Lock-free Serializable Transactions

Jeff Napper jmn@cs.utexas.edu
Lorenzo Alvisi lorenzo@cs.utexas.edu

Laboratory for Advanced Systems Research
Department of Computer Science
The University of Texas at Austin

Abstract

Software transactional memory (STM) provides access to shared data with transactional properties. Existing STM use linearizability as their correctness criterion, although serializability allows more freedom in reordering the operations of committable transactions. Serializable transactions thus provide for more concurrency than linearizable transactions. Specifically, serializability allows read operations to increase concurrency with multiple versions of data. We present the first serializable, lock-free software transactional memory. Our STM supports dynamic transactions, provides early failure indication, requires a single interface for both read-only and read-write transactions, keeps multiple versions of objects, and allows for disjoint-access parallelism—while our implementation relies only on existing lock-free data structures. We prove the lock-free progress guarantee of our STM, and that the set of committed transactions is one-copy serializable (serializable while appearing to have only one copy of the data).

1 Introduction

Transactions are the most widely used abstraction for supporting robust computation over multiple, concurrent shared objects. The atomicity property provides the illusion that all operations within a transaction occur simultaneously. Transactions either *commit* or *abort* so that all their corresponding operations are, respectively, visible or not to other transactions. The durability property prevents committed transactions from becoming aborted (and vice-versa). Consistency among concurrent transactions is defined by the correctness criterion of the transactional system—serializability [16] is one typical criterion. In transactional database systems, serializability is guaranteed by mutual-exclusion locks. Locks have many notorious disadvantages; their use is prone to deadlock and suffers from priority inversion. Further, the concurrency available using locks is directly proportional to the granularity of the locks—finer granularity yields more concurrency, but at the cost of higher complexity. Finally, most locking systems cannot tolerate the failure of a thread holding a lock.

Nonblocking data structures have been proposed to provide robust management of concurrency without relying on locks. *Wait-free* data structures [7] guarantee progress without requiring any bounds on the relative speed of threads, even allowing threads to halt. Wait-freedom has been difficult to implement efficiently in practice, giving rise to weaker notions of robustness. A *lock-free* data structure guarantees only the system as a whole makes progress, allowing starvation but not livelock [7]; *obstruction-free* structures guarantee progress only in the absence of contention, allowing both starvation and livelock [8].

Software transactional memory (STM) [18] combines the benefits of a transactional interface with the robustness of nonblocking data structures.[1] All current STM implementations [5, 9, 3] provide linearizability [10] as the correctness criterion. Linearizability requires that for any two events e_1 and e_2, if e_1 completes (receives a response) before e_2 begins (invokes the operation), then e_1 precedes e_2 in any linearizable ordering of events. This correctness criterion "can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object" [10].

It is questionable, however, whether linearizability is the appropriate correctness criterion when transactions include multiple operations on multiple objects. While linearizability provides a high degree of concurrency for individual objects, extending linearizability to transactions containing operations on multiple objects requires the transactions to appear to take effect instantaneously, or in "one-at-a-time" order [9]. Transactions must then operate simultaneously on the most recently written versions of all objects. To ensure this in existing STM, transactions acquire ownership (albeit using no physical locks) of all objects they operate upon; unfortunately, this limits the degree of concurrency achievable under linearizability.

To increase the opportunity for concurrent execution of transactions, we focus in this paper on STM that provide serializable, rather than linearizable, executions. Because serializability does not impose a real-time ordering on committed transactions, transactions need not acquire ownership of all objects simultaneously. For example, a read-only transaction may serialize using previously written versions, which is not possible in a linearizable system where the real-time ordering of events requires operations to access the most recently written version of data.

The main contribution of this paper presents the first serializable, lock-free STM. Our STM has several desirable properties. It supports dynamic transactions; that is, it does not require the set of objects accessed by a transaction to be known in advance. Early failure indication is provided in some cases, enabling a thread to quickly assess whether a transaction will be forced to abort. The single read and write object interface suffices for all transactions, rather than requiring a special interface for read-only transactions. Transactions on disjoint sets of objects do not require mutual helping, thereby achieving disjoint-access parallelism [12]. Finally, our STM keeps multiple versions of objects to increase concurrency, specifically enforcing *one-copy serializability*.

In the rest of this document, we describe our algorithm. The next section puts the paper in context with related work. In Sections 3 and 4, we present our system model and give an overview of our algorithm. In Section 5 we discuss briefly our assumptions for Sections 6 to 8, where we prove that *i)* the set of committed transactions is serializable, *ii)* the algorithm

[1] See [14] for a nice survey of software transactional memory systems.

terminates, and *iii)* the algorithm is lock-free, in that a correct thread can always guarantee some new transaction will eventually commit. For simplicity, our proofs rely on a wait-free implementation of object histories—we relax this requirement in Section 9, where we show that a lock-free implementation of object histories is sufficient to guarantee both the safety and liveness of our STM. The paper ends with conclusions in Section 10.

2 Related Work

Shavit and Touitou propose the first example of nonblocking software transactional memory [18]. Their STM is lock-free, but requires strong memory synchronization primitives and static transactions: a transaction must declare before executing the set of objects on which it will operate. Building on their work, Herlihy, et al., present dynamic software transactional memory (DSTM) [9]. DSTM allows transactions to dynamically determine the sequence of objects for operations, but it also relaxes the progress guarantee to obstruction freedom, which provides absolute progress guarantees only in the absence of contention. Fraser [3] combines the best characteristics of these earlier works by proposing an object-based STM that provides dynamic, lock-free transactions [3].

Our STM provides the same properties as Fraser's (it is object-based and supports dynamic lock-free transactions) but departs from all existing STM in choosing serializability, rather that linearizability, as its correctness criterion. Linearizability may appear a preferable option, since serializability is known to be an inherently *blocking* property [10]—a transaction may be forced to wait or abort because of the operation of another transaction. In contrast, linearizability, when applied to operations that affect a single object, is nonblocking—a total operation may always complete successfully. However, linearizable *multi-object* transactions are also blocking. For example, consider an execution in which transactions T_1 and T_2 *i)* attempt to enqueue object x to queues Q_1 and Q_2, respectively, and *ii)* attempt to dequeue x from the other queue. Though each queue is individually linearizable, whether T_1 successfully dequeues x from Q_2 depends upon whether T_2 dequeues x. There is no linearizable ordering of T_1 and T_2 that allows both transactions to successfully dequeue x from the other queue. In fact, as we have argued in the Introduction, serializability provides greater concurrency than linearizability in transactional systems where transactions span multiple objects.

Lock-free transactional objects have been extensively studied in real-time environments [1, 17]. These objects typically provide timeliness guarantees in an environment where threads have priorities and preemption is constrained by those priorities, simplifying shared data synchronization. Our work assumes a different computational model wherein threads do not have priorities and no bounds are placed on the relative speed of two threads.

Transactional monitors [19] have been proposed as an alternative to traditional monitors in Java. While Java monitors regulate access to shared data using mutual exclusion, transactional monitors do so using lightweight transactions. Transactional monitors provide serializability, but do not address nonblocking access—for example, a thread is not allowed to halt inside a monitor. We also guarantee serializability, but we ensure nonblocking access to shared data; for example, threads may halt arbitrarily in our model without preventing the progress of other threads.

Conditional critical regions (CCR) [11] are an elegant approach to manage shared data. Recent work [4, 5] has explored implementing non-blocking access to shared data using an obstruction-free, multi-word compare-and-swap primitive, while providing programmers with the familiar CCR interface. Since this interface is independent of the mechanism used to regulate access to shared data, we believe that our approach could also be used underneath a CCR interface.

Finally, our system is similar to optimistic concurrency control [13] in that it does not use locks to regulate which transactions should be executing concurrently, but rather may resort to aborting transactions to enforce serializability. However, while in optimistic concurrency control locks are used to enforce serializability at commit time, our system guarantees serializability without locks while providing progress guarantees.

3 System Model

Our system models a pool of cooperating threads that operate on shared data. We assume resource limitations so that only a finite number of threads exist in the system at any time, though new threads may be created at any time. Data shared between threads is represented as a set of objects that support read and write operations. Threads may run simultaneously (as on a multi-processor), and we assume no bounds on the relative speed between threads. A thread may fail by halting or be arbitrarily delayed (for example, by a page fault); a thread that does not halt is considered correct. We assume shared data is modified by threads only within transactions and that the Compare-And-Swap (CAS) primitive may be used to operate on shared data. CAS atomically and conditionally modifies a shared memory location (see Figure 1) and has been shown by Herlihy [7] to be sufficient to achieve nonblocking synchronization. We assume that a correct thread may complete a CAS operation in a finite number of steps, though the CAS may fail if the expected value is not present.

A transaction is an ordered sequence of read and/or write operations on a set of objects. The objects need not be declared at the beginning of the transaction, but may be determined dynamically. Read and write operations on different objects may be interleaved within a transaction. After all operations are complete, a thread either attempts to *commit* or *abort* the transaction, so that all operations are either visible or not, respectively. We do not address nested transactions (that is, a single thread does not interleave or nest operations of different transactions). Transactions may span multiple threads provided only one thread attempts to commit the transaction, though in this paper we will consider the thread that attempts to commit the transaction also responsible for executing the transaction. For ease of exposition, we consider all threads concurrently to execute transactions in sequence where only operations belonging to transactions performed by different threads may be interleaved. We assume any transaction executed in isolation transforms the state of the system from a correct state to another correct state, so that if transactions occur sequentially, safety is preserved. Informally, we require for correctness that there exist a sequential execution of the set of committed transactions (that may have executed concurrently) such that each operation returns the same value in both executions (that is, *view serializability* [16]).

Transactions are uniquely identified. A thread p executes the transaction T_i from the set

$\mathcal{T}$ of all possible transactions on the set O of all possible objects. Until a transaction commits or aborts, it is considered undecided ($T_i \in \mathcal{U}$); whereas a decided transaction is either *aborted* ($T_i \in \mathcal{A}$) or *committed* ($T_i \in \mathcal{C}$).

We take special care in allowing read-only transactions to proceed concurrently with writes. Borrowing from multiversion databases, we keep several versions of an object so that reads can proceed in parallel as new versions are written. Maintaining multiple versions allows reads to use past versions, permitting read-only transactions to commit concurrently with read/write transactions.

We assume for simplicity that each write operation on an object generates a complete copy of the object,[2] called a *version*. Since our system keeps multiple versions, the corresponding version is noted in the operation. A read operation is denoted $r_j[x_k]$ if transaction T_j reads version x_k of object x. A version is uniquely associated with the transaction that creates it, implying version x_k was written by T_k and the write creating it was $w_k[x_k]$. There is a total order, called the *history*, to versions of an object such that for all versions x_j and x_k, either $x_j \ll x_k$ or $x_k \ll x_j$ where $\ll$ represents the history order.

A *multiversion log* (MV log) [2] provides the framework to determine whether a set of concurrent transactions can be equivalent to a set of sequential transactions. An MV log is a poset over the set of operations belonging to a set of transactions $\{T_1, \ldots, T_n\}$ that both respects the local order of operations for each transaction, and orders any read operation $r_j[x_i]$ by T_j of a version of x written by T_i after the write operation $w_i[x_i]$ by T_i that produced the version. In this case T_j *reads-from* T_i.

