Efficient Randomized DCAS

George Giakkoupis, Mehrdad Jafari Giv, Philipp Woelfel

Efficient Randomized DCAS

GEORGE GIAKKOUPIS, Inria, Univ Rennes, CNRS, IRISA, France
MEHRDAD JAFARI GIV, University of Calgary, Canada
PHILIPP WOELFEL, University of Calgary, Canada

Double Compare-And-Swap (DCAS) is a tremendously useful synchronization primitive, which is also notoriously difficult to implement efficiently from objects that are provided by hardware. We present a randomized implementation of DCAS with $O(\log n)$ expected amortized step complexity against the oblivious adversary, where n is the number of processes in the system. This is the only algorithm to-date that achieves sub-linear step complexity. We achieve that by first implementing two novel algorithms as building blocks. One is a mechanism that allows processes to repeatedly agree on a random value among multiple proposed ones, and the other one is a restricted bipartite version of DCAS.

CCS Concepts: • **Theory of computation → Concurrent algorithms**.

Additional Key Words and Phrases: Shared memory, Double-Compare-And-Swap, DCAS, Randomized Algorithms, Oblivious Adversary

1 INTRODUCTION

The double compare-and-swap (DCAS or CAS2) object is one of the few fundamental shared memory primitives that have a wide variety of applications, but are currently not provided by common architectures. DCAS operations were part of the instruction set of Motorola's 68k series microprocessor, but were considered highly inefficient [16]. To the best of our knowledge, DCAS is not supported by current CPU architectures (see also [28]).

Therefore, significant effort has been made to devise software implementations from primitives available in hardware [1, 4, 7, 9, 13, 15, 17–19, 22, 24, 26, 27], such as registers and single-word compare-and-swap (CAS) objects. But none of these solutions are efficient in terms of the standard analytical step complexity measures. Most of the algorithms are lock-free (meaning that as long as some process takes sufficiently many steps, some operation will terminate), while two of the algorithms are wait-free [13, 27]. In several of these works, empirical performance comparisons are provided, while complexity bounds are given just for the basic uncontended case. However, it is not hard to see that the worst-case individual or amortized step complexity of all those algorithms is at least $\Omega(n)$ in a system with n processes. To the best of our knowledge, all known algorithms are deterministic.

In this paper we present a randomized DCAS implementation from atomic registers and CAS objects. It achieves expected amortized step complexity $O(\log n)$, against the standard oblivious adversary. I.e., a sequence of k operations on the implemented DCAS object is completed after at most $O(k \log n)$ steps in expectation. Hence, our algorithm is randomized lock-free.

A compare-and-swap (CAS) object provides the following operations: `read()`, `CAS(`old`,` new`)`, and `write(`new`)`. The `read()` and `write()` operations read and write values from/to the object.[1] Operation `CAS(`old`,` new`)` compares the current value v of the object with old, and replaces it with new, provided that $v = old$. In that case, the `CAS()` operation returns True, and we say it *succeeds*. If $v \neq old$, then the `CAS()` operation returns False, leaving the object unchanged. Nowadays, almost all common hardware architectures support atomic `CAS()` operations.

A double compare-and-swap (DCAS) object extends the CAS functionality to any pair from a set of memory locations. Precisely, for an array $D[0..m-1]$ of memory locations, it supports the operations `read(`a`)`, which returns the value of $D[a]$, and `DCAS(`$\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$`)`, which compares $D[a_0]$ with old_0 and $D[a_1]$ with old_1, and if *both* match, then it replaces the array entries with new_0 and new_1, respectively. Similar to `CAS()`, the `DCAS()` operation returns True or False to indicate whether the operation was successful. DCAS objects may or may not support `write()` operations on single memory locations, but our implementation does not (see also section *Limitations*, below). A k-CAS object has the

[1] In theoretical research, the `write()` operation is not always assumed to be available, but common hardware supporting atomic `CAS()` operations also supports atomic writes.

canonical generalized specification for k (instead of 2) memory locations. The term multi compare-and-swap (MCAS) is used for k-CAS objects, where $k \geq 2$.

Applications. The availability of DCAS objects can significantly simplify the design of concurrent algorithms, and several algorithms have been proposed that use `DCAS()` operations. Massalin and Pu [25] employed DCAS to build a lock-free operating system kernel, as well as various lock-free data structures (such as stacks, lists, queues, etc.). Greenwald [15] provided MCAS implementations using DCAS. Other examples of algorithms using DCAS include priority queues [23], double ended queues [2, 6, 15], and general k-read-modify-write primitives [9]. Attiya and Hillel [8] presented a software transactional memory system using DCAS, and state that their algorithm shows that "DCAS is, in some sense, a 'silver bullet' for highly-concurrent implementations, allowing to achieve locality properties that are difficult, maybe even impossible, to obtain using only CAS."[2]

Related Work. MCAS can easily be implemented using transactional memory, as defined by Herlihy and Moss [20], but transactional memory lacks hardware support. Israeli and Rappoport [22] presented a lock-free and disjoint-access parallel k-CAS algorithm with amortized complexity $O(n^2 \cdot k)$, in an n process system. ('Disjoint-access parallel' means that non-conflicting operations do not delay each other.) Their algorithm uses the load-linked/store-conditional (LL/SC) primitive, which is not available in hardware, but can be implemented from CAS. Attiya and Dagan [7] also presented a lock-free algorithm for general binary operations (including DCAS) from LL/SC. The performance is analyzed in terms of *sensitivity*, which is the maximum distance between operations that delay each other in the graph induced by conflicts. However, long chains of close conflicts can yield a high step complexity. Other lock-free algorithms to implement MCAS algorithms from CAS or LL/SC were proposed in [4, 5, 17–19, 26], and wait-free ones in [13, 27]. For none of these algorithms have amortized or worst-case step complexity upper bounds been stated, but it is easy to see that all of them have at least $\Omega(n)$ step complexity. For example, in the wait-free solution of [13], MCAS operations may take up to $\Omega(n^2)$ steps. Brown, Ellen, and Ruppert [10, 11] presented a specification and implementation of a multi-address LL/SC extension, called LLX/SCX from CAS. The implementations are wait-free, but this is achieved for the price of allowing operations to fail.

Model and Results. We consider the standard shared memory model, in which n processes with unique IDs in $\{1, \ldots, n\}$ communicate through atomic operations on shared objects. Our algorithms use only atomic registers and CAS objects. Processes can generate private coin flips to make random decisions. Each process runs its own algorithm. An adversary generates a schedule, by deciding at each point in time, which process takes the next step. We assume an *oblivious* adversary, which means that the schedule is independent of the random choices processes make.[3] The schedule, together with the processes' random choices, gives rise to an execution, which is simply the sequence of all steps taken by processes.

The gold standard of correctness conditions for shared memory objects is *linearizability* [21]. Linearizable objects behave in the worst-case as atomic objects: Informally speaking, anything that can happen with linearizable objects can also happen with atomic objects. Our main result is an algorithm that implements a linearizable DCAS object, supporting the operations `read()` and `DCAS()` on an array of m addresses, where $m \geq 2$ is an integer parameter of the object. For a $\texttt{DCAS}(\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle)$ operation, we require that $a_0 \neq a_1$ (naturally), and $old_i \neq new_i$ for each $i \in \{0, 1\}$.

Theorem 1.1. *There is a randomized linearizable DCAS implementation from registers and CAS objects, where each* `read()` *operation has constant step complexity, and in an execution that is scheduled by an oblivious adversary and that comprises* ℓ `DCAS()` *and* `read()` *invocations, the expected total number of steps is* $O(\ell \cdot \log n)$.

[2]Despite the fact that not everyone shares this view [12], the extensive body of literature shows that in many cases DCAS significantly simplifies the design of concurrent algorithms.
[3]However, our results hold also against some weak adaptive adversaries, which have partial view of the internal data of the object, as described in the detailed statement of our results.

We note that it is easy to implement support for (single-word) `CAS()` operations, by adding one "dummy" array entry $D[m+p-1]$ for each process p: Process p can then simulate a `CAS()` operation on $D[a]$, $a \in \{0, \ldots, m-1\}$, by performing a `DCAS()` on addresses a and $m+p-1$.

Without memory management our algorithm is not space bounded, because it uses unbounded sequence numbers, and processes need to allocate "new" base objects in each `DCAS()` operation. At any point in time, the total number of referenced objects in our algorithm is $O(m \log n)$, so if the system provides memory management, space will be bounded. There are techniques for memory management and bounding sequence numbers that do not incur asymptotic time complexity overhead (e.g., [3]). However, applying them here would distract form the core ideas and contributions.

Limitations. Our DCAS algorithm does not support write operations. We believe that it is possible to add support for writes, but leave that for future work. As already mentioned, for each `DCAS`$(\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle)$ operation and each $i \in \{0,1\}$ we require $old_i \neq new_i$. This effectively prevents double-compare single-swap (DCSS) operations, which compare the values in two memory locations, and in case of a match, change only one of them. DCSS is important for real algorithms. However, lifting the above restriction seems rather difficult, and is left as an open problems.

Outline and Ideas. We implement our DCAS algorithm from several building blocks. The first one is a new randomized object called `RepeatedChoice`, which allows processes to *propose* values and then randomly *choose* one of the proposed values at random. The probability distribution is not uniform, but it is guaranteed that if k values have been proposed, then each value is chosen with probability at most $O(1/k)$. The object is long-lived, in the sense that processes can keep proposing and randomly deciding values. We will describe the operations and semantics of the object later, but for now it suffices to know that all operations on it are wait-free and have worst-case step complexity $O(\log n)$.

We then implement a restricted randomized DCAS algorithm, called *bipartite DCAS* or *BDCAS*. It supports a `BDCAS()` operation, which is the equivalent of a `DCAS()` operation. The only restriction is that the array entries that can be modified by `BDCAS()` operations are partitioned into two disjoint parts (hence the name "bipartite"), and the two entries a `BDCAS()` operation modifies must belong to different parts. Our implementation uses the randomized `RepeatedChoice` object, which allows processes to help a randomly chosen `BDCAS()` operation among multiple conflicting ones. As a result we obtain a randomized BDCAS algorithm with $O(\log n)$ expected amortized step complexity. Our algorithm satisfies *strong linearizability* [14], a correctness property that is stronger than linearizability (see Section 2 for a definition).

Finally, we show how to implement a randomized DCAS algorithm from any strongly linearizable BDCAS algorithm, such that all operations can complete in an expected constant number of steps on the BDCAS object and registers.

While linearizable objects preserve the properties of atomic objects in deterministic algorithms, this is not the case for randomized algorithms. *Strong* linearizability is suitable to preserve properties of objects used in randomized algorithms that are scheduled by a (strong) adaptive adversary [14]. (Such an adversary can take all past coin flip results into account when making scheduling decisions.) But this is not the case in the oblivious adversary model considered here. Therefore, even though our BDCAS algorithm is strongly linearizable, we cannot treat it as atomic in our DCAS analysis. Nevertheless, we exploit the fact that the BDCAS implementation is strongly linearizable, and not just linearizable. To the best of our knowledge this is the first time that strong linearizability has been useful when composing algorithms that are scheduled by a different adversary than the strong adaptive one.

2 DEFINITIONS AND NOTATION

Linearizability, defined by Herlihy and Wing [21], is the gold standard of correctness conditions. Consider an implemented object O and an execution E on O (an execution comprises invocations and responses of operations on O and on the base objects used to implement O). For a method call op on O, we call $inv(op)$ the point of the invocation of op and $rsp(op)$ the point of the response of op. If op does not respond, we define $rsp(op) = \infty$.

A *linearization* of E is a sequential execution E' that is valid with respect to the sequential specification of O, and can be obtained in the following way: Each operation op on O in E is assigned a *linearization point* $lin(op) \in [inv(op), rsp(op)]$

(if $rsp(op) = \infty$, then $lin(op) = \infty$ is allowed), and E' is the sequence of operations op with $lin(op) < \infty$, ordered by their linearization points. An execution is linearizable, if it has a linearization, and an object is linearizable if all executions on it are linearizable.

Implemented, linearizable objects preserve the (worst-case) properties of atomic objects in deterministic algorithms. Golab, Higham, and Woelfel [14] observed that this is not the case in randomized algorithms, and proposed a stronger notion, *strong linearizability*. An object O is *strongly linearizable*, if there is a *prefix preserving* function f that maps each execution E on O to a linearization. If an object is strongly linearizable, then it preserves the properties of its atomic counterpart in randomized algorithms that are scheduled by the (strong) adaptive adversary. The same is not true for the oblivious adversary, for which our results hold. Nevertheless, our construction of DCAS uses (and its analysis requires) a strongly linearizable BDCAS.

Notation. We use the standard point operator '.' to access the fields of an object and the methods it supports, e.g., $o.x$ is field x of object o, and $o.f()$ invokes o's method $f()$. Operator `new` creates a new object, and returns a *reference* to the object. E.g., $r \leftarrow$ `new` $T(a, b, \ldots)$ creates a new object of type T whose fields are initialized to values $a, b, \ldots$, and a subsequent operation $r' \leftarrow r$ results in r' referencing the same object. We will use operator '.' to also access the fields and methods of a referenced object. Thus, in the last example above, $r.x$ is the x field of the object, and is the same variable as $r'.x$. We will often drop the distinction between a reference to an object and the object itself, when there is no danger of confusion or when the distinction is not important.

We use subscript t to denote the value of a shared variable at point t. If at t an operation is executed that changes the value of a shared variable y from a to $b \neq a$ (namely, a write or `CAS()` operation), then we define $y_t = b$.

For any execution E and $t \geq 0$, we denote by E_t the execution prefix of E in the interval $[0, t]$.

3 REPEATED CHOICE

In this section we describe the type `RepeatedChoice` and a randomized implementation, which serves as a building block in our randomized BDCAS algorithm. An object of this type allows processes to *propose* values, and then to agree on a randomly chosen *agreement-value* among the proposed ones. In order to allow this to happen repeatedly, an agreement-value remains fixed (we call it *locked*) until some process unlocks that agreement-value.

In the following we give a sequential specification. It is trivial to implement a linearizable object with these properties. However, our implementation also provides some non-trivial probabilistic guarantees that are not part of the sequential specification, and which we will describe later.

Let V be some set and $\perp$ a value with $\perp \notin V$. An object L of type `RepeatedChoice` stores an *agreement-value* in $V \cup \{\perp\}$. Moreover, L can be in one of two locking states, *locked* or *unlocked*. Initially, L's agreement-value is $\perp$ and L is unlocked. The object supports the methods `propose(`v`)` for $v \in V$, `choose&lock()`, `unlock(`u`)` for $u \in V \cup \{\perp\}$, and `read()`. We say a process *proposes* v when it calls L.`propose(`v`)`. We require that throughout an execution, no value gets proposed twice, i.e., if there are two method calls L.`propose(`v`)` and L.`propose(`v'`)`, then $v \neq v'$. This can easily be achieved by augmenting each proposed value with a pair comprising the proposer's process ID and a sequence number.

Method `propose(`v`)` does not change the agreement-value of L or its locking state, and it does not return anything; its semantics is defined through `choose&lock()` calls.

Method `choose&lock()` does not modify L, if L is locked. If L is unlocked, a `choose&lock()` call locks L, and replaces the old agreement-value of L with some $v \in V \cup \{\perp\}$ such that

(i) $v = \perp$, or
(ii) v was previously proposed and is different from all earlier agreement-values of L.

We say that the `choose&lock()` call is *successful* in this case. Method `unlock(`v`)` does not modify L, if L is unlocked or L's agreement-value is not v. If L is locked and its agreement-value is v, then `unlock(`v`)` unlocks L; we say that the call is

successful in this case. Methods `choose&lock()` and `unlock()` do not return anything. Last, method `read()` just returns the agreement-value of L.

Our implementation also has the following *probabilistic* properties: Suppose k values are proposed between two subsequent successful `unlock()` calls u_1 and u_2. Suppose also that d is the last successful `choose&lock()` operation before u_1, and d^* is the first successful `choose&lock()` operation after u_2. Then d^* chooses each of the values proposed between d and d^* with probability at most $O(1/k)$, and it chooses $\perp$ with probability at most 2^{-k}, while older values, proposed before d, have zero probability to be chosen. (This implies that if no values have been proposed, then $\perp$ is chosen.)

The exact probabilistic properties are described in Theorem 6.11. In addition, our implementation is wait-free and has $O(\log n)$ worst-case step complexity.

3.1 Implementation

Figure 1 shows the pseudocode of our `RepeatedChoice` implementation.

The idea is the following. Let $\lambda = \lceil \log n \rceil$. We use a two-dimensional array $C[a][b]$, where $a \in \{0, 1\}$ and $b \in \{1, \ldots, \lambda\}$. We refer to $C[0][]$ and $C[1][]$ as the two "sides" of C. To propose a value v, a process simply writes v to an array entry $C[a][b]$, where a is chosen uniformly at random and b is geometrically distributed.

The status of the `RepeatedChoice` object is stored in a CAS object S. Object S has 3 fields, val, i, and ℓ. The first field, val, stores the agreement-value of L. The third field, ℓ, is a sequence number that gets incremented with every successful `choose&lock()` or `unlock()` call, and which is odd if the object is locked, and even otherwise. Finally, i is a bit that describes the side of C that will get erased in the next successful `unlock()` call, and the value of i is flipped after that call.

The `read()` implementation is straightforward: A process simply returns the first component of S.

When a process p calls `choose&lock()`, it reads (val, i, ℓ) from S (line 3). If S is unlocked (i.e., ℓ is even), then p scans through all array entries $C[i][j]$, $j = 1, \ldots, \lambda$, searching for the last non-$\perp$ entry v' (lines 4–8). If it finds no non-$\perp$ entry then $v' = \perp$. Then p tries to choose value v' and lock the object, using a `CAS()` operation on S (line 9). Note that, because of the geometric distribution used in `propose()` calls, all values written to $C[i][]$ in the interval before the `choose&lock()` call and after the last successful `unlock()` call on side i, are roughly equally likely to be the last non-$\perp$ entry. Values written outside that interval are at most as likely, and, in particular, those written before the last successful `unlock()` call on side i will be overwritten before the `choose&lock()` call (as discussed next).

In an `unlock(`val`)` call, process p first reads (val', i, ℓ) from S (line 10). If the object is locked (ℓ is odd) and $val' = val$, then the process erases all the array entries in $C[i][j]$ for $j \in \{1, \ldots, \lambda\}$, by writing $\perp$ to them (lines 11–15). The mechanism in lines 13–15 ensures that no array entry $C[i][j]$ gets erased anymore, once the value of S has changed, i.e., once the object got locked by some other process. Over two consecutive successful `unlock()` calls all values $C[i][j]$ will get erased at least once, and so no outdated values can be chosen anymore.

Note that if a process proposes a value, it is possible that value gets erased immediately by an ongoing `unlock()` call. However, if the proposing process chooses the right side of array C to write to, then this won't happen. Since processes choose their side at random, they have at least a 1/2 probability of choosing a side that does not get erased.

It is obvious from the pseudo code in Figure 1 that the algorithm is wait-free and has $O(\log n)$ worst-case step complexity. We prove the correctness and analyze the probabilistic properties of our implementation in Section 6.

4 BIPARTITE DCAS

A BDCAS object stores an array $B[0..m-1]$ of values from $D \cup \{\perp\}$, where D is some set and $\perp \notin D$. The initial value of each array entry is $\perp$. The object is parameterized by two sets $M_0 \subset [m] = \{0, 1, \ldots, m-1\}$, and $M_1 = [m] \setminus M_0$. It supports operation `BDCAS(`$\langle a_0, old_0, new_0 \rangle, \langle a_1, old_1, new_1 \rangle$`)`, where $a_i \in M_i$, $old_i \in D \cup \{\perp\}$, and $new_i \in D$, for $i \in \{0, 1\}$. If $B[a_i] = old_i$ for each $i \in \{0, 1\}$, then the `BDCAS()` operation changes the value of $B[a_i]$ to new_i, for both $i \in \{0, 1\}$. Otherwise, the object's value remains unaffected by the `BDCAS()` operation. The operation does not return anything. (It is not difficult to add return values indicating success or failure, but this would further complicate our code. Our end goal

Shared Data:

- $C[i][j]$, for $i \in \{0,1\}$ and $j \in \{1,\ldots,\lambda\}$, where $\lambda = \lceil \log n \rceil$, is a CAS object that stores a value from $V \cup \{\perp\}$, and is initially $\perp$
- S is a CAS object that stores a triple from $(V \cup \{\perp\}) \times \{0,1\} \times \mathbb{N}$, and initially is $(\perp, 0, 0)$

Method propose($v \in V$)

```
   // Requirement: no two calls of this method use the same argument v
1  Choose α ∈ {0,1} uniformly at random, and β ∈ {1,...,λ} such that Pr[β = j] = π_j, where π_j = 2^{-j} for
     j ∈ {1,...,λ−1}, and π_λ = 2^{-λ+1}
2  C[α][β] ← v
```

Method choose&lock()

```
3  (val, i, ℓ) ← S
4  if ℓ mod 2 = 0 then
       // Object is unlocked
5      v ← ⊥
6      for j ← 1,...,λ do
7          v' ← C[i][j]
8          if v' ≠ ⊥ then v ← v'
9      S.CAS((val, i, ℓ), (v, i, ℓ + 1))                    // Linearization point if successful
```

Method unlock(val)

```
10 (val', i, ℓ) ← S
11 if ℓ mod 2 = 1 and val = val' then
       // Object is locked
12     for j ← 1,...,λ do
13         v ← C[i][j]
14         if S = (val', i, ℓ) then
15             C[i][j].CAS(v, ⊥)
16     S.CAS((val, i, ℓ), (val, 1 − i, ℓ + 1))              // Linearization point if successful
```

Method read()

```
17 (val, i, ℓ) ← S
18 return val
```

Fig. 1. RepeatedChoice implementation

is the DCAS algorithm, whose DCAS() operations do provide return values. And this algorithm does not rely on return values of BDCAS() operations.) The object also supports operation read(a), where $a \in M$, which returns the value of $B[a]$.

Our BDCAS implementation has the following *irreflexivity requirement*. Let $a \in [m]$. Each execution induces a binary relation $\lhd_a$ on $D \cup \{\perp\}$, where $x \lhd_a y$ if and only if there is a BDCAS() call using the argument triple $\langle a, x, y\rangle$. Let $\prec_a$ denote the transitive closure of $\lhd_a$. It is required that for each $a \in [m]$, the relation $\prec_a$ is irreflexive, i.e., there is no $x \in D \cup \{\perp\}$ with $x \prec_a x$. Note that $\prec_a$ is irreflexive if and only if $\lhd_a$ is acyclic. The irreflexivity requirement can easily be achieved by adding sequence numbers, provided that no BDCAS() calls with argument triples of the form $\langle a, x, x\rangle$ are allowed.

4.1 Implementation

In Figure 2, we present a randomized strongly linearizable implementation of a BDCAS object.

A Task object stores 7 variables: $stat \in \{\perp, \text{True}, \text{False}\}$, $add_i \in M_i$, and $old_i, new_i \in D \cup \{\perp\}$, for $i \in \{0, 1\}$. We use the new operator, described in Section 2, to create new Task objects. Precisely, operation new Task$(s, a_0, o_0, n_0, a_1, o_1, n_1)$ creates a new object, with $stat = s$, $add_i = a_i$, $old_i = o_i$, and $new_i = n_i$, for $i \in \{0, 1\}$, and returns a *reference* to that object. All objects created this way are distinct. We will use '*task*' and '*task reference*' as shorts for 'Task object' and 'reference to a Task object', respectively.

Our BDCAS implementation uses an array $A[0..m-1]$, where each entry is a task reference. Initially, for each $i \in \{0, 1\}$ and $j \in M_i$, $A[j]$ stores a reference to the *initial task* λ_j, where $\lambda_j.stat = \text{True}$, $\lambda_j.add_i = j$, $\lambda_j.old_i = \lambda_j.new_i = \perp$, and $\lambda_j.add_{1-i}$ can have an arbitrary value. In addition, we use an array L that consists, for each $j \in M_0$, of a RepeatedChoice object $L[j]$. The domain V of the RepeatedChoice objects is the set of task references.

As we will show in Claim 7.1, at any point each array entry $A[\alpha]$, $\alpha \in [m]$, stores a task reference. If $A[\alpha] = \tau$ and $\alpha \in M_i$, where $i \in \{0, 1\}$, then we define the *interpreted value* of $B[\alpha]$ to be $\tau.new_i$ if $\tau.stat = \text{True}$, and $\tau.old_i$ otherwise.

4.2 High Level Idea

For the purpose of this description, we call the portion of the array with array entries $A[a]$, $a \in M_0$, the *left side* of A, and the portion of the remaining array entries the *right side*. We do the same for the array B.

A lock-free deterministic implementation of BDCAS can be achieved as follows. At any point, each array entry $A[a \in M_i]$, $i \in \{0, 1\}$, stores a task reference τ, where $\tau.add_i = a$. Recall that the task's status field, $\tau.stat$, indicates if the interpreted value of $B[a]$ is $\tau.old_i$ (if $\tau.stat = \text{False}$) or $\tau.new_i$ (if $\tau.stat = \text{True}$). The status field allows us to *simultaneously* change the interpreted value of two array positions, $B[\alpha_0]$ and $B[\alpha_1]$, from $\tau.old_0$ to $\tau.new_0$ and from $\tau.old_1$ to $\tau.new_1$, respectively, if $A[\alpha_0]$ and $A[\alpha_1]$ both store task τ.

To perform operation read(a), the calling process simply reads the task τ currently stored in $A[a]$, then the status field of τ, $\tau.stat$, and finally returns $\tau.new_i$ if $\tau.stat = \text{True}$ and $\tau.old_i$ otherwise.

Now suppose process p wants to perform a BDCAS$(\langle a_0, v_0, v_0' \rangle, \langle a_1, v_1, v_1' \rangle)$ operation. First it checks if the interpreted values of $B[a_0]$ and $B[a_1]$ match the expected values v_0 and v_1, respectively, by performing read() operations as described above. If at least one of the values does not match, then the BDCAS() can return immediately (it fails). Hence, assume that both values match.

Then process p creates a task τ whose field values correspond to p's BDCAS() arguments (i.e., $\tau.add_i = a_i$, $\tau.old_i = v_i$, and $\tau.new_i = v_i'$ for each $i \in \{0, 1\}$, and $\tau.stat = \perp$). Process p's goal is now to place its task τ in the array entries $A[a_0]$ and $A[a_1]$. This must happen without at first changing the interpreted values of $B[a_0]$ or $B[a_1]$. Once τ is put into both array entries, p can change $\tau.stat$ to True.

We say a task is *finalized*, if its status is either True or False (i.e., it is not $\perp$). Our algorithm ensures that once a task is finalized, its status does not change anymore. To place a task into $A[a_i]$, $i \in \{0, 1\}$, process p reads the task τ' stored in that array entry *before* it performs its initial check whether the BDCAS() expected values v_0 and v_1 match. It then performs a helping mechanism (which we will describe later) that ensures task τ' is finalized. Thus, the only way the interpreted value of $B[a_i]$ can change afterwards is if task τ' gets replaced by a different task in $A[a_i]$. Now, to place its task τ into $A[a_i]$, process p can simply perform an $A[a_i]$.CAS(τ', τ) operation. If that CAS() succeeds, then τ has been put into $A[a_i]$ without changing the interpreted value of $B[a_i]$ (which remains $v_i = \tau.old_i$).

If p manages to place its task τ into both array entries, $A[a_0]$ and $A[a_1]$, it can simply change $\tau.stat$ to True, and both interpreted values of $B[a_i]$, $i \in \{0, 1\}$, change simultaneously from $v_i = \tau.old_i$ to $v_i' = \tau.new_i$. Hence, p's BDCAS() linearizes (and is successful).

Shared Data:

- $A[j]$, for $j \in [m]$, is a CAS object storing a task reference, initialized by λ_j.
- $L[j]$, for $j \in M_0$, is a RepeatedChoice object that stores a task reference or $\perp$

Method BDCAS($\langle a_0, old_0, new_0 \rangle, \langle a_1, old_1, new_1 \rangle$)

```
19 while True do
20     γ0 ← finish(a0)
21     if read(a0) ≠ old0 or read(a1) ≠ old1 then return
22     γ ← new Task(⊥, a0, old0, new0, a1, old1, new1)
23     L[a0].propose(γ)
24     L[a0].choose&lock()
25     γ' ← L[a0].read()
26     if A[a0] = γ0 then
27         if γ' ≠ ⊥ and γ'.old0 = old0 then
28             A[a0].CAS(γ0, γ')
29         finish(a0)
30         L[a0].unlock(γ')
```

Method finish($a_0 \in M_0$)

```
31 γ0 ← A[a0]
32 a1 ← γ0.add1
33 γ1 ← A[a1]
34 γ1.stat.CAS(⊥, True)
35 if γ0.old1 = γ1.new1 then
36     A[a1].CAS(γ1, γ0)
37     γ1 ← A[a1]
38     γ1.stat.CAS(⊥, True)
39 γ0.stat.CAS(⊥, False)
40 return γ0
```

Method read($a \in [m]$)

```
41 γ ← A[a]
42 Let i ∈ {0, 1} be such that a ∈ Mi
43 if i = 1 then
44     γ.stat.CAS(⊥, True)                          // Help the task finish
45 if γ.stat = True then
46     return γ.newi
47 return γ.oldi
```

Fig. 2. BDCAS Implementation

But p may not manage to put its tasks in one of the array entries, if the corresponding CAS() operation on $A[a_i]$ fails, because some other task τ^* has already been put in there. The standard lock-free approach would now be to help τ^* make progress. This can lead to a long chain of helping, and requires care that no cyclic helping is encountered.

In our bipartite case, things are easier: Each process first tries to put its task into the "left" array entry, $A[a_0]$, before it tries to put it also into $A[a_1]$. This way, if any task is successfully put into a right array entry, it is bound to succeed.

Thus, if a process p fails to put its task into the right side $A[a_1]$, because that side was taken by some other task τ^*, then some other successful BDCAS() operation (which created τ^*), with the same right side, will have succeeded once τ^* is finalized, and thus the interpreted value of $B[a_1]$ will have changed from v_1 to a different value. As a result, p's BDCAS() can terminate after helping to finalize τ^* (which means just changing $\tau^*.stat$ to True). This way long chains of helping and cyclic helping are avoided.

A process p may still not make progress, if it fails to put its task into the left side, $A[a_0]$. If that happens, it has to help the task τ^* that caused p's attempt to fail, but again, since τ^* has already been put into the left side, only the right side of τ^* remains to deal with. However, τ^* may ultimately fail (because it cannot be put in the right side), in which case none of the interpreted values of B at the addresses that p is interested in may have changed. In that case, p's only option is to start over. This may lead to high step complexity, in the case of high contention on a left side location; whereas high contention on right side locations does not contribute to high step complexity.

To deal with that, we use a mechanism to choose a task τ^* at random among concurrent tasks that use the same left side address. To that end, we use for each left address $a_0 \in M_0$ a RepeatedChoice object $L[a_0]$. Consider the processes that are concurrently performing BDCAS() operations with the same left address a_0. Instead of directly trying to put their tasks into $A[a_0]$, they propose them on $L[a_0]$, and choose a random task τ^* among all the proposed ones. Then each process helps τ^*, by first putting it into $A[a_0]$, and then trying to put it into $A[\tau^*.add_1]$. As discussed earlier, if τ^* is successfully put into its desired right side array entry, the interpreted value of $B[a_0]$ changes eventually. Hence, all processes helping τ^* can finish their BDCAS() operation (the one that created task τ^* succeeds, and all the others fail). But if τ^* cannot be successfully put into $A[\tau^*.add_1]$, then all processes helping τ^* have to start over.

To see why that achieves low amortized complexity, assume for simplicity that there are k processes that concurrently perform a BDCAS() with the same left side address, a_0. Let $a_{1,1}, \ldots, a_{1,k}$ be the right side addresses of these BDCAS() operations. Since a task τ^* is chosen at random, all k processes will help trying to put that task into the same right side address $a_{1,j} = \tau^*.add_1$. Suppose that ℓ of the right side array entries, say $A[a_{1,1}], \ldots, A[a_{1,\ell}]$, change before the first attempt is made to put $\tau^*.add_1$ into $A[a_{1,j}]$. Since j is chosen at random, the attempt to put $\tau^*.add_1$ into $A[a_{1,j}]$ fails with probability ℓ/k (assuming, for the ease of this discussion, a uniform distribution). But in that case, at least ℓ BDCAS() operations are successful, namely the ones whose tasks were put in the right side locations $A[a_{1,1}], \ldots, A[a_{1,\ell}]$. If, on the other hand, $\tau^*.add_1$ is successfully put into $A[a_{1,j}]$, then all k BDCAS() operations with left side address a_0 can make progress (all but one of them will fail). Hence, among the $k + \ell \leq 2k$ many BDCAS() operations that we considered, the expected number making progress is $\frac{\ell}{k} \cdot \ell + \frac{k-\ell}{k} \cdot k = \Omega(k)$. Thus, in expectation, a constant fraction of all BDCAS() operations succeed.

4.3 Detailed Algorithm Description

4.3.1 The Finish Method. In addition to the methods BDCAS() and read(), we implement a helper method, finish(a_0). Processes call this method to help another task stored in $A[a_0]$. This method achieves the following: If a process p calls finish(a_0) when a task γ_0 is stored in $A[a_0]$ (more precisely, it reads γ_0 from $A[a_0]$ in line 31), then by the time the finish() call returns, either γ_0 has been put into $A[\gamma_0.add_1]$ and $\gamma_0.stat$ = True, or the interpreted value of $B[\gamma_0.add_1]$ has changed, and $\gamma_0.stat$ = False. The method returns (in line 40) the task γ_0 that it "finished".

In lines 31–34, first p reads γ_0 from $A[a_0]$, then γ_1 from $A[a_1]$, where $a_1 = \gamma_0.add_1$, and next it tries to change $\gamma_1.stat$ to True using a $\gamma_1.stat$.CAS($\perp$, True). Since $\gamma_1.stat$ is stored in the right side, it is also stored in the left side at this point. It is guaranteed that after p's CAS() operation $\gamma_1.stat$ = True, so the BDCAS() operation that created γ_1 linearized already, and the interpreted value of $B[a_1]$ changed to $\gamma_1.new_1$ at that linearization point.

Next, p checks in line 35, if $\gamma_0.old_1 = \gamma_1.new_1$. If that is not the case, then either some other process has already put γ_0 into $A[a_1]$ and changed its status, or the BDCAS() operation that created γ_0 is bound to fail. The latter is the case because the interpreted value of $B[a_1]$ changed during the BDCAS() call from its expected value, $\gamma_0.old_1$, to a different value. Hence,

in line 39 process p executes $\gamma_0.stat$.CAS($\perp$, False), which changes the status of γ_0 to False in that case (but fails if γ_0 was put into $A[a_1]$).

Now assume that $\gamma_0.old_1 = \gamma_1.new_1$, and p evaluates the if-condition in line 35 to True. Then p tries to put γ_0 in the desired right location $A[a_1]$, using a $A[a_1]$.CAS(γ_1, γ_0) operation in line 36. If the CAS() is successful, it does not change the interpreted value of $B[a_1]$, because $\gamma_0.stat = \perp$ and $\gamma_1.stat$ = True at this point, and so the interpreted value remains $\gamma_1.new_1 = \gamma_0.old_1$. To make sure the status of γ_0 changes to True if γ_0 is successfully put into $A[a_1]$, in lines 37–38 p reads a task γ_1 from $A[a_1]$ and performs $\gamma_1.stat$.CAS($\perp$, True). After that, in line 39, p also executes $\gamma_0.stat$.CAS($\perp$, False). This CAS() can only succeed if neither p nor any other process managed to put γ_0 into $A[a_1]$.

4.3.2 The Read Method. In the read(a) method, the calling process p determines the interpreted value of $B[a]$ in a straightforward manner. First, process p reads the current task γ from $A[a]$ in line 41. If a is a right side address, then γ is bound to succeed, so p changes its status to True in line 44. (This is not necessary for linearizability, but for the desired randomized step complexity. A task that is put into its right side location may prevent other processes from making progress until these processes observe that the interpreted value of the corresponding address has changed.) After that, in line 46 process p simply determines and returns the interpreted vale of $B[a]$ based on the current status of γ. We will prove (see Lemma 7.7) that when the status of γ changes, γ is stored in the relevant locations of A. Hence, the read() operation can either linearize when $\gamma.stat$ changes for the first time after the read() invocation, or when p reads $\perp$ from $\gamma.stat$ in line 46.

4.3.3 The BDCAS Method. Suppose that process p calls method BDCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$). The process repeats the while-loop in lines 19–30 until it observes in line 21 that one of the interpreted values of $B[a_0]$ and $B[a_1]$ does not match the expected values old_0 and old_1, respectively, of its BDCAS() operation.

First, in line 20 p finishes the task γ_0 currently stored in $A[a_0]$. Then it checks if the interpreted values of $B[a_0]$ and $B[a_1]$ match the expected values old_0 and old_1, respectively, and if not the BDCAS() operation returns. Since we don't require a BDCAS() operation to return a value that indicates whether it succeeded (even though that would not be hard to achieve), it is always correct to return after one of the affected interpreted values has changed at some point t. The linearization point can always be immediately before or after t, if the BDCAS() is unsuccessful, or at point t if it succeeds.

In line 22 process p creates a new task γ whose status is $\perp$, and whose other fields match the arguments of its BDCAS() operation. Then, in lines 23–25, p first proposes task γ on the RepeatedChoice object $L[a_0]$, and next calls $L[a_0]$.choose&lock() to provide the possibility that a random task is chosen on $L[a_0]$ (but recall that $L[a_0]$ may also end up with agreement-value $\perp$). Finally, p reads the agreement-value γ' of $L[a_0]$. If γ' is a task, then p will now help it, by trying to put it in the left location $A[a_0]$.

In line 26, p checks if the task stored in $A[a_0]$ is still the same task γ_0 that it finished earlier. If not, p's attempt to help γ' fails, and p starts another iteration of the while-loop. Now suppose that $A[a_0]$ is still γ_0 when p executes line 26. In line 27, p checks to make sure that $\gamma' \neq \perp$ and $\gamma'.old_0 = old_0$, before trying to put γ' into its left side location using $A[a_0]$.CAS(γ_0, γ') in line 28. The check $\gamma'.old_0 = old_0$ is required for the following reason: The task γ' that p read from $L[a_0]$ may be completely unrelated to p's expected value old_0, because γ' may have been proposed on $L[a_0]$ a long time before it was chosen on $L[a_0]$. But if $\gamma'.old_0 = old_0$ and γ_0 is still stored in $A[a_0]$ (and thus the CAS() in line 28 may succeed), then the interpreted value of $B[a_0]$ is equal to $\gamma'.old_0$. Hence, if p's CAS() in line 28 succeeds, it does not change that interpreted value.

After that, p finishes the task now stored in $A[a_0]$ by calling finish(a_0) in line 29, and then it calls $L[a_0]$.unlock(γ') in line 30, unlocking $L[a_0]$ if it hasn't already been unlocked by some other process. The finish(a_0) call in line 29 ensures that every task γ' that is successfully put into $A[a_1]$, is finished before any process calls $L[a_0]$.unlock(γ'). On the other hand, even if γ' is not put into $A[a_1]$, it is now finalized (i.e., its status is False).

The detailed analysis is given in Section 7.

5 GENERAL DCAS

A DCAS object stores an array $D[0..m-1]$ of values from $\Delta \cup \{\perp\}$, where Δ is a set and $\perp \notin \Delta$. The initial value of each array entry is $\perp$. The object supports the operation DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$), where $a_i \in [m]$, $old_i \in D \cup \{\perp\}$, and $new_i \in D$, for $i \in \{0,1\}$. If for each $i \in \{0,1\}$, $D[a_i] = old_i$, then the DCAS() operation changes the value of $D[a_i]$ to new_i, for both $i \in \{0,1\}$, and returns True. Otherwise, the object's value remains unaffected by the DCAS() operation, and the operation returns False. The object also supports operation read(a) for $a \in [m]$, which returns the value of $D[a]$.

