Allocate-On-Use Space Complexity of Shared-Memory Algorithms

James Aspnes
Yale University Department of Computer Science

Bernhard Haeupler
Carnegie Mellon School of Computer Science

Alexander Tong
Yale University Department of Computer Science

Philipp Woelfel
University of Calgary Department of Computer Science

Abstract

Many fundamental problems in shared-memory distributed computing, including mutual exclusion [8], consensus [18], and implementations of many sequential objects [14], are known to require linear space in the worst case. However, these lower bounds all work by constructing particular executions for any given algorithm that may be both very long and very improbable. The significance of these bounds is justified by an assumption that any space that is used in some execution must be allocated for all executions. This assumption is not consistent with the storage allocation mechanisms of actual practical systems.

We consider the consequences of adopting a more dynamic approach to space complexity, where an object only counts toward the space complexity if it is used. We describe two basic versions of this measure, an allocate-on-access model in which any operation on an object causes it to be counted, and an allocate-on-update model in which only operations that change the initial default state cause it to be counted. We show how the allocate-on-access model can simulate allocate-on-update with overhead logarithmic in the largest address used. We also show that many known randomized algorithms for fundamental problems in shared-memory distributed computing have expected space complexity much lower than the worst-case lower bounds, and that many algorithms that are adaptive in time complexity can also be made adaptive in space complexity. Similarly, some recent lower bounds on expected time complexity for randomized implementations of restricted-use objects can be modified in a straightforward way to get lower bounds on expected space complexity.

For the specific problem of mutual exclusion, we develop a new algorithm that illustrates an apparent trade-off between low expected space complexity and low expected RMR complexity. Whether this trade-off is necessary is an open problem.

2012 ACM Subject Classification C.2.4: Distributed Systems

Keywords and phrases Space complexity; memory allocation; mutual exclusion

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Supported in part by NSF grants CCF-1637385 and CCF-1650596.
2 Supported by NSF grant CCF-1650596
1 Introduction

The space complexity of shared-memory distributed data structures and protocols, measured in terms of the number of distinct objects needed to implement them, is typically linear in the number of processes. On the upper bound side, this follows from the ability to implement most algorithms using a single output register for each process (which might hold very large values). On the lower bound side, linear lower bounds have long been known for fundamental problems like mutual exclusion [8] and implementing many common shared-memory objects [14]; and have been shown more recently for consensus [10, 18].

Linear bounds are not terrible, but they do limit the scalability of concurrent data structures for very large numbers of processes. The structure of the known lower bound proofs suggest that executions requiring linear space may be rare: known bounds on mutual exclusion and perturbable objects may construct exponentially long executions, while the bounds on consensus depend on constructing very specific executions that are avoidable if the processes can use randomization.

We propose considering per-execution bounds on the space complexity of a randomized protocol, where the protocol is charged only for those objects that it actually uses during the execution. This allows considering expected space-complexity bounds and high-probability space complexity bounds, which would otherwise be meaningless if the algorithm is charged for any object that might be used, whether it is used or not. We define, in Section 2, two models of allocate-on-use, an allocate-on-access model where any operation on an object causes the object to be counted against the space complexity, and a less restrictive allocate-on-update model where only operations that change the state cause it to be counted. We believe that these models give a more refined description of the practical space complexity of many shared-memory algorithms, and observe in our analysis of previously known algorithms in Section 3 that our measures formalize notions of allocate-on-use space complexity that have already been informally considered by other researchers.

To control for algorithms that might abuse huge address spaces by only allocating objects sparsely, we consider contiguous variants of these measures where allocating an object effectively allocates all objects at smaller addresses.

These space complexity may not make sense in all contexts, but they have strong practical justifications:

1. In a system that provides storage allocation as part of its memory management, it may be that unused registers or pages have no actual cost to the system. Alternatively, it may be possible to construct high-level storage allocation mechanisms even in an adversarial setting that allow multiple protocols with dynamic space needs to share a large fixed block of memory.

2. Given an algorithm with low expected contiguous space complexity—or better yet, with high-probability guarantees of low contiguous space complexity—we can run it in fixed space at the cost of accepting a small chance that the algorithm fails by attempting to exceed its space bound. Thus randomized space complexity can be a tool for trading off space for probability of failure.

In addition to the definitions, Section 2 also provides several utility results for proving low allocate-on-use space complexity for specific algorithms. These include basic inequalities relating the different models, a general method for interleaving implementations of objects to allow proving allocate-on-use space complexity bounds by composition, and a low-overhead simulation of allocate-on-update in the stricter allocate-on-access model, and

To show the applicability of our measures, we also include several positive results: In
Section 3, we demonstrate that many known algorithms for fundamental shared-memory
algorithms either have, or can be made to have with small tweaks, low space complexity
in most executions. In Section 4, we describe a new randomized algorithm for mutual
exclusion that achieves $O(\log n)$ space complexity with high probability for polynomially
many invocations.

Finally, we discuss open problems in Section 5.

1.1 Model

We consider a standard asynchronous shared-memory model in which a collection of $n$ pro-
cesses communicate by performing operations on shared-memory objects. Concurrency
is modeled by interleaving operations; each operation takes place atomically and is a step of
the process carrying it out. For convenience, we assume that the identity of an operation
includes the identity of the object to which it is applied.

Scheduling is assumed to be under the control of an adversary. If the processes are
randomized, then each has access to local coins that may or may not be visible to the
adversary. An adaptive adversary may observe the internal states of the process, including
the results of local coin-flips, but cannot predict the outcome of future coin-flips. An
oblivious adversary simply provides in advance a list of which process carries out an
operation at each step, without being able to react to the choices made by the processes.

1.1.1 Time complexity

The individual step complexity of an algorithm executed by a single process is the number
of steps carried out by that process before it finishes. The total step complexity is the
total number of steps over all processes. For mutual exclusion algorithms, we may consider
the remote memory reference (RMR) complexity, in which read operations on a register
are not counted if (a) the register has not changed since the last read by the same process
(in the distributed shared memory model) or (b) no operation has been applied to the
registers since the last read by the same process (in the cache-coherent model).

2 Space complexity

The traditional measure of space complexity is worst-case space complexity, the number
of distinct objects used by the system. We introduce several alternative measures of dynamic
space complexity, which may vary from execution to execution.

We assume that the objects in the system are numbered $O_1, O_2, \ldots$. The indices of these
objects form the address space of the system, which may or may not be finite.

We distinguish between read operations, which never change the state of an object, and
update operation, which might. A set of operations accesses an object if it includes at
least one operation on that object. It updates an object if it includes at least one operation
on that object that is not a read.

In an allocate-on-access model, any access to an object causes it to be counted toward
the space complexity. Formally, the space complexity of a set of operations with allocate-on-
access is the number of distinct objects that are accessed at least one of the operations. In
the allocate-on-update model, only updates to an object cause it to be counted; the space
complexity of a collection of operations with allocate-on-update is the number of distinct
objects that are updated by at least one of the operation.
Allocate-On-Use Space Complexity of Shared-Memory Algorithms

<table>
<thead>
<tr>
<th>Measure</th>
<th>Addressing</th>
<th>Allocation</th>
<th>Space complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>sparse</td>
<td>access</td>
<td>Number of objects accessed</td>
</tr>
<tr>
<td>SU</td>
<td>sparse</td>
<td>update</td>
<td>Number of objects updated</td>
</tr>
<tr>
<td>CA</td>
<td>contiguous</td>
<td>access</td>
<td>Highest index of objects accessed</td>
</tr>
<tr>
<td>CU</td>
<td>contiguous</td>
<td>update</td>
<td>Highest index of objects updated</td>
</tr>
</tbody>
</table>

Table 1: Space complexity as a function of addressing mode and allocation policy

In both cases, we assume by default sparse addressing, where processes are given random access to all objects in the system. We may also consider a model with contiguous addressing, where allocating object \( O_m \) automatically allocates all objects \( O_{m'} \) with \( m' < m \). With contiguous addressing, the space complexity of a set of operations is the highest address of any object that is accessed/updated, or 0 if no object is accessed/updated.

