((\lfloor t'/n \rfloor + 1) \cdot n) + pid$
    value$(t) \leftarrow v$
    `WriteMax`$(M_{\lceil \log t \rceil}, t - 2^{\lfloor \log t \rfloor})$ // Write to the corresponding m_ℓ-valued max register
    for $s = \lceil \log t \rceil$ **to** $\lfloor \log t \rfloor$ **do**
        spine$[s] \leftarrow 1$

procedure `WriteHelp`
    $helpTo \leftarrow helpTo + 1 \bmod n$
    if HelpReq$[helpTo, pid, loc(helpTo)]$ $==$ *1* **then**
        $(t'', -) \leftarrow$ `read`
        HelpData$[helpTo, pid, loc(helpTo)] \leftarrow t''$ // this may take many steps, possibly spread across multiple main-algorithm writes
        $loc(helpTo) \leftarrow loc(helpTo) + 1$

Algorithm 6: Implementation of operation `write` for a write-once register using an unbounded tree-based max register and a helping mechanism.

Our main result for this section is the following.

Theorem 5.1. *There is an algorithm that implements an n-process MWMR register of m bits with unbounded space, where the* amortized *step complexity of a write operation that gets associated with a timestamp t is $O(\log t + m + \log n)$ and the step complexity of a read operation that reads a value associated with a timestamp t is $O(\log t + m + \log n)$.*

Proof. We linearize a `write` operation at the first time in which all the switches on the path of the max register tree from the root to the timestamp t used by the operation are set.

We linearize a `read` operation that returns with the pair $(t, -)$ immediately after the `write` operation that used timestamp t is linearlized. This is well-defined for returning in the `ReadMain`

```
procedure read
    Alternate between steps of ReadMain and ReadHelp until one of them finishes.

procedure ReadMain
    t ← ReadMax(spineLoc)
    v ← value(t)
    return (t, v)

procedure ReadHelp
    if h > 0 then
        for helpFrom = 0, .., n − 1 do
            HelpReq[pid, helpFrom, h − 1] ← 1
        for helpFrom = 0, .., n − 1 do
            HelpReq[pid, helpFrom, h] ← 1
    while helped == false do
        helpFrom ← helpFrom + 1  mod n
        if HelpData[pid, helpFrom, h].flag == 1 then
            t ← HelpData[pid, helpFrom, h].TS
            helped ← true
    helped ← false
    h ← h + 1
    v ← value(t)
    return (t, v)
```

Algorithm 7: Implementation of operation `read` for a write-once register using an unbounded tree-based max register and a helping mechanism.

part because (a) there must be such a `write` operation because of the correctness of the max register construction, and (b) there can be only one such operation because each process is allocated a separate set of timestamps and uses them in a strictly increasing manner. We linearize a `read` operation that returns in its `ReadHelp` part with the pair $(t, -)$ immediately after the `write` operation that used timestamp t is linearlized. This is also well-defined for returning in the `ReadHelp` part by an induction on the linearization order: an operation that updates its HelpData with the timestamp t does so only after reading that value by a `read` operation.

It is easy to see that the linearization is consistent with the order of non-overlapping operations, because timestamps of `write` operations can only increase and `read` operations always start reading `ReadMain` from the last location *spineLoc* that an operation by the same process saw for the max register.

Next, we need to show that if a `read` operation *op* returns a value associated with a timestamp t then this timestamp is the largest among all `write` operations that are linearized before *op*. First, it must be the case that some `write` operation with timestamp t is linearized before *op*, by definition of the linearization points, and because a value can only be written to the HelpData array if it was previously read from the max register. Second, we need to show that there cannot be a `write` operation *op*′ with timestamp $t' > t$ is linearized before *op*. Suppose that this is not the case, and let *op* be the first `read` operation for which there is a `write` operation *op*′ with a larger timestamp that is linearized before it. Let t be the timestamp of *op* and let $t' > t$ be the timestamp of *op*′.

correctness of the max register construction, it cannot be the case the *op* returns from the `ReadMain` part while being linearized after a `WriteMain` operation with a larger timestamp. Hence, *op* returns from `ReadHelp`. This implies that it is linearized after the `read` operation op'' embedded in the `write` that helps it, which contradicts it being the first such `read` operation.