For each operation DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$), we require $a_0 \neq a_1$, and as in BDCAS, for the triplets $\langle a_i, old_i, new_i\rangle$ we require that the relation $\prec_{a_i}$ defined in Section 4 is irreflexive.

5.1 Implementation

In Figures 3 and 4, we present a randomized linearizable implementation of a DCAS object.

As in the BDCAS implementation, we use a Task data structure. In this case, the structure has two additional fields, the bits b_0, b_1, and the *stat* field is binary. Precisely, a Task object stores the variables $stat \in \{\text{True}, \text{False}\}$, $add_i \in [m]$, $old_i, new_i \in \Delta \cup \{\perp\}$, and $b_i \in \{0,1\}$, for $i \in \{0,1\}$. As before, we use the new operator to create a new Task object.

The DCAS implementation uses a BDCAS object B, with $2m$ addresses partitioned into sets $M_0 = \{2a : a \in [m]\}$ and $M_1 = \{2a+1 : a \in [m]\}$. Hence, a BDCAS() operation always acts on a pair of entries of B where one entry has an even address and the other an odd. For the ease of notation in the DCAS pseudo-code, we do not require that the two argument triples passed as parameters to a BDCAS() call always follow the same order as in Section 4. Instead, w.l.o.g. they can be ordered arbitrarily. Obviously, it is trivial to determine which of the two addresses is in M_0 and which one is in M_1, and to order them. Each entry of B is a pair (τ, c), where τ is a task reference, and $c \in \{0,1\}$ is a *control bit*.

The initialization of object B is as follows. For any $a \in [m]$, let ζ_a be a task where $\zeta_a.stat = \text{True}$, $\zeta_a.add_0 = a$, $\zeta_a.add_1 = (a+1) \bmod m$, $\zeta_a.old_i = \zeta_a.new_i = \perp$, and $\tau_a.b_i$ can be arbitrary, for $i \in \{0,1\}$. Then the initial value of $B[2\alpha]$ and $B[2\alpha+1]$ is $(\zeta_\alpha, 1)$.

If at some point pair (τ, c) is stored in entry $B[2a]$ or $B[2a+1]$, where $a \in [m]$, then our implementation guarantees that $\tau.add_i = a$, for some $i \in \{0,1\}$. Thus τ can be stored in up to four different entries $B[2\tau.add_i]$ and $B[2\tau.add_i + 1]$, for $i \in \{0,1\}$.

For any task τ such that $\tau.add_i = a$ for some $i \in \{0,1\}$, we define $val_a(\tau) = \tau.old_i$ if $\tau.stat = \text{False}$, and $val_i(\tau) = \tau.new_i$ if $\tau.stat = \text{True}$. The following invariant is maintained at all points, for all $a \in [m]$. If pairs (τ, c) and (τ', c') are stored at $B[2a]$ and $B[2a+1]$, respectively, then $val_a(\tau) = val_a(\tau')$. Moreover, if $\tau.stat = \text{True}$ then $c = 1$, and similarly for τ'.

Each entry $D[a]$, $a \in [m]$, of the implemented DCAS object D corresponds to the two entries $B[2a]$ and $B[2a+1]$ of the BDCAS object. The *interpreted value* of $D[a]$ is equal to $val_a(\tau) = val_a(\tau')$, where τ and τ' are the tasks stored in $B[2a]$ and $B[2a+1]$, respectively.

5.2 High Level Description

In a DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$) operation, a process first checks if the interpreted values of $D[a_0]$ and $D[a_1]$ match the expected values old_0 and old_1, respectively. If not, the DCAS() operation can fail immediately. Otherwise, p creates a task γ and fills $\gamma.b_0$ and $\gamma.b_1$ with two random bits, whose purpose we will explain later. All other fields of γ are filled with the corresponding parameters of the DCAS() call.

The goal of process p is then to put task γ in all four locations of B that correspond to its two addresses a_0 and a_1, namely into locations $B[2a_0]$, $B[2a_0+1]$, $B[2a_1]$, and $B[2a_1+1]$. As in the case of our BDCAS() algorithm, this must be achieved without changing the interpreted values of $D[a_0]$ and $D[a_1]$. Process p first tries to put γ into $B[2a_0]$ and $B[2a_1+1]$ with a single BDCAS() operation, and then in two subsequent BDCAS() operations it will try to put γ also into $B[2a_0+1]$ and $B[2a_1]$. The control bits in B are used to indicate which stage of that procedure a task has reached.

Shared Data:

- B is a BDCAS object with address set $[2m]$ partitioned into sets $M_0 = \{2a : a \in [m]\}$ and $M_1 = \{2a+1 : a \in [m]\}$. The entries of B are task references. Initially, $B[2\alpha] = B[2\alpha+1] = \zeta_\alpha$, for $\alpha \in [m]$. W.l.o.g., the two argument triples passed to a B.BDCAS() call can be ordered arbitrarily.

Method read(a)

```
48 (γ, ·) ← B.read(2a)
49 Let i ∈ {0,1} such that γ.add_i = a
50 if γ.stat = True then
51     return γ.new_i
52 return γ.old_i
```

Method DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$)

```
53 foreach i ∈ {0,1} do
54     (γ', ·) ← B.read(2a_i + 1 − i); finish(γ')
55     (γ_i, ·) ← B.read(2a_i + i); finish(γ_i)
56 while read(a_0) = old_0 and read(a_1) = old_1 do
57     Choose b_0, b_1 ∈ {0,1} independently and uniformly at random
58     γ ← new Task(False, a_0, old_0, new_0, a_1, old_1, new_1, b_0, b_1)
       // Try to attach γ to both endpoints:
59     B.BDCAS(⟨2a_0, (γ_0, 1), (γ, 0)⟩, ⟨2a_1 + 1, (γ_1, 1), (γ, 0)⟩)
60     foreach i ∈ {0,1} do
61         (γ', ·) ← B.read(2a_i + 1 − i); finish(γ')
62         (γ_i, ·) ← B.read(2a_i + i); finish(γ_i)
63     if γ.stat = True then return True
64 return False
```

Fig. 3. General DCAS implementation—main methods

First, process p performs a single BDCAS() operation, trying to change $B[2a_0]$ and $B[2a_1+1]$ to $(\gamma, 0)$. We say γ gets *attached* if that BDCAS() succeeds. Recall that p's next goal is to put the tasks also in the neighbouring fields $B[2a_0+1]$ and $B[2a_1]$, respectively. (A control bit of 0 indicates that a task has just been attached, and a control bit of 1 indicates that it has also been put in the neighbouring field.) We will focus on the first address, a_0, i.e., on p's goal to put γ into $B[2a_0+1]$.

There may be another task stored in $B[2a_0+1]$, say γ', that just attached itself, and is in a symmetric situation as γ. I.e., some process's goal is to put γ' into $B[2a_0]$, which is currently occupied by γ. We use a simple random experiment to choose a winner among γ and γ', and the winning task will end up occupying both, $B[2a_0]$ and $B[2a_0+1]$. All processes trying to help these two tasks agree on a unique and random winner by using the random bits stored in γ and γ' (see Section 5.3 for details). We call this random procedure a *competition* on address a.

Observe that task γ is involved in two competitions, one on address a_0 and the other one on address a_1. If γ wins both of them, then it ends up being present in all four array locations, $B[a_i + j]$ for $i, j \in \{0, 1\}$. In that case, the corresponding BDCAS() operation is bound to succeed, so p can change $\gamma.stat$ to True. If, on the other hand, γ loses at least one competition, then the *task* fails, and unless the interpreted value of $B[a_0]$ or $B[a_1]$ has changed, p may have to start over again by creating a new task. But we have that γ loses each competition independently with probability 1/2, so with probability 1/4 it will win both competitions.

Hence, if a task γ gets attached it will be successful with probability at least 1/4 (it is also possible that it is successful with higher probability, because there may be only one or even no competitor). But what happens if p does not manage to

Method compete(γ, i)

```
   // Addresses of γ and its neighbour:
65 add ← 2γ.add_i + i;  add' ← 2γ.add_i + 1 − i
66 (γ', ·) ← B.read(add')
67 if γ ≠ γ' then
       // Try to replace original neighbour:
68     B.BDCAS(⟨add, (γ, 0), (γ, 1)⟩, ⟨add', (γ', 1), (γ, 1)⟩)
69 (γ', ·) ← B.read(add')
70 if γ.b_i + γ'.b_{1−i} ≡ i (mod 2) then
71     w ← γ
72 else w ← γ'
   // w is the winner of the competition
73 B.BDCAS(⟨add, (γ, 0), (w, 1)⟩, ⟨add', (γ', 0), (w, 1)⟩)
```

Method finish(γ)

```
   // Let γ compete on both sides:
74 compete(γ, 0); compete(γ, 1)
75 if B.read(2γ.add_0)= B.read(2γ.add_1 + 1)= (γ, 1) then
       // γ won both competitions
76     γ.stat.CAS(False, True)
```

Fig. 4. General DCAS implementation—helper functions

attach task γ in the first place, i.e., its very first BDCAS() operation on locations $B[2a_0]$ and $B[2a_1+1]$ fails? Our algorithm (employing the control bits) guarantees that then in at least one of the two locations, say $B[2a_0]$, some other task γ'' gets attached. Thus, γ'' has a constant probability of succeeding, and if that happens, the interpreted value of $D[a_0]$ changes. Hence, p does not have to try again.

To summarize, each time process p performs the above procedure, there is a constant probability that either p's or some other process's BDCAS(), which shares one of p's addresses, succeeds. In either case, p's BDCAS() can terminate (either successfully or unsuccessfully). As a result, each DCAS() operation needs in expectation only a constant number of BDCAS() operations and other steps.

Figure 5 illustrates a possible execution in which two tasks γ and γ' both get attached to neighbouring entries in array B, and perform their competitions.

5.3 Detailed Algorithm Description

5.3.1 The Read Method. In line 48 of a read(a) call, process p reads the task γ stored in $B[2a]$. (It could equally well use $B[2a+1]$, because the interpreted value of $D[a]$ can be determined by the tasks stored in either of these array entries.) In line 49, p determines the index i such that $\gamma.add_i = a$. (Recall that $B[2a]$ may be the array entry to which γ was attached originally, or the one to which it was copied after winning a competition, so index i cannot be derived in any other way.) Then, in line 50, p determines the interpreted value of $D[a]$ using the status of γ. This is linearizable for the same reason as in the BDCAS() case (see Section 4.1).

5.3.2 The Helper Methods. The DCAS() algorithm implements two helper functions in addition to the methods DCAS() and read(). The first helper method is finish(γ), which performs the two competitions an attached task γ is involved in. This is done via the method calls compete($\gamma, 0$) and compete($\gamma, 1$) in line 74, which will be discussed shortly. After the competitions, γ must have either been removed from one of the array entries $B[2\gamma.a_0]$ and $B[2\gamma.a_1+1]$, or it is still

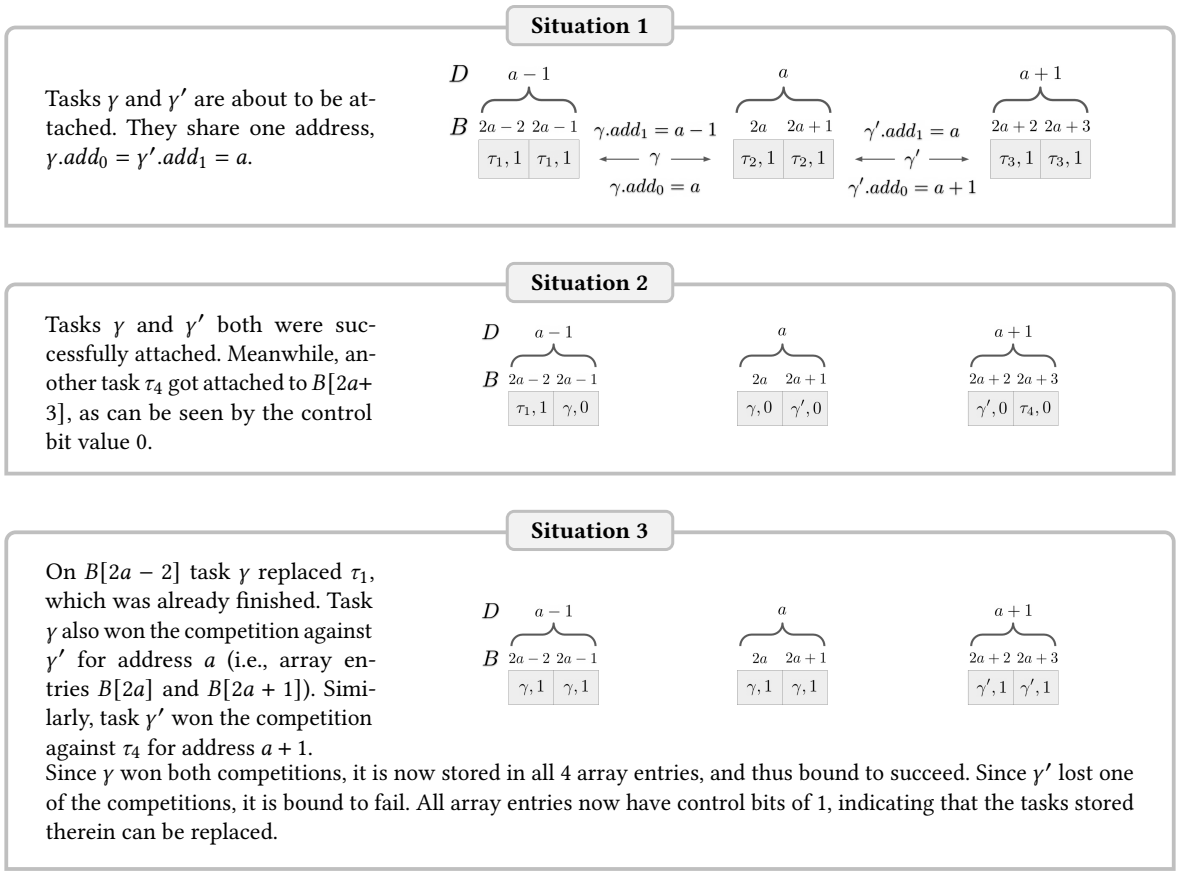

Fig. 5. A sequence of events for the DCAS implementation

present in both of them, but with control bits of 1. In the latter case, γ is (or was at some point) present in all four array entries $B[2\gamma.add_i + j]$ for $i, j \in \{0, 1\}$. Hence, task γ succeeded, so p tries to change the status of task γ to True.

We will now describe method `compete`(γ, i), which processes call from `finish`(γ), and which performs a competition for γ. The parameter i indicates which of the two addresses among $\gamma.add_0$ and $\gamma.add_1$ the competition relates to. For the ease of description, we describe the method for $i = 0$ — the other case is completely symmetric. Hence, consider a `compete`$(\gamma, 0)$ call by process p. In line 65, process p first computes the address $add = 2\gamma.add_0$ on which γ has been attached, and the neighbouring address $add' = 2\gamma.add_0 + 1$. In lines 66–68, p reads a task γ' from the neighbouring array location, and then uses a `BDCAS()` operation to replace both neighbouring array entries with $(\gamma, 1)$, provided that the control bit stored with the neighbour is already 1. It is guaranteed that if that control bit is 1 and the `BDCAS()` succeeds, then γ' has already performed all its competitions and thus has had a chance to succeed. This follows (rather subtly) from the `finish()` calls that p made in one of lines 54 and 61 before attaching γ: If γ' had not completed both its competitions, then p would not have called `finish`(γ') before attaching γ, and then its `BDCAS()` call that attached γ would have failed. Task γ' could also not have been attached after that `BDCAS()` call, because then its control bit could have only changed to 1 in a competition with γ, which γ would have lost. Consequently, γ would have been removed from $B[add]$.

In lines 69–73 process p deals with the case that the neighbouring control bit is 0. It first reads the neighbouring task γ' again in line 69, and then determines the winner of the competition among γ and γ'. More precisely, γ wins if and only if $\gamma.b_0 + \gamma'.b_1$ is an even number, and otherwise γ' wins. The use of index i (which is 0 for task γ but would be 1 for task

γ') in that computation, ensures that γ' chooses the same winner. Then, in line 73, process p finally performs a BDCAS() that replaces both neighbouring array entries $B[2\gamma.add_0]$ and $B[2\gamma.add_0 + 1]$ with $(w, 1)$, where w is the winner. The arguments of the BDCAS() ensure that it can only succeed if indeed the neighbours' control bit is 0 (the earlier BDCAS() in line 68 dealt with the case that the bit is 1).

5.3.3 The DCAS Method. We now describe the DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$) method. In the foreach-loop in lines 53–55, process p reads the tasks stored in all array entries of B that are relevant to it, namely $B[2a_i + j]$ for $i, j \in \{0, 1\}$. For each task γ that p finds, it calls finish(γ). This ensures that all competitions for γ are performed, and $\gamma.stat$ is set to True if γ wins both competitions. Process p also stores the tasks it found in $B[2a_0]$ and $B[2a_1 + 1]$ in its local variables γ_0 and γ_1, because these determine the arguments of its later BDCAS() call, in which p tries to attach γ.

Then p starts the while-loop in line 56. At the beginning of each iteration, p performs read(a_0) and read(a_1) operations, which return the interpreted values of $D[a_0]$ and $D[a_1]$, respectively. Process p breaks out of the while-loop if one of the interpreted values doesn't match the expected values v_0 or v_1, respectively. In that case, p returns False in line 64. Otherwise, p creates a new task γ in line 58, filling the bits $\gamma.b_0$ and $\gamma.b_1$ with two random values, and all other fields with the corresponding parameters from the DCAS() call. Then, in line 59 process p tries to attach task γ. That BDCAS() only succeeds if $B[2a_0]$ and $B[2a_1 + 1]$ still store the tasks γ_0 and γ_1 for which p previously called finish(). (Note that if γ_i is still in $B[2a_i + i]$, then due to the previous finish(γ_i) call the control bit must be 1.) After that, in lines 60–62, p performs a foreach-loop that is identical to the one described earlier. It prepares for the next possible while-loop iteration (as the earlier foreach-loop does), but at the same time it also makes sure that if p managed to attach task γ, then finish(γ) is called. This ensures that afterwards the status of γ is True, if and only if task γ got attached and won all competitions, and thus has been put into all array entries $B[2a_i + j]$ for $i, j \in \{0, 1\}$. If that is the case, then p returns True in line 63, and otherwise p has to try again with another task in the next while-loop iteration.

The detailed analysis is given in Section 8.

6 REPEATED CHOICE ANALYSIS

6.1 Correctness

Let L be a RepeatedChoice object and E an execution on L. We say L has *interpreted agreement-value* v, if the first component stored in S has value v. Similarly, we let L_t denote the value of the first component of S_t (note that this way L_t is defined if S_t changes at point t, even though the value of L is undefined at point t). We call the integer stored in the third component of S the *sequence number* of S. We say S is *locked*, if its sequence number is odd, and it is *unlocked* of the sequence number is even. We say two operations op_0 and op_1 on L *overlap*, if for some $i \in \{0, 1\}$ op_i's first shared memory operation is executed after op_{1-i}'s first and before op_{1-i}'s last shared memory operation.

6.1.1 Linearization Points. Let op be one of the operations propose(), choose&lock(), unlock(), and read(), and let p be the process executing op. For each such operation we define a point $lin(op)$ during the execution E, and we will prove that $lin(op)$ is a strong linearization point.

If op is a propose(v) or read() operation, then op comprises only one shared memory operation, and $lin(op)$ is the point when that operation is executed.

Now assume that op is a choose&lock() operation. If S is locked when p reads S in line 3, then $lin(op)$ is the point of that read. Otherwise, $lin(op)$ is the point of the first shared memory operation that changes S after p reads S in line 3, and $lin(op) = \infty$ if S does not get changed during the rest of the execution.

Finally suppose that op is an unlock(w) operation. Suppose p reads the triple (w', j, k) from S in line 10. If S is unlocked at that point or $w' \neq w$, then $lin(op)$ is the point of that read. Otherwise, $lin(op)$ is the point of the first shared memory operation that changes S after p reads S in line 10, and $lin(op) = \infty$ if S does not get changed during the rest of the execution.

Observation 6.1. *If $lin(op) = \infty$, then op does not respond.*

Proof. Let p be the process executing op. The claim is trivially true if op is a `propose()` or a `read()` operation.

Now suppose op is an `unlock(`w`)` operation and $lin(op) = \infty$. For the purpose of a contradiction, assume op responds. Then p reads the pair (w', j, k) from S in line 10 while $w = w'$ and S is locked. Hence, p executes the operation S.`CAS(`w', j, k`)` in line 16. Since $lin(op) = \infty$, the value of S does not change between the point when p reads it in line 10 and when p executes its `CAS()` in line 16. Hence, that `CAS()` succeeds, and $lin(op)$ is the point of this `CAS()`—a contradiction.

Finally, let op is a `choose&lock()` operation with $lin(op) = \infty$. As above, we assume that op responds, and will arrive at a contradiction. By definition of lin, S is locked when p reads S in line 3, and so p executes line 9 during op. Then the value of S changes either with the `CAS()` operation that p executes in that line, or it changes before that and after p reads it in line 3. In either case, $lin(op) < \infty$, which is a contradiction. □

6.1.2 *Correctness of Decide&Lock() and Unlock().*

Lemma 6.2. *If the interpreted agreement-value of L changes at point t, then a successful* `choose&lock()` *call linearizes at that point.*

Proof. By definition, the first component of S must change when the interpreted agreement-value of L changes. This is only possible with a successful `CAS()` in line 9, and thus at the linearization point of a successful `choose&lock()` call. □

We will now show that the interpreted agreement-value of L must always have been proposed.

Lemma 6.3. *Suppose at point t the interpreted agreement-value of L is $w \neq \perp$. Then a* `propose(`w`)` *call linearizes before t.*

Proof. Initially, the interpreted agreement-value of L is $\perp$. The interpreted agreement-value of L can only change if some process p executes a successful `CAS()` in line 9. In that case the value of L changes to w, where w is either $\perp$ or the non-$\perp$ value p most recently read in line 7 from some entry in $C[i][j]$. That value w must then have been written to $C[i][j]$ in line 2, i.e., it must have been the argument of a `propose()` call that already linearized. □

Lemma 6.4. *Between the linearization points of any two successful* `choose&lock()` *calls a successful* `unlock()` *call linearizes, and between the linearization points of any two successful* `unlock()` *calls a successful* `choose&lock()` *call linearizes. Similarly, a successful* `choose&lock()` *call linearizes before the first successful* `unlock()` *call does.*

Proof. According to the conditional in lines 4 and 11 and the parameters of the `CAS()` calls in lines 9 and 16 `choose&lock()` and `unlock()` calls can only be successful if immediately before their linearization points the sequence number of S is even and odd, respectively. Hence, the claim follows immediately from the fact that initially that sequence number is even (0), and in a successful `choose&lock()` or `unlock()` operation the `CAS()` at the operation's linearization point succeeds, and increments the sequence number of S. □

6.1.3 *ABA-Freedom.*

In this section we will prove that L exhibits no ABAs, and that each successful `choose&lock()` call either changes the interpreted agreement-value of L, or leaves it at $\perp$.

Observation 6.5. *If $S = (v, i, \ell)$ at point t and at point $t' > t$, then no successful* `CAS()` *is executed on S throughout $[t, t']$.*

Proof. This follows immediately from the fact that the sequence number of S increments with each successful `CAS()` on S. □

Observation 6.6. *Let each of op_0 and op_1 be a successful* `choose&lock()` *or* `unlock()` *call. Then op_0 and op_1 do not overlap.*

Proof. For each $i \in \{0, 1\}$ let process p_i be the process executing op_i. Then in its first step during op_i, at point t_i, process p_i reads a value s_i from S. In its last step during op_i, at point t_i', process p_i performs a successful S.`CAS(`$s_i, \cdot$`)`. It

follows from Observation 6.5, that that CAS() only succeeds if no other CAS() on S succeeds in $[t_i, t_i']$. In particular, the last step of op_i cannot be during op_{1-i}, and so op_0 and op_1 do not overlap. □

Claim 6.7. *Suppose a process p calls* unlock() *at point t_1, and the call is successful and linearizes at point t_2. Further, let k be the bit stored in the middle component of S at point t_1. If w is written to an entry of $C[k][]$ before t_1, then w is not stored in array $C[]$ at any point after t_2.*

Proof. Since the unlock() call is successful, p completes the entire for-loop. It is immediate from the code that if $C[k][\ell] = w$ when p reads $C[k][\ell]$ in line 13 and $C[k][\ell] = w$ when p executes its CAS() in line 15, then $C[k][\ell]$ gets changed to $\perp$. Hence, if w is written to $C[k][]$ before point t_1, then at some point in $[t_1, t_2]$ the value w stored in $C[k][]$ gets either overwritten or erased. Since w is only written to $C[]$ when it gets proposed, and only to one entry in the array, and no value can be proposed twice, $C[]$ does not contain w after point t_2. □

Lemma 6.8. *Let $t_1 < t_2$ be two points in time and $L_{t_1} = L_{t_2} = w \neq \perp$. Then*

(a) no successful choose&lock() *call linearizes in $(t_1, t_2]$; and*
(b) the interpreted agreement-value of L is w throughout $(t_1, t_2]$.

Proof. Part (b) follows from (a), because according to Lemma 6.2 the interpreted agreement-value of S can only change at the linearization point of a successful choose&lock() call. Hence, it suffices to prove (a).

For the purpose of a contradiction assume that a successful choose&lock() call d linearizes in $(t_1, t_2]$, and let $t' \in (t_1, t_2]$ be the last point when that happens. Then during d, the calling process reads w from some entry of array $C[]$ and at point t' it writes it into the first component of S (in line 9). Hence,

$$C[] \text{ contains value } w \text{ at a point after the first shared memory operation of } d \text{ has been executed.} \tag{1}$$

Since the interpreted agreement-value of L is initially $\perp$, and $L_{t_1} = w \neq \perp$, by Lemma 6.2 there is a point $t \leq t_1$ at which another choose&lock() call linearizes, and as a result of that $L_t = w$. Hence, during that choose&lock() call, the calling process reads w from some entry $C[i_1][j_1]$ in line 7, before at point t in line 9 it writes the triple (w, i_1, ℓ_1) for some integer ℓ_1 into the third component of S. By Lemma 6.4, there is a successful unlock() call that linearizes in (t, t'). Consider the first such unlock() call, and let q be the process executing it. Since that unlock() call succeeds, in line 10 process q reads (w, i_1, ℓ_1) from S. By Claim 6.7, array $C[]$ does not contain an entry with value w once that unlock() call completes. Since the unlock() call linearizes in (t, t'), it linearizes before d does (recall that t' is the linearization point of d). By Observation 6.6 it cannot overlap with d, so its last shared memory operation is executed before the first shared memory operation of d. Thus, $C[]$ does not contain w at any point after the first shared memory operation of d has been executed, which contradicts (1). □

6.1.4 *Linearizability.*

Theorem 6.9. *The algorithm in Figure 1 is a strongly linearizable implementation of a* RepeatedChoice *object.*

Proof. We say that L's status is locked, whenever S is locked. Clearly, a read() method returns the interpreted agreement-value of L at its linearization point, and does not change the status or interpreted agreement-value of L. Similarly, propose() does not change the status or interpreted agreement-value, and it also does not return anything.

Now consider a choose&lock() call. It follows immediately from the CAS() in line 9 and the if-condition in line 4, that a choose&lock() call is successful if and only if at its linearization point the status of L is unlocked, and in that case it changes the status to locked. If successful, then by Lemma 6.3 the new value of L was previously proposed. If unsuccessful, then it does not change the status of L, and by Lemma 6.2 also not the interpreted agreement-value.

Next consider an unlock() call. It follows immediately from the CAS() in line 16 and the if-condition in line 11, that an unlock(v) call is successful if and only if at its linearization point the status of L is locked and the interpreted

agreement-value is v. In that case it changes the status to unlocked. If unsuccessful, then it does not change the status of L. By Lemma 6.2 an `unlock()` call does not change the interpreted agreement-value.

Finally, by Lemma 6.8 the interpreted agreement-value of L is ABA-free.

From all of the above we conclude that if we order all operations op with $lin(op) < \infty$ by their linearization points, then we obtain a valid sequential history. By the definition of lin and by Observation 6.1, $lin(op)$ is between the invocation and response of op. This proves linearizability.

It is also obvious that lin defines strong linearization points: Consider a prefix E' of an execution E that ends at point $t = lin(op)$. Then from the definition of lin it is obvious that op also linearizes at point t in E'. Hence, each operation in E' linearizes at the same point in E' as in E. Since this is true for all prefixes E' of E, strong linearizability follows. □

6.1.5 *Auxiliary Lemma.*

Lemma 6.10. *Let $R \subseteq V$ and let $t_1 < t_2$ be two points in time such that during $[t_1, t_2]$ at least four successful* `choose&lock()` *calls linearize, and between the response of each successful* `unlock()` *call and the invocation of the following* `choose&lock()` *call a* `propose(`v`)` *call linearizes for some value $v \in R$. Then there is a point $t^* \in [t_1, t_2]$ such that $L_{t^*} \in R$.*

Proof. Let d_1, d_2, d_3, d_4 be the first four successful `choose&lock()` calls that linearize (in this order) in $[t_1, t_2]$. By Lemma 6.4 there are successful `unlock()` calls u_1, u_2, u_3, such that the `choose&lock()` and `unlock()` calls linearize in the order

$$d_1, u_1, d_2, u_2, d_3, u_3, d_4.$$

By Observation 6.6, no two of these operations overlap. If op is one of these operations, we denote by $inv(op)$ and $rsp(op)$ the points of its invocation and response, respectively.

For $i \in \{1, 2, 3\}$ let a_i be the value of the middle bit of S at point $inv(u_i)$. The middle bit of S changes when and only when a successful `unlock()` call linearizes (i.e., when a successful CAS is executed in line 16). Hence,

$$\text{the middle bit of } S \text{ is } a_{i+1} = 1 - a_i \text{ throughout } (lin(u_i), lin(u_{i+1})). \tag{2}$$

Since u_1 and u_2 are both called after t_1, by Claim 6.7 at point $lin(u_2)$ each array entry of C stores either $\perp$ or a value in V. Since only values in R get written to array entries of C throughout $[t_1, t_2]$, we have:

$$\text{At any point } t \in [lin(u_2), t_2] \text{ each array entry of } C \text{ is in } R \cup \{\perp\}. \tag{3}$$

Now let $y \in R$ such that a `propose(`y`)` call pr_y linearizes at some point $lin(pr_y)$ between $rsp(u_2)$ and $inv(d_3)$. At $lin(pr_v)$ value v is written to an array entry $C[a][b]$, for some $a \in \{0, 1\}$ and $b \in \{1, \ldots, \lambda\}$. Since $lin(d_3) \in (rsp(u_2), lin(d_3)) \subseteq (lin(u_2), lin(u_3))$, it follows from (2) that the middle bit of S is a_3 at point $lin(d_3)$. Similarly, at point $lin(d_4)$ the middle bit of S is $a_4 = 1 - a_3$. Hence, there is an index $\ell \in \{3, 4\}$ such that at $lin(d_\ell)$ the middle bit of S is a. Moreover, at most one successful `unlock()` call (namely u_3) linearizes in $[lin(pr_v), lin(d_\ell)]$. Hence, by Lemma 6.13, $C[a][b] \neq \perp$ throughout $[lin(pr_v), lin(d_\ell)]$.

Since d_ℓ is a successful `choose&lock()` call, $rsp(d_\ell) = lin(d_\ell)$. In addition, $lin(pr_v) < inv(d_\ell)$, so $C[a][b] \neq \perp$ throughout $[inv(d_\ell), rsp(d_\ell)]$. Hence, during the for-loop of operation d_ℓ the calling process scans array $C[a][j]$ for $j \in \{1, \ldots, \lambda\}$ until it finds a non-$\perp$ entry y. By (3) $y \in R$. Thus, the process executing d_ℓ performs a successful CAS in line 9 that changes the first element of S to y, and so the interpreted agreement-value of L becomes y. This happens at $lin(d_\ell) \in [t_1, t_2]$, and so the claim is true for $t^* = lin(d_\ell)$. □

6.2 Distribution of Interpreted Agreement-Value

In this section, we analyse the distribution of the interpreted agreement-value of a `RepeatedChoice` object immediately after a successful `choose&lock()` call.

Adversary. We consider an execution on a `RepeatedChoice` object L scheduled by a *weak adaptive* adversary. At each point, the adversary can see the interpreted agreement-value of L and its lock status, but cannot see the internal data of L. Precisely, it can see $L.S$ but not $L.C$. Also, the adversary cannot see the internal state of the processes, including the outcome of their random coins. After each step in the execution, the adversary chooses the point in time of the next step in the execution, and the process to take that step. If the process selected has no pending next operation, the adversary chooses also the method call that the process invokes. We allow the adversary to make probabilistic decisions.

Schedules. For any execution E on L, we define $\mathcal{S}_L(E)$ as follows. For each step σ of E, $\mathcal{S}_L(E)$ indicates: (i) the point in time when σ takes place, (ii) the process p that executes σ, (iii) if p invokes a new method call on L at step σ, the name and arguments of that call, (iv) the value of $L.S$ immediately after step σ. If E is finite, we define $\mathcal{S}^-_L(E)$ identically to $\mathcal{S}_L(E)$, except that $\mathcal{S}^-_L(E)$ does not indicate the value of $L.S$ after the last step of E.

For a partial execution E, one can infer from $\mathcal{S}_L(E)$ precisely which of the method calls invoked in E are completed and which ones are still pending at the end of E. The reason is that the number of steps of each method call of L is either fixed (namely, `propose()` and `read()` involve a single step each), or depends only on the method's arguments and the values of $L.S$ during the call.

Note that if E is finite and is scheduled by the adversary above, $\mathcal{S}_L(E)$ contains all the information that the adversary knows (including all its own decisions) after the last step in E, and $\mathcal{S}^-_L(E)$ contains all that the adversary knows immediately before the last step takes effect.

Main Theorem. The next theorem states the main result that we prove in this section. Recall that L_t is a shorthand for the first component of $L.S_t$, and $L_t \in V \cup \{\perp\}$.

Theorem 6.11. *Let E be a partial random execution in which an adversary as described above schedules calls to the methods of a* `RepeatedChoice` *object L. Fix $\mathcal{S}^-_L(E)$, and suppose that the last step of E is the linearization point t^* of a successful L.*`choose&lock()` *call. Suppose that a successful L.*`choose&lock()` *call linearizes at point t_1 and two successful L.*`unlock()` *calls linearize at points s_1, s_2, where $t_1 < s_1 < s_2 < t^*$, and no other successful L.*`unlock()` *call linearizes in (t_1, t^*). Finally, suppose that $L_{s_2} = u_2$, and that exactly k values from the set $V \setminus \{u_2\}$ are proposed in (s_1, s_2). Then, for any value $v \in V \setminus \{u_2\}$ proposed in (t_1, t^*),*

$$\Pr[L_{t^*} = v] \leq 8/k, \text{ if } k \neq 0, \tag{4}$$

while $L_{t^*} \neq v$ for all other $v \in V$, and

$$\Pr[L_{t^*} = \perp] \leq 2^{-k}. \tag{5}$$

6.2.1 Auxiliary Lemmas. Recall from line 1 that $\pi_j = 2^{-j}$, for $j \in \{1, \ldots, \lambda - 1\}$, and $\pi_\lambda = 2^{-\lambda+1}$.

Lemma 6.12. *Let $X_0, \ldots, X_{k-1}$ be a sequence of $k \geq 9$ random variables that take values in the range $\{0, 1, \ldots, \lambda\}$, where $\lambda \geq \log k$. Suppose that for each $1 \leq j \leq \lambda$,*

- $\Pr[X_0 = j] \leq \pi_j$, *and*
- $\Pr[X_i = j \mid X_0, \ldots, X_{i-1}] \geq \pi_j/2$, *for $1 \leq i \leq k-1$.*

Then, $\Pr[X_0 = \max_i X_i] \leq 8/k$.

Proof. Let $\lambda_0 = \lfloor \log(k-1) \rfloor - 1 \geq 2$.

$$\Pr\left[X_0 = \max_i X_i\right] = \sum_{j=0}^{\lambda} \Pr[X_0 = j] \cdot \Pr\left[\max_{i \neq 0} X_i \leq j \,\middle|\, X_0 = j\right]$$
$$\leq \sum_{j=0}^{\lambda-1} 2^{-j} \cdot (1 - 2^{-j-1})^{k-1} + 2^{-\lambda+1}$$

$$\leq 1 \cdot 2^{-k+1} + \sum_{j=1}^{\lambda_0} 2^{-j} \cdot e^{-2^{-j-1}\cdot(k-1)} + \sum_{j=\lambda_0+1}^{\lambda-1} 2^{-j} + 2^{-\lambda+1}$$

$$= 2^{-k+1} + \sum_{j'=0}^{\lambda_0-1} 2^{-(\lambda_0-j')} \cdot e^{-2^{-(\lambda_0-j')-1}\cdot(k-1)} + 2^{-\lambda_0}$$

$$\leq 2^{-k+1} + 2^{-\lambda_0} \cdot \sum_{j'=0}^{\lambda_0-1} 2^{j'} \cdot e^{-2^{j'}} + 2^{-\lambda_0}$$

$$\leq 2^{-k+1} + 2^{-\lambda_0} \cdot (3/4) + 2^{-\lambda_0}$$

$$\leq 2^{-k+1} + 7/(k-1)$$

$$\leq 8/k. \qquad \square$$

Lemma 6.13. *Let $a \in \{0,1\}$ and $b \in \{1,\dots,\lambda\}$. Suppose some value $u \neq \perp$ is written to $C[a][b]$ at point t_1, a successful* `choose&lock()` *call linearizes at point $t_2 > t_1$, the value stored in the middle component of S_{t_2} is a, and at most one successful* `unlock()` *call linearizes in the interval $[t_1, t_2]$. Then $C_t[a][b] \neq \perp$ for all $t \in [t_1, t_2]$.*

Proof. For the purpose of contradiction, suppose there is a point in $(t_1, t_2]$ at which $C[a][b] = \perp$. The value of $C[a][b]$ can change to $\perp$ only as a result of a successful `CAS()` operation in line 15. Hence, there is a point $t_{p@15} \in (t_1, t_2]$ at which some process p executes a successful $C[a][b]$.`CAS`$(u', \perp)$ operation in that line for some value $u' \neq \perp$ (possibly $u' = u$). Let $t'_1 < t_{p@15}$ be the point when u' is written to $C[a][b]$ for the first time. Then t'_1 is the linearization point of a `propose`(u') call. Since the type's specification requires that each value gets proposed at most once, we have

$$C_t[a][b] = u' \text{ if and only if } t \in [t'_1, t_{p@15}). \tag{6}$$

In particular, since u is written to $C[a][b]$ at point t_1 and t' is a point in $[t_1, t_2]$ at which $C[a][b]$ changes to $\perp$, we have

$$t_1 \leq t'_1 < t_{p@15} \leq t_2. \tag{7}$$

Prior to $t_{p@15}$, at points $t_{p@10}$ and $t_{p@14}$, process p reads S in lines 10 and 14, respectively. Due to the if-condition in line 14, the value of S does not change in $[t_{p@10}, t_{p@14}]$. Since $t_{p@13}$ is in that interval, and from the if-condition in line 11 it follows that

$$S_{t_{p@13}} \text{ is locked, and its middle bit is } a. \tag{8}$$

By (6) and (7), $t_{p@13} \in [t_1, t_2]$. Since a successful `choose&lock()` call linearizes at t_2, S is unlocked immediately before t_2. Hence, a successful `unlock()` call linearizes in $(t_{p@13}, t_2)$, and by the lemma assumption exactly one such `unlock()` call *op* linearizes in that interval. By line 16, the middle bit of S flips when and only when a successful `unlock()` call linearizes. Hence, by (8), at the linearization point of the successful `unlock()` call in $(t_{p@13}, t_2)$ the middle bit of S changes from a to $1 - a$, and then it does not change again until t_2. This contradicts the assumption that the middle bit of S_{t_2} equals a. $\square$

6.2.2 Proof of Theorem 6.11. In the following we omit prefix L when we refer to methods or variables of L, e.g., we write S instead of $L.S$. But we still use L_t to denote the first component of $L.S_t$. We use the terms *δ-call* and *μ-call* to refer to a *successful* L.`choose&lock()` call and a *successful* L.`unlock()` call, respectively.