The choice of addressing mode is independent of the choice of allocation policy, giving four variants on the space complexity measure. For a given set of operations \( \Pi \), we write these measures as \( SA(\Pi) \), \( SU(\Pi) \), \( CA(\Pi) \), and \( CU(\Pi) \), for sparse allocate-on-access, sparse allocate-on-update, contiguous allocate-on-access, and contiguous allocate-on-update, respectively. These are summarized in Table 1.

If not otherwise specified, we will generally treat space complexity as referring to the SA measure, sparse addressing with allocate-on-access.

We define these measures in terms of sets of operations for maximum generality. In the context of specific executions, we will typically consider the set of operations that are invoked in the execution, including operations that may be incomplete. This gives a definition for an execution \( H \) of \( SU(H) = SU(\Pi_H) \), \( SA(H) = SA(\Pi_H) \), etc., where \( \Pi_H \) is the set of operations invoked in \( H \).

### 2.1 Basic relations between the measures

The four space complexity measures are partially ordered as depicted in Figure 1. We state these relations as a lemma:

▶ **Lemma 1.** For any set of operations \( \Pi \), \( SU(\Pi) \leq SA(\Pi) \leq CA(\Pi) \), and \( SU(\Pi) \leq CU(\Pi) \leq CA(\Pi) \).

But there exist sets of operations \( \Pi \) and \( \Pi' \) such that \( SA(\Pi) < CU(\Pi) \) and \( SA(\Pi) > CU(\Pi') \).

**Proof.** That \( SU(\Pi) \leq SA(\Pi) \) and \( CU(\Pi) \leq CA(\Pi) \) follows from every update also being an access.

For \( SU(\Pi) \leq CU(\Pi) \) and \( SA(\Pi) \leq CA(\Pi) \), apply the Pigeonhole Principle: if \( \Pi \) updates (or accesses) \( m \) distinct objects, the largest address must be at least \( m \).

To show incomparability in general between \( SA \) and \( CU \), consider an set of operations \( \Pi \) that accesses few objects but updates some object with large address, giving \( SA(\Pi) < CU(\Pi) \); and a set of operations \( \Pi' \) that accesses many objects but updates none, giving \( SA(\Pi') > CU(\Pi') \).

### 2.2 Composability

A natural requirement for any complexity measure is composability: it should be possible to analyze the complexity of different components of a system independently, then combine
these analyses into a bound on the complexity of the system as a whole. In the context of dynamic space complexity, this means showing that if we have a bound on the space complexity of one set of operations $\Pi$ operating on a set of objects $O_1, O_2, \ldots$ and on another set of operations $\Pi'$ operating on a possibly-overlapping set of objects $O_1', O_2', \ldots$, then we can combine these bounds to get a bound on the space complexity of $\Pi \cup \Pi'$.

It is easy to see that both the allocate-on-update and allocate-on-access measures are subadditive with sparse addressing. The reason is that in either case, we are simply counting which objects are used without worrying about where they live in the address space. For contiguous addressing, we need to make a stronger assumption that we are allowed to rearrange the addresses of the objects used by two sets of operations in an appropriate way when combining them. This is a technique that will recur often when minimizing the size of the address space.

Lemma 2. Let $\Pi$ and $\Pi'$ be sets of operations. Then all of the following inequalities hold:

\[
\begin{align*}
SU(\Pi \cup \Pi') &\leq SU(\Pi) + SU(\Pi') \quad (2.1) \\
SA(\Pi \cup \Pi') &\leq SA(\Pi) + SA(\Pi') \quad (2.2) \\
CU(\Pi \cup \Pi') &\leq 2 \max(CU(\Pi), CU(\Pi')) \quad (2.3) \\
CA(\Pi \cup \Pi') &\leq 2 \max(CA(\Pi), CA(\Pi')) \quad (2.4)
\end{align*}
\]

Proof. For (2.1) and (2.2), this is immediate from the fact that the set of objects updated/accessed in $\Pi \cup \Pi'$ is the union of the sets of objects updated/accessed in $\Pi$ and $\Pi'$ respectively.

For (2.3) and (2.4), assign each object $O_i$ the address $2i - 1$ and each object $O_i'$ the address $2i$ in the combined system. In the case of overlap, where $O_i = O_j'$ for some $i$ and $j$, assign $O_i$ the smaller of $2i - 1$ and $2j$. This assigns each object in the combined system an address that is at most twice its address in each component. It follows that the largest address updated/accessed in $\Pi \cup \Pi'$ is at most twice the largest address update/accessed in $\Pi$ or $\Pi'$ individually.

While the guarantees in Lemma 2 are in general the best possible, in some specific cases it may be possible to arrange the address space more carefully to reduce $CU(\Pi \cup \Pi')$ or $CA(\Pi \cup \Pi')$.

2.3 Simulating allocate-on-update with allocate-on-access

More interesting is that we can give a wait-free, linearizable simulation of allocate-on-update using allocate-on-access, by constructing a data structure that marks which regions of memory have been updated. The overhead of this simulation is logarithmic in the largest address.
updated. The simulation assumes that we can add to the original objects in the system a
collection of one-bit atomic registers. For contiguous addressing, it also assumes that we can
assign indices to the new registers interspersed among the previous objects.

The essential idea is to place the objects $O_1, O_2, \ldots$ as the leaves of a binary search tree
whose internal nodes are one-bit registers that record if any object in their subtree has been
updated. A read operation on some object $O_i$ starts at the root of the tree and follows the
path to $O_i$ until it sees a 0, indicating that $O_i$ can be treated as still being in its initial state,
or reaches $O_i$ and applies the operation to it. Conversely, an update operation starts by
updating $O_i$ and then sets all the bits in order along the path from $O_i$ to the root.

To structure the tree, we take advantage of the well-known correspondence between
binary trees and prefix-free codes. Here the left edge leaving each node is labeled with a
0 and the right edge with a 1, the path to each leaf gives a code word, and the path to
each internal node gives a proper prefix of a code word. An appropriate choice of code will
make the depth of each object $O_i$ logarithmic in $i$. Careful assignment of addresses to each
node in the tree will also be needed to minimize the maximum address used. We defer these
issues until after showing linearizability of the implementation, which does not depend on
the precise structure of the tree.

Pseudocode for the simulation is given in Algorithm 1. Each register is labeled by a
codeword prefix. The objects are labeled with their original indices.

```
procedure apply(π)
    Let $O_i$ be the object on which $π$ is an operation
    Let $x_1x_2\ldots x_k$ be the encoding of $i$
    if $π$ is an update then
        $r \leftarrow π(O_i)$
        for $j ← k - 1 \ldots 0$ do
            $R_{x_1\ldots x_j} ← 1$
        end
        return $r$
    else
        for $j ← 0 \ldots k - 1$ do
            if $R_{x_1\ldots x_j} = 0$ then
                return $π$ applied to the initial state of $O_i$
            end
        end
        // Reached only if all nodes on path are 1
        return $π(O_i)$
    end
end
```

Algorithm 1: Applying operation $π$ to object $O_i$

Lemma 3. Algorithm 1 gives a linearizable implementation of $O_1, O_2, \ldots$.

Proof. Given a concurrent execution $H$ of Algorithm 1, we will construct an explicit lin-
erization $S$. The first step in this construction is to assign a linearization point to each
operation $π$ in $H$. If $π$ is an update operation on some object $O_i$, its linearization point is
the first step in $H$ at which (a) $π$ has been applied to $O_i$, and (b) every bit in an ancestor
of $O_i$ is set. If $π$ is a read operation, its linearization point is the step at which either $π$ is
applied to $O_i$, or the process executing $\pi$ reads a 0 from an ancestor of $O_i$. In the case of an update operation $\pi$, the linearization point follows the step in which $\pi$ is applied to $O_i$ and precedes the return of $\pi$ (since $\pi$ cannot return without setting all ancestors of $O_i$ to 1). In the case of a read operation $\pi$, the linearization point corresponds to an actual step of $\pi$. In both cases, the linearization point of $\pi$ lies within $\pi$’s execution interval.

