On-The-Fly Garbage Collection Via Sliding Views

Yossi Levanoni

On-The-Fly Garbage Collection Via Sliding Views

Research thesis
submitted in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science

Yossi Levanoni

Submitted to the Senate of the Technion—Israel Institute of Technology
Elul 5760 Haifa September 2000

The research thesis was done under the supervision of
Dr. Erez Petrank in the Department of Computer Science.

I thank Dr. Erez Petrank for his excellent guidance, knowledge and experience that were a source of inspiration for me throughout the stages of the work.
My sincere thanks to my wife Dorit for her support.
I am grateful to my mother, Amalia Levanoni, who made this work possible.
I thank Dr. Hillel Kolodner for his helpful review comments.

The generous financial help of the Gutwirth Fund is gratefully acknowledged.

Contents

List of Figures

List of Tables

Abstract

This work presents the first on-the-fly reference counting based automatic memory management algorithm with that requires no synchronization overhead in the basic memory management operations: pointer update and object allocation. The algorithm is based on a general concept, termed *Sliding View*, which is related to the atomic snapshot concept. We use the same theoretic foundation of the Sliding View to define a second garbage collection algorithm based on tracing. Since the two algorithms rely on the same foundation, they coexist and complement each other at runtime. We have implemented an integrated garbage collector containing both algorithms for Javasoft's virtual machine and have witnessed substantial performance gains.

The increasing popularity of garbage collected programming languages, especially due to the introduction of the Java programming language, has triggered a renewed interest in garbage collection. Specifically, the traditional methods of garbage collection are re-evaluated and are being adapted to contemporary computing models and programming languages.

In particular, garbage collection algorithms are being adapted to one computing model that has gained enormous acceptance in both the industry and the academic world as a most viable computing model during the past decade—the shared memory multiprocessing model. Due to this acceptance, shared memory multiprocessor machines abound as servers and as powerful desktop stations.

While shared memory multiprocessing is not a new concept the fact that the price of random access memory components had dropped recently calls for changes in the way the shared memory platform is utilized. Whereas in the past an emphasis has been put on maximizing the locality of programs in order to avoid page faults, page faults nowadays are less of a concern as the entire heap is backed by cheap random access memory and therefore faults are scarce. Instead, an emphasis must now be placed on maximizing the throughput of the processors in a shared memory platform by avoiding synchronization bottlenecks and choosing scalable algorithms in terms of growing number of threads, growing heap sizes and growing heap occupancy factors.

The reference counting algorithm presented in this work achieves all of the above goals. First, it never halts the system nor does it requires synchronization in the basic operations of memory update and object allocation (not even a compare-and-swap type of operation). User threads are only required to cooperate with the collector one at a time, for short durations and infrequently (four times per collection cycle). This fine synchronization, except for eliminating synchronization bottlenecks, also enables scalability in the number of threads, as no action requires the cooperation of more than one thread simultaneously. Second, the asymptotic behavior of the reference counting algorithm does not depend on the heap occupancy nor on the heap size. This is in sharp contract with the *de facto* standard methods of mark-and-sweep and copying.

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. The new algorithm presented in this work, on the other hand, employs extremely fine synchronization. Furthermore, the algorithm is non-disruptive: the program threads are never stopped simultaneously to cooperate with the collector. Thus, the program can run with (almost) no synchronization overhead imposed by the collection.

The on-the-fly reference counting algorithm is complemented by an on-the-fly tracing collector which uses the same principals and shares most of the code and data structures of the reference counting collector. This enhances the integration of the two algorithms and allows the reclamation of cyclic data structures which cannot be reclaimed directly by the reference counting collector.

We develop the concept of a sliding view which is an approximated atomic snapshot of the heap. In the reference counting algorithm, collection is based on the differences between two consecutive sliding views: the sliding view associated with the previous cycle and the sliding view associated with the current cycle. In the tracing algorithm, we trace through an image of the heap described by the sliding view associated with the current cycle. Thus, common to both algorithms is the calculation of the sliding view. The validity of the algorithm stems from the fact that both the collector and user threads strive to maintain just enough information records in order to construct a sliding view.

We have implemented both algorithms for Java and measured their efficiency in both server and client environments. We witnessed a dramatic improvement in response time while retaining or exceeding the throughput of the original JVM.

List of Symbols and Abbreviations

GC Garbage Collection

JVM The Java Virtual Machine

RC Reference Counting

SMP Symmetric Multi Processing or Symmetric Multi Processor

t a dimentionless, discrete, time instance

$E@t$ the value of the expression E at time t

o the address of an object

s an object field. Also called an object slot

$RC(o)$ the heap reference count of object o

$o.rc$ the value of the reference counter of object o

T_i thread identifier

R_k an atomic snapshot of the heap

V_k a sliding view of the heap

$Hist_k$ a history

$Hist_k(s)$ the value associated with s in the history $Hist_k$

Chapter 1

Introduction

Automatic memory management is well acknowledged as an important tool for fast development of large-scale reliable software. However, 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[1] run on a single thread 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, [9, 20, 7, 17, 18, 12, 22, 40, 21]). 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 comparing 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 arbitrary point in the instruction stream. 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 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 collection work is not done on parallel on all available processors (which is usually the case), then during the time the program threads are stopped expensive processing power is wasted.

Therefore, it is advantageous to use *on-the-fly* collectors [20, 22, 21]. On-the-fly collectors never stop the program threads simultaneously. Instead, each thread cooperates with the collector at its

[1] Among them the collector supplied with Javasoft's Java Virtual Machine.

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 [29, 16, 38, 30, 23, 34]. This approach does not involve wastage of processing power, yet it still imposes a possibly long pause time on the user threads. 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. [15].) 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 directly by the object o) are decremented, initiating perhaps more reclamations.

The key advantage of reference counting for traditional uniprocessor environments 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. [19, 42, 52, 13, 28]). While the computational overhead is reduced by some 80% using Dutsch and Bobrow's *Deferred Reference Counting* [19]. Only the inability to reclaim cyclic structures does not have a satisfactory solution intrinsic to reference counting [51, 33] 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.

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 [17] 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 and implement 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.[2]

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 *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 chapter4 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 chapter4 below. The time the mutators are simultaneously stopped is short: all that 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 [17], 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 concurrent 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 on-the-fly 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

[2]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.

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 on-the-fly 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 [22, 21].

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) amortized 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 Organization

We start we survey of related work in chapter2. In chapter3 we present definitions and terminology to be used in the rest of the paper. In chapter 4 we present our Snapshot algorithm. Chapter5 describes the Sliding View algorithm. Chapter6 introduces our Tracing Sliding View algorithm. In chapter7 we discuss several implementation issues of the proposed algorithms. Chapter8 describes our implementation of the algorithms for Java. In chapter9 we describe the performance results achieved with the Java implementation. We provide proofs for the Snapshot Algorithm, Reference Counting Sliding View Algorithm and Tracing Sliding View Algorithm in chapterA, B and C of the appendix, respectively. ChapterD of the appendix contains the source code of the implementation. We conclude in chapter10.

Chapter 2

Related work

The traditional method of reference counting, applicable in the realm of uniprocessing, was first developed for Lisp by Collins [15]. 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 reference counters. As such, it was used in applications requiring responsiveness that could not tolerate delays yet could stand the incurred space and computational overhead such as Smalltalk-80 [27] and the AWK [5] and Perl [48] programs.

Weizman showed in [49] 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 [19] 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 to 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+ [17]. Several studies [39, 46, 6, 57] showed that the assumption about the relatively low frequency of store operations is usually valid. Baker in [10] 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 [41] 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 [42, 53] 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 [45]. Additional schemes that use single-bit reference counters are those by Chikayama and Kimura [13] and by Goto *et al.* [28].

DeTreville describes in [17] 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 simultane-

ously 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.* [25] 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 [56] 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 [51] makes the distinction between *snapshot-at-beginning* and *incremental update* algorithms. Trying to apply the terms to our on-the-fly 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 [22, 21]: 1) we never require a full halt of the system; 2) mutators are required to cooperate four times per cycle ([22, 21] 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 [22, 21].

2.1 A complementary work

Independently of this work, Bacon *et. al.* [1] have also presented an on-the-fly reference counting algorithm. Both works introduce on-the-fly collector with extremely low pause times. But, whereas the focus in our work is in obtaining high efficiency through extremely fine synchronization (in the write barrier), the work in [1] focuses on a novel on-the-fly cycle detection method. Collecting cycles on-the-fly is a task that could not be done before. In this work, we have not dealt with this problem. Instead, we employed a mark and sweep collector (run seldom) to collect cycles. We believe that the methods in [1] can be combined with ours to obtain an efficient reference counting collector with no need for an external tracing collector.

Chapter 3

System Model, Definitions, Symbols and Abbreviations

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*[1]. 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 multi threaded 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 section7.3 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.

[1] We 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.

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.[2] This assumption allows us to conveniently define global state and time as follows:

Definition 3.1 (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 shared memory operations corresponding to the execution.*

Definition 3.2 (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 3.3 (Reachability) *We say that an object o is*

- **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 3.4 (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*[3]. In any conceivable reference counting system 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 $o.rc$. 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

[2] Thus, we assume that the shared-memory is sequentially consistent. In section7.4 we show how the memory model constraints may be relieved in order to adapt the algorithms we present to systems with weaker memory models.

[3] 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 [19].

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

Which means that all user threads are halted in unprotected code when **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 **Do-Something** operation.

A *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. **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 [22, 21], 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.

Chapter 4

The Snapshot Algorithm

In this chapterwe 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.

4.1 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 *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 *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 rc field of the object referred to by s in R_k
2. decrements the 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 rc field, that for any object o, at the time the collector completes adjusting 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 *local* has no references to it whatsoever in the system and may be reclaimed.

4.2 Implementing the algorithm efficiently

Implementing the algorithm efficiently entails two major issues:

- **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* [19] 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 $Dirty(s)$. 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 $Dirty(s)$ 3) if $Dirty(s)$ is off it saves a record of the pair $\langle s, v\rangle$ 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 $Dirty(s)$. 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.

4.3 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.

4.4 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 merges 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*.

4.5 Data structures

In this sectionwe 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 $Buff_i$, and it contains pairs of the form $\langle s, v \rangle$ 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 $\langle s, v \rangle$ 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.

4.6 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 4.1)and **New** (figure 4.2) 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 operations.

```
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 4.1: 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 4.2: Mutator Code—for Allocation

4.7 Collector code

The code for cycle k is given in procedure **Collection-Cycle**, in figure 4.3. Each of the procedures invoked during a cycle is now described.

Procedure Read-Current-State (figure 4.4). After all threads are stopped their local state, new object sets and buffers are delivered to the collector. Before resuming the threads the collector clears all dirty marks.

Procedure Update-Reference-Counters (figure 4.5). 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 4.6) asynchronously reads threads' buffers. Each thread T_i is considered at a time. The variable $CurrPos_i$ is probed. Then the range $[1 \ldots CurrPos_i - 1]$ of $Buffer_i$ (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 Collection-Cycle
begin
1.    Read-Current-State
2.    Update-Reference-Counters
3.    Read-Buffers
4.    Fix-Undetermined-Slots
5.    Reclaim-Garbage
end
```

Figure 4.3: Collector Code

```
Procedure Read-Current-State
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 4.4: Collector Code—Procedure **Read-Current-State**

```
Procedure Update-Reference-Counters
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}
```

Figure 4.5: Collector Code—Procedure**Update-Reference-Counters**

```
Procedure Read-Buffers
begin
1.    Peek_k := ∅
2.    for each thread T_i do
3.        local ProbedPos := CurrPos_i
          // copy buffer onto Peek_k.
4.        Peek_k := Peek_k ∪ Buffer_i[1 ... ProbedPos − 1]
end
```

Figure 4.6: Collector Code—Procedure **Read-Buffers**

```
Procedure Fix-Undetermined-Slots
begin
1.    for each pair ⟨s, v⟩ pair in Peek_k
2.       if s ∈ Undetermined_k do
3.          v.rc := v.rc + 1
end
```

Figure 4.7: Collector Code—Procedure **Fix-Undetermined-Slots**

```
Procedure Reclaim-Garbage
begin
1.    ZCT_{k+1} := ∅
2.    for each object o ∈ ZCT_k do
3.       if o.rc > 0 then
4.          ZCT_k := ZCT_k − {o}
5.       else if o.rc = 0 ∧ o ∈ Locals_k then
6.          ZCT_k := ZCT_k − {o}
7.          ZCT_{k+1} := ZCT_{k+1} ∪ {o}
8.    for each object o ∈ ZCT_k do
9.       Collect(o)
end
```

Figure 4.8: Collector Code—Procedure **Reclaim-Garbage**

Procedure Fix-Undetermined-Slots (figure 4.7) 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 4.8). 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 4.9) 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 accordingly. The referred objects are recursively deleted based on the same criteria applied by the **Reclaim-Garbage** procedure.

4.8 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

```
Procedure Collect(o: Object)
begin
1.    foreach slot s in o do
2.       val := read(s)
3.       val.rc := val.rc − 1
4.       write(s, null)
5.       if val.rc = 0 then
6.          if val ∉ Locals_k then
7.             Collect(val)
8.          else
9.             ZCT_{k+1} := ZCT_{k+1} ∪ {val}
10.   return o to the general purpose allocator.
end
```

Figure 4.9: Collector Code—Procedure **Collect**

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 section7.1.

The algorithm's correctness proofs are in appendix A.

Chapter 5

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 chapter, 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 collection 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.

sectionScans and sliding views

Pictorially, a scan σ and the corresponding sliding view V_σ 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* σ 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 σ. Similarly we define $End(\sigma)$.

Formally, the *sliding view* associated with a scan σ, which is denoted V_σ, is a function that assigns a pointer value to each memory word s in the heap:

$$V_\sigma(s) \stackrel{\text{def}}{=} \begin{cases} \textbf{null} & \text{if } s \text{ is not an allocated slot at } \sigma(s) \\ s@\sigma(s) & \text{otherwise} \end{cases}$$

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_σ we define the *Asynchronous Reference Count of o with respect to* V_σ to be the number of slots in V_σ referring to o:

$$ARC(V_\sigma; o) \stackrel{\text{def}}{=} |V_\sigma^{-1}(o)|$$

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 σ set to: $\forall s, \sigma(s) = t$. Then we have:

$$\forall \ Object \ \ o, ARC(V_\sigma; o) = RC(o)@t$$

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 5.1 (Sliding Views) *Let V_σ 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_σ*

sectionUsing 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 *local*. We call this probing of the $Snoop_i$ flag and the subsequent marking *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 σ and a corresponding sliding view, V_σ, concurrently with mutators' computations. The actual manner using which the collector computes V_σ 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 *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 *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 local satisfies, according to lemma 5.1, $ARC(V_\sigma; o) \geq RC(o)@End(\sigma)$ thus $0 = o.rc = ARC(V_\sigma; o)$ implies $RC(o)@End(\sigma) = 0$. It may be, however, that o is *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 $o.rc = ARC(V; o)$, 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.

sectionAlgorithm's idea We will present a concrete sliding view based reference counting algorithm which implements the generic sliding view algorithm of the previous chapter.

The concrete algorithm uses ideas similar to those presented in the context of the snapshot algorithm of chapter4. 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 reference 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.

5.1 Overview of mutator's cooperation

Mutators use the write barrier of the snapshot algorithm (figure 4.1) with the additional snooping and marking added after the store proper. Object creation is unchanged from the snapshot algorithm.

5.2 Overview of the collection cycle

Each collection cycle is comprised of the following steps, which are visually illustrated in figure 5.1:

Signaling snooping. The collector raises the $Snoop_i$ 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[1] 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[2].

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.

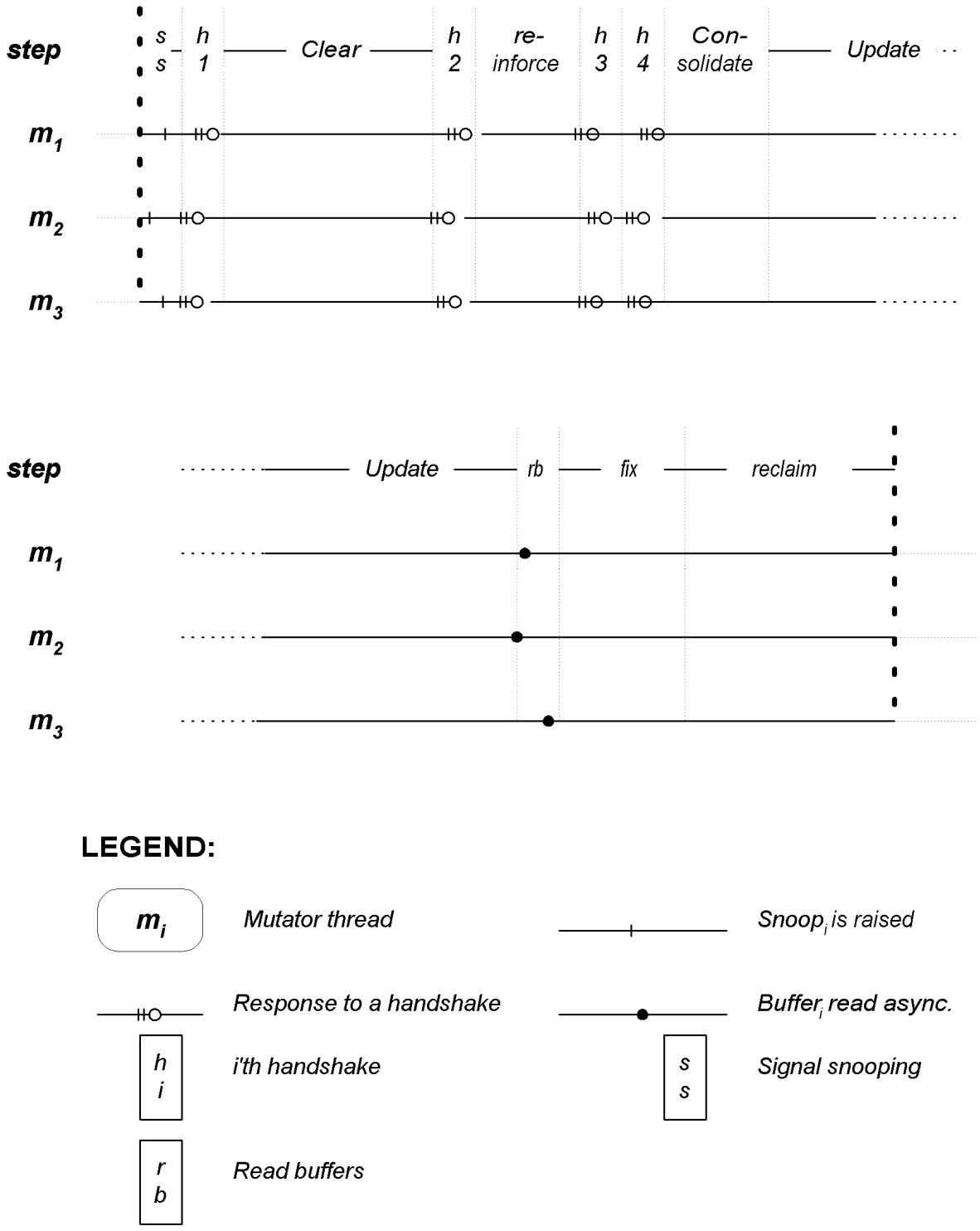
The digested history replaces the accumulated threads' buffers. i.e., the history for the next cycle is comprised of the digested history of threads' logging between the first and fourth handshakes of the current cycle, unified with threads' buffers representing updates that will occur after the fourth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

[1] These are the same thread buffers as in the first algorithm.

[2] 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 $k+1$.



Meaning of other abbreviations: Clear: clearing dirty Marks. Reinforce: reinforcing conflict slots. Consolidate: consolidating thread buffers. Update: updating reference counters. fix: incrementing reference counters due to undetermined slots. Reclaim: reclaiming garbage objects.

Figure 5.1: Timing diagram for the sliding view algorithm

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.

5.3 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 σ_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 *local*.