From the theorem's assumptions about points t^*, t_1, s_1, s_2, and from Lemma 6.4, we have that no μ-call linearizes in $(t_1, t^*) \setminus \{s_1, s_2\}$, no δ-call linearizes in $(t_1, s_1] \cup [s_2, t^*)$, and exactly one δ-call linearizes in (s_1, s_2). Let t_2 denote the linearization point of the δ-call in (s_1, s_2). Then, there are $u_1, u_2 \in V \cup \{\perp\}$, an integer $i^* \in \{0,1\}$, and an *odd* integer

$\ell^* \geq 1$, such that

$$S_t = \begin{cases} (u_1, i^*, \ell^*), & \text{if } t \in [t_1, s_1) \\ (u_1, 1-i^*, \ell^*+1), & \text{if } t \in [s_1, t_2) \\ (u_2, 1-i^*, \ell^*+2), & \text{if } t \in [t_2, s_2) \\ (u_2, i^*, \ell^*+3), & \text{if } t \in [s_2, t^*) \\ (L_{t^*}, i^*, \ell^*+4), & \text{if } t = t^*, \end{cases} \tag{9}$$

because the first component of S may change only at the linearization point of a δ-call (by Lemma 6.2), the second component changes precisely at the linearization points of μ-calls, and the third component increases by one at the linearization point of each δ- and μ-call (by similar arguments); also the third component is odd immediately after a δ-call (by lines 4 and 9).

Note that values u_1, u_2, i^*, ℓ^* can be inferred from $\mathcal{S}_L^-(E)$, but, in general, this is not the case for the value of L_{t^*}, as $\mathcal{S}_L^-(E)$ does not indicate the first component of S_{t^*}. (Yet, the fact that t^* is the linearization point of a δ-call can be inferred from $\mathcal{S}_L^-(E)$.)

Recall that each value $u \in V$ is proposed at most once, and let P be the set of all values proposed in E. For each $u \in P$, let r_u be the linearization point of the call `propose(u)`, and let $C[\alpha_u][\beta_u]$ be the array entry on which u is written at point r_u, in line 2. Note that r_u can be inferred from $\mathcal{S}_L^-(E)$, for any $u \in P$, but, in general, α_u and β_u cannot. Let

$$P_1 = \{u \in P \colon r_u \in (t_1, t^*)\} \setminus \{u_2\}, \quad P_2 = \{u \in P \colon r_u \in (s_1, s_2)\} \setminus \{u_2\},$$

and note that $|P_2| = k$, where k was defined in the theorem's statement.

First we argue that $L_{t^*} \notin V \setminus P_1$.

Claim 6.14. *If $v \in V \setminus P_1$, then $L_{t^*} \neq v$.*

Proof. If $v \in V \setminus P$, the claim follows immediately from Lemma 6.3. So, suppose that $v \in P \setminus P_1$, i.e., $r_v < t_1$ or $v = u_2 \neq \perp$. If $v = u_2 \neq \perp$ then $L_{t^*} \neq v$, because if $L_{t^*} = u_2 \neq \perp$, then Lemma 6.8(a) implies that no δ-call linearizes in $(s_2, t^*]$, contradicting the fact that t^* is the linearization point of a δ-call. In the remainder, we assume $r_v < t_1$. Then value v is written to an entry of C before point t_1. Let μ_j, for $j \in \{1, 2\}$, denote the μ-call that linearizes at point s_j, and δ^* the δ-call that linearizes at t^*. From Observation 6.6, μ_1 is invoked after point t_1, μ_2 is invoked after t_2, and δ^* is invoked after s_2. Then, from (9), at the invocation points of μ_1 and μ_2, the second component of S is respectively i^* and $1 - i^*$. By applying Claim 6.7 twice, we obtain that array C does not contain v at any point after s_2. Thus, the process that executes δ^* does not see v in C during δ^*, so the value it writes at point t^* to the first component of S, in line 9, is $L_{t^*} \neq v$. □

Next we prove (4), for $v \in P_1$. Suppose that $|P_2| = k \geq 9$, otherwise (4) holds trivially. For each $u \in P$, define

$$X_u = \begin{cases} \beta_u, & \text{if } \alpha_u = i^* \\ 0, & \text{if } \alpha_u = 1 - i^*. \end{cases}$$

Claim 6.15. *If $v \in P_1$ and $L_{t^*} = v$, then $X_v \geq \max_{u \in P_2} X_u$.*

Proof. Suppose that $v \in P_1$ and $L_{t^*} = v$, and assume, for contradiction, that $X_v < \max_{u \in P_2} X_u$. Then, there is some $u \in P_2$ such that $X_v < X_u$. It follows $X_u > 0$, thus $\alpha_u = i^*$ and $X_u = \beta_u$.

Let δ^* be the δ-call that linearizes at t^*, and p the process that executes δ^*. From (9), the middle component of S_{t^*} is i^*, thus from the code of `choose&lock()` it follows that p reads value v from $C[i^*][j]$, for some $1 \leq j \leq \lambda$, and reads $C[i^*][j'] = \perp$, for all $j < j' \leq \lambda$. Since value v is written to C only once, at point r_v, it follows $\alpha_v = i^*$ and $\beta_v = j = X_v$.

From assumption $X_v < X_u$, it then follows $\beta_v < \beta_u$. Moreover, since $u \in P_2$, value u was written to $C[i^*][\beta_u]$ at point $r_u \in (s_1, s_2)$. We can then apply Lemma 6.13 to obtain that $C_t[i^*][\beta_u] \neq \perp$ for all $t \in [r_u, t^*]$. Moreover,

from Observation 6.6, δ^* is invoked after s_2, thus while executing δ^*, p reads a non-$\perp$ value on $C[i^*][\beta_u]$, which is a contradiction. □

The next key lemma, proved in Section 6.2.3, allows us to apply Lemma 6.12 to bound the probability that $X_v \geq \max_{u\in P_2} X_u$. Recall that π_j is the probability of choosing $\beta = j$ in line 1.

Lemma 6.16. *If $v \in P_1$, then for any $1 \leq j \leq \lambda$,*

$$\Pr[(\alpha_v, \beta_v) = (i^*, j) \mid (\alpha_u, \beta_u) \text{ for all } u \in P \setminus \{v\}] \in [\pi_j/2, \pi_j]. \tag{10}$$

From Lemma 6.16, it follows that for any $v \in P_1$ and $1 \leq j \leq \lambda$,

$$\Pr[X_v = j \mid X_u \text{ for all } u \in P_2 \setminus \{v\}] \in [\pi_j/2, \pi_j]. \tag{11}$$

Thus, for any $v \in P_1$, variables X_v and X_u, $u \in P_2 \setminus \{v\}$, satisfy the conditions of Lemma 6.12, hence

$$\Pr\left[X_v \geq \max_{u\in P_2} X_u\right] = \Pr\left[X_v = \max_{u\in P_2\cup\{v\}} X_u\right] \leq 8/|P_2 \cup \{v\}| \leq 8/k.$$

From that and Claim 6.15, it follows that $\Pr[L_{t^*} = v] \leq 8/k$, if $v \in P_1$. This proves (4).

Next we show (5). Suppose that $|P_2| = k > 0$, otherwise (5) holds trivially. We use the next claim, whose proof is similar to the proof of Claim 6.15.

Claim 6.17. *If $L_{t^*} = \perp$ then $\max_{u\in P_2} X_u = 0$.*

From Claim 6.17,

$$\Pr[L_{t^*} = \perp] \leq \Pr\left[\max_{u\in P_2} X_u = 0\right] \leq \prod_{u\in P_2}\left(1 - \sum_{1\leq j\leq\lambda} \pi_j/2\right) = (1/2)^{|P_2|} = 2^{-k},$$

where the second inequality is obtained using the lower bound from (11).

6.2.3 Proof of Lemma 6.16. Recall we have fixed $\mathcal{S}_L^-(E)$ and suppose $\mathcal{S}_L^-(E) = \Xi$. We fix an arbitrary $v \in P_1$. We also fix all pairs α_u, β_u, for $u \in P \setminus \{v\}$, in a way consistent with $\mathcal{S}_L^-(E) = \Xi$, and fix all α_u, β_u, for $u \in V \setminus P$, arbitrarily. Finally, if the adversary is probabilistic, we fix the outcome of the adversary's coins.

We have thus fixed all randomness involved in E except for the random values of α_v, β_v. Next we define a collection of executions obtained by fixing those two values.

For each $i \in \{0, 1\}$ and $1 \leq j \leq \lambda$, let $E(i, j)$ be an execution such that: (i) it has the same number of steps as E, (ii) it is scheduled by the same adversary as E, and if the adversary is randomized it uses the coins we fixed above, (iii) for each value $u \in V \setminus \{v\}$ proposed in $E(i, j)$, the values α_u, β_u we fixed above are used, i.e., u is written on $C[\alpha_u][\beta_u]$, in line 2, and (iv) for v the values $\alpha_v = i$ and $\beta_v = j$ are used. It follows $E(i, j)$ and E are identical in the interval $[0, r_v)$, and v is written on $C[i][j]$ at point r_v in $E(i, j)$. Moreover, if $(\alpha_v, \beta_v) = (i, j)$ in E, then $E = E(i, j)$.

We define execution $E(\perp)$ identically to $E(i, j)$, except that the write operation at point r_v is replaced by a dummy shared memory step.

Note that unlike E, which involves some nondeterminism, namely the values of α_v, β_v, executions $E(i, j)$ and $E(\perp)$ are fixed deterministically.

Recall from (9) that i^* is the value of the middle component of S_{t^*} in E.

Lemma 6.18. *(a) $\mathcal{S}_L^-(E(\perp)) = \Xi$; and (b) for any $1 \leq j \leq \lambda$, $\mathcal{S}_L^-(E(i^*, j)) = \mathcal{S}_L^-(E(\perp))$.*

Proof of Lemma 6.18(a): Case $\alpha_v = 1 - i^*$. Suppose $(\alpha_v, \beta_v) = (1 - i^*, \hat{\jmath})$ in E, for some fixed $1 \leq \hat{\jmath} \leq \lambda$, i.e., $E = E(1 - i^*, \hat{\jmath})$. Let δ_2 denote the δ-call in E that linearizes at t_2, let p_2 be the process that executes δ_2, and let $\hat{t}$ be the point at which p_2 reads $C[1 - i^*][\hat{\jmath}]$, in line 7, during δ_2. Then, $s_1 < \hat{t} < t_2$, where the left inequality follows from Observation 6.6. Recall also that $r_v > t_1$, since $v \in P_1$.

Claim 6.19. *If $u_2 \neq \perp$ then (a) $\hat{\jmath} < \beta_{u_2}$, or (b) $r_v > \hat{t}$, or (c) $\hat{\jmath} = \beta_{u_2}$ and $r_v < r_{u_2}$. If $u_2 = \perp$ then $r_v > \hat{t}$.*

Proof. Let x denote the value that p_2 reads on $C[1-i^*][\hat{\jmath}]$ at point $\hat{t}$ in E. Suppose first that $u_2 \neq \perp$. We will show that if $\hat{\jmath} \geq \beta_{u_2}$ and $r_v < \hat{t}$, then $\hat{\jmath} = \beta_{u_2}$ and $r_v < r_{u_2}$; this implies the claim. So, suppose that $\hat{\jmath} \geq \beta_{u_2}$ and $r_v < \hat{t}$. Assume, for contradiction, that $\hat{\jmath} \neq \beta_{u_2}$. Then $x \neq u_2$. Also, since $r_v > t_1$, and $r_v < \hat{t} < t_2$, and the middle component of $S_{\hat{t}}$ is $1-i^*$ by (9), it follows from Lemma 6.13 that $x \neq \perp$. Thus, $x \notin \{u_2, \perp\}$, and since $\hat{\jmath} > \beta_{u_2}$, p_2 reads x, in line 7, *after* it reads u_2. Hence, the value that p_2 writes to the first component of S at t_2 is not u_2, which is a contradiction. Thus, $\hat{\jmath} = \beta_{u_2}$. Moreover, if we assume $r_v > r_{u_2}$, then, since $r_v < \hat{t}$, value u_2 is overwritten before p_2 reads $C[1-i^*][\beta_{u_2}]$ at $\hat{t}$, which leads to contradiction, as before. Thus, $r_v < r_{u_2}$.

The proof of the case $u_2 = \perp$ is similar: If $r_v < \hat{t}$ then as above Lemma 6.13 implies $x \neq \perp$, and this leads to the contradiction that the first component of S_{t_2} cannot be $\perp$. □

To simplify exposition, we define $\beta_\perp = 0$ and $r_\perp = 0$. Then, from Claim 6.19, it suffices to consider two sub-cases: (1) $\hat{\jmath} < \beta_{u_2}$ or $r_v > \hat{t}$, and (2) $\hat{\jmath} = \beta_{u_2}$ and $r_v < r_{u_2}$.

In the following, we use superscript $\perp$ to denote variables in execution $E(\perp)$, when we need to distinguish them from the corresponding variables in E. Recall that E_t and $E_t(\perp)$ denote the prefix of E and $E(\perp)$, respectively, in the interval $[0,t]$.

Sub-Case $\hat{\jmath} < \beta_{u_2}$ or $r_v > \hat{t}$. We show by induction that for any $t \in [r_v, t^*]$,

$$\mathcal{S}_L(E_t(\perp)) = \mathcal{S}_L(E_t) \quad \text{and} \quad C^\perp_t[i][j] = C_t[i][j], \text{ for all } (i,j) \neq (1-i^*, \hat{\jmath}). \tag{12}$$

The induction is on the steps of E. We have that (12) holds for $t = r_v$, because the two executions are identical in the interval $[0, t_v)$; and at point r_v of E a process p writes value v on $C[1-i^*][\hat{\jmath}]$ in line 2 of method `propose(`v`)`, while at point r_v of $E(\perp)$, p does a dummy step (which is accounted as a step of `propose(`v`)` in $\mathcal{S}_L(E(\perp))$).

For the induction step, suppose that (12) holds for all $t \leq \sigma$, where $\sigma \in [r_v, t^*)$ and a step of E occurs at σ. Let $\sigma' \in (\sigma, t^*]$ be the point of the next step of E. Suppose that the step of E at σ' is the kth step of some method call, and is executed by process p. Then, the next step in $E(\perp)$ after σ also occurs at point σ', is the kth step of the same method call, and is executed by p. The reason is that we assume the same adversary in both executions (using the same coins), and the adversary can only see the value of S. Since $\mathcal{S}_L(E_\sigma(\perp)) = \mathcal{S}_L(E_\sigma)$, we have $S^\perp_t = S_t$, for all $t \in [0, \sigma]$. Moreover, the number of steps of each method call depends only on the values of S and the method's arguments. It follows that the adversary makes the same choices for the next step after σ in both executions.

Next we consider all possible steps in the two executions at point σ', and argue that (12) holds for $t = \sigma'$. It suffices to consider steps that may modify S or C, namely, lines 2, 9, 15 and 16.

Suppose p executes line 2 of call `propose(`u`)` at σ' in one execution, and thus also in the other. Then $u \neq v$, as $\sigma' > r_v$. Since the same random values α_u, β_u are used in both executions, the exact same write operation takes place in both at σ'. Thus, (12) holds for $t = \sigma'$.

Similarly, if p executes line 16 of call `propose(`val`)` at σ' in one execution, it does the same in the other. Since $\mathcal{S}_L(E_\sigma(\perp)) = \mathcal{S}_L(E_\sigma)$, at the beginning of the call p reads the same value (val', i, ℓ) from S in line 10 in both executions, and also $S^\perp = S$ immediately before σ'. It follows that p executes the exact same `CAS()` operation at σ' in both executions, and the operation has the same outcome (successful or not) in both.

If p executes line 15 at point σ' in one execution, then it does so in the other as well, because the outcome of the if-condition in line 14 must have been the same in the two executions, by the assumption $\mathcal{S}_L(E_\sigma(\perp)) = \mathcal{S}_L(E_\sigma)$. In particular, if in E, p executes command $C[i][j]$.`CAS(`$v_1, \perp$`)` in line 15 at σ', then in $E(\perp)$, p executes $C[i][j]$.`CAS(`$v_2, \perp$`)`, for some v_1, v_2. Suppose $(i,j) \neq (1-i^*, \hat{\jmath})$. Then $v_1 = v_2$, because the same value was previously read from $C[i][j]$ in line 13 in both execution, as we assumed that (12) holds for all $t \leq \sigma$. Also, $C^\perp[i][j] = C[i][j]$ right before σ'. Thus the same

CAS() operation is executed at σ' in both executions, and has the same outcome in both. If $(i, j) = (1 - i^*, \hat{\jmath})$, the operation may be different in the two executions, but can only affect the value of $C[i][j]$, and thus (12) clearly holds for $t = \sigma'$.

Finally, suppose that p executes line 9 at point σ' in one execution, and thus also in the other. In both executions, p reads the same value (val, i, ℓ) on S in line 3, at the beginning of the call, and also the value of S is the same in both executions immediately before p executes the CAS() in line 9. It follows that either in both executions the CAS() operations are successful, or in both the operations are not successful. This holds independently of the second argument of the CAS() operations (which may be different in the two operations). If the operations are not successful, clearly (12) holds for $t = \sigma'$.

Suppose the two operations are successful, and let $(v_1, i, \ell + 1)$ and $(v_2, i, \ell + 1)$ be the values they write to S in E and $E(\perp)$, respectively. We argue that $v_1 = v_2$. Since the operations are successful, σ' is the linearization point of a δ-call. Since exactly two δ-calls linearize in E in the interval $(r_v, t^*] \subseteq (t_1, t^*]$, it follows that either $\sigma' = t_2$ or $\sigma' = t^*$.

If $\sigma' = t_2$, i.e., σ' is the linearization point of δ_2 in E, then $p = p_2$ and, from (9), $v_1 = u_2$ and $i = 1 - i^*$. From the code of choose&lock(), when p_2 executes δ_2, it reads value u_2 on $C[1 - i^*][\beta_{u_2}]$, in line 7, and reads $\perp$ on $C[1 - i^*][j]$, for all $j > \beta_{u_2}$. From the case assumption that $\hat{\jmath} < \beta_{u_2}$ or $r_v > \hat{t}$, and the assumption that (12) holds for $t \le \sigma$, it follows that in $E(\perp)$, p_2 reads the same values on $C[1 - i^*][j]$, for any $j \ge \beta_{u_2}$, as in E. Precisely, if $\hat{\jmath} < \beta_{u_2}$, then p_2 may only see a different value on $C[1 - i^*][\hat{\jmath}]$. While if $r_v > \hat{t}$, i.e., p_2 reads $C[1 - i^*][\hat{\jmath}]$ before point r_v, then p_2 sees the same values in both executions on all $C[i][j]$, as the two executions are identical in $[0, r_v)$ as we observed earlier. It follows that $v_2 = u_2 = v_1$.

Similarly, if $\sigma' = t^*$, then from (9), $i = i^*$. And since (12) holds for all $t \le \sigma$, it follows that p reads the same values in both executions on all $C[i^*][j]$, which implies $v_1 = v_2$.

This concludes the inductive proof of (12). Applying (12) for $t = t^*$ implies the lemma.

Sub-Case $\hat{\jmath} = \beta_{u_2}$ and $r_v < r_{u_2}$. We show by induction that for any $t \in [r_v, t^*]$,

$$\mathcal{S}_L(E_t(\perp)) = \mathcal{S}_L(E_t) \quad \text{and} \quad C_t^{\perp}[i][j] = C_t[i][j], \text{ if } (i, j) \neq (1 - i^*, \hat{\jmath}) \text{ or } t \ge r_{u_2}. \tag{13}$$

The proof is similar to (12)'s. It differs just in the induction step, precisely, in the analysis of the step at point σ', when one of lines 2, 9, 15 and 16 is executed. Below we explain these differences.

If p executes line 2 at σ', then as argued in the proof of (12), the exact same write operation takes place in both executions. In particular, if $\sigma' = r_{u_2}$, then value u_2 is written to $C[1 - i^*][\hat{\jmath}]$, as $\alpha_{u_2} = 1 - i^*$ by (9), and $\beta_{u_2} = \hat{\jmath}$ by the case assumption. It follows that (13) holds for $t = \sigma'$.

If p executes line 16 at σ', then $\sigma' \neq r_{u_2}$ and the same analysis applies as in the proof of (12).

If p executes line 15 at σ' in one execution, it does so in the other as well, as argued in the proof of (12). Precisely, if p executes command $C[i][j]$.CAS$(v_1, \perp)$ at σ' in E, it executes $C[i][j]$.CAS$(v_2, \perp)$ in $E(\perp)$, for some v_1, v_2. Let σ'' be the point when p read $C[i][j]$, in line 13, earlier in the same unlock() call (this is the same point in both execution since $\mathcal{S}_L(E_\sigma(\perp)) = \mathcal{S}_L(E_\sigma)$ and $\sigma'' \le \sigma$).

If $(i, j) \neq (1 - i^*, \hat{\jmath})$ or $\sigma'' > r_{u_2}$, then $v_1 = v_2$, and $S^{\perp} = S$ immediately before σ', by the same argument as in the proof of (12). It follows that (13) holds for $t = \sigma'$, in this case.

If $(i, j) = (1 - i^*, \hat{\jmath})$ and $\sigma' < r_{u_2}$, then clearly (13) holds for $t = \sigma'$, since the operation at σ' only affects the value of $C[1 - i^*][\hat{\jmath}]$ in the two executions.

It remains to consider the case in which $(i, j) = (1 - i^*, \hat{\jmath})$ and $\sigma'' < r_{u_2} < \sigma'$. From $\sigma' > r_{u_2}$ it follows that $C^{\perp}[i][j] = C[i][j] = v' \in V \cup \{\perp\}$ immediately before σ', since (13) holds for $t < \sigma'$. If $v' = \perp$ then the CAS() operation at σ' has no effect (since its second argument is also $\perp$), thus (13) holds for $t = \sigma'$. Suppose $v' \neq \perp$. Value v' must have been proposed at some point in $[r_{u_2}, \sigma')$, as $\sigma' > r_{u_2}$ and $(\alpha_{u_2}, \beta_{u_2}) = (1 - i^*, \hat{\jmath})$. Moreover, since $\sigma'' < r_{u_2}$, it follows that $v_1 \neq v'$ and $v_2 \neq v'$. We thus conclude that the CAS() operation at point σ' is not successful in either execution, which implies that (13) holds for $t = \sigma'$.

Finally, if p executes line 9 at σ', the analysis is the same as in the proof of (12), except for the justification of the following claim used there. If $\sigma' = t_2$, then in $E(\perp)$, p_2 reads the same values on $C[1-i^*][j]$, for any $j \geq \beta_{u_2}$, as in E. In our case, this follows immediately from the assumption that (13) holds for $t \leq \sigma$, and the fact that p_2 reads those values during the interval $(r_{u_2}, \sigma]$.

As before, applying (13) for $t = t^*$ implies the lemma. □

Proof of Lemma 6.18(a): Case $\alpha_v = i^*$. We consider now the case where $(\alpha_v, \beta_v) = (i^*, \hat{\jmath})$ in E, for some fixed $1 \leq \hat{\jmath} \leq \lambda$, i.e., $E = E(i^*, \hat{\jmath})$. As before we use superscript $\perp$ to denote variables in execution $E(\perp)$, when we need to distinguish them from variables in E. Similarly to (12), we proceed to show by induction that for any $t \in [r_v, t^*)$,

$$\mathcal{S}_L(E_t(\perp)) = \mathcal{S}_L(E_t) \quad \text{and} \quad C_t^{\perp}[i][j] = C_t[i][j], \text{ for all } (i,j) \neq (i^*, \hat{\jmath}), \tag{14}$$

and for $t = t^*$, $\mathcal{S}_L^-(E_t(\perp)) = \mathcal{S}_L(E_t)^-$.

The induction is on the steps of E. As for (12), we have that (14) holds for $t = r_v$, because the two executions are identical in $[0, t_v)$, and at point r_v of E a process p writes value v on $C[i^*][\hat{\jmath}]$, while at point r_v of $E(\perp)$, p does a dummy step (indicated as a step of `propose(`v`)` in $\mathcal{S}_L(E(\perp))$).

For the induction step, suppose that (14) holds for all $t \leq \sigma$, where $\sigma \in [r_v, t^*)$ and a step of E occurs at σ. Let $\sigma' \in (\sigma, t^*]$ be the point of the next step of E. By the same argument as in the proof of (12), if the step of E at σ' is the kth step of some method call, and is executed by some process p, then the next step in $E(\perp)$ after σ also occurs at point σ', is the kth step of the same method call, and is executed by p. Thus, as in (12)'s proof we just need to consider all possible steps at point σ' of the two executions that may modify S or C, i.e., lines 2, 9, 15 and 16.

If p executes line 2 or line 16 at σ', then as argued in the proof of (12), the exact same operation takes place in both executions, and thus (14) holds for $t = \sigma'$.

If p executes line 15 at σ', then by the same argument as in the proof of (12), either the same operation $C[i][j]$.`CAS(`v'`,` $\perp$`)` is executed in both executions (and has the same outcome in both), or possibly different `CAS()` operations are applied to $C[i^*][\hat{\jmath}]$. In either case, (14) holds for $t = \sigma'$.

Finally, suppose that p executes line 9 at σ'. By the same argument as in the proof of (12), the `CAS()` operations in the two executions have the same parameters val, i, ℓ, and are both successful or both not successful. If they are not successful, clearly (14) holds for $t = \sigma'$. Suppose the operations are successful, and let $(v_1, i, \ell+1)$ and $(v_2, i, \ell+1)$ be the values they write to S in E and $E(\perp)$, respectively. Then, as argued in the proof of (12), either $\sigma' = t_2$ or $\sigma' = t^*$.

If $\sigma' = t_2$, then from (9), $i = 1 - i^*$. And since (14) holds for all $t \leq \sigma$, it follows that p reads the same values in both executions on all $C[1-i^*][j]$, which implies $v_1 = v_2$. Thus (14) holds for $t = \sigma'$.

If $\sigma' = t^*$, we just need to show that $\mathcal{S}_L^-(E_t(\perp)) = \mathcal{S}_L(E_t)^-$ for $t = \sigma'$ (rather than $\mathcal{S}_L(E_t(\perp)) = \mathcal{S}_L(E_t)$), and this holds independently of the result of the `CAS()` operations.

This concludes the inductive proof of (14). Applying (14) for $t = t^*$ implies the lemma. □

Proof of Lemma 6.18(b). Fix an arbitrary $1 \leq \hat{\jmath} \leq \lambda$. We use superscripts $\hat{\jmath}$ and $\perp$ to denote variables in $E(i^*, \hat{\jmath})$ and $E(\perp)$, respectively. We show by induction that for any $t \in [r_v, t^*)$,

$$\mathcal{S}_L(E_t(i^*, \hat{\jmath})) = \mathcal{S}_L(E_t(\perp)) \quad \text{and} \quad C_t^{\hat{\jmath}}[i][j] = C_t^{\perp}[i][j], \text{ for all } (i,j) \neq (i^*, \hat{\jmath}), \tag{15}$$

and for $t = t^*$, $\mathcal{S}_L^-(E_t(i^*, \hat{\jmath})) = \mathcal{S}_L(E_t(\perp))^-$.

The induction proof is essentially the same as that of (14): For the base case, $t = r_v$, the same argument as in (14) applies, if E is replaced by $E(i^*, \hat{\jmath})$. For the induction step, the same analysis as in (14) applies, if we replace E and $E(\perp)$ by $E(\perp)$ and $E(i^*, \hat{\jmath})$, respectively. In particular, in the induction step, in the case where line 9 is executed in the next step, the same arguments made for E in (14)'s proof, can be made for $E(\perp)$ in the proof of (15), because $\mathcal{S}_L^-(E(\perp)) = \Xi$, as shown in part (a). E.g., $\mathcal{S}_L^-(E(\perp)) = \Xi$ implies that (9) holds also for $E(\perp)$.

Applying (15) for $t = t^*$ implies the lemma. □

Thus far, we have assumed $\mathcal{S}_L^-(E) = \Xi$. For the rest of the proof, we need to "unfix" $\mathcal{S}_L^-(E)$, i.e., unless we explicitly condition a probability on the event $\mathcal{S}_L^-(E) = \Xi$, we do not assume that the event holds. However, all probabilities are still *implicitly* conditioned on the fixed values α_u, β_u, for all $u \neq v$, and the fixed coins for the adversary, as described at the beginning of the proof.

Let $\mathcal{E}$ denote the event that $\mathcal{S}_L^-(E) = \Xi$. For any $1 \leq j \leq \lambda$,

$$\Pr[(\alpha_v, \beta_v) = (i^*, j) \mid \mathcal{E}] = \frac{\Pr[\mathcal{E} \mid (\alpha_v, \beta_v) = (i^*, j)] \cdot \Pr[(\alpha_v, \beta_v) = (i^*, j)]}{\Pr[\mathcal{E}]}.$$

We have $\Pr[(\alpha_v, \beta_v) = (i^*, j)] = (1/2) \cdot \pi_j$, since the choice of the random values of α_v, β_v at point r_v are independent of the other (already fixed) random choices in E. Also,

$$\Pr[\mathcal{E} \mid (\alpha_v, \beta_v) = (i^*, j)] = \Pr[\mathcal{S}_L^-(E(i^*, j)) = \Xi] = 1,$$

where the first equation holds because by fixing $(\alpha_v, \beta_v) = (i^*, j)$ in E we obtain $E(i^*, j)$, and the second equation follows from Lemma 6.18. Substituting these above yields

$$\Pr[(\alpha_v, \beta_v) = (i^*, j) \mid \mathcal{E}] = \frac{\pi_j}{2 \cdot \Pr[\mathcal{E}]}.$$

Since $\sum_{1 \leq j \leq \lambda} \Pr[(\alpha_v, \beta_v) = (i^*, j) \mid \mathcal{E}] \leq 1$ and $\sum_{1 \leq j \leq \lambda} \frac{\pi_j}{2 \cdot \Pr[\mathcal{E}]} = \frac{1}{2 \cdot \Pr[\mathcal{E}]}$, it follows $\Pr[\mathcal{E}] \geq 1/2$. Using that $1/2 \leq \Pr[\mathcal{E}] \leq 1$ above, gives $\Pr[(\alpha_v, \beta_v) = (i^*, j) \mid \mathcal{E}] \in [\pi_j/2, \pi_j]$, which implies (10).

7 BDCAS ANALYSIS

7.1 Correctness

Throughout the section we consider an arbitrarily long but finite execution on a BDCAS object B. To prove strong linearizability, we will assume w.l.o.g. that all operations on the `RepeatedChoice` object L are atomic. Note that in the complexity analysis of B, we do not make this assumption. For an operation op on B, we let $inv(op)$ and $rsp(op)$ denote the points of the invocation and response of op, respectively. If operation op does not complete, then $rsp(op) = \infty$. We will use the convention that $\infty \leq \infty$.

Claim 7.1. *Let $i \in \{0, 1\}$ and $\alpha \in M_i$ and let t be a point at which $A[\alpha] = \tau$.*

(a) τ is a task reference and $\tau.add_i = \tau$.

(b) If $\tau \neq \lambda_\alpha$, then τ is returned before t from a `newTask()` *call in line 22, and prior to t $A[\alpha]$ changes to τ with a successful* `CAS()` *operation executed in line 28 if $i = 0$, and in line 36 if $i = 1$.*

Proof. First assume there is no operation prior to t that changes the value of $A[\alpha]$. Then τ is the initial value of $A[\alpha]$. I.e., τ is a reference to the initial task λ_α with $\lambda_\alpha.add_i = \alpha$. Hence, the claim is true.

Now suppose the value of $A[\alpha]$ changes to τ at some point $t' < t$, and $A[\alpha] = \tau$ throughout $(t', t]$. Moreover, assume the claim is true throughout $[0, t')$. Then $A[\alpha]$ changes as a result of a successful `CAS()` operation in one of lines 28 and 36. Let op be that `CAS()` operation, and let p be the process executing it.

First assume op is a successful `CAS()` operation in line 28. Then α is the parameter a_0 of the `BDCAS()` operation, and so it is required to be in M_0. Moreover, τ is returned from p's $L[\alpha]$.`read()` in line 25, and thus by the specification of type `RepeatedChoice` either $\tau = \perp$, or τ must have earlier been proposed in an $L[\alpha]$.`propose(`τ`)` call. Due to the if-condition in line 27, $\tau \neq \perp$. Hence, τ was proposed in line 23. Prior to that, at some point $t'' < t'$, τ was obtained from a `newTask(`$\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$`)` call in line 22 for $a_0 = \alpha$. Thus, $\tau.add_0 = \alpha$ when τ is created. Since no line of the algorithm changes any fields of a task other than the *stat* field, $\tau.add_0 = \alpha$ throughout $[t'', \infty)$, and thus at point t. This proves (b).

Now assume op is a successful `CAS()` operation in line 36 by process p. Then at some point $t'' < t'$, process p reads τ from $A[\alpha_0]$ in line 31, where α_0 is the argument of a `finish()` call. By line 20 and line 29 (where `finish()` can be called),

and the requirements on the arguments of BDCAS() calls, we have $\alpha_0 \in M_0$. Since the claim is true throughout $[0, t')$, τ is either the initial value of $A[\alpha_0]$ or it is a task reference returned from a newTask() operation. In either case, $\tau.add_1 \in M_1$ (this is trivially true for an initial task, and otherwise it follows from line 22). By line 32, process p writes τ to $A[\tau.add_1]$ in its successful CAS() operation in line 36 at point t'. □

The following statement follows immediately from Claim 7.1.

Claim 7.2. *If τ is returned from a read of $A[\alpha]$ in one of lines 33 and 37, then $\alpha = \tau.add_1 \in M_1$. If it is returned from a read of $A[\alpha]$ in line 31, then $\alpha = \tau.add_0 \in M_0$.*

Claim 7.3. *Let $\alpha \in M_1$ and $\tau \neq \lambda_\alpha$ a task reference. If $A[\alpha] = \tau$ at point t_1, then there is a point $t_0 \leq t_1$ at which $A[\tau.add_0] = \tau$.*

Proof. Since $\alpha \in M_1$ and $\tau \neq \lambda_\alpha$, by Claim 7.1 (b), at some point $t' \leq t_1$ some process p executes a successful $A[\alpha]$.CAS(τ', τ) in line 36. Then prior to that, at some point $t_0 < t_1$, process p reads τ from some array entry $A[\alpha_0]$ in line 31. By Claim 7.1, $\alpha_0 = \tau.add_0$. Hence, $A[\tau.add_0] = \tau$ at point $t_0 < t' \leq t_1$. □

Claim 7.4. *Let $\alpha \in [m]$, $i \in \{0, 1\}$, and τ a task reference stored in $A[\alpha]$. If $\tau = \lambda_\alpha$ then $\tau.old_i = \tau.new_i = \perp$, and otherwise $\tau.old_i \prec_\alpha \tau.new_i$.*

Proof. If τ is the initial task λ_α, then by definition $\tau.old_i = \tau.new_i = \perp$. Otherwise, by Claim 7.1, τ is returned from a newTask() operation in line 22. That operation creates a task using the same argument triples as the BDCAS() operation from which it is being called, and so $\tau.old_i \prec_\alpha \tau.new_i$ follows from the definition of $\prec_\alpha$. □

Claim 7.5. *For any task τ, $\tau.stat \in \{\perp, \text{False}, \text{True}\}$, and if $\tau.stat \neq \perp$ at point t, then $\tau.stat$ does not change throughout $[t, \infty)$.*

Proof. For each task τ, initially $\tau.stat = \perp$. Therefore, the claim follows immediately from the fact that the only shared memory operations that change $\tau.stat$ are the $\tau.stat$.CAS($\perp$, True) operations in lines 34, 38 and 44, and the $\tau.stat$.CAS($\perp$, False) operation in line 39. □

7.1.1 A Task is in A When its Status Changes.

Claim 7.6. *Let $\alpha \in [m]$ and suppose at point t the value of $A[\alpha]$ changes from τ to $\tau' \neq \tau$. Then there is a value $v \in \{\text{True}, \text{False}\}$ such that $\tau.stat = v$ throughout $[t, \infty)$.*

Proof. We will show that at some point $t' < t$ some process executes $\tau.stat$.CAS($\perp, v$) for some value $v \in \{\text{True}, \text{False}\}$. Then the claim statement follows immediately from Claim 7.5.

At point t some process p executes a successful $A[\alpha]$.CAS(τ, τ') in one of lines 28 and 36. If that happens in line 36, then at some point $t' < t$ process p executes $\tau.stat$.CAS($\perp$, True) in line 34.

Now assume at point t process p executes a successful $A[\alpha]$.CAS(τ, τ') in line 28. Then prior to that it obtains τ as the return value of a finish() call in line 20. Hence, at some point $t' < t$ process p executes $\tau.stat$.CAS($\perp$, False) in line line 39 of that finish() call. □

Lemma 7.7. *Let τ be a task and suppose at some point t the value of $\tau.stat$ changes to $v \in \{\text{True}, \text{False}\}$. If $v = \text{False}$, then $A_t[\tau.add_0] = \tau$, and if $v = \text{True}$ then $A_t[\tau.add_0] = A_t[\tau.add_1] = \tau$.*

Proof. First, observe that $\tau \neq \lambda_\alpha$ for any $\alpha \in [m]$, because $\lambda.\alpha.stat = \text{True}$ initially and thus by Claim 7.5 throughout the entire execution.

At point t some process p executes a successful $\tau.stat$.CAS($\perp$, True) in one of lines 34 and 38 if $v = \text{True}$, or $\tau.stat$.CAS($\perp$, False) in line 39 if $v = \text{False}$. If $v = \text{False}$, then at some point $t_0 < t$ process p reads τ from some address $A[\alpha]$ in line 31. By Claim 7.2, $\alpha = \tau.add_0$. Similarly, if $v = \text{True}$, then at some point $t_1 < t$ process p reads τ from

some address $A[\tau.add_1]$ in one of lines 33 and 37. Moreover, in that case, by Claim 7.3 there is a point $t_0 < t_1$ at which $A[\tau.add_0] = \tau$. Hence, $A_{t_i}[\tau.add_i] = \tau$ for $i = 0$ if $v =$ False, and for all $i \in \{0, 1\}$ if $v =$ True. In either case, it follows from Claim 7.6 that $A[\tau.add_i]$ does not change throughout $[t_i, t]$. Thus, if $v =$ False then $A_t[\tau.add_0] = \tau$ and if $v =$ True then $A_t[\tau.add_0] = A_t[\tau.add_1] = \tau$. □

7.1.2 *Progression of Interpreted Values on Side 1.*

Claim 7.8. *Let $\alpha \in M_1$, and τ, τ' two distinct task references, so that at some point t the value of $A[\alpha]$ changes from τ to τ'. Then $\tau.new_1 = \tau'.old_1 \prec_\alpha \tau'.new_1$.*

Proof. By the assumption that $\alpha \in M_1$, the value of $A[\alpha]$ changes from τ to τ' only when some process p executes a successful $A[\alpha].\texttt{CAS}(\tau, \tau')$ in line 36. From the if-condition in line 35 and Claim 7.4, we obtain $\tau.new_1 = \tau'.old_1 \prec_\alpha \tau'.new_1$. □

This claim immediately yields the following:

Claim 7.9. *Let $\alpha \in M_1$, τ_1, τ_2 two task references and $t_1 < t_2$ two points in time, such that $A_{t_j}[\alpha] = \tau_j$ for each $j \in \{1, 2\}$.*

(a) If $\tau_1 \neq \tau_2$ then $\tau_1.new_1 \prec_\alpha \tau_2.new_1$.
(b) If $\tau_1 = \tau_2$, then $A_t[\alpha] = \tau_1$ for all $t \in [t_1, t_2]$.