If we declare $\rho \leq \sigma$ whenever $\rho$’s linearization point precedes $\sigma$’s, we get a preorder on all operations in $H$. Because each operation’s linearization point lies within its execution interval, this preorder is consistent with the observed execution order in $H$. But it is not necessarily a total order because update operations that are applied to the same object $O_i$ may be assigned the same linearization point: the first step at which all ancestors of $O_i$ are 1. Should this occur, we break ties by ordering such updates according to the order in which they were applied to $O_i$. We now argue that the resulting total order gives a sequential execution $S$ on $O_1, O_2, \ldots$. This requires showing that each operation that returns in $H$ returns the same value in $O_1, O_2, \ldots$, or the process executing $\pi$.

Fix some particular $O_i$. The operations on $O_i$ can be divided into three groups:

1. Read operations that observe 0 in an ancestor of $O_i$.  
2. Update operations that are applied to $O_i$ before all ancestors of $O_i$ are 1.  
3. Read or update operations that are applied to $O_i$ after all ancestors of $O_i$ are 1.

That these groups include all operations follows from the fact that any update operation is applied either before or after all ancestors of $O_i$ are 1, and any read operation that does not observe a 0 will eventually be applied to $O_i$ after all of its ancestors are 1.

Now observe that all operations in the first group are assigned linearization points before the step at which all ancestors of $O_i$ are 1; in the second at this step; and in the third group after this step. So $S$ restricted to $O_i$ consists of a group of read operations that return values obtained from the initial state of $O_i$, consistent with having no preceding updates; followed by a sequence of updates linearized in the same order that they are applied to $O_i$ in $H$; followed by a sequence that may contain mixed updates and reads that are again linearized in the same order that they are applied to $O_i$ in $H$. Since the first group of operations contain only read operations, the operations applied to $O_i$ in $H$ start with the same initial state as in $S$, and since they are the same operations applied in the the same order, they return the same values.

We now turn to the issue of choosing a tree structure and addressing scheme to minimize the cost of operations in the allocate-on-access model.

The particular code we will use to construct the tree is the Elias gamma code [9]. This encodes each positive integer $i$ as a sequence of bits, by first expressing $i$ as its unique binary expansion $1i_1i_2\ldots i_n$, and then constructing a codeword $\gamma(i) = 0^n1i_1i_2\ldots i_n$. This gives a codeword for each positive integer $i$ with length $2[\lg i] + 1 = O(\log i)$. The first few levels of the tree are depicted in Figure 2.

The address for the nodes in the tree are assigned by breadth-first traversal of the tree. In terms of codeword prefixes, nodes are sorted in order of increasing length of their labels, then lexicographically. This gives a sequence of registers and base objects that starts with $R_{[1]}^{[1]}$, $R_{[0]}^{O_1}$, $R_{[0]}^{O_2}$, $R_{[0]}^{O_3}$, $R_{[0]}^{O_4}$, $R_{[0]}^{O_5}$, $O_2$, $O_3$, $O_4$; the address of each object is given by its position in the sequence.

This ordering has the useful property that each node is assigned a lower address than its children. The following lemma shows that we also do not significantly increase the addresses of the original objects $O_i$.
Lemma 4. Let $S_n$ be the set consisting of all prefixes of length $n$ of codewords in the Elias gamma code. Then

$$|S_n| = 2^{\lceil n/2 \rceil}.$$ (2.5)

Proof. Use the fact that the position of the first 1 in each codeword encodes its length. If a codeword prefix starts with $0^k1$, then the original codeword has length $2k + 1$, and thus the codeword prefix itself has length at most $2k + 1$. This implies $k \geq (n - 1)/2$, which for integer $k$ means $k \geq \lceil (n - 1)/2 \rceil$. Because the first $\lceil (n - 1)/2 \rceil$ bits in each codeword prefix are 0, the remaining $\lceil n/2 \rceil$ bits give $2^{\lceil n/2 \rceil}$ possibilities. It is straightforward to see that each of those possibilities is represented in $S_n$, because any string of the form $0^\ell w$, where $\ell \geq |w|$, is a prefix of the codeword $0^\ell w0^{\ell - |w|}$. ▷

Corollary 5. For each object $O_i$, let $j$ be the index assigned to object $O_i$ by a breadth-first traversal of the tree. Then $j = O(i)$.

Proof. Let $n = 2 \lceil \log i \rceil + 1$ be the length of the codeword representing $i$. The number of codeword prefixes sorted before $c(i)$ is bounded by

$$\sum_{k=1}^{n} S_k \leq 2 \sum_{k=1}^{\lceil n/2 \rceil} 2^k < 2 \cdot 2^{\lceil n/2 \rceil + 1} \leq 2^{\lceil \log i \rceil + 3} < 2^\log i + 4 = 16i.$$ ▷

Using these results, we can bound the both the sparse and contiguous space complexity of Algorithm 1 in the allocate-on-access model based on the costs of the simulated operations in the allocate-on-update model:

Lemma 6. Consider an execution of Algorithm 1. Let $\Pi$ be the set of operations $\pi$ that are invoked in $H$ by calling $\text{apply}(\pi)$. Let $\Sigma$ be the set of operations applied to registers $R_x$ and objects $O_i$ in $H$. Then $\text{SA}(\Sigma) = O((\text{SU}(\Pi) \log \text{CU}(\Pi))$ and $\text{CA}(\Sigma) = O(\text{CU}(\Pi))$. 

Figure 2 Tree derived from Elias gamma code
Proof. Observe that invoking \( \text{apply}(\pi) \) for an update operation \( \pi \) on \( O_i \) requires accessing \( O_i \) and \( O(\log i) = O(\log \text{CU}(\Pi)) \) ancestors of \( O(i) \). Invoking \( \text{apply}(\pi') \) for any other update operation \( \pi' \) on the same object \( O_i \) accesses the same registers and object, so the sparse allocate-on-access complexity of all updates is bounded by the number of objects updated by \( \Pi \) times \( O(\log \text{CU}(\Pi)) \), or \( O(\text{SU}(\Pi) \log \text{CU}(\Pi)) \).

To this we must add any objects that are accessed in \( H \) as part of an execution of \( \text{apply}(\pi) \) for any \( \pi \) that is not an update on some \( O_i \). Consider some particular execution of \( \text{apply}(\pi) \). Any access to a register with value 1 accesses a register that has already been counted against some update operation. The only new objects accessed by \( \pi \) that are not already so counted are (a) registers containing a 0, and (b) objects \( O_i \) that are reachable through a path of ones because their siblings were updated. In each case mapping the new object to its parent gives an injective mapping from objects not already counted against updates to objects counted against updates. This at most doubles the number of objects counted, leaving us with \( \text{SA}(\Sigma) = O(\text{SU}(\Pi) \log \text{CU}(\Pi)) \) as claimed.

The second part of the lemma is a direct consequence of Lemma 4. Executing \( \text{apply}(\pi) \) for an update operation \( \pi \) on \( O_i \) accesses no object with index greater than that of \( O_i \), which is \( O(i) \). Executing \( \text{apply}(\pi) \) for a read operation \( \pi \) stops at some register or object \( X \) that is a sibling of a register or object \( X' \) already counted against an update. But then the index of \( X \) is at most one greater than the index of \( X' \), giving the same asymptotic maximum address. ◀

We can also bound the overhead in step complexity:

Lemma 7. Each call to \( \text{apply}(\pi) \) in Algorithm 1 finishes in \( O(\log i) \) steps, where \( \pi \) operates on \( O_i \).