```
Procedure Update(s: Slot, new: Object)
begin
1.    Object old := read(s)
2.    if ¬Dirty(s) then
3.       Buffer_i[CurrPos_i] := ⟨s, old⟩
4.       CurrPos_i := CurrPos_i + 1
5.       Dirty(s) := true
6.    write( s, new)
7.    if Snoop_i then
8.       Locals_i := Locals_i ∪ {new}
end
```

Figure 5.2: Sliding View Algorithm: Update Operation

It turns out that these requirement 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 determining 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" $\langle s, v\rangle$[3] 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 *Snoop* 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.

5.4 Mutator code

Mutator code in the second algorithm is almost identical to the one in the first algorithm. In particular, the **New** procedure is unchanged.

The **Update** procedure (in figure 5.2) includes an additional test, that checks whether the thread-specific flag $Snoop_i$ is set. If so, the object whose reference is stored into the slot is marked *local* by adding it to the thread-specific set $Locals_i$. This marking implements the "snooping" requirement of the generic algorithm.

[3] "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 5.3: 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 ∪ Buffer_i[1 ... CurrPos_i − 1]
6.       CurrPos_i := 1
7.       resume T_i
end
```

Figure 5.4: Sliding View Algorithm: Procedure **Initiate-Collection-Cycle**

5.5 Collector code

Collector's code for cycle k is depicted in figure 5.3. Let us describe briefly the role of each of the collector's procedures.

Procedure Initiate-Collection-Cycle (figure 5.4) is the counterpart of procedure **Read-Current-State** of the snapshot algorithm of chapter4. 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 $Snoop_i$ flag is raised, signaling mutators that they should start snoop stores into heap slots.

2. the set $Hist_k$ 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 $k-1$.

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 5.5) 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 Clear-Dirty-Marks
begin
1.    for each ⟨s, o⟩ ∈ Hist_k do
2.        Dirty(s) := false
end
```

Figure 5.5: 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 ... CurrPos_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 5.6: Sliding View Algorithm: Procedure **Reinforce-Clearing-Conflict-Set**

Procedure Reinforce-Clearing-Conflict-Set (figure 5.6) 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 during the clearing or reinforcement stages, i.e., threads that still haven't responded to the third handshake.

Procedure Consolidate (figure 5.7). 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 $Locals_i$ 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 4.5, 4.6, 4.7). Note, however that there is an additional procedure, **Merge-Fix-Sets** (figure 5.8), 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 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 ∪ Locals_i
7.       Locals_i := ⊘
         // copy thread local state and ZCT.
8.       Locals_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[1 ... CurrPos_i − 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 5.7: Sliding View Algorithm: Procedure **Consolidate**

```
Procedure Merge-Fix-Sets
begin
1.    Peek_k := Peek_k ∪ Hist_{k+1}
end
```

Figure 5.8: Sliding View Algorithm: Procedure **Merge-Fix-Sets**

Procedure Reclaim-Garbage (figure 4.8) 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 5.9) 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 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 $\langle s, v\rangle$ 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.

```
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 5.9: Sliding View Algorithm: Procedure **Collect**

Chapter 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 inter-operable 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 sectiondemonstrates 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* [51, 25] 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 and 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 σ and a corresponding sliding view V_σ. 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 σ and a corresponding sliding view, V_σ, concurrently with mutators' computations.
  3. each thread is then suspended (one at a time) and its $Snoop_i$ 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_σ. 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 [35]. 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 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).

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 $AllocColor_i$, that holds the color the thread has to *color*, i.e, assign, to the *color* field of newly allocated objects. The variable is toggled between two

```
Procedure New(size: Integer) : Object
begin
1.    Obtain an object o from the allocator,
      according to the specified size.
2.    o.color := AllocColor_i
3.    New_i := New_i ∪ {o}
4.    return o
end
```

Figure 6.1: Allocation code that supports tracing cycles

dichotomic colors, *black* and *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 *black* color to the $AllocaColor_i$ variable. Thus, during tracing and sweeping the mutator colors newly allocated objects *black*. During sweeping, the collector considers each object in heap. If the object is *black*, then it is retained. If it is colored *blue*, then it is ignored. Otherwise, the object is *white*. In that case the collector reclaims the object by coloring it *blue* and passing it back to the allocator.

Thus, when sweeping is over, the heap contains only *black* or *blue* objects since any object which had been *white* was turned *blue* and mutators color newly allocated objects *black*. Before starting the tracing of the next cycle the collector toggles the values of *black* and *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 **Update** procedure is given in figure 5.2.

The **New** procedure is modified to support both tracing and reference counting cycles. **New** carries out the mutator's part in the object coloring protocol. The modified procedure is given in figure 6.1.

6.4 Collector Code

The code for a tracing collection cycle is given in figure 6.2. 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 5.4, 5.5 and 5.6, 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 6.3, 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 $AllocColor_i$ 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

```
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 6.2: Tracing Alg.—Collector Code

newly created object *black*. Another thing to note is the omission of the addition of the New_i sets to ZCT_k. Actually, ZCT_k has no use in a tracing cycle.

Procedure Mark (figure 6.4) 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 $Locals_k$. 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 reliably can 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 $Hist_{k+1}$ (i.e., the digested history of threads' modifications to heap slots between the first and fourth handshakes). 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 $ScannedPos_i$ which equals the value of $CurrPos_i$ at the time the thread buffer was most recently read, during the previous phase.

Procedure Trace (figure 6.5). 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

```
Procedure Consolidate-For-Tracing
begin
        // initially 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 ∪ State_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⟩ ∈ Temp
19.         if s ∉ Handled then
20.             Handled := Handled ∪ {s}
21.             Hist_{k+1} := Hist_{k+1} ∪ {⟨s, v⟩}
end
```

Figure 6.3: Tracing Alg.—Procedure **Consolidate-For-Tracing**

```
Procedure Mark
begin
1.    for each thread T_i do
2.       ScannedPos_i := 1
3.    Hash-Clear
4.    for each pair ⟨s, v⟩ ∈ Hist_{k+1} do
5.       Hash-Insert(s, v)
6.    Undetermined := ∅
7.    for each object o ∈ Locals_k do
8.       Trace( o, 0 )
9.    while Undetermined ≠ ∅ do
10.      for each thread T_i do
11.         ProbedPos := CurrPos_i
12.         while ScannedPos_i < ProbedPos do
13.            ⟨s, v⟩ := Buff_i[ScannedPos_i]
14.            Hash-Insert(s, v)
15.            ScannedPos_i := ScannedPos_i + 1
16.      PrevUndetermined := Undetermined
17.      Undetermined := ∅
18.      for each slot s ∈ PrevUndetermined do
19.         v := Hash-Lookup-And-Remove(s)
20.         Trace( v, 1 )
end
```

Figure 6.4: Tracing Alg.—Collector Code—Procedure **Mark**

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 *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 virtue of being pointed from a heap slot or rather by a local reference. In the latter case, no adjustment to *o.rc* is needed, while in the former *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 6.6). The first step it takes is to eliminate from $Hist_{k+1}$ any records of slots that it is about to reclaim. This stage is needed in order that the next cycle will not adjust *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 *black* and has a zero computed reference count field is added to the ZCT of the next cycle (anticipating a reference counting cycle.) *White* objects are returned to the allocator not before being colored *blue*. *Blue*

```
Procedure Trace(o: Object, rcInc: Integer
begin
1.    if o.color = white then
2.       o.color := black
3.       o.rc := 0
4.       for each slot s of o do
5.          v := read(s)
6.          if ¬Dirty(s) then
7.             Trace(v,1)
8.          else
9.             Undetermined := Undetermined ∪ {s}
10.   o.rc := o.rc + rcInc
end
```

Figure 6.5: Tracing Alg.—Collector Code—Procedure **Trace**

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

Figure 6.6: Tracing Alg.—Collector Code—Procedure **Sweep**

objects are ignored.

Chapter 7

Implementation Issues

In this chapterwe 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.

After considering the implementation options described in this chapterwe describe in chapter8 and 9 the actual implementation we prepared and its performance results.

7.1 Dirty Flags

Both algorithms were presented in a rather high level and generic manner that leaves the implementation 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 section7.1.1 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. Section7.1.2 demonstrates how the overhead of initializing assignments can be eliminated.

7.1.1 Allotting a flag per a chunk of memory

In this sectionwe 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 sectionare 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 [43].

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).

7.1.2 Initialization

This sectiondiscusses 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 [57, 31] 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.

7.2 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 [17]: 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.

7.3 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 chapterB.

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.

7.4 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 sectionwe 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.

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[1].

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 don't know as well how to eliminate these opportunities and we may rely only on the constraints provided by processor ordering n 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:

```
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.

[1] It may involve flushing the processor's pipeline and/or cache.

However, in practice, we can identify the origins of reordering and therefore we can take advantage 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:

```
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 `(val & 3)` is bound to equal zero and `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 $Snoop_i$ 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 handshakes 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 section7.2.)

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.

Chapter 8

An Implementation for Java

We have implemented a variant of the Asynchronous Reference Counting and Asynchronous Tracing algorithms for Java. This chapterdescribes the implementation and its performance characteristics.

8.1 Java—the target platform

There is probably no need to introduce Java [3]. We chose to implement the Asynchronous Reference Counting and Tracing algorithms atop Java because of the following reasons:

1. Java is an object-oriented garbage-collected language. Obviously it needs *some* garbage collector.

2. Java is very popular and accepted as a true, rather than just academic, programming language. This allows us to check our algorithm in a realistic setting.

3. Java supports shared-memory multi-threading in the language level. The need for a garbage collector that can handle multiple threads running concurrently on multiple processors and referring to the same address space is inherent to Java.

4. Java has been recently portrayed as the language of choice for implementing portable servers (consult, for example, [4], for a coverage of contemporary server-side java based technologies). An obstacle to overcome on the path to achieving scalability for such servers is the scalability of the garbage collection process. This is exactly what we aim at in our work.

We started with Sun's JDK1.2.2 for Win32 and replaced the default collector supplied with the JDK with our on-the-fly collectors.

8.2 Object structure and garbage collection in the original Java Virtual Machine

The original Java Virtual Machine (JVM) supports a so-called "handled" model in which each object is referenced indirectly through a handle. The object itself contains the actual data members declared by the object's class while the handle contains two pointers: the first is a pointer to the object data; the second points to a memory block containing the class's runtime information (virtual table, reflection information etc.)

The heap is divided into two disjoint pools: the object pool and the handles pool.

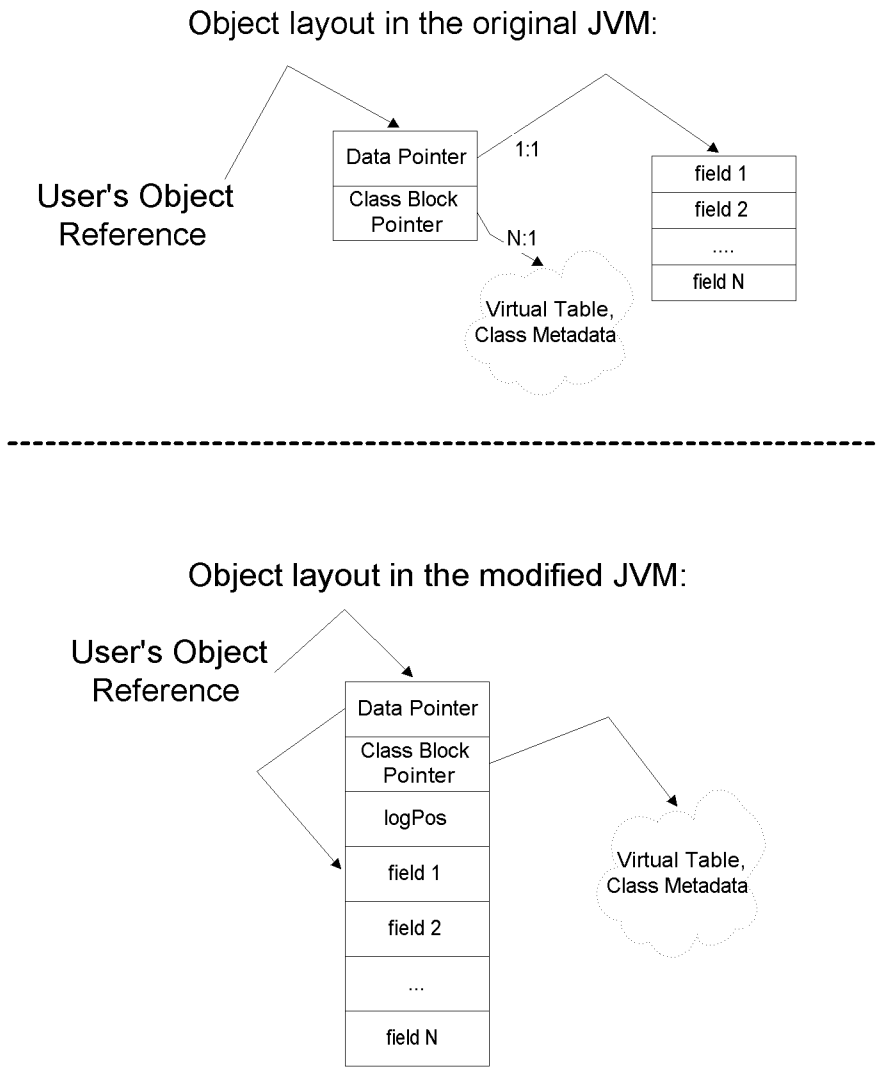


Figure 8.1: Object layout in the original and modified JVMs. In the original JVM, data is accessed indirectly through a *handle* in order to support the relocation of object data. In the modified JVM, object data is almost always referenced directly by the user yet the data pointer is retained for compatibility. The logPos field is either **null** or a pointer to a log entry that contains the logged object's reference data.

In order to allocate an instance of a class, an object and a handle are allocated, the object is zeroed out and the handle is initialized to point at the object and at the class's runtime information. See figure 8.1.

Handles are completely transparent to the user. They are used in order to facilitate memory compaction [33] yet they introduce to the system the overhead of extra indirection, decreased locality of reference and increased memory consumption (due to the handle to object pointer).

The garbage collection method which is employed in the original JVM is mark-sweep-compact. Garbage collection occurs in a stop-the-world manner, when all threads are stopped.

8.3 Object structure in the modified JVM

Since our collector does not support the moving of objects in memory, we derive no advantage from using the handles. However, eliminating the handles from the original JVM was too complicated a task to undertake. As a compromise, we have unified the handle with the object (see figure 8.1). The handle and object are allocated as a single chunk of memory and are treated as such by the memory manager. This layout increases locality. Additionally, an object is located at a fixed offset from its handle. Therefore we were able to change most of the code in the JVM to calculate the object's address, given the handle's address, by a simple add instruction rather than de-referencing the object pointer inside the handle.

We have based our implementation on the "flag per object" scheme discussed in section7.1.1. The flag is termed `logPos` and is located between the (original) handle and the object (see figure 8.1). As the name of this field implies, it is not a mere flag but it has added functionality: when the field is non-zero then indeed the object has been logged, as in the original scheme. Moreover, the value of the field in such a case is a pointer to the location in the thread's buffer where the object's contents have been logged. Below we give the pseudo-code of the **Update** procedure given this policy:

```
void updateHandle( Handle *h, int offset, Handle *value )
{
   if (h->logPos==NULL) {
      /* object has not been logged yet */
     Replica r = copies of all references contained in 'h'

     /* check if object was modified in the meanwhile */
     if (h->logPos==NULL) {
        /* OK, replica is valid, commit it */
        /* write into log, remember position */
        LogEntry *le = logIntoUpdateBuffer( h, r)

        /*
        * write position into the object and
        * thereby also turn on the flag
        */
        h->logPos = le;
     }
   }

   /* do the store proper */
   h[offset] = value;
```

```
   /* snoop store operand */
   if (currentThread->snoop) {
     /* write 'value' into snoop buffer */
     ...
   }
}
```

Note that we optimistically probe the `logPos` pointer before preparing the replica of the object's contents, hoping to minimize the number of replicas which are eventually discarded. Later in this chapterwe show that the number of updates that actually execute the "if" body is very small and therefore the use of the optimistic conditional is indeed beneficial.

8.4 Simplifying the determination of object's contents using the logPos field

By paying the extra price of allotting a whole word for the flag and transforming it into a pointer that identifies the logged contents of an object, rather than using a boolean byte-sized flag, we obviate all the cases in the original asynchronous algorithms in which the collector *failed determining* an object and had to take extra and elaborate measures to deal with the failure. This includes:

Determining the contents of an object when updating reference counters. In the sliding view reference counting algorithm, when updating the reference counters of modified slots (see procedure **Update-Reference-Counters** in figure 4.5) the collector had to *determine* the contents of a logged slot.

In our case, the collector has to determine the contents of a logged *object*.

In the original procedure, if the slot is *undetermined* the collector knows that some thread logged it along with its value but it has no clue which thread did the logging and where to find the log entry. It therefore postpones the dealing with such a slot to procedure **Fix-Undetermined-Slots** (figure 4.7) in which it iteratively goes through the mutators' log buffers and makes sure it accounts for every undetermined slot.

In the modified procedure, if the object is undetermined then the value of `logPos` gives the collector an immediate access to the log entry where a thread has logged the objects' contents. The collector therefore proceeds directly to the log entry and reads its contents. This scheme deems the *Undetermined* set and the **Fix-Undetermined-Slots** procedure unnecessary.

The following code fragment illustrates the process of determining the contents of an object.