A *serial log* is a totally ordered log respecting the local order of operations of a transaction where transactions are executed in isolation—for any T_i and T_j either all operations of T_i precede the operations of T_j or vice versa. We say that $j < k$ if operation j precedes k in a serial log. More formally, we define the serialization graph over MV log L, SG(L), to have vertices $T_0, \ldots, T_n$ from $\mathcal{C}$ and edges $T_i \to T_j$ (where $i \neq j$) iff T_j reads-from T_i. We use $T_i \rightsquigarrow T_j$ to indicate that there is a path in the graph SG(L) from T_i to T_j (and conversely $T_i \not\rightsquigarrow T_j$ when there is no such path). Bernstein and Goodman [2] show that if SG(L) is acyclic, then L is *serializable* (SR)—equivalent to (that is, including the same operations as) some serial log.

Many multithreaded applications assume only one copy of each object. In a multiversion system, however, the definition of serializability above is not sufficient to enforce executions consistent with one-copy semantics. We then impose the stronger constraint that transactions be *one-copy serializable* (1-SR), that is, equivalent to a *one-serial log*. In a one-serial log the last write of x before any read operation $r_j[x_i]$ is $w_i[x_i]$. A read thus always sees the latest write. To show one-copy serializability, we define the multiversion serialization graph MVSG($L, \ll$) over the MV log L and a total order $\ll$ of versions in the history of each object. The MVSG is composed of the edges in the SG(L) (called *reads-from* edges) with the addition of two other types of edges—for each $r_k[x_j]$ and $w_i[x_i]$ operation in L ($T_k \neq T_i$), we add a *write-read* edge $T_i \to T_j$ if $x_i \ll x_j$, and a *read-write* edge $T_k \to T_i$ if $x_j \ll x_i$. In the MVSG($L, \ll$) we thus

[2]This assumption allows a read to ignore earlier updates. Alternatively, each update could hold the abstract operation to be performed on the object or the portion of the object that has changed, requiring a read to perform the operations in the history to generate the relevant version of the object.

have either $T_i \rightarrow T_j \rightarrow T_k$ or $T_j \rightarrow T_k \rightarrow T_i$ for the operations $r_k[x_j]$ and $w_i[x_i]$, ensuring that no other write operation (in this case, $w_i[x_i]$) is ordered between a write operation ($w_j[x_j]$) and the corresponding read operation ($r_k[x_j]$). In [2], an MV log L is shown to be 1-SR iff there exists a version order $\ll$ such that MVSG($L, \ll$) is acyclic.

We prove in Section 6 that our algorithm enforces an acyclic MVSG($L, \ll$) where L is the multiversion log over the set of committed transactions and $\ll$ is the order imposed by the object history. To simplify discussion, we call CMVSG the MVSG where L contains operations only from the set of committed transactions ($\mathcal{C}$), and DMVSG the MVSG where L's operations come only from the set of decided transactions ($\mathcal{C} \cup \mathcal{A}$). Finally, the PMVSG is the MVSG(L,$\ll$) where L contains only operations from the set of committed and undecided transactions ($\mathcal{C} \cup \mathcal{U}$). Thus, PMVSG $\cap$ DMVSG = CMVSG. As we will see, to achieve 1-SR executions, our algorithm, before committing an undecided transaction T_i, check for cycles in the PMVSG involving T_i and takes appropriate action to ensure that, upon commit, the CMVSG will be acyclic.

4 Lock-Free Transactions

The data structures used by our algorithm are shown in Figure 1. The API for our software transactional memory in Figure 3 provides methods to read, write, create, and destroy objects and commit, abort, and validate[3] transactions. We assume the methods in the API are called on a particular transaction only by the thread that initiates the transaction. However, the helper functions called by some of these methods (shown in Figure 4) can be invoked for a given transaction by several different threads—we will discuss the helper functions and how they are invoked shortly.

A thread p begins a transaction by calling `begin-transaction`, which returns a data structure representing the transaction. This data structure is private to p until p calls `commit-transaction`, allowing fast aborts. If p calls `abort-transaction`, the data structure never becomes shared, and the transaction is aborted privately.

To read an object o, p calls `read-object`, which searches the history of o for the latest version that does not create a cycle in the PMVSG. Our algorithm checks at read time that the version being read does not create a cycle, but since objects are modified concurrently, that version could still become part of a cycle. If a cycle is detected at commit time, transactions are aborted to prevent the cycle from appearing in the CMVSG (see Section 6). Each version stores in its *reading transaction set* (RTS) the identifiers of the transactions that have read that particular version; for example, the identifier T_i is added to the RTS of x_k during the attempt to commit T_i if $r_i[x_k]$ is in T_i.

To perform an update, p calls `write-object`. To reduce cache contention, this method keeps all writes local until p calls `commit-transaction`. Whenever p calls `create-object`, it must also call `write-object` to create the initial version of the object. The procedure `destroy-object` marks an object for garbage collection during a transaction, although the object will be considered garbage only if the corresponding transaction commits.

[3] Validation returns a boolean indicating whether the transaction will be required to abort because of conflicting transactions.

```
// Global timestamp of transactions enqueued
int trans_tstamp

// Transaction data
transaction T_i
    int ts                  // Timestamp of enqueue
    transaction status      // 2 LSB indicate status
    obj_version * reads     // List of versions read
    obj_version * writes    // Cache of writes

// Data object
object x
    obj_version * history   // Sequence of versions

// Version of object written by transaction
obj_version x_k
    obj_version * next      // Next version in history
    trans_list * rts        // Reading trans set
    transaction * creator   // Trans. writing x_k
    void * data             // Object representation

// Compare-And-Swap (CAS) atomic primitive
procedure CAS( addr, expected_value, new_value )
1   atomic {
2       let old_value := value_of( addr )
3       if ( old_value = expected_value )
4           value_of( addr ) := new_value
5       return old_value
6   }
```

Figure 1: Overview of data structures and primitives used by our algorithm. The CAS synchronization primitive can be implemented on all major modern architectures.

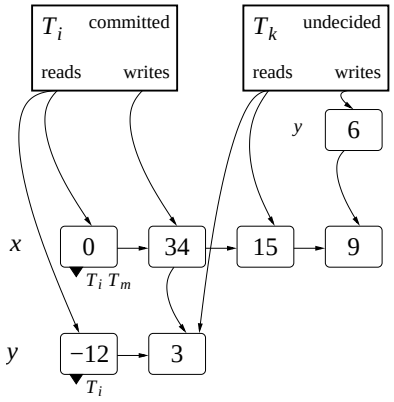

Figure 2: Object histories and transactions. The history of an object (such as x or y) are stored as a list of versions. Each version is annotated with transactions that read the version (for example, T_m reads $x = 0$). Transaction data is kept private to each thread until the transaction has finished executing when the versions written are then added to the object histories (as T_k must still add y_k where $y = 6$).

A thread attempts to commit a transaction by invoking `commit-transaction`. The algorithm to commit a transaction T, *i)* adds any read operations to the objects' histories to allow later inspection of the PMVSG (Step 1), *ii)* adds any write operations to the corresponding histories to determine the version order (Step 2), and *iii)* invokes `help-commit-transaction`. After Step 2 we consider the transaction to be *enqueued* so that other threads may help commit the transaction. The procedure `help-commit-transaction` *i)* checks for cycles in the PMVSG (Step 1 & 2), *ii)* attempts to commit T using CAS if the observed PMVSG is acyclic (Step 3), and *iii)* if the commit of T fails, recursively attempts to commit any transaction responsible for aborting T (Step 4). Read-only transactions do not need a special API because they follow the same procedure to commit as update transactions (with the obvious exception of Step 2 of `commit-transaction`).

We say that a transaction T_i is *committed* if the CAS of the status of T_i to *committed* succeeds. Conversely, T_i is aborted if the CAS of the status of T_i to *aborted* succeeds. To allow transactions to abort while guaranteeing progress, we save the id of the transaction responsible

```
// Record version written in private transaction data.
procedure write-object ( T_i, x_i )
1   x_i.read_set := ∅
2   x_i.creator := T_i
3   < add w_i[x_i] to T_i.writes >

// Read latest version of object that doesn't create a
// cycle in PMVSG and record version in transaction.
procedure read-object ( T_i, x ) returns x_k ∈ x.history
    // Loop backwards through history for latest version.
1   foreach {x_k : x_k ∈ x.history: T_k ∈ C)}
2       if ( is-readable ( x_k, trans_tstamp, T_i ) )
3           < add r_i[x_k] to T_i.reads >
4           return x_k

// Using CAS, advance global timestamp of objects,
// obj_tstamp, when a new object is created. The CAS
// ensures x.ts is greater than obj_tstamp was at the
// time the create-object method was invoked.
procedure create-object ( ) returns x ∈ O
1   < Create and init. object data structure x >
2   return x

// Mark object for garbage collection.
procedure destroy-object ( )
1   < Mark object for garbage collection >

// Abort transaction by marking status. This trans's
// operations will not be visible to any other trans.
procedure abort-transaction ( T_i )
1   CAS ( &(T_i.status), undecided,
2           aborted | mark-aborter(⊥) )

// Assign a unique transaction number and initialize
// transaction data.
procedure begin-transaction ( ) returns T_i ∈ T
1   < Create and init. transaction data structure T_i >
2   T_i.ts := ⊥
3   return T_i

// Check whether trans. is part of a cycle in PMVSG.
procedure validate-transaction ( T_i ) returns boolean
1   return T_i ↛ T_i

// Use helper function to commit transaction.
procedure commit-transaction ( T_i )
    // Step 1. Add read operations
1   foreach {x : x ∈ O : r_i[x_k] ∈ T_i}
2       let L := w_k[x_k].read_set
3       do
4           set-add-transaction ( L, T_i )
5       until ( T_i ∈ L )
    // Step 2. Enqueue write operations
6   foreach {x : x ∈ O : w_i[x_i] ∈ T_i}
7       < append once w_i[x_i] to x.history >
    // Step 3. Assign timestamp.
8   T_i.ts := trans_tstamp
9   CAS ( &(trans_tstamp), T_i.ts, T_i.ts+1 )
    // Step 4. Try to commit transaction.
10  help-commit-transaction ( T_i )
```

Figure 3: API procedures to read and write versions of objects and commit, abort, and validate transactions. The &() operator returns the address of the argument as required by CAS.

for the abort (using the `mark-aborter` function, Figure 4) in the data structure that corresponds to the aborted transaction. If a transaction T_i by thread p_i is aborted by another thread p_j attempting to commit T_j, p_i can identify p_j by using `get-aborter` (see Figure 4) and can help p_j commit T_j. Helping ensures progress even if new threads perpetually abort p_i's transactions and then promptly fail before ever successfully committing their own transactions.