Proof. Immediate from the fact that \( O_i \) has depth \( O(\log i) \) in the tree. ◀

The following theorem summarizes the results of this section:

Theorem 8. Any algorithm in the allocate-on-update model that uses a maximum address of \( m \) can be simulated in the allocate-on-access model, with a multiplicative overhead of:

- \( O(\log m) \) for the number of objects used,
- \( O(1) \) for the maximum address used, and
- \( O(\log m) \) for both individual and total step complexity.

Proof. The simulation just replaces the objects used by the extended implementation in Algorithm 1. That this preserves the correctness of the original algorithm follows from Lemma 3. The overheads follow from Lemma 6 and 7 ◀

We believe that these overheads are the best possible using a binary tree structure. However, using a tree with higher arity (equivalent to using a code with a larger alphabet) could produce a lower time complexity overhead at the cost of more wasted space. We do not have a lower bound demonstrating that this particular trade-off is necessary, so the exact complexity of simulating allocate-on-update in an allocate-on-access model remains open.

3 Examples of allocate-on-use space complexity

In this section, we analyze the space complexity of several recent algorithms from the literature. These include the current best known algorithms (in terms of expected individual step complexity) for implementing test-and-set [11] and consensus [3] from atomic registers.
Theorem 9. 1. Let $H$ be an execution of the RatRace algorithm for adaptive test-and-set of Alistarh et al. [2], with $k$ participants. Then $SA(H) = \Theta(SU(H)) = \Theta(k)$ with high probability, and $CA(H) = \Theta(CU(H)) = O(k^3)$ with high probability for an appropriate addressing scheme.

2. Let $H$ be an execution of the randomized test-and-set of Alistarh and Aspnes [1]. Then $SA(H)$, $SU(H)$, $CA(H)$, and $CU(H)$ are all $\Theta(\log \log n)$ with high probability.

3. Let $H$ be an execution of the randomized test-and-set of Giakkoupis and Woelfel [11]. Then $SA(H)$, $SU(H)$, $CA(H)$, and $CU(H)$ are all $\Theta(\log n)$ with high probability.

4. Let $H$ be an execution of the $O(\log \log n)$-time randomized consensus protocol of Aspnes [3]. Then $SA(H)$, $SU(H)$, $CA(H)$, and $CU(H)$ are all $\Theta\left(\log \log n + \frac{\log m}{\log \log m}\right)$ in expectation.

Proof. 1. The RatRace algorithm works by having each processes randomly select a path through a binary tree until it manages to acquire a node using a splitter [16], then fight its way back to the root by winning a 3-process consensus object at each node. Both the splitter and consensus object associated with each node require a constant number of registers to implement, so the space complexity is determined by the number of nodes in the subtree traversed by processes. The analysis of a similar algorithm for adaptive collect [6], is used to show that the size of the tree is $\Theta(k)$ with high probability, in which case $\Theta(k)$ of the $O(n^3)$ registers pre-allocated in their algorithm are used. Since each node access involves both reading and writing a constant number of objects, we have $SA(H) = \Theta(SU(H))$, and both are $\Theta(k)$.

The same analysis shows that the depth of the tree is at most $3 \log k$ with high probability, giving a bound of $O(k^3)$ on the largest address used provided nodes are arranged in breadth-first order.

It is worth noting that our model does not require pre-allocating a specific bounded address space. So in principle we can use RatRace with an unbounded number of possible processes and still get the claimed bounds as a function of $k$.

2. The Alistarh-Aspnes TAS runs the processes through a sequence of $\Theta(\log \log n)$ sifter objects, each implemented using a one-bit atomic register. The authors show that with high probability, a constant number of processes remain at the end of this sequence, which then enter a RatRace TAS object. The sifter array uses $\Theta(\log \log n)$ space in all executions. From the previous argument, the RatRace object uses $O(1)$ space with high probability in all four measures.

3. The Giakkoupis-Woelfel TAS also uses a sequence of sifter objects; these reduce the number of remaining processes to $O(1)$ in only $\Theta(\log^* n)$ rounds, but the cost is an increase in the space required for each object to $O(\log n)$. However, after the first sifter the number of remaining processes is reduce to $O(\log n)$ with high probability, so subsequent sifter objects can be implemented in $O(\log \log n)$ space. This makes the space required dominated by the initial sifter object, giving the claimed bound.

4. The Aspnes consensus algorithm uses a sequence of rounds, where each round has essentially the same structure as the Alistarh-Aspnes TAS followed by an adopt-commit object to detect termination. This produces agreement within $O(1)$ rounds on average. Using the adopt-commit of Aspnes and Ellen [4], we get $\Theta(\log \log n)$ space for each round for the sifters plus $\Theta\left(\frac{\log m}{\log \log m}\right)$ for the adopt-commit object. Multiplying by the $O(1)$ expected rounds gives the claimed bound.
Curiously, all of the variation in space usage for the test-and-set algorithms analyzed above can be attributed to RatRace, either by itself or as a backup for a faster algorithm for winnowing the processes down to a constant number. Using a worst-case measure of space complexity hides the cost of these winnowing steps behind the polynomial worst-case space complexity of RatRace. Using our measures instead exposes an intriguing trade-off between time and space complexity, where the Alistarh-Aspnes algorithm obtains $O(\log \log n)$ space complexity at the cost of $\Theta(\log \log n)$ individual step complexity, while the Giakkoupis-Woelfel algorithm pays $O(\log n)$ space complexity but achieves a much better $\Theta(\log^* n)$ individual step complexity. Whether this trade-off is necessary is an open problem.

4 Monte Carlo Mutual Exclusion

In this section, we present a Monte Carlo mutual exclusion algorithm, which uses only $O(\log n)$ registers, and against a weak adaptive adversary satisfies mutual exclusion with high probability for polynomially many passages through the critical section. This can be used directly, or can be combined with Lamport’s fast mutual exclusion algorithm [15] to give an algorithm that uses $O(\log n)$ space initially, then backs off to a traditional $O(n)$ space algorithm when the Monte Carlo algorithm fails.

A mutual exclusion algorithm provides two methods, `lock()` and `unlock()`. Each process repeatedly calls `lock()` followed by `unlock()`. When a process’ `lock()` call terminates, it is in the critical section (CS). The algorithm satisfies mutual exclusion, if for any execution, no processes are in the critical section at the same time. An infinite execution is fair, if each process that is in the CS or has a pending `lock()` or `unlock()` call either takes infinitely many steps or enters the remainder section (which happens when it is not in the CS and has no `lock()` or `unlock()` call pending). A mutual exclusion algorithm is deadlock-free, if in any infinite fair execution, each `lock()` and `unlock()` call terminates. If it is randomized, and in an infinite fair execution each `lock()` and `unlock()` call terminates with probability 1, then we call it randomized deadlock-free.

Burns and Lynch [8] proved that any deterministic deadlock-free mutual exclusion algorithm implemented from registers, requires at least $n$ of them. For fewer than $n$ registers, the proof constructs exponentially long executions such that at the end two processes end up in the CS. But there are no mutual exclusion algorithms known that use $o(n)$ registers and do not fail provided that only polynomially many `lock()` calls are made. Here we present a randomized algorithm that has this property with high probability, i.e., it uses only $O(\log n)$ registers, and in an execution with polynomially many `lock()` calls mutual exclusion is satisfied with high probability.

Our algorithm works for a weak adaptive adversary, which cannot intervene between a process’ coin flip and its next shared step. I.e., it schedules a process based on the entire system state, and then that process flips its next coin, and immediately performs its following shared memory step.

The time efficiency of mutual exclusion algorithms is usually measured in terms of remote memory references (RMR) complexity. Here we consider the standard cache-coherent (CC) model. Each processor keeps local copies of shared variables in its cache; the consistency of copies in different caches is maintained by a coherence protocol. An RMR occurs whenever a process writes a register (which invalidates all valid cache copies of that register), and when a process reads a register of which it has no valid cache copy. The RMR complexity of a mutual exclusion algorithm is the maximum number of RMRs any `lock()` and `unlock()` method
incurs. The best deterministic mutual exclusion algorithms use $O(n)$ registers and have an
RMR complexity of $O(\log n)$ [17], which is tight [5]. Randomized Las Vegas algorithms can
beat the deterministic lower bound (e.g. [7, 13, 12]), but they all use at least a linear or even
super-linear number of registers and stronger compare-and-swap primitives.