Proof. Part (a) follows from Claim 7.8 and the transitivity of relation $\prec_\alpha$. To prove part (b), assume, for the purpose of contradiction, that $\tau_1 = \tau_2$ and there is a point $t \in [t_1, t_2]$ such that $A_t[\alpha] = \tau \neq \tau_1$. Then by part (a) $\tau_1.new_1 \prec_\alpha \tau.new_1 \prec_\alpha \tau_2.new_1 = \tau_1.new_1$. By transitivity of $\prec_\alpha$ we have $\tau_1.new_1 \prec_\alpha \tau_1.new_1$, which contradicts the requirement that $\prec_\alpha$ is irreflexive. □

Claim 7.10. *Let $\alpha \in M_1$, and τ_1, τ_2 two distinct task references, so that at some point t the value of $A[\alpha]$ changes from τ_1 to τ_2. Then at some point $t_{p@22} < t$ task τ_2 is returned from a* `newTask()` *call in line 22, and $A[\alpha] = \tau_1$ throughout $[t_{p@22}, t)$.*

Proof. Since τ_2 is not the initial task λ_α, by Claim 7.1 (b) it is obtained from a `newTask()` operation that some process p executes at point $t_{p@22} < t$. Then at some point before that, during p's `read`(α) operation in line 21, p reads some task τ_0 from $A[\alpha]$ in line 41 at some point $t_{p@41} < t_{p@22}$. Process p's `read()` operation in line 21 either returns $\tau_0.old_1$ or $\tau_0.new_1$. Hence, since p's BDCAS call does not return in that line, and by the parameters of the `newTask()` call in $t_{p@22}$,

$$\tau_2.old_1 \in \{\tau_0.old_1, \tau_0.new_1\}. \tag{16}$$

If $\tau_0 = \tau_1$, then by Claim 7.9 (b) $A[\alpha] = \tau_1$ throughout $[t_{p@41}, t) \supseteq [t_{p@22}, t)$, and the claim is true.

Hence, assume $\tau_0 \neq \tau_1$. Recall that $A[\alpha] = \tau_0$ at point $t_{p@41}$, and at point $t > t_{p@41}$ the value of $A[\alpha]$ changes from τ_1 to τ_2. Hence, by Claim 7.9 (a) and Claim 7.8,

$$\tau_0.new_1 \prec_\alpha \tau_1.new_1 = \tau_2.old_1. \tag{17}$$

In particular, $\tau_2.old_1 \neq \tau_0.new_1$ because $\prec_\alpha$ is irreflexive. Hence, by (16), $\tau_2.old_1 = \tau_0.old_1$. By Claim 7.4, either $\tau_0.old_1 = \tau_0.new_1$ or $\tau_0.old_1 \prec_\alpha \tau_0.new_1$, and thus by transitivity of $\prec_\alpha$ and (17) $\tau_2.old_1 = \tau_0.old_1 \prec_\alpha \tau_2.old_1$. Again, this contradicts the irreflexivity of $\prec_\alpha$. □

Lemma 7.11. *Let $\alpha \in M_1$. If $A[\alpha] = \tau$ at any point in the execution, then $\tau.stat \neq$ False throughout the execution.*

Proof. Let $\tau_0, \tau_1, \ldots, \tau_k$, where $\tau_j \neq \tau_{j+1}$ for all $j \in \{0, \ldots, k-1\}$, be the sequence of task references stored in $A[\alpha]$ in this order.

Assume the lemma statement is not true. Then there is a smallest index $j \in \{0, \ldots, k\}$ such that $\tau_j.stat =$ False at some point during the execution.

By definition, the initial task, $\tau_0 = \lambda_\alpha$, satisfies $\lambda_\alpha.stat = \mathsf{True}$ initially, and thus throughout the entire execution by Claim 7.5. Hence,

$$\tau_j \neq \lambda_\alpha \text{ and } j > 0. \tag{18}$$

At some point $t_{p@39}$ some process p executes a successful $\tau_j.stat.\mathtt{CAS}(\bot, \mathsf{False})$ in line 39. By Claim 7.5,

$$\tau_j.stat = \bot \quad \text{throughout } [0, t_{p@39}). \tag{19}$$

Since $\alpha \in M_1$ and $A[\alpha] = \tau_j$ at some point, $\alpha = \tau_j.add_1$ by Claim 7.1 (a). By Lemma 7.7,

$$A[\tau_j.add_1] = \tau_j \text{ at point } t_{p@39}. \tag{20}$$

Moreover, since $j > 0$ according to (18), there is a point $t^* < t_{p@39}$ at which the value of $A[\tau_j.add_1]$ changes from τ_{j-1} to τ_j. Thus, by Claim 7.10, τ_j is returned from a $\mathtt{newTask()}$ operation at some point $t_j^{new} < t_{p@31}$ and

$$A[\tau.add_1] = \tau_{j-1} \text{ throughout } [t_j^{new}, t^*). \tag{21}$$

Prior to p's $\tau_j.stat.\mathtt{CAS}(\bot, \mathsf{False})$ in line 39, at some point $t_{p@31}$, process p reads τ_j in line 31. Then $t_j^{new} < t_{p@31}$ by Claim 7.1, and so from (21) we conclude

$$A[\tau.add_1] = \tau_{j-1} \text{ throughout } [t_{p@31}, t^*). \tag{22}$$

Now recall that $A_t[\tau.add_1] = \tau_j$ for $t = t^*$ and by (20) also for $t = t_{p@39}$. Thus, by Claim 7.9 (b)

$$A[\tau.add_1] = \tau_j \text{ throughout } (t^*, t_{p@39}]. \tag{23}$$

We now consider p's execution of $\mathtt{finish()}$ during the interval $[t_{p@31}, t_{p@39}]$. By (22) and (23), in line 33 process p reads either τ_{j-1} or τ_j from $A[\tau.add_1]$. If it reads τ_j, then in line 34 it executes $\tau_j.stat.\mathtt{CAS}(\bot, \mathsf{True})$. Because that happens before $t_{p@39}$, we have a contradiction to (19). Hence, we conclude that in line 33 process p reads τ_{j-1} from $A[\tau.add_1]$. Observe that at that point p's local variable γ_0 stores τ_j, because at $t_{p@39}$ process p performs a CAS on $\tau_j.stat$. By Claim 7.8, $\tau_{j-1}.new_1 = \tau_j.old_1$. So p proceeds to evaluate the if-condition in line 35 to true, and then executes $A[\tau_j.add_1].\mathtt{CAS}(\tau_{j-1}, \tau_j)$ in line 36. By (22) and (23), $A[\tau_j.add_1] \in \{\tau_{j-1}, \tau_j\}$ immediately before that $\mathtt{CAS()}$ operation, so $A[\tau_j.add_1] = \tau_j$ immediately after it. By (22) and (23) the value of $A[\tau_j.add_1]$ remains τ_j until $t_{p@39}$. Thus, in line 37 process p reads τ_j from $A[\tau_j.add_1]$, and then in line 38 it executes $\tau_j.\mathtt{CAS}(\bot, \mathsf{True})$. Since this happens before $t_{p@39}$, we obtain a contradiction to (19). □

Lemma 7.12. *Let $\alpha \in M_1$. Suppose at point t the value of $A[\alpha]$ changes from τ to $\tau' \neq \tau$. Then the interpreted value of $B[\alpha]$ does not change at point t.*

Proof. It follows from Claim 7.6 that $\tau.stat \neq \bot$ at point t. Hence, by Lemma 7.11 $\tau.stat = \mathsf{True}$ at that point. Thus, immediately before t the interpreted value of $B[\alpha]$ is $\tau.new_1$.

Since $A[\alpha]$ changes from τ to τ' at point t, it follows from Claim 7.9 (b) that $A[\alpha] \neq \tau'$ throughout $[0, t)$. Then by Lemma 7.7, $\tau'.stat$ does not change throughout $[0, t]$. Clearly, τ' is not the initial task λ_α, so $\tau'.stat = \bot$ throughout $[0, t]$. Thus, immediately after t the interpreted value of $B[\alpha]$ is $\tau'.old_1$.

According to Claim 7.8 $\tau'.old_1 = \tau.new_1$ so the interpreted value of $B[\alpha]$ does not change at point t. □

Claim 7.13. *Let $\alpha \in M_1$, and suppose at some point the interpreted value of $B[\alpha]$ changes from b to $b' \neq b$. Then $b \prec_\alpha b'$.*

Proof. Let t be the point when the interpreted value of $B[\alpha]$ changes from b to b'. By definition, at that point either $A[\alpha] = \tau$ and $\tau.stat$ changes, or $A[\alpha]$ changes. Lemma 7.12 states that the latter is not possible. Hence, at point t the value of $\tau.stat$ changes while $A[\alpha] = \tau$. Then by Claim 7.5 and Lemma 7.11 the value of $\tau.stat$ changes from $\bot$ to True, so the interpreted value of $B[\alpha]$ changes form $b = \tau.old_1$ to $b' = \tau.new_1$. This change is a result of a $\tau.stat.\mathtt{CAS}(\bot, \mathsf{True})$ in one of lines 34 and 38. Then τ is not the initial task λ_α, because $\lambda_\alpha = \mathsf{True}$ initially and thus by Claim 7.5 also throughout the entire execution. Thus, by Claim 7.4 $b = \tau.old_1 \prec_\alpha \tau.new_1 = b'$. □

7.1.3 Linearization Points of Reads. Consider an execution E on B. We will define for each operation op in E a linearization point, $lin(op)$ (possibly $lin(op) = \infty$).

In this section we will do so for `read()` operations. Suppose op is a `read`(α) operation by process p, where $\alpha \in M_i$ for $i \in \{0, 1\}$. Let t be the point when p reads a task τ from $A[\alpha]$ in line 41, and t' the point when p reads $\tau.stat$ in line 45. If p does not read $\tau.stat$ during op, then we define $t' = \infty$. By Claim 7.1, $\alpha = \tau.add_i$. We define $lin(op)$ as the last point in $[t, t']$ at which $A[\tau.add_i] = \tau$.

The following statement is immediate from the definition of $lin(op)$ for a `read()` operation op.

Observation 7.14. *If op is a* `read()` *operation, then $inv(op) < lin(op) \le rsp(op)$.*

Lemma 7.15. *Let $\alpha \in [m]$. Each complete* `read`(α) *operation op returns the interpreted value of $B[\alpha]$ at $lin(op)$.*

Proof. Let p be the process executing op. Further, let t be the point when p reads τ from $A[\alpha]$ in line 41, and t' the point when p reads $\tau.stat$ in line 45. (Since the `read`(α) method completes, $t' < \infty$.)

First assume that $\tau.stat = \perp$ at point t'. By Claim 7.5, $\tau.stat = \perp$ throughout $[0, t']$ and thus at $lin(op)$. The `read()` method returns $\tau.old_i$ in this case, and since $A[\alpha] = \tau$ and $\tau.stat = \perp$ at $lin(op)$, this is the interpreted value of $B[\alpha]$ at that point.

Now assume that either $\tau.stat = v \in \{\mathsf{False}, \mathsf{True}\}$ at point t'. If during $[t, t']$ the value of $A[\alpha]$ changes from τ to a different task reference, then by Claim 7.6 $\tau.stat = v$ when that happens. Hence, $\tau.stat = v$ at $lin(op)$. The `read()` method returns $\tau.old_i$ if $v = \mathsf{False}$ and $\tau.new_i$ if $v = \mathsf{True}$. Since $A[\alpha] = \tau$ at $lin(op)$, this is the interpreted value of $B[\alpha]$ at that point. □

7.1.4 No ABAs on Side 0.

Lemma 7.16. *Let $\alpha \in M_0$. If $A[\alpha] = \tau$ at point t and $A[\alpha] \neq \tau$ at point $t' > t$, then $A[\alpha] \neq \tau$ throughout $[t', \infty)$.*

Proof. Let $\tau_0, \tau_1, \ldots, \tau_k$ be the first $k+1$ values of $A[\alpha]$ in chronological order, such that $\tau_{i+1} \neq \tau_i$ for each $i \in [k]$. I.e., τ_0 is the initial task λ_α.

We will prove by induction on k that $\tau_0, \ldots, \tau_k$ are all distinct. Obviously this is true for $k = 0$. Hence, suppose $k \ge 1$ and $\tau_0, \ldots, \tau_{k-1}$ are all distinct. By definition, $\tau_k \neq \tau_{k-1}$. Thus, it suffices to show that $\tau_k \notin \{\tau_0, \ldots, \tau_{k-2}\}$.

Since $\alpha \in M_0$, by Claim 7.1 (b), each τ_i, $i \in \{1, \ldots, k\}$, is written to $A[\alpha]$ in a successful operation $A[\alpha]$.`CAS`(τ_{i-1}, τ_i) in line 28. Let t_i be the point of that `CAS()` operation, and let p_i be the process performing it. Further, let $t_0 = 0$ (which is the point when the execution begins). For $i \ge 1$, τ_i is the return value of the $L[\alpha]$.`read()` operation that p_i executes in line 25. Let $t_i^L < t_i$ be the point of that $L[\alpha]$.`read()` call. Thus,

$$\forall i \in \{1, \ldots, k\}\colon L[\alpha].val = \tau_i \text{ at point } t_i^L. \tag{24}$$

Furthermore, p_i obtains τ_{i-1} from a `finish()` call in line 20, and thus it reads τ_{i-1} from $A[\alpha]$ in line 31 at some point $t_{i-1}^{fin} < t_i^L < t_i$. By definition and the inductive hypothesis, t_{i-1} is the earliest point in the execution such that $A_{t_{i-1}}[\alpha] = \tau_{i-1}$. Hence, $t_{i-1} < t_{i-1}^{fin}$ for $i \in \{1, \ldots, k\}$. Thus, to summarize we have $t_{i-1} < t_{i-1}^{fin} < t_i^L < t_i$, and thus

$$t_{i-1}^L < t_i^L \text{ for all } i \in \{2, \ldots, k\}. \tag{25}$$

Now assume for the purpose of a contradiction that $\tau_k = \tau_\ell$ for some $\ell \in \{0, \ldots, k-2\}$. Since τ_k is not the initial task (because it was read from $L[\alpha]$ in line 25), we have $\ell \in \{1, \ldots, k-2\}$. Then by (24), $L[\alpha].val = \tau_k$ at points t_ℓ^L and t_k^L, but it equals $\tau_{k-1} \neq \tau_k$ at point t_{k-1}^L. From (25) we conclude $t_\ell^L < t_{k-1}^L < t_k^L$. This contradicts that, according to the `RepeatedChoice` specification $L[\ell].val$ is ABA-free. □

7.1.5 Progression of Interpreted Values on Side 0.

Claim 7.17. *Suppose a* `finish()` *call returns task τ at point t. Then there is a value $v \in \{\mathsf{False}, \mathsf{True}\}$ such that $\tau.stat = v$ throughout $[t, \infty)$.*

Proof. This is immediate from Claim 7.5 and the fact that a process executes $\tau.stat$.CAS($\perp$, False) in line 39 before returning τ in its finish() call. □

Lemma 7.18. *Let $\alpha \in M_0$. Suppose at point t the value of $A[\alpha]$ changes from τ to $\tau' \neq \tau$. Then at that point*

(a) $\tau'.stat = \perp$, and

(b) either $\tau.stat =$ True and $\tau'.old_0 = \tau.new_0$, or $\tau.stat =$ False and $\tau'.old_0 = \tau.old_0$.

In particular, the interpreted value of $B[\alpha]$ does not change at point t.

Proof. By Claim 7.1 at point $t_{p@28} = t$ some process p executes a successful $A[\alpha]$.CAS(τ, τ') in line 28, and $\alpha = \tau'.add_0$. By Lemma 7.16, $A[\alpha] \neq \tau'$ throughout $[0, t_{p@28})$. Hence, by Lemma 7.7, $\tau'.stat = \perp$ throughout $[0, t_{p@28}]$. This proves (a).

Let $t_{read} \in [t_{p@31}, t_{p@28}]$ be the linearization point of p's last read(α) operation in line 21, before p executes its successful CAS at point $t_{p@28}$. Prior to t_{read}, process p completes a finish(α) call that returns τ in line 20. Hence, by Claim 7.17,

$$\exists v \in \{\text{True}, \text{False}\} : \tau.stat = v \text{ throughout } [t_{read}, \infty). \tag{26}$$

Since $A[\alpha] = \tau$ when p reads $A[\alpha]$ in line 31 of its last finish(α) call before t_{read}, and $A[\alpha] = \tau$ immediately before point $t_{p@28}$ (because the CAS at that point is successful), by Lemma 7.16, $A[\alpha] = \tau$ at point t_{read}. Thus, by Lemma 7.15 and (26),

$$B_{t_{read}}[\alpha] = \begin{cases} \tau.old_0 & \text{if } v = \text{False ; and} \\ \tau.new_0 & \text{otherwise.} \end{cases} \tag{27}$$

Due to the while-loop condition, $B_{t_{read}}[\alpha]$ equals the parameter old_0 of p's BDCAS call. Since just before $t_{p@28}$ process p evaluates the if-condition in line 27 to true, $\tau'.old_0$ also equals that parameter, and in particular $\tau'.old_0 = B_{t_{read}}[\alpha]$. By (27) $\tau'.old_0 = \tau.old_0$ if $v =$ False and $\tau.old_0 = \tau.new_0$ if $v =$ True. By (26) $v = \tau.stat$ at point $t = t_{p@28}$, and so the proof is complete. □

Corollary 7.19. *Fix $\alpha \in M_0$, and let $b_1, \ldots, b_k$ be the interpreted values of $B[\alpha]$ obtained in this order (i.e., $b_{i+1} \neq b_i$ for $i \in \{1, \ldots, k-1\}$). Then $b_i \prec_\alpha b_j$ for all $1 \leq i < j < k$.*

Proof. By the definition of the interpreted value and Lemma 7.18, the interpreted value of $B[\alpha]$ can only change if $A[\alpha] = \tau$ and $\tau.stat$ changes. Since $\tau.stat$ can only change from False to True (by Claim 7.5), the interpreted value of $B[\alpha]$ can only change from $\tau.old_0$ to $\tau.new_0$. By line 22, these are the corresponding values old_0 and new_0 of an argument triple of a BDCAS() call, so by definition of $\lhd_\alpha$, $\tau.old_0 \lhd_\alpha \tau.new_0$. Hence, the claim follows from the fact that $\prec_\alpha$ is the transitive closure of $\lhd_\alpha$. □

7.1.6 Linearizability of BDCAS Operations. Before we define linearization points for BDCAS() operations, we will make some simple observations about interpreted values and status changes of tasks.

Claim 7.20. *Let τ be a task, and suppose at point t the value of $\tau.stat$ changes to* True*. Then at that point the interpreted value of $B[\tau.add_i]$, $i \in \{0, 1\}$, changes from $\tau.old_i$ to $\tau.new_i$, where $\tau.old_i \prec_{\tau.add_i} \tau.new_i$*

Proof. Let $\alpha = \tau.add_i$. By Lemma 7.7, $A[\alpha] = \tau$ at point t. By Claim 7.5, $\tau.stat = \perp$ throughout $[0, t)$. In particular, τ is not the initial task λ_α stored in $A[\alpha]$, because $\lambda_\alpha.stat =$ True initially (and throughout the execution). Thus, by Claim 7.4 $\tau.old_i \prec_\alpha \tau.new_i$. As $A[\alpha] = \tau$ at point t and $\tau.stat$ changes from $\perp$ to True at point t, by definition, $B[\alpha]$ changes from $\tau.old_i$ to $\tau.new_i$ at that point. □

Claim 7.21. *For each* BDCAS() *operation by some process p, there is at most one task that p creates in line 22, whose status changes to* True *during the execution.*

Proof. For the purpose of a contradiction, suppose there are two tasks, τ_1 and τ_2, that process p creates using newTask() calls in line 22 of the same BDCAS() operation, and $\tau_1.stat$ changes to True at point t_1 and $\tau_2.stat$ does so at point $t_2 > t_1$.

Since τ_1 and τ_2 are created during the same BDCAS() operation, both are created using the same arguments to the newTask() call in line 22. In particular, $\alpha = \tau_1.add_1 = \tau_2.add_1$, and $\tau_1.new_1 = \tau_2.new_1$.

By Lemma 7.7, $A[\alpha] = \tau_i$ at point t_i for each $i \in \{1, 2\}$. Then by Claim 7.9, $\tau_1.new_1 \prec_\alpha \tau_2.new_1$. Since relation $\prec_\alpha$ is irreflexive, this contradicts $\tau_1.new_1 = \tau_2.new_1$. □

Claim 7.22. *Let $i \in \{0, 1\}$ and suppose process p executes a* BDCAS$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$ *operation in during which it creates a task τ in line 22. Let t_i be the linearization point of p's* read(α_i) *operation in line 21 during the while-loop iteration in which p creates task τ. If the interpreted value of $B[\alpha_i]$ changes at some point $t > t_i$, and $\tau.stat_t \neq$ True then $\tau.stat \neq$ True throughout the entire execution.*

Proof. Since τ is obtained in line 22, $\tau.add_i = \alpha_i$. For the purpose of a contradiction assume that $\tau.stat =$ True at some point during the execution. Then by Claim 7.5 and the assumption that $\tau.stat_t \neq$ True, it follows that $\tau.stat$ changes to True at some point $t^* > t$. By Lemma 7.7, $A[\alpha_i] = \tau$ at point t^*, so at that point the interpreted value of $B[\alpha_i]$ changes from $\tau.old_i$ to $\tau.new_i$. Since p's BDCAS call does not return in line 21 after the read(α_i) operation that linearizes at point t_i, by Lemma 7.15 the interpreted value of $B[\alpha_i]$ equals $\tau.old_i$ at point t_i. Thus, we showed that $B[\alpha_i]$ at t_i and immediately before t^*. Then by Corollary 7.19 for $i = 0$ and Claim 7.13 for $i = 1$, the interpreted value of $B[\alpha_i]$ equals $\tau.old_i$ throughout $[t_i, t^*)$. This contradicts the assumption that the interpreted value of $B[\alpha_i]$ changes at point $t \in (t_i, t^*)$. □

Definition of Linearization Points. Let op be a BDCAS$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$ operation invoked by process p. If during the execution the status of one of the tasks created by p in line 22 changes to True, then let $t_{stat}(op)$ be the first point when this happens, and otherwise $t_{stat}(op) = \infty$. If one of p's operations read(α_i), $i \in \{0, 1\}$, in line 21 returns a value different from v_i, then let $t_{read}(op)$ be the linearization point of the first such read() operation, and otherwise $t_{read}(op) = \infty$. We define $lin(op) = \min\{t_{stat}(op), t_{read}(op)\}$.

We say a BDCAS() operation op is *successful*, if $t_{stat}(op) < \infty$, and otherwise it is unsuccessful.

Claim 7.23. *If op is a successful* BDCAS$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$ *operation, then $t_{stat}(op) = lin(op)$.*

Proof. Let p be the process executing op. If op is successful, then p creates a task τ during op and at point $t_{stat}(op)$ during the execution $\tau.stat$ changes to True. If $t_{read}(op) = \infty$, then $t_{stat}(op) < t_{read}(op)$, and so $lin(op) = t_{stat}(op)$. Hence, assume $t_{read}(op) < \infty$.

During each iteration of the while-loop, p executes read(α_0) and read(α_1) in line 21, where $\alpha_i = \tau.add_i$ for each $i \in \{0, 1\}$. In the iteration in which process p creates task τ, this read(α_i) returns v_i, because p's BDCAS operation does not return in line 21. Let t_i be the linearization point of that read(α_i). Then $B_{t_i}[\alpha_i] = v_i$ for each $i \in \{0, 1\}$.

Since $t_{read}(op) < \infty$, there is an index $j \in \{0, 1\}$, such that at point $t_{read}(op) > t_j$ process p's read(α_j) in line 21 returns a value that is different from v_j. I.e., $B_{t_j'}[\alpha_j] \neq B_{t_j}[\alpha_j]$. Then the interpreted value of $B[\alpha_j]$ changes at some point in $t^* \in [t_j, t_{read}(op))$. From Claim 7.22 and the fact that $\tau.stat =$ True at some point during the execution, it follows that $\tau.stat_{t^*} =$ True. Hence, $t_{stat}(op) < t^* < t_{read}(op)$. Thus, by definition, $lin(op) = t_{stat}(op)$. □

Lemma 7.24. *For any address $\alpha \in [m]$ and any point t during the execution, the following is true:*

(a) If at point t the interpreted value of $B[\alpha]$ changes from v to $v' \neq v$, then there is a successful BDCAS() *operation op such that $lin(op) = t$, and one of the argument triples used in the invocation of op is $\langle\alpha, v, v'\rangle$.*

(b) Let op be a successful BDCAS$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$ *operation. Then for each $i \in \{0, 1\}$ at point $lin(op)$ the interpreted value of $B[\alpha_i]$ changes from v_i to v_i', and $lin(op)$ is between the invocation and response of op.*

Proof. We first prove (a). Suppose at point t the interpreted value of $B[\alpha]$ changes from v to $v' \neq v$. By definition, this can only happen if either $A[\alpha] = \tau$ and the value of $\tau.stat$ changes, or if the value $A[\alpha]$ changes. If $A[\alpha]$ changes, then the interpreted value of $B[\alpha]$ is not affected: this follows from Lemma 7.12 for $\alpha \in M_1$ and from Lemma 7.18 for $\alpha \in M_0$. Hence, at point t the value of $\tau.stat$ changes, where τ is a task stored in $A[\alpha]$ at that point. By Claim 7.5, $\tau.stat$ can only

change from $\perp$ to False or True. If it changes from $\perp$ to False, then the interpreted value of $B[\alpha]$ is not affected, so at point t $\tau.stat$ changes from $\perp$ to True. Hence, by Claim 7.20 at that point $B[\tau.\alpha_i]$ changes from $\tau.old_i$ to $\tau.new_i$ for $i \in \{0,1\}$. Observe that the change of $\tau.stat$ at point t can only affect the interpreted value of an entry $B[\alpha]$, where $A[\alpha] = \tau$. Thus, by Claim 7.1, $\alpha = \tau.add_i$ for some $i \in \{0,1\}$, and so $v = \tau.old_i$ and $v' = \tau.new_i$. Clearly, τ is not the initial task λ_α, because $\lambda_\alpha.old_i = \lambda_\alpha.new_i$ whereas $v \neq v'$. Then by Claim 7.1, τ is obtained from a `newTask()` operation in line 22 of a `BDCAS()` call op. From that line we conclude that op is a `BDCAS()` call with the argument triple $\langle \tau.add_i, \tau.old_i, \tau.new_i \rangle = \langle \alpha, v, v' \rangle$. By Claim 7.21, τ is the only task created by op whose status changes to True. Hence, $t_{stat}(op) = t$, and thus op is a successful BDCAS operation. Moreover, by Claim 7.23, $t_{stat}(op) < t_{read}(op)$, so $lin(op) = t_{stat}(op) = t$. This proves (a).

We now prove (b). Let p be the process executing the successful `BDCAS`$(\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle)$ operation op. By Claim 7.23, $lin(op) = t_{stat}(op)$. Process p creates a task τ in line 22 with $\tau.add_i = \alpha_i$, $\tau.old_i = v_i$ and $\tau.new_i = v'_i$ for each $i \in \{0,1\}$, and

$$\tau.stat \text{ changes to True at point } t_{stat}(op) = lin(op). \tag{28}$$

Thus, by Claim 7.20, at point $lin(op)$ the interpreted value of $B[\alpha_i]$, $i \in \{0,1\}$, changes from v_i to v'_i.

It remains to show that $t_{stat}(op) = lin(op)$ is between the invocation and response of op. Clearly, $\tau.stat$ can only change to True after p creates τ in line 22, and thus after p's invocation of op. Moreover, since $t_{stat}(op) = lin(op)$, we have $t_{stat}(op) < t_{read}(op)$. Recall that $t_{read}(op)$ is the linearization point of p's first `read`(α_i) in line 21 that returns a value different from v_i for some $i \in \{0,1\}$. Therefore, op does not respond before $t_{read}(op) > t_{stat}(op) = lin(op)$. □

Lemma 7.25. *Let op be an unsuccessful* `BDCAS`$(\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle)$ *operation. Then $inv(op) < lin(op) \leq rsp(op)$, and if $lin(op) < \infty$, then there is an index $i \in \{0,1\}$ such that $B_t[\alpha_i] \neq v_i$ at point $t = lin(op)$.*

Proof. Let p be the process executing op. If $lin(op) = \infty$, then $t_{read}(op) = \infty$. Then p does not break out of the while-loop, so $inv(op) < lin(op) = \infty = rsp(op)$. Hence, assume $lin(op) < \infty$.

By the assumption that op is unsuccessful $t_{stat} = \infty$. Thus, $lin(op) = t_{read}(op)$, so there is an index $j \in \{0,1\}$ such that at $lin(op)$ one of p's `read`(α_j) operations in line 21 linearizes, and that `read()` returns a value $u \neq v_j$. Then obviously $inv(op) < t < rsp(op)$, and $B_t[\alpha_j] = u \neq v_j$ by Lemma 7.15. □

Theorem 7.26. *The BDCAS implementation is strongly linearizable.*

Proof. Consider an execution E on an instance B of the BDCAS implementation, and let O be the set of operations op on B for which $lin(op) < \infty$. Order all operations $op \in O$ by their linearization points $lin(op)$, and let S be the resulting sequential history.

By Observation 7.14 and Lemmas 7.24 and 7.25 for each operation $op \in O$, $lin(op)$ is between the invocation and the response of op. Note also that O contains all operations that respond in E (because $lin(op) = \infty$ only if op does not respond).

We will now prove that S is valid. Consider an arbitrary `BDCAS`$(\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle)$ operation in E. By Lemma 7.24 that operation is successful if and only if at its linearization point the interpreted value of $B[\alpha_i]$ changes from v_i to v'_i for $i \in \{0,1\}$, and the interpreted value of $B[\alpha_i]$ can only change at the linearization point of such a successful `BDCAS()`. Hence, for each $\alpha \in [m]$, after the k-th operation in S, the value of $B[\alpha]$ is the same as the interpreted value of $B[\alpha]$ after the first k operations in E have linearized. Since each `read`(α) operation in E returns the interpreted value of $B[\alpha]$ at its linearization point (by Lemma 7.15), and each unsuccessful `BDCAS`$(\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle)$ operation linearizes at a point when the interpreted value of $B[\alpha_i]$ does not equal v_i for at least one $i \in \{0,1\}$, the sequential execution S is valid.

It is obvious that lin defines strong linearization points: Consider a prefix E' of E that ends at point $t = lin(op)$ (as defined for E). Then from the definition of lin it is obvious that op also linearizes at point t in E'. Hence, each operation in E' linearizes at the same point in E' as in E. Since this is true for all prefixes E' of E, strong linearizability follows. □

7.2 Step Complexity

In this section, we prove a logarithmic upper bound on the expected amortized step complexity of BDCAS() operations. In the following, we do not distinguish between a task reference and the task object itself, and use the term *task* to refer to both.

Adversary. We consider an execution on a BDCAS object B scheduled by a *weak adaptive* adversary. The adversary has only a partial view of the internal data of B, as described below, which however suffices to compute the interpreted value of $B[a]$, for each $a \in [m]$, at any point. The adversary cannot see the internal state of processes, including the outcome of their coin-flips .

To specify the adversary's view of B, we make the (theoretical) assumption that each task is associated with a unique *task-id*, which is independent of the value of a task or the point at which it was created. Concretely, each task-id is a real number chosen independently and uniformly at random from $[0, 1)$.

The adversary can see all the tasks created, but does not see the task-id associated with a task τ if $\tau.stat = \perp$. However, as soon as $\tau.stat$ becomes True or False, the task-id of τ is immediately revealed to the adversary.

On the other hand, at each point t, the adversary can see the task-id of $A_t[a]$, for each $a \in [m]$, but does not see which one is the actual task τ associated with that task-id, unless $\tau.stat \in \{\text{True}, \text{False}\}$. Similarly, the adversary can see the task-id of each $L_t[a]$, $a \in M_0$, if $L_t[a] \neq \perp$. As in Section 6.2, the adversary can see also the two other components of $L[a].S$, but cannot see $L[a].C$.

Finally, whenever a command $A[a]$.CAS() is executed, immediately after that, the adversary sees that a CAS() was applied to $A[a]$ (even if the CAS() was unsuccessful). Thus the adversary knows when lines 28 and 36 were executed.

Based on all the above information, after each step the adversary chooses the point in time of the next step and the process to take that step, and if the process has completed its previous method call of B (or has not invoked any method yet), the adversary decides the process's next method call. As before, we allow the adversary to make probabilistic decisions.

It is easy to verify that at any point t, for each method call of B or $L[a]$, $a \in M_0$, invoked before t, the adversary can infer the call's name and argument (if any), its invocation point, and its return point if the call is completed.

Main Theorem. The next theorem states the main result of this section.

Theorem 7.27. *Let E be a random execution of finite expected length, in which an adversary as described above schedules calls to the methods of a BDCAS object B. For any $a \in M_0$, if O_a denotes the number of* BDCAS() *operations on address a that are invoked in E, then the expected total number of steps of those operations is at most $c \log n \cdot \mathbf{E}[O_a]$, where c is a constant.*

7.2.1 Notation.

A Stronger Adversary. In the proof of Theorem 7.27 we use a stronger adversary, which sees the complete internal state of B except for $B.A[x]$ and $B.L[x]$, for a specified $x \in M_0$. Precisely, it sees all that the weaker adversary described above can see, and in addition it can see (1) the task-id of a task τ with $\tau.stat = \perp$, if $\tau.add_0 \neq x$ or $A[a] = \tau$ for some $a \in M_1$, and (2) for each $a \in M_0 \setminus \{x\}$, the task-id of each non-$\perp$ entry of array $L[a].C$.

Schedules. For any execution E on B and $x \in M_0$, we define $\mathcal{S}_{B,x}(E)$ as follows. For each step σ of E, $\mathcal{S}_{B,x}(E)$ indicates: (i) the point when σ takes place, (ii) the process p that executes σ, (iii) if p invokes a new method call on B at step σ, the name and arguments of that call, (iv) if a new task τ is created at σ, the values of its fields, and if $\tau.add_0 \neq x$, also its task-id, (v) the task-id of a task τ with $\tau.add_0 = x$, if $\tau.stat$'s value changes to True or False at σ, or the value of $A[\tau.add_1]$ changes to τ, and (vi) whether a CAS() command was applied to some $A[a]$ at σ; also if any of the following changes at σ, its new value is indicated: (vii) the task-id of $A[a]$, for each $a \in [m]$, (viii) the complete state of $L[a]$, for each $a \in M_0 \setminus \{x\}$, and (ix) the task-id of the first component of $L[x].S$, if it is not $\perp$, and the values of the other two components.

If E is finite, we define $\mathcal{S}^-_B(E)$ identically to $\mathcal{S}_{B,x}(E)$, except that for the last step σ of E, $\mathcal{S}^-_{B,x}(E)$ indicates only information (i)–(iii), and does not indicate (iv)–(ix).

Note that if E is finite and is scheduled by the adversary above, $\mathcal{S}_{B,x}(E)$ contains all the information that the adversary knows (including all its own decisions) after the last step in E, and $\mathcal{S}^{-}_{B,x}(E)$ contains all that it knows immediately before the last step takes effect.

Doomed & Successful Tasks. We say a process creates task τ when a `newTask()` operation in line 22 returns τ. Suppose a process creates task τ at point $t_{\tau\text{-new}}$. For any $t \geq t_{\tau\text{-new}}$, we say that task τ is *doomed* at point t, if there is a point $t_1 \in [t_{\tau\text{-new}}, t]$ such that one of the following is true:

(d1) $A_{t_0}[\tau.add_1] \neq \tau$ for all $t_0 \leq t_1$, and $A_{t_1}[\tau.add_1].new_1 \neq \tau.old_1$, or
(d2) $A_{t_0}[\tau.add_0] \neq \tau$ for all $t_0 \leq t_1$, and
  (d2-a) $B_{t_1}[\tau.add_0] \neq \tau.old_0$, or
  (d2-b) $A_{t_1}[\tau'.add_1] = \tau'$ for some non-initial task τ', where $\tau'.add_0 = \tau.add_0$ and $\tau'.old_0 = \tau.old_0$.

On the other hand, task τ is *successful* at point t if $A_{t_0}[a_1] = \tau$ for some $t_0 \leq t$. Each initial task λ_α, $\alpha \in [m]$, is successful and not doomed at any point t of the execution. For any $t \geq 0$, we denote by D_t and S_t the set of tasks that are respectively doomed or successful at t.

Disclosed & Resolved Task-Ids. We say that a task-id is *disclosed* in execution prefix E_t, if it is contained in $\mathcal{S}_{B,x}(E_t)$, and it is *resolved* in E_t, if $\mathcal{S}_{B,x}(E_t)$ indicates which task τ is associated with that task-id. We also say a task-id is disclosed (or resolved) at t, if it is disclosed (resp. resolved) in E_t, but not in $E_{t'}$ for any $t' < t$. E.g., the task-id of a (non-initial) task τ with $\tau.add_0 \neq x$ is resolved (and thus also disclosed) as soon as τ is created. While if $\tau.add_0 = x$, the task-id of τ is disclosed when τ is stored on $L[x].S$, and is resolved when $A[\tau.add_1] = \tau$ or $\tau.stat \neq \perp$. We say that *task* τ is resolved, when its task-id is resolved. For any initial tasks λ_j, we assume it is resolved at point $t = 0$.

7.2.2 Proof of Theorem 7.27. In the following we omit prefix B when we refer to methods or variables of B, e.g., we write $L[a]$ instead of $B.L[a]$.

We assume that no `BDCAS()` operation is pending at the end of E. We can make this assumption w.l.o.g., because we can always construct a finite extension E' of E with that property as follows. After the last step of E, we consider all processes p one by one (in some predetermined order), and if p has a pending `BDCAS()` operation, we let p run solo until the operation returns. Then E' has the same number of `BDCAS()` operations and at least the same number of steps as E, thus it suffices to analyze E'. The fact that E' is finite follows from Lemma 7.50.

We also assume that the execution is scheduled by the stronger adversary described in Section 7.2.1, where x equals the address $a \in M_0$ in the theorem's statement.

Let us fix an arbitrary $x \in M_0$. Let N denote the total number of successful $L[x]$.`choose&lock()` calls that linearize in E, and for $1 \leq k \leq N$, let t_k denote the linearization point of the kth such call. Note that N is a random variable and $\mathbf{E}[N] < \infty$, since the expected length of E is finite. Similarly, let N' be the number of successful $L[x]$.`unlock()` calls that linearize in E, and for $1 \leq k \leq N'$, let s_k be the linearization point of the kth such call. From Lemma 6.4, it follows that $N' \in \{N-1, N\}$, and that $t_k < s_k$ for $1 \leq k \leq N'$, and $s_k < t_{k+1}$ for $1 \leq k < N$.