Because threads may help commit transactions that they have not initiated, different threads may end up performing identical operations on the data structures that represent object histories. Despite this, these data structures should ensure that an update is only added once. Fortunately, it is easy to modify existing data structures, such as ordered lists [6] and FIFO-queues [15], to return success if the update has already been added to the list or an error indication if it cannot be determined (for example, if the list has been garbage collected possibly including the new version).

To ensure that marking the aborted transaction occurs atomically with the successful abort,

```
// Help decide reachable transactions to ensure that
// CMVSG is acyclic.
procedure decide-mvsg-reachable ( T_m )
1   let ts := T_m.ts
2   let obj_tails := ∅      // End of objects already seen
    // Step 1: Get trans. reachable on objects T_m operates
3   let pmvsg_trans := get-pmvsg-reachable ( T_m, ts, obj_tails )
    // Step 2: Decide newly reachable transactions
4   foreach {T_j : T_j ∈ pmvsg_trans : T_j ∈ U}
5       help-decide-transaction ( T_j, T_m )
    // Step 3: Save committed reachable transactions.
6   let cmvsg_trans := pmvsg_trans ∩ C
7   let checked_trans := ∅      // Trans. already seen
    // Step 4: Repeat for transitive edges from other objects
8   foreach {T_i : T_i ∈ ( cmvsg_trans − checked_trans )}
9       checked_trans ∪= {T_i}
10      pmvsg_trans := get-pmvsg-reachable ( T_i, ts, obj_tails )
        // Step 5: Decide newly reachable transactions
11      foreach {T_j : T_j ∈ pmvsg_trans : T_j ∈ U}
12          help-decide-transaction ( T_j, T_m )
        // Step 6: Save committed reachable transactions.
13      cmvsg_trans ∪= pmvsg_trans ∩ C

// Get all pmvsg-reachable nodes along single objects.
procedure get-pmvsg-reachable ( T_i, ts, obj_tails )
    returns T^pmvsg ⊂ T
1   let pmvsg_trans := ∅   // Direct edges
2   foreach {x : x ∈ O : (r_i[x_k] ∈ T_i) ∨ (w_i[x_i] ∈ T_i)}
        // Get version interval bound x_tail for x
3       if ( x_tail ∉ obj_tails )
4           let x_tail := get-last-version ( x )
5           obj_tails ∪= {x_tail}
        // Add all edges on this object by operation.
6       if ( r_i[x_k] ∈ T_i )
            // Add read-write and reads-from (x_j) edges
7           foreach {x_j : x_j ∈ x.history : (T_j.ts ≤ ts)
8                       ∧ (x_k ≪ x_j) ∧ (x_j ≠ x_i)
9                       ∧ (x_j ≪= x_tail)}
10              pmvsg_trans ∪= {T_j} ∪ get-readers ( x_j, ts )
11      if ( w_i[x_i] ∈ T_i )
            // Add reads-from ( x_i ) edges
12          pmvsg_trans ∪= get-readers ( x_i, ts )
            // Add write-read and reads-from (x_j) edges
13          foreach {x_j : x_j ∈ x.history : (T_j.ts ≤ ts)
14                      ∧ (x_i ≪ x_j) ∧ (x_j ≪= x_tail)
15                      ∧ (∃T_l | T_l ∈ T : r_l[x_j] ∈ T_l)}
16              pmvsg_trans ∪= {T_j} ∪ get-readers ( x_j, ts )
17  return pmvsg_trans

// Help commit transactions begun by any thread.
procedure help-commit-transaction( T_i )
    // Step 1. Ensure there are no cycles with T_i in CMVSG.
1   if ( T_i ∈ U )
2       decide-mvsg-reachable ( T_i )
    // Step 2. Ensure there are no cycles from write-read edges.
3   foreach {T_j : T_j ∈ T : r_i[x_j] ∈ T_i}
4       if( ¬ is-readable ( T_j, T_j.ts, T_i ) )
5           help-decide-transaction ( T_i, T_i )
6           return
    // Step 3. Commit with Compare-and-Swap status.
7   CAS ( &(T_i.status), undecided, committed )
    // Step 4. Help aborter, as needed.
8   if ( T_i ∈ A )
9       let T_j := get-aborter ( T_i )
        // Recurse only if new transaction
10      if ( T_j ≠ T_i )
11          help-commit-transaction ( T_j )

// Help undecided transactions begun by any thread.
procedure help-decide-transaction( T_i, T_h )
1   if ( T_i ∈ U )
        // Help commit only earlier transactions.
2       if ( T_i < T_h )
3           help-commit-transaction ( T_i )
4       else
5           CAS ( &(T_i.status), undecided,
6                   aborted | mark-aborter ( T_h ) )

// Get finite set of transactions that have read version x_i
procedure get-readers ( x_i, ts ) returns T^r ⊂ T
1   return {T_j : T_j ∈ T : (r_j[x_i] ∈ T_j) ∧ (T_j.ts ≤ ts)}

// Return aligned pointer to transaction.
procedure mark-aborter ( T_h )
1   return &(T_h)

// Return transaction pointer masked from status field.
procedure get-aborter ( T_i ) returns T_j ∈ T
1   return T_j : &(T_j) = < T_i.status with cleared 2 LSB >

// Return last committed version of object.
procedure get-last-version ( x ) returns x_i ∈ x.hist
1   return x_i : (x_i ∈ T_i) ∧ (T_i ∈ C)
2               : (∀x_j | (x_j ∈ x.hist ) ∧ (x_j ∈ T_j) ∧ (T_j ∈ C)
3                       : x_j ≪= x_i)
```

Figure 4: Procedures to help commit transactions. Aborted transactions and those not enqueued are implicitly ignored. The notation "$\overset{\cup}{=}$ " indicates assignment to the variable on the left-hand side, the variable's value (before assignment) unioned with the right-hand side. The boolean operator $(x_i \ll= x_j)$ is equivalent to $(x_i \ll x_j) \vee (x_i = x_j)$. The operation `min` at line 28 returns the earlier version of the pair according to the object history. If one version does not exist (for example, x_k if T_i did not read x), the other version is returned.

```
// Determine whether reading version creates a cycle.
procedure is-readable ( T_m, ts, T_h ) returns boolean
1   let obj_tails := ∅
    // Step 1. Add possible read-write
    and reads-from (x_j) edges
2   let cmvsg_trans :=
3       get-cmvsg-reachable+ ( T_m, ts, obj_tails, T_h )
4   let checked_trans := ∅      // Trans. already seen
    // Step 2: Get transitive edges from other objects
5   foreach {T_j : T_j ∈ ( cmvsg_trans − checked_trans )}
6       checked_trans ∪= {T_j}
7       cmvsg_trans ∪=
8           get-cmvsg-reachable+ ( T_j, ts, obj_tails, T_h )
        // Step 3: Check whether there is a cycle.
9       if ( T_m ∈ cmvsg_trans )
10          return false
11  return true
```

```
// Get all committed transactions with the addition of T_h
// reachable along single object paths from T_i.
procedure get-cmvsg-reachable+ ( T_i, ts, obj_tails, T_h )
    returns T^{cmvsg+} ⊂ T
1   let cmvsg_trans := ∅   // Direct edges
2   foreach {x : x ∈ O : (r_i[x_k] ∈ T_i) ∨ (w_i[x_i] ∈ T_i)}
        // Get version interval bound x_tail for x
3       if ( x_tail ∉ obj_tails )
4           let x_tail := get-last-version ( x )
5           obj_tails ∪= {x_tail}
        // Add all edges on this object by operation.
6       if ( r_i[x_k] ∈ T_i )
            // Step 1. Add read-write edges
7           foreach {x_j : x_j ∈ x.history : (T_j.ts ≤ ts)
8                       ∧ ((T_j ∈ C) ∨ (T_j = T_h))
9                       ∧ (x_k ≪ x_j) ∧ (x_j ≠ x_i)
10                      ∧ (x_j ≪= x_tail)}
11              cmvsg_trans ∪= ({T_j}
12                  ∪ get-readers ( x_j, ts ) ) ∩ (C ∪ {T_h})
13      if ( w_i[x_i] ∈ T_i )
            // Step 2. Add reads-from edges
14          cmvsg_trans ∪=
15              get-readers ( x_i, ts ) ∩ (C ∪ {T_h})
            // Step 3. Add write-read edges
16          foreach {x_j : x_j ∈ x.history : (T_j.ts ≤ ts)
17                      ∧ ((T_j ∈ C) ∨ (T_j = T_h))
18                      ∧ (x_i ≪ x_j) ∧ (x_j ≪= x_tail)
19                      ∧ (∃T_l | T_l ∈ (C ∪ {T_h}) : r_l[x_j] ∈ T_l)}
20              cmvsg_trans ∪= ({T_j}
21                  ∪ get-readers ( x_j, ts ) ) ∩ (C ∪ {T_h})
22  return cmvsg_trans
```

Figure 5:

we use only the two least significant bits of the status field to show the committed, aborted, or undecided status,[4] while if the transaction is aborted, the rest of the bits represent an aligned pointer to the transaction responsible for the abort—we are similar in this to others who have used the least significant bits of pointers to mark, for example, the logical deletion of a member from a list [6].

Threads help commit or abort transactions not only to guarantee progress, but also to prevent cycles in the CMVSG, ensuring one-copy serializability. We prevent such cycles despite concurrent modifications of object histories in two steps. First, whenever p calls `commit-transaction`(T_i) we let the undecided T_i add its operations to the object histories, making them visible to other threads—this results in new edges being added to the PMVSG. Second, we only attempt to commit an undecided, enqueued transaction after we can be sure that the

[4] If we may reasonably assume that 0 is an invalid pointer and that pointers are aligned, we can use a single bit. If the status is 0, the transaction is undecided; 1 implies committed; any other value represents the pointer to the aborting transaction.

CMVSG that would result from committing T_i is acyclic. The procedure responsible for enforcing this check is `decide-mvsg-reachable`, shown in Figure 4 and invoked in line 2 of `help-commit-transaction`.

Intuitively, `decide-mvsg-reachable` should identify all transactions that are reachable from T_i along a path of committed transactions. If T_i is reachable from itself, then it should be aborted. Checking for the presence of such a cycle, however, presents a challenge. It is not sufficient to check for the absence of a cycle involving T_i in the current CMVSG. That check would cover only transactions that are already committed, while the cycle may involve transactions that are going to commit concurrently with T_i—when we perform the check, the cycle may not yet have appeared in the CMVSG. On the other hand, it is not necessary to require that the PMVSG contain no cycles involving T_i as a precondition for committing T_i. If such a cycle is detected in the PMVSG, the cycle may include undecided transactions other than T_i—to prevent a cycle in the CMVSG it is sufficient to abort any one of these transactions.

The approach we use in `decide-mvsg-reachable` is to visit each enqueued transaction that is reachable from T_i in the PMVSG along edges on a single object by building the set pmvsg_-trans (Step 1 of the function). The algorithm then saves the transactions that are committed in the set cmvsg_trans to search paths over multiple objects (Step 2 of the function). When an undecided transaction is found, the algorithm attempts either to commit or to abort the transaction (we are going to discuss the appropriate course of action in a moment). If in so doing we return to T_i on a path that includes only committed transactions, then to guarantee an acyclic CMVSG, we abort T_i. Finally, the outer loop of the function performs a breadth-first search for transactions transitively reachable from T_i by iterating the previous process over committed transactions saved in the set cmvsg_trans.

