A Scalable Reference Counting Garbage Collector

Yossi Levanoni * Erez Petrank †

November 14, 1999

Abstract

We study concurrent garbage collection via reference counting. While tracing variants of garbage collection have been well studied with respect to concurrency, the study of reference counting has been somewhat behind. The straightforward concurrent version of reference counting is not at all scalable. Furthermore, a more advanced study by DeTreville yielded an algorithm which acquires a single lock per update of a pointer, thus, executing all updates sequentially and hindering the scalability of the algorithm.

In this paper we propose a new concurrent reference counting algorithm with several desired properties. First, the algorithm employs extremely fine synchronization. In particular, updates of pointers and creation of objects require no synchronization overhead whatsoever (even not a compare-and-swap type of operation). Furthermore, the algorithm is non-disruptive: the program threads are never stopped simultaneously to cooperate with the collector. Instead, each program thread cooperates with the collector at its own pace, infrequently, and for very short periods. Thus, the program can run with (almost) no synchronization overhead imposed by the collection.

Keywords: Runtime systems, Memory management, Garbage collection, Reference counting.

1 Introduction

Automatic memory management is well acknowledged as an important tool for a fast development of large reliable software. It turns out that the garbage collection process has an important impact on the overall runtime performance. The amount of time it takes to handle allocation and reclamation of memory spaces may reach as high as 30% of the overall running time for realistic benchmarks; in particular garbage collection may take a long time if the memory management is not well designed. Thus, a clever design of efficient memory management and garbage collector is an important goal in today’s technology.

1.1 Automatic memory management on a multiprocessor

In this work, we concentrate on garbage collection for multiprocessor machines. Multiprocessor platforms have become quite standard for server machines and are also beginning to gain popularity as high performance desktop machines. Many well studied garbage collection algorithms are not suitable to work with a multiprocessor. In particular, many collectors† run on a single thread among them the collector supplied with JavaSoft’s Java Virtual Machine.
after all program threads have all been stopped. This causes bad processor utilization, and hinders scalability.

In order to make better use of a multiprocessor, concurrent collectors have been presented and studied (see for example, [5, 16, 3, 13, 14, 8, 18, 37, 17])\(^2\). A concurrent collector is a collector that does most of its collection work concurrently with the program without stopping the program threads. Most of the concurrent collectors need to stop all program threads at some point during the collection, in order to initiate and/or finish the collection, but the time the mutators must be in a halt is short. Usually the pause time is negligible compared to the time it takes to execute the full collection cycle.

Stopping all the threads for the collection is an expensive operation by itself. Usually, the program threads cannot be stopped at any point. Rather, they should be stopped at safe points at which the collector can safely determine the reachability graph and properly reclaim unreachable objects. Thus, each thread must wait until all the last of all threads cooperate and come to a halt. This hinders the scalability of the system, as the more threads there are the more delay the system suffers. Furthermore, if the collector is not running in parallel (which is usually the case), then during the time the program threads are stopped, only one of the processors is utilized.

Therefore, it is advantageous to use on-the-fly collectors [16, 18, 17]. On-the-fly collectors never stop the program threads simultaneously. Instead, each thread cooperates with the collector at its own pace through a mechanism called (soft) handshakes.

We remark that another alternative for an adequate garbage collection on a multiprocessor is to perform the collection in parallel (see for example [25, 12, 35, 26, 19, 30]). We do not explore this avenue further in this work.

1.2 Reference counting

Reference counting is a most intuitive method for automatic storage management. As such, systems using reference counting were implemented starting from the sixties (c.f. [11].) The main idea is that we keep for each object a count of the number of references that reference the object. When this number becomes zero for an object \(o\), we know that \(o\) can be reclaimed. At that point, \(o\) is added to the free list and the counter of all its predecessors (i.e., the objects that are referenced by the object \(o\)) are decremented, initiating perhaps more reclaimations.

The key advantage of reference counting for traditional uniprocessor is that it operates in a decentralized manner, allowing the mutator to recycle an object as soon as it becomes garbage. Since reference counting is local and decentralized in nature there is no pause time incurred in order to compute global features, such as object-graph reachability. Such computations are necessary in tracing collectors such as mark-and-sweep and copying collectors. Its disadvantages are a per-object space overhead required to maintain the reference count of an object, a computational overhead associated with pointer manipulation in order to maintain the reference count invariant and the inability to reclaim cyclic data-structures.

The space overhead issue is ameliorated by the fact that a two-bit heap reference count field is more than enough in the striking majority of cases (c.f. [15, 39, 49, 9, 24]). While the computational overhead is reduced by some 80\% using Dutsch and Bobrow's Deferred Reference Counting [15]. Only the inability to reclaim cyclic structures does not have a satisfactory solution intrinsic to reference counting [48, 29] and therefore reference counting systems are usually combined with a tracing collector. Usually, a simple tracing collector is used infrequently to reclaim cyclic unreachable structures.

\(^2\)Historically, concurrent collectors were invented to ameliorate the long pauses caused by garbage collection, but incremental and concurrent collectors were found appropriate and were quickly adopted for use on multiprocessors.
The transformation of sequential reference counting into concurrent reference counting must cope with maintaining the reference counting invariant. A straightforward adaptation of the sequential algorithm to a concurrent environment imposes a non-tolerable synchronization overhead on pointer update operations: both pointer update and reference counters updates must be atomic. DeTreville describes in [13] a concurrent reference counting garbage collection algorithm used for a Modula-2+ system. This is the only concurrent reference counting system that works on a stock SMP. The scheme used is an adaptation of Deutsch and Bobrow’s algorithm to an SMP environment. The system achieves a certain amount of parallelism, notably, threads are not required to stop simultaneously. Rather, when needed, each thread is stopped at a time. Thus, the algorithm is an on-the-fly algorithm, as above. However, each update of a pointer is done in a critical section common to all threads, no matter which pointer slot is modified. This solution is obviously not scalable since at most a single update can occur in the system at any given moment. This synchronization overhead is unacceptable on a multiprocessor.

1.3 This work

In this work we propose a new scalable and efficient concurrent reference counting algorithm. Our algorithm employs extremely fine synchronization. In particular, updates of pointers and creation of objects require no synchronization overhead whatsoever (even not a compare-and-swap type of operation). Furthermore, the algorithm is non-disruptive: the program threads are never stopped simultaneously to cooperate with the collector. Instead, each program thread cooperates with the collector at its own pace, infrequently, and for very short periods.\(^3\)

Our central goal is achieving the shortest possible response time for typical mutator requests such as object allocation and pointer manipulation.

1.3.1 The Snapshot Algorithm

We start with a simple algorithm denoted \textit{The Snapshot Algorithm}. In this algorithm, there is a point in time in the beginning of the collection in which all mutators are halted. A virtual snapshot of the heap is taken then and used for the collection. Of course, taking a real snapshot is too expensive both in time and space. It turns out that what we really need is to find which reference fields have been modified since the last snapshot. For each modified field, we need to know the value in the previous snapshot and in the current snapshot since we must decrement the reference count of the previously referenced objects and increment the reference count of the newly referenced object. To help with this goal, the mutators record the first time a pointer field is modified after the snapshot. This is the information that is really recorded, and it is enough information to perform the collection. The exact details are given in Section 3 below.

At first glance, it seems that a race condition may foil the correctness of this process: two mutators may write to the same location and record conflicting values. However, with a careful design of the write barrier code performed by the mutators while updating, this can be solved. The main idea is that even if two mutators think they are performing the first modification after the snapshot, they will properly record the same value into their records causing no inconsistencies. An additional important idea to make the algorithm efficient and scalable is to use local buffering for the records. The details are in Section 3 below. The time the mutators are simultaneously

\(^3\)We remark that the simple version of reference counting seems non-disruptive at first glance: there is no collector thread that stops the mutators. However, the work of the collection is done by the mutators, thus delaying (disrupting) the program’s actual work; furthermore, running this algorithm naively on a multiprocessor requires heavy synchronization on each update of a pointer, thus, making the algorithm non-scalable.
stopped is short: all is needed is to read all the local buffers, and mark that all reference fields are untouched for the current new snapshot.

Note that as in DeTreville’s work in [13], our algorithm is based on the mutators logging information about the modifications they apply to heap references. However, in our algorithm, a thread takes a record of a modification at most once per slot per cycle (as opposed to always keeping a record) and there is no synchronization incurred due to the logging action.

The Snapshot Algorithm is a reasonable candidate for a scalable reference count garbage collector: it requires little cooperation of the mutators (there is no need for synchronization operations such as compare-and-swap) plus one halt of the program for a fast initialization of the collection. However, the fact that the mutators are stopped simultaneously and that they must wait till the collector handles all mutators is not satisfactory. This may still hinder the scalability of the system. Thus, we propose a more advanced collector denoted The Sliding View Algorithm that achieves better efficiency and scalability.

1.3.2 The Sliding View Algorithm

Our proposed Sliding View Algorithm has low synchronization requirements just like the Snapshot Algorithm. Namely, mutators never compete on locks or use strong memory operations, such as Compare & Swap. But in the Sliding View Algorithm, the mutators are never stopped simultaneously. Instead, in each collection cycle, they cooperate with the collector through four handshakes. In these handshakes, each mutator is stopped for a short while (for example, until its buffers are read by the collector) and then resumes. Since in this algorithm there is no specific time in which all mutators are stopped, an an overall different approach to using a snapshot is called for. To this end, the sliding view concept was developed.

All previous reference-counting algorithms are based on the same strict invariant: there is a time point $t$ such that the reference-count field of each object is equal to (or not smaller than, in some limited reference-count field variants) the actual number of references to the object. This invariant requires some form of an atomic snapshot, referring to time $t$. In the sliding view algorithm we maintain a weaker invariant which still allows for safe and efficient garbage collection. In the algorithm, we interchange the notion of an atomic snapshot with that of a sliding view which is, as its name implies, a fuzzier picture of the heap state. In a sliding view, each reference field in the heap can be checked at a different time. However, between the time the first reference field is read and the time the last reference field is read, an extreme care is taken with any reference modification. In particular, objects that are referenced by pointers that are modified during this time will not be collected in this collection cycle.

Like in the Snapshot Algorithm, The sliding view algorithm considers only the differences between the sliding view of the current collection and the sliding view of the previous collection. Thus, the sliding views are never computed explicitly but are rather inferred from the records mutators keep in local history buffers.

1.3.3 The Tracing Sliding View Algorithm

In the reference counting algorithms that we propose there still remains the inability to reclaim cyclic structures and restore stuck reference counts. In our opinion the only realistic way to tackle these problems is by combining frequent reference counting cycles with infrequent tracing cycles. Yet we are not willing to sacrifice scalability and impose additional overhead on mutator’s write-barrier in order to support both paradigms. Therefore, we present a scalable tracing collector that uses the same write-barrier that is used by the reference counting collector. The tracing and
reference counting collectors are thus interchangeable and any of which may be invoked on each cycle.

1.4 Efficiency of the Sliding View Algorithm

The complexity of the write-barrier on pointer modification amounts to three additional load operations in most cases and to a handful of additional memory accesses otherwise. The cooperation through handshakes is proportional to the size of thread’s state. In this respect, our algorithm has synchronization characteristics similar to those of [18, 17].

However, opposed to tracing collectors, the amount of work the collector has to invest in a cycle is not proportional to the volume of live data, nor to the size of the heap, but is rather dominated by the number of slots that have been modified since the last cycle plus the amount of garbage that is recovered. Thus, any mutator operation incurs a close to (small) constant overhead, bearing in mind both operations carried out directly, by the mutator, and indirectly, by the collector. We thus expect the algorithm to demonstrate scalability in both the size of the heap and the number of mutator threads.

1.5 Related work

The traditional method of reference counting, applicable in the realm of uniprocessing, was first developed for Lisp by Collins [11]. In its simplest form, it allowed immediate reclamation of garbage in a localized manner, yet with a notable overhead for maintaining the space and semantics of the counters. As such, it was used in applications that could not tolerate delays yet could stand the incurred overhead such as Smalltalk-80 [23] and the AWK [1] and Perl [45] programs.

Weizman showed in [46] how the delay introduced by recursive deletion (which is the only non-constant delay caused by classic reference counting) can be ameliorated by distributing deletion over object creation operations.

Deutsch and Bobrow [15] eliminated most of the computational overhead required to adjust reference counters in their method of deferred reference counting. According to the method, local references are not counted thus the need to track fetches, local pointer duplication and cancellation are deemed unnecessary. Only stores into the heap need be tracked. However, the immediacy of reference counting is lost in a certain extent, since garbage may be reclaimed only after the mutator state is scanned. Nevertheless, the method proved to be very efficient and was later adapted for Modula-2+ [13]. Several studies [36, 43, 2, 54] showed that the assumption about the relatively low frequency of store operations is usually valid. Baker in [6] advocates for a less sweeping treatment of local variables: deferring the manipulation of reference counters and reclamation of objects is controlled by pointing to them using special anchored pointers. Baker claims that the Deutsch and Bobrow technique is not feasible with modern compilers since it is difficult to scan the stack for pointers. However, the stack scan can be done conservatively with no difficulty involved. Park and Goldbreg [38] show how one can detect scopes in which it is known at compile time that an object is “anchored” and accordingly eliminate reference count manipulations due to stack operations.

Addressing the issue of storage overhead and noting that most objects are singly-threaded, except for the duration of short transitions, Wise and Roth [39, 50] suggested using a single bit for the reference count and an auxiliary cache for objects which momentarily have a reference count of two. It is further claimed that this uniqueness bit should reside in any pointer to the object rather than in the object itself, thus saving extraneous memory accesses. This idea was introduced by Stoye [42]. Additional schemes that use single-bit reference counters are those by Chikayama and Kimura [9] and by Goto et al. [24].
DeTreville describes in [13] a concurrent multiprocessor reference counting collector for Modula-2+. The algorithm used adapts Deutsch and Bobrow's ideas of deferred reference counting and transaction log for a multiprocessor system. However, the update operation is done inside a critical section that uses a single central lock. This implies that only a single update can occur simultaneously in the system, placing a hard bound on the scalability of it.

Our algorithms are based on the sliding view notion, which is semantically close to a snapshot. The sliding views are used to compute reference counts, on which the collection criteria is based. Additionally, we present a tracing collector that traces according to a sliding view. Thus, our algorithms have points of similarity with other concurrent algorithms which are snapshot based. Furusou et al. [21] presents a collector based on copy-on-write facilities of the operating system. This mechanism is used in order to obtain an atomic snapshot of the heap. Tracing proceeds according to this atomic snapshot. Yuasa [53] uses an implicit snapshot obtained by a software write-barrier that records the values of slots before they are overwritten. These “old” values, a superset of the values that were in effect when the conceptual snapshot was taken, are then traced and retained by the collector.

In the context of incremental tracing collectors Wilson [48] makes the distinction between snapshot-at-beginning and incremental update algorithms. Trying to apply the terms to our concurrent reference counting collector we note that our algorithm takes both approaches simultaneously. The inter-cycle reference counting activity is based on spotting differences between consecutive sliding views. Thus, the system strives to retain the information that is contained in the most recent sliding view, which is similar to the pattern of operation in a snapshot-at-beginning algorithm. On the other hand, intra-cycle activity is centered at linking the sliding view to an eventual atomic state of the system, based on which collection decisions are made. This linking is done using incremental update techniques.

In terms of synchronization requirements and characteristics our work is similar to that of Doligez-Leroy-Gonthier [18, 17]: 1) we never require a full halt of the system; 2) mutators are required to cooperate four times per cycle ([18, 17] requires three handshakes per cycle); 3) no locks are used. In our tracing algorithm we have used an object sweeping method similar to that presented in [18, 17].

1.6 Organization

In section 2 we present definitions and terminology to be used in the rest of the paper. In section 3 we present our Snapshot algorithm. Sec 4 describes the Sliding View algorithm. In section 5 we discuss several implementation issues of the proposed algorithms. Section 6 introduces our Tracing Sliding View algorithm. We provide proofs for the Snapshot Algorithm, Reference Counting Sliding View Algorithm and Tracing Sliding View Algorithm in sections A, B and C of the appendix, respectively. We conclude in section 7.

2 System Model, Definitions

Memory management. User programs assume the existence of system level services encapsulated in the Memory Manager and Garbage Collector subsystems. The role of the memory manager is to provide the application program, upon request, with contiguous regions of the memory, called objects. The memory-manager is also responsible for the explicit deletion of objects. A chunk of memory which has been returned by the memory manager to the application program but has not
yet been deleted is an *allocated object*. The task of the garbage collector is to find objects which are unreachable (see definition below) and pass them to the memory-manager for deletion.

In a concurrent system it is convenient to perceive (and usually also to implement) the garbage collector as a separate thread. Then the garbage collector dedicated thread is termed the collector while the ordinary threads that carry out the user program are called *mutators*. We sometimes call the mutators *user threads* or just *threads*.

**The Heap: Objects and Roots.** Some of the memory locations inside an object are designated as pointer-containers, i.e., they assume the value of addresses of objects, or the special value *null*. We call such locations *heap-slots* or just slots. This name stresses the fact that heap slots are residing inside the heap, as opposed to global roots and local references (defined below), which are not part of the heap. It is a common requirement, that we adopt as well, that all object’s heap-slots would contain *null* upon allocation.

The system contains *global roots* which are a set of fixed memory locations, disjoint of the heap, that may be accessed, for reading and writing, directly by any thread.

Each thread has a local state which can contain references to objects. These references are termed *local roots* or *local references*. On a typical system, a thread local state is comprised of thread specific registers and stack. Only the thread itself can access its local state.

**Simplifying assumptions regarding the heap.** For convenience, we assume in the exposition of the algorithms and their proofs that there are no global roots. In section 5.3 we show how global roots should be actually treated on a real system. For now, let us just say that global roots may be simply treated as ordinary heap slots.

In the correctness proofs we adopt the assumption that objects contain only reference fields, i.e., they never contain non-pointer fields. It can be readily seen that our algorithms operate correctly when this is not the case.

**Global state and time.** All shared-memory operations requested by all threads (i.e., both mutators and the collector) during a run are interleaved into a single linear order by the shared-memory system. This assumption allows us to conveniently define global state and time as follows:

**Definition 2.1 (Time)** For a given execution, we say that a shared-memory operation occurs at time $t$ if it is operation number $t$ in the linear sequence of operations corresponding to the execution.

**Definition 2.2 (State)** For any expression $E$ which depends only on the values of shared-memory locations and for any time point $t$ in the execution, we denote by $E@t$ the value of entity $E$ at time $t$, i.e., $E@t$ is the value of $E$ just prior to the execution of instruction number $t$.

Finally, we define the address-space of a given execution $E$, denoted by $Mem(E)$, to be the set of all memory locations which are addressed by the instructions of $E$.

**Reachability.** A thread can access an object only if it has a local reference to it. A thread can obtain a reference to an object only by one of two methods: (1) by reading the contents of a slot of an object to which it already has a local reference. (2) by allocating a new object. This pattern of access calls for the following standard definition of reachability.

**Definition 2.3 (Reachability)** We say that an object $o$ is

---

4We assume that memory manager allocation and deletion operations are atomic, i.e., an object cannot be allocated if it has not been fully deleted, etc.

5Thus, we assume that the shared-memory is sequentially consistent. In section 5.4 we show how the memory model constraints may be relaxed in order to adapt the algorithms we present to systems with weaker memory models.
- directly reachable from thread $T_i$ at time $t$ if $T_i$ has a local-reference to $o$ at $t$.
- reachable from thread $T_i$ at time $t$ if it is directly reachable from thread $T_i$ at $t$ or there exists a reference to $o$ in object $y$ at time $t$ and $y$ is reachable from thread $T_i$ at time $t$.
- reachable at time $t$ if there exists a thread $T_i$ such that $o$ is reachable from $T_i$ at time $t$.
- unreachable, or garbage, at time $t$ if it is not reachable at time $t$.

Reference counters. Garbage collection by reference counting is based upon counting the number of references referring to each object at a given time. We formally define the reference count of an object as follows:

**Definition 2.4 (Heap Reference Count)** The Heap Reference Count of an object $o$ at time $t$, denoted by $RC(o)@t$, is the number of heap slots referring to $o$ at time $t$.

We usually abbreviate and refer to an object Heap Reference Count as its *Reference Count*. There is a field associated with each object that is used to record the number of references to the object. For an object $o$ this field is denoted by $arc$. The field is invisible to the user program; it is only accessible to the memory management subsystem.

Coordination of threads. We assume that the garbage collector thread, by virtue of being a privileged system thread, can control scheduling of mutator threads to a certain extent. Specifically, the collector may suspend and subsequently resume user threads. When a thread is suspended, the collector may inspect and change its local state with the effects taking place after the thread is resumed.

Each thread’s code is comprised of protected and unprotected code. When a thread is executing unprotected code the collector may suspend it. Suspension of a thread means that no instructions on its behalf are scheduled, up to the time it is resumed. In our algorithm, the only pieces of code which are protected are procedures Update and New, which are in charge of updating heap-slots and allocating new objects, respectively.

The following pseudo-code:

1. suspend thread $T_i$
2. **Do-Something**
3. resume thread $T_i$

when executed by the collector, means that the collector waits until thread $T_i$ is not executing protected code, then it suspends it, executes the code in **Do-Something** and then it resumes the thread. When referring to such a construct and stating that $T_i$ was suspended at time $t$ it is meant that at time $t$ the first instruction of **Do-Something** was scheduled. Accordingly, we say that $T_i$ was resumed at time $t$ if the last instruction of **Do-Something** was scheduled at time $t - 1$.

A *Hard Handshake* is a collector code construct of the form:

1. for each thread $T_i$ do
2. suspend thread $T_i$
3. **Do-Something**
4. for each thread $T_i$ do
5. resume thread $T_i$

---

An object Reference Count is sometimes defined as the number of references (including local references) to an object. We do not include local roots in the count. This definition is the same as presented in the context of Deferred Reference Counting, see [15].
Which means that all user threads are halted in unprotected code when \texttt{Do-Something} is executed. A hard handshake is usually a costly operation whose execution ties up the entire system for a time duration that depends on the number of threads and on the complexity of the \texttt{Do-Something} operation.