```
void determineObjectContents(Handle *h)
{
   LogEntry le = h->logPos;

   if (le) {
      /* object has been logged */
objectIsLogged:
      for each reference 'child' logged in 'le' do
         incrementRC( child )
      return
   }
   /* Prepare a replica of the references contained
    * in the object
    */
   Replica r = copies of all references contained in 'h'
```

```
   /* check if object was modified in the meanwhile */
   le = h->logPos;
   if (le) goto objectIsLogged;

   /* OK, replica is valid */
   for each reference 'child' in the replica 'r' do
     incrementRC( child )
}
```

Determining the contents of an object when deleting it. Recall that in procedure **Collect** (figure 5.9) we had to postpone the collection of garbage objects which where modified by some thread between $HS1$ and $HS4$. The reason for that was that we had no reasonable means to eliminate the redundant log entry. Deleting the object without eliminating its corresponding log entry (or entries) would have caused inconsistencies in the next cycle.

Using the `logPos` field, however, facilitates the elimination of log entries. If the object is indeed logged then we have to:

1. decrement the reference counters of the objects appearing in the log entry (rather then the counters of the objects appearing currently inside the object).

2. invalidate the log entry.

These changes are illustrated in the following pseudo-code for `freeObject`:

```
void freeObject(Handle *h)
{
   if (h->logPos) {
      /* object has been logged */
      LogEntry le = h->logPos;
      for each reference 'child' logged in 'le' do {
         decrementRC( child) /* takes care for recursive deletion */
      }
      /* invalidate log entry, for next cycle */
      markInvalid( le );
   }
   else {
      /*
       * Delete based on current contents
       */
      for each reference 'child' contained in 'h' {
         decrementRC( child) /* takes care for recursive deletion */
      }
   }
}
```

Note the difference between `freeObject` and `determineObjectContents`: `freeObject` does not have to prepare a replica of the object's contents and then recheck its validity since we are guaranteed that the object is garbage at the time `freeObject` examines it, hence no contention with mutators is possible.

Determining the contents of an object when tracing through it. In the tracing algorithm we have to determine an object's contents when tracing through it (see procedure **Trace** in

figure 6.5). The inability to gain immediate access to the object's log entry when the object is undetermined dictated a multi-phased algorithm in which the threads' log buffers are repeatedly read asynchronously in order to read the values of modified slots (see procedure **Mark** in figure 6.4).

Using the `logPos` field in the same manner as for the case of updating reference counters eliminates the need for multiple phases and the related data structures. Tracing then always proceeds immediately after accessing the object's `logPos` field, either as dictated by the current objects' contents or according to the previous state of the object, as recorded in the log entry. Which of the two routes is taken is determined by the value of `logPos` in the same manner done in the update of reference counters.

Procedure `traceThroughObject` demonstrates the principal described above:

```
void traceThroughObject(Handle *h)
{
   /* trace only once through any reachable object */
   if (getObjectRC(h) > 1) return;

   /* trace through the object */
   LogEntry le = h->logPos;
   if (le) {
ObjectIsLogged:
      /* object has been logged */
      for each reference 'child' logged in 'le' do {
         /*
         * account for the pointer to 'child'
         * which is currently being traced
         * through.
         */
         incrementObjectRC( child );
         traceThroughObject( child );
      }
      return
   }

   /*
    * Prepare a replica of the references contained
    * in the object.
    */
   Replica r = copies of all references contained in 'h'

   /* check if object was modified in the meanwhile */
   LogEntry le = h->logPos;
   if (le) goto objectIsLogged;

   /* OK, replica is valid */
   for each reference 'child' in the replica 'r' do {
     /*
      * account for the pointer to 'child'
      * which is currently being traced
      * through.
      */
     incrementObjectRC( child );
     traceThroughObject( child );
   }
```

```
}
```

8.5 Additional advantages of the logPos field

Beside the easier determination of objects' contents we derive the following two advantages from using the `logPos` field:

Using the logPos field in the resolution of the Create vs. Sweep conflict in the sliding view tracing collector. The memory manager, in concert with the garbage collector, use the `logPos` filed, along with other means, to resolve the race condition between the **Create** and **Sweep** procedures. The net gain from this scheme is that we don't have to use a per object color entry any more. We will elaborate on this subject after describing the memory manager.

Eliminating duplicates in the update buffers. Recall that the original algorithm allowed two mutators to log the same slot (in our case object). In such a case it was guaranteed that the two log entries will be identical. The collector had to eliminate the duplicates and process exactly one log entry per each object that was logged by any number of threads.

When using the `logPos` field we have a method for identifying duplicates without using auxiliary data structures other then the log entries and the objects themselves.

We say that a log entry is "cycle closing" if the object it refers to has its `logPos` field pointing back at the log entry.

During the clearing phase, i.e., between $HS1$ and $HS2$ the collector examines the mutators' buffers that were passed to it during $HS1$. For each log entry, the collector checks whether the entry is "cycle closing". If it is, then it clears the `logPos` field of the referred object. Otherwise, it invalidates the log entry so that the rest of the collection cycle ignores it completely.

Let us explain why this method eliminates all duplicates and only duplicates. Consider an object at address `h` which is logged by at least one thread before responding to $HS1$. Assume further that this object is not logged by any thread between $HS1$ and $HS2$. Obviously, `h->logPos` is constant during the clearing phase and therefore the collector will identify exactly one log entry as cycle closing. Any other log entry referring to `h` will be marked as invalid. So the method works in this case.

What happens in the case that `h` is logged by at least one thread before $HS1$ but it is also logged by some thread between $HS1$ and $HS2$? In such a case the collector may identify a single log entry (the one that was created before responding to $HS1$—the latter is not looked-at at all by the collector) as cycle closing. However, the log entry created between $HS1$ and $HS2$ might as well "overtake" the prior assignment to `logPos` resulting in the collector identifying no cycle closing log entries for `h`.

This is the sequence of events we are describing:

1. `T1:  r1 = create a replica of the handles in 'h'.`
2. `T1:  write 'r1' into the update log buffer.`
3. `T2:  Respond to HS1.`
4. `T2:  r2 = create a replica of the handles in 'h'.`
5. `T2:  write 'r2' into the update buffer.`
6. `T1:  h->logPos = position in T1's buffer.`
7. `T2:  h->logPos = position in T2's buffer.`

8. `T1: h[ offset ] = val1.`
9. `T1: h[ offset ] = val2.`
10. `T1: Respond to HS1.`

The log entry created by `T1` is not cycle closing during the clearing phase and the log entry created by `T2`, as it's created after responding to $HS1$, is not considered at all during the clearing phase of the current cycle.

While it looks as if we have a bug here it actually turns out that we may ignore the update of `h` by `T1` altogether because the object has been changed only after the current cycle has commenced. i.e., only after some thread has already responded to $HS1$ (in our case, `T2` responded to $HS1$ before the modification). According to the sliding view algorithm, we have to account for all changes occurring between the beginning of the previous cycle and the beginning of the current cycle. The update of `h` does not qualify.

The reinforcement phase (refer to section5.5) becomes simpler as well using this method. All we have to do is go over the entries in the clearing conflict set and for each log entry `le` check the object, `h`, it refers to. If `h->logPos` is null, then reinforce the log entry by re-closing the cycle. i.e., perform `h->logPos := le`.

The consolidation phase becomes unnecessary because no two log entries which are logged before $HS4$ can be cycle closing by the time the next cycle will commence.

To conclude, the "cycle closing?" predicate is a powerful tool that arbitrates automatically between log entries that refer to the same object. This mechanism solves all cases of multiple occurrences of log entries by itself with no need for extra data-structures and/or procedures for conflict resolution.

8.6 The Create Procedure

Recall that in the original algorithm an object is created "clean", i.e., with it's dirty flag turned off, and it is logged in a special create buffer which is treated as a thread local ZCT.

If we were to take the same approach in our implementation then immediately after creating the object, as "clean", we would have to dirty it because of initialization code to its fields.

We had not the development resources to explore the initialization approach suggested in section7.1.2 so we implemented a simpler yet efficient method. In our implementation, we create objects as "dirty". According to the principles of the sliding view algorithms we have to supply the collector with a log entry containing the contents of the object when we dirtied it. But the contents of the object at initialization are void.

Therefore, the mutator puts a reference to the object in a special "create buffer" and makes `logPos` point to the log entry. The collector knows that entries in create buffers signify objects which were logged when created, i.e., with empty contents and treats them accordingly:

1. the dirty flag is cleared
2. the current contents of the object are determined and the corresponding reference counters are incremented
3. no reference counters are decremented
4. the object is considered a candidate for deletion (i.e., it's in the ZCT).

Below we outline procedure `createObject`