We can now go back to the problem of deciding what the thread p that initiated transaction T_i should do when, as it visits the transactions reachable from T_i in the PMVSG, it finds an undecided transaction, T_j. It is tempting, in the interest of ensuring progress, to always have p try to commit T_j. Unfortunately, helping to commit *all* undecided transactions reachable in the PMVSG does not work. Suppose two transactions T_i and T_j by threads p_i and p_j respectively, both add updates to the histories of objects x and y in the opposite order—without loss of generality, consider $x_j \ll x_i$ and $y_i \ll y_j$. Thread p_i may then help commit T_j as $T_i \rightarrow T_j$ from object y, while p_j helps to commit T_i because $T_j \rightarrow T_i$ from object x. Clearly, one of the transactions must abort to guarantee no cycles in the CMVSG.

To choose in general which transaction to abort, we introduce a total order on undecided transactions, called the *transaction priority order* (TPO).[5] We say that $T_i < T_j$ (or that T_i has higher priority than T_j) iff T_i precedes T_j in the transaction priority order; further, we define the *distance* between T_i and T_j as the number of transactions T_k such that $T_i < T_k < T_j$. We require two properties of our total order: first, it must have a minimum; and second, the distance between any two transactions must be finite. A simple way to achieve both properties is to order transactions according to a timestamp—ties can be broken using the id of the thread that intiates the transaction so that timestamps need not be unique across threads.

[5] A similar order is required in other STM. For example, in FSTM [3] the address of objects is used as a total order to prevent the cycle described.

Note that the TPO is not used to determine the order in which transactions will eventually commit—transactions with higher priority may well commit after transactions with lower priority. Rather, the TPO is used to determine the fate of any undecided, enqueued transactions T_j that a thread p encounters as it checks the PMVSG for cycles involving T_i, the transaction that p is trying to commit. Specifically, p will attempt to commit only those T_j that have higher priority than T_i; the rest, p will attempt to abort. The two properties of the TPO ensure that any recursion that may be triggered while applying this policy will terminate.

5 A Prelude to the Proofs

In the following sections we show that our algorithm is both lock-free and commits only a one-copy serializable set of transactions. Our proofs depend on three assumptions.

First, we assume that all correct threads attempt to commit a transaction infinitely often. This *commit assumption* eliminates the trivial condition in which a correct thread aborts all attempted transactions.

Second , we assume that the FIFO queue implementing object histories provides a wait-free insert operation. We use this assumption only to simplify our initial presentation of the proofs; in Section 9 we will relax this assumption and discuss an implementation of our algorithm that relies only on lock-free FIFO queues.

Third, we assume that a thread can acquire read access to the lists of versions that represents each object's history and, within each version, to the reading transaction set—even while these structures are concurrently updated. We call this the *prefix-read assumption*—specifically, it implies that (i) it is possible to determine x_{tail} in finite time in `get-last-version` (Figure 4) and that (ii) the function `get-readers` terminates and returns a finite set of transactions. Both requirements can be met using existing algorithms: it is easy to determine x_{tail} from the tail of an unmodified implementation of Michael and Scott FIFO queue [15]; furthermore, the RTS can be implemented using the ordered list of Harris [6] with only slight modifications. In particular, during any operation on the list we set a pointer[6] to the item currently at the end of the list. This guarantees that when `get-readers` begins, the end of the list can be determined even though new items may be added to the list during the execution of `get-readers`.

As a shorthand in the proofs, we use `hct` as an abbreviation of `help-commit-transaction` and `dmr` as an abbreviation of `decide-mvsg-reachable`. As before, transactions are ordered $T_i < T_j$ according to the TPO, which has a minimum and a finite distance between any two transactions. Again, we say that a transaction T_i is *enqueued* if the corresponding timestamp has been assigned ($T_i.\text{ts} \neq \perp$).

Finally, our proofs ignore all transactions that are explicitly aborted by a thread through `abort-transaction`. It is safe to do so because the operations performed by these transactions are invisible to all other transactions—to become visible, the operations need to be enqueued, and operations are enqueued only as a consequence of executing `commit-transaction`.

[6] This is similar to the tail pointer update in Michael and Scott's FIFO queue algorithm [15].

6 One-Copy Serializability

We prove the algorithm is one-copy serializable by showing there are no cycles in the CMVSG—the MVSG(L,$\ll$) where L is the MV log over the set $\mathcal{C}$ of committed transactions, and $\ll$ is the order in which versions appear in the history.

Lemma 1.1 If the function invocation `hct`(T_i) terminates, T_i is decided.

Proof. If `hct`(T_i) terminates, the Compare-And-Swap on the status of T_i in step 2 of the algorithm must have completed. Either the CAS succeeds, in which case the status of T_i is *committed*; or the CAS fails, implying that the status of T_i must not have been *undecided* when CAS was invoked—that is, T_i was already *aborted* or *committed*. In either case, T_i is decided when `hct`(T_i) terminates. □

Lemma 1.2. Let T_j be an enqueued transaction that reads a version x_i and with a timestamp no later than timestamp ts. An invocation of `get-readers`(x_i, ts) then returns a set containing T_j.

Proof. The Lemma is direct from the definition of the function in Figure 4 and the prefix-read assumption. □

The next lemmas hold that the algorithm correctly tracks the necessary paths in PMVSG to ensure that the CMVSG is acyclic. We will only consider paths in the PMVSG containing enqueued transactions; that is, transactions that have all of their operations enqueued. The lemma allows us to later assume that if a transaction T_i is committed, there were no cycles in the PMVSG containing T_i during the execution of `hct`(T_i). We will then argue that some transaction in the cycle must have been able to detect the cycle before committing, allowing us to derive a contradiction if a cycle is created.

Lemma 1.3. Consider an invocation of `gpr`(T_i, ts, obj_tails). Let T_j be such that *i)* there exists, when `gpr` is invoked, a path $T_i \rightsquigarrow T_j$ in the PMVSG where *ii)* all the edges in the path correspond to operations on some object x, and *iii)* all transactions in the path have timestamps no later than ts. Further, *iv)* the version x_k operated upon by T_j is no later than x_{tail} if $x_{\text{tail}} \in$ obj_tails. Then, the set returned by `gpr` contains T_j.

Proof. Since the outer loop of `gpr` is over every object on which T_i operates, we consider only the execution of the loop over the object x on which the path $T_i \rightsquigarrow T_j$ is defined.

Consider the shortest path between T_i and T_j. If the operation of T_j responsible for the last edge in that path is a write, then the path is of length one, since in that case the edge is either read-write or write-read by definition. Transaction T_j is added to pmvsg_trans for the former type of edge at line 9 where the guard of the loop holds because *i)* $x_k \ll x_j$ by definition of the read-write edge, *ii)* T_j.ts $\leq$ ts by condition *(iii)* of the Lemma, and similarly *iii)* $x_j \ll\!= x_{\text{tail}}$[7] by condition *(iv)* of the Lemma. On the last point, we also note that since T_j is enqueued before `gpr` is invoked, then the version operated upon by T_j must be enqueued before `get-last-version`

[7]The boolean operator $(x_i \ll\!= x_j)$ is equivalent to $(x_i \ll x_j) \vee (x_i = x_j)$.

determines the tail of x.hist if $x_{\text{tail}} \notin$ obj_tails. The loop is reached during the iteration because $r_i[x_j]$ exists (at line 6) by definition of the read-write edge $T_i \rightarrow T_j$.

If the edge is write-read, T_j is added to pmvsg_trans at line 15. The loop is reached again because $w_i[x_i]$ exists by definition of the write-read edge $T_i \rightarrow T_j$. The loop guard holds in this case for reasons identical to those for the read-write edge. Additionally, by condition *(i)* of the Lemma there exists a read of x_j by some transaction that satisfies the loop guard at line 14 before `gpr` is invoked.

Suppose instead T_j's operation is a read. If T_j reads the version written by T_i, then there exists a direct reads-from edge between T_i and T_j. In this case, T_j is added to pmvsg_trans at line 11 when T_j is returned by `get-readers`. Transaction T_j is enqueued by condition *(i)* of the Lemma, and T_j.ts $\leq$ ts by condition *(iii)*, allowing us to apply Lemma 1.2 to show that T_j is returned by `get-readers`.

If instead T_j reads some version x_k after the version read or written by T_i, then by definition there exists a reads-from edge between T_k and T_j and an edge $T_i \rightarrow T_k$ that is either a read-write or write-read edge, depending on whether the operation of T_i is, respectively, a read or a write. Transaction T_k is added to pmvsg_trans at line 9 or 15, as argued above with a slight change: in this case the version operated upon by T_j is x_k. The inequality $x_k \ll= x_{\text{tail}}$ holds by condition *(iv)* of the Lemma or by condition *(i)*—that T_j is already enqueued, depending upon whether $x_{\text{tail}} \in$ obj_tails or not, respectively. As in the case of the direct reads-from edge, `get-readers` will return T_j (again by conditions *(i)* and *(iii)*) when T_k is added, proving that T_j is also added to pmvsg_trans. □

Lemma 1.4. Consider an invocation of `gcr+`(T_i, ts, obj_tails, T_h). Let T_j be such that *i)* there exists, when `gcr+` is invoked, a path $T_i \rightsquigarrow T_j$ in the PMVSG where *ii)* all the edges in the path correspond to operations on some object x, *iii)* all transactions in the path have timestamps no later than ts, and *iv)* all operations defining edges in the path belong to transactions in the set $(\mathcal{C} \cup \{T_h\})$. Further, *v)* the version x_k operated upon by T_j is no later than x_{tail} if $x_{\text{tail}} \in$ obj_tails. Then, the set returned by `gcr+` contains T_j.

Proof. The proof is similar to that of Lemma 1.3 with a modification for the extra condition *(iv)*. The extra assumption given in *(iv)* is necessary because the definition of `gcr+` is similar to that of `gpr` except that all operations defining edges belong to transactions in the set $(\mathcal{C} \cup \{T_h\})$ by construction, as given by condition *(iv)*. □

Lemma 1.5. Suppose *i)* $T_m \rightsquigarrow T_j$ in PMVSG (T_j not necessarily distinct from T_m) at the time `dmr`(T_m, o_{b}) is executed by a correct thread, and *ii)* T_m has the latest timestamp of any transaction in the path. If *iii)* the transactions in the path (possibly excluding T_m) are eventually committed, then T_j is a member of pmvsg_trans during an execution of Step 2 or 5 of `dmr` by the thread.

Proof. By induction on the length of the path $T_m \rightsquigarrow T_j$. The base case is a single edge $T_m \rightarrow T_j$. By Lemma 1.3, T_j must be added to pmvsg_trans when initialized at Step 1 of the procedure, implying T_j is a member of pmvsg_trans during Step 2. The conditions for applying Lemma 1.3 are discharged as follows: *i)* $T_m \rightarrow T_j$ in PMVSG before `dmr` is invoked, *ii)* every edge is defined

over a single object; *iii)* T_m is assumed to have the latest timestamp in the path; and *iv)* obj_tails is empty for this invocation of `gpr`.