A \textit{Soft Handshake} is much more scalable. It is a collector code construct of the form:

1. for each thread $T_i$ do
2. suspend thread $T_i$
3. \texttt{Do-Something-Related-To-$T_i$}
4. resume thread $T_i$

In a soft handshake, at most one thread is halted in each moment and an operation related to it is executed. This construct is useful for specifying transactions in which a mutator and the collector exchange data. We note that the soft handshake mechanism is equivalent to the handshake mechanism described in [18, 17], where mutators voluntarily cooperate in order to complete transactions with the collector. We chose this style of cooperation construct in order to facilitate the exposition of the algorithm: using our approach all actions are seemingly carried out by the collector.

3 The Snapshot Algorithm

In this section we introduce our first algorithm, which is based on computing differences between heap snapshots. We first present a naive algorithm that demonstrates the idea behind the snapshot algorithm, then we present the snapshot algorithm itself. Correctness proof is given in appendix A.

3.1 A naive algorithm based on snapshot difference

The algorithm operates in cycles; we are describing collector actions during cycle $k$ (throughout the paper we let the subscript $k$ denote the number of a garbage collection cycle.) To start a cycle, the collector stops all threads. While the world is stopped, the collector makes a replica of the heap, denoted $R_k$. Additionally, it marks \textit{local} any object which is directly reachable. Then, it resumes the threads.

Note that since no mutator is running during the time the replica is constructed, $R_k$ is an \textit{atomic snapshot} of the heap. The collector then adjusts $rc$ fields due to differences between $R_k$ and the replica of the previous cycle, $R_{k-1}$. Specifically, the collector considers any slot $s$ whose value in $R_k$ differs from that in $R_{k-1}$ and:

1. increments the \textit{rc} field of the object referred to by $s$ in $R_k$
2. decrements the \textit{rc} field of the object referred to by $s$ in $R_{k-1}$.

It is easy to verify, by induction on the cycle number and assuming that each object is allocated with zeroed-out \textit{rc} field, that for any object $o$, at the time the collector completes adjusting \textit{rc} fields, $o.rc$ equals $o$’s heap reference count at the time the snapshot was taken. Thus, any object $o$ which has $o.rc = 0$ after adjusting is done and which is not marked \textit{local} has no references to it whatsoever in the system and may be reclaimed.

3.2 Implementing the algorithm efficiently

Implementing the algorithm efficiently entails two major problems:
• **efficiently finding differences between heap snapshots.** Of course, it is not practical to make a copy of the heap. We are only interested in those portions of the heap that have changed since the last collection. We need a method to efficiently spot these differences.

• **efficiently finding garbage objects.** We need an efficient method (other than examining all heap objects) to find all those objects with a zero rc field which are not marked local.

The latter problem is conveniently solved using a Zero Count Table \[15\] which records any object whose reference count field drops to zero. In particular, objects are inserted into a thread specific ZCT as they are created since upon their creation they have a zero heap reference count. These local ZCTs are merged into a global ZCT. The global ZCT contains survivals from the previous cycle as well, i.e., objects that at the end of the previous cycle had zero rc field but were marked local.

We now turn our attention to the former problem. Taking a snapshot of the entire heap when all mutators are stopped is not practical: it requires too much space and time. In our algorithm, these snapshots are only conceptual: they are never computed in full. Instead, we require the mutators themselves to record slots’ values as they are about to modify them. Using this recorded information the collector can tell what was a modified slot’s value in the last conceptual snapshot, i.e., in \(R_{k-1}\).

It remains for the collector to find out what is such a slot value in \(R_k\). Trivially, the collector can read the slot while all mutators are stopped. This simple solution is not scalable, however, since it implies that for each changed slot there will be a time slice in which the entire system will be tied-up attending to its update, jeopardizing the parallelism promised by the presence of multiple processors. Thus, we require that the collector would find the value of such a slot in \(R_k\) **while the mutators are running.** This is done using an arbitration mechanism using which the collector tries to determine a slot. The mechanism reliably reports success or failure. In case of success, the value is immediately revealed to the collector and the collector is guaranteed that no thread has changed the slot since the conceptual snapshot \(R_k\) was taken. Otherwise, when the collector fails determining a slot, it is guaranteed that some thread has already kept a record of the slot along with its value in \(R_k\). The collector therefore looks up the threads’ records and finds the desired information.

This mechanism is implemented in the following manner: every slot \(s\) has a unique dirty flag associated with it denoted \(\text{Dirty}(s)\). This flag signifies whether the slot has been overwritten since the last conceptual snapshot. The dirty flags are then manipulated using these patterns of operation:

- all dirty flags are cleared on each cycle, when all mutators are stopped.
- in order to modify a slot \(s\) a thread takes these actions, that comprise its write barrier: 1) it reads the contents of \(s\). Let \(v\) stand for the value it has fetched 2) it reads \(\text{Dirty}(s)\) 3) if \(\text{Dirty}(s)\) is off it saves a record of the pair \((s,v)\) stating that \(v\) was the contents of \(s\) in the most recent conceptual snapshot and then it raises the flag 4) now the store proper occurs.
- in order to **determine** a slot’s value in \(R_k\) the collector takes the following steps, which are a prefix of the steps of a write barrier: 1) it loads the value \(v\) from \(s\), 2) it probes \(\text{Dirty}(s)\), 3) if the flag is off then \(v\) is the value of \(s\) in \(R_k\), otherwise \(s\) is undetermined and a record of it was taken by some mutator.

This protocol guarantees that only and exactly the values that were current at the time the recent conceptual snapshot was taken are recorded by mutators. Additionally, this protocol has the
property of compression of the information recorded in the sense that only initial modifications to a slot are recorded. Subsequent modifications are not relevant for the algorithm's execution since it only need know what are the values of a changed slot in the current and previous conceptual snapshots.

3.3 Overview of mutator's cooperation

The mutators cooperate with the collector through executing the update protocol described above for each modification of a pointer in the heap. We stress that there is no need for executing this protocol for updates of pointers in the registers or stack (i.e., the local roots.)

During object creation, the address of the newly created object is recorded for use of the collector.

3.4 Overview of the collection cycle

Let us present the steps of a garbage collection cycle.

The hard handshake—obtaining values from the previous snapshot and taking a new conceptual snapshot. During this handshake the collector gathers information regarding all slots that have been changed since the previous handshake from the mutators. The information gathered contains slots' values in previous snapshot \( R_{k-1} \). There exists information on any slot that has been modified since the previous conceptual snapshot was taken.

While the mutators are stopped their local states are scanned in order to mark as local all objects that are directly reachable. Their local ZCTs are merged into the global ZCT and are then cleared. Finally, all dirty flags are cleared in order to signal the mutators that they should start taking records of the modifications they apply to heap slots that refer to the current conceptual snapshot, i.e., to \( R_k \).

Adjusting rc fields due to modified slots. After resuming mutators, the collector adjusts \( rc \) fields due to each modified slot by:

- trying to determine the value of it at the time of the current snapshot \( R_k \), without interfering with the program threads. To do that, the collector reads the value of the slot from the heap, and verifies that its dirty flag is clear. If the dirty flag is indeed clear, then the slot has not been modified since the handshake and the value of it in the \( R_k \) snapshot has been obtained. The \( rc \) value of the referenced object is incremented. If the dirty flag is set, then the slot is undetermined. Then the collector has to obtain the value of such a slot by peeking at the mutators modification records.

- decrementing the \( rc \) field of the object the slot was referring to in the snapshot of the previous cycle. The identity of this object is known to the collector from the information recorded in between the cycles by the mutators and communicated to the collector during the handshake. If the decremented \( rc \) field drops to zero the referred object is considered a candidate for reclamation and is accordingly added to the ZCT.

Incrementing \( rc \) fields of objects referenced by undetermined slots. The collector asynchronously, i.e., without suspending the threads, gathers information about those slots that have been changed since the first handshake of the same cycle. A subset of these slots are the undetermined slots. The collector infers from the recorded information undetermined slots' values in the conceptual snapshot \( R_k \). It then increments the \( rc \) fields of the referenced objects.
**Reclaiming garbage.** The collector proceeds to reclaim unreachable objects, according to the following criteria: collect objects which have zero *rc* field and which are not marked *local*.

### 3.5 Data structures

In this section we briefly present the data-structures which are used in the algorithm.

**Thread’s history buffers.** Each thread has a local buffer in which it records the value of a slot that is modified for the first time after a snapshot is “announced”, i.e., after the handshake of a cycle. This local buffer is denoted $Buffer_i$, and it contains pairs of the form $(s, v)$ where $v$ is the contents of $s$ as read by the thread before updating $s$.

The buffer is implemented as an array of pairs with an associated pointer to the next entry to be used, denoted $CurrPos_i$. We assume that both $Buffer_i$ and $CurrPos_i$ reside in shared memory and thus are accessible to the collector at any moment.

If a thread logs the pair $(s, v)$ in its buffer then we say that it *associates* $v$ with $s$. It holds that if $T_i$ associated $v$ with $s$ then $v$ is the object $s$ was referring to in the last conceptual snapshot.

The collector gathers mutators’ histories and computes their union. This action is done twice per cycle: the first time when the world is stopped, in order to learn exactly which slots have been changed since then last cycle and what value they then assumed; the second time is done asynchronously in order to find what are the values of undetermined slots. The resulting sets of pairs are denoted $Hist_k$ and $Peek_k$ respectively.

**Slots’ dirty flags.** A unique dirty flag is associated with every slot. The purpose of the slot’s flag is to signify whether the slot is being modified during the current cycle. A mutator should be able to atomically write and atomically read the flag. To outline a feasible implementation, the flag can be implemented as a byte of memory, modifiable and accessible using ordinary memory accesses.

**Global and local Zero Count Tables.** The *Zero Count Table* or *ZCT* for short is a collector maintained set which records any object that its reference count field drops to zero at some point in the operation of the algorithm. The set $ZCT_k$ denotes the contents of the ZCT at cycle $k$. $ZCT_k$ contains primary candidates for reclamation in cycle $k$. That is, if an object is collected during cycle $k$ then either it’s in $ZCT_k$ or it was reachable from an object in $ZCT_k$.

Each mutator thread $T_i$ keeps a local ZCT of newly allocated objects, denoted $New_i$, in which it stores references to objects it creates. The set is cleared by the collector at the handshake of each cycle not before its contents are copied into the collector-maintained ZCT.

**Local marks.** According to the algorithm all objects which are directly reachable should be marked as *local* atomically with the construction of the snapshot. In our algorithm’s notation, we refer to the set of objects directly reachable from thread $T_i$ as $State_i$. During the handshake, the union of all $State_i$ sets is computed and stored in the set $Locals_k$, effectively marking all objects which are directly reachable at the time of the conceptual snapshot.

**Undetermined slots.** The collector need record which slots it failed determining, so that it may later look-up their value in the threads’ buffers. This is done by saving a reference to undetermined slots in the *Undetermined_k* set.

### 3.6 Mutator code

The mutators need execute garbage-collection related code on two occasions: when updating a slot and when allocating a new object. This is accomplished by the *Update* (figure 1) and *New* (figure 2) procedures, respectively. These operations are *protected*, i.e., a thread may not be suspended after it has executed the first instruction and before executing the last instruction of these
Procedure Update(s: Slot, new: Object)
begin
1. local old := read(s)
   // was s written to since the last cycle?
2. if ¬Dirty(s) then
   // ... no; keep a record of the old value.
3. Buffer_i[CurrPos_i] := ⟨s, old⟩
4. CurrPos_i := CurrPos_i + 1
5. Dirty(s) := true
6. write(s, new)
end

Figure 1: Mutator Code—Update Operation

Procedure New(size: Integer) : Object
begin
1. Obtain an object o from the allocator, according to the specified size.
   // add o to the thread local ZCT.
2. New_i := New_i ∪ {o}
3. return o
end

Figure 2: Mutator Code—for Allocation

operations.

3.7 Collector code

The code for cycle k is given in procedure Collection-Cycle, in figure 3. Each of the procedures invoked during a cycle is now described.

Procedure Read-Current-State (figure 4). After all threads are stopped their local state, new object sets and buffers are delivered to the collector. Before resuming the threads the collector cleans all dirty marks.

Procedure Update-Reference-Counters (figure 5), reference counters are updated by decrementing the “old” values and trying to determine current values and increment them. Undetermined slots are recorded.

Procedure Read-Buffers (figure 6) asynchronously reads threads’ buffers. Each thread T_i is considered at a time. The variable CurrPos_i is probed. Then the range [1...CurrPos_i – 1] of

Procedure Collection-Cycle
begin
1. Read-Current-State
2. Update-Reference-Counters
3. Read-Buffers
4. Fix-Undetermined-Slots
5. Reclaim-Garbage
end

Figure 3: Collector Code
Procedure **Read-Current-State**

```plaintext
begin
1. suspend all threads
2. Hist_k := ∅
3. Locals_k := ∅
4. for each thread T_i do
   // copy buffer (without duplicates.)
   5. Hist_k := Hist_k ∪ Buffer_i[1...CurrPos_i − 1]
   6. CurrPos_i := 1
   // “mark” local references.
   7. Locals_k := Locals_k ∪ State_i
   // copy and clear local ZCT.
   8. ZCT_k := ZCT_k ∪ New_i
   9. New_i := ∅
10. Clear all dirty marks
11. resume threads
end
```

**Figure 4:** Collector Code—Procedure **Read-Current-State**

Procedure **Update-Reference-Counters**

```plaintext
begin
1. Undetermined_k := ∅
2. for each <s,v> pair in Hist_k do
3.   curr := read(s)
4.   if ¬Dirty(s) then
5.     curr.rc := curr.rc +1
6.   else
7.     Undetermined_k := Undetermined_k ∪ {s}
8.     v.rc := v.rc − 1
9.   if v.rc = 0 ∧ v ∉ Locals_k then
10.    ZCT_k := ZCT_k ∪ {v}
end
```

**Figure 5:** Collector Code—Procedure **Update-Reference-Counters**
Figure 6: Collector Code—Procedure Read-Buffers

```plaintext
Procedure Read-Buffers
begin
1. \( Peek_k := \emptyset \)
2. for each thread \( T_i \) do
3. \hspace{1em} \text{local} \ ProbEdPos := CurrPos_i
4. \hspace{2em} \text{// copy buffer onto} \ Peek_k.
5. \hspace{1em} Peek_k := Peek_k \cup Buffer[1\ldots ProbEdPos - 1]
end
```

Figure 7: Collector Code—Procedure Fix-Undetermined-Slots

```plaintext
Procedure Fix-Undetermined-Slots
begin
1. \hspace{1em} for each pair \( \langle s, v \rangle \) pair in \( Peek_k \)
2. \hspace{2em} if \( s \in Undetermined_k \) do
3. \hspace{3em} v.rc := v.rc + 1
end
```

Buffer_i (which is empty if \( CurrPos_i = 1 \)) is copied onto the set \( Peek_k \) (the set is called \( Peek_k \) because it allows the collector to peek at the mutators buffers without stopping them.)

Procedure Fix-Undetermined-Slots (figure 7) passes item by item on the set \( Peek_k \) and finds the missing values of all undetermined slots. The \( rc \) fields of these values are incremented.

Procedure Reclaim-Garbage (figure 8). As a first stage in the operation of Reclaim-Garbage the collector considers all objects in \( ZCT_k \) and checks their reference count and local status. If an object has a positive reference count, then it is ignored. Otherwise, if the object is local, then it is added to \( ZCT_{k+1} \). The last case is when an object has both zero reference count and is not local, such an object is kept in \( ZCT_k \).

After applying this sieving pass on \( ZCT_k \), it contains only objects with zero \( rc \) field which are not marked \( local \). This is a sufficient condition for the objects to be garbage, hence the collector proceeds by deleting these objects by means of the Collect procedure, which is next described.

Procedure Collect (figure 9) is responsible for deleting garbage objects. It stores null into each of its operand’s slots not before the reference counts of the pointed objects are decremented.

Figure 8: Collector Code—Procedure Reclaim-Garbage

```plaintext
Procedure Reclaim-Garbage
begin
1. \( ZCT_{k+1} := \emptyset \)
2. for each object \( o \in ZCT_k \) do
3. \hspace{1em} if \( o.rc > 0 \) then
4. \hspace{2em} \( ZCT_k := ZCT_k \setminus \{o\} \)
5. \hspace{1em} else if \( o.rc = 0 \land o \in Locals_k \) then
6. \hspace{2em} \( ZCT_k := ZCT_k \setminus \{o\} \)
7. \hspace{1em} \( ZCT_{k+1} := ZCT_{k+1} \cup \{o\} \)
8. for each object \( o \in ZCT_k \) do
9. Collect(o)
end
```
Figure 9: Collector Code—Procedure Collect

The referred objects are recursively deleted based on the same criteria applied by the Reclaim-Garbage procedure.

### 3.8 Intuition

A central point in the algorithm’s operation is that logging always records a slot’s value at the time the last handshake occurred. Indeed, several competing threads may log the same slot, yet they would all associate it with one agreed value—the one that prevailed at the last handshake. It is easy to see that this is the case since no thread modifies the slot prior to raising its dirty flag. In the write barrier, a thread first reads the slot and only then the flag. Thus, a fetched turned-off flag implies that the previously read value is the original one from the time of the handshake. The collector uses exactly the same mechanism in order to determine a slot.

Another important point to note is that a slot and its associated value are fully logged by a mutator before it raises the slot’s dirty flag. Thus, if the collector senses that a slot is raised, it is guaranteed that it will find a record of the slot in some thread’s buffer, when it would look up threads’ buffers asynchronously in order to resolve undetermined slots.

We further comment that the price that appears to be involved in copying the mutators’ buffers and local ZCTs is non-existent in practice, since in a real implementation the mutator would deliver its buffer to the collector and would start working using a new buffer, thus the true overhead of delivering and clearing these sets amounts to a handful of pointer updates. Consequently, the mutators are stopped for as long as it takes to clear the dirty flags. Using a bitmap and some help from the virtual memory system this can be done rather quickly. We elaborate on the implementation of dirty flags in section 5.1.

The algorithm’s correctness proofs are in appendix A.

### 4 The Sliding View Algorithm

In the snapshot algorithm we have managed to execute a major part of the collection while the mutators run concurrently with the collector. The main disadvantage of this algorithm is the hard handshake in the beginning of the collection. During this handshake all threads are stopped while the collector clears the dirty flags and receives the mutators’ buffers and local ZCTs. This hard handshake hinders both efficiency, since only one processor executes the work and the rest are idle, and scalability, since more threads will cause more delays. While efficiency can be enhanced
by parallelizing the flags’ clearing phase, scalability calls for eliminating hard handshakes from
the algorithm. This is indeed the case with our second algorithm, which avoids hard handshakes
completely.

In this section, we present an algorithm which uses four soft handshakes per cycle. Thus, the
system never comes to a grinding halt. Mutators are only stopped one at a time, and only for a
short interval, its duration depends on the size of mutators’ local states.

In the snapshot algorithm we had a fixed point of time, namely, when all mutators were stopped
in a hard handshake, to which all logging and successful determining of slots referred. By dispensing
with the hard handshake we no longer have this fixed point of time. Rather, we have a fuzzier picture
of the system, formalized by the notion of a sliding view which is essentially a non-atomic picture
of the heap.

We show how sliding views can be used instead of atomic snapshots in order to devise a collec-
tion algorithm. Then, we present an algorithm which implicitly computes a sliding view (bearing
similarity to the first algorithm which implicitly computes an atomic snapshot) and collects garbage
using it. In appendix B we prove the algorithm correct.

4.1 Scans and sliding views

Pictorially, a scan \( \sigma \) and the corresponding sliding view \( V_\sigma \) can be thought of as the process of
traversing the heap along with the advance of time. Each word of memory \( s \) in the heap is probed
at time \( \sigma(s) \); if at that particular moment \( s \) contains a reference, then we record that value as the
value of \( V_\sigma(s) \), otherwise, the word is not a slot at \( \sigma(s) \), which we signify by letting \( V_\sigma(s) \) be equal
null.

That is, a scan \( \sigma \) is a function that assigns a time stamp to each word in the heap. we define
Start(\( \sigma \)) to be the earliest time assigned to any slot by \( \sigma \). Similarly we define End(\( \sigma \)).

Formally, the sliding view associated with a scan \( \sigma \), which is denoted \( V_\sigma \), is a function that
assigns a pointer value to each memory word \( s \) in the heap:

\[
V_\sigma(s) \overset{\text{def}}{=} \begin{cases} 
null & \text{if } s \text{ is not an allocated slot at } \sigma(s) \\
\mathit{s@}\sigma(s) & \text{otherwise} 
\end{cases}
\]

Note that a snapshot of the heap is just a special case of a sliding view in which all slots are
scanned at the same time.

For an object \( o \) and a sliding view \( V_\sigma \) we define the Asynchronous Reference Count of \( o \) with
respect to \( V_\sigma \) to be the number of slots in \( V_\sigma \) referring to \( o \):

\[
\text{ARC}(V_\sigma; o) \overset{\text{def}}{=} |V_\sigma^{-1}(o)|
\]

The usual reference count of heap pointers to an arbitrary object \( o \) at time \( t \) is also just a special
case of the above formulation with \( \sigma \) set to: \( \forall s, \sigma(s) = t \). Then we have:

\[
\forall \ \text{Object } o, \text{ARC}(V_\sigma; o) = \text{RC}(o)@t
\]

The feature of sliding views of being incrementally constructed is appealing since it implies that
one need not stop all mutator threads simultaneously in order to compute the view. But can we
find a safe collection criteria based on sliding views? Of course, using a sliding view is not as simple
as using a snapshot. Clearly, trying to use the snapshot algorithm when we are only guaranteed
that logging and determining reflects some sliding view is bound to fail. For example, the only
reference to object \( o \) may "move" from slot \( s_1 \) to slot \( s_2 \), but a sliding view might miss the value
of \( o \) in both \( s_1 \) (reading it after modification) and \( s_2 \) (reading it before modification). Thus object \( o \) has a zero asynchronous reference count with respect to the aforementioned sliding view, yet it never had a true zero reference count.