It remains to analyze the step complexity of the algorithm. A `read` operation requires m steps for reading $\mathsf{value}(t)$ and at most $O(\log t + \log n)$ steps for reading the max register in the `ReadMain` part, where $t \cdot n + pid$ is the timestamp returned, regardless of which part it returns from. This is because the max register construction requires a number of steps that is logarithmic in the value written, and because the `ReadHelp` part can only double the number of steps performed by the `ReadMain` operation.

A `write` operation associated with a timestamp $t \cdot n + pid$ requires m steps for writing $\mathsf{value}(t)$ and at most $O(\log t + \log n)$ steps for reading the max register (which returns a smaller timestamp). This is because the leaf t is in depth $O(\log t)$. The last issue we need to address is the `WriteHelp` part. Though it can only double the number of steps performed by the `WriteMain` operation, the problem is that to allow the helping mechanism to be correct, the `write` operation has to perform another embedded `read` operation whose value it uses for the helping mechanism, and the returned timestamp t' of this operation may be much larger than t.

However, an important property that the above construction satisfies is that the step complexity of a `write` operation that uses timestamp t is only $O(\log t + m + \log n)$, when *amortized* over all `write` operations. The reason for this is that although a `write` operation may perform an embedded `read` operation whose complexity depends on the value t' that is associated with it, this complexity can be accounted for $O(t'/n)$ different `write` operations, because the implementation satisfies the *n-bounded increments* property [4]. This property states that a value (timestamp in our context) written to the max register by an operation *op* is at most $2n - 1$ larger than the value of latest completed `write` operation. This implies that a timestamp of t' can be reached only after at least t'/n `write` operations have been invoked. □

6 Lower bounds

For any implementation of a standard register from write-once memory, it is trivial to see that we must use an infinite amount of space in order to support an unbounded number of write operations. This holds even without concurrency, because a finite number of bits can encode a finite number of values, and because we cannot reset a 1 bit to 0. In this section, we provide two additional, non-trivial, lower bounds.

The first is an $\Omega(\log t)$ lower bound on the worst-case cost of a `read` operation in an execution with t `write` operations, even when implementing a one-bit register. This is an immediate consequence of Kraft's inequality [27]. Consider a family of executions $\Xi_0, \Xi_1, \ldots \Xi_t$, in which a single writer process alternates between writing a 0 value and a 1 value, with i writes in Ξ_i. In each execution, following these writes is a second reader process p that executes `read`.

Assume that the reader is deterministic. Let x_i be the sequence of bits read by p in Ξ_i. Observe that the x_i form a prefix-free code (where no codeword is a prefix of another), because the reader chooses to stop deterministically based on the bits it has read so far. Observe further that because write-once bits can never switch from 1 to 0, the x_i can only increase in lexicographic order: in particular this means they are all distinct. Kraft's inequality [27] then gives that $\sum_{i=0}^{t} 2^{-|x_i|} \leq 1$, implying that at least one (and indeed most) of the x_i have length $\Omega(\log t)$.

By treating a randomized reader as a mixture of deterministic readers, the same result applies to the expected worst-case cost of a read. Note that this holds even with an oblivious adversary, because the argument depends only on the information-theoretic properties of the possible sequences of bits observed by a reader, and not on any interaction between the reader and the schedule.

The previous lower bound assumes that the reader performs only one read operation. A reader that performs multiple reads may be able to save work by avoiding re-reading bits that it already knows to be 1. However, we can still show a second lower bound that is a trade-off between the number of bits written by a `write` operation that writes an m-bit value and the number of bits a `read` operation *op* has to look at to get new value, even if it observed the contents of memory immediately before the `write`.