The induction step of our proof assumes T_k is added to pmvsg_trans for any path $T_m \rightsquigarrow T_k$ of length up to n where all transactions (possibly excluding T_m) are enqueued and eventually committed. We prove that extending the path to $T_m \rightsquigarrow T_k \rightarrow T_j$ requires T_j to be added to pmvsg_trans before Step 5. Note that Lemma 1.3 can be applied as above if the path $T_m \rightsquigarrow T_j$ is over a single object using the argument of the base case where condition *(ii)* holds instead by construction of the path. In this case T_j would still be added at Step 1 of `dmr`. We now focus on paths over multiple objects where T_j is not added at Step 1.

By the induction hypothesis, T_k is added to pmvsg_trans before Step 2 or 5, so the correct thread that invoked `dmr` will either find T_k committed or will invoke `hdt`(T_k, T_m). In `hdt`, either `hct`(T_k) is invoked, resulting in T_k being decided by Lemma 1.1, or CAS is called in an attempt to abort T_k. The CAS if successful, would abort T_k, which contradicts our assumption that $T_k \in \mathcal{C}$. Hence, if the CAS is called, T_k must already be decided. In either case, when `hdt` returns T_k is decided; further, T_k must be committed by the induction hypothesis. At Step 3 or 6 then, T_k is added to cmvsg_trans, and eventually the correct thread will execute an iteration of the loop at Step 4 over T_k.[8]

The edge $T_k \rightarrow T_j$ is defined over some object y where by assumption the corresponding operations are already enqueued before `dmr` executes. During the iteration of Step 4 over T_k, at line 10 `gpr` will return T_j to be added to pmvsg_trans using the same reasoning as the base case. We again apply Lemma 1.3 to show that T_j will be added to pmvsg_trans where the conditions of the Lemma are discharged as in the base case with the exception of the last. The set obj_tails is initially empty—x_{tail} must be determined during some execution of `gpr`. The operations of T_j are enqueued before `dmr`, and thus `gpr`, is executed so that the version of y operated upon by T_j can be no later than the tail of y.hist determined during `gpr`. □

Lemma 1.6. Let *i)* T_i be a transaction that reads version x_m. Suppose *ii)* $T_m \rightsquigarrow T_j$ in MVSG($\mathcal{C} \cup \{T_i\}$, $\ll$) (T_j not necessarily distinct from T_m) at the time `is-readable`(T_m, ts, T_i) is executed by a correct thread, and *iii)* the timestamp of every transaction in the path is no later than ts. Then, T_j is a member of pmvsg_trans during Step 3 of the procedure.

Proof. We prove the lemma by induction on the length of the path $T_m \rightsquigarrow T_j$ in the PMVSG. The base case is the single edge $T_m \rightarrow T_j$. Suppose conditions *(i)* through *(iv)* of the Lemma are met. By Lemma 1.4, T_j must be returned by `gcr+` at Step 1 of `is-readable`. The conditions of Lemma 1.4 are discharged as follows, *i)* condition *(ii)* of this Lemma implies $T_m \rightarrow T_j$ before `gcr+` is invoked; *ii)* the edge $T_m \rightarrow T_j$ is defined over a single object; *iii)* condition *(iii)* of this Lemma is the same; *iv)* ; and *v)* x_{tail} is initially the empty set. The thread must then execute Step 2 at least once because cmvsg_trans is nonempty, containing at least T_j, while checked_trans is initially empty. During the iteration of Step 2, T_j must be a member of cmvsg_trans when Step 3 is reached, as transactions are never removed from the set.

[8] We show later that `gpr` returns a finite set so that cmvsg_trans grows finitely at each iteration of Step 4.

For the induction hypothesis, we assume that the Lemma holds for the path $T_m \rightsquigarrow T_k$ of length n and proceed to show that it also holds for the extended path $T_m \rightsquigarrow T_k \rightarrow T_j$. By the induction hypothesis T_k will be added to cmvsg_trans, and eventually the thread will execute an iteration of Step 2 over T_k.[9] During this iteration, we again apply Lemma 1.4 to show that `gcr+`(T_k, ts, obj_tails, T_i) will return T_j. The conditions of the Lemma are met as follows, *i)* the path $T_m \rightsquigarrow T_j$ exists by condition *(ii)* of this Lemma before `gcr+` is invoked; *ii)* the edge $T_k \rightarrow T_j$ is defined over a single object; *iii)* condition *(iii)* of this Lemma is the same; and *iv)* since T_j is enqueued (assigned a timestamp) before the function is invoked, the object version operated upon by T_j is enqueued before `is-readable` is invoked, and thus appears in the history no later than the current tail of the object. Transaction T_j will then be a member of cmvsg_trans during the execution of Step 3 that follows the invocation of `gcr+`.

□

We now prove that the algorithm commits only 1-SR transactions.

Theorem 1. The set of committed transactions is one-serializable.

Proof. We prove the Theorem by showing that the CMVSG is acyclic [2] using a proof by contradiction.

Suppose, by way of contradiction, that such a cycle existed in the CMVSG. Since each transaction in the cycle is committed, each is assigned a timestamp. Transaction timestamps are totally ordered. Let transaction T_l have the latest timestamp of the transactions whose operation(s) defines an edge(s) in the cycle. A write-read edge $T_i \rightarrow T_j$ is defined by an read operation $r_l[x_j]$ if $x_i \ll x_j$. Note that the reading transaction is not an endvertex of the edge, though the read operation is required to define the edge, allowing a cycle to be created without the reading transaction. We first show that the cycle cannot contain T_l; that is, if T_l does not create a write-read edge, there can be no cycle.

Consider the invocation of `dmr`(T_l, o_b) from `hct`(T_l) in which T_l is committed (that is, the CAS at Step 3 succeeds)—note that at this time, T_l is still undecided. Further, all transactions that, once committed, will generate the cycle in the CMVSG have already enqueued their operations before `hct`(T_l) is invoked since a timestamp is only assigned after all of the corresponding transaction's operations are enqueued, and T_l has the latest timestamp. The cycle containing T_l, therefore, is already present in the PMVSG at the invocation of `dmr`(T_l, o_b). By Lemma 1.5, T_l, which by construction has the latest timestamp in the cycle, must be a member of pmvsg_trans during Step 2 or 5 of the procedure. In either case, the thread executes `hdt`(T_l, T_l) since T_l is not committed until `dmr` returns. For this invocation of `hdt`, T_h and T_i both represent T_l, forcing execution of the CAS to abort transaction T_l that is still undecided. Hence, assuming a cycle of committed transactions including T_l implies that T_l is aborted, providing a contradiction.

Suppose instead that a read $r_l[x_j]$ by T_l creates a write-read edge $T_i \rightarrow T_j$ in the cycle in the CMVSG. For the edge to appear in the CMVSG, T_l must be committed. Again, consider the execution of `hct`(T_l, o_b) by the thread that successfully commits T_l in Step 3 of `hct`. In Step 2,

[9] The **foreach** loop can execute over the set in order by TPO, ensuring that each transaction is eventually considered.

for each version read by T_l, the thread invokes `is-readable` on the version; we focus on the invocation `is-readable`(T_j, T_l.ts, T_l). We apply Lemma 1.6 to show that T_j must be a member of cmvsg_trans during the execution of Step 3 so that the invocation of `is-readable` must return false. The conditions of Lemma 1.6 are discharged as follows, *i)* T_l reads x_j by construction; *ii)* all operations defining the cycle at this point belong to transactions in the set $(\mathcal{C} \cup \{T_l\})$ by assumption and are enqueued before `hct`(T_l) is invoked because T_l has the latest timestamp, and timestamps are only assigned after all operations of the corresponding transaction are enqueued; and *iii)* T_l has the latest timestamp of the transactions with operations defining the cycle by construction. Since `is-readable`(T_j, T_l.ts, T_l) returns false, the thread executes the CAS following. By construction T_l is not yet decided—indeed, not yet committed—until Step 3 of `hct`. Hence, the CAS will succeed, aborting T_l and providing the contradiction.

□

7 Termination

In Theorem 2 we prove termination for an invocation of `commit-transaction`. In Section 5, we assumed that the object histories were implemented using wait-free queues and that a lock-free ordered list implemented the reading transaction set. We argue here that the insert operation in the reading transaction set also has wait-free access guarantees although it is implemented using a lock-free ordered list—simply retrying the operation until success is sufficient. We consider only insertions to the list as transactions are never removed.[10] The lock-free ordered list of Harris [6] can represent the reading transaction set. The order of insertion is unimportant for the set property, so we order transactions in the list by transaction priority order. In the list an insertion of T_i then fails or starves only due to a concurrent insertion of T_j where $T_j < T_i$. The transaction priority order provides a finite distance between T_i and the minimum in the order, bounding the number of failed attempts to insert T_i at that distance. Hence, if retried continuously as in the loop in lines 3–5 of `commit-transaction`, every insert operation eventually succeeds.

Given that they perform effectively wait-free operations (by design or assumption), Steps 1 and 2 of `commit-transaction` are guaranteed to complete for any correct thread, leaving us to show that Step 3 completes to show termination. The CAS terminates by assumption. Finally, we note that an invocation of `hct` will clearly terminate if *i)* the invocation of `dmr` in Step 1 terminates; *ii)* the invocation of `is-readable` and `hdt` in Step 2 terminate; and *iii)* the recursive invocation of `hct` in Step 4 terminates. Again, the CAS in Step 3 terminates by assumption, and the `get-aborter` procedure clearly terminates from Figure 4. We discharge the assumptions in order. First, we prove that `gpr` terminates and use that lemma to show `dmr` terminates provided invocations of `hdt` (and thus recursive invocations of `hct`) terminate. Second, we show similarly that `is-readable` terminates. Finally, we prove that `hct` itself terminates.

Lemma 2.1. An invocation of `gpr`(T_i, ts, obj_tails) by a correct thread p terminates and returns a set containing transactions T_j such that T_j.ts $\leq$ ts.

[10] The entire set of reading transactions may be garbage collected when the version is no longer needed, rather than incrementally removing members from the set.

Proof. The procedure terminates if all of the loops within it terminate because the functions `get-readers` and `get-last-version` terminate by the prefix-read assumption. The main loop (lines 2–16) iterates over the finite set of objects on which T_i operates. Within each of its iterations two more loops are executed, at lines 7–10 and 13–16, over the finite set of versions that precede x_{tail} in the history of x. The main loop thus terminates because all the loops it contains are over finite sets.

Transaction T_j added to the set pmvsg_trans at lines 10 and 16 is required to have a timestamp no later than ts by the loop guards (lines 7 and 13, respectively). The transactions returned by `get-readers` and added to pmvsg_trans at lines 10 and 16 all have timestamps no later than ts according to the definition of `get-readers`. □

Given that an invocation of `gpr` terminates, the following lemma easily shows that `dmr` terminates.

Lemma 2.2. An invocation of `dmr`(T_m) terminates provided invocations of `hdt` terminate.