For $1 \leq k \leq N$, let $\tau_k = L_{t_k}[x]$. For $3 \leq k \leq N$, let Q_k be the set of all tasks τ such that τ is proposed in the interval (t_{k-2}, t_k), $\tau.add_0 = x$, and $\tau \neq \tau_{t_{k-1}}$. Also, let P_k be the subset of tasks $\tau \in Q_k$ proposed in (s_{k-2}, s_{k-1}), and let $p_k = |P_k|$. The next claim follows from Theorem 6.11, and bounds the distribution of τ_k given $\mathcal{S}^{-}_{B,x}(E_{t_k})$.

Claim 7.28. *Let $k \geq 3$, suppose that $N \geq k$ and $p_k = \ell$, and fix $\mathcal{S}^{-}_{B,x}(E_{t_k})$. We have that $\Pr[\tau_k \in Q_k \cup \{\perp\}] = 1$, $\Pr[\tau_k = \perp] \leq 2^{-\ell}$, and, if $\ell > 0$, $\Pr[\tau_k = \tau] \leq 8/\ell$ for any $\tau \in Q_k$.*

Proof. We prove the claim for an even stronger adversary $\mathcal{A}$, which can see the task-id of any task. Therefore, the adversary sees the values of all variables of B except for $L[x].C$'s.

We assume E is scheduled by the above adversary $\mathcal{A}$, and let $\hat{E}$ be the restriction of E consisting only of the steps of method calls to $L[x]$.

Let $\mathcal{A}'$ be an adversary that schedules an execution E' on a single RepeatedChoice object L', such that E' has the same distribution as $\hat{E}$ (where $L[x]$ is replaced by L' in E'). Adversary $\mathcal{A}'$ has the same power as the adversary described in Section 6.2, i.e., $\mathcal{A}'$ can see $L'.S$ but not $L'.C$, and proceeds by simulating the steps in $E \setminus \hat{E}$ (using its random coins), and for each call to a method of $L[x]$ in $\hat{E}$, $\mathcal{A}'$ schedules the same call to L' in E'.

Let $\mathcal{I}_t$ indicate which task-id is associated with each task created in E_t, and note that all information $\mathcal{A}$ knows about E_{t_k} before point t_k is contained in $\mathcal{S}^-_{B,x}(E_{t_k})$ and $\mathcal{I}_{t_k}$. If we fix $\mathcal{S}^-_{B,x}(E_{t_k}) = \Xi$ and $\mathcal{I}_{t_k} = I$, then $\mathcal{S}^-_{L[x]}(\hat{E}_{t_k})$[4] is also fixed, say $\mathcal{S}^-_{L[x]}(\hat{E}_{t_k}) = \hat{\Xi}$, but no additional information about $\hat{E}_{t_k}$, and in particular about $L_{t_k}[x]$, is revealed that cannot be inferred from $\mathcal{S}^-_{L[x]}(\hat{E}_{t_k}) = \hat{\Xi}$.

It follows that the conditional distribution of $L_{t_k}[x]$ in E given $\mathcal{S}^-_{B,x}(E_{t_k}) = \Xi$ and $\mathcal{I}_{t_k} = I$, is the same as the conditional distribution of L'_{t_k} in E' given $\mathcal{S}^-_{L'}(E'_{t_k}) = \hat{\Xi}$. Applying Theorem 6.11 to E' then proves the claim. □

Let point ρ_k, for $1 \le k \le N$, be defined as follows: if $\tau_k = \perp$ then $\rho_k = t_k$, otherwise, ρ_k is the first point $t \ge t_k$ for which we have $\tau_k \in D_t \cup S_t$. From Lemma 7.34 and the assumption that no calls are pending at the end of the execution, it follows that point ρ_k always exists. The next claim computes the conditional distribution of τ_k given $\mathcal{S}_{B,x}(E_t)$, for $t_k \le t \le \rho_k$, in terms of τ_k's distribution given $\mathcal{S}^-_{B,x}(E_{t_k})$. (The latter distribution was bounded in Claim 7.28.) The claims says essentially that at any step in $[t_k, \rho_k]$, the information that τ_k in not in the set D of tasks doomed before the step, does not provide any information about which one of the remaining tasks $\tau \in Q_k \setminus D$ is τ_k.

Claim 7.29. *Let $k \ge 3$, and suppose that $N \ge k$. Fix $\mathcal{S}^-_{B,x}(E_{t_k})$, and let $\pi(\tau) = \Pr[\tau_k = \tau]$, for each $\tau \in Q_k \cup \{\tau\}$. Let δ_j, for $j \ge 1$, denote the point when the jth step after t_k is executed in E; let also $\delta_0 = t_k$ and $\delta_{-1} = 0$. For $j \ge 0$, let $\mathcal{E}_j = \{\rho_k \ge \delta_j\} \cap \{\tau_k \ne \perp\} \cap \{\tau_k \notin S_{\delta_j}\}$. Fix some $j \ge 0$, suppose the event $\mathcal{E}_j$ holds, and fix the set $D_{\delta_{j-1}}$. Then, for any task $\tau \in Q_k \setminus D_{\delta_{j-1}}$, $\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j})] = \pi(\tau)/\pi(Q_k \setminus D_{\delta_{j-1}})$, where $\pi(Q) = \sum_{\tau' \in Q} \pi(\tau')$.*

Proof. The proof is by induction on j. We fix $\mathcal{S}^-_{B,x}(E_{t_k})$, and define π, δ_j, and $\mathcal{E}_j$ as in the statement of the claim. We show that for any $j \ge 0$,

$$\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j}); \mathcal{E}_j] = \pi(\tau)/\pi(Q_k \setminus D_{\delta_{j-1}}), \quad \text{if } \tau \in Q_k \setminus D_{\delta_{j-1}}. \tag{29}$$

First, we consider the base case $j = 0$. For the event $\mathcal{E}_0$, we have

$$\mathcal{E}_0 = \{\rho_k \ge t_k\} \cap \{\tau_k \ne \perp\} \cap \{\tau_k \notin S_{t_k}\} = \{\tau_k \in Q_k\},$$

because $\tau_k \in Q_k \cup \{\perp\}$ from Claim 7.28, inequality $\rho_k \ge t_k$ is by definition always true, and also $\tau_k \notin S_{t_k}$ is always true. That $\tau_k \notin S_{t_k}$ follows from Lemma 7.39, since the earliest point when any task $\tau \in Q_k$ can be written to $L[x].S$ is t_k. It follows that for any $\tau \in Q_k$,

$$\Pr[\tau_k = \tau \mid \mathcal{E}_0] = \Pr[\tau_k = \tau \mid \tau_k \in Q_k] = \Pr[\tau_k = \tau]/\Pr[\tau_k \in Q_k] = \pi(\tau)/\pi(Q_k).$$

Recall that we have fixed $\mathcal{S}^-_{B,x}(E_{t_k})$, but not $\mathcal{S}_{B,x}(E_{t_k})$. We now argue that, for any $\tau \in Q_k$,

$$\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{t_k}); \mathcal{E}_0] = \Pr[\tau_k = \tau \mid \mathcal{E}_0] = \pi(\tau)/\pi(Q_k). \tag{30}$$

Given $\mathcal{E}_0$, i.e., $\tau_k \in Q_k$, the only difference between $\mathcal{S}^-_{B,x}(E_{t_k})$ and $\mathcal{S}_{B,x}(E_{t_k})$, is that the latter indicates also the task-id of τ_k. Since the earliest point that any $\tau \in Q_k$ can be written to $L[x].S$ is t_k (and since $L[x].C$ is not visible), it follows from Lemma 7.39 that for any $\tau \in Q_k$, τ's task-id is not disclosed[5] before t_k. Moreover, since a task-id is just a random number from $[0, 1)$, the task-id disclosed at t_k does not give any information about τ_k's distribution. This proves (30). Since $D_{\delta_{-1}} = D_0 = \emptyset$, (30) implies (29) for $j = 0$.

[4] Defined in Section 6.2.

[5] Recall that a task-id is *disclosed* in E_t if it is contained in $\mathcal{S}_{B,x}(E_t)$, and is *resolved* in E_t if $\mathcal{S}_{B,x}(E_t)$ indicates which task τ is associated with it.

For the induction step, we fix some $j \geq 0$, and assume that (29) holds for that j. Observe that $\mathcal{E}_j \cap \{\tau_k \in Q_k \setminus D_{\delta_j}\} = \{\rho_k > \delta_j\}$. Then, if $\tau \in Q_k \setminus D_{\delta_j}$,

$$\begin{aligned}
\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j}); \rho_k > \delta_j] &= \Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j}); \mathcal{E}_j, \tau_k \in Q_k \setminus D_{\delta_j}] \\
&= \frac{\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j}); \mathcal{E}_j]}{\Pr[\tau_k \in Q_k \setminus D_{\delta_j} \mid \mathcal{S}_{B,x}(E_{\delta_j}); \mathcal{E}_j]} \\
&= \frac{\pi(\tau)/\pi(Q_k \setminus D_{\delta_{j-1}})}{\sum_{\tau' \in Q_k \setminus D_{\delta_j}} \pi(\tau')/\pi(Q_k \setminus D_{\delta_{j-1}})}, \quad \text{by (29)} \\
&= \pi(\tau)/\pi(Q_k \setminus D_{\delta_j}).
\end{aligned}$$

Also,

$$\Pr[\tau_k = \tau \mid \mathcal{S}^-_{B,x}(E_{\delta_{j+1}}); \rho_k > \delta_j] = \Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_j}); \rho_k > \delta_j],$$

because the extra information contained in $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$, i.e., the value of δ_{j+1}, the process that executes the step at point δ_{j+1}, and B's method invoked at that step (if any), can be determined from $\mathcal{S}_{B,x}(E_{\delta_j})$ and the random bits of the adversary generated before δ_{j+1}. And these random bits give no additional information about B's state before δ_{j+1}.

Therefore, to complete the induction step it suffices to show that, if $\tau \in Q_k \setminus D_{\delta_j}$,

$$\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_{j+1}}); \mathcal{E}_{j+1}] = \Pr[\tau_k = \tau \mid \mathcal{S}^-_{B,x}(E_{\delta_{j+1}}); \rho_k > \delta_j]. \tag{31}$$

We assume that $\rho_k > \delta_j$, and fix $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. Observe that $\mathcal{E}_{j+1} = \{\rho_k > \delta_j\} \cap \{\tau_k \notin S_{\delta_{j+1}}\}$. Let σ denote the step executed at δ_{j+1}, let p denote the process that executes σ, and let $e_{j+1} = \mathcal{S}_{B,x}(E_{\delta_{j+1}}) \setminus \mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$ be the information about σ contained in $\mathcal{S}_{B,x}(E_{\delta_{j+1}})$. To prove (31), we must argue that conditioning on $\tau_k \notin S_{\delta_{j+1}}$ and on e_{j+1} does not change the distribution of τ_k. I.e., knowing $\tau_k \notin S_{\delta_{j+1}}$ and e_{j+1} does not provide new information about the value of τ_k, directly or indirectly (e.g., by disclosing the task-id of some task $\tau \in Q_k \setminus \{\tau_k\}$). Next we consider all possibilities for step σ.

Case: σ is a step of `finish(a)`. Suppose that σ is a step of command `finish(a)`. If p executes one of lines 31–33 at σ, then e_{j+1} gives no new information about B's state, so assume p executes a different line. We distinguish the cases $a \neq x$ and $a = x$. If $a \neq x$, then the task that p read on $A[a]$ in line 31 was resolved before command `finish(a)` was invoked, and since any task stored on $A[a']$, for $a' \in M_1$, is also resolved, it follows that the command executed at σ (and its outcome) can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$.

Suppose now that the argument of call `finish(a)` is $a = x$. Let τ be the task that p read on $A[x]$ in line 31 of the call. We distinguish the cases $\tau \neq \tau_k$ and $\tau = \tau_k$. Suppose first that $\tau \neq \tau_k$. Then $A[x] = \tau$ at some point before δ_j (precisely, at the point when p read $A[x]$). Also $L_{\delta_j}[x] = \tau_k$, otherwise Lemma 7.40 would imply that $\tau_k \in D_{\delta_j} \cup S_{\delta_j} \cup \{\perp\}$, contradicting the assumption $\rho_k > \delta_j$. Since $\tau \neq \tau_k$, it follows from Lemma 7.41 that $\tau.stat \neq \perp$ at point δ_j. Therefore, task τ is resolved in E_{δ_j}. Then, similarly to the case where $a \neq x$, we have that the command executed at σ (and its outcome) can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$.

Last, suppose that p read task $\tau = \tau_k$ on $A[x]$ in line 31 of call `finish(x)`. From Lemma 7.33 and assumption $\rho_k > \delta_j$, it follows that (i) if p executes line 34 at step σ, the `CAS()` operation fails, (ii) p executes line 36 at σ, the `CAS()` operation is successful, and (iii) p does not execute line 39 at σ. From (i), if p executes line 34, then the state of B does not change, and no new information about it is contained in e_{j+1}. (Note that if the `CAS()` were successful, it would reveal $\tau_k.add_1$!) From (ii), if p executes line 36 at σ, then $A_{\delta_{j+1}}[\tau_k.add_1] = \tau_k$ and thus $\tau_k \in S_{\delta_{j+1}}$. Since $\mathcal{E}_{j+1}$ does not hold in that case, the case is not relevant in the proof of (31). It follows also that p does not execute lines 37 and 38 at σ, as it must have previously executed line 36, implying $\rho_k \leq \delta_j$. Finally, if p executes a command in line 35, then B's state does not change and e_{j+1} does not reveal new information about it.

Case: σ is a step of `read(a)`. Consider first the case in which p does not execute the `CAS()` command of line 44 at σ. Then, clearly B's state does not change and e_{j+1} provides no new information about it. Moreover, if σ is the last step of the `read(a)` command, the return value can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. Suppose now that p executes the `CAS()` command of

line 44 at step σ. This command may change the state of B, but the outcome of the command can already be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. Moreover, since the task τ involved was resolved before δ_{j+1}, it follows that e_{j+1} provides no new information. Case: σ is a step of `BDCAS()`. It remains to consider the case where σ is a step of command `BDCAS()`, excluding the steps of commands `finish()` and `read()`, in lines 20, 21 and 29. If a new task is created at step σ, in line 22, then clearly e_{j+1} provides no new information about τ_k. So, we assume that σ is a step in one of lines 23–28 and 30.

Suppose that the `DCAS()` operation is applied to address $a \in M_0 \setminus \{x\}$. Then $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$ contains the complete values of $L[a]$ and $A[a]$ at point δ_j; in particular, the tasks stored on $L[a]$ and $A[a]$ are resolved before δ_{j+1}. It follows that the command that p executes at σ (and its outcome) can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$, in all cases except for the case where p executes line 23. In that case, the position of the write of the `propose()` command is only revealed in e_{j+1}, but this information is not related to τ_k.

Suppose now that the `DCAS()` operation is applied to address $a = x$. If p executes line 23 at σ, then e_{j+1} provides no new information as the position of the write on array $L[x].C$ is not revealed. If p executes a step of $L[x]$.`choose&lock()` in line 24, then B does not change and e_{j+1} provides again no new information, because from Lemma 7.40 and assumption $\rho_k > \delta_j$, it follows that $L_t[x] = \tau_k \neq \perp$ for all $t \in [t_k, \delta_{j+1}]$. Note also that one can infer from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$ if σ is the last step of call $L[x]$.`choose&lock()`, based on the history of values of $L[x].S$ stored in $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. If p executes line 25 at σ, then clearly e_{j+1} provides no new information. The same is true if p executes line 26, and also the value of the if-condition can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$.

It remains to consider the case where σ is a step in one of lines 27, 28 and 30. Let τ denote the value that p read in line 25. We distinguish the cases $\tau \neq \tau_k$ and $\tau = \tau_k$. First suppose that $\tau \neq \tau_k$. Recall that $L_t[x] = \tau_k \neq \perp$ for all $t \in [t_k, \delta_{j+1}]$, from Lemma 7.40 and assumption $\rho_k > \delta_j$. Assume that p executes a step of command $L[x]$.`unlock(`τ`)`, in line 30. Then e_{j+1} contains no new information, because step σ cannot change the value of $L[x].S$, since $L_{\delta_j}[x] = \tau_k \neq \tau$. Note also that one can infer from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$ whether σ is the last step of call $L[x]$.`unlock(`τ`)`.

Consider now the case where $\tau \neq \tau_k$, and p executes a step in one of lines 27 and 28 at σ. Then $\tau \neq \perp$. If p executes line 27 then e_{j+1} contains no new information. Note that if the last step of p before σ was in line 27, then step σ can be either in line 28 or in line 29, and this information cannot always be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. Since e_{j+1} contains that information, e_{j+1} may reveal new information about the value of $\tau.old_0$. However, this does not reveal any information about τ_k, because task τ must have been written to $L[x].S$ at some point in $(0, t_{k-2}] \cup \{t_{k-1}\}$, thus $\tau \notin Q_k$. Suppose p executes line 28 at σ. If $A_{\delta_j}[x] = \tau_k$ then the `CAS()` operation will fail, because the task that p obtained the last time it executed line 20 was obtained before p read τ in line 25, thus before $t_k \leq \delta_j$. While if $A_{\delta_j}[x] \neq \tau_k$ then the result of the `CAS()` operation does not reveal any information about τ_k.

Last, suppose that p read value $\tau = \tau_k$ in line 25. If p executes line 27 at σ (this is possible since $\tau_k \neq \perp$), then e_{j+1} contains no new information. Moreover, one can infer the result of the second if-condition, as assumption $\rho_k > t_k$ implies $\tau.old_0 = B_{t_k}[x]$ (by the definition of a doomed task-id), and $B_{t_k}[x]$ can be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$. If p executes line 28 at σ, then the outcome can also be inferred from $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$, as $\mathcal{S}^-_{B,x}(E_{\delta_{j+1}})$ contains the task-ids of $A_{\delta_j}[x]$ and of the task parameters of the `CAS()` command. Finally, it is impossible that p executes a step of call $L[x]$.`unlock(`τ_k`)` in line 30 at σ, because if it did, Lemma 7.51 would contradict the assumption that $\rho > \delta_j$.

This completes the proof of (31) and the induction step of the proof of (29), concluding the proof of Claim 7.29. □

For $3 \leq k \leq N$, we define the set $F_k = Q_k \cap (D_{\rho_k} \cup S_{\rho_k})$, and let $f_k = |F_k|$. Next we compute a *lower* bound on f_k in terms of $p_k = |P_k|$, using Claim 7.29 and the simple game of Lemma 7.52.

Claim 7.30. *Let $k \geq 3$, and suppose that $N \geq k$. Then, $\mathbf{E}[f_k \mid \mathcal{S}^-_{B,x}(E_{t_k})] \geq p_k/64$.*

Proof. Fix $\mathcal{S}^-_{B,x}(E_{t_k})$, and suppose that $p_k = \ell$. We assume that $\ell > 0$, otherwise the statement holds trivially. Let $\pi(\cdot)$ denote the distribution of τ_k, as in Claim 7.29. We have

$$\mathbf{E}[f_k] \geq \mathbf{E}[f_k \mid \tau_k \neq \perp] \cdot \Pr[\tau_k \neq \perp] = \mathbf{E}[f_k \mid \tau_k \neq \perp] \cdot (1 - \pi(\perp)). \tag{32}$$

Next we lower bound $\mathbf{E}[f_k \mid \tau_k \neq \perp]$, by expressing f_k as the score of the simple guessing game described in Lemma 7.52.

Suppose that $\tau_k \neq \perp$. Then $\tau_k \in Q_k$, from Claim 7.28. Define δ_j as in Claim 7.29, and define $i^* \geq 0$ by $\rho_k = \delta_{i^*}$. Let $G = Q_k$ and $g^* = \tau_k$. For each $\tau \in Q_k$, let $\lambda(\tau) = \Pr[\tau_k = \tau] = \pi(\tau)/\pi(Q_k)$. For each $0 \leq i \leq i^*$, let $G_i = Q_k \cap (D_{\delta_i} \cup S_{\delta_i})$. We will show that for any $\tau \in Q_k$,

$$\Pr[g^* = \tau \mid G_0, \ldots, G_i; i \leq i^*] = \Pr[g^* = \tau \mid G_0, \ldots, G_{i-1}; i \leq i^*]. \tag{33}$$

This implies that given $G_0, \ldots, G_{i-1}$ and $i \leq i^*$, the set G_i is independent of the task $g^* = \tau_k$. In other words, G_i can be determined without knowing τ_k. Thus, the sequence $G_0, \ldots, G_{i^*}$ corresponds to a valid (randomized) strategy for the guessing game of Lemma 7.52. The lemma then implies

$$\mathbf{E}\left[\left|\bigcup_{0 \leq i \leq i^*} G_i\right|\right] \geq \left(2 \max_{\tau \in Q_k} \lambda(\tau)\right)^{-1} = \left(2 \max_{\tau \in Q_k} \pi(\tau)/\pi(Q_k)\right)^{-1} \geq \pi(Q_k) \cdot \ell/16,$$

where for the last inequality we used the fact that $\pi(\tau) \leq 8/\ell$ for any $\tau \in Q_k$, from Claim 7.28. Since $F_k = Q_k \cap (D_{\rho_k} \cup S_{\rho_k}) = Q_k \cap (D_{\delta_{i^*}} \cup S_{\delta_{i^*}}) = G_{i^*} = \bigcup_{0 \leq i \leq i^*} G_i$, the inequality above yields $\mathbf{E}[f_k] \geq \pi(Q_k) \cdot \ell/16$. Recall that the above analysis was conditional on the assumption that $\tau_k \neq \perp$. By lifting this assumption, and stating it instead as conditioning, the last inequality becomes

$$\mathbf{E}[f_k \mid \tau_k \neq \perp] \geq \pi(Q_k) \cdot \ell/16.$$

Substituting that to (32) gives

$$\mathbf{E}[f_k] \geq (1 - \pi(\perp)) \cdot \pi(Q_k) \cdot \ell/16 = (1 - \pi(\perp))^2 \cdot \ell/16 \geq (1 - 2^{-\ell})^2 \cdot \ell/16 \geq \ell/64,$$

where for the second-last inequality we used $\pi(\perp) \leq 2^{-\ell}$, from Claim 7.28. It remains to show (33).

We will use the following two simple observations. First, we observe that for any $\tau \in Q_k \setminus \{\tau_k\}$, $\tau \notin S_{\rho_k}$. Indeed, if $\tau \in S_{\rho_k}$ for some $\tau \in Q_k \setminus \{\tau_k\}$, then Lemma 7.39 implies that τ was written to $L[x].S$ at some point $t < \rho_k$. Moreover, $t > t_k$ since t_k is the earliest point that any task from Q_k can be written to $L[x].S$ (and $\tau_k \neq \tau$ is written to it at t_k). From Lemma 7.40 and $t > t_k$, it follows $\tau_k \in D_t \cup S_t$, contradicting that $t < \rho_k$.

The second observation is that if $\tau_k \in S_{\rho_k}$, then for any $\tau \in Q_k \setminus \{\tau_k\}$, $\tau \in D_{\rho_k}$. Indeed, suppose that $\tau_k \in S_{\rho_k}$, and note that $\rho_k > t_k$, as $\tau_k \notin S_{t_k}$ by Lemma 7.39. Then, at point ρ_k, a process p changes the value of $A[\tau_k.add_1]$ to τ_k, by executing a successful `CAS()` command in line 36. From that, and the fact that for any $\tau \in Q_k \setminus D_{t_k}$ (including for $\tau = \tau_k$), $\tau.add_0 = x$ and $\tau.old_0 = B_{t_k}[x]$, it follows that for any $\tau \in Q_k \setminus \{\tau_k\}$, $\tau \in D_{\rho_k}$, by the definition of a doomed task.

Recall that $\rho_k = \delta_{i^*}$. From the first observation above it follows $G_j = Q_k \setminus D_j$, for $0 \leq j < i^*$. Moreover, $G_{i^*} = Q_k \setminus D_{i^*}$ if $\tau_k \notin S_{\rho_k}$, while if $\tau_k \in S_{\rho_k}$, then $G_{i^*} = Q_k$ by the second observation.

Let $i \geq 0$, and suppose that $i^* \geq i$, i.e., $\rho_k \geq \delta_i$. Let p denote the process that executes the step at point δ_i. We distinguish two cases. In case (i), we assume that in its last step before δ_i, process p just finished line 35 of call `finish(`x`)`, and that p read $A[x] = \tau_k$ in line 31 of the same call. Case (ii), is the complementary of case (i).

We observe that if case (i) holds, then $\tau_k \in S_{\delta_i}$. The reason is that p executes either line 36 or line 39 at δ_i, since it just completed line 35 in its previous step. But, from Lemma 7.33, if p executed a failed `CAS()` in line 36, or executed line 39, it would imply $\rho_k < \delta_i$. Thus, the only possibility left is that p executes a successful `CAS()` in line 36, and thus $\tau_k \in S_{\delta_i}$.

It is also easy to see that if case (ii) holds, then $\tau_k \notin S_{\delta_i}$.

Suppose now that case (i) holds. As observed above, $\tau_k \in S_{\delta_i}$ in this case, thus $\delta_i = \rho_k$, i.e., $i = i^*$. Then $G_i = Q_k$, as noted earlier, therefore the value of G_i is completely determined in this case (as Q_k was fixed at point δ_0). It follows that for any $\tau \in Q_k$,

$$\Pr[\tau_k = \tau \mid G_0, \ldots, G_i] = \Pr[\tau_k = \tau \mid G_0, \ldots, G_{i-1}],$$

thus (33) holds in this case.

Next, suppose that case (ii) holds. As observed earlier, $\tau_k \notin S_{\delta_i}$ in this case. Note also that from $\mathcal{S}_{B,x}(E_{\delta_i})$, one can infer that case (ii) holds. Then, from Claim 7.29, if $\tau \in Q_k \setminus D_{\delta_{i-1}}$,

$$\Pr[\tau_k = \tau \mid \mathcal{S}_{B,x}(E_{\delta_i})] = \pi(\tau)/\pi(Q_k \setminus D_{i-1}).$$

(Thus, the probability is zero if $\tau \notin Q_k \setminus D_{\delta_{i-1}}$.) Since $G_0, \ldots, G_i$ can be inferred from $\mathcal{S}_{B,x}(E_{\delta_i})$ and $G_{i-1} = Q_k \setminus D_{\delta_{i-1}}$, the above equation implies that if $\tau \in Q_k \setminus D_{\delta_{i-1}} = Q_k \setminus G_{i-1}$,

$$\Pr[\tau_k = \tau \mid G_0, \ldots, G_i] = \pi(\tau)/\pi(Q_k \setminus G_{i-1}).$$

Since the right side is independent of G_i, given G_{i-1}, it follows that $\Pr[\tau_k = \tau \mid G_0, \ldots, G_i] = \Pr[\tau_k = \tau \mid G_0, \ldots, G_{i-1}]$. This completes the proof of (33), and of Claim 7.30. □

We compute now an *upper* bound on $\sum_k f_k$, in terms of the number O_x of `BDCAS()` operations on x invoked in E.

Claim 7.31. *$\sum_{3\le k\le N} f_k \le \beta \cdot O_x$, where β is some constant.*

Proof. Suppose that process p invokes a `BDCAS()` operation on x. Let Γ be the set of tasks that p proposes during that operation, and let Γ_k, for $3 \le k \le n$, be the set of tasks $\gamma \in \Gamma$ proposed in the interval (t_{k-2}, t_k). From Lemma 7.45, it follows that for any $3 \le k \le N$, $|\Gamma_k| \le c_1$, where c_1 is some constant. We note also that if $\Gamma \cap F_k \ne \emptyset$, then the number of tasks $\gamma \in \Gamma$ proposed after point ρ_k is upper bounded by some constant c_2. This is immediate from Lemma 7.44, because if $\tau \in \Gamma \cap F_k$ then $\tau \in D_{\rho_k} \cap S_{\rho_k}$. Last, we note that if $k < N$ then $\rho_k < t_{k+1}$, because if $\rho_k \ge t_{k+1}$ then Lemma 7.40 implies $\tau_k \in D_t \cup S_t \cup \{\perp\}$ for some $t < t_{k+1}$, contradicting $\rho_k \ge t_{k+1}$.

We combine the above to bound $\sum_{3\le k\le N} |\Gamma \cap F_k|$, as follows. Suppose that the sum is not zero, and let $k^* = \min\{k\colon F_k \cap \Gamma \ne \emptyset\}$. Then

$$\sum_{3\le k\le n} |\Gamma \cap F_k| = \sum_{k^*\le k\le n} |\Gamma \cap F_k| \le \sum_{k^*\le k\le n} |\Gamma_k| = \sum_{k^*\le k\le \min\{n,k^*+2\}} |\Gamma_k| + \sum_{k^*+3\le k\le n} |\Gamma_k|.$$

For the last two sums above, we have $\sum_{k^*\le k\le \min\{n,k^*+2\}} |\Gamma_k| \le 3c_1$, because $|\Gamma_k| \le c_1$; and

$$\sum_{k^*+3\le k\le n} |\Gamma_k| \le 2 \cdot \left| \bigcup_{k^*+3\le k\le n} \Gamma_k \right| \le 2c_2,$$

where the last inequality holds because if $\tau \in \bigcup_{k^*+3\le k\le n} \Gamma_k$, then τ is proposed after $t_{k^*+1} > \rho_{k^*}$, thus there are at most c_2 such τ. Substituting these above, yields $\sum_{3\le k\le n} |\Gamma \cap F_k| \le 3c_1 + 2c_2$. This is a bound on the contribution to $\sum_{3\le k\le N} f_k$ of the tasks proposed by a single `BDCAS()` operation on x among the O_x operations in total. It follows that $\sum_{3\le k\le N} f_k \le (3c_1 + 2c_2) \cdot O_x$. □

Putting the Pieces Together. We use Claims 7.30 and 7.31 to derive an upper bound on $\sum_k p_k$ in terms of O_x. From Claim 7.30, we have that $\mathbf{E}[f_k \mid \mathcal{S}^-_{B,x}(E_{t_k}); N \ge k] \ge p_k/64$, for any $k \ge 3$. From that, it follows $\mathbf{E}[f_k \mid N \ge k] \ge \mathbf{E}[p_k \mid N \ge k]/64$, which implies

$$\mathbf{E}[f_k \cdot \mathbb{1}_{N\ge k}] \ge \mathbf{E}[p_k \cdot \mathbb{1}_{N\ge k}]/64.$$

Using Claim 7.31 and the inequality above, we obtain

$$\begin{aligned} \beta \cdot \mathbf{E}[O_x] &\ge \mathbf{E}\left[\sum_{3\le k\le N} f_k\right] = \mathbf{E}\left[\sum_{3\le k<\infty} f_k \cdot \mathbb{1}_{N\ge k}\right] = \sum_{3\le k<\infty} \mathbf{E}\left[f_k \cdot \mathbb{1}_{N\ge k}\right] \\ &\ge \sum_{3\le k<\infty} \mathbf{E}\left[p_k \cdot \mathbb{1}_{N\ge k}\right]/64 = \mathbf{E}\left[\sum_{3\le k<\infty} p_k \cdot \mathbb{1}_{N\ge k}\right]/64 = \mathbf{E}\left[\sum_{3\le k\le N} p_k\right]/64, \end{aligned} \tag{34}$$

where the reordering of summation and expectation is allowed because $\mathbf{E}[O_x]$ is upper bounded by the expected length of execution E which is finite.

Let Y denote the set of all tasks τ with $\tau.add_0 = x$ proposed in E (in line 23). Recall that P_k contains all tasks $\tau \in Y$ proposed in (s_{k-2}, s_{k-1}), except for $A_{t_{k-1}}[x]$, if task $A_{t_{k-1}}[x]$ was proposed in that interval. It follows that $Y = \bigcup_{1 \le j \le 3} Y_j$, where: $Y_1 = \bigcup_{3 \le k \le N} P_k$; Y_2 is the set of $\tau \in Y$ proposed during I, where $I = (0, s_1) \cup (s_{N-1}, \infty)$ if $N \ge 3$, and $I = (0, \infty)$ if $N \le 2$; and $Y_3 = \bigcup_{1 \le k \le N} \{A_{t_k}[x]\}$. Next we bound the sizes of these three sets.

From (34), we have $\mathbf{E}[|Y_1|] \le 64\beta \cdot \mathbf{E}[O_x]$. From Lemma 7.45, a process executes line 23 at most a constant number of times during I, while executing a single `BDCAS()` operation on x. It follows $|Y_2| \le \beta' \cdot O_x$, where β' is a constant. Last, from Lemma 7.47, at most a constant number β'' of task proposed in each `BDCAS()` operation gets written to $A[x]$, thus $|Y_3| \le \beta'' \cdot O_x$. Then,

$$\mathbf{E}[|Y|] \le \mathbf{E}[|Y_1|] + \mathbf{E}[|Y_2|] + \mathbf{E}[|Y_2|] \le (64\beta + \beta' + \beta'') \cdot \mathbf{E}[O_x].$$

To conclude the proof, we observe that if line 23 is executed $y \ge 0$ times during a `BDCAS()` call, then the total number of steps of this call is at most $c_1 + y \cdot c_2 \log n$, where c_1, c_2 are constants. This is because each operation in lines 20–30 involves a constant number of steps, except for operations `choose&lock()` and `unlock()` in lines 24 and 30, respectively, which involve $O(\log n)$ steps. It follows that the total number steps of all `BDCAS()` operations on x is at most $c_1 \cdot O_x + |Y| \cdot c_2 \log n$, and the mean is at most

$$c_1 \cdot \mathbf{E}[O_x] + \mathbf{E}[|Y|] \cdot c_2 \log n \le c_1 \cdot \mathbf{E}[O_x] + (64\beta + \beta' + 1) \cdot \mathbf{E}[O_x] \cdot c_2 \log n.$$

This completes the proof of Theorem 7.27.

7.2.3 Auxiliary Lemmas. Recall that sets D_t and S_t were defined in Section 7.2.1, and denote the set of doomed and successful tasks at point t, respectively.

Claim 7.32. *Suppose process p creates task τ (in line 22) at some point $t_{\tau\text{-new}}$. Let $t_{read} < t_{\tau\text{-new}}$ be the linearization point of p's* `read`(a_1) *operation in line 21 during the same iteration of the while-loop. Further, let $t > t_{\tau\text{-new}}$.*

(a) If the value of $A[\tau.add_1]$ changes during $[t_{read}, t]$ then $\tau \in S_t \cup D_t$.
(b) If there is a task τ^* with $\tau^*.add_1 = \tau.add_1$, such that $\tau^*.stat$ changes from $\perp$ to True *in $[t_{read}, t]$, then $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.*

Proof. Let $\alpha = \tau.add_1$. We assume that

$$A_z[\alpha] \ne \tau \text{ for all } z \le t, \tag{35}$$

because otherwise $\tau \in S_t$, and the claim is true.

Consider the while-loop iteration in which p creates task τ in line 22 (at point $t_{\tau\text{-new}}$). Then p's `read`(α) in line 21 of the same while-loop iteration returns $\tau.old_1$, as otherwise p would return from its BDCAS call in that line. Hence, by Lemma 7.15, $B_{t_{read}}[\alpha] = \tau.old_1$.

Let κ be the task stored in $A[\alpha]$ at point t_{read}. Then by the definition of the interpreted value of $B[\alpha]$,

$$\tau.old_1 = \begin{cases} \kappa.new_1 & \text{if } \kappa.stat = \text{True at point } t_{read}, \text{ and} \\ \kappa.old_1 & \text{otherwise.} \end{cases} \tag{36}$$

First assume that (a) is true. Then at some point in $[t_{read}, t]$ the value of $A[\alpha]$ changes from κ to a different task. Let $\tau' = A_t[\alpha]$. By Claim 7.9 (b), $\kappa \ne \tau'$. Then by Claim 7.9 (a) $\kappa.new_1 \prec_\alpha \tau'.new_1$. Since either $\kappa.old_1 \prec_\alpha \kappa.new_1$ or $\kappa.old_1 = \kappa.new_1$ (the latter is the case if $\kappa = \lambda_\alpha$), we obtain from (36) and transitivity of $\prec_\alpha$ that $\tau.old_1 \prec_\alpha \tau'.new_1$. Hence, by irreflexivity $\tau.old_1 \ne \tau'.new_1$. By (35) and since $A_t[\alpha_1] = \tau'$, it follows that $\tau \in D_t$ by condition (d1).

Now assume that (b) is true but (a) is not true. Since (a) is not true, $A[\alpha] = \kappa$ throughout $[t_{read}, t]$. Consider the point $t^* \in [t_{read}, t]$ at which $\tau^*.stat$ changes from $\perp$ to True. Since $\tau^*.add_1 = \alpha$, it follows from Lemma 7.7 that $A_{t^*}[\alpha] = \tau^*$. In particular, $\tau^* = \kappa$.

Thus, $\kappa.stat$ changes to True after t_{read}. Then by Claim 7.5 $\kappa.stat \neq$ True at point t_{read}, so by (36), $\tau.old_1 = \kappa.old_1$. Moreover, then κ is not an initial task, and so $\kappa.old_1 \neq \kappa.new_1$, and so $\tau.old_1 \neq \kappa.new_1$. Now let $t' = \max\{t^*, t_{\tau\text{-new}}\}$. Then $t' < t$ and $A_{t'}[\alpha] = \kappa$. Hence, $\tau \in D_{t'}$ follows from (35) by condition (d1). □

Lemma 7.33. *Suppose that at point $t_{p@31}$, process p reads $A[\alpha] = \tau$ in line 31 of operation* `finish(`α`)`*, and at some point $t > t_{p@31}$ during the same* `finish()` *call one of the following happens:*

(a) p executes a successful `CAS()` *operation in line 34,*

(b) p executes a failed `CAS()` *operation in line 36, or*

(c) p executes line 39.

Then $\tau \in D_{t'} \cup S_{t'}$ for some $t' < t$.

Proof. If $\tau = \lambda_\alpha$, then $\tau \in S_{t'}$ for all $t' \geq 0$. Hence, assume that $\tau \neq \lambda_\alpha$. By Claim 7.1 (b), task τ is created in line 22 at some point $t_{\tau\text{-new}} < t_A$.

First assume that (a) is true, i.e., at point t process p performs a successful $\tau_1.stat$.`CAS(`$\perp$`, True)` in line 34, where τ_1 is the task that p reads from $A[\tau.add_1]$ in line 33. Then $\tau_1.add_1 = \tau.add_1$, and so by Claim 7.32 (b) $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.

Now assume that (b) is true. Then at point t process p executes a failed $A[\tau.\alpha_1]$.`CAS(`τ_1, τ`)` in line 36, where τ_1 is the task p previously read from $A[\tau.\alpha_1]$ in line 33 at some point $t_{33} < t$. Since the CAS fails, the value of $A[\tau.add_1]$ changes at some point $t' \in [t_{33}, t)$ from τ_1 to a different value. Thus, by Claim 7.32 (a) $\tau \in S_{t'} \cup D_{t'}$.

Finally, assume that (c) is true. Then at point t process p executes a $\tau.stat$.`CAS(`$\perp$`, False)` in line 39. If prior to that p executes a failed CAS in line 36 during the same `finish()` call, then the claim follows from part (b). If p executes a successful CAS in line 36 at point $t_{p@36}$, then as a result of that $A_{t_{p@36}}[\tau.add_1] = \tau$, and so $\tau \in S'_t$ for $t' = t_{p@36} < t$.

Hence, assume that p does not execute line 36. Then p evaluates the if-condition in line 35 to False. Hence, $\tau_1.new_1 \neq \tau.old_1$, where τ_1 is the task stored in $A[\tau.add_1]$ when p reads that register in line 33 at point $t_{p@33}$. Thus, for $t' = t_{p@33} < t$, if $A[\tau.add_1] \neq \tau$ throughout $[0, t']$ then $\tau \in D_{t'}$ by (d1), and otherwise $\tau \in S_{t'}$. □

Lemma 7.34. *If a process p creates task τ during a* `BDCAS(`$\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle$`)` *operation, and the operation returns before point t, then $\tau \in D_t \cup S_t$.*

Proof. Since p's `BDCAS(`$\langle \alpha_0, v_0, v'_0 \rangle, \langle \alpha_1, v_1, v'_1 \rangle$`)` operation returns, there is an index i such that p's last `read(`α_i`)` operation in line 21 returns a value different from v_i. Let t' be the linearization point of that `read(`α_i`)` operation. Then by Lemma 7.15, $B_{t^*}[\alpha_i] \neq v_i$.