Now suppose that, as in the above example, \( ARC(V_\sigma; o) = 0 \), that is, every slot in the heap was probed and none referred to \( o \). This time, however, we assume additionally that for any slot \( s \), there has not been a store of \( o \) into \( s \) performed in the time interval \( \sigma(s) \) to \( End(\sigma) \). If we took an atomic snapshot of the heap at time \( End(\sigma) \) we would have discovered that no slot is referring to \( o \) for the simple reason that it did not refer to it at \( \sigma(s) \) and no pointer to \( o \) was stored into it until \( End(\sigma) \). The same arguments are used to show the more general claim:

**Lemma 4.1 (Sliding Views)** Let \( V_\sigma \) be a sliding view and let \( o \) be an object. If for any slot \( s \), no reference to \( o \) is stored into \( s \) at, or after, \( \sigma(s) \) and before \( End(\sigma) \) then \( RC(o)@End(\sigma) \leq ARC(V_\sigma; o) \). Furthermore, the set of slots that refer to \( o \) at \( End(\sigma) \) is a subset of those that point to it in \( V_\sigma \).

### 4.2 Using sliding views to reclaim objects

Based on the above observations we present a generic garbage collection algorithm:

1. Each thread \( T_i \) has a flag, denoted \( Snoop_i \) which signifies whether the collector is in the midst of constructing a sliding view. This flag is modifiable by the collector and readable by the mutator \( T_i \).

2. Mutator \( T_i \) executes a write barrier in order to perform a heap slot update. The generic algorithm requires that after the store proper to the slot is performed, i.e., object \( o \) is actually written into slot \( s \), the thread would probe its \( Snoop_i \) flag and, if the flag is set, would mark \( o \) as \textit{local}. We call this probing of the \( Snoop_i \) flag and the subsequent marking \textit{snooping}. Any specific implementation of the generic algorithm may require additional steps to be taken as part of the write barrier.

3. As usual, threads may not be suspended in the midst of an update.

4. A collection cycle contains the following stages:

   1. the collector raises the \( Snoop_i \) flag of each thread. This indicates to the mutators that they should start snooping.

   2. the collector computes, using an implementation-specific mechanism, a scan \( \sigma \) and a corresponding sliding view, \( V_\sigma \), concurrently with mutators’ computations. The actual manner using which the collector computes \( V_\sigma \) is immaterial, it’s just important that it arrives at a valid sliding view.

   3. each thread is then suspended (one at a time) its \( Snoop_i \) flag is turned off and every object directly reachable from it is marked \textit{local}. The thread is then resumed.

   4. now, for each object \( o \) we let \( o.rc := ARC(V_\sigma; o) \).

   5. at that point, we can deduce that any object \( o \) that has \( o.rc = 0 \) and that was not marked \textit{local} is garbage.

Since for each thread the \( Snoop_i \) flag is set for the entire duration of the sliding view computation we conclude that any object which is not marked \textit{local} satisfies, according to lemma 4.1, \( ARC(V_\sigma; o) \geq RC(o)@End(\sigma) \) thus \( 0 = o.rc = ARC(V_\sigma; o) \) implies \( RC(o)@End(\sigma) = 0 \). It may be, however, that \( o \) is \textit{directly reachable} from some thread at \( End(\sigma) \). Nevertheless, since no local
reference to \( o \) was observed by any thread when its state was scanned (in stage (3) of the collector) and it was not “snooped” prior to it, any thread which possessed such a local reference must have discarded it prior to responding the handshake of stage (3) without ever raising the heap reference count of \( o \) above zero. We conclude that by the time the handshake of stage (3) ends, \( o \) is garbage.

The snooping mechanism may lead to some floating garbage as we conservatively not collect objects which are marked local, although such objects may become garbage before the cycle ends. However, such objects are bound to be collected in the next cycle.

We have termed this algorithm “generic” since the mechanism for computing the sliding view is unspecified. In the fleshed out algorithm that we next present we rely on the methods of logging and arbitration that were introduced in the context of the snapshot algorithm in order to implicitly construct a sliding view. When the implicit construction is done, it holds for each object that \( arc = ARC(V; o) \), where \( V \) is the sliding view that was constructed implicitly. Since we are not interested in the sliding view itself but rather on its manifestation through the rc fields, this implicit computation suffices for collection purposes.

### 4.3 Algorithm’s idea

We will present a concrete sliding view based reference counting algorithm which implements the generic sliding view algorithm of the previous section.

The concrete algorithm uses ideas similar to those presented in the context of the snapshot algorithm of section 3. In particular, it uses mutators’ logging in order to obtain modified slots’ values in the last sliding view.

Whereas in the snapshot algorithm the mutators and collector cooperate synchronously, using one hard handshake per cycle, in order to compute a rereference count reflecting an atomic snapshot, in the sliding view algorithm they cooperate asynchronously, using four soft handshakes per collection, in order to compute a reference count reflecting a sliding view. This significantly improves scalability at the cost of having to deal with fuzzier information. We provide augmented arbitration and race-detection mechanisms in order to overcome the difficulties introduced by the enhanced asynchronicity.

### 4.4 Overview of mutator’s cooperation

Mutators use the write barrier of the snapshot algorithm (figure 1) with the additional snooping and marking added after the store proper. Object creation is unchanged from the snapshot algorithm.

### 4.5 Overview of the collection cycle

Each collection cycle is comprised of the following steps, which are visually illustrated in figure 10:

**Signaling snooping.** The collector raises the \( Snoop \) flag of each thread, signaling to the mutators that it is about to start computing a sliding view.

**Reading buffers (first handshake.)** This step initiates a soft handshake during which thread’s buffers\(^7\) are retrieved and then are cleared. The slots which are listed in the buffers are exactly those slots that have been changed since the last cycle\(^8\).

\(^7\)These are the same thread buffers as in the first algorithm.

\(^8\)The meaning of “changing” in this asynchronous setting is defined as follows. A slot is changed during cycle \( k \) if some thread changed it after responding to the first handshake cycles \( k \) and before responding to the first handshake of cycle \( k + 1 \).

Figure 10: Timing diagram for the sliding view algorithm
Clearing. The dirty flags of the slots listed in the buffers are cleared. Note that the clearing occurs *while the mutators are running*. Clearing the dirty flags tells the mutators that they should start logging slots from fresh, i.e., that a new cycle and a sliding view associated with it have begun so that the mutators should log slots’ values in this new sliding view.

**Reinforcing dirty marks (second handshake.)** The collector carries a second handshake during which it reads the contents of the threads’ buffers. The collector then *reinforces* the flags of the listed slots, i.e., it turns them on.

Note that the slots listed in the read buffers are slots that have been logged between the first and second handshake. Thus, such a slot’s flag is raised by mutators and might be concurrently turned off by the collector. Hence these slots are subjected to a race condition between two conflicting processes and are accordingly termed *clearing conflict slots*.

**Assuring reinforcement is visible to all mutators (third handshake.)** The third handshake is carried out. No action is taken during it.

**Consolidation (fourth handshake.)** This stage has two objectives: 1) solving conflicting logging of conflict slots. 2) marking thread local states. In order to achieve these goals a fourth soft handshake is performed. During the handshake thread local states are scanned and marked *local*. Threads’ buffers are retrieved once more and are *consolidated*.

Consolidating threads’ buffers amounts to the following. For any slot that appears in the threads’ buffers accumulated between the first and fourth handshakes, pick *any* occurrence of the slot and copy it to a digested, inconsistencies free, history. All other occurrences of the slot are discarded.

The digested history replaces the accumulated threads’ buffers, i.e., the history for the next cycle is comprised of the digested history of threads’ logging between the first and fourth handshakes of the current cycle, unified with threads’ buffers representing updates that will occur after the fourth handshake of the current cycle but before the first handshake of the next cycle.

**Updating.** After clearing, reinforcing, making sure that the reinforcement is visible to all mutators and consolidating the buffers the collector proceeds to adjust *rc* fields due to differences between the sliding views of the previous and current cycle. This is done exactly as in the snapshot algorithm. Recall that the collector may fail determining what is the “current” value of a slot. Such a slot is *undetermined*.

**Gathering information on undetermined slots.** The collector asynchronously reads mutators’ buffers. It then unifies the set of read pairs with the digested history computed in the consolidation step. The set of undetermined slots is a subset of the slots appearing in the unified set so the collector may now proceed to look up the values of these undetermined slots.

**Incrementing rc fields of objects referenced by undetermined slots.** Any undetermined slot is looked up in the unified set and the *rc* field of the associated object is incremented.

**Reclamation.** Reclamation generally proceeds as in the previous algorithm, i.e., recursively freeing any object with zero *rc* field which is not marked *local*. Due to the extended meaning of *locality*, that is, it encapsulates the “snooping” requirement of the generic algorithm, the condition for being garbage is the same as in the snapshot algorithm. There is a problem, however, with reclaiming objects whose slots appear in the digested history, i.e., objects which were modified since the cycle commenced but became garbage before it ended. We elaborate on this problem in the sequel.
4.6 Intuition: where’s the sliding view?

Each cycle of the algorithm has a conceptual scan and a corresponding sliding view associated with it which encapsulate the agreed knowledge of the mutators and collector regarding the “current” value of each slot in the cycle. We denote the scan for cycle $k$ as $\sigma_k$ and the corresponding sliding view is termed $V_k$. Consider a slot $s$. The value of $\sigma_k(s)$ is defined as follows:

- **Rule 1**—slots which are **not logged during cycle** $k - 1$. If no thread logs $s$ prior to responding to the first handshake of cycle $k$ then we set $\sigma_k(s)$ as the time at which the first thread responds to the first handshake.

- **Rule 2**—slots which are **logged during cycle** $k - 1$ that were **not logged between the first and third handshakes**. Here we set $\sigma_k(s)$ to be the time at which the second handshake terminates (i.e., when the last thread responds to it.)

- **Rule 3**—slots which are **logged during cycle** $k - 1$ that were **logged between the first and third handshakes of cycle** $k$. Any such slot is consolidated. Let $v$ be the chosen consolidated value of $s$. We define $\sigma(s)$ to be the time at which some particular thread which logged the pair $\langle s, v \rangle$, between responding to the first and fourth handshakes, fetched $v$ from memory as the first instruction of its write barrier (note that this thread might have already responded to the third handshake when fetching $v$).

We now explain this particular choice of a cycle’s sliding view. What we require from the sliding view, and from the algorithm with respect to this particular sliding view, is that:

1. In case the history for the next cycle contains the pair $\langle s, v \rangle$ then $v$ must be $V_k(s)$. The history for the next cycle may not contain conflicting values for $s$.

2. In case the collector succeeds in determining a slot $s$, i.e., it succeeds determining the “current” value of $s$, we require that the determined value be the same one as the slot’s value in the cycle’s sliding view, i.e., $V_k(s)$. Again, the history for the next cycle may not contain conflicting values.

3. Each slot which is modified between two consecutive scans (i.e., a store to the slot is scheduled at, or after $\sigma_k(s)$ and before $\sigma_{k+1}(s)$) should be logged, making the value it assumed during the last sliding view available to the collector.

4. Any update of $s$ whose store proper operation is scheduled at, or after $\sigma_k(s)$ and before $End(\sigma_k)$ should snoop its operand; i.e., mark it $\textit{local}$.

It turns out that these requirements are all met by the algorithm with respect to the sliding view we have just defined. We give intuition for this according to the rule by which $\sigma(s)$ is defined.

If $\sigma(s)$ is defined according to rule (1) then because no thread logged $s$ up to the moment the first handshake of cycle $k$ started the dirty flag of $s$ is clear at that particular moment. If some thread would log $s$ after responding to the first handshake it is bound to associate $s$ with the value it assumed when the handshake started. Similarly, if the collector will succeed determining the slot, it will find the value it assumed at that moment as well.

Otherwise, if $\sigma(s)$ is defined according to rule (2) then it is easy to see that at the time the second handshake ends the dirty flag of $s$ is clear (because the collector cleared it and no mutator raised it) and no update is occurring. This implies that any subsequent updates and determinining will relate to this point of time, as required.
Finally, if \( \sigma(s) \) is defined by rule (3), i.e., by picking the time at which a thread which logged the “winning pair” \( \langle s, v \rangle \) loaded \( v \) from \( s \), we trivially have that the digested history agrees with \( V_k(s) \). Also, since some thread logs \( s \) prior to responding to the third handshake no thread will log \( s \) after responding to the fourth handshake. Therefore, the non-digested part of the history for the next cycle will not contain any record of \( s \). Similarly, the collector would fail determining \( s \), satisfying our requirement for determining slots.

Note that the scan of a cycle spans, at most, from the beginning of the first handshake up to the end of the third handshake. Since the Snoo p flags are turned on prior to the first handshake and are turned off only at the fourth handshake we conclude that the snooping requirement is kept.

We now turn to specify the pseudo-code for the algorithm.

### 4.7 Mutator code

Mutator code in the second algorithm is almost identical to the one in the first algorithm. In particular, the \texttt{New} procedure is unchanged.

The \texttt{Update} procedure (in figure 11) includes an additional test, that checks whether the thread-specific flag \texttt{Snoop} is set. If so, the object whose reference is stored into the slot is marked \texttt{local} by adding it to the thread-specific set \texttt{Locals}. This marking implements the “snooping” requirement of the generic algorithm.

### 4.8 Collector code

Collector’s code for cycle \( k \) is depicted in figure 12. Let us describe briefly the role of each of the collector’s procedures.

\textbf{Procedure Initiate-Collection-Cycle} (figure 13) is the counterpart of procedure \texttt{Read-Current-State} of the snapshot algorithm of section 3. However, since it stops each thread at a time (i.e., it carries out a soft handshake,) there is no atomic state being read. Also note these additional actions:

1. before the handshake is started, the \texttt{Snoop} flag is raised, signaling mutators that they should start snoop stores into heap slots.

2. the set \texttt{Hist}_k is not cleared as the first step of each cycle. Rather, the set already contains digested information about part of the logging relating to cycle \( k \) which has been accumulated by the collector during cycle \( k - 1 \).

\footnote{“winning” in the sense that \( v \) is chosen to be the consolidated value of \( s \).}
Procedure **Collection-Cycle**
begin
1. Initiate-Collection-Cycle
2. Clear-Dirty-Marks
3. Reinforce-Clearing-Conflict-Set
4. Consolidate
5. Update-Reference-Counters
6. Read-Buffers
7. Merge-Fix-Sets
8. Fix-Undetermined-Slots
9. Reclaim-Garbage
end

Figure 12: Sliding View Algorithm: Collector Code

Procedure **Initiate-Collection-Cycle**
begin
1. for each thread $T_i$ do
2. $Snoop_i := true$
3. for each thread $T_i$ do
4. suspend thread $T_i$
   // copy (without duplicates) and clear buffer.
5. $Hist_k := Hist_k \cup Buffer_i[1...CurrPos_i - 1]$
6. $CurrPos_i := 1$
7. resume $T_i$
end

Figure 13: Sliding View Algorithm: Procedure **Initiate-Collection-Cycle**

3. the $New_i$ sets are not retrieved by the collector during the handshake. Rather, they will be retrieved during the forthcoming fourth handshake.

Procedure **Clear-Dirty-Marks** (figure 14) clears all dirty marks that were set by mutators prior to responding to the first handshake. Note that the clearing takes place while the mutators are running.

Procedure **Reinforce-Clearing-Conflict-Set** (figure 15) implements the reinforcement step and assures that it is visible to all mutators. A second handshake takes place, during which thread buffers are read. The unified set of pairs is stored in the set $ClearingConflictSet_k$. Then, flags of slots that appear in $ClearingConflictSet_k$ are reinforced to be **true**. Finally, the third handshake of the cycle takes place. There is no action taken during it. The reason for this additional handshake is that a thread can fall behind a sibling thread by at most one handshake. Thus threads that have responded to the fourth handshake will not be interfered by operations carried out by threads

Procedure **Clear-Dirty-Marks**
begin
1. for each $(s, o) \in Hist_k$ do
2. $Dirty(s) := false$
end

Figure 14: Sliding View Algorithm: Procedure **Clear-Dirty-Marks**
Procedure Reinforce-Clearing-Conflict-Set
begin
1. ClearingConflictSet_k := ∅
2. for each thread T_i do
3. suspend thread T_i
4. ClearingConflictSet_k := ClearingConflictSet_k ∪ Buffer_i[1..Curr Pos_i – 1]
5. resume thread T_i
6. for each s ∈ ClearingConflictSet_k do
7. Dirty(s) := true
8. for each thread T_i do
9. suspend thread T_i
10. nop
11. resume T_i
end

Figure 15: Sliding View Algorithm: Procedure Reinforce-Clearing-Conflict-Set

during the clearing or reinforcement stages, i.e., threads that still haven’t responded to the third handshake.

Procedure Consolidate (figure 16). The task of the procedure is to implement the fourth handshake, during which mutators’ buffers are read again and then are cleared. The accumulated set of pairs is stored in a temporary set, denoted Temp. The temporary set is then consolidated into the set Hist_{k+1}.

Additionally, the Locals sets, which record snooped objects are copied onto the set Locals_k and are cleared. Objects directly reachable from a thread’s local state (denoted in the algorithm by State_i) are copied onto Locals_k as well. The thread local ZCTs, which reside in the New_i sets, are copied onto the set ZCT_k and are then cleared.

Procedures Update-Reference-Counters, Read-Buffers and Fix-Undetermined-Slots are the same ones used by the snapshot algorithm (see figures 5, 6, 7). Note, however that there is an additional procedure, Merge-Fix-Sets (figure 17), invoked between Read-Buffers and Fix-Undetermined-Slots. Since an undetermined slot may appear either in the set of buffers read after the fourth handshake, or in the set of buffers read before the handshake, we need merge the two sets into a single set in order to resolve undermined slots. This is done by procedure Merge-Fix-Sets.

Procedure Reclaim-Garbage (figure 8) is the same procedure used in the first algorithm. Due to the extended meaning of the Locals_k set the conditions for reclaiming objects in Reclaim-Garbage need not be changed.

Procedure Collect (figure 18) does require modifications, however. The dirty flag of each slot of the candidate object is checked. If all flags are off, then there cannot be any record of a constituent slot of it in the digested history for the next cycle and there will not be any further logging of such a slot after the fourth handshake as well, as o is unreachable then. Hence, the collector may simply clear o’s slots and return it to the memory manager without causing inconsistencies.

If, however, some slot has its dirty flag set, then some thread modified the slot prior to responding to the fourth handshake and logged the slot’s previous value before hand. Only afterwards did the containing object become unreachable and the collector detected that fact. This is possible, for example, due to the following scenario: object o is only directly reachable from thread T_i. After responding to the first handshake, T_i stores a value, v_1, into the slot s of o. Then it stores a second value v_2, into the slot. Then it discards its local reference to o, before responding to the fourth
Procedure **Consolidate**

begin

1. local Temp := ∅
2. Locals_k := ∅
3. for each thread $T_i$ do
4.   suspend thread $T_i$
5.   Snoop_i := false
   // copy and clear snooped objects set
6.   Locals_k := Locals_k ∪ Localsi
7.   Local_i := ∅
   // copy thread local state and ZCT.
8.   Local_k := Locals_k ∪ State_i
9.   ZCT_k := ZCT_k ∪ New_i
10. New_i := ∅
    // copy local buffer for consolidation.
11. Temp := Temp ∪ Buffer[i ... CurrPos; - 1]
    // clear local buffer.
12. CurrPos_i := 1
13. resume thread $T_i$
    // consolidate Temp into Hist_k+1.
14. Hist_k+1 := ∅
15. local Handled := ∅
16. for each $(s, v) ∈ Temp$
17. if $s ∉ Handled$ then
18.   Handled := Handled ∪ $\{s\}$
19.   Hist_k+1 := Hist_k+1 ∪ $\{(s, v)\}$
end

Figure 16: Sliding View Algorithm: Procedure **Consolidate**

Procedure **Merge-Fix-Sets**

begin

1. Peek_k := Peek_k ∪ Hist_k+1

end

Figure 17: Sliding View Algorithm: Procedure **Merge-Fix-Sets**
procedure Collect(o: Object)
begin
1. local DeferCollection := false
2. foreach slot s in o do
3. if Dirty(s) then
4. DeferCollection := true
5. else
6. val := read(s)
7. val.rc := val.rc - 1
8. write(s, null)
9. if val.rc = 0 then
10. if val ∉ Locals_k then
11. Collect(val)
12. else
13. ZCT_{k+1} := ZCT_{k+1} ∪ {val}
14. if ~DeferCollection then
15. return o to the general purpose allocator.
16. else
17. ZCT_{k+1} := ZCT_{k+1} ∪ {o}
end

Figure 18: Sliding View Algorithm: Procedure Collect

handshake. Thus, s is both a part of Hist_{k+1} and is supposed to be reclaimed during cycle k.
Note that when the collector consolidated s it considered v_1 as its current value, rather than v_2.
Consequently, the collector may not simply clear s and decrement v_2.rc, as this will not undo the
previous action of incrementing v_1.rc.

The solution we adopted to the problem is to defer the collection of o to the next cycle. Since it
is unreachable already in the current cycle, the problem described above cannot reoccur during the
next cycle. This is computationally efficient but has the drawback of retaining uncollected garbage
more than is really needed.

An alternative solution is to let the collector find what is the value of s in the sliding view of
the current cycle as it appears in the digested history Hist_{k+1}. Suppose v stands for this looked-up
value. The collector then decrements v.rc and discards the pair (s, v) from Hist_k, in order to avoid
another, spurious, decrement during cycle k+1. We have preferred the former solution to the latter
since the latter incurs the computational overhead of the search, introducing an O(n log n) term to
the step complexity of a cycle, which is otherwise of linear complexity.