```
Handle* createObject(int size)
{
   Handle* h = allocate(size);
   LogEntry *le = logNewObject( h );
   h->logPos = le;
}
```

8.7 Implementation of the log buffers

Principally, there are two kinds of log buffers in the sliding view algorithms:

1. **sets of object references**—these log buffers are "flat"; they contain one type of data: pointers to objects. This variety includes the create buffers and the snoop buffers. We have implemented the global ZCT (the one that lives between cycles) using this data structure as well.

2. **update buffers**—the update buffers are a collection of records of the form $< object, replica >$ where *replica* is a set of object pointers that where observed to be contained in *object*.

The two types of buffers are implemented as a doubly linked list of memory blocks. The size of a memory block is tunable but we have usually opted to use a block size of 64KB.

We assume objects are aligned on an 8-byte boundary thus we can utilize the lower 3 bits of a logged reference for auxiliary information. The bits are used to mark log entries with the following tags:

- **BUFF_NOMARK**(=0)—flat object reference.
- **BUFF_LINK_MARK**(=2)—the rest of the word is a pointer to the next or previous log block, depending on the direction of traversal.
- **BUFF_LOGGED_HANDLE_MARK**(=1)—this kind of entry appears only in update buffers. It signifies that the pointer is to an object which has been logged. The contents of the object are logged just before it in the buffer (as flat references). The word preceding the replica in the buffer can either be another entry marked with the **BUFF_LOGGED_HANDLE_MARK** tag or an entry with the **BUFF_LINK_MARK** tag.
- **BUFF_DUPLICATE_HANDLE_MARK**(=3)—this kind of entry appears only in update buffers. It is created by the collector by ORing a value of 2 into a slot marked with the **BUFF_LOGGED_HANDLE_MARK** tag. This action *invalidates* the entry (and the contents of the object appearing just before it in the buffer). Recall that the collector invalidates log entries on two occasions:

  1. during deletion, so that the entry would be skipped in the next cycle.
  2. during clearing, if the entry is not "cycle closing", in order to eliminate duplicates in the collection of mutators' update buffers.

A log buffer is controlled by a *log buffer header* that contains the following information:

- **start address**—address of the first block in the log.

- **limit address**—address of last slot in the current block, minus some elbow room. This limit address is compared against in logging, to check if there is enough space for the logging operation.

- **current position**—address of the next slot to be written into in the current block.

Recall that in the update protocol we have to prepare a replica of the object before committing it to the buffer. In our implementation, the replica is written directly into the buffer. Committing the replica is done simply by writing the address of the updated object ORed with **BUFF_LOGGED_HANDLE_MARK** into the buffer (after the replica of the object's contents) and updating the current position pointer in the log buffer header.

8.8 Cooperation model

We implemented the approach appearing in the original algorithms for cooperation. i.e., threads are suspended one at a time, the collector takes some action on their behalf and then the thread is resumed. We use a per-thread flag called `cantCooperate` which is turned on in sections of code during which the thread can not cooperate (i.e., during the write barrier, snooping of writes to global pointers and the logging of newly created objects).

In order to carry out a handshake the collector suspends the threads one at a time. If a thread is caught in non-cooperative code then the collector resumes it immediately and proceeds to handle other threads. The collector repeats this process until all threads have cooperated.

We were careful to limit the size of the non-cooperative code sections to a fixed and small number of instructions. This entailed reserving space in advance, in the snoop, create and update buffers prior to entering a non-cooperative section.

8.9 The memory manager

In the design of the memory manager we tried to satisfy these requirements:

1. allocation should be as fast as possible and should avoid synchronization bottlenecks. i.e., the allocator should be *scalable*.

2. both the tracing and reference counting asynchronous algorithms do not accommodate the relocation of objects in memory. The allocator should not suffer from fragmentation (except maybe for some pathological cases) due to this property.

3. in the asynchronous reference counting algorithm, reclamation of objects occurs sporadically rather than linearly as in the sweep phase of the tracing algorithm. The memory manager should handle efficiently this sporadic reclamation of objects. Even though objects will not be freed linearly it should still try to minimize fragmentation and increase the locality of allocation requests. i.e., it is preferable that two objects which are created in a row will be located closely in memory rather than chosen randomly from the entire heap space.

4. the vast majority of objects which are created are smaller than 60 bytes. The memory manager should take advantage of this fact by optimizing the allocation of small object. Allocation of medium sized and large objects may be less efficient than that of their smaller counterparts.

We found the original allocator inadequate to the requirements for the following reasons:

1. it allocates memory by assigning big chunks of memory to threads which later cut them into smaller pieces by incrementing a pointer. Naturally, the allocated memory spaces contain a mixture of object sizes. Since the original collector supports compaction, there is no fragmentation problem imposed by this allocation method. However, using this allocation method with a non-compacting collector would very quickly lead to irrecoverable fragmentation.

2. the original allocator is synchronized on a single lock. This of course hinders scalability.

3. the original allocator maintains two disjoin pools for handles and for objects. Yet we want to allocate a handle contiguously with its object.

4. the original allocator maintains information needed for compaction (e.g., object pinning information) which is useless for our algorithms.

Due to these reasons we decided to implement a custom allocator. Our allocator is divided into two levels of management: the chunk manager and the block manager. We now outline the roles of these managers.

The block manager manages big, equally sized, blocks of memory. The block size is tunable at compile time and we elected to equate it with the hardware page size, which is 4KB. It supports the following operations:

- allocate a range of blocks.
- free a range of blocks given the start address of the range.
- free a collection of ranges of blocks.

The block manager is totally serial and it is implemented using linked lists of equally sized regions of blocks. The block manager is utilized either directly, by the allocation code, or indirectly, using the chunk manager. When a user requests an allocation bigger than half a size of a block then the number of necessary blocks is allocated directly from the block manager. Smaller allocations are satisfied by the chunk manager which chunks single blocks into equally sized chunks that are consumed by the user.

The chunk manager is highly concurrent and efficient since it uses very fine locking, thread local allocation and it does not support coalescing or splitting: once a block is chunked into a specific size, all allocations from within it will use the same chunk size until (and if) the block is completely freed, in which case it will be returned to the block manager. Hence, allocation code need not perform costly checks due to variable sized chunks located on the same block. There is a fixed number of allocation sizes (approximately 20). The allocation sizes are chosen to balance between internal fragmentation (which calls for many different allocation sizes) and external fragmentation (which calls for a small number of allocation sizes so that blocks of one size can be used by objects of differing sizes instead of allocating separate pages for each object size).

A typical object oriented application will issue many allocation calls that will be implemented solely by the chunk manager and only relatively few calls will require allocating entire blocks from the block manager.

Let us now review in greater detail the implementation of the block and chunk managers.

8.9.1 The block table

The block table as an array of block table entries each describing a block in the heap.

Each entry is four words (16 bytes) wide and its format changes according to the current *state* of the block. However, all formats share one field: the state field. This field assumes one of the following states at each given moment:

- **BLOCK**—this state denotes blocks which are currently under the control of the block manager and are the first or last blocks in a contiguous free region.
- **BLOCKLIST**—same as **BLOCK** but the block is also the head of a linked list of regions, all of which are of the same size as this list-header region.
- **BLOCKINTERNAL**—denotes a block which is currently under the control of the block manager but is not the first or last block of a region. i.e., an *internal free block*. This state exists only for facilitating debugging. It does not exist in a non-debug build.
- **CHUNKING**—denotes a single block which has been allocated by the chunk manager, from the block manager, and it is currently being chunked into small pieces. This state serves two purposes. For the manager, it signifies that the block is no longer under its control. For the sweep phase of the collector it signifies that no objects should be collected from this block.
- **OWNED**—means that the block is currently *owned* by a mutator thread allocating off this block.
- **VOID**—signifies that the block is no longer owned by any mutator **and** that the collector has not yet recycled any chunks from this block.
- **PARTIAL**—signifies that the collector has recycled some chunks from this block. The block in that case is linked in a *partial list* and no mutator can allocate off it.
- **ALLOCBIG**—marks that the page is the first or last page in a big object (at least one block wide) that was allocated by the mutator and has not yet been reclaimed.
- **ALLOCINTERNAL**—same as **ALLOCBIG** only that the block is not the first or last block in the object. This state is used only for debugging purposes[1].

The block manager applies to following state transitions[2]:

- **BLOCK to ALLOCBIG**. Happens when a user requests the allocation of a big object.
- **BLOCK to CHUNKING**. Occurs when the chunk manager requests the allocation of a block for chunking it into small objects.
- **VOID or PARTIAL or ALLOCBIG to BLOCK**. Occurs whenever a chunked block is fully freed or when a big object is reclaimed.

[1] Actually, this state might be required for systems that do not maintain the invariant that a base pointer to a live object should always be present *somewhere*. i.e., systems that allow for pointers into the middle of objects with no corresponding pointers to the base of the object. On such a system this state can be used to locate the base pointer given a pointer to the middle of the object.

[2] in the following we treat BLOCK and BLOCKLIST interchangeably since they are equivalent logically from an external point of view.

The user allocation code applies the following transformation to block states:

- **CHUNKING to OWNED**—when the block is completely chunked it becomes owned by the thread on behalf of which it was allocated.
- **OWNED to VOID**—occurs when allocation can no longer proceed from the page, since there are no free chunks left on it.
- **PARTIAL to OWNED**—after some chunks have been recycled and the block is in the partial state a thread can gain ownership over it and start allocating off it.

The last set of transitions is applied by the collector (when executing chunk manager code that performs them):

- **VOID to PARTIAL**—occurs when some, but not all, objects on a VOID-marked block are recycled. At the same time the block is linked into a *partial list* (see below).
- **PARTIAL to BLOCK**—occurs when all objects on a partial page are recycled. This transition occurs atomically with the removal of the block from the partial list on which it resides and handing it back to the block manager.
- **OWNED to OWNED**—happens when some objects on an owned page are recycled. Thread ownership is not revoked but rather the newly recycled objects become available for the thread to use.

8.9.2 Partial lists

Partial blocks are linked on partial lists. There is a partial list for each possible chunk size.

As mentioned, when a page is transitioned from the VOID state to the PARTIAL state it is inserted into the partial list corresponding to the size of the chunks the block hosts.

Conversely, when a mutator needs to allocate a chunk of a certain size, and it does not own a block hosting chunks of the required size, then it may take a block off the corresponding partial list and become the owner of it.

Finally, when the collector finds that a partial block is fully vacant then it may evacuate the block from the list and return it to the block manager.

8.9.3 Chunked object lists

Whenever a block is in one of the *chunked states*. i.e., OWNED, VOID and PARTIAL, there are three lists of objects associated with it:

- **Allocation list**—this list contains objects which can be allocated directly by the owning thread. The list contains elements only when the page is owned and its header is cached by the owner thread. This list is accessed solely by the owner thread hence there is no contention incurred for using it and allocation becomes as simple as popping an element off a linked list.
- **Recycled list**—this lists contains elements which have been recently recycled by the collector. It is accessible only to the collector. The collector maintains a hash table of recycled lists headers thus occasionally it needs to flush a recycled list associated with a block. When it does so, the list is merged with the block's *free list* (see below).

- **Free list**—is the list used to transfer objects from the recycled list to the allocation list. As said, the collector sporadically flushes the recycled list into the free list. The owner mutator, when it sees that the allocation list is empty, tries to move all current elements from the free list to the allocation list. If it fails doing so (i.e., the free list is empty) it transforms the block state into VOID.

Synchronization is only needed for accessing the free list and is achieved by a lightweight lock implemented at the block level. The lock is imbedded in the block table entry and is implemented using low-level atomic operations (specifically, compare-and-swap).

Objects on the chunked object lists are linked through the `logPos` field. This helps maintain the invariant that an object is eligible for reclamation by the mark-and-sweep algorithm only if it has a null pointer in this field. Essentially, this trick solves the infamous race condition between allocation and the sweep phase which otherwise requires object coloring schemes.

Chapter 9

Performance Results

In this chapterwe assess our algorithms performance characteristics compared to the original algorithm used in the JVM and comparing the tracing collector to the reference counting collector.

9.1 The benchmarks used—instrumentation results

We used two standard testing suites: SPECjbb2000 and JPECjvm98. These benchmarks are described in detail in SPEC's Web site[2].

Our primary instrumentation goal was to study the memory consumption behavior of these benchmarks. To that end, we have compiled the JVM with the GC and allocator modules in instrumented mode and the rest of the JVM in production mode. That way, the runs were realistic ones, with the amount of objects allocated and running times not significantly different from an all-production JVM yet still we gained the GC instrumentation information.

In order to appreciate the "sensitivity" of each benchmark to reference counting, i.e., the amount of garbage cycles and stuck reference counters that the benchmark produces, we ran each benchmark only with the tracing collector and also only with the reference counting collector, without the use of the auxiliary tracing collector. Figure 9.1 shows the number of objects allocated, average object size and the average number of references in an object. Overall, the number of allocated objects when using the reference counting collector is comparable to the number of allocated objects using the tracing collector, though almost always smaller by a maximal factor of 5%. This is consistent with the performance figures we present later.

All tests were conducted with an equal setting for the two collectors: a four way Pentium III at 550Mhz with 2GB of physical memory and a 600MB java heap for the JBB server benchmark and a single Pentium III at 500Mhz with 256MB of physical memory and 64MB for the jvm98 client benchmarks. However, the reference counted runs of the compress and javac benchmarks were not able to complete with 64MB heap and therefore the instrumentation results presented here refer to runs of these two benchmarks with a java heap of 200MB.

As figure 9.1 shows, the small number of references per object (e.g., a reference or two in a typical object) supports our premises that the number of references in most objects is relatively small hence the use of a flag per object instead of a flag per slot does not involve a significant amount of extra logging.

Figure 9.2 shows the number of objects that have reached a stuck count (i.e., $o.RC = 3$) in the reference counted runs and the relative percentage of these objects in the entire object population. These numbers support our assumption that a two-bit reference count is enough for the striking majority of objects.

Benchmark	Tracing			RC		
	No. allocated objects	Object size	No. References	No. allocated objects	Object size	No. References
jbb	26,753,615	49.9	1.6	25,113,179	52.4	1.7
compress	55,126	2,041.1	0.8	58,061	1,940.4	0.9
db	3,261,467	34.0	2.6	3,263,358	34.0	2.6
jack	6,919,637	40.3	1.7	6,917,102	40.3	1.6
javac	6,403,821	42.9	1.9	6,405,478	43.0	1.9
jess	7,994,215	46.4	3.6	7,993,946	46.4	3.6
mpegaudio	65,539	31.6	1.1	58,329	29.5	0.9

Table 9.1: Number of allocated objects, average object size and the average number of references in an object.

Benchmark	Stuck objects	Relative percentage
jbb	141,141	0.6%
compress	2,727	4.7%
db	30,637	0.9%
jack	51,607	0.7%
javac	235,605	3.7%
jess	12,566	0.2%
mpegaudio	2,728	4.7%

Table 9.2: Number of objects that have reached a stuck count (i.e., 3) and their percentage in the reference counted runs.

Benchmark	% Reclaimed by tracing	% Reclaimed by RC	RC Inefficiency
jbb	97.5%	96.5%	1.2%
compress	73.5%	72.1%	2.1%
db	99.6%	90.5%	9.1%
jack	99.6%	96.8%	2.8%
javac	99.6%	66.1%	33.6%
jess	99.8%	99.5%	0.3%
mpegaudio	74.2%	69.6%	6.2%

Table 9.3: Percentage of objects reclaimed by the tracing and reference counting collectors and the associated estimate for reference counting inefficiency in collecting the benchmark.

Benchmark	No. stores to new objects	No. stores to old objects	No. object log actions	No. references logged	Create vs. log ratio
jbb	61,070,693	9,940,664	52,410	264,115	0.00209
compress	63,892	1,013	13	51	0.00022
db	31,297,167	1,827,613	36	30,696	0.00001
jack	135,013,882	160,893	824	1,546	0.00012
javac	21,774,697	267,331	189,395	535,296	0.02946
jess	26,206,218	51,889	544	27,333	0.00007
mpegaudio	5,517,487	308	12	51	0.00021

Table 9.4: Demographics of the write barrier: number of reference stores applied to new and old objects; number of object logging actions; total number of references that were logged and the ratio of the number of object logging actions to the number of allocations. This ratio is an upper bound to the percentage of objects which ever get logged in the write barrier.

In an attempt to measure the sensitivity of each benchmark to reference counting we compared the ratio of collected to allocated objects between the tracing and reference counting collectors. For example, if in a tracing run 90% of the objects were reclaimed and in the corresponding RC run only 81% of the objects were reclaimed then the amount of sensitivity, or inherent inefficiency, of reference counting for this benchmark is estimated to be 10%. The results are summarized is figure 9.3. Except for javac, which uses many cyclic structures, and to a lesser degree the db benchmark, the benchmarks have demonstrated a low degree of sensitivity to reference counting. This supports the assumption that we may use reference counting for most garbage collection cycles and only occasionally resort to tracing.

We now turn our attention to the use of the write barrier. Figure 9.4 shows the number of reference stores that have been applied to "new" vs. "old" objects (i.e., objects that still haven't undergone a collection cycle versus those which have survived at least one collection cycle), the number of object logging actions, and the ratio of logging actions to object creation actions (this is an upper bound for the percentage of objects which ever get logged). (The figures are for the reference counted runs; very similar results were obtained for the tracing runs.) We learn from these figures the following:

- Most reference stores are applied to new objects, probably because there are more of them compared to old objects and because new objects have to be initialized.
- From the reference stores which are applied to old objects only a fraction leads to logging. This means that the same old objects are accessed repeatedly. Yet we have to log the object only the first time it is accessed in a cycle.

To conclude, due to this essentially generational behavior it is indeed beneficial to mark new objects as dirty. Also, the price paid for the write barrier almost always equals the price of a memory load and register test. Due to the large amount of new objects versus old, logged, objects, the complexity of a reference counted cycle is in reality proportional to the number of objects that were allocated during the cycle. It does appear, though, that those old objects which are repeatedly changed contain much more references compared to the average. See for example the ratio between the number of logged references to the number of logged objects in the jbb, db and jess benchmarks which far exceeds the average number of references per objects in these benchmarks. This suggests

	jbb	compress	db	jack	javac	jess	mpegaudio
No. cycles	7	4	4	7	10	10	2
GC time	18.2	0.3	1.6	3.3	5.0	4.8	0.1
Clear	31%	3%	26%	33%	21%	26%	20%
Trace	15%	3%	19%	3%	28%	6%	20%
Sweep	50%	7%	47%	55%	45%	54%	20%

Table 9.5: GC time for the tracing collector, in seconds and the time spent in clearing dirty marking, tracing and sweeping.

	jbb	compress	db	jack	javac	jess	mpegaudio
No. cycles	7	2	4	9	6	10	2
GC time	46.1	0.1	5.7	10.3	9.0	17.0	0.1
Clear	12%	7%	7%	10%	11%	7%	0%
Update	36%	19%	37%	31%	43%	37%	19%
Create buff	8%	7%	13%	13%	11%	8%	7%
Reclaim	42%	21%	41%	41%	33%	43%	21%

Table 9.6: GC time for the reference counting collector, in seconds. "Clear" refers to procedure **Clear_Dirty_Marks**; "Update" refers to **Update_Reference_Counters**; "Create buff" refers to the pass over the create buffers, checking whether an object is garbage and adding it to the ZCT; "Reclaim" is the final pass over the ZCT, when objects are deleted recursively.

that we might need to explore ways to log large objects "by pieces" and not in their entirety, as is currently done.

Finally, let us look at the execution times of each of the collectors. Figure 9.5 shows the number of collection cycles, total elapsed time of the collection cycles and how this time distributes between the major stages of a tracing garbage collection. Figure 9.6 presents these data for the reference counting collector.

Looking at the time distribution for the tracing collector, we see that sweeping takes more time than tracing. This is despite the fact that we sweep using the block table and reference counters bitmap, without looking at the object unless it is actually freed. This implies that the sweeping code has yet to be optimized.

For benchmarks that deal with smaller amounts of larger objects, such as compress, we see that most GC time is spent in garbage collection overheads (handshakes, etc.)

We note that the reference counter used as many garbage collection cycles as the tracing collector (except for the case of compress and javac where the reference counter was allotted a bigger heap). Also the elapsed time for the reference counting garbage collector is longer. In an attempt to find the culprits for this situation we observe that the duration of both the update of reference counters and the reclamation of dead objects is in effect browsing almost all of the objects that were allocated during the cycle, on a one-by-one basis. Since the heap occupancy in all of these benchmarks is quite low for the given heap sizes, a lot of objects are created during the cycle, resulting in a better performance for the tracing collector. We will elaborate on this point in section9.5.

We now turn to investigate the collectors' performance results compared to the original JVM. We start with server performance and then continue with client performance.

Heap size (MB)	Score	% Improv. in score (throughput)	Maximal response time (sec)	% Imrov. in response time
600	642.7	-2.6%	0.12	98.5%
900	641.0	-2.3%	0.11	99.1%
1200	633.3	0.7%	0.11	99.2%

Table 9.7: Reference counting performance in a standard SPECjbb run.

Heap size (MB)	Elapsed GC time	% increase. in GC time	No. sync cycles	No. RC cycles	No. tracing cycles
600	147	227%	2	11.7	1.0
900	144	269%	2	6.3	0.0
1200	143	225%	2	5.0	0.0

Table 9.8: Elapsed time of garbage collection in a standard SPECjbb run with the reference counting collector; the percentage of increase in elapsed time over the original garbage collector and the types of garbage collection cycles that were performed. "sync" is a synchronous GC cycle requested explicitly by the benchmark.

9.2 Server performance

A standard execution of SPECjbb requires a multi-phased run with increasing number of threads. Each phase lasts for two minutes with a ramp-up period of half a minute before each phase. Prior to the beginning of each phase a synchronous GC cycle may or may not occur, at the discretion of the tester. We decided not to perform this synchronous garbage collection as we believe it defeats capturing real world scenarios in which the server is not given a change for this "offline" behavior so often. The results presented here are averaged over three standard runs.

Figure 9.7 shows the two most important performance meters for the reference counting collector compared to the original JVM: while we do pay a small price of up to 2.6% decreased throughput, we improve the maximal response time by two orders of magnitude. To illustrate, the original JVM may pause for as long as 16 seconds while we never cause a mutator to pause for more than 130 milliseconds. This problem of the original JVM becomes aggravated as the heap grows in size. As can be seen from figure 9.8, the reason for the performance penalty is the prolonged elapsed time of garbage collection, compared to the original JVM. This implies that by further optimizing the collector code we may obtain better scores than the original JVM while maintaining the very short response time.

This is exactly the case with the tracing collector, which outperforms the original JVM in both

Heap size (MB)	Score	% improv. in score	Maximal response time (sec)	% Imrov. in response time
600	1124.0	-0.6%	0.14	98.3%
900	1129.3	1.3%	0.12	99.0%
1200	1146.3	4.1%	0.13	99.2%

Table 9.9: Tracing collector performance in the standard SPECjbb run.

Heap size (MB)	Elapsed GC time	% increase. in GC time over original	No. sync cycles	No. tracing cycles
600	51	13%	2	11.0
900	51	30%	2	7.3
1200	47	7%	2	5.0

Table 9.10: Elapsed time of garbage collection in a standard SPECjbb run with the tracing collector; the percentage of increase in elapsed time over the original garbage collector and the types of garbage collection cycles that were performed. "sync" is a synchronous GC cycle requested explicitly by the benchmark.

Threads	1	2	4	6	8	10	15	20
Original	637	1125	1728	963	928	903	887	847
RC	0.4%	4.0%	-5.4%	-2.0%	-1.0%	-2.2%	-0.3%	2.4%
Tracing	1.0%	4.6%	2.3%	-1.3%	0.2%	-0.9%	0.1%	2.8%

Table 9.11: Scores of the original JVM on a series of fixed number of threads runs with 600MB heap; increase/decrease in score for the reference counting and tracing collectors.

throughput and maximal response time. See figure 9.9. Figure 9.10 shows that the elapsed time of garbage collection for this algorithm is much closer to the elapsed running time of the original collector, resulting in improved performance.

Next we seek to check how our collectors perform relative to the original collector as a function of the number of threads and heap size. We have performed a series of stand-alone SPECjbb runs with 1, 2, 4, 6, 10, 15 and 20 threads; 600MB, 900MB and 1200MB heaps; the original, reference counting and tracing collector. The results are summarized in figures 9.11 through 9.16. From throughput perspective, our collectors have compatible performance with that of the original collector with the tracing collector performing better than the reference counting collector. We do see a slip in performance in the range of 4 to 10 threads and this effect worsens as the heap grows. This is probably related to two factors: inefficient reclamation, which worsens as the heap grows, and tuning of spin locks for these numbers of threads. Examining the maximal response time we again see a remarkable behavior of our collectors where the original collector consumes longer and longer pause times as the heap grows. Figure 9.16 might seem an exception to this rule at first glance but actually what happens is that since garbage collections with such a large heap are scarce (one or two in a run) they actually might occur when the benchmark is not measuring response time hence the original JVM manages "to get away" with its long pause times unnoticed on most cases. However, examining the pause time for 4 and 20 threads we see that these pauses nonetheless occur.

We now examine our memory consumption behavior. Given that we have added an extra pointer

Threads	1	2	4	6	8	10	15	20
Original	645	1137	1742	978	947	918	858	893
RC	-1.3%	3.2%	-3.8%	-3.3%	-3.3%	-3.3%	3.2%	-4.0%
Tracing	-1.6%	2.6%	2.1%	-2.6%	-2.4%	-2.1%	4.0%	-2.2%

Table 9.12: Scores of the original JVM on a series of fixed number of threads runs with 900MB heap; increase/decrease in score for the reference counting and tracing collectors.

Threads	1	2	4	6	8	10	15	20
Original	629	1155	1683	935	908	884	882	870
RC	-2.5%	1.7%	-7.1%	-8.0%	-7.8%	-6.8%	0.2%	-0.8%
Tracing	-6.3%	3.6%	-2.7%	-6.9%	-6.5%	-5.4%	1.1%	-0.2%

Table 9.13: Scores of the original JVM on a series of fixed number of threads runs with 1200MB heap; increase/decrease in score for the reference counting and tracing collectors.

Threads	1	2	4	6	8	10	15	20
Original	7.43	8.04	8.47	6.92	7.86	7.54	6.59	6.00
RC	0.02	0.02	0.05	0.08	0.11	0.15	0.25	0.33
Tracing	0.02	0.02	0.06	0.09	0.13	0.18	0.25	0.35

Table 9.14: Maximal response time, in seconds, of the original JVM, reference counting and tracing collectors in a series of fixed number of threads runs with 600MB heap.

Threads	1	2	4	6	8	10	15	20
Original	0.02	11.17	12.07	10.70	10.53	10.30	9.82	9.23
RC	0.02	0.02	0.05	0.08	0.11	0.14	0.23	0.34
Tracing	0.02	0.02	0.05	0.08	0.12	0.15	0.25	0.33

Table 9.15: Maximal response time, in seconds, of the original JVM, reference counting and tracing collectors in a series of fixed number of threads runs with 900MB heap.

Threads	1	2	4	6	8	10	15	20
Original	0.02	0.02	14.67	0.05	0.08	0.01	0.18	13.03
RC	0.02	0.02	0.05	0.07	0.11	0.15	0.22	0.32
Tracing	0.02	0.02	0.06	0.09	0.12	0.15	0.25	0.33

Table 9.16: Maximal response time, in seconds, of the original JVM, reference counting and tracing collectors in a series of fixed number of threads runs with 1200MB heap.

Threads	1	2	4	6	8	10	15	20
Original	24	39	70	100	139	160	236	312
RC	27	44	77	108	170	171	251	329
Tracing	27	44	77	108	130	171	251	330

Table 9.17: Memory consumption at the end of a series of fixed number of threads runs with 600MB heap.

Threads	1	2	3	4	8	12	16
Original	93.0	71.9	56.3	57.2	58.2	58.0	59.0
RC	88.6	68.5	52.5	54.2	52.3	57.9	59.1
Tracing	89.1	68.9	52.3	55.8	50.8	53.4	53.5

Table 9.18: Time to completion, in seconds, of the MTRT benchmark, with varying number of threads.

	Heap (MB)	Time (sec)
Original	25	120
RC	20	85
Tracing	20	86

Table 9.19: Minimal heap size required to complete successfully a four thread mtrt run and the time to completion with that heap size.

to each object (the log pointer) we would expect to see some increase in the memory consumption, relative to the average object size in each benchmark. Furthermore, since we do not compact the heap we are more vulnerable to internal fragmentation compared to the original JVM. When our collector is asked to report the amount of free memory it sums up (non-atomically) the amount of storage available in the block manager and in partial blocks. It ignores owned blocks so actually the amount of free memory is larger than reported. Given this metric, the results of used memory as reported by SPECjbb (for the 600MB test series) are summarized in figure 9.17. Note that except for an unexplained (yet reproducible) bump in the memory consumption for 8 threads with the reference counting collector[1] we consume no more than 8% more memory compared to the original JVM. This can be further improved once we eliminate completely the handle-to-object pointer in each object, which is not required by our collectors.

The second benchmark that we have used is MTRT (multi-threaded ray tracer), a member of SPECjvm98 which can be used with a varying number of threads. We have ran this benchmark with the default heap size—64MB. This benchmark does not measure response time, only elapsed running time, which corresponds to the JVM's throughput. As can be seen from figure 9.18 both on-the-fly collectors have outperformed the original JVM with an improvement of up to 12.6% in the total running time.

The ordinary measure of heap consumption—probing the free space left at the run does not capture transient effects and the ability to handle stressful situations. Figure 9.19 shows the minimal heap size (in 1MB granularity) required to complete the mtrt benchmark successfully and the corresponding time to completion. The concurrent collectors require about 20% less the memory to complete successfully and arrive at completion at about 70% the time. This is clearly a defect of the original JVM as it should actually require no more memory than our collectors and since in this stressful situation we resort to synchronous GC there should be no gain from concurrent collection as well.

[1]This bump cannot be explained by reference counting issues since the amount of consumed memory is calculated only after the benchmark requests a synchronous garbage collection cycle, which is always implemented by our collectors using a tracing cycle.

Benchmark	Original	RC	Tracing
Total	2582.2	2676.0	2610.9
compress	720.8	723.3	718.4
db	374.0	383.7	374.0
jack	264.6	299.7	285.0
javac	225.0	235.2	233.7
jess	181.7	209.7	182.1
mpegaudio	607.1	610.6	611.1

Table 9.20: Elapsed time for the execution of the entire SPECjvm98 suite and intermediate execution time of a double-run for each of the suite's members.

9.3 Client performance

While we have targeted our collectors for multi-processor environments we still wanted to verify that they are competent in a single-processor setting. To that end we have used the SPECjvm98 benchmark suite. We used the suite using the test harness, performing standard[2] automated runs of all the benchmarks in the suite. In a standard automated run, each benchmark is ran twice and all benchmarks are ran on the same JVM one after the other. Figure 9.20 shows the elapsed time of the entire automated run and the time for each double run of each benchmark. We see that the tracing collector was only 1.1% percent slower than the original JVM and the reference counting collector only 3.6% slower. Given that we pay the overheads of concurrent run while we're not benefiting from the availability of multiple processors these are remarkably good results.

9.4 Allocator scalability

We have designed the custom allocator with scalability in mind. In order to check whether the collector indeed meets design goals we have written a small allocation benchmark that tries to measure allocation overhead in isolation from garbage collection overheads. The program works in phases. Each phase, N threads are started and each of them allocates 1,000,000 arrays of references, each with a random number of slots, chosen uniformly from the range $\{1, ..., 5\}$. Each thread links 10% (chosen randomly) of the objects it allocates into a linked list.

On the end of each phase the elapsed time of the phase is determined; the linked list of objects from the cycle prior to the one just ended is discarded (the list from the current cycle is held "alive" for the next cycle) and finally synchronous garbage collection is invoked. It is verified externally that the heap is big enough so that no garbage collector ever occurs during a phase run. The entire test is comprised of four phases with the results being the average of the last three runs.

As described, during a measured phase, there is little happening in the system besides concurrent allocation. Furthermore, since the heap is dotted with allocated objects from the previous phase, allocation cannot be just a matter of bumping a pointer. This behavior mimics real world scenarios were the heap contains differently aged objects. We do give a chance, however, to the collector, to do compaction between phases.

The results for the four-way server with 1200MB heap are presented in figure 9.21 as the throughput of the JVM (objects created per second) relatively to the number of working threads. The custom allocator achieves excellent scalability where there is almost no loss in performance

[2]The standard run requires running the harness through a Web server while we performed the tests directly off the disk. Aside from that, the executions were standard.

# Threads	1	2	4	8	12	16
Original allocator	1143	1768	1747	1570	1537	1525
Custom allocator	917	1638	2349	2356	2371	2373

Table 9.21: Objects created per second in the allocation benchmark on a four-way server.

# Threads	1	2	4	8	12	16
Original allocator	1044	1054	823	873	911	796
Custom allocator	844	843	838	843	852	818

Table 9.22: Objects created per second in the allocation benchmark on a single processor workstation.

when going from 4 threads to 16 threads. The original allocator, however, performs less well, loosing 12% of its throughput over the same range. With 16 worker threads, the custom allocator achieves a throughput higher by 56% than the one achieved by the original allocator.

Refer to figure 9.22 for the comparison of the allocators on a client machine. For this test we used 300MB heaps. This necessitated changing the number of objects allocated by each thread, on each phase, from 1,000,000 to 100,000 with 8 threads or more. On the single processor client machine the situation is less decisive compared to the server environment. Here, the custom allocator performs less well with 1 or 2 threads by a factor of 20%. With 4 threads and more, however, it is as marginally better than the original allocator.

9.5 Discussion

As can be seen from the throughput results for our on-the-fly reference counting collector versus the results of our on-the-fly tracing collector, the tracing collector exhibits better results in most cases. In this section, we try to explain this result. We characterize the differences between the two algorithms and thus we find the conditions under which each may perform better than the other. In our discussion, we abstract away several factors which may still be important in some environments, for example:

- we assume that it is harmless to use the entire heap space which is allocated for the program since this heap is backed by fast RAM. This holds for servers, but not for memory constrained systems.
- we assume that fragmentation is not an issue. Again, this is tied to the previous point. If we have a big heap and the access of pattern to memory is uniform, as is the case with servers, then this assumption holds as well.
- we assume that the price of initiating a garbage collection is negligible relative to the price of the collection itself. This can be evidently seen from our instrumentation measurements (refer to section9.1).
- we assume that our target programs have steady states. While this may not be the case in reality, there is not much we can say about programs with irregular behavior.

Suppose we are given a heap of size H. Let us assume that the benchmark has a steady state at which it consumes a fraction γ of H. That is, most of the time, γH heap space is alive. Also, let R_A be the *allocation rate*, which equals the garbage production rate in steady state, in units

of memory volume per time, and let R_M be the *mutation rate* which is defined to be the rate at which the mutators mutate old data structures, i.e., data structures that have at least survived one garbage collection cycle.

As we noted before, the overhead of a reference counting garbage collection cycle is proportional to the amount of space allocated since the last cycle, the amount of space that became garbage since the last cycle and the amount of space that was mutated since the last cycle. Therefore, reference counting is less sensitive to the *triggering* used. i.e., two short reference counting cycles cost roughly the same as one longer, combined reference counting cycle. Thus, each implementation is characterized by two constants, c_1 and c_2 that characterize the overhead incurred by reference counting per unit of time. That overhead is:

$$Overhead_{RC} = c_1 R_A + c_2 R_M$$

Referring to the set of benchmarks we have used, we see that the actual old objects mutation rate, R_M, is very low thus in practice we may approximate the overhead with $c_1 R_A$.

The price of a tracing cycle, on the other hand, is fixed no matter when was the last cycle executed. The price of the cycle is proportional to the amount of live data (in the tracing phase) and to the entire size of the heap (in the sweep phase). So the price is $c_{tr}\gamma H + c_{sw} H$ where c_{tr} and c_{sw} are the proportion constants for the mark and sweep stages, respectively. In practice, tracing dominates the price of a tracing cycle, so the price can be approximated by $c_{tr}\gamma H$.

Assuming that it is beneficial to use the entire heap space, we want to delay a tracing cycle as much as possible. We can do that until we run out of heap space, which happen after $(1-\gamma)H/R_A$ units of time (the amount of free space divided by the allocation rate). Thus, the overhead of tracing garbage collection, per unit of time, is:

$$Overhead_{Tracing} = c_{tr} R_A \gamma/(1-\gamma)$$

Let us define the *load factor*, α, to be $\gamma/(1-\gamma)$. We now see that the tracing overhead is bigger than the reference counting overhead if, and only if:

$$\alpha > c_1/c_{tr}$$

This has two implications:

1. The reference counting algorithm reacts better than the tracing algorithm to a growth in the heap occupancy factor. Note that the load factor grows very quickly as the heap occupancy grows. Apparently, the set of benchmarks that were available to us do allocate a lot of objects but do not maintain a large volume of live data over time. It seems that this is not the case with true servers, such as Web servers, that utilize the entire heap allotted to them for caching web pages etc.

2. Unfortunately, the "implementation quality factor", c_1/c_{tr}, is not favorable for the reference counting algorithm. While c_{tr} depends only on the implementation of the tracing phase, c_1 describes the complexity involved in clearing the dirty bits of newly allocated objects, updating the reference counters for objects pointed by newly allocated objects, maintaining the ZCT, and, finally, recursively deleting dead objects. Thus, improving the overhead of the reference counting algorithm is a much harder task than improving the mark and sweep algorithm.

Chapter 10

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 [17], 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 we have presented an on-the-fly, 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 have implemented the proposed algorithms for Java, atop Sun's JDK for Microsoft's Windows NT Operating System. Our algorithms attain a dramatic improvement in response time over the original garbage collection algorithm. The reference counting algorithm achieves throughput comparable with that of the original JVM while the on-the-fly tracing collector outperforms the original JVM.

This work opens avenues for additional research on the following areas, among them: (1) studying the behavior of tracing vs. reference counting collector in high heap occupancy environments, as suggested in section9.5, (2) combining the ideas presented in [1] with the ideas presented in this work and (3) applying generational principals to the reference counting algorithm.

Bibliography

[1] David Bacon, Dick Attanasio, Han Lee, and Stephen Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector Manuscript Nov. 2001.

[2] Standard Performance Evaluation Corporation, http://www.spec.org

[3] Bill Joy, Guy Steele, James Gosling and Gilad Bracha. *The Java Language Specification, Second Edition (The Java Series)* Addison-Wesley, 2000.

[4] Andrew Patzer, Sing Li, Paul Houle, Mark Wilcox, Ron Phillips, Danny Ayers, Hans Bergsten, Jason Diamond, Mike Bogovich, Matthew Ferris, Marc Fleury, Ari Halberstadt, Piroz Mohseni, Krishna Vedati and Stefan Zeiger. *Professional Java Server Programming: with Servlets, JavaServer Pages (JSP), XML, Enterprise JavaBeans (EJB), JNDI, CORBA, Jini and Javaspaces* Wrox Press, 1999.

[5] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[6] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[7] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[8] Thomas H. Axford. Reference counting of cyclic graphs for functional programs. *Computer Journal*, 33(5):466–470, 1990.

[9] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[10] Henry G. Baker. Minimising reference count updating with deferred and anchored pointers for functional data structures. *ACM SIGPLAN Notices*, 29(9), September 1994.

[11] Daniel G. Bobrow. Managing re-entrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[12] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[13] T. Chikayama and Y. Kimura. Multiple reference management in Flat GHC. In *4th International Conference on Logic Programming*, pages 276–293, 1987.

[14] T. W. Christopher. Reference count garbage collection. *Software Practice and Experience*, 14(6):503–507, June 1984.

[15] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[16] Jim Crammond. A garbage collection algorithm for shared memory parallel processors. *International Journal Of Parallel Programming*, 17(6):497–522, 1988.

[17] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, DEC Systems Research Center, Palo Alto, CA, August 1990.

[18] John DeTreville. Experience with garbage collection for modula-2+ in the topaz environment. In Eric Jul and Niels-Christian Juul, editors, *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, October 1990.

[19] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[20] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[21] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, January 1994.

[22] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[23] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of High Performance Computing and Networking (SC'97)*, 1997.

[24] Daniel P. Friedman and David S. Wise. Reference counting can manage the circular environments of mutual recursion. *Information Processing Letters*, 8(1):41–45, January 1979.

[25] Shinichi Furusou, Satoshi Matsuoka, and Akinori Yonezawa. Parallel conservative garbage collection with fast allocation. In Paul R. Wilson and Barry Hayes, editors, *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[26] Edward F. Gehringer and Ellis Chang. Hardware-assisted memory management. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[27] Adele Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[28] Atsuhiro Goto, Y. Kimura, T. Nakagawa, and T. Chikayama. Lazy reference counting: An incremental garbage collection method for parallel inference machines. In *Proceedings of Fifth International Conference on Logic Programming*, pages 1241–1256, 1988. Also ICOT Technical Report TR-354, 1988.

[29] Robert H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[30] Maurice Herlihy and J. Eliot B Moss. Non-blocking garbage collection for multiprocessors. Technical Report CRL 90/9, DEC Cambridge Research Laboratory, 1990.

[31] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *OOPSLA'92 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 27(10) of *ACM SIGPLAN Notices*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press.

[32] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7) of *ACM SIGPLAN Notices*, pages 73–82, San Diego, CA, May 1993. ACM Press.

[33] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Novemeber 1998.

[34] Elliot K. Kolodner and Erez Petrank. Parallel copying garbage collection using delayed allocation. Technical Report 88.384, IBM Haifa Rsearch Lab, November 1999.

[35] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proceedings of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[36] Rafael D. Lins and Márcio A. Vasques. A comparative study of algorithms for cyclic reference counting. Technical Report 92, Computing Laboratory, The University of Kent at Canterbury, August 1991.

[37] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[38] James S. Miller and B. Epstein. Garbage collection in MultiScheme. In *US/Japan Workshop on Parallel Lisp, LNCS 441*, pages 138–160, June 1990.

[39] David A. Moon. Garbage collection in a large LISP system. In Steele [44], pages 235–245.

[40] James W. O'Toole and Scott M. Nettles. Concurrent replicating garbage collection. Technical Report MIT–LCS–TR–570 and CMU–CS–93–138, MIT and CMU, 1993. Also LFP94 and OOPSLA93 Workshop on Memory Management and Garbage Collection.

[41] Young G. Park and Benjamin Goldberg. Static analysis for optimising reference counting. *Information Processing Letters*, 55(4):229–234, August 1995.

[42] David J. Roth and David S. Wise. One-bit counts between unique and sticky. In Richard Jones, editor, *Proceedings of the First International Symposium on Memory Management*, volume 34(3) of *ACM SIGPLAN Notices*, pages 49–56, Vancouver, October 1998. ACM Press. ISMM is the successor to the IWMM series of workshops.

[43] Patrick Sobalvarro. A lifetime-based garbage collector for Lisp systems on general-purpose computers. Technical Report AITR-1417, MIT AI Lab, February 1988. Bachelor of Science thesis.

[44] Guy L. Steele, editor. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, Austin, TX, August 1984. ACM Press.

[45] Will R. Stoye, T. J. W. Clarke, and Arthur C. Norman. Some practical methods for rapid combinator reduction. In Steele [44], pages 159–166.

[46] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp architecture. In *Proceedings of the Thirteenth Symposium on Computer Architecture*, June 1986.

[47] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[48] Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly and Associates, Inc., 1991.

[49] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, September 1963.

[50] J. Weizenbaum. More on the reference counter method. *Communications of the ACM*, 7(1):38, 1964.

[51] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[52] David S. Wise. Stop and one-bit reference counting. *Information Processing Letters*, 46(5):243–249, July 1993.

[53] David S. Wise. Stop and one-bit reference counting. Technical Report 360, Indiana University, Computer Science Department, March 1993.

[54] David S. Wise, Brian Heck, Caleb Hess, Willie Hunt, and Eric Ost. Uniprocessor performance of a reference-counting hardware heap. Technical Report TR-401, Indiana University, Computer Science Department, May 1994.

[55] David S. Wise, Caleb Hess, Willie Hunt, and Eric Ost. Research demonstration of a hardware reference-counting heap. *Lisp and Symbolic Computation*, 10(2):151–181, July 1997.

[56] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.

[57] Benjamin Zorn. Barrier methods for garbage collection. Technical Report CU-CS-494-90, University of Colorado, Boulder, November 1990.

Appendix 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 absence of global roots. Instead, we take the burden of showing how to incorporate them into the algorithm: 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 $HS_{-1} \stackrel{\text{def}}{=} HS_0$. 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 Road map for 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.
- $LA.2_k$: 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.
- $LA.3_k$: 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.
- $LA.4_k$: The collector finds out, eventually, in procedure **Fix-Undetermined-Slots**, what are the values of undetermined slots.
- $LA.5_k$: 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.
- $LA.2_k \Longleftarrow \bigwedge_{j<k} SafetyTheorem_j$
- $LA.5_k \Longleftarrow \bigwedge_{j<k}(LA.5_j \wedge SafetyTheorem_j) \wedge LA.2_k \wedge LA.4_k$
- $SafetyTheorem_k \Longleftarrow LA.5_j$

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 sectionprove 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. $UPD(s) = \oslash \Longrightarrow ASSOC(s) = \oslash$

2. $UPS(s) \neq \oslash \Longrightarrow ASSOC(s) = \{s@t\}$

Proof. 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 HS_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 **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 τ 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 τ. Since there were 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 τ. Just before T_j executed the store in line (6) the value of $Dirty(s)$ must have been **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 **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

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

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

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

□

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:

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

For brevity we write $s \in H$ meaning $\exists v : \langle s, v\rangle \in H$

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 HS_m and HS_{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.
- If $m < k - 1$, 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 HS_{k-1} and HS_k then*

$$VAL(Hist_k; s) = \{\textbf{null}\}$$

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),*

$$VAL(Hist_k; s) = \oslash$$

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 that the safety theorem hold for previous cycles.

Suppose s is new to cycle k. Let τ 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 τ. In addition, if $j > 0$, when x was recycled, **null** was assigned to s, in line (4) of procedure **Collect**. Finally, as all dirty flags are cleated while the threads are halted, we have $Dirty(s)@HS_j =$ **false**. Since these values must remain in effect until time τ we can apply lemma A.1 to s and τ yielding that either claim (1) or (3) hold, depending on whether s has been modified prior to HS_k.

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_k.

□

A.1.3 Determined vs. undetermined slots

We say that the collector *determines* the value of a slot s if during the **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 **Read-Buffers** and **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 τ be the time at which the collector executed line (4) of **Update-Reference-Counters** . At time τ the flag was still off. Thus, no line (5) of procedure **Update** was scheduled in the interval HS_k to τ. Hence the later store from line (6) of **Update** hasn't been scheduled in this interval as well. This means that s remained unchanged from HS_k to τ. This interval includes the time at which the collector read the value of s, in line (3) of **Update-Reference-Counters**. Hence the collector read s to contain $s@HS_k$.

□

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 HS_k. i.e., T_i has stored the pair $\langle s, s@HS_k \rangle$ into its buffer and incremented $CurrPos_i$ prior to raising the flag. Thus, when the collector would process $Buffer_i$ during **Read-Buffers** it will see the logged pair $\langle s, s@HS_k \rangle$ in T_i's buffer ($s@HS_k$ is associated with s according to lemma A.2.) and thus the pair will be added to the set $Peek_k$.

We conclude the following:

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

$$VAL(Peek_k; s) = s@HS_k$$

A.1.4 Linking rc field with reference count

In this sectionwe 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 END_k denote the time at which cycle k has ended. That is, END_k is the earliest time at which all instructions of cycle k have already been scheduled.*

Definition A.3 *Let $COLLECT_k$ 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, $COLLECT_k$.*

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 HS_k.

Lemma A.5 (Meaning of The rc Field) *$o.rc@COLLECT_k = RC(o)@HS_k$ for any object o which is allocated at HS_k.*

Proof. The claim holds for $k = 0$ since there are no objects which are allocated at HS_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 s correctly. If s does not change after HS_{k-1} and before HS_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 HS_{k-1} is decremented. Likewise, we have to show that the value of the object that s was referring to at HS_k is incremented. s is in exactly one of these states at HS_k: allocated old, allocated new, non-allocated.

Decremening 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 $Hist_k$ will contain the pair $\langle s, s@HS_{k-1} \rangle$. $Hist_k$ will not contain elements associating s with a value other than $s@HS_{k-1}$. During the operation of

Update-Reference-Counters, when the pair $\langle s, s@HS_{k-1}\rangle$ is considered, the rc field of $s@HS_{k-1}$ is decremented, as desired.

Decrementing new slots: Let s be a new slot for cycle k. According to lemma A.2 either **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 **null** and its dirty flag was off at system startup. Also, no thread could have accessed s at HS_{k-1}, since it was not a part of a reachable object (or any object) at that time. Thus, $s@HS_{k-1} =$ **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 **Collect** during cycle j. If $j < k-1$ then $s@HS_{k-1} =$**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. **Reclaim-Garbage** and **Collect** can only reduce the value of an rc field. Thus, there is a single point during the operation of **Reclaim-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 $s@HS_k =$**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 **Fix-Undetermined-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 **Fix-Undetermined-Slots** terminates, so the claim holds at $COLLECT_k$.

□

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 **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 **Collect**) taking place prior to its reclamation, thus $o_1.rc@COLLECT_k = 0$. This implies, according to lemma A.5 applied to cycle k, that $RC(o)@HS_k = 0$. Additionally, by the code, o_1 is collected only if $o_1.rc = 0 \wedge o_1 \notin Locals_k$ so we conclude that in addition of not being pointed by any heap slot at HS_k, o_1 is also not pointed by any global or local reference at that particular moment, or it would have been marked *local*. Thus, o_1 is unreachable at HS_k.

For the inductive step, consider o_i which has $c \stackrel{\text{def}}{=} o_i.rc@COLLECT_k = RC(o)@HS_k$ (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_i is garbage at HS_k.

Otherwise, we have $c > 0$. Since o_i is recycled, it must satisfy at some point during **Reclaim-Garbage** or **Collect** $o_i.rc = 0 \wedge o_i \notin Locals_k$. Thus, the value of $o_i.rc$ 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_i we have by the inductive hypothesis that all c references to o_i were from objects that were unreachable at HS_k. Thus, at HS_k, o_i is pointed only by unreachable objects, and it is not pointed by any local thread state or global reference. We conclude that o_i is unreachable at HS_k.

□

A.2 Progress

In this sectionwe 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 [37].

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 [36], 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 $o.rc@COLLECT_{k-1} = 0$ then $RC(o)@HS_{k-1} = 0$ 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, $o.rc@COLLECT_{k-1} > 0 \wedge o.rc@END_{k-1} = 0$. This implies that $o.rc$ had reached zero by the decrements applied by one of the invocations of procedure **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.

□

Ideally, we would like the algorithm to collect at cycle k any object which is garbage at HS_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 HS_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.

Appendix 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 *local*. In addition, mutators carry the following write-barrier for global references:

1. $s := new$
2. if $Snoop_i$ then
// mark *new* as *local*.
3. $Locals_i := Locals_i \cup \{new\}$

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: $HS_k(i)$, $HS2_k(i)$, $HS3_k(i)$ and $HS4_k(i)$ 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:

$HS_k \stackrel{\text{def}}{=} \min_{T_i} HS_k(i)$
$HSEND_k \stackrel{\text{def}}{=} \max_{T_i} HS_k(i)$
$HS2_k \stackrel{\text{def}}{=} \min_{T_i} HS2_k(i)$
$HS2END_k \stackrel{\text{def}}{=} \max_{T_i} HS2_k(i)$
$HS3_k \stackrel{\text{def}}{=} \min_{T_i} HS3_k(i)$
$HS3END_k \stackrel{\text{def}}{=} \max_{T_i} HS3_k(i)$
$HS4_k \stackrel{\text{def}}{=} \min_{T_i} HS4_k(i)$
$HS4END_k \stackrel{\text{def}}{=} \max_{T_i} HS4_k(i)$

Additionally we define $COLLECT_k$ to be the time at which procedure **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.
- If $m < k - 1$, 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 sectionwe 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 σ_k that we associate with each cycle. We abbreviate V_{σ_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 $HS3_k(i)$ then let v be the consolidated value chosen for s. Let τ be the time a particular thread T_j loaded v before logging the pair $\langle s, v \rangle$. Set $\sigma_k(s) \stackrel{\text{def}}{=} \tau$.
  - em Rule 3: otherwise, no thread T_i logs s prior to $HS3_k(i)$, but s is logged by some thread T_j prior to $HS_k(i)$. On such an event set $\sigma_k(s) \stackrel{\text{def}}{=} HS2END_k$.

Note that $\sigma_k(s)$ is uniquely defined. We denote by $R1_k$ the set of all slots whose definition of σ_k is derived by rule (1). Similarly we define the sets $R2_k$ and $R3_k$.

The next lemma characterizes the span of σ_k.

Lemma B.1 $Start(\sigma_k) \geq HS_k \wedge End(\sigma_k) \leq HSU3END_k$

Proof. Let s be a memory word. Certainly if $s \in R1_k \cup R3_k$ then $\sigma_k(s)$ lies within the specified time limits. Otherwise, s is defined according to rule (2). we note that τ must be earlier than $HS3END_k$ as some thread is logging s prior to responding the third handshake. If this logging is done during clearing than 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., $o \notin Locals_k$) at $COLLECT_k$ satisfies*

$$ARC(V_k; o) \geq RC(o)@HS4_k$$

Moreover, the set of pointers that point to o at $HS4_k$ is a subset of those that point to it in V_k.

Proof. According to lemma 5.1 it suffices to show that if a reference to o is stored to a slot s at, or after $\sigma_k(s)$ and before $End(\sigma_k)$, then o is marked local. By lemma B.1 we know that $End(\sigma_k) < HS4_k$, hence we can replace $End(\sigma_k)$ with $HS4_k$, hardening the requirements of lemma 5.1. i.e., we require that if a reference to o is stored to a slot s during the interval $[\sigma_k(s), HS4_k)$ then o is marked local.

Since updates are not interruptible and since the $Snoop_i$ flag is reset only after $HS4_k(i)$, 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 $HS4_k(i)$. Consider a store of o into s which is scheduled at, or after $\sigma_k(s)$ and before $HS4_k(i)$. 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*.

□

Lemma B.3 *The following claims hold: (1) if thread T_i logs s between responding to the first and third handshakes then $Dirty(s)@HS3_k(i) =$**true**. (2) if thread T_i logs s between responding to the first and fourth handshakes then $Dirty(s)@HS4_k(i) =$**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 $HS2_k$. Claim (2) is trivial due to the validity of claim (1).

□

B.4 Road map 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:

$$HS_{-1} \stackrel{\text{def}}{=} HS_0$$

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 sectionwe 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 $k > 0$
  - $LB.4_k \Longleftarrow LB.4_{k-1}$
  - $LB.5_k \Longleftarrow LB.4_{k-1} \wedge \bigwedge_{j<k} SafetyTheorem_j$
  - $LB.6_k \Longleftarrow LB.5_k \wedge LB.6_{k-1} \wedge LB.7_{k-1} \wedge \bigwedge_{j<k}(SafetyTheorem_j)$
  - $SafetyTheorem_k \Longleftarrow LB.6_k$
  - $LB.7_k \Longleftarrow LB.5_k \wedge SafetyTheorem_k$

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 **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* $s \in R1_k$ *than* $t_k(s) \stackrel{\text{def}}{=} \sigma_k(s) \stackrel{\text{def}}{=} HS_k$.
- *if* $s \in R2_k$ *then* $t_k(s)$ *exists and it satisfies* $HSEND_k < t_k(s) < HS2_k$.
- *if* $s \in R3_k$ *then* $t_k(s) \stackrel{\text{def}}{=} HS2END_k$. *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 $R1_0$. 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 $s \in R1_k$ then either $s \in R1_{k-1}$ or $s \in R3_{k-1}$. $R2_{k-1}$ is impossible because it implies that $s \in Hist_k$.

  If $s \in R1_{k-1}$ then by the inductive hypothesis $Dirty(s)@HS_{k-1} = \textbf{false}$. Had some thread T_i turned on the flag on after HS_{k-1} and before $HS_k(i)$ 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, $s \in R3_{k-1}$. Thus, according to the inductive hypothesis $Dirty(s)@HS2END = \textbf{false}$. By definition of $R3_{k-1}$, 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 $Dirty(s)@HS_k = \textbf{false}$.

- if $s \in R2_k$ 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 $s \in R3_k$ 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 $HS2END_k$ 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 $R3_k$. We conclude that there are no ongoing updates at $HS2END_k$.

□

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* $VAL(Hist_k; s) = \{\textbf{null}\}$.
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* $VAL(Hist_k; s) \supseteq \{\textbf{null}\}$.

Proof. For garbage collection number zero the claims trivially hold since $Hist_0 = \oslash$ 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 $s \notin Hist_k$. In that case we have $\sigma_k(s) \stackrel{\text{def}}{=} HS_k$ and we have to show that s is not changed between HS_{k-1} and HS_k.

Since $s \notin Hist_{k-1}$ we conclude, by the inductive hypothesis, that no thread modified s between $\sigma_{k-2}(s)$ and HS_{k-1}. Additionally we know that at HS_{k-1} the dirty mark of s is off. The dirty mark must be off at $HS4END_{k-2}$ as well and no update is ongoing at the moment as that update would have rendered s part of $Hist_{k-1}$. Using the same arguments of lemma A.1 applied for s and $HS4END_{k-2}$ and since s is not cleared before $HSEND_k$ any update whose store proper operation is scheduled between $HS4END_{k-2}$ and HS_k would result in the association of $s@HS4END_{k-2}$ with s in either $Hist_{k-1}$, or $Hist_k$, neither of which is the case. We conclude that s is indeed not modified during cycle $k-1$.

Now suppose $s \in Hist_k$. In that case we want to show that $VAL(Hist_k; s) = s@HS_{k-1}$. Again, we've concluded that any thread T_i that would log s prior to $HS_k(i)$ would associate it with $s@HS4END_{k-2}$. Since a store to s could not have been scheduled between $s@HS4END_{k-2}$ and HS_{k-1} without logging the slot we conclude that $s@HS_{k-1} = s@HS4END_{k-2}$, 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 $HS4_{k-1}$, 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 $Hist_k$. By definition of σ_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 $t_{k-1}(s) = \sigma_{k-1}(s) = HS2END_{k-1}$ and no update is occurring at that moment. Suppose $s \notin Hist_k$. In that case $\sigma_k(s) = HS_k$ and we have to show that no store is scheduled between $HS2END_{k-1}$ and HS_k. But this is trivial since the probing of the dirty mark associated with such a store must start after $HS2END_{k-1}$, as no updates occur at that moment. Thus, had such an update been scheduled, it must have sensed that the flag is off and s would have become a member of $Hist_k$ a contradiction.

Suppose now that $s \in Hist_k$. We have to show that $VAL(Hist_k; s) = s@HS2END_{k-1}$. Again, since at $HS2END_{k-1}$ 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 $s@HS2END_{k-1}$ with s.

new slots allocated for the first time. If s is allocated for the first time, then $\sigma_{k-1} \stackrel{\text{def}}{=} HS_{k-1}$ and at that time s contained **null** and its dirty flag was initialized to **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 $Hist_k$ 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 $Hist_k$ 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 re-allocated, 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 $HS4_{k-1}(i)$. s could have not occurred in the digested part of $Hist_k$ 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 **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 **null**, and the dirty flag for it was **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.

□

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 $COLLECT_k$ it holds that $o.rc@COLLECT_k = ARC(V_k, o)$.*

Proof. The claim trivially holds for collection cycle zero, since there are no allocated objects at $COLLECT_0$. 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 $[COLLECT_{k-1}, COLLECT_k)$, 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 **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 $\sigma_{k-1}(s) \stackrel{\text{def}}{=} HS_{k-1}$ and by the definition of sliding views we have $V_{k-1}(s) =$**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 **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 **null** at the time it was allocated.

We conclude that at any rate, by the time s is allocated, it contains **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 $V_k(s) \neq$**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 $\sigma_k(s) \stackrel{\text{def}}{=} HS_k$. At $\sigma_k(s)$ s still assumes the value of **null** and thus $V_k(s) =$**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, $VAL(Hist_k; o) = \{$**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 **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 $V_(s)$. Otherwise, when s is undetermined, the collector adds it to the set $Undetermined_k$. 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 $VAL(Hist_k; s) = \oslash$ 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 **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 $HS4END_m$. Thus, at HS_{k-1} 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.

□

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 $HS4END_k$.*

Proof. We prove the claim by induction on the cycle number, k. For $k = 0$ we have an empty ZCT_0 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 **ReclaimGarbage** for it. Thus, $0 = o_1.rc@COLLECT_k$. This implies

$$0 = ARC(V_k, o) \geq RC(o)@HS4_k \Longrightarrow RC(o)@HS4_k = 0$$

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. $RC(o_1) = 0$ continuously during the interval, and

2. o_1 was not directly reachable from any of the threads in the set $P \stackrel{\text{def}}{=} \{T_1, \ldots, T_{i-1}\}$ 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 $T_l \in P$ 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 $T_l \notin P$ cannot violate (1) or (3) because the operand of the store cannot be o_1 since the $Snoop_l$ flag is set during the interval and such a step would have marked o_1 *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 *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 $o_j.rc@COLLECT_k = 0$ then the same arguments that were employed for o_1 are repeated. Otherwise, we have

$$c \stackrel{\text{def}}{=} o_j.rc@COLLECT_k > 0$$

Since o_j is eventually processed by **Collect** there must have been c slots pointing at o_j that were cleared and $o_j.rc$ decremented accordingly, in lines (7-8) of **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 c 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 c slots after responding to the fourth handshake. We conclude that these c slots have not been changed after $HS4_k$ and before the collector processed them.

In order to prove **I1** we prove an additional invariant:

Invariant B.4 (I4) *No reference to o_j has been stored during the interval $[HS4_k(1), HS4_k(n)]$ to either a heap slot or a global reference.*

Define **I5** as the logical conjunction of **I1** and **I4**. We prove that **I5** holds for o_j.

We have already said that at $HS4_k(1)$ 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_1 at $HS4_k(1)$. Additionally, o_j was not directly reachable from T_1 at $HS4_k(1)$, or it would have been marked local. o_j has not been directly reachable from a global reference at $HS4_k(1)$ since that would have caused it being marked *local*, for the same arguments that were applied for o_1. Finally, had o_j been indirectly reachable from a global reference r at $HS4_k(1)$ 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_1 at $HS4_k(1)$, contradicting the inductive hypothesis on objects. Thus, **I1**, restricted to the interval $[HS4_k(1), HS4_k(1)]$ holds for o_j.

I4, restricted to the interval $[HS4_k(1), HS4_k(1)]$, holds as well since $HS4_k(1)$ is the time at which T_1 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_1 that **I5** restricted to the interval $[HS4_k(i-1), HS4_k(i)]$, where $1 < i \leq n$, holds provided it holds during the interval $[HS4_k(1), HS4_k(i-1)]$. We also use the inductive hypothesis on j that asserts that for any object o_a, $a < j$, **I1** holds for the entire interval $[HS4_k(1), HS4_k(n)]$.

Invariant **I5** applied to o_j and restricted to the interval $[HS4_k(i-1), HS4_k(i)]$ 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 $HS4_k(i)$.

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 by a thread T_l could not have maid o_j reachable from T_l 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 $T_l \in P$ cannot make o_j reachable for any thread in P unless o_j has been already reachable from T_l just before the action took place, which is not the case. So a store by T_l preserves (1), (2) and (3) provided (1) and (3) hold for previous steps.
- a store by a thread $T_l \notin P$ cannot make o_j reachable from any thread in P for the following reasons:
  - T_l 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_l 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 $HS4_k$. 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 $T_l \notin P$.

- 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 de-allocates 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 $Dirty(s)@\sigma_k(s) = \textbf{false}$ 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.

Appendix 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. $s := new$
2. if $Snoop_i$ then
    // mark *new* as *local*.
3. $Locals_i := Locals_i \cup \{new\}$

Let $MARK_k$ 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 $white@MARK_k$ when tracing starts.

Lemma C.1 *Let s be a slot such that $V_k(s) = o \neq$ **null**. Then $o.color@MARK_k = white@MARK_k$.*

Proof. if $V_k(s) \neq$ **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 reclaimed. 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 $HS4_k(i)$ and which is reclaimed during cycle k. For each thread T_l it holds that o is inaccessible from T_l from $HS4_k(l)$ onward.*

Proof. We assume that the threads are ordered by their response time to the fourth handshake, i.e., $HS4_k(1) < HS4_k(2) \ldots < HS4_k(n)$. 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 $HS4_k(1)$ 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 $HS4_k(1)$ is bound to snoop its operand, as the $Snoop_i$ flags are all set at $HS4_k(1)$ 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 $HS4_k(1)$. That is, there exists a local reference of T_1 or a global reference r such that at $HS4_k(1)$:

$$r = x_1 \wedge \exists s_1 \in x_1 : s_1 = x_2$$

$$\exists s_2 \in x_2 : s_2 = x_3$$

$$\vdots$$

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

If the chain existed in this exact form in V_k, i.e., $\forall j \leq m : s_j @ HS4_k(1) = V_k(s_j)$, then tracing through r will eventually blacken o, according to the observation. If, on the other hand, there exists a slot s_l in the chain which has been modified since $\sigma_k(s_l)$ then let s_l be the highest indexed slot with a modified value, that is, $\forall l < j \leq m : s_j @ HS4_k(1) = V_k(s)$. By lemma B.2 we know that when the pointer to x_{l+1} was stored into the slot s_l the storing thread marked 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 $[HS4_k(1), HS4_k(1)]$.

We now show that if the claim holds in the interval $[HS4_k(1), HS4_k(i-1)]$, where $i > 0$ then it holds in the interval $[HS4_k(i-1), HS4_k(i)]$. Specifically, we have to show that:

1. o remained inaccessible during the interval from any local reference of any thread in the set $P \stackrel{\text{def}}{=} \{T_1, \ldots, T_{i-1}\}$.

2. o remained inaccessible during the interval from any global reference.

3. o was inaccessible from T_i at $HS4_k(i)$.

In order to prove the claims (1) and (2) we note that any individual load or store operation by a thread $T_l \in P$ cannot render o reachable from T_l 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 $[HS4_k i-1, t]$ where $HS4_k(i-1) \leq t < HS4_k(i)$ 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 $Snoop_j$ 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 σ_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 $HS4_k(i)$. 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 $HS4_k(i)$ andV_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.

□

We conclude that:

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

Appendix D

Source Code

In this appendix we bring the source code listings of the garbage collector. We had to change many files in the Javasoft JVM is order to implement the write barrier required by the on-the-fly algorithms. This is the complete list of files that required a change due to the implementation of the write barrier:

```
./src/share/javavm/include/alloc_cache.h
./src/share/javavm/include/gc.h
./src/share/javavm/include/interpreter.h
./src/share/javavm/include/oobj.h
./src/share/javavm/runtime/classinitialize.c
./src/share/javavm/runtime/classload.c
./src/share/javavm/runtime/classresolver.c
./src/share/javavm/runtime/classruntime.c
./src/share/javavm/runtime/executeJava.c
./src/share/javavm/runtime/executeJava_p5.inc
./src/share/javavm/runtime/executeJava_p5.m4
./src/share/javavm/runtime/interpreter.c
./src/share/javavm/runtime/javai.c
./src/share/javavm/runtime/jni.c
./src/share/javavm/runtime/jvm.c
./src/share/javavm/runtime/jvmpi.c
./src/share/javavm/runtime/threads.c
./src/share/javavm/runtime/util.c
./src/win32/hpi/src/threads_md.c
./src/win32/javavm/runtime/signals_md.c
./src/win32/native/sun/awt_common/awt_makecube.cpp
./src/win32/native/sun/windows/awt.h
./src/win32/native/sun/windows/awt_Component.cpp
./src/win32/native/sun/windows/awt_Cursor.cpp
./src/win32/native/sun/windows/awt_Dialog.cpp
./src/win32/native/sun/windows/awt_DnDDS.cpp
./src/win32/native/sun/windows/awt_Font.cpp
./src/win32/native/sun/windows/awt_Graphics.cpp
./src/win32/native/sun/windows/awt_MenuItem.cpp
./src/win32/native/sun/windows/awt_PrintJob.cpp
./src/win32/native/sun/windows/awt_Robot.cpp
./src/win32/native/sun/windows/awt_Toolkit.cpp
./src/win32/native/sun/windows/awt_Window.cpp
```

The list of files that actually implement the garbage collector and allocator themselves is considerably shorter and is given below. In the rest of this appendix we list the source of this files along with a description of the role of each one of them.