Suppose that the `read` operation accesses at most r bits, and the `write` operation sets at most k bits. As in the previous bound we can consider each possible sequences of bits $x_0, \dots x_{2^m-1}$ read by the reader, where x_i gives the sequence corresponding to the value i. Each such sequence is distinct, has length at most r and contains at most k ones, so we have $\sum_{i=1}^{k} \binom{r}{i} \geq 2^m$. For $k = 1$, this bound is reached (up to constants) by the construction of §4. It is an interesting question whether the trade-off can be realized in general for larger k.

7 Discussion

The present work initiates the study of write-once memory in a concurrent setting. Our results demonstrate that it is in principle possible to implement operations for standard, rewritable shared-memory using write-once memory with low space overhead and polynomial amortized time complexity. Several open questions remain:

1. Is it possible to combine low space overhead with low time overhead?
2. To what extent could a small amount of rewritable shared memory allow more efficiency in use of write-once shared memory?
3. What can one say about stronger write-once primitives, such as (non-resettable) test-and-set bits, either as a target or a base object for implementations?

References

[1] Zahra Aghazadeh and Philipp Woelfel. On the time and space complexity of ABA prevention and detection. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 193–202, 2015.

[2] Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*, pages 97–109, 2011.

[3] James Aspnes, Hagit Attiya, and Keren Censor-Hillel. Polylogarithmic concurrent data structures from monotone circuits. *J. ACM*, 59(1):2, 2012.

[4] James Aspnes and Keren Censor-Hillel. Atomic snapshots in $O(\log^3 n)$ steps using randomized helping. In *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, pages 254–268, 2013.

[5] James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. *Theory Comput. Syst.*, 55(3):451–474, 2014.

[6] Hagit Attiya and Arie Fouren. Adaptive and efficient algorithms for lattice agreement and renaming. *SIAM J. Comput.*, 31(2):642–664, 2001.

[7] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[8] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Trans. Comput. Syst.*, 31(4):10:1–10:24, December 2013.

[9] Aman Bhatia, Aravind Iyengar, and Paul H. Siegel. Multilevel 2-cell t-write codes. In *IEEE Information Theory Workshop (ITW)*, 2012.

[10] Elizabeth Borowsky, Eli Gafni, Nancy A. Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, 2001.

[11] David Burshtein and Alona Strugatski. Polar write once memory codes. *IEEE Transactions on Information Theory*, 59(8):5088–5101, 2013.

[12] Yuval Cassuto and Eitan Yaakobi. Short (Q)-ary fixed-rate WOM codes for guaranteed rewrites and with hot/cold write differentiation. *IEEE Transactions on Information Theory*, 60(7):3942–3958, July 2014.

[13] Gérard D. Cohen, Philippe Godlewski, and Frans Merkx. Linear binary code for write-once memories. *IEEE Transactions on Information Theory*, 32(5):697–700, 1986.

[14] Eyal En Gad, Huang Wentao, Yue Li, and Jehoshua Bruck. Rewriting flash memories by message passing. In *IEEE International Symposium on Information Theory (ISIT)*, 2015.

[15] Amos Fiat and Adi Shamir. Generalized 'write-once' memories. *IEEE Transactions on Information Theory*, 30(3):470–479, 1984.

[16] Fang-Wei Fu and A. J. Han Vinck. On the capacity of generalized write-once memory with state transitions described by an arbitrary directed acyclic graph. *IEEE Transactions on Information Theory*, 45(1):308–313, 1999.

[17] George Giakkoupis and Philipp Woelfel. On the time and space complexity of randomized test-and-set. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 19–28, 2012.

[18] Philippe Godlewski. WOM-codes construits à partir des codes de Hamming. *Discrete Mathematics*, 65(3):237–243, 1987.

[19] Chris Heegard. On the capacity of permanent memory. *IEEE Transactions on Information Theory*, 31(1):34–41, 1985.