5 Implementation Issues

In this section we shift from the abstract treatment of the dirty flags and the log buffers and
suggest concrete implementations for these data structures. Then we show how to treat global
roots. Finally we address the issue of memory consistency.

5.1 Dirty Flags

Both algorithms were presented in a rather high level and generic manner that leaves the imple-
mentation of several data structures unspecified. This method of exposition is useful for showing
the algorithms correct and it reveals the ideas behind the algorithms more clearly. In order to
implement the algorithms, we must select concrete data-structures for each abstract data-structure that is used. The algorithms share most data structures and access them similarly. Yet the most crucial data structure, the dirty flags, are accessed in a fundamentally differing manner by the two algorithms.

The first algorithm calls for an implementation of the slots’ dirty marks that allows setting and reading by the mutators and collector on one hand and that supports a fast “clear all” operation by the collector, on the other hand. The “clear all” operation need be fast since mutators are halted whilst it takes place. The second algorithm is less demanding in that respect. Dirty flags may be cleared less hastily as the mutators are running during the operation. While the expeditiousness of the clearing operation is still important, it may yield to other factors, such as space conservation and increased locality. Thus, the snapshot algorithm calls for bitmapped solutions, since bitmaps are easier to clear quickly, while the sliding view algorithm can work both with bitmapped and non-bitmapped solutions.

Non-bitmapped solutions locate the flags interspersed with the data. This has two notable benefits: (1) conservation of space, since we can allocate space per flags on a per type basis, rather then conservatively for every word of memory, as is done in a bitmapped solution and (2) increased locality of reference, as the flags are accessed by the mutators in conjunction with their respective slots and there is no need for the collector to implement the “clear all” operation. The downside of non-bitmapped solutions is the inability to clear the dirty flags quickly; they must be cleared one at a time, or in small batches, depending on the specific solution.

In section 5.1.1 we show how it is possible to associate a flag with a group of slots, rather then a flag for a single slot, thus saving space. Section 5.1.2 demonstrates how the overhead of initializing assignments can be eliminated.

5.1.1 Alloting a flag per a chunk of memory

In this section we elaborate on the idea according to which a flag can serve an indicator to a change in any of the slots within a fixed chunk of memory. The ideas contained in this section are similar to those that arise in the context of tracking inter-generational pointers in a generational collector that uses card marking. Details on the method of card marking can be found in [40].

If we let a single flag signify a change in a chunk of memory then the write barrier takes the following form, assuming that we want to store into the slot \( s \) the value \( v \):

- the flag for the chunk of memory containing \( s \) is optimistically probed, assuming that it is turned on. If it is indeed turned on, then we proceed directly to the store operation.
- otherwise, a replica of the slots that reside inside the chunk is created and stored locally.
- the flag for the chunk is then probed again. If it is now turned on, we proceed to the store operation.
- otherwise, we commit the replica just prepared to the log buffer, raise the flag and only then execute the store.

The collector code for determining a slot is changed accordingly. The collector tries to determine the value of an entire chunk instead of a single slot.

The scheme is characterized by a decreased memory consumption yet by spurious work imposed on the mutator and collector that have to process slots which haven’t really changed.

We think of three feasible methods for associating a group of slots with a flag: (1) associating each card, i.e., aligned chunk of \( 2^l \) bytes (where \( l \) is a parameter) with a flag, the flags reside in a
bitmap. (2) associating a flag with an object, the flags reside in a bitmap, and (3) a flag per object, where the flag is located inside the object.

Options (1) and (2) are suitable for both algorithms while (3) is appropriate only for the sliding view algorithm.

We note that it is not needed to log the identity of individual slots within a chunk. It suffices simply to log which non-null pointers the chunk contains. This property may ameliorate the cost of spurious logging. There is a tradeoff between: (1) logging the entire chunk conservatively and letting the collector figure out which part of the chunk’s replica are pointer slots and: (2) letting the mutator store precisely only true references. This is related to the nature of a chunk: does it correspond to an object or is it just an aligned piece of memory.

Working on an object basis lets the mutator efficiently record precisely object slots: we can produce a per-type slot-storing code that stores any heap slots contained in the object into the history buffer of the thread, or produce a per-type vector of slots’ indices and an efficient routine that logs the slots specified by the vector, given a base pointer to the object.

Identifying a flag with an object is also quite natural in terms of locality, i.e., we might expect that when a slot of an object is changed, then its sibling slots are likely to change as well, so the amount of unneeded information recorded is minimized. This might not be the case for an arbitrary chunk of memory that is prone to hosting unrelated objects.

The disadvantage of working with a flag-per-object scheme is dealing with objects which are too big. Applying the scheme for them will result in a wasteful replication of probably unchanged data. This can be avoided by treating big objects differently. Special care need be taken that the methods for small and big objects coexist.

Using a flag per object and a flags bitmap can be quite wasteful in terms of space: we need to allocate a flag in the bitmap per the granule of object alignment. Since objects are usually aligned on 16 bytes or smaller granules and since a typical object is some 50 bytes long, inlining the flag inside the object results in a substantial saving of space (not to mention the cases in which some unused bit in the object header is waiting to be exploited).

5.1.2 Initialization

This section discusses an optimization regarding the initialization of slots when the method of a flag-per-object is used.

By an initializing update we mean an update to an object’s slot that is bound to occur within a small fixed number of instructions from the object allocation site. For example, referring to languages such as C++ or Java, we expect initializing updates to abound in inlined constructors. As noted by [54, 27] initializing updates comprise the majority of updates in functional languages and garbage collected object oriented languages.

By treating the entire code block that executes the object creation and the initializing updates as a single transaction (i.e., we treat it as a protected piece of code), we can save a substantial amount of our algorithms’ overhead: after the object is created it is logged in the thread buffer with no contained pointers. The initializing updates then proceed without any write barrier.

Note that this protocol also deems the use of the local ZCT unnecessary as newly created objects are tracked using the ordinary history buffer.

5.2 Log buffers

The primary design factor in the implementation of the log buffers is how to make writing into them as fast as possible for a mutator executing an update. A secondary consideration is how
to allow the collector to read those records that have been fully logged (i.e., both slot and value members of a logged pair) without interrupting the mutator.

In order to satisfy the primary goal we suggest the following design, which is similar to the one described in [13]: a buffer will be implemented as a linked list of buffer-chunks. Each chunk is of size $2^k$, aligned on a $2^k$ boundary ($k$ is a parameter.)

A mutator that is executing an update will always have enough room to log the current transaction. This is an invariant which is maintained in the following manner: after logging a pair to the current chunk, the mutator checks whether the next update would cause the chunk to overflow (this check is a simple arithmetic one due to the chunk size alignment.) If that is the case, it tries extracting a new chunk from a list of free chunks. If it succeeds, it lets the new chunk point to the old one and starts using the new one. Otherwise, a new garbage collection cycle is started. The mutator then waits for the collector to notify it when there are free chunks. The collector makes part of the chunks available to mutators after processing them in the procedure **Update-Reference-Counters** and the rest after the execution of procedure **Fix-Undetermined-Slots**. In case the collector falls behind freeing chunks, a mutator may initiate a synchronous reference counting cycle or a synchronous tracing cycle.

Using a linked list of chunks allows the **Update** operation to be efficient in the common case that there is no overflow, yet it allows a finer grained load-balancing by letting each thread consume a different amount of chunks from its peers.

New sets can be implemented in much the same way, even sharing the same pool of chunks with the log buffers.

Implementing the second requirement, i.e., that the collector can read asynchronously the set of completely logged pairs can be achieved efficiently in the following manner.

- before the mutator starts using a buffer-chunk it zeroes it out.
- in order to store a record in the buffer a mutator first writes the value read, then it writes the slot address.
- the collector reads the records in the thread’s buffer sequentially. It knows that it has read a record which has not been completely logged when it sees a slot field with the value of null (note that the mutator never logs a slot whose address is null.)

Thus, the mutator can manipulate the buffer using only a single register that points to the next address to be written.

5.3 Global roots

We have left the treatment of global roots outside the specification of the algorithms. This choice has rendered the specification simpler while, as is next explained, it does not involve any loss of generality.

To see that this is indeed the case, we postulate that global roots can be treated exactly as heap slots. That is, each global root has a dirty flag corresponding to it and it is subject to the write barrier. This treatment is valid for the following reason. We picture all global roots as being the slots of a conceptual “globals” object. The “globals” object is directly reachable from any thread. Thus, reads and writes of global roots are equivalent to reads and writes of the respective slots of the directly reachable “globals” object. The “globals” object itself need not be marked or otherwise be operated on explicitly since it does not really exist and therefore there is no risk that it would be collected.
This argument directly suggests a concrete method for treating global references: associating a dirty flag with any such reference and applying the write barrier to it. However, all is not well. Implementing this policy can be quite involved because unlike for heap objects it is hard to find a systematic manner to associate a dirty flag with each global reference. We therefore propose alternative approaches to global variables.

In the snapshot algorithm, global references may be simply treated as their local counterparts, i.e., when all threads are stopped during the hard handshake, all objects which are directly reachable from a global reference are marked local. No write barrier is employed for global references.

In the sliding view algorithm we may treat global references in the following manner. 1) A mutator $T_i$ executes the following write-barrier in order to perform a global reference update, which includes the familiar snooping test:

1. $s := new$
2. if $Snoop_i$ then
3. mark $new$ as $local$

2) the collector, before carrying the fourth handshake, reads all global roots and marks the pointed objects $local$.

To gain some intuition that safety is indeed provided by this protocol we consider a global root $r$. $r$ is read by the collector before the fourth handshake and the object referenced is marked; so is any other object which is stored into $r$ by a mutator which still hasn’t responded to the fourth handshake, as the mutator has its $Snoop_i$ flag raised. We conclude that the only baleful scenario in which a reachable object (when reclamation commences) is mistakenly collected starts in a store into $r$ by a thread which has already responded to the fourth handshake. But it is an invariant which is kept by the algorithm, and is not broken by this protocol for global roots, that any object which is collected is unreachable from any thread (considering global roots as immediately reachable to the thread as well) after the thread has responded to the fourth handshake. Hence such a store is impossible in the first place, since it implies that the reachable object that has been mistakenly collected was already directly reachable from the thread which executed the store after it has responded to the fourth handshake. The detailed proof is in section B.

This protocol is effective when the number of global references is low relative to the number of modified slots that the collector has to process so that the constant time spent marking global roots does not dominate the overall running time of a cycle. Another advantage of it is the lighter write barrier. To conclude, we would opt treating global references using this protocol rather than as ordinary heap slots whenever the number of global references is relatively low or it is cumbersome to associate a dirty flag with each global reference.

5.4 Memory consistency

Throughout the paper we have assumed that the system conforms to sequential consistency constraints. In a sequential consistent system all memory accesses, carried out by all processors, are seemed to be serialized one after the other while preserving the order of instructions carried out by individual processors. However, some modern SMP systems do not provide sequential consistency but weaker consistency models in order to improve performance through processor level parallelism, speculative execution and non-uniform memory access. In this section we show how our algorithms can be adapted to weaker memory models. In particular, we show how our algorithms can operate on a platform which is processor ordered. Processor ordering is a memory model which is adhered to by contemporary platforms such Intel’s P6 processors’ family [7].
In a processor ordered system, like in a sequential consistent system, there is a linear sequence of all memory accesses carried out by all processes, however, it is not guaranteed that any two instructions that were carried out by a particular processor would appear in the linear sequence in the same order that they appeared in the processor’s program. Rather, only these orderings are guaranteed:

1. any two store instructions that are performed by a processor are bound to appear in the linear sequence in the same order as in the processor’s program.

2. if a processor contains in its program a load followed by a store then the store will follow the load in the linear sequence as well.

3. any two instructions that are performed by a processor which access the same consistency granule (see below) are bound to appear in the linear sequence in the same order as in the processor’s program.

4. a processor that does not communicate with other processor’s through shared memory (i.e., it doesn’t access locations that other processors access) may not witness that the instructions issued on its behalf are reordered.

The consistency granule of a system is an implementation dependent parameter that specifies the size and alignment of memory chunks for which rule (3) applies. Usually the consistency granule coincides with a cache line.

Implicit in the above definitions of sequential and processor ordering is the requirement that the linear sequences are sensible in the sense that they maintain the semantics of load operations, i.e., the result of a load from location X should be the value which is most recently stored into X in the linear sequence, or some prescribed initial value, if no such store exists.

Concisely, processor ordering amounts to sequential consistency with these two exceptions:

- “a load can pass a load” unless the two instructions address the same consistency granule and unless a non-communicating program can tell that the two instructions were performed out of order.

- “a load can pass a store” unless the two instructions address the same consistency granule.

Note that a non-communicating program can never tell whether such a reordering occurred (unless it can tell that another reordering of the form “a load can pass a load” occurred).

The most crucial aspect of adapting our algorithm to processor ordering is how to preserve the validity of the write barrier. Note that, aside from logging and snooping, the write barrier is comprised of a read-only part followed by a write-only part:

- **read-only part.** Read from s, then read from Dirty(s).

- **write-only part.** Optionally Write to Dirty(s), then write to s.

We note that under processor ordering the only pair of instructions that may be performed out of order are the load of s and the load of Dirty(s). It is easy to see that the algorithms do not operate correctly when such a reordering occurs. In order to prevent it, we may issue a synchronizing instruction between the loads. This is, however, a very expensive operation\(^\text{10}\).

If we have no knowledge on the specific mechanisms that allow this reordering to happen, that is, we don’t know which opportunities are exploited by the system to reorder instructions, then we

\(^{10}\text{It may involve flush high the processor’s pipeline and/or cache.}\)
don’t know as well how to eliminate these opportunities and we may rely only on the constraints provided by processor ordering in order to prevent the reordering. For example, we can allocate two adjacent bytes for the dirty flag where the two bytes reside on the same consistency granule. Then, in order to read the slot and then its flag we execute this code snippet:

```c
Update(WORD *s, ...) {
    register WORD slot_val;
    register BYTE *flag_addr,
        *dummy_addr,
        flag_val;

    (1) slot_val = *s; // LOAD slot
        flag_addr = calc_flag_addr(s);
        dummy_addr = flag_addr + 1;
    (2) *dummy_addr = MAGIC_NUM; // WRITE dummy
    (3) flag_val = *flag_addr; // LOAD flag
 ..."
```

The load of (3) cannot pass the store of (2) as they refer to the same granule. The store of (2)
cannot pass the load of (1) as a store may not pass a load. Thus we are guaranteed that the load
of the slot will precede the load of its flag is the linear sequence.

Although this protocol operates on any processor ordered system it is inefficient since it requires
doubling the space needed for the already space demanding dirty flags and it incurs an additional
write access on each invocation of the barrier.

However, in practice, we can identify the origins of reordering and therefore we can take advan-
tage of this knowledge and efficiently eliminate reordering when needed. We consider as an
example a PowerPC system with a MESI cache protocol. MESI is a cache protocol that requires
the processors to gain exclusive ownership over memory locations prior to modifying them. At the
time a location is owned it may not be cached neither for reading nor for writing by any processor
other than the owner. Thus, it is easy to see that the requests which are serviced by the cache
protocol adhere to sequential consistency. It follows that reordering can only emanate from the
processor itself, which issues its external cache requests in an out of order fashion. In order to
eliminate the out of order execution of the loads in the write barrier it suffices to guarantee that
the processor presents these load requests in their original order to the cache mechanism. This is
achieved by creating a faked dependency among the two loads, fooling the processor to believe that
it must carry out the first fetch prior to starting the second one. Such a dependency can be created
using this code fragment:

```c
 void Update(WORD *s,...) {
    register WORD val;
    register BYTE *flag_calc_addr;
    register BYTE flag_val;

    val = *s;
    flag_calc_addr = (val & 3) + calc_flag_addr(s);
    flag_value = *flag_calc_addr;
 }
```

In the code fragment we assume that a pointer value is aligned on a four-byte boundary, such that the expression \((\text{val} & 3)\) is bound to equal zero and \(\text{flag\_calc\_addr}\) evaluates to
calc_flag_addr(s). However, the processor does not possess this knowledge a priori and it is fooled to believe that in order to load the flag it must first know the value of the slot. The extra price paid is two additional arithmetic operations (perhaps a single operation on some architectures.)

We admit that an aggressively speculative processor could have executed the second load prior to the first load if it is designed to predict the results of load operations and can accordingly execute code speculatively based on the predications. We know not of a processor which behaves in this manner.

We now turn to the lighter problems of snooping and logging under weak memory constraints.

Snooping requires that a mutator would first execute the store proper into the slot and only then would load its Snoop flag. Under processor ordering the load may pass the store. However, we care that these two instructions would not be reordered only in order to snoop stores into slots by mutators which still haven’t responded to the first handshake. Otherwise, i.e., between the first and fourth handshakes, the flag is continuously raised and the test is bound to succeed even if the instructions are reordered (of course, we assume that a soft handshake synchronizes the mutator’s view of the memory with that of the collector.) Instead of combating this reordering we may simply carry out an additional handshake before the one that used to be the first handshake. In the additional handshake we would raise the Snoop flags.

In the context of logging we have relied on the order of store operations by the mutator, i.e., first logging the value and then the slot, to allow the collector to read a mutator’s buffer reliably without stopping it (see section 5.2.)

We note that under processor ordering the collector may execute the loads (of the slot and value parts of a record in a mutator’s buffer) in any order for if the slot field of the record does not contain null, then the store into the slot field by the mutator must have preceded the collector’s load in the linear sequence. But that implies that the mutator’s store into the value field precedes both collector’s load operations in the linear sequence, providing the collector with an accurate account of both value and slot parts of the record.

Under weaker memory models than processor ordering we eschew the problem of collector’s perceived partial logging by reading the buffers in an additional soft handshake.

6 A Supplemental Sliding View Tracing Algorithm

We chose to tackle the problems of cyclic data structures and stuck reference count fields using a supplemental concurrent mark&sweep algorithm that reclaims those cyclic garbage structures and reinstates stuck reference count fields. The algorithm is designed to be interoperable with the sliding view algorithm meaning that it is possible to decide on a cycle by cycle basis which algorithm should be invoked and that the code for updating a pointer is common to both algorithms. However, we do have to change the New operation in order to support object coloring which is needed for the tracing algorithm.

6.1 Tracing using a sliding view

This section demonstrates how it is possible to use a sliding view in order to develop a tracing procedure which assures that any reachable object at the end time of the sliding view is marked and therefore not reclaimed later.

The basic mark&sweep algorithm operates by stopping all threads, marking any object which is directly reachable (either from a local or a global reference) and then recursively marking any
object which is pointed by a marked object. Then, any object which is not marked is swept, i.e., reclaimed. Finally, mutator threads are resumed.

Concurrent mark&swEEP collectors perform some, or all, of the above steps concurrently with mutators. SnapSHoT at the beginning [18, 21] mark&swEEP collectors exploit the fact that a garbage object remains garbage until the collector recycles it, i.e., being garbage is a stable property. Thus, snapshot at the beginning operates by:

1. stopping the mutators,
2. taking a snapshot of the heap ans roots,
3. resuming the mutators,
4. tracing the replica,
5. sweeping all objects in the original heap whose replicated counterparts are unmarked. These reclaimed objects must have been garbage at the time the snapshot was taken and hence they are garbage also when the collector eventually frees them.

We take the idea of “snapshot at the beginning” one logical step further and show how it is possible to trace and sweep given a “sliding view at the beginning”.

Suppose we are given a scan $\sigma$ and a corresponding sliding view $V_\sigma$. Using the scan, we want to deduce which objects are garbage at $End(\sigma)$. To that end, we ask ourselves what is the value of a slot $s$ at time $End(\sigma)$. The trivial answer is of course either $V_\sigma(s)$ or any other value which has been stored into $s$ between $\sigma(s)$ and $End(\sigma)$. If we want to trace any object which is reachable at time $End(\sigma)$ it suffices to start tracing from a root set which includes the true root set at $End(\sigma)$ and adopt the following tracing discipline: whenever a slot $s$ is traced, trace through all of the candidate values it assumed at $End(\sigma)$, i.e., proceed tracing through $V_\sigma(s)$ and through any value that has been stored into it in the interval $\sigma(s)$ to $End(\sigma)$. These stored values are known to the collector since they are snooped by mutators, i.e., the mutators keep a record of any such value which might be stored in the specified interval.

It still remains to identify a set of pointers that includes the true root set at $End(\sigma)$. This can be done using the same mechanism that was employed in the reference counting sliding view algorithm: “snooping” and the fourth handshake that marks thread states. Any local reference that exists at $End(\sigma)$ is either still existent at the time of the fourth handshake or is discarded before the thread responds to the fourth handshake. If it is discarded without being stored into a heap slot (and thus snooped) then it has no contribution to reachability after the fourth handshake ends and we may simply ignore it (although it is a valid local reference at $End(\sigma)$).

We thus arrive at the following garbage collection algorithm:

- A mutator $T_i$ executes the following write-barrier in order to perform a heap slot update, which includes the snooping test:
  
  1. $s := new$
  2. if $Snoop_i$ then
  3. mark $new$ as local

- A collection cycle contains the following stages:
  
  1. the collector raises the $Snoop_i$ flag of each thread. This indicates to the mutators that they should start snooping.
2. the collector computes a scan $\sigma$ and a corresponding sliding view, $V_\sigma$, concurrently with mutators’ computations.

3. each thread is then suspended (one at a time) and its $\text{Snoop}_B$ flag is turned off. Each object which is directly reachable from the thread is marked local. The thread is then resumed.

4. The collector traces the heap according to the image of it contained in $V_\sigma$. The starting point for the trace is all objects which are marked local.

5. After tracing is completed, any object which is not marked and which has been allocated by thread $T_i$ before $T_i$ was stopped in order that its state be scanned (in stage (3) above), is garbage.