Now consider the iteration of the while-loop in which p creates task τ. Then $\tau.old_i = v_i$ and $\tau.add_i = \alpha_i$ due to the arguments used in the `newTask()` call in line 22. Let $t_{read} < t^*$ be the point when p's `read(`α_i`)` in line 21 of that iteration linearizes. Since p's BDCAS call does not return in that while-loop iteration $B_{t_{read}}[\alpha_i] = v_i$. Hence,

$$\tau.old_i = B_{t_{read}}[\tau.add_i] \neq B_{t^*}[\tau.add_i]. \tag{37}$$

First assume that $i = 1$. By (37) and Lemma 7.24 a successful `BDCAS()` operation op that uses an argument triple $\langle \tau.add_1, u_1, u'_1 \rangle$ for some values u_1, u'_1, linearizes at some point $lin(op) \in (t_{read}, t^*]$. By the definition of linearization points and by Claim 7.23, the process that executes op, creates a task κ whose status changes to True at point $lin(op)$. Then $\kappa.add_1 = \tau.add_1$. Since $lin(op) \in (t_{read}, t^*] \subseteq (t_{read}, t]$ it follows from Claim 7.32 (b) that $\tau \in S_t \cup D_t$.

Now assume that $i = 0$. If $A_{t'}[\tau.add_0] \neq \tau$ for all $t' \leq t^*$, then $\tau \in D_{t^*} \subseteq D_t$ by (37) and (d2-a). Hence, assume

$$\exists t' \leq t : A_{t'}[\tau.add_0] = \tau. \tag{38}$$

First consider the case that $A_{t^*}[\tau.add_0] = \tau$. Then by the definition of interpreted value, $B_{t^*}[\tau.add_0] \in \{\tau.old_0, \tau.new_0\}$. Thus, by (37), $B_{t^*}[\tau.add_0] = \tau.new_0$. Then by the definition of interpreted value, $\tau.stat =$ True at point t^*. Since $\tau.stat =$ False when τ is being created, which is before t^*, it follows from Claim 7.32 (b) that $\tau \in S_{t^*} \cup D_{t^*} \subseteq S_t \cup D_t$.

Finally, consider the case that $A_{t^*}[\tau.add_0] \neq \tau$. By (38), before point t^* the value of $A[\tau.add_0]$ changes from τ to a different value, say τ'. This can only happen when some process executes a successful $A[\tau.add_0].\texttt{CAS}(\tau, \tau')$ in line 28. Then that process previously obtained τ from a `finish()` call in line 20, and hence, executed line 39 after reading τ from A in line 31. Then it follows from Lemma 7.33 (c) that $\tau \in S_t \cup D_t$. □

Claim 7.35. *Let $\alpha \in M_0$, and t a point in time such that $\tau = L_t[\alpha_0]$ is a task. Then at any point $t' \geq t$, $B_{t'}[\alpha_0] = \tau.old_0$ or $\tau.old_0 \prec_\alpha B_{t'}[\alpha_0]$.*

Proof. Since $\tau = L_t[\alpha]$ is a task, some process p calls `propose(`τ`)` on $L[\alpha]$ in line 23 prior to t. (According to the `RepeatedChoice` specification, initially $L[\alpha] = \perp$.) Due to the if-statement in line 21 and the arguments of p's `newTask()` call in line 22, p's preceding `read(`α`)` in line 21 returns $\tau.old_0$. Hence, by Lemma 7.15, there is a point $t^* < t$ (namely the linearization point of p's `read(`α`)`), such that $B_{t^*}[\alpha] = \tau.old_0$. Thus, it follows immediately from Corollary 7.19 that $B_{t'}[\alpha_0] = \tau.old_0$ or $\tau.old_0 \prec_\alpha B_{t'}[\alpha_0]$ for any $t' \in [t^*, \infty) \supseteq [t, \infty)$. □

Claim 7.36. *Let τ be a task, $\alpha = \tau.add_0$, and t a point in time such that $\tau.old_0 \prec_\alpha B_t[\alpha]$. Then either $A[\alpha] = \tau$ at point t or $A[\alpha] \neq \tau$ throughout $[t, \infty)$.*

Proof. For the purpose of contradiction, assume that $A_t[\alpha] \neq \tau$ but there is a point $t' > t$ at which the value of $A_t[\alpha]$ changes to τ. Then by Lemma 7.18 (b) $\tau.stat = \perp$ at point t', and thus $B_{t'}[\alpha] = \tau.old_0$. Thus, we have $B_{t'}[\alpha] \prec_\alpha B_t[\alpha]$, which contradicts Corollary 7.19, since $\prec_\alpha$ is transitive and irreflexive. □

Claim 7.37. *Suppose process p calls `BDCAS`$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$, and in one while-loop iteration p first reads τ from $L[\alpha_0]$ in line 25 at point t, and then it evaluates the if-condition in line 27 to* False. *Then $\tau.old_0 \prec B_t[\alpha_0]$.*

Proof. Let τ_0 be the value that p's `finish(`α_0`)` call in line 20 returns at point $t_{p@20}$. Then by Claim 7.17

$$\exists s \in \{\text{False}, \text{True}\} : \tau_0.stat = s \text{ throughout } [t_{p@20}, \infty). \tag{39}$$

Since in line 26 process p also reads τ_0 from $A[\alpha_0]$ at point $t_{p@26}$ (because it proceeds to line 27), it follows from Lemma 7.16 that

$$A[\alpha_0] = \tau_0 \text{ throughout } [t_{p@20}, t_{p@26}]. \tag{40}$$

Thus, by (39) p's `read(`α_0`)` in line 21 returns $\tau.old_0$ if v = False and $\tau.new_0$ if v = True. Since p's `BDCAS`$(\langle\alpha_0, v_0, v_0'\rangle, \langle\alpha_1, v_1, v_1'\rangle)$ call does not return in line 21, the `read(`α_0`)` operation returns v_0. Hence, by (39) and (40)

$$B[\alpha_0] = v_0 \text{ throughout } [t_{p@20}, t_{p@26}]. \tag{41}$$

Now consider the task τ that p reads from $L[\alpha_0]$ in line 25 at point $t_{p@25} \in [t_{p@20}, t_{p@26}]$. Since p evaluates the if-condition in line 27 to False, $\tau.old_0 \neq v_0$. Thus, by (41) and Claim 7.35, $\tau.old_0 \prec B_t[\alpha]$. □

Claim 7.38. *Let $\alpha \in M_0$.*

(a) If at point t the value of $A[\alpha]$ changes from τ_0 to $\tau \neq \tau_0$, then $L_t[\alpha] = \tau$.

(b) Suppose at point t some process executes a successful $L[\alpha]$.`unlock(`τ`)` operation. Then

(b1) either at point t it holds $A[\alpha] = \tau$ and $\tau.stat \neq \perp$, or $A[\alpha] \neq \tau$ throughout the entire execution, and

(b2) if $\tau \neq \perp$ then $\tau \in S_{t'} \cup D_{t'}$ for some $t' < t$.

Proof. Let $T_0 = 0$ be the beginning of the execution E and for each integer $\ell \geq 1$ let T_ℓ be the point of the ℓ-th shared memory operation in E. We will show by induction on ℓ that the claim is true provided $t \in [0, T_\ell)$. This is trivially true for $\ell = 0$.

Hence, assume we have proved that the claim is true for $t \in [0, T_\ell)$. Let $t \in [T_\ell, T_{\ell+1})$. If $t \in (T_\ell, T_{\ell+1})$, then no shared memory operation occurs at point t, and (a) and (b) are trivially true. Hence, assume $t = T_\ell$.

Part (a): Suppose at point $t = T_\ell$ the value of $A[\alpha]$ changes from τ_0 to $\tau \neq \tau_0$. Then some process p executes a successful $A[\alpha].\texttt{CAS}(\tau_0, \tau)$ operation in line 28 at point t. Prior to that, at point $t_{p@25} < t$ process p reads τ from $L[\alpha]$ in line 25. Thus, $L_{t_{p@25}}[\alpha] = \tau$, so there must be a successful `choose&lock()` call that linearizes before $t_{p@25} < t$ and which decides τ. (By the if-condition in line 27, $\tau \neq \perp$, so τ is not the initial value of $L[\alpha]$, which is $\perp$ according to the `RepeatedChoice` specification.)

For the purpose of a contradiction assume $L_t[\alpha] \neq \tau$. Then some successful $\texttt{unlock}(\tau)$ call linearizes at a point $t_u < t = T_\ell$. By the assumption that the claim is true for all $t < T_\ell$, we conclude from statement (b1) that either $A[\alpha] = \tau$ at point t_u or $A[\alpha] \neq \tau$ throughout the entire execution. Since at point $t > t_u$ the value of $A[\alpha]$ changes from τ_0 to τ, the latter cannot be the case, so $A_{t_u}[\alpha] = \tau$. But then $A_{t_u}[\alpha] = A_t[\alpha] = \tau$, and so by Lemma 7.16 $A[\alpha] = \tau$ throughout $[t_u, t)$. This contradicts that at point t the value of $A[\alpha]$ changes from $\tau_0 \neq \tau$ to τ. Thus, (a) is true for $t = T_\ell$ and thus also for $t \in [0, T_{\ell+1})$.

Part (b): Suppose that at point $t = T_\ell$ some process p executes a successful $L[\alpha].\texttt{unlock}(\tau)$ operation. If $\tau = \perp$, then by Claim 7.1 $A[\alpha] \neq \tau$ throughout the entire execution, so (b) is true. Hence, assume $\tau \neq \perp$. Let $t_{p@25}$ be the point when p reads $L[\alpha]$ in line 25 prior to its successful $L[\alpha].\texttt{unlock}(\tau)$. Then $L[\alpha] = \tau$ at that point. Since p's $\texttt{unlock}(\tau)$ at point t is successful, and $L[\alpha]$ is ABA-free,

$$L[\alpha] = \tau \text{ throughout } [t_{p@25}, t]. \tag{42}$$

Now let t_f be the point when p reads $A[\alpha]$ in line 31 during the `finish()` call p executes in line 20, and let $t_{p@26}$ be the point when it reads $A[\alpha]$ again in line 26. Since p evaluates the if-statement in line 26 to True, it reads the same value τ_0 from $A[\alpha]$ at points t_f and $t_{p@26}$. Then by Lemma 7.16, $A[\alpha] = \tau_0$ throughout the entire interval $[t_f, t_{p@26}]$, and in particular

$$A_{t_{p@25}} = \tau_0. \tag{43}$$

Since we assume that the claim is true for all $t < T_\ell$ and already proved that (a) is true for $t = T_\ell$, it follows from part (a) of the claim and (42) that if the value of $A[\alpha]$ changes in the interval $[t_{p@25}, t]$, then it changes to τ. Thus,

$$A[\alpha] \in \{\tau_0, \tau\} \text{ throughout } [t_f, t]. \tag{44}$$

First assume that p evaluates the if-condition in line 27 to True. Then at some point $t_{p@28}$ process p executes $A[\alpha].\texttt{CAS}(\tau_0, \tau)$ in line 28. Hence, by (44) $A[\alpha] = \tau$ throughout $[t_{p@28}, t]$. Moreover, during its `finish`(α) call in line 29 process p reads τ from $A_t[\alpha]$ in line 31, and executes $\tau.stat.\texttt{CAS}(\perp, \texttt{False})$ in line 39. Hence, $\tau.stat \neq \perp$ at point t by Claim 7.5, and since $A[\alpha] = \tau$ at point t (b1) is true. Moreover, (b2) follows from Lemma 7.33 (c).

Now assume that p evaluates the if-condition in line 27 to False. Then by Claim 7.37, $\tau.old_0 \prec B_{t_{p@25}}[\alpha]$. Hence, by (43) and Claim 7.36, $A[\alpha] \neq \tau$ throughout $[t_{p@25}, \infty)$. We will now show that

$$A[\alpha] \neq \tau \text{ throughout } [0, t_{p@25}].$$

Then clearly (b1) is true. Moreover, $p \in D_{t_{p@25}}$ by (d2-a), and so (b2) is true.

For the purpose of contradiction, assume there is a point in $[0, t_{p@25})$ at which $A[\alpha] = \tau$. Since $L_{t_{p@25}}[\alpha] = \tau$, some process proposes τ in line 23, so τ is not the initial task λ_α. Hence, at some point $t' < t_{p@25} < t = T_\ell$ the value of $A[\alpha]$ changes to τ. Then by part (a) of the claim (which we proved true for all $t < T_\ell$), we have $L_{t'}[\alpha] = \tau \neq \perp$. By (42) and since $L[\alpha]$ is ABA-free,

$$L[\alpha] = \tau \text{ throughout } [t', t]. \tag{45}$$

Since $A_{t'}[\alpha] = \tau$, it follows from (44) that at some point $t'' \in [t', t_f) \subseteq [t', t]$ the value of $A[\alpha]$ changes to τ_0 from a different value. By part (a) of the claim $L[\alpha] = \tau_0 \neq \tau$ at point t'', which contradicts (45). □

Lemma 7.39. *Let $\alpha \in [m]$, t a point in time, and $\tau \neq \lambda_\alpha$ a task such that $A_t[\alpha] = \tau$.*

(a) If $\alpha \in M_0$, then there is a point $t' < t$ such that $L_{t'}[\alpha] = \tau$.

(b) If $\alpha \in M_1$, then there is a point $t' < t$ such that $A_{t'}[\tau.add_0] = \tau$.

Proof. Part (a) follows immediately from Claim 7.38 (a) and part (b) from Claim 7.3. □

Lemma 7.40. *Let $\alpha \in M_0$, let τ be a task, and let $t_1 < t_2$ be two points in time such that $L_{t_1}[\alpha] = \tau \neq L_{t_2}[\alpha]$.*

Proof. Since τ is a task, $\tau \neq \perp$. As $L_{t_1}[\alpha] = \tau \neq L_{t_2}[\alpha]$, at some point $t_{p@30} \in [t_1, t_2]$ some process p executes a successful $L[\alpha]$.`unlock(`τ`)` operation in line 30. Hence, by Claim 7.38 (b) there exists $t < t_{p@30} \leq t_2$ such that $\tau \in D_t \cup S_t$. □

Lemma 7.41. *Suppose $\alpha \in M_0$ and $t_1 < t_2$ are two points in time such that $A_{t_1}[\alpha] = \tau$ and $L_{t_2}[\alpha] \neq \tau$. Then $\tau.stat \neq \perp$ at point t_2.*

Proof. By Claim 7.5 it suffices to show that there is a point $t \leq t_2$ at which $\tau.stat \neq \perp$.

If τ is the initial task λ_α, then $\tau.stat$ = True initially, and so the claim is true. Hence, assume that $\tau \neq \lambda_\alpha$. Then at some point no later than $t_1 < t_2$ the value of $A[\alpha]$ changes to τ. By Claim 7.38 (a), $L[\alpha] = \tau$ at that point. Since $L_{t_2}[\alpha] \neq \tau$, there is a successful L_{t_2}.`unlock(`τ`)` call at some point $t \leq t_2$. By Claim 7.38 (b1), $\tau.stat \neq \perp$ at point t. □

Claim 7.42. *Let τ be a task that is created at $t_{\tau\text{-new}}$, and suppose one of the following is true for some point $t \geq t_{\tau\text{-new}}$:*

(a) There is an index $i \in \{0, 1\}$ such that $B_t[\tau.add_i] \neq \tau.old_i$; or

(b) There is a task τ' with $\tau'.add_1 = \tau.add_1$ such that at point t $\tau'.stat$ changes to True.

Then $B_{t'}[\tau.add_i] \neq \tau.old_i$ for all $t' \geq t$.

Proof. Let p be the process creating τ at point $t_{\tau\text{-new}}$, i.e., p's `newTask()` in line 22 returns τ at that point. Then before that, in line 21 process p executes a `read(`$\tau.add_i$`)` operation for each $i \in \{0, 1\}$. Since p does not return from its `BDCAS()` call in this line, that `read(`$\tau.add_i$`)` operation returns $\tau.old_i$. Hence, by Lemma 7.15, the interpreted value of $B[\tau.add_i]$ is $\tau.old_i$ at the linearization point of that `read(`$\tau.add_i$`)`, and thus at some point before $t_{\tau\text{-new}}$.

First assume that (a) is true, i.e., $B_t[\tau.add_i] \neq \tau.old_i$ for some $i \in \{0, 1\}$. Let $\alpha = \tau.add_i$. By Claim 7.13, $\tau.old_i \prec_\alpha B_t[\tau.add_i]$. Now the claim follows from Claim 7.13 by irreflexivity and transitivity of $\prec_\alpha$.

Now assume that (b) is true, i.e., at point t the value of $\tau'.stat$ changes to True for some task τ' with $\tau.add_1 = \tau'.add_1$. Then by Claim 7.20, at point t the interpreted value of $B[\tau'.add_1] = B[\tau.add_1]$ changes. Hence, either $B_t[\tau.add_1] \neq \tau.old_1$, or for a point t^* immediately before t, it holds $B_{t^*}[\tau.add_1] \neq \tau.old_1$. In either case, the claim now follows from part (a). □

Claim 7.43. *Let $t_1 < t_2$. Suppose that process p calls `BDCAS(`$\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle$`)`, and during that call it creates a task τ, such that $\tau \in D_{t_1} \cup S_{t_1}$, and during $[t_1, t_2]$ process p completes a `finish(`α_0`)` call in line 20. Then there is an index $i \in \{0, 1\}$ such if p calls `read(`α_i`)` in line 21 after point t_2, then that `read()` operation returns a value different from v_i.*

Proof. Since τ is not an initial task, process p creates τ at some point $t_{\tau\text{-new}} \leq t_1$, and for each $i \in \{0, 1\}$

$$\tau.old_1 = v_1 \neq v_1' = \tau.new_1. \tag{46}$$

First assume that either $\tau \in S_{t_1}$, or $\tau \in D_{t_1}$ because of property (d1). I.e., there is a task τ' (in case $\tau \in S_{t_1}$ we have $\tau = \tau'$) such that $A_{t_1}[\tau.add_1] = \tau'$, and $\tau'.new_1 \neq \tau.old_1 = v_1$. Consider p's `read(`α_1`)` call after point t_2. At some point $t_{p@41}$ process p reads some task τ' from $A[\alpha_1]$ in line 41, and then at point $t_{p@44}$ it performs $\tau'.stat$.`CAS(`$\perp$, True`)` in line 44. Then by Lemma 7.11 and Claim 7.5 $\tau'.stat$ = True throughout $(t_{p@44}, \infty)$. Hence, p's `read(`α_1`)` operation returns $\tau'.new_1$. Since $\tau'.new_1 \neq v_1$, the claim is true.

Now assume that $\tau \in D_{t_1}$ because of property (d2-a). I.e., $B_{t_1}[\tau.add_0] \neq \tau.old_0$. Since $t_{\tau\text{-new}} \leq t_1$, it follows from Claim 7.42 (a) that $B_{t'}[\tau.add_0] \neq \tau.old_0$ for all $t' > t_1$. Hence, by Lemma 7.15, the `read(`α_0`)` that p calls at point t_2 returns a value different from $\tau.old_0 = v_0$.

Finally, assume that $\tau \in D_{t_1}$ because of property (d2-b). Hence, there is a non-initial task τ' such that $A_{t_1}[\tau'.add_1] = \tau'$ and $\tau'.add_0 = \tau.add_0 = \alpha_0$ and $\tau'.old_0 = \tau.old_0 = v_0$. By Claim 7.3,

$$\text{there is a point } t_0 \le t_1 \text{ at which } A[\tau'.add_1] = A[\alpha_0] = \tau'. \tag{47}$$

By the claim assumptions, p completes a `finish`(α_0) call in $[t_1, t_2]$. Let $t_f \ge t_1$ be the point when the `finish`(α_0) call returns. If the `finish()` call returns $\tau'' \ne \tau'$, then p reads τ'' from $A[\alpha_0]$ in line 31 of that `finish()` call. Hence, by Lemma 7.18 (b) and Claim 7.5 $\tau'.stat \ne \perp$ at point t_f. If, on the other hand, the `finish()` call returns τ', then by Claim 7.17 $\tau'.stat \ne \perp$ at point t_f. In either case, by Lemma 7.11 $\tau'.stat$ = True when p's `finish()` call terminates at point t_f. Since by Claim 7.5 $\tau'.stat$ does not change once its True, and by Lemma 7.7 it changes to True while $A[\alpha_0] = \tau'$, by (47) there is a point $t^* \in [t_0, t_f]$ at which $A[\alpha_0] = \tau'$ and $\tau'.stat$ = True. Hence, $v_0 = \tau'.old_0 \prec \tau'.new_0 = B_{t^*}[\alpha_0]$ (because τ' is a non-initial task). Thus, by Claim 7.13 and transitivity and irreflexivity of $\prec_\alpha$, $B_{t'}[\tau.add_0] \ne v_0$ for all $t' > t^*$. Hence, by Lemma 7.15, the `read`(α_0) that p calls at point t_2 returns a value different from v_0. □

Lemma 7.44. *If process p proposes task τ during a* `BDCAS()` *operation and $\tau \in D_t \cup S_t$, then p executes line 23 at most twice after t and before the* `BDCAS()` *returns.*

Proof. This lemma follows immediately from Claim 7.43. □

Lemma 7.45. *Let $\alpha \in M_0$ and let $t_0 < t_1 < t_2 < t_3$ be four points in time, such that at each point t_i, $i \in \{0, \dots, 3\}$, process p executes an $L[\alpha]$.`choose&lock()` call in line 24. Then some successful $L[\alpha]$.`choose&lock()` call linearizes in $(t_0, t_3]$.*

Proof. Suppose the claim is not true, i.e., no successful $L[\alpha]$.`choose&lock()` call linearizes in $(t_0, t_3]$. Since p calls `choose&lock()` at point t_0, $L[\alpha]$ is locked throughout $(t_0, t_3]$ (otherwise one of p's `choose&lock()` calls at points t_1, t_2, and t_3 would be successful) and thus there is a value τ such that $L[\alpha] = \tau$ throughout $(t_0, t_3]$. By Claim 7.38, if the value of $A[\alpha]$ changes in the interval $(t_1, t_3]$, then it changes from $\tau' \ne \tau$ to τ. In particular, the value of $A[\alpha]$ changes at most once in the interval $(t_0, t_3]$. Process p executes at least two complete iterations of the while-loop in the interval $(t_0, t_3]$, and thus throughout at least one complete iteration $A[\alpha]$ remains unchanged. Hence, let τ_0 be a task such that $A[\alpha] = \tau_0$ throughout one of the two iterations. Then p's `finish()` call in line 20 returns τ_0 because p reads that value from $A[\alpha]$ in line 31. Moreover, p reads τ_0 from $A[\alpha]$ in line 26, and so the if-statement in that line evaluates to True. In line 25 process p reads τ from $L[\alpha]$, and so in line 30 p calls $L[\alpha]$.`unlock`(τ). Clearly, this `unlock()` call is successful, which a contradiction. □

Claim 7.46. *Suppose process p proposes a task τ and $\tau \in S_t \cup D_t$. Further, let R_t be the set of tasks τ' that p proposes before t, such that the value of $A[\tau'.add_0]$ changes to τ' at some point after t. Then $|R_t| \le 3$.*

Proof. For the purpose of a contradiction, assume that $|R_t| > 3$. Hence, there are four tasks $\tau_1, \tau_2, \tau_3, \tau_4 \in R_t$. Let $t \le t_1 < t_2 < t_3 < t_4$ such that for each $i \in \{1, 2, 3, 4\}$ at point t_i the value of $A[\tau.add_0]$ changes to τ_i. By Claim 7.38 (a), for each $i \in \{1, \dots, 4\}$, $L_{t_i}[\alpha] = \tau_i$. Hence, in the interval $[t_1, t_4]$ there are points $t_1' < s_1 < s_2 < t^*$ such that at point t_1' a successful `choose&lock()` call decides τ_2, at points s_1 and s_2 the successful calls `unlock`(τ_2) and `unlock`(τ_3), respectively, take place, and at t^* a successful `choose&lock()` call decides τ_4. Since τ_4 is proposed before t, $L_{t^*} \ne \tau_4$ by Theorem 6.11. This is a contradiction. □

Lemma 7.47. *Let $\alpha \in M_0$ and let R be the set of tasks τ that process p proposes during a* `BDCAS()` *operation, such that $A[\alpha] = \tau$ at some point during the execution. Then $|R| \le 6$.*

Proof. Let t_1 be the earliest point at which $A[\alpha] = \tau_1$ for some task $\tau_1 \in R$, and t_2 be the point when for the first time $A[\alpha]$ changes to a task in $R \setminus \{\tau_1\}$. By Claim 7.6 the value of $\tau_1.stat$ changes to True or False before t_2. If $\tau_1.stat$ changes to True, then by Lemma 7.7 there is a point $t < t_2$ at which $A[\tau_1.add_1] = \tau_1$, and so $\tau_1 \in S_t$. If $\tau_1.stat$ = False at point t_2, then before point t_2 some process q performs a successful $\tau_1.stat$.`CAS`($\perp$, False) call in line 39, and thus by Lemma 7.33 (c), $\tau_1 \in D_t \cup S_t$. Hence, in either case, there is a point $t < t_2$ such that $\tau_1 \in S_t \cup D_t$.

Thus, by Lemma 7.44 process p can perform line 23 at most twice after t, and thus there are at most two tasks that p proposes after point t. Moreover, by Claim 7.46, there are at most three tasks τ that p proposes before t, and for which $A[\alpha]$ changes to τ after point t. Finally, τ_1 is the only task stored in $A[\alpha]$ before point t. Thus, we have $|R| \leq 2 + 3 + 1 = 6$. □

Claim 7.48. *Let $[t_1, t_2]$ be a time interval during which only process p takes steps, and only executes a single* BDCAS$(\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle)$ *call. If p completes k iterations of the while-loop during $[t_1, t_2]$, then in each complete iteration it executes line 30, and the $L[\alpha_0]$.unlock() call in that line is successful. Moreover, p's* choose&lock() *calls in line 24 during iterations $2, \ldots, k$ are successful.*

Proof. Consider any complete iteration of the while-loop during interval $[t_1, t_2]$. Let τ be the task that p's finish() call in line 20 returns. Then p reads τ from $A[\alpha_0]$ in line 31 during the corresponding finish() call. Since $A[\alpha_0]$ does not change until p reads $A[\alpha]$ again in line 26, the if-condition in that line evaluates to True. Hence, in each of the k iterations of the while-loop, process p executes lines 27–30. In particular, when p calls $L[\alpha_0]$.unlock(τ') in line 30, it uses the argument τ' that it read from $L[\alpha_0]$ in line 25. Hence, each such unlock() call succeeds.

Thus, whenever p calls choose&lock() in line 24 during one of iterations $2, \ldots, k$, $L[\alpha_0]$ is unlocked. Hence, each such choose&lock() call is successful. □

Claim 7.49. *Let $[t, t']$ be a time interval during which only process p takes steps, and only executes a single* BDCAS$(\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle)$ *call. If p completes at least 5 iterations of the while-loop during $[t, t']$, then there is a point $t^* \in [t, t']$ such that $L_{t^*}[v_0] = \tau$, where τ is one of the tasks p proposes in line 23 during $[t, t']$.*

Proof. For $i \in \{1, \ldots, 5\}$ let t_i be the point when p starts its i-th iteration and let t_6 be the point when it ends its 5th iteration during $[t, t']$.

By Claim 7.48, during each of the five iterations process p executes a successful $L[\alpha_0]$.unlock() call in line 30. Moreover, during $[t_2, t_6]$ process p executes four successful $L[\alpha_0]$.choose&lock() calls, and between the response of each successful $L[\alpha_0]$.unlock() call and the following successful $L[\alpha_0]$.choose&lock() call, process p executes $L[\alpha_0]$.propose() in line 23. Then by Lemma 6.10, there is a point $t^* \in [t_2, t_6]$ such that $L_{t^*}[\alpha_0] = \tau$, where τ is a task that p proposed in line 23 during $[t_1, t_6] \subseteq [t, t']$. □

Lemma 7.50 (Obstruction-Freedom). *If process p invokes a* BDCAS$(\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle)$ *operation, and at some point during the operation, p starts to run solo, then the operation returns after $O(1)$ iterations of the while-loop.*

Proof. Suppose p completes 10 iterations of the while-loop while running solo. By Claim 7.49, there is a point t during the first 5 iterations and a point t' during the last 5 iterations, such that $L_t[\alpha] = \tau$ and $L_{t'}[\alpha] = \tau'$, where τ and τ' are two distinct tasks proposed by p. Thus, by Lemma 7.40 $\tau \in D_{t'} \cup S_{t'}$. Hence, after point t process p completes at most two more iterations of the while-loop according to Lemma 7.44. □

Lemma 7.51. *Let $a \in M_0$, let τ be a task, and suppose at point t some process is poised to call $L[a]$.unlock(τ) in line 30. Then $\tau \in D_{t'} \cup S_{t'}$ for some $t' < t$.*

Proof. Consider the execution prefix that ends at point t. If $L_t[a] = \tau$ and $L[a]$ is locked at point t, then we can let p run solo until its $L[a]$.unlock(a) completes. Hence, by Claim 7.38 (b2) $\tau \in D_{t'} \cup S_{t'}$. Then this is obviously also true for any other execution that has the same prefix up to point t.

Now suppose that either $L_t[a] \neq \tau$ or $L[a]$ is unlocked. Since p reads τ from $L[a]$ in line 25 prior to t, there is a point $t^* < t$ at which a successful $L[a]$.unlock(τ) call linearizes. For that case we have already proved that there exists $t' < t^* < t$ such that $\tau \in D_{t'} \cup S_{t'}$. □

Lemma 7.52 (Guessing Game). *Consider the following simple game. An element g^* of a finite set G is picked at random from a distribution λ over G, satisfying $\lambda(g) \leq \epsilon < 1$, for any $g \in G$. The player does not know g^* (but knows λ), and tries to*

guess g^*. In each round i of the game, the player proposes a set $G_i \subseteq G$, and the game ends when $g^* \in G_i$. The score of the player is then $|\bigcup_i G_i|$. For any strategy of the player, we have that the expected score is at least $(2\epsilon)^{-1}$.

Proof. Clearly, an optimal strategy proposes just one element per round. Let $g_1, g_2, \ldots, g_{|G|}$ be the order of the proposals in an optimal strategy. The expected score is then

$$\sum_{1 \le i \le |G|} i \cdot \lambda(g_i) \ge \sum_{1 \le i \le 1/\epsilon} i \cdot \epsilon = \frac{\lfloor 1/\epsilon \rfloor (1 + \lfloor 1/\epsilon \rfloor)}{2} \cdot \epsilon \ge \frac{(1/\epsilon)^2}{2} \cdot \epsilon = 1/(2\epsilon),$$

where the first inequality holds because $\lambda(g_i) \le \epsilon$, and the sum is minimized by assigning the max probability of ϵ to each of the first $\lfloor 1/\epsilon \rfloor$ terms, and probability zero to the remaining terms. □

8 DCAS ANALYSIS

8.1 Preliminaries

Recall that B is strongly linearizable. In order to prove correctness (linearizability) it suffices to assume that all operations are atomic. In the randomized analysis, in Section 8.5, we do not make that assumption. Thus, references in the following statements to points in time at which operations on B are executed, will in Section 8.5 refer to the strong linearization points of those operations.

We say that process p *attempts to attach* task τ, when p executes a `BDCAS()` operation in line 59, where the arguments new_0 and new_1 of the operation equal $(\tau, 0)$. If the `BDCAS()` operation is successful we say that τ *gets attached*. We also say that τ gets attached to $B[a]$, for each $a \in \{2\tau.add_0, 2\tau.add_1 + 1\}$.

For any task τ and $i \in \{0, 1\}$, we define

$$x_{\tau,i} = 2\tau.add_i + i \quad \text{and} \quad y_{\tau,i} = 2\tau.add_i + 1 - i.$$

Task τ *succeeds* when some process sets $\tau.stat$ to True, in line 76. Task τ is *finished* when some process has completed a `finish(`τ`)` call.

For $\alpha \in [m]$, we say a process *completes an α-loop*, when it completes the entire for-loop starting in line 53 or line 60 during a `DCAS()` call that uses address α in one of its argument triples.

8.2 Some Observations

Observation 8.1. *For each task τ obtained from a* `newTask()` *operation, the values $x_{\tau,0}$, $x_{\tau,1}$, $y_{\tau,0}$, and $y_{\tau,1}$ are all distinct. The same is true for each initial task.*

Proof. If τ is not an initial task, then $\tau.add_0$ and $\tau.add_1$ are the two addresses used in the two argument triples of one `DCAS()` call. Hence, it follows from the `DCAS()` requirements that these values are distinct. Hence, the values $2\tau.add_j + i$ for $i, j \in \{0, 1\}$ are all distinct. By definition, these are the four values $x_{\tau,0}$, $x_{\tau,1}$, $y_{\tau,0}$, and $y_{\tau,1}$.

If τ is an initial task, the same follows immediately from the fact that by defintion $\tau.add_1 = (\tau.add_0 + 1) \mod m$. □

Observation 8.2. *A successful* `BDCAS()` *call in one of lines 68 and 73 changes the values of two array entries $B[2\alpha]$ and $B[2\alpha + 1]$ for some $\alpha \in [m]$.*

Proof. It follows immediately from the assignment of values to *add* and *add′* in line 65, that the array entries affected by such a `BDCAS()` are $B[2\alpha]$ and $B[2\alpha + 1]$ for some $\alpha \in [m]$. Moreover, from the `BDCAS()` arguments in line 73 it follows that the control bits in both array entries change from 0 to 1. A successful `BDCAS()` in line 68 changes the control bit of one of the two affected array entries, and due to the if-condition in line 67 the task of the other affected array entry. □

Observation 8.3. *Let $a \in [2m]$ and $B[a] = (\tau, c)$ for some $c \in \{0, 1\}$.*

(a) If $c = 0$, then $a \in \{x_{\tau,i} \mid i \in \{0, 1\}\}$, and
(b) if $c = 1$, then $a \in \{x_{\tau,i}, y_{\tau,i} \mid i \in \{0, 1\}\}$.

Proof. Initially, $B[a] = (\zeta_a, 1)$, where $a = \lfloor \zeta_a.add_0/2 \rfloor$. Thus, (a) and (b) are initially true.

Now consider a non-initial state. Part (a) follows immediately from the fact that a pair $(\tau, 0)$ can only be written to $B[a]$ in a successful BDCAS() operation in line 59, and then $a = 2\tau_i + i = x_{\tau,i}$ for some $i \in \{0, 1\}$.

For part (b) let $a = 2\alpha + j$ for some $j \in \{0, 1\}$. Observe that if $(\tau, 1)$ is written to $B[2\alpha + j]$, then this must happen in a successful BDCAS() operation in one of lines 68 and 73, and the BDCAS() only succeeds if immediately before it $B[2\alpha] = (\tau, 0)$ or $B[2\alpha + 1] = (\tau, 0)$. Hence, by (a) $x_{\tau,i} \in \{2\alpha, 2\alpha + 1\}$ for some $i \in \{0, 1\}$. Then one of $x_{\tau,i}$ and $y_{\tau,i}$ equals $2\alpha + j = a$.

□

Observation 8.4. *If task τ gets attached at point t, then for each $i \in \{0, 1\}$*

(a) $B_t[x_{\tau,i}] = (\tau, 0)$ and $B_t[y_{\tau,i}] \notin \{(\tau, 0), (\tau, 1)\}$; and
(b) for any $t' < t$ and any $a \in [2m]$: $B_{t'}[a] \notin \{(\tau, 0), (\tau, 1)\}$.

Proof. From inspecting the arguments of all BDCAS() operations, one can easily see that a task τ can only be written to $B[a]$, if it either already exists in $B[a']$ for some $a' \in [2m]$, or if τ gets attached in a succesful BDCAS() operation in line 59. Since each task can get attached in that line only once, part (b) follows.

To prove (a) first observe that when τ gets attached at point t, the pair $(\tau, 0)$ gets written to $B[x_{\tau,0}]$ and $B[x_{\tau,1}]$. Moreover, from (b) we have that $B[y_{\tau,i}]$ does not contain τ prior to point t for any $i \in \{0, 1\}$, and since by Observation 8.1 $y_{\tau,i} \notin \{x_{\tau,j} \mid j \in \{0, 1\}\}$ it follows that $B_t[y_{\tau,i}] \notin \{(\tau, 0), (\tau, 1)\}$. □

8.2.1 Progression of Values at Neighbouring Addresses. Throughout this section we consider a fixed $\alpha \in [m]$. We consider a points T and let $\beta_i = B_T[2\alpha + i]$ for $i \in \{0, 1\}$. Further, let T' be the first point after T at which $B_{T'}[2\alpha + i^*] \neq \beta_{i^*}$ for some $i^* \in \{0, 1\}$. (We assume that such a point T' exists.) Define $\beta'_i = B_{T'}[2\alpha + i]$ for $i \in \{0, 1\}$.

Claim 8.5. *Let $(\tau_i, c_i) = \beta_i$ for $i \in \{0, 1\}$. Suppose at point T' a* BDCAS() *call is executed in line 68. Then $c_0 + c_1 = 1$, and $\beta'_j \neq \beta_j$ for both $j \in \{0, 1\}$.*

Proof. Let op be the BDCAS() call executed in line 68. Then op is successful because we assumed that at T' either $B[2\alpha]$ changes from β_0 to $\beta'_0 \neq \beta_0$ or $B[2\alpha + 1]$ changes from β_1 to $\beta'_1 \neq \beta'_1$.

The addresses affected by op are some addresses $2x$ and $2x + 1$, as can be seen from line 65. Hence, $x = \alpha$, and so op uses the argument triples $\left\langle 2\alpha + j, \beta_j, \beta'_j \right\rangle$ for $j \in \{0, 1\}$.

As can be seen from the arguments of the BDCAS() call, the control bit of exactly one of β_0 and β_1 is 0, while the control bits of β'_0 and β'_1 are both 1. Hence, $c_0 + c_1 = 1$. By symmetry we may assume w.l.o.g. that $c_0 = 0$.

Then $\beta_0 \neq \beta'_0$ (because the control bit of β'_0 is 1). Moreover, we have $\beta_1 = (\tau_1, 1)$ and $\beta'_1 = (\tau', 1)$, and by the if-condition in line 67 also $\tau_1 \neq \tau'$. □

Claim 8.6. *At T' a successful* BDCAS() *is executed in line ℓ, where $\ell = 59$ if $\beta_0 = \beta'_0$ or $\beta_1 = \beta'_1$, and otherwise $\ell \in \{68, 73\}$.*

Proof. The value of $B[x]$, $x \in \{2\alpha, 2\alpha + 1\}$, can only change as the result of a BDCAS() operation in line $\ell \in \{59, 68, 73\}$.

The BDCAS() call in line 59 affects $B[2a_0]$ and $B[2a_1 + 1]$, where $a_0 \neq a_1$ (because a_0 and a_1 are addresses used in two argument triples of the same DCAS() operation, and thus are required to be distinct). Hence, as a result of such a BDCAS() call only one of $B[2\alpha]$ and $B[2\alpha + 1]$ can change.