Our algorithm has an expected amortized RMR complexity of $O(n)$: In any execution
with $L$ lock() calls, the total expected number of RMRs is $O(n \cdot L)$.

### 4.1 The algorithm

Pseudocode for our Monte Carlo mutual exclusion algorithm can be found in Figure 3. Slightly
simplified, the idea is the following. We use $\Gamma = O(\log n)$ Boolean registers $S_0, \ldots, S_{\Gamma - 1}$,
which are initially 0. In a lock() call a process tries to write 1 to all registers $S_0, S_1, \ldots, S_{\Gamma - 1}$
in order, and after writing to $S_{\Gamma - 1}$ it enters the critical section. But before writing, a process
tries to find out if some other process is already ahead of it. To do so, it decides with
probability $1/2$ to read a register $S_j$ instead of writing it. If a process sees that $S_j = 0$, then
it restarts the random experiment at register $S_j$ (i.e., it randomly chooses to writes 1 to
$S_j$ or to read $S_j$). But if in its read it sees that $S_j = 1$, then it knows some other process
has already written there, so it drops back to register $S_0$, waiting there until $S_0 = 0$, upon
which it starts the procedure over. (In the pseudocode it waits for some auxiliary variable $A$
to change, which is explained below.) Once a process has reached $S_{\Gamma - 1}$ and written there,
all registers $S_0, \ldots, S_{\Gamma - 1}$ have value 1, and the process can enter the critical section. In
an unlock() call, a process simply writes 0 to all registers $S_{\Gamma - 1}, \ldots, S_0$, thus releasing all
waiting processes.

Intuitively, we achieve a low error probability, for the following reason. Consider a
situation in which $S_j = 0$ and some processes are poised to perform their next operation
on $S_j$. Only a process writing 1 to $S_j$ can continue to $S_{j+1}$. Once a first write of 1 to $S_j$
has occurred, each process will with probability $1/2$ continue to $S_{j+1}$ and with probability
$1/2$ drop back to $S_0$ and start spinning there. Thus, if the adversary schedules $k$ processes,
then roughly $k/2$ of them will move on to $S_{j+1}$, and $k/2$ will drop back. This way, the
number of processes reduces by half from one register to the next, and after $\log n$ registers
in expectation only one process is left. Then after $c \log n$ registers for some constant $c > 1$,
only one process is left with high probability.

We said that if in a lock() call a process drops to $S_0$ it waits there until $S_0 = 0$. But
in order to bound the RMR complexity by $O(n)$, we need to avoid that all processes are
spinning (busy waiting) on $S_0$. If, for example, $\Omega(n)$ processes are writing 1 to $S_0$ while $\Omega(n)$
processes are spinning on that register, then $\Omega(n^2)$ RMRs in total are incurred without any
process making progress. Such a scenario can repeat for each lock() call, and thus lead to
an amortized RMR complexity of $\Omega(n^2)$.

To avoid this, processes that dropped back to $S_0$ in the lock() call, read an auxiliary
register $A$, then they check if $S_0 = 1$, and if yes, they spin on $A$, waiting until $A$ changed. A
process that calls unlock(), after having erased all registers, changes the value of $A$. This
ensures that all waiting processes are woken up, and while waiting, each process incurs only
a constant number of RMRs for each unlock() call.

To avoid ABAs, register $A$ stores a sequence number that a process increments with each
write. Thus, for infinitely many lock() calls, the values stored in $A$ are unbounded. But if
we allow each process to call lock() at most $n^{O(1)}$ times (after which no guarantee for the
mutual exclusion property can be made anyway), then $O(\log n)$ bits suffice for $A$.

> **Theorem 10.** There is a randomized exclusion algorithm implemented from $O(\log n)$
bounded shared registers with expected amortized RMR complexity $O(n)$, such that for a polynomial number of lock() calls, the algorithm is randomized deadlock-free, and satisfies mutual exclusion with high probability.

**Class Lock($\Gamma$)**

shared:

Boolean Register $S_0, \ldots, S_{\Gamma - 1}$

Register $A$ initially ($\bot, 0$)

local:

Integers $i, seq = 0$ ($seq$ has global scope)

**Method lock()**

1. $i = 0$
2. while $i < \Gamma$
   3. Choose random $rnd \in \{R, W\}$ s.t. $\text{Prob}(rnd = W) = \frac{1}{2}$
   4. if $rnd = R$ then
      5. if $S_i = 1$ then
         6. $i = 0$
      7. end
   8. else
      9. $S_i$.write(1); $i = i + 1$
   10. end
   11. if $i = 0$ then
      12. $a = A$
      13. if $S_0 = 1$ then
         14. while $A = a$ do “nothing”
      15. end
   16. end
17. end

**Method unlock()**

18. $i = \Gamma$
19. while $i > 0$
   20. $S_{i-1}$.write(0); $i := i - 1$
21. end
22. $A$.write($\text{myID}, seq + 1$); $seq = seq + 1$

**Figure 3** Monte Carlo Mutual Exclusion

### 4.2 Proof of Theorem 10

The proof is divided into three parts. Section 4.2.1 shows mutual exclusion holds for polynomially many passages through the critical section with high probability. Section 4.2.2 shows deadlock-freedom. Section 4.2.3 gives the bound on RMR complexity.
4.2.1 Mutual exclusion

We consider a random execution of the algorithm, and let $C_t$ denote the configuration reached at point $t$, and $L_t$ denote the number of completed lock() calls at point $t$.

The idea of this part of the proof is that we define a potential function $\Phi(C_t)$ that becomes exponentially large if more than one process enters the critical section simultaneously. We then show that the expected value of $\Phi(C_t)$ is proportional to $L_t$, and in particular that it is small if few lock() calls have finished. This gives a bound on the probability that two processes are in the critical section using Markov’s inequality.

To denote the value of a local variable of a process $p$, we add subscript $p$ to the variable name. For example, $i_p$ denotes the value of $p$’s local variable $i$. To bound the probability of error, we define a potential function. The potential of a process $p \in \{1, \ldots, n\}$ is

$$\alpha(p) = \begin{cases} 
-1 & \text{if } p \text{ is poised to read in lines 12-14 and entered this section through line 6} \\
2^{p} - 1 & \text{otherwise.} 
\end{cases} \tag{4.1}$$

Hence, $\alpha(p) = -1$ if and only if $p$ is poised in lines 12-14, and prior to entering that section it read $S_p = 1$ in line 5. The potential of register index $j \in \{0, \ldots, \Gamma - 1\}$ is

$$\beta(j) = -S_j \cdot w^j \tag{4.2}$$

Finally, the potential of the system at time $t$ is

$$\Phi(C_t) = \sum_{p=1}^{n} \alpha(p) + \sum_{j=0}^{\Gamma-1} \beta(j) + (n - 1) \tag{4.3}$$

> **Lemma 11.** Suppose at some point $t_1$ process $p$ reads $S_{j_1} = 1$ in line 5, and at a point $t_2 \geq t_1$ it reads $S_{j_2} = 1$ in line 5. Then at some point $t' \in [t_1, t_2]$ either the value of $S_0$ changes from 0 to 1, or the value of $A$ changes.

**Proof.** After reading $S_{j_1} = 1$ at point $t_1$, process $p$ proceeds to execute line 13, and thus it executes lines 12-14 during $[t_1, t_2]$. Let $t' \in [t_1, t_2]$ be the point when it reads $S_0$ in line 13.

First assume $S_0 = 1$ at point $t'$. Then $p$ enters the while-loop in line 14, and does not leave the while-loop until $A$ has changed at least once since $p$’s previous read of $A$ in line 12. Hence, in that case $A$ changes at some point between $[t_1, t_2]$, and the claim is true.