Proof. We first note that cmvsg_trans is created by invocations of `gpr` with the same timestamp ts. Since these invocations return only a finite set of transactions with timestamps no later than ts by Lemma 2.1, the set cmvsg_trans is bounded in size. Timestamps are assigned such that only concurrent transactions obtain the same timestamp, and transactions that begin after another commits must have a later timestamp.

The procedure terminates because the function invocations of `gpr` terminate by Lemma 2.1 and `hdt` by assumption. Further, the loops in Step 1 and 4 (lines 4–5 and 11–12, respectively) are over the finite set of transactions returned by `gpr`. The loop in Step 4 (lines 8–13) is over the set of transactions cmvsg_trans previously shown to be finite. □

In order to show `is-readable` terminates for a correct thread, we first argue that `gcr+` (like `gpr`) terminates.

Lemma 2.3. An invocation of `gcr+`(T_i, ts, obj_tails, T_h) by a correct thread p terminates and returns a set containing transactions T_j such that T_j.ts $\leq$ ts.

Proof. The proof of termination of `gcr+` is similar to that of Lemma 2.1. The loop guards of `gcr+` are simply more restrictive than those of `gpr`, requiring that operations belong to the set $(C \cup \{T_h\})$ rather than the set of enqueued transactions. □

Using the previous lemma, we can show termination for `is-readable`. Since `is-readable` does not recursively invoke `hct`, the proof requires no extra assumptions.

Lemma 2.4. An invocation of `is-readable`(T_m, ts, T_j) terminates.

Proof. The set cmvsg_trans created during `is-readable` contains only transactions with timestamps no later than ts by Lemma 2.3. This set is bounded because timestamps are assigned such that only concurrent transactions obtain the same timestamp, and transactions that begin after another commits must have a later timestamp. Hence, the procedure terminates because the

loop in Step 2 (lines 5–10) is over the finite cmvsg_trans, and the invocations of gcr+ terminate according to Lemma 2.3. □

We now show that hct itself terminates.

Lemma 2.5. An invocation of hct(T_i) by a correct thread terminates.

Proof. We first show that for any recursive invocation of hct(T_k) from hct(T_m) (possibly indirectly through hdt(T_k, T_m) from dmr(T_m) (lines 5 and 12) from hct(T_m) (line 2)), $T_k < T_m$. The order property holds for the recursive invocation of hct from hdt once we substitute T_k for T_i and T_m for T_h in the guard on line 2.

Consider now the recursive invocation of hct(T_j) from hct(T_i) at line 11 of hct(T_i), and substitute T_m for T_i and T_k for T_j. The guard at line 10 requires $T_k \neq T_m$; further, the function get-aborter(T_m) (line 9) returns the transaction T_k, implying T_m.status was set by a CAS with mark-aborter(T_k) during hdt(T_m, T_k). The guard at line 2 of hdt requires $T_m \geq T_k$. Remembering that the guard of hct required $T_k \neq T_m$ implies $T_k < T_m$ for any invocation of hct(T_k) from hct(T_m). The recursive invocations of hct are thus only upon transactions earlier in the TPO as argued above. We defined TPO to have a minimum transaction and a finite distance between any two transactions, ensuring that the number of recursive invocations of hct is bounded.

We have shown that recursive invocations of hct will terminate provided hct terminates without recursion. The function hct will terminate without recursion if each function invocation terminates since the loop in Step 2 is over the finite set of operations in T_i. The invocation of dmr terminates according to Lemma 2.2 while the invocation of is-readable terminates by Lemma 2.4. The procedure hdt and get-aborter both clearly terminate by inspection, although hdt executes CAS. As in Step 3 of hct, CAS terminates by definition.

□

Now we prove our general termination theorem.

Theorem 2. An invocation of commit-transaction(T_i) by a correct thread will terminate in a finite number of steps.

Proof. Step 1 terminates if reading transaction sets are implemented with a lock-free list ordered by transaction priority order. Eventually every transaction will be added to the list on retry given the assumptions of the transaction priority order. By assumption for this proof, wait-free queues are used to implement object histories, providing termination for Step 2. Step 3 terminates by the definition of the CAS primitive. The invocation of commit-transaction(T_i) then terminates because hct(T_i) terminates in Step 3 by Lemma 2.5. □

8 Lock-Freedom

To prove the protocol lock-free, we show that in a finite number of steps some new (that is, previously uncommitted) transaction will commit provided there is at least one correct thread. We build on Theorem 2, which states that the invocation of hct terminates for some correct

thread p. By the commit assumption, a correct thread p will infinitely often attempt to commit a new transaction, ensuring that a sequence of transactions are attempted. The following lemma holds that there exists an infinite sequence $\mathcal{T}^p$ of transactions where hct(T_i) is invoked and T_i is decided for each transaction T_i in the sequence provided there is at least one correct thread.

Lemma 3.1. If there is at least one correct thread, there exists an infinite sequence of decided transactions $\mathcal{T}^p$ such that for each transaction T_i in the sequence, some correct thread executes hct(T_i).

Proof. We consider the sequence of transactions $\mathcal{T}^p$ such that a single correct thread p invokes commit-transaction(T_i) for each transaction T_i in the sequence. The commit assumption holds that p will always attempt a new transaction by calling commit-transaction. The execution of commit-transaction(T_i) terminates by Theorem 2, implying the sequence $\mathcal{T}^p$ is infinite. Further, during the execution of commit-transaction(T_i) p invokes hct(T_i) at Step 3, and since p is correct T_i is decided by Lemma 1.1. □

Note that the lemma refers only to *decided* transactions—lock freedom requires us to show that from $\mathcal{T}^p$ there exists an infinite sequence of committed transactions. We show that for each transaction T_i in the sequence, we can find a committed transaction T_c such that T_c could not have been committed before T_i began, ensuring that we can generate an infinite sequence of committed transactions from $\mathcal{T}^p$. Towards this goal, we define two subsets over $\mathcal{T}^p$: (i) helped(T_i) $\equiv \{T_h \mid$ hct(T_h) was recursively called during the execution of hct(T_i)$\}$ and (ii) benefactors(T_i) $\equiv \{T_p \mid$ hct(T_i) was recursively called during the execution of hct(T_p) $\}$. Informally, helped(T_i) includes the transactions that T_i helped towards a decision, while benefactors(T_i) includes the transactions that T_i was helped by. We will use the set helped(T_i) to help find a committed transaction, but we first bound the number of benefactors for a single transaction.

Lemma 3.2. For any decided transaction T_i, benefactors(T_i) is finite.

Proof. We restrict our attention to recursive invocations of hct(T_i), because the only transaction to join benefactors(T_i) from a non-recursive invocation of hct(T_i) (in commit-transaction(T_i)) is T_i itself. A thread invokes hct in two different places in the algorithm.

The invocation in hdt requires $T_i \in \mathcal{U}$. By the Lemma T_i is eventually decided, ensuring that only concurrent invocations of hdt(T_i, T_h) will find $T_i \in \mathcal{U}$. Hence, the finite bound on the number of threads ensures that only a finite number of threads will help T_i by invoking hdt.

The procedure hct(T_i) may also be recursively invoked during Step 3 of hct(T_k), for some transaction T_k, if get-aborter(T_k) = T_i where $T_k \neq T_i$. The function get-aborter returns T_i set by mark-aborter(T_i) executed in hdt(T_k, T_i),[11] possibly by another thread. The procedure hdt(T_k, T_i) is in turn invoked by threads only from dmr(T_i) if $T_k \neq T_i$.

The execution of dmr(T_i) eventually terminates by Lemma 2.5, implying that only a finite number of transactions T_k may be aborted for any invocation of dmr(T_i). Further, dmr(T_i) is

[11] Transactions aborted from abort-transaction are not enqueued, and thus can safely be ignored. Their operations remain private and unreachable in the PMVSG.

invoked only from hct(T_i) when T_i is undecided, implying, as above, a finite bound on the invocations of dmr(T_i) by any thread. Hence, only a finite number of transactions may recursively invoke hct(T_i) from hct(T_k). □

The following lemma is useful towards proving properties of the composition of helped(T_i) for any decided transaction T_i:

Lemma 3.3. Suppose T_j is a member of the set returned by gpr(T_i, ts, obj_tails), then $T_i \neq T_j$ and $T_i \rightsquigarrow T_j$.

Proof. The procedure gpr returns the set pmvsg_trans built during execution of the procedure. The set is initially empty, and transactions are added at lines 9, 11, and 15. We show that the lemma holds at each addition to pmvsg_trans.

At line 9, the loop guard maintains both that $T_j \neq T_i$ and T_j has a read-write edge $T_i \rightarrow T_j$. According to Lemma 1.2, get-readers(T_j) returns only transactions T_k such that $T_j \rightsquigarrow T_k$. By the definition of get-readers only T_k with edges defined on the object x_j are returned. Since T_i does not read x_j, T_i is not returned by get-readers. Hence, the lemma holds for transactions added at line 9.

A transaction T_j added to pmvsg_trans at line 11 must have read the version x_i written by T_i by Lemma 1.2, which excludes T_i by definition and ensures $T_i \rightarrow T_j$ by a reads-from edge.

For any transaction T_j at line 15, $T_i \neq T_j$ and $T_i \rightarrow T_j$ along a write-read edge by construction of the loop guard. Further, as at line 9, by the definition of get-readers only T_k with paths $T_j \rightsquigarrow T_k$ defined on the object x_j are returned. In this case, we infer T_i does not read x_j because $x_i \ll x_j$, and x_i is enqueued only after T_i reads a version of x. The version x_k read by T_i must be before x_i such that $x_k \ll x_i \ll x_j$. Hence, the lemma also holds for transactions added at line 15, completing our discussion of the makeup of pmvsg_trans. □

Lemma 3.4. Suppose T_j is a member of the set returned by gcr+(T_i, ts, obj_tails, T_h), then $T_i \neq T_j$, $T_i \rightsquigarrow T_j$ and $T_i \in (\mathcal{C} \cup \{T_h\})$.

Proof. The procedure gcr+ returns the set cmvsg_trans built during execution of the procedure. The set is initially empty, and transactions are added at lines 11–12, 14–15, and 20–21. Each addition ensures the last claim of the lemma: $T_i \in (\mathcal{C} \cup \{T_h\})$. We show that the other claims of the lemma hold at each addition to cmvsg_trans.

At lines 11–12, the loop guard maintains both that $T_j \neq T_i$ and T_j has a read-write edge $T_i \rightarrow T_j$. According to Lemma 1.2, get-readers(T_j) returns only transactions T_k such that $T_j \rightsquigarrow T_k$. By the definition of get-readers only T_k with edges defined on the object x_j are returned. Since T_i does not read x_j, T_i is not returned by get-readers. Hence, the lemma holds for transactions added at lines 11–12.

A transaction T_j added to cmvsg_trans at lines 14–15 must have read the version x_i written by T_i by Lemma 1.2, which excludes T_i by definition and ensures $T_i \rightarrow T_j$ by a reads-from edge.