The BDCAS() calls in lines 68 and 73 both affect array entries $B[2a]$ and $B[2a + 1]$ for $a \in [m]$. Due to the control bits used in the arguments, it is obvious that the BDCAS() calls in line 73 change both affected array entries. By Claim 8.5 the same is true for the BDCAS() call in line 68. □

Claim 8.7. *Suppose $\beta_0 = \beta_1 = (\tau, 1)$. Then there is a task τ' and an index i such that*

(a) at T' a successful BDCAS() *in line 59 is executed and τ' gets attached to $B[2\alpha + i]$;*
(b) $\beta'_i = (\tau', 0)$ and $\beta'_{1-i} = (\tau, 1)$;

(c) $\tau' \neq \tau$ *and* τ' *is not stored in array* $B[]$ *throughout* $[0, T')$*; and*
(d) τ *is finished at point* t'*.*

Proof. Only the BDCAS() operations in lines 59 and 68 can change $B[x]$, $x \in \{2\alpha, 2\alpha+1\}$, from $(\tau, 1)$ to a different value. The BDCAS() operation in line 68 can only succeed if at exactly one of the control bits stored in $B[2\alpha]$ and $B[2\alpha+1]$ is 0. Since the values of β_0 and β_1 both have a control bit of 1, the BDCAS() executed at point T' is in line 59. From the arguments of the BDCAS() operation in that line, we can see that there is an index $i \in \{0,1\}$ so that it changes the array entry of $B[2\alpha+i]$ to $(\tau', 0)$. Thus, by Observation 8.3 (a), $2\alpha+i \in \{\tau'.add_0, \tau'.add_1 + 1\}$, and so τ' gets attached to $B[2\alpha+i]$. This proves (a).

The BDCAS() operation at point T' uses two argument triples $\langle 2a_j + j, (\gamma_j, 1), (\gamma, 0)\rangle$ for $j \in \{0,1\}$. Then a_0 and a_1 are the addresses of a DCAS() call, and thus $a_0 \neq a_1$. Hence, there is only one index $j \in \{0,1\}$ such that $2a_j + j \in \{2\alpha, 2\alpha+1\}$. Thus, $\alpha = a_j$, and the BDCAS() operation at point T' changes $B[2\alpha+j]$ from $(\tau, 1)$ to $(\gamma, 0)$, while $B[2\alpha+1-j]$ remains unchanged. By part (a), $j = i$. This proves (b) for $\tau' = \gamma$.

As argued above, the successful BDCAS() operation at point T' in line 59 uses the argument triple $\langle 2\alpha+i, (\tau, 1), (\tau', 0)\rangle$. Thus, the calling process must have previously executed finish(τ) in one of lines 55 and 62, and thus τ is finished at point T'. This proves (d).

Moroever, at point T' task τ' gets attached. Hence, part (c) follows immediately from Observation 8.4. □

Claim 8.8. *Suppose* $\{\beta_0, \beta_1\} = \{(\tau_0, 0), (\tau_1, 1)\}$ *for two tasks* τ_0, τ_1*. Then one of the following is true:*

- *at* T' *a successful* BDCAS() *in line 68 is executed and* $\beta'_0 = \beta'_1 = (\tau_0, 1)$*; or*
- *at* T' *a successful* BDCAS() *in line 59 is executed and there is an index* $j \in \{0,1\}$ *such that* $\beta'_j = (\tau', 0)$ *and* $\beta_{1-j} = \beta'_{1-j} = (\tau_0, 0)$ *where* τ' *is a task that is not stored in array* $B[]$ *throughout* $[0, T')$*, and* τ' *gets attached to* $B[2\alpha + j]$*.*

Proof. Since the control bits of β_0 and β_1 differ, At T' a successful BDCAS() is executed in line 59. This cannot be in line 73, because there the affected addresses are $B[2\alpha]$ and $B[2\alpha+1]$, and a successful DCAS would require both control bits of β_0 and β_1 to be 0. Hence, at T' a successful BDCAS() is executed in one of lines 59 and 68. Let p be the process executing that operation.

First assume p executes its successful BDCAS() in line 59. The BDCAS() operation changes the control bits of its affected array positions from 1 to 0. Hence, it only changes the array entry $B[2\alpha+j]$, $j \in \{0,1\}$, which has value $(\tau_1, 1)$ prior to the BDCAS(). The new value of that array entry is then $\beta'_j = (\tau', 0)$ for a task τ' that p created in its preceding newTask() operation in line 58. Thus, the BDCAS() in line 59 attaches τ' to $B[2\alpha+j]$. By Observation 8.4 τ' is not stored in $B[]$ throughout $[0, T')$. Since the array entry with control bit 0 does not change with p's BDCAS() we have $\beta'_{1-j} = \beta_{1-j} = (\tau_0, 0)$.

Now assume p executes its successful BDCAS() in line 68. In that case it follows immediately from the BDCAS() arguments that this operation successfully changes both of $B[2\alpha+j]$, $j \in \{0,1\}$ to $(\tau_0, 1)$. □

Claim 8.9. *Suppose* $\beta_0 = (\tau_0, 0)$ *and* $\beta_1 = (\tau_1, 0)$*. Then at* T' *a successful* BDCAS() *in line 73 is executed and there is an index* $j \in \{0,1\}$ *such that* $\beta'_0 = \beta'_1 = (\tau_j, 1)$*.*

Proof. By Claim 8.6 at T' a successful BDCAS() operation is executed in one of lines 59, 68 and 73. Since the control bits of β_0 and β_1 are both 0, that BDCAS() is not executed in line 59, and by Claim 8.5 also not in line 68.

Hence, the successful BDCAS() is executed in line 73. By Observation 8.2, the BDCAS() affects neighbouring array entries, i.e., $B[2\alpha]$ and $B[2\alpha+1]$ for some $\alpha \in [m]$. From the argument triples used in the BDCAS() calls in line 73 it follows then that $\beta'_0 = \beta'_1 = (\tau_j, 1)$ for some $j \in \{0,1\}$. □

Corollary 8.10. *For any* $\alpha \in [m]$*, if* $B[2\alpha] = (\tau_0, 1)$ *and* $B[2\alpha+1] = (\tau_1, 1)$*, then* $\tau_0 = \tau_1$*.*

Proof. This is true initially, because then $B[2\alpha] = B[2\alpha+1] = (\zeta_\alpha, 1)$. Now suppose that immediately before some point t we have $(\beta_0, \beta_1) = (B[2\alpha], B[2\alpha+1])$, and at point t the value of that pair changes to (β'_0, β'_1). If the control bits of

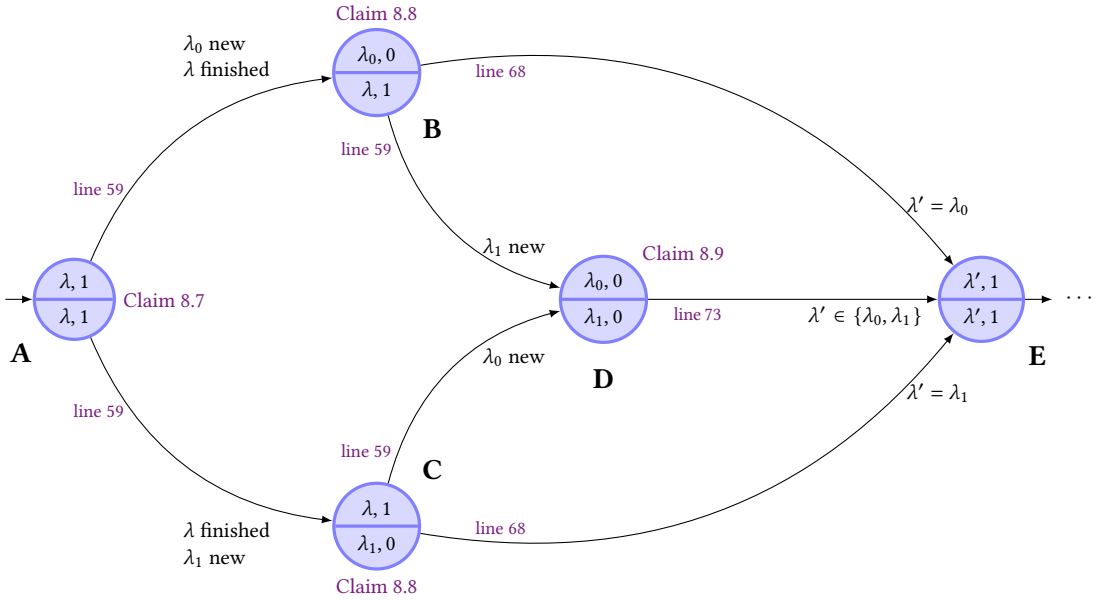

Fig. 6. Each node represents a state of the ordered pair $(B[2\alpha], B[2\alpha+1])$ at a given time, with $B[2\alpha]$ drawn above $B[2\alpha+1]$, where λ, λ_0, and λ_1 are distinct and $\lambda' \in \{\lambda_0, \lambda_1\}$. Each edge represents a possible transition of such a state to the next one, as a result of a successful BDCAS(). The line number in which that BDCAS() can occur is indicated near the start of the edge. Each node is labeled with the claim that proves that each possible transition leaving that node is represented by an edge. When it says that a task is *finished*, then this indicates that the task is finished in the corresponding state. When it says that a task is *new*, then this means that the task was attached in the state transition leading to this state, and thus not stored in $B[\,]$ prior to the state transition. Finally, letters A-E are used in the proofs to refer to the indicated states.

β_0 and β_1 are both 1, then by the assumption that the claim is true before t, we have $\beta_0 = \beta_1$. Hence, by Claim 8.7 the control bit of β_0' or β_1' is 0, so the corollary statement remains trivially true. In all other cases it follows from Claims 8.8 and 8.9 that if the control bits of β_0' and β_1' are both 1, then $\beta_0' = \beta_1'$. □

Thus, $(B[2\alpha], B[2\alpha+1])$ can never be in a state $((\tau_0, 1), (\tau_1, 1))$ for $\tau_0 \neq \tau_1$. Therefore, Claims 8.7–8.9 characterize all state transitions that can happen for the pair of values $(B[2\alpha], B[2\alpha+1])$ for a given $\alpha \in [m]$. These transitions are depicted in Figure 6. It follows from Claims 8.7–8.9 that only the depicted transitions can occur, and all reachable states of the pair $(B[2\alpha], B[2\alpha+1])$ correspond to states depicted. (Note that initially, $B[2\alpha] = B[2\alpha+1] = (\zeta + \alpha, 1)$, so the initial state is the one labeled with A in Figure 6.)

Corollary 8.11. *Figure 6 represents all reachable states and all possible transitions of the pair $(B[2\alpha], B[2\alpha+1])$ for each $\alpha \in [m]$, where tasks λ, λ_0, and λ_1 are distinct and $\lambda' \in \{\lambda_0, \lambda_1\}$.*

Inspecting the reachable states of the pair $(B[2\alpha], B[2\alpha+1])$ and noting that $\{x_{\tau,i}, y_{\tau,i}\} = \{2\alpha, 2\alpha+1\}$ for each attached task τ, we obtain the following corollaries.

Corollary 8.12. *Let τ be a task and $i \in \{0, 1\}$. If $B[x_{\tau,i}] = (\tau, 0)$, then $B[y_{\tau,i}] \notin \{(\tau, 0), (\tau, 1)\}$.*

Corollary 8.13. *Let $a \in [2m]$ and $t_1 < t_2$ be two points in time, and suppose that $B_{t_1}[a] = (\tau, 1) \neq B_{t_2}[a]$. Then τ is finished at point t_2.*

Proof. Let $\alpha = \lfloor a/2 \rfloor$. Then one of $B[2\alpha]$ and $B[2\alpha+1]$ has value $(\tau, 1)$. Hence, the state of pair $(B[2\alpha], B[2\alpha+1])$ at point t_1 corresponds to one of nodes A, B, or C in Figure 6 for $\lambda = \tau$. In case of states B and C, task τ is already finished. Hence, suppose the state of $(B[2\alpha], B[2\alpha+1])$ at point t_1 corresponds to node A, i.e., $B_{t_1}[2\alpha] = B_{t_1}[2\alpha+1] = B_{t_1}[a] = (\tau, 1)$. Since a state transition occurs between t_1 and t_2 it follows from Claim 8.7 that τ finishes before that state transition. □

Corollary 8.14. *Let τ be a task, $i \in \{0,1\}$, and $t_1 < t_2$ be two points in time such that $B_{t_1}[x_{\tau,i}] = (\tau,0) \neq B_{t_2}[x_{\tau,i}]$. Then there is a task τ' and a point $t \in [t_1,t_2]$ such that*

(a) $B_t[x_{\tau,i}] = B_t[y_{\tau,i}] = (\tau',1)$; and

(b) if the control bit of $B_{t_1}[y_{\tau,i}]$ is 1, then either $\tau = \tau'$ or in $(t_1,t_2]$ task τ' gets attached to $B[y_{\tau,i}]$.

Proof. Let $\alpha = \tau.add_i$. Then one of $B[2\alpha]$ and $B[2\alpha{+}1]$ has value $(\tau,0)$. Hence, the state of the pair $(B_{t_1}[2\alpha], B_{t_1}[2\alpha{+}1])$ is represented in Figure 6 by one of the nodes B (for $\tau = \lambda_0$), C (for $\tau = \lambda_1$), or D (for $\tau \in \{\lambda_0,\lambda_1\}$). By the assumption that $B_{t_2}[x_{\tau,i}] \neq (\tau,0)$, sufficiently many state transitions must occur in $[t_1,t_2]$, so that at some point $t \in [t_1,t_2]$ the state depicted by node E is reached, and thus $(\tau,0) \neq B_{t_2}[x_{\tau,i}]$. In particular, $B_t[x_{\tau,i}] = B[y_{\tau,i}] = (\tau',1)$ for some task τ'. This proves (a).

Now assume that the control bit of $B_{t_1}[y_{\tau,i}]$ is 1. Then the state of $(B_{t_1}[2\alpha], B_{t_1}[2\alpha+1])$ corresponds to node B or C in Figure 6. Above we argued that in $[t_1,t_2]$ the state of that pair must transition to E. If the state transition is directly from B or C to E, then $\tau = \tau'$.

Otherwise, the first transition goes from B or C to D, which means that (by Claim 8.8) a task τ^* gets attached to $B[y_{\tau,i}]$, while the value of $B[x_{\tau,i}]$ remains unchanged, i.e., it remains $(\tau,0)$. Thus, from the state transition from D to E it follows that when state E is reached at point t, then $B_t[x_{\tau,i}] = B_t[y_{\tau,i}] = (\tau',1)$ for $\tau' \in \{\tau,\tau^*\}$. Hence, either $\tau' = \tau$ or $\tau' = \tau^*$ gets attached to $B_t[y_{\tau,i}]$. Thus, (b) is true. □

Corollary 8.15. *Let τ be a task, $i \in \{0,1\}$, and $t_1 < t_2$ be two points in time such that $B_{t_1}[x_{\tau,i}] = B_{t_2}[x_{\tau,i}] = (\tau,0)$ and $B_{t_1}[y_{\tau,i}] \neq B_{t_2}[y_{\tau,i}]$. Then there is a task $\tau' \neq \tau$ such that $B_{t_2}[y_{\tau,i}] = (\tau',0)$.*

Proof. Let $\alpha = \tau.add_i$. Since $B_{t_1}[x_{\tau,i}] = B_{t_2}[x_{\tau,i}] = (\tau,0)$, the state of the pairs $(B[2\alpha], B[2\alpha+1])$ at points t_1 and t_2, respectively, is represented in Figure 6 by one of the nodes B (for $\tau = \lambda_0$), C (for $\tau = \lambda_1$), or D (for $\tau \in \{\lambda_0,\lambda_1\}$). Moreover, since $B_{t_1}[y_{\tau,i}] \neq B_{t_2}[y_{\tau,i}]$, a state transition occurs between points t_1 and t_2, so the state at t_2 is represented by node D for a new task τ' and $\{\tau,\tau'\} = \{\lambda_0,\lambda_1\}$. Thus, $B_{t_2}[y_{\tau,i}] = (\tau',0)$ and $\tau \neq \tau'$ according to Corollary 8.11. □

Corollary 8.16. *Let $\alpha \in [m]$, $i \in \{0,1\}$, and t_1 a point in time such that $B_{t_1}[2\alpha] = B_{t_1}[2\alpha+1] = (\tau,1)$. For any point $t_2 > t_1$, if $B_{t_2}[2\alpha+i] \neq (\tau,1)$, then in $[t_1,t_2]$ some task τ' gets attached to $B[2\alpha+i]$.*

Proof. This follows immediately from Claim 8.7, which says that when then the control bits of $B[2\alpha]$ and $B(2\alpha+1)$ are both 1, then one of those array entries can only change if a successful `BDCAS()` is executed in line 59, which attaches τ' to that array entry. □

8.3 There are no ABAs

Lemma 8.17. *Let t, t_1, t_2 be three points in time with $t_1 < t < t_2$, and let $a \in [2m]$. If $B_{t_1}[a] = (\tau,c_1)$ and $B_{t_2}[a] = (\tau,c_2)$ then $B_t[a] \in \{(\tau,c_1),(\tau,c_2)\}$ and $c_1 \leq c_2$.*

Proof. Consider the directed path in the automaton depicted Figure 6 that corresponds to the state transitions of the pair $(B[2\alpha], B[2\alpha+1])$ in the execution, where $a \in \{2\alpha, 2\alpha+1\}$. Note that when for the first time a state is reached that contains a task τ_j in the diagram, $j \in \{0,1\}$, then τ_j was not previously stored in $B[]$. (This is explicitly stated in Claims 8.7 and 8.8.)

Moreover, according to Corollary 8.11 the tasks τ, τ_0, and τ_1 depicted in Figure 6 are distinct and $\tau' \in \{\tau_0,\tau_1\}$. Hence, it is easy to check that for on any directed path that starts with a state where $B[a] = (\tau,c_1)$ and ends with a state where $B[a] = (\tau,c_2)$, all intermittent states satisfy $B[a] \in \{(\tau,c_1),(\tau,c_2)\}$, and $c_1 \leq c_2$. □

This lemma immediately implies that if $B[a] = (\tau,1)$ at some point t, then the value of $B[a]$ can only change to $(\tau',\cdot)$ for $\tau' \neq \tau$, and after it changes, τ can never be in $B[a]$ again.

Corollary 8.18. *If $B_t[a] = (\tau,1)$, and $B_{t'}[a] \neq (\tau,1)$ for some $t' > t$, then $B[a] \notin \{(\tau,0),(\tau,1)\}$ throughout $[t',\infty)$.*

8.3.1 Progress through Finish Calls.

Claim 8.19. *If a process calls* `compete`(τ, i) *at point* t'*, then there is a point* $t < t'$ *at which* τ *gets attached, and in particular* $B_t[x_{\tau,j}] = (\tau, 0)$ *for each* $j \in \{0, 1\}$.

Proof. The `compete()` method is called from within `finish`(τ) that p called in one of lines 55 and 62 prior to t. Hence, p previously read τ from an entry of $B[x]$ in one of lines 55 and 62. By Corollary 8.11 it must have gotten attached prior to t, i.e., $(\tau, 0)$ was written to both array entries $B[x_{\tau,j}]$, $j \in \{0, 1\}$ in line 59. □

Claim 8.20. *Suppose a process* p *attempts to attach task* τ *in line 59 at point* t*. Let* $j \in \{0, 1\}$ *and* $t' \geq t$ *be a point in time at which* $B_{t'}[x_{\gamma,j}] \neq (\tau, \cdot)$ *and* $B_{t'}[y_{\gamma,j}] \neq (\tau, \cdot)$*. Then* $B[x_{\gamma,j}] \neq (\tau, \cdot)$ *and* $B[y_{\gamma,j}] \neq (\tau, \cdot)$ *throughout* $[t', \infty)$.

Proof. It follows from Corollary 8.11 and Figure 6 that if τ is not stored in one of the array entries $B[x_{\gamma,j}]$ and $B[y_{\gamma,j}]$, then it can only get written to one of them as a result of a successful `BDCAS()` in line 59, at which point it gets attached. Since for each task there is only one attempt to attach it, the claim follows. □

Lemma 8.21. *Let* $i \in \{0, 1\}$*. At any point after a* `compete`(τ, i) *call completes,* $B[x_{\tau,i}] \neq (\tau, 0)$.

Proof. Suppose process p calls `compete`(τ, i), and t_2 is some point after that `compete()` call. Then τ gets attached at some point t_1 before that `compete()` call (because p reads τ from $B[]$ before it calls `finish`(τ), and thus before it calls `compete`(τ, i)). By Observation 8.1, $B_{t_1}[x_{\tau,i}] = (\tau, 0)$. For the purpose of a contradiction assume $B_{t_2}[x_{\tau,i}] = (\tau, 0)$, and thus by Lemma 8.17

$$B[x_{\tau,i}] = (\tau, 0) \text{ throughout } (t_1, t_2). \tag{48}$$

Let $t_{66} \in (t_1, t_2)$ be the point when p reads some task τ' from $B[y_{\tau,i}]$ in line 66. Since $B[x_{\tau,i}] = (\tau, 0)$ at point t_{66}, it follows from Corollary 8.12 that $\tau \neq \tau'$. Hence, the if-condition in line 67 evaluates to true, and at some point t_{68} process p executes an operation `BDCAS`$(\langle x_{\tau,i}, (\tau, 0), (\tau, 1)\rangle, \langle y_{\tau,i}, (\tau', 1), (\tau, 1)\rangle)$ in line 68. This `BDCAS()` must fail, because otherwise $B[x_{\tau,i}]$ changes, which would contradict (48). Since $B[x_{\tau,i}] = (\tau, 0)$ at point t_{68} we have $B[y_{\tau,i}] \neq (\tau', 0)$ at that point. Thus, the value of $B[y_{\tau,i}]$ changes in $[t_{66}, t_{68}]$, and so by Corollary 8.15, there is a task τ^* such that $B[y_{\tau,i}] = (\tau^*, 0)$ at point t_{68}. By Observation 8.3 (a), then $y_{\tau,i} = x_{\tau^*,i^*}$ for some $i^* \in \{0, 1\}$. Hence, if $B_{t_2}[y_{\tau,i}] \neq (\tau^*, 0)$, then by Corollary 8.14 (a) the control bits of both, $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ are 1 at some pint in $[t_{68}, t_2]$, which contradicts (48). Thus, assume $B[y_{\tau,i}] = (\tau^*, 0)$ at point t_2, and so by Lemma 8.17

$$B[y_{\tau,i}] = (\tau^*, 0) \text{ throughout } [t_{68}, t_2]. \tag{49}$$

Then in line 69 process p reads τ^* from $B[y_{\tau,i}]$. Moreover, at point t_2 process p calls `BDCAS`$(\langle x_{\tau,i}, (\tau, 0), v\rangle, \langle y_{\tau,i}, (\tau^*, 0), v\rangle)$ in line 73, where either $v = (\tau, 1)$ or $v = (\tau^*, 1)$. From (48) and (49) we conclude that the `BDCAS()` operation at point t_2 succeeds, so $B_{t_2}[x_{\tau,i}] = B_{t_2}[y_{\tau,i}] = v \in \{(\tau, 1), (\tau^*, 1)\}$. This contradicts (48). □

Corollary 8.22. *Let* $\alpha \in [m]$*,* $j \in \{0, 1\}$ *and* $t_1 < t_2$ *be two points in time such that* $B_{t_1}[2\alpha + j] = (\tau, 0)$*, and during* $[t_1, t_2]$ *some process* p *completes a* `finish`(τ) *call. Then there is a point* $t \in (t_1, t_2]$ *and a task* τ' *such that*

(a) $B_t[2\alpha] = B_t[2\alpha + 1] = (\tau', 1)$*; and*

(b) if the control bit of $B_{t_1}[2\alpha + 1 - j]$ *is 1 and* $\tau \neq \tau'$*, then in* $(t_1, t_2]$ *task* τ' *gets attached to* $B[2\alpha + 1 - j]$.

Proof. Since $B_{t_1}[2\alpha + j] = (\tau, 0)$, we obtain from Observation 8.3 (a) that $2\alpha + j = x_{\tau,i}$ and $2\alpha + 1 - j = y_{\tau,i}$ for some $i \in \{0, 1\}$. Since p completes a `finish`(τ) call in $[t_1, t_2]$ it also completes a `compete`(τ, i) call in that interval. Thus, by Lemma 8.21, $B_{t_1}[x_{\tau,i}] = (\tau, 0) \neq B_{t_2}[x_{\tau,i}]$. Hence, the claim follows immediately from Corollary 8.14. □

Lemma 8.23. *Let* $j \in \{0, 1\}$*,* $\alpha \in [m]$*, and define* $a_0 = 2\alpha + j$ *and* $a_1 = 2\alpha + 1 - j$*. Suppose there are points* $t_0 < t_0' \leq t_1 < t_1' \leq t_2$ *and tasks* τ_0, τ_1 *such that for each* $j \in \{0, 1\}$ *it holds* $B_{t_j}[a_j] \in \{(\tau_j, 0), (\tau_j, 1)\}$*, and during* $[t_j, t_j']$ *some processes* p *completes a* `finish`(τ_j) *call. Then either*

(a) $B_{t_2}[a_0] = B_{t_2}[a_1] = (\tau_1, 1)$*; or*

(b) in $[t_0, t_2]$ some task gets attached to $B[a_0]$ or $B[a_1]$.

Proof. We will first argue that one of (a) and (b) follows from

$$\exists \tau' \in \Gamma, t^* \in [t_0, t_1] : B_{t^*}[a_0] = B_{t^*}[a_1] = (\tau', 1). \tag{50}$$

Suppose (50) is true. If neither $B[a_0]$ nor $B[a_1]$ changes in the interval $[t^*, t_2]$, then $B_{t_2}[a_0] = B_{t_2}[a_1] = B_{t^*}[a_0] = B_{t^*}[a_1] = (\tau', 1)$. Since $t_1 \in [t^*, t_2]$, by Corollary 8.18 $(\tau', 1) = B_{t_1}[a_0] = B_{t_1}[a_1] = (\tau_1, 1)$, which means (a) is true. If on the other hand one of $B[a_0]$ or $B[a_1]$ changes in $[t^*, t_2]$, then by Claim 8.7 some task τ' gets attached to the array entry that changes first, and so (b) is true. Hence, one of (a) or (b) follows from (50).

We now consider all possible cases, and each one yields either (a), (b), or (50).

If $B_{t_0}[a_0] = (\tau_0, 0)$, then (50) follows from Corollary 8.22 (a). Hence, assume $B_{t_0}[a_0] = (\tau_0, 1)$. If the control bit of $B[a_1]$ is also 1 at point t_0, then by Corollary 8.10 $B_{t_0}[a_0] = B_{t_0}[a_1] = (\tau_0, 1)$, so (50) is also true.

Thus, assume $B_{t_0}[a_0] = (\tau_0, 1)$ and $B_{t_0}[a_1] = (\tau_0', 0)$. (Then the state of $(B[a_0], B[a_1])$ corresponds to one of nodes B and C in Figure 6.) First, assume the state of $(B[a_0], B[a_1])$ changes in the interval $[t_0, t_1]$, and let t^* be the first point when that happens. According to Claim 8.8, either a task gets attached at point t^* to $B[a_0]$ or $B[a_1]$ (and thus (b) is true), or after the state transition we have $B_{t^*}[a_0] = B_{t^*}[a_1] = (\tau_0, 1)$, and thus (50) is true.

Finally, consider the case that $B[a_0] = (\tau_0, 1)$ and $B[a_1] = (\tau_0', 0)$ throughout $[t_0, t_1]$. In particular, then $B_{t_1}[a_0] = (\tau_0, 1)$ and $B_{t_1}[a_1] = (\tau_0', 0) = (\tau_1, 0)$. Since p completes a `finish(`τ_1`)` call in $[t_1, t_2]$, by Corollary 8.22 (a) there is a point $t' \in [t_1, t_2]$ and a task τ' such that

$$B_{t'}[a_0] = B_{t'}[a_1] = (\tau', 1). \tag{51}$$

(Note that $t' > t_1$, so this does not imply (50).) Since the control bit of $B_{t_1}[a_0]$ is 1, by Corollary 8.22 (b) either $\tau' = \tau_1$, or a task gets attached to $B[a_0]$ in $[t_1, t_2]$. In the latter case we proved (b), so assume $\tau' = \tau_1$. If neither $B[a_0]$ nor $B[a_1]$ changes in $[t', t_2]$, then $B_{t_2}[a_0] = B_{t_2}[a_1] = (\tau_1, 1)$, so (a) is true. Otherwise, if one of $B[a_0]$ and $B[a_1]$ changes during that interval, then it follows from (51) and Corollary 8.16 that some task gets attached to the array entry that changes first in that interval, and so (b) is true. □

8.4 Finished Tasks

Claim 8.24. *Let τ be a task, and suppose process p completes a* `finish(`τ`)` *call before point t. Then $\gamma.stat$ does not change in $[t, \infty)$.*

Proof. Assume the claim is not true, i.e., $\gamma.stat$ changes in $[t, \infty)$. Since $\gamma.stat$ can only change from False to True (in line 76), $\gamma.stat$ = False at point t. Then p does not execute line 76 during its `finish(`τ`)` call, so there is an index i such that $B[x_{\tau,i}] \neq (\tau, 1)$ when p reads that array entry in line 75 at some point $t_1 < t$. Moreover, this read happens after a `compete(`τ, i`)` call, so by Lemma 8.21, $B_{t_1}[x_{\tau,i}] \neq (\tau, 0)$. Thus, we showed that

$$\text{there is a point } t_1 < t \text{ such that } B_{t_1}[x_{\tau,i}] \notin \{(\tau, 0), (\tau, 1)\}. \tag{52}$$

Now observe that there is an earlier point $t_0 < t_1$ at which τ got attached, because process p calls `finish(`τ`)` only after reading τ from an array entry of $B[]$, and by Observation 8.4 (b) this can only happen after τ got attached. Hence, by Observation 8.4 (a) $B_{t_0}[x_{\tau,i}] = (\tau, 0)$. It follows from (52) and Lemma 8.17 that $B_{t^*}[x_{\tau,i}] \notin \{(\tau, 0), (\tau, 1)\}$ for any $t^* > t_1$. Thus, each read of $B[x_{\tau,i}]$ after point t_1 during any `finish(`τ`)` call returns a value different from $(\tau, 1)$, so $\tau.stat$ does not change after point t. □

Claim 8.25. *Let τ be a task, and suppose for each $i \in \{0, 1\}$ there is a point t_i such that $B_{t_i}[x_{\tau,i}] = (\tau, 1)$. If a* `finish(`$\tau$`)` *call completes during the execution, then τ succeeds.*

Proof. W.l.o.g. let t_i be the first point such that $B_{t_i}[x_{\tau,i}] = (\tau, 1)$. Let f_i denote the point when the first compete(τ, i) call completes during the execution (such a point exists because we assume that a finish(τ) call completes). Then by Lemma 8.21, $B_{f_i}[x_{\tau,i}] \neq (\tau, 0)$.

Now let t'_i be the point when τ gets attached. Then $B_{t'_i}[x_{\tau,i}] = (\tau, 0)$ according to Observation 8.4, and by Lemma 8.17 we have $B[x_{\tau,i}] = (\tau, 0)$ throughout $[t'_i, t_i)$. Since τ gets attached before compete(τ, i) can be called, we conclude $t_i \leq f_i$.

Now let t be the first point when some process p completes a finish(τ) call. Thus, $t > f_i$. If for each $i \in \{0, 1\}$ the value of $B[x_{\tau,i}]$ does not change in the interval $[t_i, t)$, then p reads $(\tau, 1)$ from $B[x_{\tau,i}]$ for both $i \in \{0, 1\}$, and thus it changes $\gamma.stat$ to True in line 76.

Hence, assume there is an index $i \in \{0, 1\}$ such that $B[x_{\tau,i}]$ changes in the interval $[t_i, t)$. Then by Corollary 8.13 task τ is finished before point t, which contradicts that t is the first point at which a finish(τ) call completes. □

Lemma 8.26. *Suppose at point t $\tau.stat$ changes. Then $B_t[x_{\tau,0}] = B_t[y_{\tau,0}] = B_t[x_{\tau,1}] = B_t[y_{\tau,1}] = (\tau, 1)$.*

Proof. At point t some process p executes a successful τ.CAS(False, True) in line 76. Due to the if-condition in line 75, for each $i \in \{0, 1\}$ there is a point $t_i < t$ such that $B_{t_i}[x_{\tau,i}] = (\tau, 1)$. It follows from Corollary 8.11 and Figure 6 (because nodes B and C in that figure can only be reached from node A) that there is a point $t'_i < t_i$ such that $B_{t'_i}[x_{\tau,i}] = B_{t'_i}[y_{\tau,i}] = (\tau, 1)$.

Hence, if for each $i \in \{0, 1\}$ none of $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ change in the interval $[t_i, t)$, then the claim is true. So assume that there is an index $i \in \{0, 1\}$ and a point $t' \in [t_i, t)$ at which one of $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ changes. Then by Claim 8.7 task τ is finished at point t', and so by Claim 8.24 $\tau.stat$ cannot change in $[t', \infty)$. This contradicts that $\tau.stat$ changes at point $t > t'$. □

8.4.1 *Interpreted Values and Consistency.*

Definition of Interpreted Value. Consider $a \in [2m]$, and assume that $B[a] = (\tau, c)$ for some task τ and control bit c. By Observation 8.3 (b) there are $i, j \in \{0, 1\}$ such that $a = 2\tau.add_i + j$. Let $a \in [m]$ and $B_t[a] = (\tau, c)$ for some point t. Further, let $i \in \{0, 1\}$ such that $\tau.add_i = a$ (by Observation 8.1 i exists and is unique). We define

$$\delta_t(a) = \begin{cases} \tau.new_i & \text{if } \tau.stat_t = \text{True; and} \\ \tau.old_i & \text{otherwise.} \end{cases}$$

We say that $\delta_t(a)$ denotes the interpreted value of $B[a]$ at point t. We omit the index t and simply write $\delta(a)$ if it is clear from the context to which point t we are referring to.

For $\alpha \in [m]$, the interpreted value of $D[\alpha]$ at point t is $D_t[\alpha] = \delta_t(2\alpha)$.

Suppose op is a read(α) operation by process p, where $\alpha \in [m]$. Let t be the point when p reads τ from $B[2\alpha]$ in line 48, and t' be the point when p reads $\tau.stat$ in line 50 ($t' = \infty$ if p does not execute that line). We define $lin(op)$ as the last point in $[t, t']$ at which $B[2\alpha] \in \{(\tau, 0), (\tau, 1)\}$ (and $lin(op) = \infty$ if no such point exists in the execution, because $t' = \infty$).

Linearizability of Reads.

Claim 8.27. *If a* read(α) *operation completes, then it returns $D_t[\alpha]$, where $t = lin(op)$.*

Proof. Suppose p executes read(α), and at point t_1 it obtains task τ from $B[2\alpha]$ in line 48. Let t_2 be the point when p reads $\tau.stat$ in line 50.

Let $t = lin(op)$. By definition, $B_t[2\alpha]$ contains τ. If $t = t_2$, then it is immediate that p's read(α) method returns $\delta_{t_2}(2\alpha) = D_{t_2}[\alpha]$.

Thus, assume $t < t_2$. Then t is the latest point during $[t_1, t_2]$ at which $B[2\alpha]$ contains τ. By Lemma 8.17, then $B[2\alpha]$ does not contain τ throughout (t, ∞). Hence, by Lemma 8.26, $\gamma.stat$ does not change in $[t, \infty)$. In particular, $\gamma.stat$ has the same value at point t as at point t_2 when p reads it in line 50. It follows that p's read(α) operation returns $\delta_t(2\alpha) = D_t[\alpha]$. □

Claim 8.28. *$\delta_t(2\alpha) = \delta_t(2\alpha + 1)$ for any point t during the execution and for any $\alpha \in [m]$.*

Proof. The lemma is true at the beginning of the execution (i.e., for $t = 0$), because $B_0[2\alpha] = B[2\alpha+1] = (\zeta_\alpha, 1)$.

Hence, assume the lemma is true for each point in $[0, t)$. If at point t no shared memory operation is executed then δ_t does not change. Hence, assume at point t a shared memory operation is executed.

If at point t the value of $\tau.stat$ changes, then it follows immediately from Lemma 8.26 that $\delta_t(2\alpha) = \delta_t(2\alpha+1)$ for all $\alpha \in [m]$. Otherwise, the value of $\delta_t(a)$, $a \in [2m]$, can only be affected if at point t a successful BDCAS() operation is executed, which changes $B[a]$. If such a BDCAS() operation is executed in one of lines 68 and 73, then it changes two entries $B[2\alpha]$ and $B[2\alpha+1]$ to the same values, so $\delta_t(2\alpha) = \delta_t(2\alpha+1)$.

Hence suppose at point t a successful BDCAS($\langle 2\alpha_0, (\tau_0, 1), (\tau, 0)\rangle, \langle 2\alpha_1+1, (\tau_1, 1), (\tau, 0)\rangle$) operation is executed in line 59. In particular, the value of $B[2\alpha_i + i]$ changes from $(\tau_i, 1)$ to $(\tau, 0)$ for both $i \in \{0, 1\}$. Let p be the process executing that BDCAS() operation, and let t' be the point immediately before the BDCAS() operation is executed. Then due to the for-loop that p executes prior to that, and the read in line 56, there are points $t_1 < t_2 < t_3 < t'$ such that

- at t_1 process p reads τ_i from $B[x_{\tau,i}]$;
- at t_2 process p's finish(τ_i) call returns; and
- $\delta_{t_3}(2\alpha_i) = \tau.old_i$. (This follows from Claim 8.27, if we let $t_3 = lin(op)$, where op is the read(α_i) operation that p executes in line 56.)

Since $B_{t_1}[x_{\tau,i}]$ and $B_{t'}[x_{\tau,i}]$ contain τ_i, we conclude from Lemma 8.17 that $B[x_{\tau,i}]$ contains τ_i at any point in $[t_1, t']$, and thus at any point in $[t_1, t)$. By Claim 8.24, $\tau_i.stat$ does not change in $[t_2, t)$. Hence, $\delta(x_{\tau,i})$ does not change in $[t_2, t)$.

Now let $\alpha_i = \lfloor x_{\tau_i}/2 \rfloor$. By the assumption that the claim is true throughout $[0, t)$, we obtain $\delta_{t_3}(2\alpha_i) = \delta_{t_3}(2\alpha_i+1)$. Since $x_{\tau,i} \in \{2\alpha_i, 2\alpha_i+1\}$ we have $\tau.old_i = \delta_{t_3}(2\alpha_i) = \delta_{t_3}(x_{\tau,i})$. Because $\delta(x_{\tau,i})$ does not change in $[t_2, t)$, and t_3 and t' fall into this interval, we have $\tau.old_i = \delta_{t'}(x_{\tau,i})$. Moreover, when τ gets attached at point t, we have $\tau.stat =$ False. Hence, $\delta_t(x_{\tau,i}) = \tau.old_i = \delta_{t'}(x_{\tau,i})$ Thus, the interpreted value of $B[x_{\tau,i}]$ does not change due to the BDCAS() at point t. Obviously, the interpreted value of $B[y_{\tau,i}]$ does also not change, so since the claim was true before the BDCAS() at point t, it is still true immediately after the BDCAS(). □

8.4.2 *Linearizability of DCAS.*

Lemma 8.29.

(a) If at point t the interpreted value of $D[\alpha]$ changes for some $\alpha \in [m]$, then there is a task τ such that $\tau.stat$ changes at point t.

(b) If at point t the status of a task τ changes, then for each $i \in \{0, 1\}$ the interpreted value of $D[\tau.add_i]$ changes from $\tau.old_i$ to $\tau.new_i$.

Proof. Part (b) follows immediately from Lemma 8.26 and the definition of the interpreted value.

To prove (a), suppose at point t the interpreted value of $D[\alpha]$ changes from v to v', but the status field of no task changes.