[20] Maryam Helmi, Lisa Higham, and Philipp Woelfel. Strongly linearizable implementations: possibilities and impossibilities. In *ACM Symposium on Principles of Distributed Computing, PODC '12, Funchal, Madeira, Portugal, July 16-18, 2012*, pages 385–394, 2012.

[21] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[22] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[23] Michiko Inoue and Wei Chen. Linear-time snapshot using multi-writer multi-reader registers. In *Distributed Algorithms, 8th International Workshop, WDAG '94, Terschelling, The Netherlands, September 29 - October 1, 1994, Proceedings*, pages 130–140, 1994.

[24] Amos Israeli and Amnon Shaham. Optimal multi-writer multi-reader atomic register. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '92, pages 71–82, New York, NY, USA, 1992. ACM.

[25] Amos Israeli and Amnon Shaham. Time and space optimal implementations of atomic multi-writer register. *Information and Computation*, 200(1):62 – 106, 2005.

[26] Adam N. Jacobvitz, Robert Calderbank, and Daniel J. Sorin. Coset coding to extend the lifetime of memory. In *IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[27] Leon G. Kraft. A device for quantizing, grouping, and coding amplitude-modulated pulses. *M.S. Thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering*, 1949.

[28] Leslie Lamport. On interprocess communication. part I: basic formalism. *Distributed Computing*, 1(2):77–85, 1986.

[29] Leslie Lamport. On interprocess communication. part II: algorithms. *Distributed Computing*, 1(2):86–101, 1986.

[30] Jiayin Li and Kartik Mohanram. Write-once-memory-code phase change memory. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2014.

[31] Fabio Margaglia and André Brinkmann. Improving MLC flash performance and endurance with extended P/E cycles. In *IEEE 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[32] Serge A. Plotkin. Sticky bits and universality of consensus. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, PODC '89, pages 159–175, New York, NY, USA, 1989. ACM.

[33] Ronald R. Rivest and Adi Shamir. How to reuse a "write-once" memory. *Information and Control*, 55:1–19, 1982.

[34] Amir Shpilka. New constructions of WOM codes using the Wozencraft Ensemble. *IEEE Transactions on Information Theory*, 59(7):4520–4529, 2013.

[35] Amir Shpilka. Capacity achieving multiwrite WOM codes. *IEEE Transactions on Information Theory*, 60(3):1481–1487, 2014.

[36] Jack K. Wolf, Aaron D. Wyner, Jacob Ziv, and Janos Korner. Coding for a write-once memory. *AT&T Bell Laboratories Technical Journal*, 63(6):1089–1112, 1984.

[37] Yunnan Wu. Low complexity codes for writing a write-once memory twice. In *IEEE International Symposium on Information Theory (ISIT)*, 2010.

[38] Yunnan Wu and Anxiao Jiang. Position modulation code for rewriting write-once memories. *IEEE Transactions on Information Theory*, 57(6):3692–3697, 2011.

[39] Eitan Yaakobi, Scott Kayser, Paul H. Siegel, Alexander Vardy, and Jack K. Wolf. Codes for write-once memories. *IEEE Transactions on Information Theory*, 58(9):5985–5999, 2012.

[40] Eitan Yaakobi and Amir Shpilka. High sum-rate three-write and non-binary WOM codes. In *IEEE International Symposium on Information Theory (ISIT)*, 2012.

[41] Gala Yadgar, Eitan Yaakobi, and Assaf Schuster. Write once, get 50% free: Saving SSD erase costs using WOM codes. In *13th USENIX Conference on File and Storage Technologies (FAST)*, 2015.

[42] XianWei Zhang, Le Jang, Youao Zhang, Chuanjun Zhang, and Jun Yang. WoM-SET: Low power proactive-SET-based PCM write using WoM code. In *IEEE International Symposium on Low Power Electronics and Design (ISLPED)*, 2013.