Note that we can reason only regarding objects which were allocated prior to the handshake of stage (3). Since sweeping occurs after the handshake we need devise a mechanism that prevents the collector from collecting objects that were allocated after the handshake. We use a variant of the color toggle trick, first introduced in [31]. It is assumed that every object has a color field associated with it. The field can take on three different values, say 0, 1 and 2. The value of 2 is interpreted as the color blue, which is assigned to unallocated objects. In the initial cycle, the color white, which is the color of objects which still haven’t been traced, is represented by zero and black is represented by one. Black is the color of objects which have been already traversed in the trace. On each subsequent cycle the black and white colors are toggled, i.e., the meanings of zero and one are reversed.

A mutator toggles during the handshake of stage (3) the color using which it colors newly allocated objects and the collector reverses the meaning of black and white prior to starting a new trace.

6.2 The algorithm

The tracing algorithm uses exactly the same mechanisms used in the reference counting sliding view algorithm in order to implicitly compute a sliding view based on which collection decisions are made. Specifically, it uses the same four handshakes. Only the operations carried out in the fourth handshake are modified in order to support the subsequent tracing and sweeping, rather than reference counting. Let us elaborate on the tracing and sweeping stages.

**Tracing.** After the consolidation stage the collector starts tracing according to the sliding view associated with the cycle. When in need to trace through a slot the collector tries to determine its value in the sliding view as was done in the previous algorithms, i.e., by first reading the slot and then its flag. Determining the slot is successful if the flag is off. In that case the value read from the slot is the slot’s value in the cycle’s sliding view. If determining is not successful, then the collector retrieves the slot’s value from the threads’ buffers. This is done in phases: first, the collector tries to determine and then trace through any slot that it can. Then, when all the slots which need to be traced are all undetermined slots, it reads threads’ buffers, resolves the slots and resumes tracing. Resolving a slot means looking-up the value mutators have associated with it in their buffers. Resolution is always successful since it is guaranteed that any undetermined slot is logged by some mutator prior to the time the collector inspects mutators’ buffers.

Since any undetermined slot is due to appear in some buffer when trying to resolve it each phase contributes to the progress of tracing. Additionally, the graph induced by the sliding view is finite, so tracing is bound to complete after a finite number of phases. We believe that in practice only handful phases will be actually needed in order to complete tracing since if the collector traces fast
Procedure \texttt{New(size: Integer) : Object}
begin
  1. Obtain an object \(o\) from the allocator, according to the specified \(size\).
  2. \(o\.color := \text{AllocColor}\);
  3. \(New_i := New_i \cup \{o\}\)
  4. return \(o\)
end

Figure 19: Allocation code that supports tracing cycles

enough then it reveals quickly the picture of the heap contained in the sliding view. If, on the other hand, it falls behind a mutator which rapidly changes the heap, then it learns about the contents of the sliding view from the mutator’s buffer in few phases as well. Thus, sustained tracing can occur only when the collector is running almost in unison with the mutator, falling just behind it, as they compete for the same slots in memory, which is an improbable scenario.

As tracing proceeds, the collector incrementally computes the \(rc\) field for each object. Eventually, when tracing is done, the \(rc\) field has the same semantics which are expected by a reference counting cycle, i.e., it equals the asynchronous reference count according to the sliding view associated with the cycle (disregarding pointers from garbage objects).

\textbf{Sweeping.} Finally, the collector proceeds to reclaim garbage objects by sweeping the heap. As said, the algorithm can infer whether an object is garbage or not only if it has been allocated prior to the fourth handshake. Thus, we need a mechanism to prevent the collector from sweeping objects which have been allocated after the handshake. We use a color toggle scheme in order to prevent the reclamation of such objects.

Each thread has a variable, denoted \texttt{AllocColor\(_i\)}, that holds the color the thread has to \texttt{color}, i.e., assign, to the \texttt{color} field of newly allocated objects. The variable is toggled between two dichotomic colors, \texttt{black} and \texttt{white}, which are interpreted by the collector as “marked” and “not marked” respectively.

When a thread responds to the fourth handshake we assign the current \texttt{black} color to the \texttt{AllocColor\(_i\)} variable. Thus, during tracing and sweeping the mutator colors newly allocated objects \texttt{black}. During sweeping, the collector considers each object in heap. If the object is \texttt{black}, then it is retained. If it is colored \texttt{blue}, then it is ignored. Otherwise, the object is \texttt{white}. In that case the collector reclaims the object by coloring it \texttt{blue} and passing it back to the allocator.

Thus, when sweeping is over, the heap contains only \texttt{black} or \texttt{blue} objects since any object which had been \texttt{white} was turned \texttt{blue} and mutators color newly allocated objects \texttt{black}. Before starting the tracing of the next cycle the collector toggles the values of \texttt{black} and \texttt{white} variables, so all objects allocated prior to the next cycle’s fourth handshake are considered “unmarked”.

We now proceed to specify the algorithm’s pseudo-code.

6.3 Mutator code

As required, the algorithm uses the same write-barrier used in the reference counting sliding view algorithm. The code for the \texttt{Update} procedure is given in figure 11.

The \texttt{New} procedure is modified to support both tracing and reference counting cycles. \texttt{New} carries out the mutator’s part in the object coloring protocol. The modified procedure is given in figure 19.

37
Procedure **Tracing-Collection-Cycle**
begin
1. Initiate-Collection-Cycle
2. Clear-Dirty-Marks
3. Reinforce-Clearing-Conflict-Set
4. Consolidate-For-Tracing
5. Mark
6. Sweep
end

Figure 20: Tracing Alg.—Collector Code

### 6.4 Collector Code

The code for a tracing collection cycle is given in figure 20. Procedures **Initiate-Collection-Cycle**, **Clear-Dirty-Marks** and **Reinforce-Clearing-Conflict-Set** are the same ones used in the cycles of the reference counting sliding view algorithm. They are given in figures 13, 14 and 15, respectively. They serve for the same purpose here as well: after they are executed logging and determining of slots is consistent.

**Consolidate-For-Tracing.** This procedure, given in figure 21, is the counterpart of procedure **Consolidate** from the reference counting algorithm. As such, it carries out the fourth handshake during which thread local states are marked and the buffers accumulated between the first and fourth handshakes are retrieved for consolidation. However, note the differences from **Consolidate**, which are highlighted with an asterisk in front of the relevant lines of code: the values of *black* and *white* are toggled; the *AllocColor* variable of each thread is toggled, signaling to the mutator that any creation of objects after the fourth handshake and until sweeping is over should color a newly created object *black*. Another thing to note is the omission of the addition of the *New* sets to *ZCT*.

**Procedure Mark** (figure 22) implements the tracing stage of the algorithm. Tracing proceeds according to the graph induced by the sliding view associated with the cycle and starting from objects in *Locals*.

Recall from the reference counting sliding view algorithm that after taking the fourth handshake the collector may coherently try to *determine* what is a slot’s value in the sliding view of the cycle. It relies on the determined value to tell whether it has succeeded or failed in determining. In case it succeeds, it simply need continue tracing from the object pointed by the determined slot. Otherwise, it is guaranteed that some thread has recorded the undetermined slot’s value in its buffer. The collector tries to determine and trace more and more slots, until all slots that have to be traced through are all undetermined. Then, it *resolves* those undetermined slots by looking up their associated values in the threads’ buffers.

The collector uses a hash table or a similar data structure in order to store and retrieve the values which mutators have associated with slots. We assume that the hash table supports these operations:

- **Hash-Clear.** Clear the hash table.
- **Hash-Insert**(*s*, *v*). Associate *v* with *s*.
- **Hash-Lookup-And-Remove**(*s*). Lookup the value associated with *s*. Remove the association for *s* and return the value which has been read.

Initially, the collector clears the hash table and fills in the associations contained in *Hist*[i] (*i.e.*, the digested history of threads’ modifications to heap slots between the first and fourth handshakes).
Procedure \textbf{Consolidate-For-Tracing}
\begin{verbatim}
begin
  // initila y black = 1 and white = 0
*1. black := 1 − black
*2. white := 1 − white
3. local Temp := ∅
4. Locals_k := ∅
5. for each thread $T_i$ do
6.   suspend thread $T_i$
*7. AllocColor$_i$ := black
8. Snoop$_i$ := false
   // copy and clear snooped objects set
9. Locals$_k$ := Locals$_k$ ∪ Locals$_i$
10. Locals$_i$ := ∅
    // copy thread local state.
11. Locals$_k$ := Locals$_k$ ∪ States$_i$
    // clear thread local ZCT.
12. New$_i$ := ∅
    // copy local buffer for consolidation.
13. Temp := Temp ∪ Buffer$_i[1...CurrPos; i − 1]$
    // clear local buffer.
14. CurrPos$_i$ := 1
15. resume thread $T_i$
    // consolidate Temp into Hist$_{k+1}$.
16. Hist$_{k+1}$ := ∅
17. local Handled := ∅
18. for each $(s, v) \in Temp$
19.   if $s \notin Handled$ then
20.      Handled := Handled ∪ \{s\}
21.      Hist$_{k+1}$ := Hist$_{k+1}$ ∪ \{(s, v)\}
end
\end{verbatim}

Figure 21: Tracing Alg.—Procedure \textbf{Consolidate-For-Tracing}
Figure 22: Tracing Alg.—Collector Code—Procedure Mark

After each non-terminal tracing phase, when the collector can no longer proceed tracing through determined slots but still there are undetermined slots to trace through, the collector reads the portions of the thread buffers which have accumulated since the read of the last phase and populates the hash table with the associations contained therein. Then, it looks up any undetermined slot in the hash table and finds its associated value. The associated value is then traced through. Since a slot is traced at most once, a slot which has been looked up will not be needed in the future hence the collector deletes the association of $s$ jointly with looking it up.

The collector knows which portions of the buffer have been accumulated since the last tracing phase by using the thread specific marker $\text{ScannedPos}_i$ which equals the value of $\text{CurrPos}_i$ at the time the thread buffer was most recently read, during the previous phase.

**Procedure Trace** (figure 23). Actual tracing is carried out by this procedure. The procedure takes two arguments: a reference to an object to trace through and a reference count increment value. An object is traced only if its color is white, i.e., it was not traced before. If this is indeed the case then the reference count field of the object is reset and it is colored black. Then, the collector tries to determined each slot contained in the object and trace through it. If a slot is determined then the collector carries out line (7) which traces recursively through the determined value, which is the value of the slot at the sliding view associated with the cycle. If a slot is undetermined then line (9) adds it to the $\text{Undetermined}$ set where it will wait until its resolution.

It is important to note that the trace cannot be interrupted by objects which are allocated black by procedure New. Let us explain this point. The collector traces through the graph induced by the sliding view and the corresponding scan of the cycle. The scan is complete before the fourth handshake starts hence it cannot reference an object which is created black because a thread may leave a newly allocated object blackened only after responding to the fourth handshake.

The reference count increment argument signifies whether Trace has been invoked for $o$ by
Procedure Trace(\texttt{Object}, \texttt{rcInc}: \texttt{Integer})
begin
  1. if \texttt{o.color = white} then
  2. \texttt{o.color := black}
  3. \texttt{o.rc := 0}
  4. for each slot \texttt{s} of \texttt{o} do
  5. \texttt{v := read(s)}
  6. if \texttt{¬Dirty(s)} then
  7. \texttt{Trace}(v, 1)
  8. else
  9. \texttt{Undetermined := Undetermined ∪ \{s\}}
10. \texttt{o.rc := o.rc + rcInc}
end

Figure 23: Tracing Alg.—Collector Code—Procedure Trace

Procedure Sweep
begin
  1. for each pair \langle s, v \rangle ∈ Hist_{k+1} do
  2. Let \texttt{o} be the object containing \texttt{s}
  3. if \texttt{o.color = white} then
  4. \texttt{Hist}_{k+1} := \texttt{Hist}_{k+1} - \{\langle s, v \rangle\}
  5. \texttt{ZCT}_{k+1} := ∅
  6. Let \texttt{swept} point to the first
    object in the heap
  7. while \texttt{swept} does not point pass the heap do
  8. if \texttt{swept.color = white} then
  9. clear the slots and flags of \texttt{swept}
10. \texttt{swept.color := blue}
11. return \texttt{swept} to the allocator
12. else if \texttt{swept.color = black ∧ swept.rc = 0} then
13. \texttt{ZCT}_{k+1} := \texttt{ZCT}_{k+1} ∪ \{\texttt{swept}\}
14. advance \texttt{swept} to the next object
end

Figure 24: Tracing Alg.—Collector Code—Procedure Sweep

virtue of being pointed from a heap slot or rather by a local reference. In the latter case, no
adjustment to \texttt{o.rc} is needed, while in the former \texttt{o.rc} should be incremented by one. Thus,
procedure Mark passes 0 for this argument when tracing local objects (in lines (7-8)), while all
other invocations pass 1 as they are due to heap slot references to the traced object.

Sweeping is carried out by procedure Sweep (figure 24). The first step it takes is to eliminate
from \texttt{Hist}_{k+1} any records of slots that it is about to reclaim. This stage is needed in order that
the next cycle will not adjust \texttt{rc} fields incorrectly due to the slot, trying to determine its value etc.
Such a slot may exist since the algorithm is capable of reclaiming objects which are reachable (and
therefore modifiable) between the first and fourth handshakes.

Sweeping then proceeds in the following manner: any object which is colored \texttt{black} and has a
zero computed reference count field is added to the ZCT of the next cycle (anticipating a reference
counting cycle.) \texttt{White} objects are returned to the allocator not before being colored \texttt{blue}. \texttt{Blue}
objects are ignored.
7 Conclusions

We have presented a reference counting garbage collector with an explicit attempt to make it suitable for a multiprocessor. The algorithm uses extremely low synchronization overhead: the barriers for modifying a reference and the barrier for creating a new object are very short and in particular, require no strong synchronized operations such as a compare-and-swap instruction. Furthermore, there is no particular point in which all threads must be suspended simultaneously. Instead, each thread cooperates with the collector by being shortly suspended four times during each collection cycle. In three of these four handshakes, the time of suspension is just enough to allow a short operation that does not depend on the heap structure or the local state of the threads. One of the four handshakes requires reading the local roots of the thread. Thus, the overall overhead is small.

The two main new ideas presented in this work are first, the clever mechanism for logging of reference modifications, which requires no synchronization, yet introduces no inconsistencies due to race conditions, and second, the fact that a fuzzy snapshot of the heap, which we denote the sliding view, is enough to get an approximation of the reference count and perform the garbage collection.

Note that as in the previous work of DeTreville [13], our algorithm is based on the mutators logging information about the modifications they apply to heap references. However, in our algorithm, a thread takes a record of a modification at most once per slot per cycle (as opposed to always keeping a record) and the heavy synchronization incurred due to the logging action is completely eliminated.

In order to reclaim cyclic structures and to reinstate stuck reference count fields (anticipating that a real implementation will use a relatively small field in order to represent reference counters) we have presented a concurrent, scalable, tracing collector. The tracing collector relies on the same notion of a sliding view as its reference counting counterpart and thus it is inter operable with the reference counting sliding view algorithm. In particular, the tracing collector as well never stops all mutators simultaneously and it uses the same write barrier used by the reference counting algorithm.

We are now preparing to implement the algorithm and expect to report experimental results in the near future.

References


A Snapshot Algorithm Correctness Proofs

This appendix contains safety and progress proofs for the snapshot algorithm. In the correctness proofs we abandon our assumption about the inexistence of global roots. Instead, we assume that during the handshake of a cycle, when all threads are stopped, the collector marks any object which is directly reachable from a global root as reachable.

A.1 Safety

In this section, we will prove that the algorithm recycles an object only if it is garbage at the time it is recycled. Actually, an object is recycled only if it garbage at the time the conceptual snapshot is taken. Let us first define precisely this moment at which the conceptual snapshot $R_k$ is taken:

Definition A.1 Let $HS_k$ be the earliest time at which all dirty marks have been cleared during the execution of procedure Read-Current-State in collection cycle number k.

We assume that at system initialization, before any mutator has taken any step, there occurs an initial garbage collection cycle. As can easily be seen, this cycle leaves all data structures that are carried across cycles (e.g., reference counters, ZCT) untouched, so there is no loss of generality in our assumption. We use this assumption in order to simplify the correctness proofs of the base cases of inductive claims. So, $HS_0$ happens at system initialization.

We further define $HS_{-1} \equiv HS_0$. This definition as well simplifies the proof of claims that depend on the two preceding cycles.

Ultimately, in terms of safety, we would like to prove the following:

Theorem A.1 (Safety) An object is recycled during cycle $k$ only if it is unreachable at $HS_k$.

A.1.1 Roadmap of the proof

Due to the cycle-by-cycle nature of the algorithm its properties are proved by induction on the cycle number. For convenience, we will assume that there is a garbage collection cycle numbered zero scheduled at system startup. This assumption facilitates the proof of the induction basis and does not involve any loss of generality.

Most lemmas are interdependent meaning, for example, that we prove lemma $X$ correct at cycle $k$ provided lemma $Y$ is correct at cycle $k - 1$. In order to make clear the relation between the claims and to demonstrate that there is no circular logic in the proof we provide herein a complete description of the interdependencies among the claims. We denote by $Li_k$ the assertion of lemma $i$ for cycle $k$.

Here is a short description for each of the claims involved:

- SafetyTheorem$_k$: An object is collected during cycle $k$ only if it is garbage at $HS_k$. 


• LA.2: If a slot is modified between \(HS_{k-1}\) and \(HS_k\) then only and exactly the value it assumed at \(HS_{k-1}\) is recorded. No information is recorded for slots which are not modified.

• LA.3: The collector can distinguish, during cycle \(k\), whether it is reading a slot's value which was current at \(HS_k\), or, that the slot has been overwritten since.

• LA.4: The collector finds out, eventually, in procedure Fix-Undetermined-Slots, what are the values of undetermined slots.

• LA.5: Just before the invocation of Reclaim-Garbage of cycle \(k\), the \(rc\) field of each object equals the heap reference count of the object at \(HS_k\).

These are the dependencies between the claims:

• the basis for each claim, i.e. its correctness for cycle zero is proven independently for each claim.

• \(LA.2_k \iff \bigwedge_{j \leq k} SafetyTheorem_j\)

• \(LA.5_k \iff \bigwedge_{j \leq k} (LA.5_j \land SafetyTheorem_j) \land LA.2_k \land LA.4_k\)

• \(SafetyTheorem_k \iff LA.5_j\)

### A.1.2 Update protocol properties

Consider any slot \(s\) which is modified between \(HS_{k-1}\) and \(HS_k\). The snapshot algorithm requires us to adjust \(rc\) fields due to \(s\) by decrementing the \(rc\) field of \(s@HS_{k-1}\) and incrementing the \(rc\) field of \(s@HS_k\). The first part of the requirement, decrementing \(s@HS_{k-1}\), is implemented by letting the mutators record the identity of \(s@HS_{k-1}\) into their buffers. Thus, we would like to prove for any such modified slot \(s\) that only and exactly \(s@HS_{k-1}\) is associated with \(s\) by the mutators.

If \(s\) is not modified between the current and previous cycles, then we want to show that no record of \(s\) is kept.

The lemmas in this section prove that the algorithm possesses these properties.

**Lemma A.1** Let \(s\) be a slot and let \(t\) be a time point satisfying

1. \(HS_{k-1} \leq t < HS_k\), and
2. \(Dirty(s)@t = false\), and
3. No update of \(s\) is occurring at \(t\).

Let \(UPD(s)\) be the set of all update operations applied to \(s\) which are scheduled between \(t\) and \(HS_k\). Let \(ASSOC(s)\) be the set of values which are associated with \(s\) by the operations in \(UPD(s)\).

It holds that:

1. \(UPD(s) = \emptyset \implies ASSOC(s) = \emptyset\)
2. \(UPS(s) \neq \emptyset \implies ASSOC(s) = \{s@t\}\)
The first claim is quite trivial since a value is associated with $s$ only as part of an update. Since no update is scheduled, no value is associated.

Suppose that $s$ is indeed modified between $t$ and $H S_k$. Consider the set of threads, denoted $P$, that apply the subset of operations of $UPD(s)$ which read the value of $Dirty(s)$ as $\text{false}$ in line (2) of procedure $Update$, while updating $s$. $P$ is not empty since some thread modifies $s$ ($UPD(s)$ is non-empty) and the dirty flag is off at $t$.

Consider a thread $T_i \in P$. We want to show that when $T_i$ executed line (1) of procedure $Update$ it read the value of $s$ at $t$. Suppose that it did not. Let $\tau$ be the time at which thread $T_i$ executed line (1). Then some thread $T_j$ must have executed a store to $s$ after, or at, $t$ and before $\tau$. Since there was no updates occurring at $t$ and since the store is the last instruction of an update operation we conclude that the entire update operation by $T_j$ has started after, or at, $t$ and ended before $\tau$. Just before $T_j$ executed the store in line (6) the value of $Dirty(s)$ must have been $\text{true}$ either by line (5) or by virtue of another thread (note that the collector resets the flag only during the next cycle) so $T_i$ should have read a value of $\text{true}$ from $Dirty(s)$, in line (2), which was not the case. A contradiction. We conclude that $T_i$ must have associated $s@t$ with $s$. So we have

$$\{s@t\} \subseteq \text{ASSOC}(s)$$

According to the code, any thread $T_i \notin P$ would not associate any value with $s$ thus

$$\text{ASSOC}(s) = \{s@t\}$$

For a given history buffer $H$ (be it collector or mutator maintained set) and a slot $s$ we define the set of values that $H$ associates with $s$, denoted by $VAL(H; s)$, as:

$$VAL(H; s) \overset{\text{def}}{=} \{v | (s, v) \in H\}$$

For brevity we write $s \in H$ meaning $\exists v : (s, v) \in H$.