Now assume $S_0 = 0$ at point $t'$. We show that $S_0$ changes from 0 to 1 at some point in $[t', t_2]$, which proves the claim. If $j_2 = 0$, then at point $t_2$ process $p$ reads $S_0 = 1$, so this is obvious. Hence, assume $j_2 > 0$. Then before point $t_2$ process $p$ must increment its local variable $i_p$ by at least one, which means it writes 1 to $S_0$ in line 9.

> **Lemma 12.** For a random execution that ends at point $t$ with $L$ lock() calls completed, $E[\Phi(C_t)] \leq 2n(L + 2)$.

**Proof.** Consider the initial configuration $C_0$ where each process is poised to begin a lock() call and all registers are 0. Then for all processes $p$, $\alpha(p) = 0$, and for all $j \in \{0, \ldots, \Gamma - 1\}$, $\beta(j) = 0$. Hence, $\Phi(C_0) = n - 1 < n$. We bound the expected value of $\Phi(C_t)$ in subsequent steps by case analysis. Whenever the adversary schedules a process $p$ that has a pending lock() or unlock() call, $p$ will do one of the following:

1. Set $S_{j_p} = 0$ in line 20;
2. Exit lines 12-14 having entered from line 5;
Choose \(\text{rand}_p\) at random in line 3 and then immediately either read \(S_{i_p}\) in line 5 or write \(S_{i_p}\) in line 9.

We will show that in cases (1), (2), (4), and (5) the expected change in \(\Phi\) is less than or equal to 0. In case (3) \(\Phi\) increases by 1. However, case (3) can only occur at most twice per process per successful lock call leading to our bound on \(\Phi(C_1)\).

(1) Suppose \(p\) sets \(S_{i_p} \leftarrow 0\) in line 20. Then \(\alpha(p)\) decreases by \(2^{\nu-1}\). If \(S_{i_p} \leftarrow 1\), then \(\beta_{i_p} \leftarrow 1\) and \(\Phi\) does not change. If \(S_{i_p} = 0\) then \(\Phi\) decreases.

(2) Next suppose \(p\) reads \(S_0 = 0\) in line 13 or reads some \(A \neq a_p\) in line 14 having entered from line 5 (i.e., \(\alpha(p) = 0\)). Then \(p\) becomes poised to execute line 3 and \(\Phi\) does not change.

(3) Next suppose \(p\) reads \(S_0 = 0\) in line 13 or \(p\) reads some \(A \neq a_p\) in line 14 having entered from line 6. \(p\) then proceeds to write \(1\) to \(S_{i_p}\) in line 9, and becomes poised to either read \(A\) in line 12 (if \(i_p = 0\)) entering this section from line 5 so \(\alpha(p)\) does not change, or becomes poised to choose \(\text{rand}_p\) at random again in line 3. In either case \(\Phi\) does not change.

(4) Next suppose \(p\) reads \(S_0 = 1\) in line 13, reads \(A\) in line 12, or reads \(A = a_p\) in line 14. Then no register gets written, \(\alpha(p)\) does not change, and \(p\)'s local variable \(i\) remains at 0, so \(\Phi\) stays the same.

(5) Finally, suppose that when \(p\) gets scheduled it chooses \(\text{rand}_p\) at random, and then it either reads or writes \(S_{i_p}\), depending on its random choice \(\text{rand}_p\). First assume \(S_{i_p} = 0\) when \(p\) gets scheduled. If \(\text{rand}_p = R\), then \(p\) reads \(S_{i_p}\) in line 5, and becomes poised to either read \(A\) in line 12 (if \(i_p = 0\)) entering this section from line 5 so \(\alpha(p)\) does not change, or becomes poised to choose \(\text{rand}_p\) at random again in line 3. In either case \(\Phi\) does not change.

If \(\text{rand}_p = W\), then \(p\) proceeds to write \(1\) to \(S_{i_p}\) in line 9, increments its local variable \(i\) to \(i_{p}' = i_p + 1\), and either enters the critical section (if \(i_{p}' = \Gamma\)), or becomes poised to make another random choice in line 3. Hence, the value of \(\alpha(p)\) increases by \(2^\nu\) (from \(2^\nu - 1\) to \(2^\nu + 1\)). Since \(S_{i_p}\) changes from 0 to 1, the value of \(\beta(i_{p}')\) decreases by \(2^{\nu+1}\). Therefore, the change of potential \(\Phi\) is 0.

Now suppose \(S_{i_p} = 1\) when \(p\) gets scheduled. If \(\text{rand}_p = R\), then \(p\) reads \(S_{i_p} = 1\) in line 5, and then immediately sets \(i\) to 0, and becomes poised to read \(A\) in line 12 entering from line 6. Thus, \(p\)'s new potential is \(-1\). No register gets written, so \(\Phi\) changes by the same amount as \(\alpha(p)\), which is \(-2^\nu\). If \(\text{rand}_p = W\), then \(p\) writes \(1\) to \(S_{i_p}\) in line 5, then increments its local variable \(i\) to \(i_{p}' = i_p + 1\), and either enters the critical section if \(i_{p}' = \Gamma\), or become poised to make another random choice in line 3. Hence, \(p\)'s potential increases by \(2^\nu\). To summarize, if \(S_{i_p} = 1\), then with probability \(1/2\) the value of \(\Phi\) increases by \(2^\nu\), and with probability \(1/2\) it decreases by \(2^{\nu+1}\). Therefore the expected value of \(\Phi\) does not change in this case.

The only time \(\Phi\) can increase in expectation is in case (3). Next we will see that for any process \(p\) this case can happen at most twice per critical section. Case (3) can only occur by entering lines 12-14 by reading \(S_{i_p} = 1\) in line 5.

By Lemma 11 we have that if process \(p\) reads \(S_{i_p} = 1\) at \(t_1\) and \(t_2 > t_1\), then the value of \(S_0\) changes from 0 to 1 or the value of \(A\) changes at some point \(t' \in [t_1, t_2]\). Let \(U_t\) be the number of completed \(\text{unlock()}\) calls, \(L_t\) be the number of completed \(\text{lock()}\) calls, and \(A_t\) be the value of \(A_{\text{seq}}\) at time \(t\). Since \(A_{\text{seq}}\) is only incremented at the end of a completed lock call, \(A_t \leq U_t\). Since an unlock call is preceded by a successful \(\text{lock()}\) call \(U_t \leq L_t\). Hence \(A_t \leq L_t\). The number of times \(S_0\) changes from 0 to 1 is also bounded by the 1 more than the number of completed \(\text{lock()}\) calls at time \(t\). Value 0 is written to \(S_0\) only once per \(\text{unlock()}\) call. Thus the number of times \(S_0\) changes from 0 to 1 is at most \(1 + U_t \leq 1 + L_t\),

\[ CVIT \ 2016 \]
Lemma 13. In any execution, at any point there exists at least one process $p_{\text{max}}$ with local variable $i_{\text{max}}$ such that $S_j = 0$ for all $j \in \{i_{\text{max}}, \ldots, \Gamma - 1\}$.

Proof. Consider any point $t$ during an execution of the mutual exclusion algorithm. Let $p_{\text{max}}$ be a process such that $i_{\text{max}}$ is maximal at that point. For the purpose of contradiction assume there is an index $j \in \{i_{\text{max}}, \ldots, \Gamma - 1\}$, such that $S_j = 1$ at point $t$. Let $p'$ be the last process that writes 1 to $S_j$ at some point $t' \leq t$. I.e.,

$$S_j = 1 \text{ throughout } (t', t).$$

Moreover, when $p'$ writes 1 to $S_j$ in line 9 at point $t'$, $i_{\text{max}} = j$, and immediately after writing it increments $i_{\text{max}}$ to $j + 1$. Since $i_{\text{max}} \leq j$ at point $t$, process $p'$ must at some later point $t^* \in (t', t)$ decrement $i_{\text{max}}$ from $j + 1$ to $j$. This can only happen when $p'$ executes line 20 while $i_{\text{max}} = j + 1$. But then $p'$ also writes 0 to $S_j$ at $t^* \in (t', t)$, which contradicts (4.5). ▷

Lemma 14. Given any reachable configuration $C$, the value of $\Phi(C)$ is non-negative.