For any transaction T_j at lines 20–21, $T_i \neq T_j$ and $T_i \rightarrow T_j$ along a write-read edge by construction of the loop guard. Further, as at lines 11–12, by the definition of get-readers only T_k with paths $T_j \rightsquigarrow T_k$ defined on the object x_j are returned. In this case, we infer T_i does not

read x_j because $x_i \ll x_j$, and x_i is enqueued only after T_i reads a committed version of x. The committed version x_k read by T_i must be before x_i such that $x_k \ll x_i \ll x_j$. Hence, the lemma also holds for transactions added at lines 20–21, completing our discussion of the makeup of cmvsg_trans. □

The final lemma allows us to assert that if T_i is the earliest transaction in `helped`(T_p) and is not committed, then there exists a committed transaction T_j that cannot be decided before T_i begins, providing a bound on the number of transactions that can fulfill the role of T_j.

Lemma 3.5. Consider an execution of `hct`(T_p) that terminates, and let T_i be the earliest transaction in `helped`(T_p) according to the transaction priority order. Either T_i eventually commits, or if T_i is not aborted during Step 2 of `hct`(T_i), there is a committed transaction T_j such that $T_i \rightsquigarrow T_j$ in the PMVSG, and T_j commits after T_i begins execution.

Proof. The invocation of `hct`(T_i) implied by the definition of `helped` terminates because `hct`(T_p) terminates, which by Lemma 1.1 requires T_i to be decided. Lemma 3.5 trivially holds if T_i is committed; we consider the case where T_i is aborted. We first argue that T_i will not be aborted as a result of some thread executing `hdt`(T_i, T_k)—in other words, T_i may only be aborted during an execution of `hdt`(T_i, T_i).

For contradiction, we assume T_i is aborted during some execution of `hdt`(T_i, T_k). The successful abort implies `get-aborter`(T_i) = T_k. Some thread must then invoke `hct`(T_k) during Step 4 of the execution of `hct`(T_i)—implying $T_k \in$ `helped`(T_p)—since $T_i \neq T_k$ by construction. By the proof of Lemma 2.5, the invocation of `hct`(T_k) from `hct`(T_i) requires $T_k < T_i$, contradicting our assumption that T_i is the earliest transaction by the transaction priority order in the set `helped`(T_p).

We have established that, if T_i is aborted, it occurs during the execution of `hdt`(T_i, T_i). Consider the execution of `hdt` by the thread that successfully aborts T_i. The lemma excludes `hdt` from being invoked at Step 2 of `hct`(T_i). The procedure `hdt` is thus invoked from `dmr`(T_i), implying there is a cycle $T_i \rightsquigarrow T_i$ since `hdt` is invoked only on members of pmvsg_trans. The set pmvsg_trans is created by invocations of `gpr` over members of cmvsg_trans and T_i. The set cmvsg_trans is initialized by invoking `gpr`(T_i, ts, `obj_tails`). By Lemma 3.3, the transactions returned by `gpr` are reachable from the argument so that any member of the sets pmvsg_trans and cmvsg_trans are transitively reachable from T_i—the initial argument.

Reflexive edges are not defined in the PMVSG, implying some transaction T_j ($T_j \neq T_i$) belongs to the cycle containing T_i. Further, Lemma 3.3 implies that T_i is added to pmvsg_trans during an iteration of the loop over T_j in Step 4 since `gpr` cannot return T_i at Step 1. The loop guard states that transaction T_j must be a member of cmvsg_trans, and members of cmvsg_trans are committed by construction (see lines 6 & 13). Hence, if T_i is not aborted during Step 2 of `hct`(T_i), there is a committed transaction T_j such that $T_i \rightsquigarrow T_j$.

We have shown that some reachable, committed T_j exists. It remains to show that some transaction in the cycle does not commit until after T_i begins. Note that all transactions in the cycle other than T_i are members of cmvsg_trans and are thus committed by construction. We proceed by contradiction. Assume all transactions in the cycle (except T_j) commit before T_i begins.

procedure find-committed-transaction(T_p) **returns** $T \in \mathcal{C}$
1 **let** T_i := < *earliest trans. in* `helped` (T_p) >
2 **if** ($T_i \in \mathcal{C}$)
3 **return** T_i
4 **else if** (T_i aborted during Step 2 of `hct`)
5 **return** $T_j \in \mathcal{C} : (\exists T_m \mid r_i[x_m] \in T_i) \wedge (T_m \rightsquigarrow T_j) \wedge (T_j$ *decided after* T_i *begins*)
6 **else** // T_i *aborted from* `dmr`
7 **return** $T_j \in \mathcal{C} : (T_i \rightsquigarrow T_j) \wedge (T_j$ *decided after* T_i *begins*)

Figure 6: The definition of `fct` that converts an infinite sequence of attempted (and decided) transactions to an infinite sequence of committed transactions.

Consider the first edge $T_j \rightarrow T_k$ in the cycle. It is easy to see that this edge can be neither a reads-from nor a write-read edge. These edges require the existence of a version x_j in the history of some object x, created by T_j, that either coincides with or precedes the version read by T_k. If T_k is committed before T_j begins, x_j cannot exist while T_k executes, making such edges impossible.

The only edge left to consider is a read-write edge. For every read $r_j[x_l]$ of some object x during the execution of T_j, `is-readable`(T_l, trans_tstamp, T_j) returned true. We derive a contradiction using Lemma 1.4 to show that during the execution of `is-readable`(T_l, trans_-tstamp, T_j), transaction T_l, where $r_j[x_l]$ defines the edge $T_j \rightarrow T_k$ in the cycle, will be returned by some invocation of `gcr+` in Step 2. Thus, the procedure will return false at line 10. During the execution of `is-readable`, the conditions of Lemma 1.4 are met during our informal proof by induction as follows: *(i)* all transactions in the path are committed before T_j begins execution so that the path exists; *(ii)* all timestamps of committed transactions are no later than the current global trans_stamp; *(iv)* by construction all transactions in the path are committed except for T_j; and *(v)* all committed operations are enqueued before the tail is determined. Condition *(ii)* is met when we apply the Lemma repeatedly to each edge in the cycle. The first transaction in the cycle from T_j is returned at Step 1 according to Lemma 1.4 and is added to cmvsg_trans. Further transactions (including T_j) are likewise returned and added to cmvsg_trans in Step 2 as `gcr+` is in turn invoked on the previous transaction in the path. Since T_j is added to cmvsg_trans, the procedure returns false at line 10 in contradiction to the execution of T_j where all reads return true. Our assumption that all transactions are committed before T_j begins must be false so that Lemma 1.4 is not applicable to all edges in the cycle. One of the transactions in the cycle must have committed after T_j began execution.

□

We prove the protocol is lock-free by showing that our protocol guarantees an infinite sequence of transactions will commit (provided there is at least one correct thread).

Theorem 3. There exists an infinite sequence of committed transactions $\mathcal{T}^c$ provided there is at least one correct thread.

Proof. By Lemma 3.1, there exists an infinite sequence $\mathcal{T}^p$ of decided transactions such that for each transaction T_i in the sequence, `hct`(T_i) is executed by some correct thread.

We generate from $\mathcal{T}^p$ a (possibly disjoint) sequence of committed transactions $\mathcal{T}^c$ by applying to $\mathcal{T}^p$ the procedure `find-committed-transaction` (abbreviated `fct`) shown in Figure 6. According to the definition of `fct`, only committed transactions are returned.

Transactions T_i returned at line 3 and T_j returned at line 7 exist by Lemma 3.5. A transaction that maps to $T_i \in \mathcal{T}^c$ because of line 3 must belong to `benefactors`(T_i)—by Lemma 3.2, `benefactors`(T_i) is finite. A transaction that maps to $T_j \in \mathcal{T}^c$ because of line 7 is instead a benefactor of T_i such that $T_i \rightsquigarrow T_j$. As T_i must have begun before T_j commits, there exists only a finite number of such T_i corresponding to the transaction T_j returned. By Lemma 3.2, the number of transactions in `benefactors`(T_i) is finite for each such T_i.

Transaction T_j is returned at line 5 if T_i is aborted during Step 2 of `hct` because `is-readable`(T_m, ts, T_i) returns false for some transaction T_m such that $r_i[x_m] \in T_i$. We first use the definition of `is-readable` and Lemma 3.4 to show that T_j as defined in `fct` exists. The set cmvsg_trans contains only transactions returned by invocations of `gcr+` on other elements of cmvsg_trans and T_m. By Lemma 3.4, any transaction returned by `gcr+` is reachable from the argument. Hence, all members of cmvsg_trans are (transitively) reachable from T_m. Since all members of cmvsg_trans are committed by construction, it then suffices to show that T_m is not added to cmvsg_trans at Step 1 to show that some transaction T_j exists in Step 2 on which `gcr+` is invoked when T_m is added to cmvsg_trans. Lemma 3.4 states precisely that `gcr+`(T_m, ts, T_i) at Step 1 cannot return T_m. As a member of cmvsg_trans T_j is thus both committed and reachable from T_m in the PMVSG.

It remains to be shown that T_j is decided after T_i begins. Note that T_j is returned at line 5 if T_i is aborted during Step 2 of `hct` because `is-readable`(T_m, ts, T_i) returns false; whereas, `is-readable`(T_m, ts$'$, T_i) (ts$' \leq$ ts) returned true when T_i performed the read operation $r_i[x_m]$. Hence, during the execution of T_i there was no such transaction T_j that caused `is-readable` to return false. Timestamps are assigned before a transaction is committed, and ts$'$ was the global `trans_tstamp` at the time of the read, ensuring that no transaction that was decided before T_i began could meet the properties required of T_j. Yet, T_j exists as argued above, implying that T_j must have been committed after T_i began. As before, there are only a finite number of such T_j for each T_i so that the transaction T_j in $\mathcal{T}^c$ returned at line 5 can only be mapped by a finite number of transactions in $\mathcal{T}^p$.

□

9 Optimizations

The assumption of a wait-free queue implementation to manage objects' histories is problematic. Though it does not simplify greatly the consistency problem of providing 1-SR transactions, wait-free queues are required to guarantee termination. Wait-freedom is an expensive nonblocking access property. With some modifications, lock-free FIFO queues can maintain the lock-free progress guarantee, as shown in Figure 4.