The next lemma summarizes and proves the desired properties of the write-barrier employed by the algorithm. We need some definitions first:

- We say that an object $o$ is allocated for cycle $k$. If some thread has allocated $o$ between $H S_m$ and $H S_{m+1}$, where $m < k$. And there has not been a cycle $l$, where $m \leq l < k$ during which $o$ was reclaimed.
- $o$ is allocated new for cycle $k$ if $m = k - 1$ in the above definition.
- Otherwise, we say that $o$ is allocated old for cycle $k$. 
- We say that a slot is allocated (new/old) for cycle $k$ if its containing object is allocated (new/old) for cycle $k$.
- We abbreviate and say that a slot or an object are new (old) to a cycle meaning that the slot or the object are allocated new (old) for that particular cycle.

Lemma A.2 Let $s$ be an allocated slot for cycle $k$. Then:

1. if $s$ is new to cycle $k$ and is modified between $H S_{k-1}$ and $H S_k$ then
   $$VAL(H ist_k; s) = \{\text{null}\}$$

48
2. If \( s \) is old to cycle \( k \) and is modified between \( HS_{k-1} \) and \( HS_k \) then
\[
VAL(Hist_k; s) = \{s@HS_{k-1}\}
\]

3. Otherwise (\( s \) is not modified between \( HS_{k-1} \) and \( HS_k \)),
\[
VAL(Hist_k; s) = \emptyset
\]

**Proof.** The lemma vacuously holds for \( k = 0 \) since there are no slots which are modified during the interval \( HS_{-1} \) to \( HS_0 \).

We now show that the lemma holds for cycle \( k > 0 \) provided safety theorem hold for previous cycles.

Suppose \( s \) is new to cycle \( k \). Let \( \tau \) be the time at which the object \( o \) containing \( s \) was allocated. Let \( j < k \) be the cycle during which the object \( x \) that most recently contained \( s \) was reclaimed, or 0 if no such cycle exists. Applying the safety theorem to cycle \( j \) we know \( x \) was unreachable at \( HS_j \). Thus, no thread could have accessed \( s \) from \( HS_j \) until \( \tau \). In addition, if \( j > 0 \), when \( x \) was recycled, null was assigned to \( s \), in line (4) of procedure \text{Collect}. Finally, as all dirty flags are cleared while the threads are halted, we have \( Dirty(s)@HS_j = false \). Since these values must remain in effect until time \( \tau \) we can apply lemma A.1 to \( s \) and \( \tau \) yielding that either claim (1) or (3) hold, depending on whether \( s \) has been modified prior to \( HS_j \).

If, on the other hand, \( s \) is old to cycle \( k \) then we have \( Dirty(s)@HS_{k-1} = false \) and no update of \( s \) is occurring at \( HS_{k-1} \). Thus, we can apply lemma A.1 to \( s \) and time \( HS_{k-1} \) yielding that either claim (2) or (3) hold, depending on whether \( s \) has been modified prior to \( HS_j \).

\[\Box\]

### A.1.3 Determined vs. undetermined slots

We say that the collector **determines** the value of a slot \( s \) if during the \text{Update-Reference-Counters} procedure it reads the value \( v \) from \( s \) (in line (3)) and then sees \( Dirty(s) = false \) (in line (4)). Such a slot is **determined**, as opposed to **undetermined** slots which are taken care of by the collector in procedures \text{Read-Buffers} and \text{Fix-Undetermined-Slots}. The following lemma tells us that if the collector determines the contents of a slot then it has indeed read its contents as they were at the time the recent conceptual snapshot was taken.

**Lemma A.3 (Determined Slots)** If the collector determines \( s \) to contain \( v \) during cycle \( k \) then \( v = s@HS_k \).

**Proof.** Let \( s \) be a determined slot. As all dirty slots are cleared when the threads are stopped we have \( Dirty(s)@HS_k = false \). Let \( \tau \) be the time at which the collector executed line (4) of \text{Update-Reference-Counters}. At time \( \tau \) the flag was still off. Thus, no line (5) of procedure \text{Update} was scheduled in the interval \( HS_k \) to \( \tau \). Hence the later store from line (6) of \text{Update} hasn’t been scheduled in this interval as well. This means that \( s \) remained unchanged from \( HS_k \) to \( \tau \). This interval includes the time at which the collector read the value of \( s \), in line (3) of \text{Update-Reference-Counters}. Hence the collector read \( s \) to contain \( s@HS_k \).

\[\Box\]
What happens when the collector does not succeed determining a slot? A slot is undetermined if the collector senses that its flag is raised during Update-Reference-Counters. The only reason for the flag to be raised is that some thread, say $T_i$, has applied line (5) of procedure Update to the flag (i.e., raised it.)

Since updates are non-interruptible, $T_i$ has executed the preceding lines of (3) and (4) of the same invocation after $H S_k$, i.e., $T_i$ has stored the pair $\langle s, s@H S_k \rangle$ into its buffer and incremented $C u r r P o s$; prior to raising the flag. Thus, when the collector would process $B u f f e r_i$ during Read-Buffers it will see the logged pair $\langle s, s@H S_k \rangle$ in $T_i$’s buffer (s@H S_k is associated with $s$ according to lemma A.2.) and thus the pair will be added to the set $P e e k_k$.

We conclude the following:

**Lemma A.4 (Undetermined Slots)** If the collector does not determine a slot $s$ in cycle $k$ then

$$V A L(P e e k_k; s) = s@H S_k$$

### A.1.4 Linking $rc$ field with reference count

In this section we show that the $rc$ fields that the algorithm computes equal, eventually, the heap reference counts at the time the conceptual snapshot is taken. We need some definitions first.

**Definition A.2** Let $E N D_k$ denote the time at which cycle $k$ has ended. That is, $E N D_k$ is the earliest time at which all instructions of cycle $k$ have already been scheduled.

**Definition A.3** Let $C O L L E C T_k$ be the time at which the invocation of Fix-Undetermined-Slots, during cycle $k$, is complete. The collector starts executing Reclaim-Garbage after, or at, $C O L L E C T_k$.

The following lemma proves that the value of the $rc$ field of each object, after the collector has finished adjusting $rc$ fields due to all logged modifications, i.e., when procedure Reclaim-Garbage starts its operation, equals the object’s heap reference count at time $H S_k$.

**Lemma A.5 (Meaning of The $rc$ Field)** $o.rc@C O L L E C T_k = R C(o)@H S_k$ for any object $o$ which is allocated at $H S_k$.

**Proof.** The claim holds for $k = 0$ since there are no objects which are allocated at $H S_0$.

For $k > 0$, we prove that the lemma holds for cycle $k$ provided this lemma and the safety theorem both hold for previous cycles.

It’s enough to show that the algorithm adjusts $rc$ fields due to each slot correctly. If $s$ does not change after $H S_{k-1}$ and before $H S_k$ then, by lemma A.2, $s$ will not be logged and there will be no modifications to any $rc$ fields due to $s$.

Let’s consider the cases in which $s$ does change. We have to show that the $rc$ field of the object that $s$ was referring to at $H S_{k-1}$ is decremented. Likewise, we have to show that the value of the object that $s$ was referring to at $H S_k$ is incremented. $s$ is in exactly one of these states at $H S_k$: allocated old, allocated new, non-allocated.

**Decrementing old slots:** If $s$ is old for cycle $k$ then $s$ is changed by mutators, and not by the collector (by deleting it.) Due to lemma A.2 $H i s t_k$ will contain the pair $\langle s, s@H S_{k-1} \rangle$. $H i s t_k$ will not contain elements associating $s$ with a value other than $s@H S_{k-1}$. During the operation of Update-Reference-Counters, when the pair $\langle s, s@H S_{k-1} \rangle$ is considered, the $rc$ field of $s@H S_{k-1}$ is decremented, as desired.

50
Decrementing new slots: Let $s$ be a new slot for cycle $k$. According to lemma A.2 either $\mathit{null}$, or no value at all, are associated with $s$. Thus, there are no decrements that occur due to $s$ during cycle $k$. Let us explain why this is the desired behavior.

If $s$ is new for cycle $k$ then either $s$ becomes allocated for the first time, or it was part of an object $o$ which was recycled during cycle $j$, where $j < k$.

In the former case, we know that $s$ was initialized to $\mathit{null}$ and its dirty flag was off at system startup. Also, no thread could have accessed $s$ at $HS_{j-1}$, since it was not a part of a reachable object (or any object) at that time. Thus, $s@HS_{k-1} = \mathit{null}$ and therefore no $rc$ field should be decremented due to $s$ during cycle $k$.

In the latter case, according to the safety theorem applied to cycle $j$, $o$ is not reachable at $HS_j$. Thus, the collector has exclusive access to $s$, during cycle $j$. It follows that the collector may decrement the $rc$ field of the object pointed by $s$ and clear $s$ without being interfered by mutators’ actions, all part of the operation of $\mathit{Collect}$ during cycle $j$. If $j < k - 1$ then $s@HS_{k-1} = \mathit{null}$, thus there is no “old” value to decrement.

Otherwise, $j = k - 1$. In this case, the collector decrements the $rc$ field of $s@HS_{k-1}$ during cycle $k - 1$, when it reclaims $o$. An object is reclaimed only if its $rc$ field drops to zero. $\mathit{Reclaim}$-$\mathit{Garbage}$ and $\mathit{Collect}$ can only reduce the value of an $rc$ field. Thus, there is a single point during the operation of $\mathit{Reclaim}$-$\mathit{Garbage}$ at which $o.rc = 0$. Therefore $o$ is reclaimed exactly once and likewise the $rc$ field of $s@HS_{k-1}$ is decremented exactly once.

Decrementing and incrementing non-allocated slots: If $s$ is not allocated at $HS_k$ then the same argument that was applied to new slots is used to show that the value of $s@HS_{k-1}$ is taken care of. Again, due to the safety theorem applied to the cycle at which the object containing $s$ was recycled we have $s@HS_k = \mathit{null}$ so there is no need to increment any field due to $s$. Indeed, since $s$ is not allocated at $HS_k$ and it is unreachable at $HS_{k-1}$ no record of it would appear in $Hist_k$ and no $rc$ field will be manipulated due to it in cycle $k$.

Incrementing old and new slots: It remains to show that the $rc$ field of $s@HS_k$ is incremented exactly once due to $s$, when $s$ is allocated at $HS_k$. We have two cases: either $s$ is determined, or it is undetermined. If $s$ is determined, then due to lemma A.3 we have that the collector increments the $rc$ value of $s@HS_k$. Otherwise, by lemma A.4, $VAL(Peek_k; s) = \{s@HS_k\}$. Thus, during the $\mathit{Fix}$-$\mathit{Undetermined}$-$\mathit{Slots}$ procedure the collector will find the value of $s@HS_k$ associated with $s$. It will increment the $rc$ field of that object exactly once, by the code.

All $rc$ adjustments are finished by the time $\mathit{Fix}$-$\mathit{Undetermined}$-$\mathit{Slots}$ terminates, so the claim holds at $\mathit{COLLECT}_k$.

\[ \square \]

A.1.5 Conclusion of safety proof

We are now ready to prove the safety theorem which claims that an object is collected at cycle $k$ only if it is unreachable at time $HS_k$.

Proof of safety theorem. The claim trivially holds for cycle zero since $ZCT_0$ is an empty set and thus no object is recycled during the initial cycle.

Consider cycle $k > 0$. We prove that the theorem holds for cycle $k$ if lemma A.5 holds for cycle $k$.

Let $\{o_1, \ldots, o_n\}$ be the sequence of objects for which $\mathit{Collect}$ is invoked, where the sequence is chronologically ordered. We show by induction on $i$, that $o_i$ is unreachable at $HS_k$. For the basis, consider $o_1$. As it is the first object to be collected, there is no clearing of slots (carried out in line (4) of procedure $\mathit{Collect}$) taking place prior to its reclamation, thus $o_1.rc@\mathit{COLLECT}_k = 0$. This
implies, according to lemma A.5 applied to cycle \( k \), that \( RC(o)@HS_k = 0 \). Additionally, by the code, \( o \) is collected only if \( o.rc = 0 \wedge o \notin Locals_k \) so we conclude that in addition of not being pointed by any heap slot at \( HS_k \), \( o \) is also not pointed by any global or local reference at that particular moment, or it would have been marked local. Thus, \( o \) is unreachable at \( HS_k \).

For the inductive step, consider \( o \) which has \( c = o.rc@COLLECT_k = RC(o)@HS_k \) (the last equality is again by lemma A.5). If \( c = 0 \) then the same arguments that were employed for \( o_1 \) are repeated in order to demonstrate that \( o \) is garbage at \( HS_k \).

Otherwise, we have \( c > 0 \). Since \( o \) is recycled, it must satisfy at some point during Reclaim-Garbage or Collect \( o.rc = 0 \wedge o \notin Locals_k \). Thus, the value of \( o.rc \) is decremented \( c \) times during the operation of Reclaim-Garbage. Since decrements are only applied to objects which are pointed from objects that are collected and since those objects are collected prior to \( o \) we have by the inductive hypothesis that all \( c \) references to \( o \) were from objects that were unreachable at \( HS_k \). Thus, at \( HS_k \), \( o \) is pointed only by unreachable objects, and it is not pointed by any local thread state or global reference. We conclude that \( o \) is unreachable at \( HS_k \).

\[ \square \]

A.2 Progress

In this section we show the capabilities of the algorithm in collecting garbage objects. The algorithm, in that respect, has the same limitations as the traditional single-threaded reference counting algorithms [34].

The best that we can hope to achieve with reference counting, without employing special techniques for detecting cycles of garbage, such as those surveyed in [33], is to detect any object that its reference count drops to zero, in order that it would be considered for reclamation based on the existence of local pointers to it. The following lemma tells us that this feature is achieved by the ZCT data-structure.

**Lemma A.6 (ZCT Property)** If \( o \) is allocated at \( HS_k \) and \( RC(o)@HS_k = 0 \) then \( o \in ZCT_k \).

**Proof.** The proof is by induction on \( k \). There are three cases to consider:

1. \( o \) is new to cycle \( k \). In this case, a mutator created \( o \) between \( HS_{k-1} \) and \( HS_k \). When it created \( o \) it added it to its New set, which becomes part of \( ZCT_k \).

2. \( o \) is old to cycle \( k \) and it had a positive \( rc \) field at \( END_{k-1} \). Since we have \( 0 = RC(o)@HS_k = o.rc@COLLECT_k \) (by lemma A.5), the value of \( o.rc \) must have reached zero due to the decrements applied by procedure Update-Reference-Counters of cycle \( k \). At that point \( o \) was added to \( ZCT_k \) (see lines (8-10) of that procedure.)

3. \( o \) is old at \( HS_{k-1} \) and it had zero \( rc \) field at \( END_{k-1} \). This case splits into two sub-cases:

   (a) if \( o.rc@COLLECT_k = 0 \) then \( RC(o)@HS_k = 0 \) by lemma A.5. Using the inductive assumption we know that \( o \in ZCT_{k-1} \). Since \( o \) was not recycled we must have \( o \in Locals_{k-1} \). By the code, when \( o \) is considered during Reclaim-Garbage it satisfies
   
   \[ o.rc = 0 \wedge o \in Locals_{k-1} \]

   by the code (lines (5-7)), \( o \) is added to \( ZCT_k \) in this case.
(b) Otherwise, \( o.re @ \text{COLLECT}_{k-1} > 0 \land o.re @ \text{END}_{k-1} = 0 \). This implies that \( o.re \) had reached zero by the decrements applied by one of the invocations of procedure \textbf{Collect}. By the code (lines (5-9)), when an object reference count reaches zero but it is not reclaimed, it is moved to the ZCT of the next cycle.

\( \square \)

Ideally, we would like the algorithm to collect at cycle \( k \) any object which is garbage at \( H.S_k \). However, this algorithm has the ordinary weaknesses of reference counting, with respect to cyclic structures, and thus only the following progress theorem can be guaranteed:

**Theorem A.2 (Progress)** If at \( H.S_k \) object \( o \) is unreachable and additionally \( o \) is not reachable from any cycle of objects, then \( o \) is collected in cycle \( k \).

The theorem is quite obvious due to lemma A.6 and the fact that we use ordinary recursive-freeing.

**B Sliding View Algorithm Safety Proof**

In this appendix we prove that the sliding view algorithm is safe. In the proof we abandon our assumption that there are no global references in the system. Instead, we assume that the collector, between carrying the third and fourth handshakes of a cycle, reads any global reference and marks the pointed object \textit{local}. In addition, mutators carry the following write-barrier for global references:

1. \( s := \text{new} \)
2. if \( \text{Snoop} \) then
   // mark \text{new} as local.
3. \( \text{Locals}_i := \text{Locals}_i \cup \{\text{new}\} \)

**B.1 Definitions**

First we need to stretch our definitions a bit in order to accommodate the looser timing of the second algorithm.

Let us define the time instances at which a thread \( T_i \) is suspended during the four handshakes of each cycle: \( H.S_k(i) \), \( H.S2_k(i) \), \( H.S3_k(i) \) and \( H.S4_k(i) \) denote the time instances at which thread \( T_i \) is suspended during the first, second, third and fourth handshakes of cycle \( k \), respectively. Next, we define the "global" time markers at which each handshake starts and ends:

\[
\begin{align*}
H.S_k &\overset{\text{def}}{=} \min_{T_i} H.S_k(i) \\
H.S.E.N.D_{k1} &\overset{\text{def}}{=} \max_{T_i} H.S_k(i) \\
H.S2_k &\overset{\text{def}}{=} \min_{T_i} H.S2_k(i) \\
H.S2.E.N.D_{k} &\overset{\text{def}}{=} \max_{T_i} H.S2_k(i) \\
H.S3_k &\overset{\text{def}}{=} \min_{T_i} H.S3_k(i) \\
H.S3.E.N.D_{k} &\overset{\text{def}}{=} \max_{T_i} H.S3_k(i) \\
H.S4_k &\overset{\text{def}}{=} \min_{T_i} H.S4_k(i) \\
H.S4.E.N.D_{k} &\overset{\text{def}}{=} \max_{T_i} H.S4_k(i)
\end{align*}
\]
Additionally we define $COLLECT_k$ to be the time at which procedure $\text{Reclaim-Garbage}$ starts its operation.

We need modify our notions of “being allocated” of the snapshot algorithm’s proof due to the lack of the hard handshake. This is done in the following definitions:

- We say that an object $o$ is allocated for cycle $k$ if some thread $T_i$ allocated $o$ after $HS_m(i)$ but before $HS_{m+1}(i)$, where $m < k$, and there had not been a cycle $l$, where $m \leq l < k$, such that $o$ was reclaimed on cycle $l$.
- $o$ is allocated new for cycle $k$ if $m = k - 1$ in the above definition.
- Otherwise, $o$ is allocated old for cycle $k$.
- We abbreviate and say that $o$ is new (old) to cycle $k$ if it is allocated new (old) for cycle $k$.
- Any of the above definitions apply to slots. The implied meaning is that the definition holds for the object containing the slot.

### B.2 The sliding view associated with a cycle

In this section we define a per-cycle sliding view that we later show that is computed implicitly by the collector and mutators (bearing similarity to the conceptual snapshot taken at $HS_k$ by the first algorithm which is never explicitly computed.)

Let us define the scan $\sigma_k$ that we associate with each cycle. We abbreviate $V_{\sigma_k}$ to $V_k$. Consider any memory word $s$.

- **Rule 1**: if $s \notin Hist_k$ then we set $\sigma_k(s) = HS_k$.

- if $s \in Hist_k$ then:
  - **Rule 2**: if $s$ is logged by some $T_i$ between $HS_k(i)$ and $HS_{3k}(i)$ then let $v$ be the consolidated value chosen for $s$. Let $\tau$ be the time a particular thread $T_j$ loaded $v$ before logging the pair $(s, v)$. Set $\sigma_k(s) \overset{\text{def}}{=} \tau$.
  - **Rule 3**: otherwise, no thread $T_i$ logs $s$ prior to $HS_{3k}(i)$, but $s$ is logged by some thread $T_j$ prior to $HS_k(i)$. On such an event set $\sigma_k(s) \overset{\text{def}}{=} HS2END_k$.

Note that $\sigma_k(s)$ is uniquely defined. We denote by $R_1(k)$ the set of all slots whose definition of $\sigma_k$ is derived by rule (1). Similarly we define the sets $R_2(k)$ and $R_3(k)$.

The next lemma characterizes the span of $\sigma_k$.

**Lemma B.1** \( \text{Start}(\sigma_k) \geq HS_k \land \text{End}(\sigma_k) \leq HSU3END_k \)

**Proof.** Let $s$ be a memory word. Certainly if $s \in R_1(k)$ or $R_3(k)$ then $\sigma_k(s)$ lies within the specified time limits. Otherwise, $s$ is defined according to rule (2), we note that $\tau$ must be earlier than $HS2END_k$ as some thread is logging $s$ prior to responding the third handshake. If this logging is done during clearing then the flag will be reinforced. Otherwise, the flag must remain on until the clearing of the next cycle. In particular, it’s on at $HS3END_k$. Thus no thread could load a value from $s$ after $HS3END_k$ and then log it since it is bound to sense that the dirty flag of $s$ is on.
B.3 Some basic claims

Recall that as asserted for the generic algorithm, we have to implement the snooping requirement in order to deduce on the “real” reference count of an object, based on its asynchronous reference count. The following lemma shows that the requirement is indeed enforced and that thus its implications hold:

**Lemma B.2** Any object \( o \) which is not marked local (i.e., \( o \not\in \text{Local}_k \)) at \( \text{COLLECT}_k \) satisfies

\[
\text{ARC}(V_k; o) \geq \text{RC}(o)@HS_{4k}
\]

Moreover, the set of pointers that point to \( o \) at \( HS_{4k} \) is a subset of those that point to it in \( V_k \).