```
./src/share/javavm/include/mok_win32.c
./src/share/javavm/include/rcblkmgr.c
./src/share/javavm/include/rcchunkmgr.c
./src/share/javavm/include/rcgc.c
./src/share/javavm/include/rcgc.h
./src/share/javavm/include/rcbmp.c
./src/share/javavm/include/rcbmp_inline.h
./src/share/javavm/include/rcgc_internal.h
./src/share/javavm/include/rchub.c
./src/share/javavm/include/ylrc_protocol.h
./src/share/javavm/runtime/gc.c
```

D.1 Organization of the code

The garbage collector code contains only one translation unit, which is the file `gc.c`. This file is inherited from the original JVM. It contains, among other things, the entry point to user's allocation code. This is the only file that was inherited from the original JVM, all other files are specific to the new collector.

The file `gc.c` includes the file `rchub.c`, which in turn includes the files `rcblkmgr.c` (the block manager), `rcchunkmgr.c` (the chunk manager) and `rcgc.c`. Thus, all code is lumped into one translation unit. This allows us to use `static` and `inline` functions extensively, which opens the room for compiler optimizations.

D.2 mok_win32.c

We tried to keep the garbage collector portable. For that end, we encapsulated the required Win32 into a single file: mok_win32.c. The services include low level memory management and thread support.

Source listing for file mok_win32.c