Proof. By Lemma 13 there exists a process $p_{\text{max}}$ such that $S_j = 0$ for all $j \in \{i_{\text{max}}, \ldots, \Gamma - 1\}$. Then

$$\alpha(p_{\text{max}}) = 2^{i_{\text{max}} - 1} - \sum_{j=0}^{i_{\text{max}} - 1} 2^j \geq \sum_{j=0}^{\Gamma - 1} S_j \cdot 2^j = - \sum_{j=0}^{\Gamma - 1} \beta(j).$$

Since $\alpha(p) \geq -1$ for each other process $p$,

$$\Phi(C) = n - 1 + \sum_p \alpha(p) + \sum_{j=0}^{\Gamma - 1} \beta(j) \geq n - 1 + \sum_{p \neq p_{\text{max}}} \alpha(p) \geq n - 1 + \sum_{p \neq p_{\text{max}}} -1 = 0.$$

Lemma 15. Let $C_{\text{fail}}$ be the set of configurations where at least two processes are in the critical section. Then for all configurations $C^* \in C_{\text{fail}}$ the value of $\Phi(C^*) \geq 2^\Gamma$.

Proof. Suppose in some configuration $C^*$ two distinct processes, $p_1$ and $p_2$, are in the critical section. Then $\alpha(p_1) = \alpha(p_2) = 2^\Gamma - 1$. Since $\alpha(p) \geq -1$ for each other process, and $\beta(j) \geq -2^j$

$$\Phi(C^*) \geq (2(2^\Gamma - 1) + (n - 2) \cdot (-1)) + \left( \sum_{j=0}^{\Gamma - 1} -2^j \right) + (n - 1) = 2^\Gamma$$

Lemma 16. For $\Gamma = c \log n$, the probability that mutual exclusion is violated at any point before $\text{Lock}()$ calls completed is $O(L^2 \cdot n^{-c+1})$. 
Proof. Let $t_j$ for $j \in \{2, \ldots, L\}$ be the point when the $j$-th lock() call completes. By Lemma 12, $E[\Phi(C_{t_j})] = O(n \cdot j)$, and thus by Lemmas 14, 15 and Markov Inequality

$$
Pr \left[ C_{t_j} \in C_{\text{fail}} \right] \leq Pr \left[ \Phi(C_{t_j}) \geq 2^{\Gamma} \right] = O \left( \frac{n \cdot j}{2^{\Gamma}} \right).
$$

Mutual exclusion is violated before $L$ lock() calls complete if and only if it is violated after $\ell$ lock() calls completed for some $\ell \in \{2, \ldots, L-1\}$. The probability of that event is

$$
Pr \left[ \exists j \in \{2, \ldots, L-1\} : C_{t_j} \in C_{\text{fail}} \right] \leq Pr \left[ \exists j \in \{2, \ldots, L-1\} : \Phi(C_{t_j}) \geq 2^{\Gamma} \right]
$$

$$\leq \sum_{j=2}^{L-1} Pr \left[ \Phi(C_{t_j}) \geq 2^{\Gamma} \right] = O \left( \sum_{j=2}^{L-1} \frac{n(j+1)}{2^{\Gamma}} \right) = O \left( \frac{n \cdot L^2}{2^{\Gamma}} \right).
$$

\[ \blacksquare \]

### 4.2.2 Deadlock-freedom

Lemma 17. The algorithm is randomized deadlock-free.

Proof. Consider any point $t$ in an infinite fair execution, in which at least one process has a pending lock() call. We will show that some process enters the critical section after point $t$ with probability 1.

Suppose no process enters the critical section in $[t, \infty)$. Since unlock() is wait-free, there is a point $t_1 \geq t$ such that after $t_1$ there are no more pending unlock() calls. Hence, throughout $[t_1, \infty)$ no process writes 0 to any register $S_j$, $j \in \{0, \ldots, \Gamma-1\}$. In other words, only value 1 may get written to any register $S_j$ after point $t_1$. Since there are only a finite number of registers $S_j$, there is a point $t_2$ such that no register $S_j$, $j \in \{0, \ldots, \Gamma-1\}$, changes value after $t_2$. By Lemma 13 there is a process $p_{\max}$ such that at point $t_2$ we have $S_j = 0$ for all $j \in \{i_{p_{\max}}, \ldots, \Gamma-1\}$. Let $i^*$ be the value of $i_{p_{\max}}$ at point $t_2$. Thus,

$$S_i = \cdots = S_{i^{\Gamma}-1} = 0 \text{ throughout } [t_2, \infty). \quad (4.7)$$

If $i^* > 0$, then at $t_2$ process $p_{\max}$ is not poised to execute a shared memory operation in lines 12-14 (because $i_{p_{\max}} = i^*$ at that point). Hence, $p_{\max}$ is either poised to read in line 5 or to write in line 9. The latter is not possible, as $p_{\max}$ would eventually write 1 to $S_i$, contradicting (4.7). If $p_{\max}$ reads in line 5, then it reads 0 from $S_{i_{p_{\max}}}$, where $i_{p_{\max}} = i^* > 0$, and so it will begin another iteration of the while-loop with $i_{p_{\max}} = i^*$. Repeating the argument, $p_{\max}$ will execute an infinite number of iterations of the outer while-loop, each time choosing at random $\text{rnd} = W$, and then reading $S_i$ in line 5. This event has probability 0.

Hence, consider the case $i^* = 0$. First assume that at some point after $t_2$ some process $p$ is not poised to execute line 14. Then due to (4.7) the if-condition in line 13 remains false for $p$ throughout $[t_3, \infty)$, so $p$ executes an infinite number of iterations of the outer while-loop.

With probability 1 process $p$ will eventually in some iteration choose $\text{rnd} = W$ in line 3 and then write 1 to some register $S_j$, $j \in \{0, \ldots, \Gamma-1\}$. This contradicts (4.7) since we assumed $i^* = 0$.

Thus, throughout $[t_2, \infty)$ all processes with pending lock() calls are stuck in the inner while-loop in line 14. Consider any process $q$ stuck in the while-loop, and let $T$ be the point when it read $A$ for last time prior to becoming stuck. Let $a^*$ be the value of $A$ at point $T$. Register $A$ gets only written in line 22, and due to the increasing sequence number, the same
value never gets written twice. Hence, since \( q \) is stuck in line 14, it reads \( A = a^* \) infinitely
many times, and thus

\begin{equation}
\text{no process writes } A \text{ throughout } [T, \infty).
\end{equation}

But at some point \( T_1 > T \) and before \( q \) becomes stuck in the while-loop, it reads \( S_0 = 1 \) in
line 13. By (4.7), after \( T_1 \) some process writes 0 to \( S_0 \), and then it will eventually write to \( A \).
This contradicts (4.8). ▫

4.2.3 RMR Bound

Lemma 18. In an execution with \( L \) invoked lock() calls, the expected total number of
RMRs is \( O((n + \Gamma)L) \).

The remainder of this section is devoted to the proof of this lemma.

Let \( X_{p, \ell} \) denote the number of RMRs a process \( p \) incurs in line \( \ell \), where \( \ell \) is one of 5, 9,
12, 13, 14, 20, and 22. These are the only lines where a process executes shared memory
operations, so the total number of RMRs is obtained by summing over all \( X_{p, \ell} \).

We now consider a random execution, and condition on the event that the random
execution contains \( L \) lock() calls.

Lemma 19. For each \( j \in \{0, \ldots, \Gamma - 1\} \), each process incurs in total at most \( L + 1 \) RMRs
by reading value 0 from register \( S_j \).