**Proof.** According to lemma 4.1 it suffices to show that if a reference to \( o \) is stored to a slot \( s \) at or after \( \sigma_k(s) \) and before \( \text{End}(\sigma_k) \), then \( o \) is marked local. By lemma B.1 we know that \( \text{End}(\sigma_k) < HS_{4k} \), hence we can replace \( \text{End}(\sigma_k) \) with \( HS_{4k} \), hardening the requirements of lemma 4.1. i.e., we require that if a reference to \( o \) is stored to a slot \( s \) during the interval \( [\sigma_k(s), HS_{4k}] \) then \( o \) is marked local.

Since updates are not interruptible and since the Snoop\(_i\) flag is reset only after \( HS_{4k}(i) \), it suffices to show that the test of Snoop\(_i\) in the Update procedure returns true in the case that the store proper into \( s \) is executed after \( \sigma_k(s) \) and before \( HS_{4k}(i) \). Consider a store of \( o \) into \( s \) which is scheduled at, or after \( \sigma_k(s) \) and before \( HS_{4k}(i) \). Due to lemma B.1, the store is scheduled at or after \( HS_k \). At that time, for any thread \( T_i \), the Snoop\(_i\) flag is set. Since the test of Snoop\(_i\), in line (7) of procedure Update, is executed after the store proper, of line (6), it would return true and the object will be marked accordingly local.

\[ \square \]

**Lemma B.3** The following claims hold: (1) if thread \( T_i \) logs \( s \) between responding to the first and third handshakes then \( \text{Dirty}(s)@HS_{3k}(i) = \text{true} \). (2) if thread \( T_i \) logs \( s \) between responding to the first and fourth handshakes then \( \text{Dirty}(s)@HS_{4k}(i) = \text{true} \).

**Proof.** Claim (1): The only reason the flag might be off after \( T_i \) has raised it is that the collector has reset it in procedure Clear-Dirty-Marks. If that is the case, then the collector has reset the flag after the it has completed logging the slot. Hence, in procedure Reinforce-Dirty-Mark, the collector will see the slot in \( T_i \)'s buffer and would reinforce it. This happens before \( HS_{2k} \). Claim (2) is trivial due to the validity of claim (1).

\[ \square \]

B.4 Roadmap for the proof

In the proof of the algorithm we assume again that a garbage collection cycle number zero takes place just before any mutator is started, i.e., at initialization time. As stated for the first algorithm, there is no loss of generality involved, this is just a mere issue of convenience. Convenience is also the cause for the following definition:

\[
HS_{k} \triangleq HS_0
\]

Or, equivalently, we may assume that yet another garbage collection cycle is occurring before cycle number zero. The reason we need this definition is that we can reason freely about what
happened in the interval \([HS_{k-2}, HS_{k-1}]\), while reasoning on cycle \(k > 0\). The above definition allows us to escape dealing with garbage collection cycle number one as a special case.

The proof is naturally by induction on the cycle number. We have several interdependent claims that jointly prove that the algorithm is safe. In the next section we present the claims and show their inter-dependencies. Then, we prove the claims.

The goal of the proof is to show that any object is reclaimed only if it is garbage. This claim is contained in the safety theorem—theorem B.1.

The validity of theorem B.1, for cycle \(k\), stems from lemma B.6 which links the computed \(rc\) field of each object to its \(ARC\) in \(V_k\), the sliding view associated with cycle \(k\).

This linking is proved correct for cycle \(k\), provided:

- the linking argument holds for cycle \(k - 1\)
- theorem B.1 holds for previous cycles.
- all differences between \(V_{k-1}\) and \(V_k\) are recorded consistently by mutators. This claim is contained in lemma B.5.
- the collector reclaimed objects in a sensible manner during cycle \(k - 1\). “In a sensible manner” means it took into account the values of reclaimed slots as they appeared in \(V_{k-1}\). This claim is contained in lemma B.7.

Lemma B.7 itself builds on the logging capabilities of mutators (lemma B.5) and on theorem B.1. Lemma B.5 which summarizes the algorithm properties with respect to thread buffers and logging is proved correct based on the validity of theorem B.1 and lemma B.4 for previous cycles.

Lemma B.4 itself asserts that any slot has a time point in the beginning of each cycle whence the dirty flag of the slot is off. This rather lame-looking lemma is crucial for the operation of the logging mechanism. Its proof relies on the correctness of the same claim for previous cycles.

Using the notation of the proof of the snapshot algorithm we summarize the interdependencies:

- each of the claims is proved correct for cycle zero independently.
- for a cycle \(k > 0\)
  - \(LB.A_k \iff LB.A_{k-1}\)
  - \(LB.5_k \iff LB.A_{k-1} \land \bigwedge_{j<k} SafetyTheorem_j\)
  - \(LB.6_k \iff LB.5_k \land LB.6_{k-1} \land LB.7_{k-1} \land \bigwedge_{j<k} (SafetyTheorem_j)\)
  - \(SafetyTheorem_k \iff LB.6_k\)
  - \(LB.7_k \iff LB.5_k \land SafetyTheorem_k\)

### B.5 Inductive safety arguments

Compensating for the lack of the hard handshake of the snapshot algorithm, during which all dirty marks were turned off we have procedure \texttt{Clear-Dirty-Marks} in the sliding view algorithm. The following lemma asserts that indeed each slot experiences a point in time, after the start of a cycle, at which the dirty flag is off. This is essential for the logging mechanism to operate correctly since it instructs mutators to start logging modifications from fresh, relating to the new cycle.

**Lemma B.4** Let \(s\) be a memory word. There exists a time point, denoted \(t_k(s)\) at which the dirty slot for \(s\) is off. Specifically:

\(\)
• if \( s \in R_{1, k} \) then \( t_k(s) \overset{\text{def}}{=} \sigma_k(s) \overset{\text{def}}{=} HS_k \).

• if \( s \in R_{2, k} \) then \( t_k(s) \) exists and it satisfies \( HSEN D_k < t_k(s) < HS_{2, k} \).

• if \( s \in R_{3, k} \) then \( t_k(s) \overset{\text{def}}{=} HSEN D_k \). There are no ongoing updates of \( s \) at \( t_k(s) \).

**Proof.** The proof is by induction on the cycle number, \( k \). For \( k = 0 \) the claim holds since all slots are cleared at \( HS_0 \) and all slots are members of \( R_{1, 0} \). For \( k > 0 \) we prove the claim correct provided it holds for the previous cycle and theorem B.1 holds for all previous cycles. We divide to cases:

• if \( s \in R_{1, k} \) then either \( s \in R_{1, k-1} \) or \( s \in R_{3, k-1} \). \( R_{2, k-1} \) is impossible because it implies that \( s \in Hist_k \).

If \( s \in R_{1, k-1} \) then by the inductive hypothesis \( Dirty(s)@HS_{k-1} = \text{false} \). Had some thread \( T_i \) turned on the flag on after \( HS_{k-1} \) and before \( HS_k \) then \( s \) would have been recorded in either \( Hist_{k-1} \) or \( Hist_k \), neither of which is the case, so the dirty flag must be continuously off from \( HS_{k-1} \) to \( HS_k \).

Otherwise, \( s \in R_{3, k-1} \). Thus, according to the inductive hypothesis \( Dirty(s)@HSEN D = \text{false} \). By definition of \( R_{3, k-1} \), no thread logged \( s \) before responding to the third handshake of cycle \( k - 1 \). Thus no thread had turned the flag on prior to responding to that handshake. Had some thread logged \( s \) after the third handshake of cycle \( k - 1 \) but before the first handshake of cycle \( k \) then we would have \( s \in Hist_k \), which is not the case. Again we have \( Dirty(s)@HS_k = \text{false} \).

• if \( s \in R_{2, k} \) then the collector has turned off \( Dirty(s) \) during the clearing stage. We define \( t_k(s) \) to be the time instance just after the clearing of \( Dirty(s) \) was scheduled.

• if \( s \in R_{3, k} \) then the collector has turned off \( Dirty(s) \) during the clearing stage and no thread has turned it on prior to responding to the third handshake. We conclude that the flag must have been off at the time the second handshake ended. At \( HSEN D_k \) only updates of threads that have already responded to the second handshake may be ongoing. But had such an update occurred, it must have sensed that the flag is off and it would consequently log \( s \), contradicting the definition of \( R_{3, k} \). We conclude that there are no ongoing updates at \( HSEN D_k \).

\( \Box \)

We proceed to consider the properties of the write-barrier. The next lemma, which is the equivalent of lemma A.2 of the snapshot algorithm, states that any slot which is modified between scans is recorded along with its value in the previous sliding view and that no other value is associated with the slot.

**Lemma B.5** Let \( s \) be a slot. The following claims hold:

1. if \( s \) is old for cycle \( k \) and modified during cycle \( k - 1 \) then \( VAL(Hist_k; s) = \{ V_{k-1}(s) \} \).

2. if \( s \) is new for cycle \( k \) and modified during cycle \( k - 1 \) then \( VAL(Hist_k; s) = \{ \text{null} \} \).

3. if \( s \) is old for cycle \( k \) and is not modified during cycle \( k - 1 \) then \( VAL(Hist_k; s) \supseteq \{ V_{k-1}(s) \} \).

4. if \( s \) is new for cycle \( k \) and is not modified during cycle \( k - 1 \) then \( VAL(Hist_k; s) \supseteq \{ \text{null} \} \).
Proof. For garbage collection number zero the claims trivially hold since \( \text{Hist}_0 = \emptyset \) and indeed no slot is modified prior to the cycle. We prove that the claim holds for cycle \( k > 0 \) provided it itself hold for cycle \( k - 1 \) and that theorem B.1 and lemma B.4 hold for earlier cycles.

We divide into cases according to the state of \( s \):

**s is old for cycle \( k \) and \( s \in R1_{k-1} \)**. Suppose that \( s \notin \text{Hist}_{k-1} \). In that case we have \( \sigma_k(s) \overset{\text{def}}{=} H S_k \) and we have to show that \( s \) is not changed between \( H S_{k-1} \) and \( H S_k \).

Since \( s \notin \text{Hist}_{k-1} \) we conclude, by the inductive hypothesis, that no thread modified \( s \) between \( \sigma_{k-2}(s) \) and \( H S_{k-1} \). Additionally we know that at \( H S_k \) the dirty mark of \( s \) is off. The dirty mark must be off at \( H S_{END_{k-2}} \) as well and no update is ongoing at that moment as that update would have rendered \( s \) part of \( \text{Hist}_{k-1} \). Using the same arguments of lemma A.1 applied for \( s \) and \( H S_{END_{k-2}} \) and since \( s \) is not cleared before \( H S_{END_k} \) any update whose store proper operation is scheduled between \( H S_{END_{k-2}} \) and \( H S_k \) would result in the association of \( s @ H S_{END_{k-2}} \) with \( s \) in either \( \text{Hist}_{k-1} \), or \( \text{Hist}_k \), neither of which is the case. We conclude that \( s \) is indeed not modified during cycle \( k - 1 \).

Now suppose \( s \in \text{Hist}_k \). In that case we want to show that \( \text{VAL} (\text{Hist}_k; s) = s @ H S_{k-1} \).

Again, we've concluded that any thread \( T_i \) that would log \( s \) prior to \( H S_k(i) \) would associate it with \( s @ H S_{END_{k-2}} \). Since a store to \( s \) could not have been scheduled between \( s @ H S_{END_{k-2}} \) and \( H S_{k-1} \) without logging the slot we conclude that \( s @ H S_{k-1} = s @ H S_{END_{k-2}} \), which is the desired result.

**s is old for cycle \( k \) and \( s \in R2_{k-1} \)**. Since some thread modified and logged \( s \) between the first and third handshakes of cycle \( k - 1 \) We have to show that claim (1) holds for \( s \). Due to the reinforcement step, the dirty flag of \( s \) must be on at \( H S_{k-1} \), thus, there is no possibility that a thread would log \( s \) after responding to the fourth handshake. As for the records kept regarding \( s \) between the first and fourth handshakes, the collector chooses a single pair, say \( \langle s, v \rangle \) and moves it to \( \text{Hist}_k \). By definition of \( \sigma_k \) we have \( V_{k-1}(s) = v \).

**s is old for cycle \( s \) and \( s \in R3_{k-1} \)**. We have noted in lemma B.4 that \( t_{k-1}(s) = \sigma_{k-1}(s) = H S_{END_{k-1}} \) and no update is occurring at that moment. Suppose \( s \notin \text{Hist}_k \). In that case \( \sigma_k(s) = H S_k \) and we have to show that no store is scheduled between \( H S_{END_{k-1}} \) and \( H S_k \).

But this is trivial since the probing of the dirty mark associated with such a store must start after \( H S_{END_{k-1}} \), as no updates occur at that moment. Thus, had such an update been scheduled, it must have sensed that the flag is on and \( s \) would have become a member of \( \text{Hist}_k \), a contradiction.

Suppose now that \( s \in \text{Hist}_k \). We have to show that \( \text{VAL} (\text{Hist}_k; s) = s @ H S_{END_{k-1}} \). Again, since at \( H S_{END_{k-1}} \) the dirty bit is off and no update of it is occurring. And since the dirty mark is reset only after all threads have responded to the first handshake of cycle \( k \), by lemma A.1 they are bound to associate \( s @ H S_{END_{k-1}} \) with \( s \).

**new slots allocated for the first time**. If \( s \) is allocated for the first time, then \( \sigma_{k-1} \overset{\text{def}}{=} H S_{k-1} \) and at that time \( s \) contained \text{null} and its dirty flag was initialized to \text{false}. These values remain in effect until \( s \) is allocated. Additionally, no update of \( s \) occurs at the moment it is allocated. Again, the claim follows using the arguments of the previous cases.

**new slots which are reallocated**. We first show that \( \text{Hist}_k \) cannot contain “leftovers”: i.e., logging that refer to the “previous life” of \( s \), before it was reallocated. Suppose that \( s \) was last reclaimed during cycle \( m, m < k \). If \( m < k - 1 \), then there will be no record of the “previous life” of \( s \) in \( \text{Hist}_j \) due to the safety theorem applied to cycle \( m \) that assures us that \( s \) was unreachable from its reclamation point up to the time it was reallocated, during cycle \( k - 1 \). If, on the other hand, \( s \) was reclaimed during cycle \( k - 1 \), then as the safety theorem tells us, no thread \( T_i \) had access to \( s \) after \( H S_{END_{k-1}}(i) \). \( s \) could have not occurred in the digested part of \( \text{Hist}_k \) as that would
have caused the deferral of the reclamation of its containing object to cycle \( k \). So there are no leftovers in this case as well.

Applying the safety theorem to cycle \( m \), we know that the object that contained \( s \) was garbage when it was reclaimed. Its dirty marks, the one of \( s \) included, were off. When the collector freed the object it stored \texttt{null} into \( s \). Since the object was unreachable, \( s \) remained inaccessible up to the time it was re-allocated. Just when \( s \) was re-allocated, there was no update of it ongoing, it contained \texttt{null}, and the dirty flag for it was \texttt{false}. We conclude that the lemma holds due to the same arguments employed for the previous cases.

We have considered all possible cases for old and new allocated slots and have shown that they always satisfy the claims.

\[ \square \]

It has just been demonstrated that the collector has full knowledge on which slots have changed since the most recent scan and what were their contents. We now show that the collector can find out what are these slots values in a current cycle as well. These two abilities combined amount for the collector’s ability to calculate the asynchronous reference count of each object, relative to the sliding view of the current cycle.

**Lemma B.6** For any object \( o \) which is allocated at time \( \text{COLLECT}_k \) it holds that \( \alpha r c@\text{COLLECT}_k = \text{ARC}(V_k, o) \).

**Proof.** The claim trivially holds for collection cycle zero, since there are no allocated objects at \( \text{COLLECT}_0 \). To prove that the claim holds for cycle \( k > 0 \) we assume that it holds for cycle \( k - 1 \) and that lemmas B.7 hold for cycle \( k - 1 \) and B.5 hold for cycle \( k \).

We note that it suffices to show that:

1. for any slot \( s \) due to which \( rc \) fields are adjusted by the algorithm the \( rc \) field of \( V_{k-1}(s) \) is decremented exactly once, during the interval \( [\text{COLLECT}_{k-1}, \text{COLLECT}_k) \), while the \( rc \) field of \( V_k(s) \) is incremented exactly once during the same interval.

2. if \( V_{k-1}(s) \neq V_k(s) \) then the algorithm adjusts \( rc \) fields due to \( s \).

Consider a memory word \( s \), it is in exactly one of three states, with respect to cycle \( k \): allocated new, allocated, not allocated.

**Adjusting \( rc \) fields due to allocated new slots.** If \( s \) has been collected during cycle \( k - 1 \) then according to lemma B.7, the collector decremented the \( rc \) field of \( V_{k-1}(s) \) when the object containing \( s \) was reclaimed. At that point, \( s \) assumed the value of \texttt{null}, which remained in effect at least until \( s \) was reallocated, assuming that theorem B.1 holds for cycle \( k - 1 \).

Another possibility is that the object containing \( s \) was reclaimed during cycle \( m \), where \( m < k - 1 \). Since \( s \) is new to cycle \( k \), it was not allocated for cycle \( k - 1 \) and we have \( \sigma_{k-1}(s) \overset{\text{def}}{=} H S_{k-1} \) and by the definition of sliding views we have \( V_{k-1}(s) = \texttt{null} \). Thus, we would expect that no \( rc \) field will be decremented due to \( s \). Indeed, since the object containing \( s \) was not reclaimed during cycle \( k - 1 \), no decrement was applied due to \( s \) as the result of recursive deletion of cycle \( k - 1 \).

Again, due to theorem B.1, we know that when \( s \) was reallocated it assumed the value of \texttt{null}.

Finally, if \( s \) has not been ever allocated before then surely it was not subject to recursive deletion during cycle \( k - 1 \) and it contained \texttt{null} at the time it was allocated.

We conclude that at any rate, by the time \( s \) is allocated, it contains \texttt{null} and all necessary adjustments have been made to the \( rc \) field of \( V_{k-1}(s) \) in order to reflect that.
Now we have to show that if $V_k(s) \neq \text{null}$ then the $rc$ field of $V_k(s)$ is incremented and otherwise no field is incremented, and, that no $rc$ field is decremented due to $s$ in updating of cycle $k$.

If no thread modifies $s$ between its allocation point and before $HS_k(i)$, then, according to lemma B.5, $s \notin Hist_k$ and $\sigma_k(s) \overset{\text{def}}{=} HS_k$. At $\sigma_k(s)$ $s$ still assumes the value of $\text{null}$ and thus $V_k(s) = \text{null}$. Therefore, we would expect that no $rc$ field will be incremented due to $s$ in cycle $k$. Since $Hist_k$ does not contain any reference of $s$, this is actually the case. For the same reason no $rc$ field will be decremented as well.

If, on the other hand, some thread $T_i$ modifies $s$ between its allocation point and before $HS_k(i)$ then according to lemma B.5, applied for cycle $k$, $VAL(Hist_k; o) = \{\text{null}\}$. Thus, the collector would adjust $rc$ field due to $s$ during the execution of Update-Reference-Counters. No $rc$ field will be decremented due to $s$ as $\text{null}$ is associated with the slot in $Hist_k$. The collector will then either determine $s$, or declare it undetermined. If $s$ is determined, it will increment the $rc$ value of the determined value, which we have shown to be equal to $V_l(s)$. Otherwise, when $s$ is undetermined, the collector adds it to the set Undetermined. It will subsequently consolidate $s$ during the operation of Fix-Undetermined-Slots. The $rc$ field of the resolved value, which also equals $V(s)$, will be incremented exactly once, due to the Handled set. No matter whether $s$ is determined or not, we’ve shown that the $rc$ field of $V_k(s)$ is incremented exactly once.

**Adjusting $rc$ fields due to allocated old slots.** Since $s$ is not reclaimed during cycle $k - 1$ there is no $rc$ adjustments due to it during the recursive deletion of cycle $k - 1$. It is left to consider the effects due to $s$ in the course of updating during cycle $k$.

If $s$ is an allocated old slot for cycle $k$ then it may be either modified or non-modified during cycle $k$.

If $s$ is modified, then (due to lemma B.5) $VAL(Hist_k; s) = \{V_{k-1}(s)\}$. Consequently, $V_{k-1}(s).rc$ will be decremented during Update-Reference-Counters. Then, $s$ will be either determined or consolidated and the $rc$ value of $V_k(s)$ will be incremented accordingly as shown in the previous paragraphs for new slots.

Otherwise, $s$ is not modified. Then we have $VAL(Hist_k; s) = \varnothing$ and no $rc$ updating due to it occur during cycle $k$, which is the desired result since $V_{k-1}(s) = V_k(s)$.

**Adjusting $rc$ fields due to non-allocated slots.** If $s$ has not ever been allocated then the claim trivially holds.

If $s$ has been reclaimed during cycle $k - 1$ then we have shown, while dealing with new slots, that at the time $s$ is reclaimed $\text{null}$ is assigned to it and the respective $rc$ value of $V_{k-1}(s)$ is decremented accordingly.

Consider a slot $s$ which is not allocated for cycle $k$ that has been most recently been reclaimed during cycle $m < k - 1$. According to the safety theorem, applied for cycle $m$, no thread $T_i$ had access to $s$ after $HS END_m$. Thus, at $HS_k$ no thread had access to $s$ which leads to $s \notin Hist_k$. Additionally, $s$ could not be the subject of recursive deletion during cycle $k - 1$, because that would have meant that the object containing $s$ was deleted twice in a row, which is contradictory to the safety theorem. We conclude that $s$ is neither the subject of recursive deletion during cycle $k - 1$, nor of $rc$ field updating during cycle $k$, as desired.

Since we have covered all possible options for the state of $s$, the claim holds.

\[ \square \]

Building on the foundations provided by the link between the conceptual asynchronous reference count and the concrete $rc$ field and by the correct implementation of the snooping requirement, proved by lemma B.2, we are now ready to prove our main claim.
Theorem B.1 An object $o$ is garbage when it is reclaimed. More specifically, $o$ is not reachable from any thread $T_i$ after $HS4_k(i)$ and hence $o$ is garbage at $HS4(END_k)$.