```
/* File name: mok_win32.c
 * Author:    Yossi Levaoni
 * Purpose:   Win32 abstraction layer
 */

/*
 * Memory
 *
 */
/* Advanced */

#define WIN32PGGRANULE (64*1024)

void* mokMemReserve(void *starting_at_hint, unsigned sz )
{
  void *p = VirtualAlloc( starting_at_hint, sz, MEM_RESERVE, PAGE_READWRITE );
  sysAssert( sz );
  sysAssert( p );
  return p;
}

void mokMemUnreserve( void *start, unsigned sz )
{
  BOOL res;
  mokMemDecommit( start, sz );
  res = VirtualFree( start, 0, MEM_RELEASE );
  sysAssert( res );

}
```

```
void* mokMemCommit( void *start, unsigned sz, bool zero_out )
{
  void *p = VirtualAlloc( start, sz, MEM_COMMIT, PAGE_READWRITE );
  sysAssert( start );
  sysAssert( sz );
  sysAssert( p );
  return p;
}

void mokMemDecommit( void *start, unsigned sz )
{
  BOOL res;
  sysAssert( start );
  sysAssert( sz );
  res = VirtualFree( start, sz, MEM_DECOMMIT );
  sysAssert( res );
}

/* C style */
void* mokMalloc( unsigned sz, bool zero_out )
{
  void *p;
  sysAssert( sz );
  p = malloc( sz );
  sysAssert( p );
  if (zero_out)
    memset( p, 0, sz );
  return p;
}

void mokFree( void * p)
{
  sysAssert( p );
  free( p );
}

/* zero out */
void mokMemZero( void *start, unsigned sz )
{
  mokMemDecommit( start, sz );
  mokMemCommit( start, sz, TRUE );
}


/*
 * YLRC --
 *
 * The functions:
 *
 *   mokThreadSuspendForGC
 *   mokThreadResumeForGC
 *
 * are needed for on the fly garbage collection
 *
 */
void mokThreadSuspendForGC(sys_thread_t *tid)
{
  sysAssert( tid != sysThreadSelf() );

  if (SuspendThread(tid->handle) == 0xffffffffUL) {
    jio_printf( "sysThreadSuspendForGC: SuspendThread failed" );
    __asm { int 3 }
  }
  {
    CONTEXT context;
    DWORD *esp = (DWORD *)tid->regs;

    context.ContextFlags = CONTEXT_INTEGER | CONTEXT_CONTROL;
    if (!GetThreadContext(tid->handle, &context)) {
      jio_printf( "sysThreadSuspendForGC: GetThreadContext failed" );
      __asm { int 3 }
    }
    *esp++ = context.Eax;
    *esp++ = context.Ebx;
    *esp++ = context.Ecx;
    *esp++ = context.Edx;
    *esp++ = context.Esi;
    *esp++ = context.Edi;
    *esp   = context.Ebp;
  }
}

void mokThreadResumeForGC(sys_thread_t *tid)
{
  sysAssert( tid != sysThreadSelf() );

  if (ResumeThread(tid->handle) == 0xffffffffUL) {
    printf( "sysThreadResumeForGC: ResumeThread failed" );
    __asm { int 3 }
  }
}
```

```
typedef struct xxpair {
  int (*func)(sys_thread_t*, void*);
  void *param;
} xxpair;

static int  _mokThreadEnumerateOverHelper( sys_thread_t *thrd, xxpair* xx)
{
  int res;
  ExecEnv *ee;
  if (thrd == gcvar.sys_thread) return SYS_OK;
  ee = SysThread2EE( thrd );
  if (!ee->gcblk.gcInited) return SYS_OK;
  res = xx->func( thrd, xx->param );
  return res;
}

int mokThreadEnumerateOver( int(*f)(sys_thread_t *, void*), void *param)
{
  xxpair xx;
  int ret;

  xx.func = f;
  xx.param = param;

#ifdef RCDEBUG
  {
    sys_thread_t* self = sysThreadSelf();
    mokAssert( self == gcvar.sys_thread );
  }
#endif
  ret = sysThreadEnumerateOver( _mokThreadEnumerateOverHelper, &xx );
  return ret;
}
```

End of file source listing

D.3 rcblkmgr.c

This file contains the code of the block manager (see section8.9 for more details).

Source listing for file rcblkmgr.c

```
/*
 * File:    rcblkmgr.c
 * Author:  Mr. Yossi Levanoni
 * Purpose: implementation of the block manager
 */
/****************** Initialization ******************************/
GCFUNC void blkInit(unsigned nMB)
{
  unsigned      sz;

  /* Zero out all vars */
  memset( &blkvar, 0, sizeof(blkvar) );

  /* Allocate the heap */
  mokAssert( nMB < (1<<BLOCKBITS) && nMB>0);
  blkvar.heapSz = nMB << 20;
  blkvar.heapStart = (byte*)mokMemReserve( NULL, blkvar.heapSz );
  blkvar.heapTop = blkvar.heapStart + blkvar.heapSz;
  mokMemCommit( blkvar.heapStart, blkvar.heapSz, false );

#ifdef RCVERBOSE
  jio_printf(
         "heap[%x<-->%x]\n",
         (unsigned)blkvar.heapStart,
         blkvar.heapSz + (unsigned)blkvar.heapStart);
  fflush( stdout );
#endif

  /* Allocate block headers table */
  blkvar.nWildernessBlocks = blkvar.nBlocks = blkvar.heapSz >> BLOCKBITS;
  sz = sizeof( BlkAllocHdr ) * (blkvar.nBlocks + 3);
  blkvar.allocatedBlockHeaders  = (BlkAllocHdr*)mokMemReserve( NULL, sz );
  mokMemCommit( blkvar.allocatedBlockHeaders, sz, true );

  blkvar.allocatedBlockHeaders ++;

  blkvar.pRegionLists =
    (BlkListHdr*)blkvar.allocatedBlockHeaders + blkvar.nBlocks + 1;


  bhSet_status( (blkvar.allocatedBlockHeaders-1) , DUMMYBLK );
  bhSet_status( (blkvar.allocatedBlockHeaders+blkvar.nBlocks) , DUMMYBLK );

  blkvar.blockHeaders =
```

```
    blkvar.allocatedBlockHeaders - ((unsigned)blkvar.heapStart>>BLOCKBITS);

  blkvar.heapTopRegion = (BlkRegionHdr*)OBJBLOCKHDR( blkvar.heapTop );
  blkvar.wildernessRegion = (BlkRegionHdr*)OBJBLOCKHDR( blkvar.heapStart );

  /* Allocate mutex */
  blkvar.blkMgrMon = sysMalloc(sysMonitorSizeof());
  sysMonitorInit( blkvar.blkMgrMon );

#ifdef RCDEBUG
  jio_printf("headers1[%x<-->%x]\n",
 (unsigned)blkvar.allocatedBlockHeaders,
 sz + (unsigned)blkvar.allocatedBlockHeaders);
  jio_printf("headers2[%x<-->%x]\n",
 OBJBLOCKHDR(blkvar.heapStart),
 OBJBLOCKHDR( (((byte*)blkvar.heapStart)+(nMB<<20)) ) );
#endif
}



/*****************************************************
*                   LOCKING                          *
*****************************************************/

static void _LockBlkMgr(sys_thread_t *thrd)
{
  sysMonitorEnter( thrd, blkvar.blkMgrMon );
}

static void _UnlockBlkMgr(sys_thread_t* thrd )
{
  sysMonitorExit( thrd, blkvar.blkMgrMon );
}


/*****************************************************
* Allocate nBlocks from the part of the heap that
* hasn't been touched thus far.
*****************************************************/
static  BlkAllocHdr* _allocFromWilderness( int nBlocks )
{
  BlkRegionHdr* base = blkvar.wildernessRegion;
  BlkRegionHdr* target =  base + nBlocks;
  if (target > blkvar.heapTopRegion)
    return NULL;
  blkvar.wildernessRegion = target;

  return (BlkAllocHdr*)base;
}

/*****************************************************
 *
 * Insert this block, with the specified size, into the
 * respective quick list.
 *
 * No merging with neighboring regions is attempted nor
 * should be applicable.
 *
 * The limitting blocks have their "regionSize" set.
 *****************************************************/
static void _insertRegionIntoQuickLists( BlkRegionHdr *brh, int sz )
{
  BlkRegionHdr *lastBlk = brh + (sz-1);

  brh->StatusUnused = BLK << 24;
  brh->regionSize = sz;

  if (lastBlk != brh) {
    lastBlk->StatusUnused = BLK << 24;
    lastBlk->regionSize = -sz;
  }

  brh->nextRegion = blkvar.quickLists[sz];
  if (brh->nextRegion)
    brh->nextRegion->prevRegion = brh;
  brh->prevRegion = (BlkRegionHdr *)&blkvar.quickLists[sz];
  blkvar.quickLists[sz] = brh;
}