A lock-free FIFO queue can provide the total order of versions in an object's history. We consider only enqueue operations and leave dequeue operations to discussions of garbage collection. A successful enqueue obviously poses no difficulties, but as stated in Section 4, the enqueue must occur only once during concurrent thread operations. On the other hand, a failure

```
// Commit transaction using lock-free data structures.
procedure commit-transaction-lf( T_i )
    // Step 1. Enqueue read operations
1   foreach {x : x ∈ O : r_i[x_k] ∈ T_i}
2       let L := w_k[x_k].read_list
3       do
4           list-insert-operation ( L, T_i )
5       until ( T_i ∈ L )
    // Step 2. Enqueue write operations
6   let S := < T_i >        // LIFO queue containing T_i
7   while ( S not empty )
8       let T_j := top of S
9       let T_q := help-enqueue-transaction ( T_j )
10      if ( T_q ≠ T_j )
11          push( S, T_q )
12      if ( ∃T_k | T_k ∈ S : T_k.ts ≠ ⊥ )
13          < pop stack until T_k removed >
14          help-commit-transaction ( T_k )
```

```
// Help transaction to enqueue operations
procedure help-enqueue-transaction( T_m )
1   foreach {x : x ∈ O : w_i[x_i] ∈ T_i}
2       let Q := x.hist
        // enqueue returns transaction corresponding to
        // successful enqueue.
3       let T_q := enqueue ( Q, w_i[x_i] )
4       if ( T_q ≠ T_i )
5           return T_q
    // Assign timestamp to transaction
6   CAS ( &(T_i.ts), ⊥, trans_tstamp )
7   CAS ( &(trans_tstamp), T_i.ts, T_i.ts+1 )
```

Figure 7: Procedures to commit a transaction using a lock-free data structures. The transaction read list is implemented by a lock-free ordered list. Object histories use a modified lock-free queue. Step 2 of `commit-transaction` needs to be modified to use the lock-free data structures, while Step 3 is removed entirely.

to enqueue a version occurs when another version is successfully enqueued instead. To use a lock-free instead of wait-free queue for histories, we guarantee that the enqueue operation terminates (not succeeds). The queue should return the successful competing transaction as an error code on failure, allowing our algorithm to act on the successful enqueue, rather than suffer possible infinite retries within the enqueue function. Hence, we would like a lock-free FIFO queue where the enqueue operation always terminates and returns the transaction corresponding to a successfully enqueued version. Error indication is then obtained by simply comparing the return value to the requested enqueue version.

We believe that the existing lock-free FIFO queue of Michael and Scott [15] can be modified as follows: *i)* if the enqueue operation succeeds (or has already succeeded), the transaction whose operation was enqueued is returned; and *ii)* if the enqueue fails, the transaction corresponding to the successful competing enqueue is returned. Michael and Scott's queue requires a simple modification to return the new tail if the CAS to append a new tail fails. Further, the queue can be checked to determine whether the version is already enqueued.

The new `commit-transaction` algorithm, called `commit-transaction-lf`, using lock-free queues appears in Figure 4 with modified Steps 2 and 3. Enqueueing a new version to an object's history uses a stack S to track transactions with successful enqueues. Transactions in the stack are helped with the method `help-enqueue-transaction` (abbreviated `het`) to guarantee that some transaction will enqueue all of its operations. The remaining functions `hct` and `dmr` used to commit a transaction are unmodified. Note that without contention, the modified algorithm behaves exactly as the original presented in Figure 4. Specifically, without

contention the invocation of `het` will return T_i and set the timestamp of T_i, ensuring that T_i is removed from S at line 13 and the stack is emptied. At line 14, `hct`(T_i) will then be invoked. Since the stack was cleared, the loop terminates, and the function ends. The modifications apply only when `het`(T_j) returns a transaction $T_q \neq T_j$ (where $T_j = T_i$ initially); that is, when an operation of T_j is not successfully added to a history during `het`.

The proof of one-copy serializability (Theorem 1) for the modified algorithm needs only one change, though the proof applies only to committed transactions, which have the same properties as before. The modified algorithm does not change the functions `hct` and `dmr` that are used in the proof of 1-SR. Instead, several lemmas and the theorem itself implicitly assume that a transaction with a timestamp is necessarily enqueued. This assumption still holds for the modified algorithm because the guard at line 12 ensures that a timestamp is assigned to T_i before invoking `hct`(T_i).

The proof of Theorem 2 no longer holds. The new algorithm for `commit-transaction` is not guaranteed to terminate. Termination is not our concern (though it may be for some applications); lock-freedom is our goal. Though we cannot use the proof of termination for Lemma 3.1, Lemmas 2.1 through 2.5 still hold as neither `dmr` or `hct` have been modified. In fact, Theorem 2 is the only proof that no longer holds from the previous Sections 6 through 8.

Without relying on termination, the following lemma holds that an infinite sequence of transactions exists under the modified algorithm. The previous proof of lock-freedom in Theorem 3 can be satisfied with the same definition of `fct` used in Figure 6 by replacing Lemma 3.1 with the following Lemma 4.1.

Lemma 4.1 requires a further assumption. Previously, it was sufficient to assume that at any time there were only a finite number of threads in the system. We now assume a stronger condition that we call the *commit-rate assumption*—the rate of new threads in the system is less than the rate of commit of transactions. This assumption is easy to enforce by requiring new threads to be created within a transaction. Relaxing the wait-free access guarantee of object histories requires directly addressing the starvation that may occur while attempting to add a version to an object history. By bounding the rate at which new threads are created, we prohibit starvation due to an overwhelming number of new threads such that each thread successfully enqueues a new operation but does not have a chance to help another thread.

Lemma 4.1. If there is at least one correct thread, under the modified algorithm of Figure 4 there exists an infinite sequence of decided transactions $\mathcal{T}^p$ such that for each transaction T_i in the sequence, some thread executes `hct`(T_i).

Proof. In this proof, we abbreviate `commit-transaction-lf` with `ct-lf`. We prove that in a finite number of steps *i)* a thread will always invoke `hct`(T_i) for an undecided transaction T_i, and *ii)* T_i will be decided.

Suppose that a correct thread p attempts to commit a transaction T_p. The function `ct-lf` terminates when the stack S is empty, exiting the loop in Step 2 of the function. The stack initially contains T_p, requiring T_p to be removed at line 13 to empty S. The following line of the function then invokes `hct`(T_p) in this case. The commit assumption provides that p will attempt to commit another transaction again in a finite number of steps. Hence, the lemma holds as

long as ct-lf terminates. In our modified algorithm, the execution of ct-lf(T_p) by a correct thread p is not guaranteed to terminate. We show that if the function does not terminate, a new transaction will always be decided by hct in a finite number of steps, considering the execution of ct-lf(T_p) by p and provided no other thread commits a new transaction. If another thread commits a transaction, the Lemma holds because hct must have been invoked to commit the transaction.

During different iterations of the loop in Step 2, the stack S contains different sequences of transactions $T_p, \ldots, T_n$. We prove that if another thread doesn't commit a transaction, in a finite number of steps some transaction T_k in the sequence will have a timestamp assigned, enabling the guard at line 12. At the guard, S contains the sequence $T_p, \ldots, T_k, \ldots, T_n$ (T_n not necessarily distinct from T_k), and at line 14 thread p invokes hct(T_k), providing $T_k \in \mathcal{T}^p$. The execution of hct(T_k) by p terminates in a finite number of steps (by Lemma 2.5) with T_k decided (by Lemma 1.1).

We first show that T_k is not added to the stack after a timestamp is assigned (though T_k may be concurrently added to several threads' stacks in ct-lf), eventually generating a new member of $\mathcal{T}^p$ rather than trying repeatedly to commit the same set of transactions. When the guard at line 12 is true, some T_k in the sequence $T_p, \ldots, T_n$ is assigned a timestamp, and at line 13 $T_k, \ldots, T_n$ is then removed from the stack. Transaction T_k is assigned the timestamp upon successful completion of het(T_k) after all of the operations of T_k are enqueued. Hence, no further invocations of het by any thread will return T_k because an invocation of het(T_i) for some T_i returns T_k only when an operation of T_k is enqueued concurrently.

Finally, it remains to be shown that the guard at line 12 becomes true—that some T_i in S is assigned a timestamp—in a finite number of steps if ct-lf does not terminate, and some other thread (besides p) does not commit a transaction. First, we argue that there is a bound on the length of S, then use the bound to show that either a transaction commits (not necessarily in S) or some transaction in S is assigned a timestamp. We note that transactions privately aborted can never appear in S because their operations are not enqueued and thus cannot be returned by the enqueue procedure. The guard at line 12 is false when T_i.ts $= \perp$ for every T_i in S. A transaction cannot be decided until T_i.ts $\neq \perp$ because the procedures hct and hdt required to commit or abort transaction T_i are only invoked on enqueued transactions. Hence, all transactions in the stack must be undecided. Since there are only a finite number of transactions that may be undecided at any time, the stack cannot grow longer than the finite number of threads in the system.

If the number of threads is fixed, then when S reaches its maximum length, het only returns transactions already in S, implying another operation is enqueued for a transaction in S. Since there are a finite number of operations belonging to transactions in S, eventually some transaction will have all of its operations enqueued and will be assigned a timestamp either by the thread that successfully enqueues the last operation or by the correct thread p executing het(T_n) on the last transaction T_n in S. Hence, in a finite number of steps the guard at line 12 is enabled if the number of threads is fixed.

The number of threads is not fixed. The bound on the length of S thus may grow as the number of threads are increased, although the commit-rate assumption holds that new threads

are created slower than transactions are committed. Hence, unless a new transaction commits (increasing the number of committed transactions), the number of threads is fixed.

□

10 Conclusions

Serializability is a correctness criterion that allows for concurrent execution of transactions, while ensuring the objects' values are equivalent to some sequential execution of the transactions. We have presented a one-copy serializable, lock-free software transactional memory (STM) based on multiversion objects. Our algorithm concurrently executes and commits transactions without mutual exclusion locks. Only existing lock-free data structures and atomic primitives are required to implement the algorithm. Multiple versions allow read-only transactions to read previous data values, increasing the likelihood that they will commit under contention. Threads help commit transactions to provide progress and to ensure the one-copy serializability property. The proof of 1-SR shows how helping transactions that are reachable in the PMVSG guarantees one-copy serializability. The proof of lock-freedom instead shows how helping transactions can provide lock-free progress, although some transactions might be aborted to guarantee one-copy serializability. In the future, we intend to assess the performance of our algorithm through an implementation.

References

[1] J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM TOCS*, 15(2):134–165, May 1997.

[2] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, 1983.

[3] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, September 2003.

[4] T. Harris. Exceptions and side-effects in atomic blocks. In *PODC Work. on Concurrency and Synch. in Java Prog.*, July 2004.

[5] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003.

[6] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. *LNCS*, 2180:300–314, Oct 2001.

[7] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1):124–149, 1991.

[8] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proc. IEEE ICDCS*, pages 522–529, 2003.

[9] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *Proc. ACM PODC*, pages 92–101, July 2003.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[11] C. Hoare and R. H. Perrott, editors. *Towards a theory of parallel programming*, volume 9 of *Operating Systems Techniques*, pages 61–71. Academic Press, 1972.

[12] A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proc. ACM PODC*, pages 151–160. ACM Press, 1994.

[13] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), June 1981.

[14] V. J. Marathe and M. L. Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Department of Computer Science, University of Rochester, June 2004.

[15] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. ACM PODC*, pages 267–275, 1996.

[16] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[17] F. Pizlo, M. Prochazka, S. Jagannathan, and J. Vitek. Transactional lock-free objects for real-time java. In *PODC Workshop on Concurrency and Synchronization in Java Programs*, July 2004.

[18] N. Shavit and D. Touitou. Software transactional memory. In *Proc. ACM PODC*, pages 204–213, 1995.

[19] A. Welc, S. Jagannathan, and A. L. Hosking. Transactional monitors for concurrent objects. In *Proc. ECOOP*, June 2004.