Proof. We prove the claim by induction on the cycle number, $k$. For $k = 0$ we have an empty ZCT, therefore no object is reclaimed during this cycle and the claim vacuously holds. For $k > 0$ we prove that the claim is correct provided lemma B.6 holds for cycle $k$.

Let $\{T_1, T_2, \ldots, T_n\}$ be the set of all mutator threads, ordered by the time they respond to the fourth handshake, i.e., $HS4_k(1) < HS4_k(2) < \ldots < HS4_k(n)$. Let $\{o_1, \ldots, o_m\}$ be the set of objects which Collect is invoked for during cycle $k$, ordered chronologically by the time of the invocation (i.e., $o_1$ was processed first and $o_m$—last.)

Consider any object $o_j$ that was processed by Collect. We prove that the following invariant holds for $o_j$:

**Invariant B.1 (I1)** For each thread $T_i$, $o_j$ was continuously unreachable from $T_i$ in the time interval $[HS4_k(i), HS4_k(n)]$, i.e., was not reachable through any of $T_i$'s local references and through any global root at any time point in the interval.

The proof is by double induction: the outer induction variable is $j$, subscripting the objects that were processed. The inner induction variable is $i$, denoting the index of threads in the order they responded to the fourth handshake.

For the basis, we consider $o_1$. In order to prove that I1 holds for $o_1$ we prove that an additional assertion holds:

**Invariant B.2 (I2)** $RC(o_1) = 0$ continuously in the time interval $[HS4_k(1), HS4_k(n)]$.

Define I3 as the logical conjunction of I1 and I2. First we show that I3 holds for $o_1$ in the (single-pointed) interval $[HS4_k(1), HS4_k(1)]$. Then we show that given that I3 holds in the interval $[HS4_k(1), HS4_k(i-1)]$, then it holds in the interval $[HS4_k(i-1), HS4_k(i)]$ as well and hence in the entire interval $[HS4_k(1), HS4_k(i)]$.

Note that I3, restricted to the interval $[HS4_k(1), HS4_k(1)]$ simply asserts that $o_1$ was not directly reachable from any of $T_1$'s local references and from any global root at $HS4_k(1)$ and that $RC(o_1)@HS4_k(1) = 0$. We prove that this is indeed the case.

Since $o_1$ was processed the first, Collect must have been invoked directly from Reclaim-Garbage for it. Thus, $0 = o_1.rc@COLLECT_k$. This implies

\[ 0 = ARC(V_k, o) \geq RC(o)@HS4_k \implies RC(o)@HS4_k = 0 \]

by lemmas B.6 and B.2 and the fact that a reference count is non-negative. Additionally, $o_1$ was not directly reachable from $T_1$ at $HS4_k(1)$, or it would have been marked local when $T_1$'s state was scanned when it responded to the fourth handshake. Finally, $o_1$ was not directly reachable from any global root at $HS4_k(1)$. To see that this is indeed the case consider any global root $r$. The collector read $r$ prior to starting the fourth handshake and marked the referenced object local. Since the time the collector read $r$ and up to $HS4_k(1)$ all threads would have marked an object local had they stored a reference to the object into $r$. Thus, at any rate, the object which is pointed by $r$ at $HS_k$ is marked and thus it cannot be $o_1$.

If $n = 1$ then we are done. Otherwise, we prove that I3 holds for the interval $[HS4_k(i-1), HS4_k(i)]$, where $1 < i \leq n$, provided it holds during the interval $[HS4_k(1), HS4_k(i-1)]$. I3, restricted to the interval in question, requires that:

1. $RC(o_i) = 0$ continuously during the interval, and
2. \( o_1 \) was not directly reachable from any of the threads in the set \( P = \{ T_1, \ldots, T_{i-1} \} \) continuously during the interval, and

3. \( o_1 \) was not directly reachable from any global root continuously during the interval, and

4. \( o_1 \) was inaccessible from \( T_i \) at \( HS4_k(i) \).

The inductive hypothesis (on \( i \)) assures us that \( o_1 \) was not directly reachable from all the threads in \( P \) and from any global root at \( HS4_k(i-1) \) and that \( RC(o_1)@HS4_k(i-1) = 0 \). Examining any possible operation which is scheduled during the interval \([HS4_k(i-1), HS4_k(i)]\) we learn that \( I3 \) remained continuously in effect. We show that any instruction of time \( t \in [HS4_k(i-1), HS4_k(i)] \) cannot violate \((1),(2)\) or \((3)\) provided \((1),(2)\) and \((3)\) hold up to time \( t - 1 \) then we show that \((4)\) holds.

- a load cannot violate requirements \((1)\) or \((3)\) simply because it is a load, and not a store. It cannot violate requirement \((2)\) since no object or global root is referring to \( o_1 \), due to the validity of \((1)\) and \((3)\) in previous steps.
- a store operation cannot violate \((2)\) since only a load can.
- a store by a thread \( T_i \in P \) cannot violate \((1)\) or \((3)\) since the operand of the store cannot be \( o_1 \), due to the validity of \((2)\) in previous steps.
- a store by a thread \( T_i \notin P \) cannot violate \((1)\) or \((3)\) because the operand of the store cannot be \( o_1 \) since the \textit{Snoop} flag is set during the interval and such a step would have marked \( o_1 \) \textit{local}.
- to prove that \((4)\) is satisfied: at time \( HS4_k(i) \) \( o_1 \) is not indirectly reachable, from any thread or global root, since \((1)\) holds at \( HS4_k(i) \). It is not directly reachable from \( T_i \), because that would have caused it being marked \textit{local}. It is not directly reachable from a global root at \( HS4_k(i) \) since \((3)\) holds at that moment.

That completes the proof that \( I3 \) and therefore \( I1 \) in particular, hold for \( o_1 \).

Consider now the object \( o_j, 1 < j \leq m \). If \( o_j.re@COLLECT_k = 0 \) then the same arguments that were employed for \( o_1 \) are repeated. Otherwise, we have

\[
\epsilon \overset{\text{def}}{=} o_j.re@COLLECT_k > 0
\]

Since \( o_j \) is eventually processed by \textbf{Collect} there must have been \( \epsilon \) slots pointing at \( o_j \) that were cleared and \( o_j.re \) decremented accordingly, in lines (7-8) of \textbf{Collect}. Note that the collector tested the dirty flags of these slots and found that they were off prior to their processing. Since the dirty flag is off for these slots after \( HS4END_k \), no thread could have changed them after, or at \( HS_k \) and before responding to the fourth handshake (due to lemma B.3).

Moreover, since these \( \epsilon \) slots were contained in objects that were processed prior to \( o_j \) the inductive lemma (on objects) apply and we know that no thread had access to any of the \( \epsilon \) slots after responding to the fourth handshake. We conclude that these \( \epsilon \) slots have not been changed after \( HS4_k \) and before the collector processed them.

In order to prove \( I1 \) we prove an additional invariant:

\textbf{Invariant B.4 (I4)} \textit{No reference to} \( o_j \) \textit{has been stored during the interval} \([HS4_k(1), HS4_k(n)]\) \textit{to either a heap slot or a global reference}.
Define $i_5$ as the logical conjunction of $i_1$ and $i_4$. We prove that $i_5$ holds for $o_j$.

We have already said that at $H S 4_k(1)$ there existed exactly $c$ references to $o_j$. All these references were contained in objects that, according to the inductive hypothesis on objects, were unreachable from $T_i$ at $H S 4_k(1)$. Additionally, $o_j$ was not directly reachable from $T_i$ at $H S 4_k(1)$, or it would have been marked local. $o_j$ has not been directly reachable from a global reference at $H S 4_k(1)$ since that would have caused it being marked local, for the same arguments that were applied for $o_i$. Finally, had $o_j$ been indirectly reachable from a global reference $r$ at $H S 4_k(1)$ then the chain of references must have passed through some of the $c$ slots which are contained in objects which are assumed to be inaccessible from $T_i$ at $H S 4_k(1)$, contradicting the inductive hypothesis on objects. Thus, $i_1$, restricted to the interval $[H S 4_k(1), H S 4_k(1)]$ holds for $o_j$.

$i_4$, restricted to the interval $[H S 4_k(1), H S 4_k(1)]$, holds as well since $H S 4_k(1)$ is the time at which $T_i$ responded to the handshake and naturally it did not execute a store at the same time.

We now show by similar arguments to those applied for $o_i$ that $i_5$ restricted to the interval $[H S 4_k(i-1), H S 4_k(i)]$, where $1 < i \leq n$, holds provided it holds during the interval $[H S 4_k(1), H S 4_k(i-1)]$. We also use the inductive hypothesis on $j$ that asserts that for any object $o_k$, $a < j$, $i_1$ holds for the entire interval $[H S 4_k(1), H S 4_k(n)]$.

Invariant $i_5$ applied to $o_j$ and restricted to the interval $[H S 4_k(i-1), H S 4_k(i)]$ requires that:

1. $o_j$ is not reachable continuously during the interval from any local reference of a thread in $P$.
2. a reference to $o_j$ is not stored during the interval.
3. $o_j$ is not reachable continuously during the interval from any global reference.
4. $o_j$ is not reachable from $T_i$ at $H S 4_k(i)$.

We show that any instruction of time $t \in [H S 4_k(i-1), H S 4_k(i)]$ cannot violate (1), (2) or (3) provided (1), (2) and (3) hold up to time $t - 1$ then we show that (4) holds.

- a load by a thread $T_i$ could not have made $o_j$ reachable from $T_i$ unless it was reachable from it prior to the load. It also has no effect on the reachability of $o_j$ from other threads. Therefore such an action cannot violate neither (1) nor (3), assuming (1) and (3) hold for previous steps. Naturally it cannot violate (2).

- a store by a thread $T_i \in P$ cannot make $o_j$ reachable for any thread in $P$ unless $o_j$ has been already reachable from $T_i$ just before the action took place, which is not the case. So a store by $T_i$ preserves (1), (2) and (3) provided (1) and (3) hold for previous steps.

- a store by a thread $T_i \notin P$ cannot make $o_j$ reachable from any thread in $P$ for the following reasons:
  - $T_i$ could not have stored a reference to $o_j$ itself since the $Snoop_l$ flag is set during the interval and such a step would have marked $o_j$ local, preventing its processing by $Collect$.
  - $T_i$ could not have stored a pointer to $x$ from which $o_j$ is reachable since all references to $o_j$ at the time of the store, by the validity of (2) for previous steps, are a subset of the the set of $c$ references that pointed to $o_j$ at $H S 4_k$. Thus, the chain of references from $x$ to $o_j$ must pass through an object $o_a$, with $a < j$. The store would have rendered $o_a$ reachable from some thread in $P$, which is contradictory to the inductive assumption on $o_a$.

So (1), (2) and (3) are not violated by a store by $T_i \notin P$. 

63
• it remains to show that (4) is not violated. Suppose that at $HS4_k(i) o_j$ is reachable from $T_i$. $o_j$ could not have been directly reachable at the time, or it would have been marked local.

By the validity of (2) for $HS4_k(i)$ we know that if $o_j$ is reachable from $T_i$ then it is reachable through some object $o_a$, with $a < j$. This implies that $o_a$ is reachable from $T_i$ at $HS4_k(i)$. Again, a contradiction to the inductive assumption on $o_a$.

That completes the proof that $I5$ and therefore $I1$ hold for $o_j$.

Applying $I1$ for any object which is processed we learn that any such object is garbage at $HS4END_k$ (which equals, by definition, $HS4_k(n)$.) Since the objects which are eventually reclaimed are a subset of those processed (the rest have their reclamation deferred to the next cycle) the algorithm is indeed safe.

Last but not least we have to prove lemma B.7, whose correctness was assumed by lemma B.6. The lemma asserts that the collector sensibly deallocates objects. That is, that it decrements the $rc$ field of slots in a manner which is not discordant with their linkage to the sliding view.

**Lemma B.7** Let $o$ be an object which is reclaimed during cycle $k$ and let $s$ be a slot of the object. Then the collector decrements $V_k(s)$ exactly once due to recursive deletion in cycle $k$.

**Proof.** The claim vacuously holds for cycle $k = 0$. We prove that it holds for cycle $k > 0$ provided theorem B.1 and lemma B.5 hold for cycle $k$.

As the reference count of an object is monotonically non-increasing due to recursive deletion and since an object is processed by Collect only when its $rc$ field reaches zero, $o$ is processed exactly once before being reclaimed.

Since $o$ is reclaimed, the collector resets all its slots, including $s$. When the collector considers $s$ it probes the value of $Dirty(s)$ and finds it off. As noted in lemma B.5, $s$ could not have been modified by any thread between responding to the first handshake and fourth handshake. So $s$ is not in the digested history for the next cycle.

If $s \notin Hist_k$ then $\sigma_k(s) = HS_k$. By lemma B.4 $Dirty(s) \circ \sigma_k(s) = false$ thus no thread $T_i$ could have changed $s$ between $\sigma_k(s)$ and $HS_k(i)$. If $s \in Hist_k$ then it must be that $s \in R3_k$. So in that case $\sigma_k(s) = HS2END_k$. At any rate, no thread $T_i$ changed $s$ between $\sigma_k(s)$ and $HS4_k(i)$.

Theorem B.1 asserts that $s$ was inaccessible for any thread after responding to the fourth handshake.

Assembling these facts we get that any rate $s$ was not modified between $\sigma_k(s)$ and the time the collector read its value, prior to resetting it in procedure Collect. So the collector indeed decremented the $rc$ value of $V_k(s)$.

This completes the safety proof of the algorithm.

C Tracing Sliding View Algorithm Safety Proof

The tracing algorithm possess the same properties of the sliding view reference counting algorithm with respect to logging, determining of slots and resolution of undetermined slots. Therefore, in this proof, we take these properties for granted and we are concerned only with their application to tracing and sweeping. Thus appendix B is a prerequisite for this proof.
In order to prove safety we consider two kinds of reachable objects: those that were allocated prior to the fourth handshake, juxtaposed with those allocated after it. For the first kind, we show that mistaken reclamation is impossible since there exist a chain in the graph induced by the sliding view of the cycle that starts from a local object and leads to the reachable object in question and that tracing proceeds uninterrupted along such a chain, so reclamation is impossible. The second kind of objects are protected from reclamation by the object coloring protocol.

As in the proof of correctness of the sliding view algorithm, we abandon our assumption that there are no global references in the system. Instead, we assume that the collector, between carrying the third and fourth handshakes of a cycle, reads any global reference and marks the pointed objects local. In addition, mutators perform the following write-barrier for global references:

1. \( s \leftarrow \text{new} \)
2. if \( \text{Snoop}_i \) then
   \[
   \text{// mark new as local.}
   \]
3. \( \text{Locals}_i \leftarrow \text{Locals}_i \cup \{\text{new}\} \)

Let \( \text{MARK}_k \) be the time at which procedure Mark is invoked in cycle \( k \). The next lemma shows that any object which is allocated by some thread prior to the response to the fourth handshake is interpreted by the collector as “unmarked”. i.e., it assumes the color of \( \text{white} \oplus \text{MARK}_k \) when tracing starts.

**Lemma C.1** Let \( s \) be a slot such that \( V_k(s) = o \neq \text{null} \). Then \( o.\text{color} \oplus \text{MARK}_k = \text{white} \oplus \text{MARK}_k \).

**Proof.** if \( V_k(s) \neq \text{null} \) then \( s \) must be allocated prior to the fourth handshake, and so must be \( o \), the referred object. If \( o \) is allocated after the fourth handshake of the previous tracing cycle, then by the code, it is colored using the previous black color, which is considered the white color of cycle \( k \).

Otherwise, \( o \) has been allocated prior to the fourth handshake of the previous tracing cycle. As such, it has been examined by the sweeping process of that cycle and was found to be marked, or otherwise it would have been recalaimed. Again, due to the color toggle, it is considered white in the tracing of cycle \( k \).

From the above lemma we conclude that any object which is reachable by a chain of objects, induced by \( V_k \), where the first object is marked local, will be eventually blackened since, by the arguments from the proof of the reference counting sliding view algorithm, tracing indeed proceeds according to \( V_k \) and all objects referenced by the chain are colored white when tracing starts, so there is no obstacle in tracing through a referenced object, i.e., the “if” in procedure Trace is bound to succeed exactly once for any object in the chain. We take advantage of this observation in the next lemma which proves that elderly reachable objects are not reclaimed by mistake.

**Lemma C.2** Let \( o \) be an object which is allocated by thread \( T_i \) before \( H4k(i) \) and which is reclaimed during cycle \( k \). For each thread \( T_l \) it holds that \( o \) is inaccessible from \( T_l \) from \( H4k(i) \) onward.

**Proof.** We assume that the threads are ordered by their response time to the fourth handshake, i.e., \( H4k(1) < H4k(2) \ldots < H4k(n) \). We prove that the claim holds by induction on the
events in the algorithm’s execution. For the basis we have to show that when \( T_1 \) responds to the fourth handshake, \( o \) is reachable neither from any of \( T_1 \)’s local references nor from any global root. Suppose the contrary, \( o \) could not have been directly reachable from \( T_1 \) at the time of the handshake or it would have been marked local and thus not reclaimed. \( o \) could not have been directly reachable from a global reference at \( H S4_k(1) \) as the collector reads any global root prior to the fourth handshake and marks the read objects local. Any store into a global reference that is scheduled between the time the collector read the reference and \( H S4_k(1) \) is bound to snoop its operand, as the \( \text{Snoop}_P \) flags are all set at \( H S4_k(1) \) and updates are non-interruptible.

So the only remaining option is that \( o \) is indirectly reachable from \( T_1 \) or from a global reference at \( H S4_k(1) \). That is, there exists a local reference of \( T_1 \) or a global reference \( r \) such that at \( H S4_k(1) \):

\[
r = x_1 \land \exists s_1 \in x_1 : s_1 = x_2
\]

\[
\exists s_2 \in x_2 : s_2 = x_3
\]

\[
\vdots
\]

\[
\exists m \geq 1, s_m \in x_m : s_m = x_{m+1} = o
\]

If the chain existed in this exact form in \( V_k \), i.e., \( \forall j \leq m : s_j @ H S4_k(1) = V_k(s_j) \), then tracing through \( r \) will eventually blacken \( o \), according to the observation. If, on the other hand, there exists a slot \( s_i \) in the chain which has been modified since \( x_i(s_i) \) then let \( s_i \) be the highest indexed slot with a modified value, that is, \( \forall l < j \leq m : s_j @ H S4_k(1) = V_k(s) \). By lemma B.2 we know that when the pointer to \( x_{l+1} \) was stored into the slot \( s_i \), the storing thread mark \( x_{l+1} \) local, thus we have the chain of objects from \( x_{l+1} \) to \( x_m \) with \( x_{l+1} \) marked local and the entire chain contained in \( V_k \), we conclude that each element in the chain will be blackened, \( o \) included. We have proved the claim, restricted to the interval \([H S4_k(1), H S4_k(1)]\).

We now show that if the claim holds in the interval \([H S4_k(1), H S4_k(i - 1)]\), where \( i > 0 \) then it holds in the interval \([H S4_k(i - 1), H S4_k(i)]\). Specifically, we have to show that:

1. \( o \) remained inaccessible during the interval from any local reference of any thread in the set \( P = \{ T_1, \ldots, T_{i-1} \} \).
2. \( o \) remained inaccessible during the interval from any global reference.
3. \( o \) was inaccessible from \( T_i \) at \( H S4_k(i) \).

In order to prove the claims (1) and (2) we note that any individual load or store operation by a thread \( T_i \in P \) cannot render \( o \) reachable from \( T_i \) if it was unreachable before the operation was scheduled. Similarly, a load by a thread \( T_j \notin P \) cannot make \( o \) accessible to any thread in \( P \). We conclude that the only possibility that an object will become reachable again from a thread in \( P \) is due to a store operation carried out by a thread which is not in \( P \). We now show that such a store is impossible.

Assume, by way of contradiction, that the claim holds in the interval \([H S4_k(i-1), t] \) where \( H S4_k(i-1) \leq t < H S4_k(i) \) and that \( T_j \notin P \) indeed executes a store of a reference to the object \( x \) into a slot or a global reference which renders \( o \) reachable from some thread in \( P \) at time \( t \). Thus, the claim breaks for the first time at time \( t + 1 \).

Note that when the reference to \( x \) is stored, it is marked local, since \( T_j \) has its \( \text{Snoop}_j \) flag set during the interval. Now there are three possibilities:

- if \( x \) and \( o \) are the same object then \( o \) is marked local and thus not reclaimed later.
• otherwise, if the chain of references that exists from \( x \) to \( o \) (note that \( x \neq o \)) at the time of the store exists in \( V_k \) as well, then \( o \) will be eventually blackened, according to our observation.

• finally, if the chain that exists at time \( t + 1 \) and \( V_k \) differ in some point, then we again consider the longest suffix of the chain which hasn’t been modified relatively to \( \sigma_k \). Denote the first object in the suffix \( y \). When the pointer to \( y \) was stored into the slot referring to it in the chain, \( o \) was reachable from the storing thread. Since this operation took place prior to the current operation, we can apply the inductive hypothesis for it and deduce that the storing thread could have not responded to the fourth handshake before executing the update. Thus, it must have marked \( y \) local. The claim then follows.

In order to prove the second claim we assume by way of contradiction that \( o \) is indeed reachable from \( T_i \) at \( HS_{4_k}(i) \). Again we note that if \( o \) is directly reachable from either a local or a global reference, or reachable through a non-empty chain which exists in \( V_k \), then it will be blackened. Thus, \( o \) must be reachable by a chain which differs in some point from its respective values in \( V_k \). By arguing that the reference to the first object in the longest suffix of the chain mutual to time \( HS_{4_k}(i) \) and \( V_k \) was stored to its referring slot in the chain by a thread which still hasn’t responded to the fourth handshake we again conclude that \( o \) will be eventually blackened.

\[ \square \]

We conclude that:

**Theorem C.1** The tracing sliding view algorithm is safe.