By definition of the interpreted value and by Claim 8.28, then both, $\delta(2\alpha)$ and $\delta(2\alpha+1)$, change from v to v' at point t. By definition of δ, and since the status field of no task changes, the tasks stored in $B[2\alpha]$ and $B[2\alpha+1]$ must both change at point t. This can only happen as a result of a successful BDCAS() operation. But a BDCAS() in line 59 can affect only one of $B[2\alpha]$ and $B[2\alpha+1]$, and each of the BDCAS() operations in lines 68 and 73 preserves the task stored in at least one of $B[2\alpha]$ and $B[2\alpha+1]$ (and only changes the control bit). This is a contradiction. □

Claim 8.30. *Let $t_1 < t_2$ be two points in time, such that at point t_1 task τ gets attached, and in the interval $[t_1, t_2]$ some process p completes an α-loop for $\alpha \in \{\tau.add_0, \tau.add_1\}$. Then*

(a) for each $i \in \{0, 1\}$ the value of $B[x_{\tau,i}]$ changes in $[t_1, t_2]$; and

(b) $\tau.stat$ does not change in $[t_2, \infty)$.

Proof. Since τ gets attached at point t_1, it follows from Observation 8.4 that

$$B_{t_1}[x_{\tau,i}] = (\tau, 0). \tag{53}$$

During its α-loop process p reads $B[a]$ for each $a \in \{2\alpha, 2\alpha+1\}$ and calls `finish(`τ'`)` for each task τ' it obtains from those reads. In particular, since $\alpha \in \{\tau.add_0, \tau.add_1\}$, there is an index $i \in \{0,1\}$ such that p does that for $a = x_{\tau,i}$. Let $t \in [t_1, t_2]$ be the point when p reads $B[x_{\tau,i}]$.

First assume $B_t[x_{\tau,i}] \notin \{(\tau,0),(\tau,1)\}$. Then (a) is trivially true. Moreover, by Lemma 8.17, $B[x_{\tau,i}] \notin \{(\tau,0),(\tau,1)\}$ throughout $[t, \infty)$. Hence, by Lemma 8.26, $\tau.stat$ does not change in $[t, \infty)$, which proves (b).

Now assume $B_t[x_{\tau,i}] \in \{(\tau,0),(\tau,1)\}$. Since p completes a `finish(`τ`)` call during $[t, t_2]$, (b) follows immediately from Claim 8.24. Moreover, by Corollary 8.22 (a), there is a point in $[t, t_2]$ at which the control bit of $B[x_{\tau,i}]$ is 1. Thus, by (53) the value of $B[x_{\tau,i}]$ changes in the interval $[t_1, t_2]$, which proves (a). □

Corollary 8.31. *Suppose process p creates task τ at point t_1 in line 58, and at point $t_2 > t_1$ process p is poised to execute line 63. Then $\tau.stat$ does not change in $[0, t_1] \cup [t_2, \infty)$.*

Proof. If p does not attach τ before t_2, then τ will never be attached, so by Lemma 8.26 its status cannot change throughout the entire execution. If p attaches τ at point t_1, then for the same reason as above $\tau.stat$ can only change after t_1. Hence, the statement follows immediately from Claim 8.30 (b) and the fact that p completes an α-loop after attaching τ and before t_2. □

Claim 8.32. *Suppose p invokes a* `DCAS()` *operation op at point t_1 and op responds at point t_2. Let G be the set of tasks p creates in* `newTask()` *calls in line 58 during op.*

(a) If op returns True*, then there is exactly one task $\tau \in G$ that succeeds, and that task succeeds in $[t_1, t_2]$; and*

(b) if op returns False*, then for each task $\tau \in G$ it is $\tau.stat =$ False throughout the entire execution.*

Proof. Suppose at least one task in G succeeds. Let τ be the first task that p creates in line 58, and which succeeds. By Corollary 8.31, $\tau.stat =$ True when p executes line 63 in the same iteration of the while-loop in which it creates τ. Hence, p's `DCAS()` call returns True in that line, and p creates no more tasks. This proves (b), and it also shows that no more than one task in G can succeed.

Thus, for (a) it suffices to show that at least one task in G succeeds in $[t_1, t_2]$, provided that op returns True. If op returns True, then this must happen in line 63, where p reads $\tau.stat =$ True for some $\tau \in G$, so clearly there is a task $\tau \in G$ that succeeds. By Corollary 8.31, τ succeeds in $[t_1, t_2]$. □

Let op be a `DCAS(`$\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle$`)` operation executed by process p. If one of p's operations `read(`α_i`)`, $i \in \{0,1\}$, in lines 48 and 56 returns a value different from v_i, then $lin(op)$ is the linearization point of the first such `read()`. Now assume that each `read(`α_i`)` operation, $i \in \{0,1\}$, in one of those lines returns v_i. If during p's `DCAS()` call the status of one of the tasks created by p in line 58 changes to True, then $lin(op)$ is the first point when this happens. Otherwise, define $lin(op) = \infty$.

We say a `DCAS()` operation by process p is *successful* if it returns True, and it is unsuccessful, otherwise.

Theorem 8.33. *The DCAS implementation is linearizable.*

Proof. Consider an execution E on an instance D of the DCAS implementation, and let O be the set of operations op on D, for which $lin(op) < \infty$. Order all operations in O by their linearization points, and let S be the resulting sequential history.

If op is a `DCAS()` operation and $lin(op) = \infty$, then op does not respond during the execution, because it does not return in line 63, and it also never terminates its while-loop. If op is a `read()` operation and $lin(op) = \infty$, then it is immediate from the definition that op does not respond. Hence, O contains all operations that respond. Moreover, it follows immediately from the definition of lin that $lin(op)$ is between the invocation and response of op for each $op \in O$.

Hence, it remains to prove that S is valid. Consider an arbitrary DCAS$(\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle)$ operation in E. It follows from Lemma 8.29 and Claim 8.32 that that DCAS() operation is successful if and only if at its linearization point the interpreted value of $D[\alpha_i]$ changes from v_i to v_i' for $i \in \{0, 1\}$, and that the interpreted value of $D[\alpha_i]$ can only change at the linearization point of such a successful DCAS(). Hence, for each $\alpha \in [m]$, after the k-th operation in S, the value of $D[\alpha]$ is the same as the interpreted value of $D[\alpha]$ after the first k operations in E have linearized. Since each read(α) operation in E return the interpreted value of $D[\alpha]$ at its linearization point (by Claim 8.27), and each unsuccessful DCAS$(\langle \alpha_0, v_0, v_0' \rangle, \langle \alpha_1, v_1, v_1' \rangle)$ operation linearizes at a point when the interpreted value of $D[\alpha_i]$ does not equal v_i for at least one $i \in \{0, 1\}$, the sequential execution S is valid. □

8.5 Step Complexity

We show that the expected number of BDCAS() operations executed during a DCAS() operation is $O(1)$. Combining that with the step complexity bound for the BDCAS implementation in Section 4, gives a logarithmic upper bound on the expected amortized step complexity of DCAS() operations.

Assumptions on the BDCAS Object. We assume that the implementation of the BDCAS object B is strongly linearizable. We also require that the implementation does not read or modify the bits $\tau.b_j$, $j \in \{0, 1\}$, of any task τ passed as an argument to a B.BDCAS() command. We do not assume any bounds on the step complexity of operations on B. To simplify the analysis, we just require that all operations on B are obstruction-free.

In the following, we will say that some operation on B is executed *at point t* to denote that t is the (strong) linearization point of that operation. Similarly, the value of $B[a]$, $a \in [2m]$, *at point t*, denoted $B_t[a]$, refers to the value of $B[a]$ at point t in the linearized execution.

Adversary. We consider an execution E on a DCAS object D scheduled by a *weak adaptive* adversary. The adversary cannot see the internal state of processes, including the outcome of their coin-flips. As a result, the adversary cannot see the random bits $\tau.b_i$, $i \in \{0, 1\}$, of a task τ created in line 58. We assume, however, that the value of $\tau.b_i$ is revealed to the adversary immediately after some process reads $\tau.b_i$ (in line 70). We do not specify what information on the internal state of B the adversary can see, but we assume that for any point $t > 0$, and for each operation B.BDCAS() and B.read() invoked in E_t, the adversary knows at point t whether the operation has been completed, and if completed, the adversary knows also the return value (if any).

Based on that information, after each step the adversary chooses the point in time of the next step and the process to take that step, and if the process has completed its previous method call of D (or has not invoked any method yet), the adversary decides the process's next method call.

Main Theorem. The following theorem states the main result that we prove in this section.

Theorem 8.34. *Let E be a random execution in which an adversary as described above schedules calls to the methods of a DCAS object D. Suppose a process q executes the first step of a D.DCAS() operation at point s. Given the prefix of E in the interval $(0, s)$, the expected number of operations on the BDCAS object B that q invokes during the D.DCAS() call is bounded by a constant.*

The following result is immediate from Theorems 7.27 and 8.34.

Corollary 8.35. *If B is implemented by the algorithm in Figure 2, E is an execution of finite expected length on the resulting DCAS object D scheduled by a weak adversary, and O is the number of D.DCAS() and D.read() operations in E, then $\mathbf{E}[|E|] \le c \log n \cdot \mathbf{E}[O]$, where c is a constant.*

Proof. We use the adversary of Theorem 8.34, and assume it has the same (partial) view of B's state as the adversary in Theorem 7.27. For $1 \le k \le O$, let X_k be the number of B.BDCAS() and B.read() operations executed by the kth operation

on D (D.DCAS() or D.read()) in E. Let also $X = \sum_{1 \le k \le O} X_k$. From Theorem 8.34, we have $\mathbf{E}[X_k \mid O \ge k] \le c_1$, where c_1 is a constant. Then,

$$\mathbf{E}[X] = \mathbf{E}\left[\sum_{1 \le k \le O} X_k\right] = \mathbf{E}\left[\sum_{1 \le k < \infty} X_k \cdot \mathbb{1}_{O \ge k}\right] = \sum_{1 \le k < \infty} \mathbf{E}\left[X_k \cdot \mathbb{1}_{O \ge k}\right]$$
$$= \sum_{1 \le k < \infty} \mathbf{E}\left[X_k \mid O \ge k\right] \cdot \Pr[O \ge k] \le c_1 \cdot \sum_{1 \le k < \infty} \Pr[O \ge k] = c_1 \cdot \mathbf{E}[O].$$

From Theorem 7.27 and the simple observation that each operation B.read() consists of at most a constant number of steps, it follows that the expected total number of steps of B.BDCAS() and B.read() operations in E is at most $c_2 \log n \cdot \mathbf{E}[X]$, where c_2 is a constant. It is also immediate from the algorithm that the remaining steps in E (which are not steps of some B.BDCAS() or B.read() operation) are at most $c_3 \cdot X$, where c_3 is another constant. It follows that the expected total number of steps of E is $\mathbf{E}[|E|] \le c_3 \cdot \mathbf{E}[x] + c_2 \log n \cdot \mathbf{E}[X] \le (c_3 + c_2 \log n) \cdot c_1 \cdot \mathbf{E}[O]$. □

8.5.1 Proof of Theorem 8.34. We assume no DCAS() operation is pending at the end of E, as we can always construct a finite extension of E with that property: For each process p that has a pending operation at the end of E, we let it run solo until the operation returns, which takes a finite number of steps, by Lemma 8.43.

Recall that the adversary cannot see the random bit $\tau.b_i$, $i \in \{0, 1\}$, of a task τ created in line 58, before some process reads $\tau.b_i$, in line 70. Observe also that the bit is not used by the algorithm at any point before that read operation, and that the read operation may be executed only after task τ gets attached (formally, after the strong linearization point of the corresponding B.BDCAS() operation in line 59). We can thus employ the *principle of deferred decisions*, to assume that, for any task γ, its random bits $\gamma.b_i$ are decided *immediately after task τ gets attached*. The fact that B is strongly linearizable guarantees that when the random bits are decided, the adversary cannot change the order of past operations on B anymore (i.e., change their linearization points).

Claim 8.36. *Suppose that task τ gets attached at point t. Given the execution prefix E_t, the probability that τ succeeds in E is at least 1/4.*

Proof. For $i \in \{0, 1\}$, let $(\tau_i, c_i) = B_t[y_{\tau,i}]$, and if $c_i = 1$, let $t_i = \min\{r > t \colon B_r[y_{\tau,i}] \ne B_t[y_{\tau,i}]\}$ and $(\tau_i', c_i') = B_{t_i}[y_{\tau,i}]$. From Lemma 8.40, point t_i always exists, if $c_i = 1$. Let $\mathcal{E}_i$ denote the event

$$\big(\{c_i = 0\} \cap \{\tau.b_i + \tau_i.b_{1-i} \equiv i \pmod 2\}\big)$$
$$\cup \big(\{c_i = 1\} \cap \{\tau.b_i + \tau_i'.b_{1-i} \equiv i \pmod 2\}\big)$$
$$\cup \big(\{c_i = 1\} \cap \{\tau_i' = \tau\}\big).$$

Let $r_i = t$ if $c_i = 0$, and $r_i = t_i$ if $c_i = 1$. We show now that

$$\Pr[\mathcal{E}_i \mid E_{r_i}] \ge 1/2. \tag{54}$$

We consider three cases. Suppose first that $c_i = 0$. Then task τ_i gets attached before t, thus its random bit $\tau_i.b_{1-i}$ is decided before t, while $\tau.b_i$ is decided immediately after t, independently of any previous choices. Therefore, given E_t, $\tau.b_i + \tau_i.b_{1-i}$ is equally likely to be odd or even. Thus, $\Pr[\tau.b_i + \tau_i.b_{1-i} \equiv i \pmod 2 \mid E_t] = 1/2$, which implies (54), as we assumed $c_i = 0$.

Next suppose that $c_i = 1$ and $\tau_i' \ne \tau$. From Lemma 8.40, $c_i' = 0$, thus τ_i' gets attached at point $t_i > t$. In this case, $\tau.b_i$ is decided before t_i, and $\tau_i'.b_{1-i}$ is decided right after t_i, independently of previous choices. Therefore, given E_{t_i}, $\tau.b_i + \tau_i'.b_{1-i}$ is equally likely to be odd or even. Thus, $\Pr[\tau.b_i + \tau_i.b_{1-i} \equiv i \pmod 2 \mid E_t] = 1/2$, which implies (54), as $c_i = 1$.

Last, if $c_i = 1$ and $\tau_i' = \tau$, then $\Pr[\mathcal{E}_i \mid E_{t_i}] = 1$, which implies (54).

Next we show that

$$\Pr[\mathcal{E}_0 \cap \mathcal{E}_1 \mid E_t] \ge 1/4. \tag{55}$$

Suppose first that $r_i < r_{1-i}$. Then

$$\Pr[\mathcal{E}_i \cap \mathcal{E}_{1-i} \mid E_{r_i}] = \Pr[\mathcal{E}_i \mid E_{r_i}] \cdot \Pr[\mathcal{E}_{1-i} \mid E_{r_i}; \mathcal{E}_i] \geq (1/2) \cdot (1/2),$$

because $\Pr[\mathcal{E}_i \mid E_{r_i}] \geq 1/2$ by (54), and

$$\Pr[\mathcal{E}_{1-i} \mid E_{r_i}; \mathcal{E}_i] = \mathbf{E}[\Pr[\mathcal{E}_{1-i} \mid E_{r_{1-i}}] \mid E_{r_i}; \mathcal{E}_i] \geq 1/2,$$

where the first equation holds because $r_{1-i} > r_i$, thus the execution prefix E_{r_i} and the fact whether event $\mathcal{E}_i$ holds can be determined from $E_{r_{1-i}}$; and the second equation above follows from (54).

Next suppose that $r_0 = r_1$. We argue that in this case,

$$\Pr[\mathcal{E}_0 \cap \mathcal{E}_1 \mid E_{r_0}] \geq (1/2) \cdot (1/2). \tag{56}$$

The proof is similar to that of (54). If $r_0 = r_1 = t$, then for each $i \in \{0, 1\}$, $c_i = 0$, and τ_i gets attached before t, while $\tau.b_i$ is decided immediately after t, independently of any previous choices *and also independently of* $\tau.b_{1-i}$, which is chosen concurrently.

If $r_0 = r_1 > t$ then, from Lemma 8.40, $c'_i = 0$ for each $i \in \{0, 1\}$, thus τ'_i gets attached at point $r_i = t_i > t$. Since $r_0 = r_1$ and not two different B.BDCAS() operations linearize at the same point, it follows $\tau'_0 = \tau'_1$. We distinguish the cases $\tau'_0 = \tau'_1 \neq \tau$ and $\tau'_0 = \tau'_1 = \tau$.

If $r_0 = r_1 > t$ and $\tau'_0 = \tau'_1 \neq \tau$, then for each $i \in \{0, 1\}$, $\tau'_i.b_{1-i}$ is decided right after $r_i = t_i$, independently of previous choices, and also of $\tau'_{1-i}.b_i$.

If $r_0 = r_1 > t$ and $\tau'_0 = \tau'_1 = \tau$, then $\Pr[\mathcal{E}_0 \cap \mathcal{E}_1 \mid E_{r_0}] = 1$.

In all cases we obtain similarly to (54) that (56) holds. This completes the proof of (55).

Finally, from Lemma 8.39 and the assumption that all DCAS() operations invoked in E are completed, event $\mathcal{E}_0 \cap \mathcal{E}_1$ implies that task τ succeeds in E. Thus, given E_t, the probability that τ succeeds is at least equal to $\Pr[\mathcal{E}_0 \cap \mathcal{E}_1 \mid E_t] \geq 1/4$. □

As in the statement of Theorem 8.34, we assume that process q invokes the operation DCAS($\langle a_0, old_0, new_0\rangle, \langle a_1, old_1, new_1\rangle$) at point s. Let $N \geq 0$ denote the total number of times that q executes the BDCAS() command in line line 59 during the DCAS() operation, and let s_k, for $1 \leq k \leq N$, denote the (strong) linearization point of the kth BDCAS() operation.

Claim 8.37. *There is a constant $\lambda \in \mathbb{N}$ such that for any $k \geq 1$,*

$$\Pr[N \leq k + \lambda \mid E_{s_k}; N \geq k] \geq 1/4.$$

Proof. Let $k \geq 1$, suppose that $N \geq k$, and fix point s_k and the execution prefix E_{s_k}. Let f_k be the earliest point $t \geq s_k$ such that either (i) q's DCAS() operation returns at point t, or (ii) a task τ gets attached at point t, where $\tau.add_i = a_j$ for some pair $i, j \in \{0, 1\}$. Note that point f_k always exists because of the assumption that all DCAS() operations are completed in E.

Next we argue that

$$f_k \leq s_{k+1}, \text{ if } N > k. \tag{57}$$

Indeed if we assume, for contradiction, that $f_k > s_{k+1}$, then no task τ, with $\tau.add_i = a_j$ for some pair $i, j \in \{0, 1\}$, gets attached in the interval $[s_k, s_{k+1}]$. This implies that both BDCAS() commands that q executes at points s_k and s_{k+1} in line 59 are unsuccessful. Then, from Lemma 8.41, some task τ as above gets attached in (s_k, s_{k+1}), which is a contradiction.

If condition (i) above holds for $t = f_k$, then $N = k$, because if $N > k$ then (57) would imply that s_{k+1} is after the return point of q's DCAS() operation, which is a contradiction. Thus the claim holds in this case.

Suppose now that (i) does not hold, and thus (ii) holds for $t = f_k$. Then, from Claim 8.36, the task τ attached at point f_k succeeds with probability at least $1/4$. Moreover, if τ succeeds then Lemma 8.42 implies that q executes line 59 at

most a constant λ' number of times after f_k before q's DCAS() operation returns. From that and (57), it follows that with probability at least 1/4, $N \geq (k+1) + \lambda'$, for some constant λ'. This completes the proof of Claim 8.37. □

From Claim 8.37, it is immediate that $\mathbf{E}[N \mid E_s] \leq 4(1+\lambda)$. The theorem then follows, because the total number of B.BDCAS() and B.read() operations that q invokes during the DCAS() operation is at most $c_1 \cdot N + c_2$, where c_1, c_2 are constants.

8.5.2 *Auxiliary Lemmas.*

Claim 8.38. *Suppose a task τ gets attached at point t during a* DCAS() *call, and the call returns at a point $t' > t$. Moreover, assume that for each $i \in \{0,1\}$ there is a point t_i such that $B_{t_i}[x_{\tau,i}] = (\tau, 1)$. Then τ succeeds during $[t, t']$.*

Proof. Let p be the process that attaches τ. Assume that τ does not succeed during $[t, t']$. Then by Claim 8.25, no finish(τ) call completes during the execution. Then by Corollary 8.13,

$$B[x_{\tau,i}] = (\tau, 1) \text{ throughout } [t_i, \infty] \text{ for all } i \in \{0,1\}. \tag{58}$$

For the purpose of a contradiction, assume that τ does not succeed during $[t, t']$, so $\tau.stat = \text{False}$ throughout $[t, t']$. Then by (58) $B[\tau.add_i] = \tau.old_i$ throughout $[t, t']$. Hence, p does not break out of the while-loop in line 56, and its DCAS() call does also not return in line 63 during $[t, t']$. This contradicts the assumption that p's DCAS() call returns at point t'. □

Lemma 8.39. *Suppose that task τ gets attached at point t during a* DCAS() *call, and the call returns at point $t' > t$. For each $i \in \{0,1\}$, let $(\tau_i, c_i) = B_t[y_{\tau,i}]$. Also if $c_i = 1$, suppose that point $t_i = \min\{s > t : B_s[y_{\tau,i}] \neq (\tau_i, c_i)\}$ exists and let $(\tau_i', c_i') = B_{t_i}[y_{\tau,i}]$. Suppose for each $i \in \{0,1\}$ at least one of the following conditions is true:*

(a) $c_i = 0$ and $\tau.b_i + \tau_i.b_{1-i} \equiv i \pmod 2$,
(b) $c_i = 1$ and $\tau_i' = \tau$, or
(c) $c_i = 1$ and $\tau.b_i + \tau_i'.b_{1-i} \equiv i \pmod 2$.

Then τ succeeds in $[t, t']$.

Proof. By Observation 8.4 (a), $B_t[x_{\tau,i}] = (\tau, 0)$. We will show in all three cases that

$$\forall i \in \{0,1\}\ \exists\, t_i^* \in [t, t'] : B_{t_i^*}[x_{\tau,i}] = (\tau, 1). \tag{59}$$

Thus, the lemma statement follows from Claim 8.38.

Part (a): Assume (a) is true, i.e., $c_i = 0$ and $(\tau.b_i + \tau_i.b_{1-i}) \bmod 2 = i$ for each $i \in \{0,1\}$. Then $B_t[y_{\tau,i}] = (\tau_i, 0)$ and $B_t[x_{\tau,i}] = (\tau, 0)$. Then $B[x_{\tau,i}]$ and $B[y_{\tau_i,i}]$ changes in the interval $[t, t']$ (if not, then p completes a finish(τ) respectively finish(τ_i) call in line 62, and we obtain a contradiction to Corollary 8.22). According to Claim 8.9 the value of $B[x_{\tau,i}]$ can only change at point $t^* > t$ with a successful BDCAS() in line 73. From the arguments of such a BDCAS() operation we conclude that it can only be successful if it is executed during a compete(τ, i) or compete($\tau_i, 1-i$) call.

Suppose a process executes a successful BDCAS() at point in line 73 of compete(τ, i). Then it must first read τ_i from $B[y_{\tau,i}]$, because otherwise the BDCAS() cannot succeed. Hence, in one of lines 71 and 72 it determines $w = \tau$ because $(\tau.b_i + \tau_i.b_{1-i}) \bmod 2 = i$, and thus if it executes a successful BDCAS() in line 73, the values of $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ both change to $(\tau, 1)$. This proves (59).

Now suppose a process executes a successful BDCAS() in line 73 of compete($\tau_i, 1-i$). Then with a symmetric argument as above, the process reads τ from $B[x_{\tau,i}]$, and determines $w = \tau$ because $(\tau_i.b_{1-i} + \tau.b_i) \bmod 2 \neq 1-i$. Thus its successful BDCAS() in line 73 changes the values of $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ also to $(\tau, 1)$. This proves (59).

Part (b): Now assume $c_i = 1$ and $\tau_i' = \tau$. Thus, $B_t[x_{\tau,i}] = (\tau, 0)$, $B_t[y_{\tau,i}] = (\tau_i, 1)$ and $B_{t_i}[y_{\tau,i}] = (\tau, c_i') \neq (\tau_i, 1)$. From Claim 8.8 it follows that then $B_{t_i}[x_{\tau,i}] = B_{t_i}[y_{\tau,i}] = (\tau, 1)$, and thus we proved (59).

Part (c): Finally, assume $c_i = 1$ and $(\tau.b_i + \tau'_i.b_{1-i}) \bmod 2 = i$. Hence, $B_t[x_{\tau,i}] = (\tau, 0)$ and $B_t[y_{\tau,i}] = (\tau_i, 1)$.

By Claim 8.8, given the situation at point t, two state transitions are possible, and both change $B[y_{\tau,i}]$. Hence, each of them occurs at point t_i. The first possibility is that $B_{t_i}[x_{\tau,i}] = B_{t_i}[y_{\tau,i}] = (\tau, 1)$, which proves (59). The other possibility is that $B_{t_i}[x_{\tau,i}] = (\tau, 0)$ and $B_{t_i}[y_{\tau,i}] = (\tau'_i, 0)$. Thus, at point t_i we are now in an identical situation as in the proof of part (a), and so we obtain (59) again. □

Lemma 8.40. *Suppose that task τ gets attached at point t during a* DCAS() *call, and the call returns at point $t' > t$. Suppose also that $B_t[y_{\tau,i}] = (\tau_i, 1)$, for some $i \in \{0, 1\}$ and task τ_i. Then point $t_i = \min\{s > t : B_s[y_{\tau,i}] \neq B_t[y_{\tau,i}]\}$ exists, $t_i < t'$, and $B_{t_i}[y_{\tau,i}]$ is $(\tau, 1)$ or $(\tau', 0)$, where $\tau' \neq \tau$.*

Proof. Since τ gets attached at point t, we have $B_t[x_{\tau,i}] = (\tau, 0)$ by Observation 8.4. Let t^* be the first point in $[t, t']$ at which one of $B[x_{\tau,i}]$ and $B[y_{\tau,i}]$ changes (by Claim 8.30 t^* exists). Then by Claim 8.8 either $B_{t^*}[x_\tau, i] = B_{t^*}[y_{\tau,i}] = (\tau, 1)$, or $B_{t^*}[x_\tau, i] = (\tau, 0)$ and $B_{t_i}[y_{\tau,i}] = (\tau', 0)$ for a task τ' that is not stored in $B[]$ throughout $[0, t_i)$, and thus $\tau' \notin \{\tau, \tau_1\}$. Hence, $t^* = t_i$, and thus $t_i \in [t, t']$. □

Lemma 8.41. *Suppose a process executes two unsuccessful* BDCAS() *commands in line 59 at points $t_1 < t_2$, during a* DCAS() *operation on addresses $a_0, a_1 \in [m]$. Then some task τ gets attached in the interval (t_1, t_2), where $\tau.add_i = a_j$ for some pair $i, j \in \{0, 1\}$.*

Proof. Let $a_i = \tau.add_i$ for $i \in \{0, 1\}$. Then at point t_2 process p executes a failed BDCAS() call using argument triples $\langle 2a_i + i, (\tau_i, 1), (\tau'_i, 0)\rangle$ for $i \in \{0, 1\}$.

Suppose in interval (t_1, t_2) a task τ' gets attached to $B[2a_i + k]$ for some $i, k \in \{0, 1\}$. Then by Observation 8.3, $\tau'.add_j = a_i$ for some $j \in \{0, 1\}$, so the claim is true.

Hence, assume in the interval (t_1, t_2) no task gets attached to $B[2a_i + k]$ for any $i, k \in \{0, 1\}$. For each $i \in \{0, 1\}$, in the foreach-loop preceding the BDCAS() call, process p first reads a task δ_i from $B[2a + j]$ for some $j \in \{0, 1\}$, then calls finish(δ_i), then reads τ_i from $B[2a + 1 - j]$, and calls finish(τ_i). Hence, by Lemma 8.23 and the assumption that in $(t_1, t_2]$ no task τ' gets attached to $B[2a]$ or to $B[2a + 1]$, we have $B_{t_2}[2a_i] = B_{t_2}[2a_i + 1] = (\tau_i, 1)$. Since this is true for both $i \in \{0, 1\}$, the BDCAS() at point t_2 succeeds, which is a contradiction. □

Lemma 8.42. *Suppose process p attaches task τ at point t, and task τ succeeds. Suppose also that process p' (possibly $p' = p$) calls* DCAS($\langle \alpha_0, v_0, v'_0\rangle, \langle \alpha_1, v_1, v'_1\rangle$)*, where one of α_0 and α_1 is in $\{\tau.add_0, \tau.add_1\}$, and executes line 58 at least once before point t. Then during that* DCAS() *operation process p' executes line 63 at most twice after point t.*

Proof. Let $i \in \{0, 1\}$ such that $\alpha_i \in \{\tau.add_0, \tau.add_1\}$. Since the p' executes line 58 before t, it does not break out of the while-loop in its first iteration. Hence, there is a point $t_{read} < t$ (the linearization point of p's read(α_i) in line 56) such that $D_{t_{read}}[\alpha_i] = v_i$.

Let $t_{p'@63}$ be the point when p' becomes poised to execute line 63 for the second time after t. Then p' performs an α_i-loop in $[t, t_{p'@63}]$. Since $\alpha_i \in \{\tau.add_0, \tau.add_1\}$, by Claim 8.30 (b) $\tau.stat$ does not change after $t_{p'@63}$. Hence, by the assumption that τ succeeds, $\tau.stat$ = True when p' executes line 63 for the second time after t'.

Let t^* be the point when $\tau.stat$ changes to True. By Lemma 8.26, τ is attached at that point, so $t^* \in [t, t_{p'@63}]$. Moreover, by Lemma 8.29, the interpreted value of $D[\tau.add_i]$ changes at that point. Hence, $D[\alpha_i] \neq v_i$ throughout $[t^*, \infty) \supseteq [t_{p'@63}, \infty)$. Thus, p' breaks out of the while-loop when it performs line 56 for the first time after $t_{p'@63}$, and thus before it can execute line 63 for a third time after t. □

Lemma 8.43 (Obstruction-Freedom). *If at any point during a* DCAS() *operation, the process p that executes the operation starts running solo, then p completes the operation after a bounded number of steps.*

Proof. Consider a time-interval $[T, T']$ in which p runs solo during a DCAS($\langle a_0, v_0, v'_0\rangle, \langle a_1, v_1, v'_1\rangle$) operation. We will show that p performs at most three complete iterations of the while-loop. Since all other loops and function calls are wait-free, the lemma statement follows.

For the purpose of a contradiction, assume that p performs at least four complete iterations of the while-loop during $[T, T']$.

Then p performs a BDCAS() in line 59 at least four times during $[T, T']$. Then by Lemma 8.41 at least twice in the interval process p attaches some task. Let t_ℓ, $\ell \in \{1, 2\}$, be the point of p's ℓ-th successful BDCAS() operation, and let τ_ℓ be the task that p attaches at point t_ℓ. Thus,

$$B_{t_\ell}[2a_0] = B_{t_\ell}[2a_1 + 1] = (\tau_\ell, 0). \tag{60}$$

Further, let $t_3 > t_2$ be the point when p becomes poised to execute line 63 in the same iteration of the while-loop in which it attaches τ_2.

In the same while-loop iteration in which p attaches τ_ℓ for $\ell \in \{1, 2\}$, p executes the foreach-loop in lines 60–62, and so

$$\text{completes a } \texttt{finish}(\tau_\ell) \text{ call during } (t_\ell, t_{\ell+1}). \tag{61}$$

Thus, by Corollary 8.22, for each $j \in \{0, 1\}$ there is a point $t_{\ell,j} \in (t_\ell, t_{\ell+1}] \subseteq (t_\ell, t_{\ell+1})$, and a task $\tau_{\ell,j}$ such that

$$B_{t_{\ell,j}}[2a_j] = B_{t_{\ell,j}}[2a_j + 1] = (\tau_{\ell,j}, 1). \tag{62}$$

In particular, by (60) and (62), the control bit of $B[2a_j + j] = B[x_{\tau,j}]$ is 1 at point $t_{1,j}$, 0 at point t_2, and then again 1 at point $t_{2,j}$. Therefore, by Lemma 8.17, for each $j \in \{0, 1\}$ the tasks stored in $B[2a_j + j]$ at points $t_{1,j}$ and $t_{2,j}$ are distinct. In particular, $\tau_{1,j} \neq \tau_{2,j}$. Hence, $\tau_{2,j}$ get attached in $(t_{1,j}, t_{2,j})$. Since only p can attach a task in that interval, and $t_{1,j} < t_2 < t_{2,j}$, it follows that $\tau_{2,j} = \tau_2$ for both $j \in \{0, 1\}$.

Thus, we have $B_{t_{2,0}}[2a_0] = (\tau_2, 1)$ and $B_{t_{2,1}}[2a_1 + 1] = (\tau_2, 1)$. Since by (61) process p completes a $\texttt{finish}(\tau_2)$ call in (t_2, t_3), it follows from Claim 8.38 that τ_2 succeeds before t_3. Hence, p returns in line 63 during the same while-loop iteration, and thus before T'. This is a contradiction. □

9 CONCLUSION

We presented the first randomized DCAS algorithm that achieves amortized sub-linear step complexity in expectation. An interesting open problem is to determine the exact (amortized) randomized step complexity of DCAS objects implemented from CAS and registers (or other common synchronization primitives available in hardware). To the best of our knowledge, lower bounds are not known even for deterministic DCAS algorithms. We conjecture that sub-linear worst-case step complexity cannot be achieved. However, we believe that it may be possible to improve the expected amortized step complexity to sub-logarithmic, by improving the RepeatedChoice algorithm. It would be interesting to determine to what extend randomization can help in stronger adversary models. Another important open problem is to provide support for write and double-compare single-swap operations, and to extend the solution to k-CAS operations for $k > 2$.

ACKNOWLEDGMENTS

We thank the anonymous reviewers, who provided numerous insightful comments and suggestions. Support is gratefully acknowledged from the Agence Nationale de la Recherche (ANR) Project PAMELA (ANR16-CE23-0016-01), the Natural Science and Engineering Research Council of Canada (NSERC) under Discovery Grant RGPIN-2019-04852, and the Canada Research Chairs program.

REFERENCES

[1] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. 1997. Disentangling multi-object operations. In *Proc. 16th PODC*. 111–120. https://doi.org/10.1145/259380.259431

[2] Ole Agesen, David Detlefs, Christine Flood, Alexander Garthwaite, Paul Martin, Mark Moir, Nir Shavit, and Guy Steele. 2002. DCAS-based concurrent deques. *Theory of Comp. Syst.* 35 (06 2002), 349–386. https://doi.org/10.1007/s00224-002-1058-2

[3] Zahra Aghazadeh and Philipp Woelfel. 2016. Upper bounds for boundless tagging with bounded objects. In *Proc. 30th DISC*. 442–457. https://doi.org/10.1007/978-3-662-53426-7_32

[4] James H. Anderson and Mark Moir. 1995. Universal constructions for multi-object operations. In *Proc. 14th PODC*. 184–193. https://doi.org/10.1145/224964.224985

[5] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. 1997. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.* 15, 2 (1997), 134–165. https://doi.org/10.1145/253145.253159

[6] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theor. Comp. Sci.* 34, 2 (2001), 115–144. https://doi.org/10.1007/s00224-001-0004-z

[7] Hagit Attiya and Eyal Dagan. 2001. Improved implementations of binary universal operations. *J. of the ACM* 48, 5 (Sept. 2001), 1013–1037. https://doi.org/10.1145/502102.502105

[8] Hagit Attiya and Eshcar Hillel. 2007. The power of DCAS: Highly-concurrent software transactional memory. In *Proc. 26th PODC*. 342–343. https://doi.org/10.1145/1281100.1281163

[9] Hagit Attiya and Eshcar Hillel. 2011. Highly concurrent multi-word synchronization. *Theor. Comp. Sci.* 412, 12-14 (2011), 1243–1262. https://doi.org/10.1016/j.tcs.2010.12.049

[10] Trevor Brown, Faith Ellen, and Eric Ruppert. 2013. Pragmatic primitives for non-blocking data structures. In *Proc. 32nd PODC*. 13–22. https://doi.org/10.1145/2484239.2484273

[11] Trevor Brown, Faith Ellen, and Eric Ruppert. 2014. A general technique for non-blocking trees. In *Proc. 19th PPOPP*. 329–342. https://doi.org/10.1145/2555243.2555267

[12] Simon Doherty, David Detlefs, Lindsay Groves, Christine H. Flood, Victor Luchangco, Paul Alan Martin, Mark Moir, Nir Shavit, and Guy L. Steele Jr. 2004. DCAS is not a silver bullet for nonblocking algorithm design. In *Proc. 16th SPAA*. 216–224. https://doi.org/10.1145/1007912.1007945

[13] Steven Feldman, Pierre LaBorde, and Damian Dechev. 2015. A wait-free multi-word compare-and-swap operation. *Int. J. of Parallel Progr.* 43, 4 (2015), 572–596. https://doi.org/10.1007/s10766-014-0308-7

[14] Wojciech Golab, Lisa Higham, and Philipp Woelfel. 2011. Linearizable implementations do not suffice for randomized distributed computation. In *Proc. 43rd ACM STOC*. 373–382. https://doi.org/10.1145/1993636.1993687

[15] Michael Greenwald. 2002. Two-handed emulation: How to build non-blocking implementation of complex data-structures using DCAS. In *Proc. 21st PODC*. 260–269. https://doi.org/10.1145/571825.571874

[16] Michael Greenwald and David R. Cheriton. 1996. The synergy between non-blocking synchronization and operating system structure. In *Proc. 2nd OSDI*. 123–136. https://doi.org/10.1145/238721.238767

[17] Rachid Guerraoui, Alex Kogan, Virendra J. Marathe, and Igor Zablotchi. 2020. Efficient multi-word compare and swap. In *Proc. 34th DISC*. 4:1–4:19. https://doi.org/10.4230/LIPIcs.DISC.2020.4

[18] Phuong Hoai Ha and Philippas Tsigas. 2004. Reactive multi-word synchronization for multiprocessors. *J. Instr. Level Parallelism* 6 (2004). http://www.jilp.org/vol6/v6paper3.pdf

[19] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. 2002. A practical multi-word compare-and-swap operation. In *Proc. 16th DISC*. 265–279. https://doi.org/10.1007/3-540-36108-1_18

[20] Maurice Herlihy and Eliot Moss. 1993. Transactional memory: Architectural support for lock-free data structures. In *Proc. 20th ISCA*. 289–300. https://doi.org/10.1145/165123.165164

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

[22] Amos Israeli and Lihu Rappoport. 1994. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. 13th PODC*. 151–160. https://doi.org/10.1145/197917.198079

[23] Yujie Liu and Michael F. Spear. 2012. A lock-free, array-based priority queue. In *Proc. 17th PPOPP*. 323–324. https://doi.org/10.1145/2145816.2145876

[24] Victor Luchangco, Mark Moir, and Nir Shavit. 2003. Nonblocking k-compare-single-swap. In *Proc. 25th SPAA*. 314–323. https://doi.org/10.1145/777412.777468

[25] Henry Massalin and Calton Pu. 1991. *A lock-free multiprocessor OS kernel.* Technical Report CUCS–005–9. Columbia University, New York, NY.

[26] Mark Moir. 1997. Transparent support for wait-free transactions. In *Proc. 11th WDAG*. 305–319. https://doi.org/10.1007/BFb0030692

[27] Håkan Sundell. 2011. Wait-free multi-word compare-and-swap using greedy helping and grabbing. *Int. J. of Parallel Progr.* 39, 6 (2011), 694–716. https://doi.org/10.1007/s10766-011-0167-4

[28] Wikipedia contributors. 2018. Double compare-and-swap — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Double_compare-and-swap&oldid=852157310 [Online; accessed 7-November-2020].