Proof. Value 0 is written to \( S_j \) (in line 20) only once per unlock() call. Only the first read
by a process in the execution, or the first read following such a write of value 0 can at the
same time return 0 and incur an RMR. Now the claimed bound follows from the fact that
there are at most \( L \) lock() calls, and thus at most \( L \) unlock() calls. ▫

Lemma 20. For any process \( p \) we have

\[ X_{p, 12} + X_{p, 14} \leq L + 1 \text{ and } X_{p, 13} \leq 2(L + 1). \]

Proof. The value of \( A \) changes at most once per unlock() call, and thus at most \( L \) times
during execution \( E \). Hence, process \( p \) incurs at most \( L + 1 \) RMRs by reading \( A \). This proofs
the claimed upper bound on \( X_{p, 12} + X_{p, 14} \).

By Lemma 19, the number of RMRs incurred by \( p \)'s reads of value 0 in line 13 is at most
\( L + 1 \). If process \( p \) reads 1 from \( S_0 \) in that line, then, due to the while-loop in line 14, it
does not read \( S_0 \) again until \( A \) changed at least once since \( p \)'s preceding read of \( A \) in line 12.
In particular, for each read of value 1 from \( S_0 \), there is a distinct RMR incurred by \( p \) when
reading \( A \) in line 14. Hence, \( X_{p, 13} \leq L + 1 + X_{p, 14} \leq 2(L + 1) \). ▫

Lemma 21. \( E[\sum_p X_{p, 9}] = O((n + \Gamma)L) \)

Proof. For \( b \in \{0, 1\} \) let \( Z_b \) denote the number of times a write in line 9 (by any process)
overwrites value \( b \) with value 1. Thus,

\[ \sum_p X_{p, 9} = Z_0 + Z_1. \]  

Since each register \( S_j \) is reset to 0 only once per unlock() call, it can change from 0 to 1 at
most \( L + 1 \) times. Accounting for \( \Gamma \) registers, we obtain

\[ Z_0 \leq \Gamma(L + 1). \]
Now suppose $S_j = 1$ when process $p$ makes a random choice in line 3. With probability $1 - 1/w$ process $p$ decides to read, and if it does so, it reads $S_j = 1$. Hence, $p$ overwrites in $9\left(1 - \frac{1}{w}\right)$ process $p$ reads a register that has value 1 in expectation at most $1/(1 - 1/w) - 1 = 1/(w - 1)$ times before $p$ reads a register with value 1 in line 5. By Lemma 11 between any two such reads, either $S_0$ changes from 0 to 1 or $A$ changes, and each of these events happens at most once per $unlock()$ call. Thus, the expected number of times process $p$ writes to a register $S_j$ that has value 1 is at most $1 + 1/(w - 1) \cdot L$. Summing over all processes we obtain $E[Z_1] = O(n \cdot L)$ (recall that $Z_0 = Z_1 = 0$ if $L = 0$). Now the claim follows from (4.9) and (4.10).

Lemma 22. For any process $p$ we have

$$E[X_{p, 9}] = O((n + \Gamma)L).$$

Proof. Let $Y_p$ denote the number of times process $p$ reads a value of 1 in line 5. By Lemma 11, between any two such consecutive reads, either the value of $A$ changes, or $S_0$ changes from 0 to 1. Since $S_0$ can change from 1 to 0 at most $L$ times (once for each $unlock()$ call), it can change from 0 to 1 at most $L + 1$ times. The value of $A$ can also change at most once for each $unlock()$ call, and thus at most $L$ times. Hence, $Y_p \leq 2L + 2$.

By Lemma 19 process $p$ incurs at most $L + 1$ RMRs by reading value 0 from $S_0$ in line 5. Now suppose $j > 0$. Then $p$ reads $S_j$ only after writing 1 to $S_{j-1}$ in line 9, which contributes to $X_{p, 9}$. Because $p$ chooses to write $S_j$ (instead of reading it) with probability $1/w$, the expected number of times $p$ can read $S_j$ in consecutive iterations of the while-loop (and thus before changing $i_p$) is at most $w - 1$. Hence, for all $x$,

$$E[X_{p, 9} | X_{p, 9} = x] \leq E[Y_p | X_{p, 9} = x] + L + 1 + x(w - 1) \leq 3(L + 1) + x(w - 1)$$

Summing this conditional expectation weighted with $Pr[X_{p, 9} = x]$ over all values of $x$, yields

$$E[X_{p, 9}] \leq 3L + 3 + E[X_{p, 9}] \cdot (w - 1).$$

Now the claim follows from Lemma 21.

Proof of Lemma 18. If $L = 0$, then no process calls $lock()$ or $unlock()$, so the lemma is trivially true. Hence, we assume w.l.o.g. that $L \geq 1$. Since there are at most $L$ $unlock()$ calls in total, we have

$$\sum_p X_{p, 20} \leq \Gamma \cdot L$$

and

$$\sum_p X_{p, 22} \leq L.$$
acquire this mutex. In the even that a process does not acquire the Lamport mutex, then
our algorithm has failed; the process can then unlock the randomized algorithm and move
over to a backup algorithm to attempt to acquire a mutex there. A single 2-process mutex
algorithm (using \(O(1)\) space and \(O(1)\) time) can then be used to choose between processes
leaving the Lamport mutex and the backup mutex.

By placing the objects used by the Monte Carlo, Lamport, and 2-process mutex algorithms
at low addresses, we get:

\[ \text{Corollary 23. There is a randomized mutual exclusion algorithm with expected amortized}
\]
\[ \text{RMR complexity } O(n), \text{ such that the algorithm is randomized deadlock-free; satisfies mutual}
\]
\[ \text{exclusion in all executions; and, for a polynomial number of lock() calls, has a contiguous}
\]
\[ \text{allocate-on-access space complexity } CA = O(\log n) \text{ in expectation and with high probability.}
\]

5 Open problems

While we have started a formal approach to analyzing allocate-on-use space complexity for
shared-memory distributed algorithms, much remains to be done.

We have shown that a system that assumes the allocate-on-update model can be simulated
in the stricter allocate-on-access model with a constant increase in the largest address used
and a logarithmic increase in the number of objects used. It is not clear whether this latter
overhead is necessary, and it would be interesting to see if it could also be reduced to a
constant.

We have also demonstrated that it is possible to solve mutual exclusion for a polynomial
number of locks with logarithmic space complexity with high probability. Our algorithm
pays for its low space complexity with linear RMR complexity. Curiously, it is possible to
achieve both \(O(1)\) space and RMR complexity with high probability using very long random
delays under the assumption that critical sections are not held for long; this follows from
Lamport’s fast mutual exclusion algorithm [15] and is essentially a randomized version of
the delay-based algorithm of Fischer described by Lamport. However, this algorithm has
poor step complexity even in the absence of contention. We conjecture that there exists
a randomized algorithm for mutual exclusion that simultaneously achieves \(O(\log n)\) space
complexity, \(O(\log n)\) RMR complexity, and \(O(\log n)\) uncontended step complexity, all with
high probability assuming polynomially many passages through the critical section.

References

1 Dan Alistarh and James Aspnes. Sub-logarithmic test-and-set against a weak adversary. In
  Distributed Computing: 25th International Symposium, DISC 2011, volume 6950 of Lecture
2 Dan Alistarh, Hagit Attiya, Seth Gilbert, Andrei Giurgiu, and Rachid Guerraoui. Fast
  randomized test-and-set and renaming. In Nancy A. Lynch and Alexander A. Shvartsman,
  editors, Distributed Computing, 24th International Symposium, DISC 2010, Cambridge,
3 James Aspnes. Faster randomized consensus with an oblivious adversary. In 2012 ACM
4 James Aspnes and Faith Ellen. Tight bounds for adopt-commit objects. Theory of Com-
5 Hagit Attiya, Danny Hendler, and Philipp Woelfel. Tight RMR lower bounds for mutual
  exclusion and other problems. In Proceedings of the 40th Annual ACM Symposium on