/*****************************************************************
* Insert a region into the list of lists of regions.  If a list
* for the region size exists then it is added to it.  Otherwise,
* a new list is inserted to the list of lists for holding regions
* of "sz" blocks.
*
* If the region becomes an element in a list of regions than its
* "regionSize" field is updated to "sz".  The last block in the
* region has its size updated to "-sz" at any rate.
*****************************************************************/
static void _insertRegionIntoRegionLists( BlkRegionHdr *brh, int sz )
{
```

```
  int regionSize = -1;
  BlkListHdr *pPrevList, *pList;
  BlkListHdr *blh = (BlkListHdr *)brh;

  BlkRegionHdr *lastBlk = brh + (sz-1);

  lastBlk->StatusUnused = BLK << 24;
  lastBlk->regionSize = -sz;

  mokAssert( sz > 1 );

  pList = blkvar.pRegionLists->nextList;
  pPrevList = blkvar.pRegionLists;
  for (; pList; pPrevList = pList, pList = pList->nextList) {
    regionSize = pList->listRegionSize;
    if (sz <= regionSize)
      break;
  }


  /**
   * Perfect match
   */
  if (regionSize == sz ) {

    brh->StatusUnused = BLK<<24;
    brh->regionSize = sz;

    brh->nextRegion = pList->nextRegion;
    brh->prevRegion = (BlkRegionHdr *)pList;
    if (pList->nextRegion)
      pList->nextRegion->prevRegion = brh;
    pList->nextRegion = brh;

    return;
  }

  /**
   * Create new empty list.
   */
  blh->nextRegion = NULL;

  blh->StatusPrevListID = BLKLIST << 24;
  blh->listRegionSize = sz;

  /**
   * we want to insert after pPrevList and before
   * pList.
   */
  bhSet_prev_region_list( blh, pPrevList );
  blh->nextList = pList;
  pPrevList->nextList = blh;
  if (pList) {
    bhSet_prev_region_list( pList, blh);
  }
}

/*****************************************************
* Extract the argument region from the list it's
* in.  Assumes that the region is not a list header.
*****************************************************/
static void _extractFromRegionList( BlkRegionHdr *ph )
{
  ph->prevRegion->nextRegion = ph->nextRegion;
  if (ph->nextRegion)
    ph->nextRegion->prevRegion = ph->prevRegion;
}

/*****************************************************
* Extract the argument region, which is a list header,
* from the list of lists.
*****************************************************/
static void _extractFromListOfLists( BlkListHdr *ph )
{
  BlkListHdr *newHeader = (BlkListHdr *)ph->nextRegion;
  BlkListHdr *prevList = bhGet_prev_region_list( ph );

  /**
   * Change list header to the next element in the
   * list
   */
  if (newHeader) {
    int sz = ((BlkRegionHdr *)newHeader)->regionSize;
    bhSet_prev_region_list( newHeader, prevList );
    newHeader->nextList = ph->nextList;

    prevList->nextList = newHeader;
    if (newHeader->nextList) {
      bhSet_prev_region_list( newHeader->nextList, newHeader );
    }
    bhSet_status( newHeader, BLKLIST );
    newHeader->listRegionSize = sz;
```

```
    return;
  }
  /**
   * Eliminate the list.
   */
  prevList->nextList = ph->nextList;
  if (ph->nextList) {
    BlkListHdr *prevList = bhGet_prev_region_list( ph );
    bhSet_prev_region_list( ph->nextList, prevList );
  }
}

/*****************************************************
* See if the region adjacent to the argument region
* from the right (i.e., with higher address) is in
* the hands of the block manager.
*
* If so, extract it from wherever it is.
*****************************************************/
static void _tryExtractRightNbr( BlkRegionHdr **pph, int *pSz)
{
  BlkRegionHdr *nbr = *pph + *pSz;
  int status = bhGet_status( nbr );
  int size = nbr->regionSize; // conicides with the size field of BLKLIST

#ifdef RCDEBUG
  if (status==BLK || status==BLKLIST) {
    BlkRegionHdr *lastBlock = nbr + size - 1;
    mokAssert( size > 0 );
    mokAssert( bhGet_status( lastBlock ) == BLK );
    mokAssert( lastBlock==nbr || lastBlock->regionSize == -size );
  }
#endif

  if (status == BLK) {
    _extractFromRegionList( nbr );
    blkvar.nListsBlocks -= size;
    *pSz += size;
  }
  else if (status == BLKLIST) {
    BlkListHdr *blh = (BlkListHdr *)nbr;
    _extractFromListOfLists( blh );
    blkvar.nListsBlocks -= size;
    *pSz += size;
  }
}

/*****************************************************
* See if the region adjacent to the argument region
* from the left (i.e., with lower address) is in
* the hands of the block manager.
*
* If so, extract it from wherever it is.
*****************************************************/
static void _tryExtractLeftNbr( BlkRegionHdr **pph, int *pSz)
{
  BlkRegionHdr *nbr = *pph - 1;
  int status = bhGet_status( nbr );
  int size = nbr->regionSize==1 ? 1 : -nbr->regionSize;

  /**
   * That's because items in the list are
   * bigger than a single block and their
   * final block is marked with BLK.
   */
  mokAssert( status != BLKLIST );

  if (status == BLK) {
    mokAssert( size > 0 );
    nbr = nbr + 1 - size;

    status = bhGet_status( nbr );
    mokAssert( nbr->regionSize == size );

    if (status == BLK) {
      _extractFromRegionList( nbr );
    }
    else {
      mokAssert( status == BLKLIST );
      _extractFromListOfLists( (BlkListHdr *)nbr );
    }

    blkvar.nListsBlocks -= size;

    *pSz += size;
    *pph = nbr;
  }
}

/*************************************************************
* Free the specified region:
*
```

```
* 1. see if it can be added to the wilderness.
* 2. if not, try coalescing from the left and right.
* 3. finally, add the resulting block to either the
*    quick lists or the list of lists, depending on its
*    size.
*************************************************************/
static void _blkFreeRegion_locked( BlkRegionHdr *ph, int sz )
{
  blkvar.nAllocatedBlocks -= sz;

  _tryExtractLeftNbr( &ph, &sz );

  if (ph + sz == blkvar.wildernessRegion) {
    blkvar.wildernessRegion = ph;
    blkvar.nWildernessBlocks += sz;
    return;
  }

  _tryExtractRightNbr( &ph, &sz );

  blkvar.nListsBlocks += sz;

  if (sz<N_QUICK_BLK_MGR_LISTS )
    _insertRegionIntoQuickLists( ph, sz );
  else
    _insertRegionIntoRegionLists( ph, sz );
}

/*****************************************************
 * Find the first non-empty list with size at least
 * "sz".  Then take the first element out.
 * If there is leftover, put it in the respective list.
 *****************************************************/
static BlkAllocHdr* _allocFromQuickLists( unsigned sz )
{
  BlkRegionHdr** pList = &blkvar.quickLists[sz];
  BlkRegionHdr*  brh, *nextB;
  unsigned i;

  for (i=sz; i<N_QUICK_BLK_MGR_LISTS; i++, pList++) {
    brh = *pList;
    if (brh)
      goto __found_list;
  }
  return NULL;

 __found_list:

  nextB = brh->nextRegion;
  if (nextB)
    nextB->prevRegion = (BlkRegionHdr *)pList;
  (BlkRegionHdr*)*pList = nextB;

  if (sz != i) {
    BlkRegionHdr *leftover = brh + sz;
    int newSz = i - sz;
    _insertRegionIntoQuickLists( leftover, newSz );
  }
  return (BlkAllocHdr*)brh;
}

/*************************************************
*
* Allocates "sz" blocks from the lists of regions.
* Try finding a list with elements at list of size
* "sz".
*
* If the found list contains additional elements
* besides the header, then the element after the
* header is extracted from the list.
*
* Otherwise, the list header itself is extracted
* from the list of lists.
*
* Finally, if the list is not an exact match, the
* leftover is returned to the system.
*************************************************/
static BlkAllocHdr* _allocFromRegionLists( int sz )
{
  BlkRegionHdr *brh;
  BlkListHdr *pList = blkvar.pRegionLists->nextList;
  int regionSize, leftover;

  for (; pList; pList = (BlkListHdr *)pList->nextList) {
    regionSize = pList->listRegionSize;
    if (sz <= regionSize)
      goto __found_list;
  }
  return NULL;

 __found_list:
```

```
  brh = pList->nextRegion;
  if (brh) { // extract next element in the list
    BlkRegionHdr *nextB = brh->nextRegion;
    if (nextB)
      nextB->prevRegion = (BlkRegionHdr*)pList;
    pList->nextRegion = nextB;
  }
  else { // extract list header itself
    BlkListHdr *prevList = bhGet_prev_region_list( pList );
    if (pList->nextList) {
      bhSet_prev_region_list( pList->nextList, prevList);
    }
    /**
     * the next assignment may update *blkvar.pRegionLists
     * itself since the first element in the list
     * has its prevList pointer pointing at this
     * variable.
     */
    prevList->nextList = pList->nextList;
    brh = (BlkRegionHdr*)pList;
  }

  // do we have leftover
  leftover = regionSize - sz;

  if (leftover >= N_QUICK_BLK_MGR_LISTS) {
    _insertRegionIntoRegionLists( brh + sz, leftover );
  }
  else if (leftover >= 1) {
    _insertRegionIntoQuickLists( brh + sz, leftover );
  }
  return (BlkAllocHdr*)brh;
}

static void _sweepBig(BlkAllocBigHdr *ph)
{
  GCHandle *h;
  uint *p;

  if (ph->allocInProgress) return;

  h = (GCHandle*)BLOCKHDROBJ( (BlkAllocHdr*)ph );

  if (gcGetHandleRC(h)>0) return;

  p = h->logPos;

  if (p) {
    mokAssert( ((*p)&~3) == (uint)h );
    mokAssert( ((*p)&3) == 0 || ((*p)&3) == BUFF_HANDLE_MARK);
    /* leave it for next cycle */
    return;
  }
#ifdef RCDEBUG
  gcvar.dbg.nFreedInCycle++;
  gcvar.dbg.nBytesFreedInCycle += ph->blobSize * BLOCKSIZE;
#endif
  blkFreeRegion( ph );
}

/*********************************************************
* Allocate "nBlocks" of memory. Self explaining.
*
*********************************************************/
static BlkAllocHdr* _blkAllocRegion_locked( int nBlocks )
{
  BlkAllocHdr *res;

  if (nBlocks < N_QUICK_BLK_MGR_LISTS) {
    res = _allocFromQuickLists( nBlocks );
    if (res) {
      blkvar.nAllocatedBlocks += nBlocks;
      blkvar.nListsBlocks -= nBlocks;
      goto __checkout;
    }
  }
  res = _allocFromRegionLists( nBlocks );
  if (res) {
    blkvar.nAllocatedBlocks += nBlocks;
    blkvar.nListsBlocks -= nBlocks;
    goto __checkout;
  }

  res = _allocFromWilderness( nBlocks );
  if (!res) return NULL;
  blkvar.nAllocatedBlocks += nBlocks;
  blkvar.nWildernessBlocks -= nBlocks;

 __checkout:
  return res;
}
```

```
static int _calcAllocSize(int nBytes)
{
  int blocks = nBytes / BLOCKSIZE;
  if (blocks==0 || nBytes%BLOCKSIZE)
    blocks++;
  return blocks;
}


/**** Exported Functions **************/
GCFUNC BlkAllocHdr* blkAllocBlock( ExecEnv *ee )
{
  BlkAllocHdr *ph;
  sys_thread_t *self = EE2SysThread( ee );

  _LockBlkMgr( self );

  ph = (BlkAllocHdr *)_blkAllocRegion_locked( 1 );
  if (ph) {
    bhSet_status( ph, CHUNKING );
  }
  _UnlockBlkMgr( self );

  gcCheckGC();

  return ph;
}

GCEXPORT BlkAllocBigHdr* blkAllocRegion( unsigned nBytes, ExecEnv *ee )
{
  sys_thread_t *self = EE2SysThread( ee );

#ifdef RCDEBUG
  BlkAllocInternalHdr *inter;
  unsigned i;
#endif

  unsigned nBlocks;
  BlkAllocBigHdr *ph;
  BlkAllocBigHdr *lastBlk;

  nBlocks = _calcAllocSize( nBytes );

  _LockBlkMgr( self );
  ph = (BlkAllocBigHdr *)_blkAllocRegion_locked( nBlocks );

  if (!ph) {
    _UnlockBlkMgr( self );
    return NULL;
  }


  lastBlk = ph + (nBlocks-1);
  lastBlk->StatusUnused = ALLOCBIG << 24;
  lastBlk->blobSize = nBlocks;

  ph->allocInProgress = 1;
  ph->StatusUnused = ALLOCBIG << 24;
  ph->blobSize = nBlocks;

  _UnlockBlkMgr( self );

#ifdef RCDEBUG
  inter = (BlkAllocInternalHdr *)(ph+1);
  for (; inter < (BlkAllocInternalHdr *)lastBlk; inter++) {
    inter->startBlock = ph;
    bhSet_status( inter, INTERNALBIG );
  }
#endif

  gcCheckGC();

  return ph;
}

GCFUNC void blkFreeSomeChunkedBlocks( BlkAllocHdr **pph, int n )
{
  int i, status;
  BlkAllocHdr *ph;

  _LockBlkMgr( gcvar.sys_thread );

  for (i=0; i<n; i++) {
    ph = pph[i];
    status = bhGet_status(ph);
    mokAssert( status == DUMMYBLK );
    _blkFreeRegion_locked( (BlkRegionHdr*)ph, 1 );
  }

  _UnlockBlkMgr( gcvar.sys_thread );
}
```

```
GCFUNC void blkFreeChunkedBlock( BlkAllocHdr *ph )
{
#ifdef RCDEBUG
  int status = bhGet_status( ph );
  mokAssert ( status==VOIDBLK || status==PARTIAL );
#endif

  _LockBlkMgr( gcvar.sys_thread );
  _blkFreeRegion_locked( (BlkRegionHdr*)ph, 1 );
  _UnlockBlkMgr( gcvar.sys_thread );
}

GCFUNC void blkFreeRegion( BlkAllocBigHdr *ph )
{
  unsigned sz = ph->blobSize;

#ifdef RCDEBUG
  {
    BlkAllocBigHdr *lastBlk;
    BlkAllocInternalHdr *inter;
    unsigned i;

    lastBlk = ph + (sz-1);

    mokAssert( ph->StatusUnused = ALLOCBIG << 24 );
    mokAssert( lastBlk->StatusUnused = ALLOCBIG << 24 );
    mokAssert( lastBlk->blobSize == sz );
    mokAssert( ! ph->allocInProgress  );

    inter = (BlkAllocInternalHdr *)(ph+1);
    for (; inter < (BlkAllocInternalHdr *)lastBlk; inter++) {
      uint status = bhGet_status( inter );
      mokAssert( status == INTERNALBIG );
      mokAssert( inter->startBlock == ph );
    }
  }
#endif

  _LockBlkMgr( gcvar.sys_thread );
  _blkFreeRegion_locked( (BlkRegionHdr *)ph, sz );
  _UnlockBlkMgr( gcvar.sys_thread );

}

#ifdef RCDEBUG
GCFUNC void blkPrintStats(void)
{
  jio_printf("_____________ BLK STATS _____________\n" );
  jio_printf("wild=%d list=%d used=%d\n",
         blkvar.nWildernessBlocks, blkvar.nListsBlocks, blkvar.nAllocatedBlocks );
}
#endif

#pragma optimize( "", off )
GCFUNC void blkSweep(void)
{
  BlkRegionHdr *wildernessHdr = blkvar.wildernessRegion;
  BlkRegionHdr *brh = (BlkRegionHdr*)blkvar.allocatedBlockHeaders;
  volatile int *volatile p;

  while (brh < wildernessHdr) {
    volatile int size, status;

    p = (volatile int *volatile)&brh->regionSize;
    size = *p;
    p++;
    status = (*p) >> 24;

  __next_round:
    switch (status) {
    case BLK:
    case BLKLIST:
      mokAssert( size >= 1 );
      brh += size;
      break;

    case ALLOCBIG:
      _sweepBig( (BlkAllocBigHdr*)brh );
      mokAssert( size >= 1 );
      brh += size;
      break;

    case OWNED:
    case VOIDBLK:
    case PARTIAL: {
      int nextStatus;
      BlkRegionHdr *nextBrh = brh + 1;
      p = (volatile int *volatile)&nextBrh->regionSize;
      size = *p;
      p++;
      nextStatus = (*p) >> 24;
```

```
      chkSweepChunkedBlock( (BlkAllocHdr*)brh, status );
      brh = nextBrh;
      status = nextStatus;
      if (nextBrh >= wildernessHdr) return;
      goto __next_round;
    }

    default:
      mokAssert( status == CHUNKING );
      brh++;
      break;
    }
  }
}
#pragma optimize( "", on )
```

End of file source listing

D.4 rcchunkmgr.c

This file contains the code of the chunks manager (see section8.9 for more details).

Source listing for file rcchunkmgr.c

```
/*
 * File:    rcchunkmgr.c
 * Author:  Mr. Yossi Levanoni
 * Purpose: implementation of the chunk manager
 */
/***********************************************
*
* Lock a partial list.  Implemented by a spin
* lock which is imbedded in the list header.
*/
#define _lockPartialList(pList, ee)\
do {\
        mokAssert( ee );\
        gcSpinLockEnter( &pList->lock, (unsigned)ee );\
} while(0)

/***********************************************
*
* Unlock a partial list
*/
#define _unlockPartialList(pList, ee)\
do {\
        mokAssert( ee );\
        gcSpinLockExit( &pList->lock, (unsigned)(ee) );\
} while(0)


static void  _getPartialListStats( int iList,
                                   int *pFreeBlocks,
                                   int *pFreeBytes )
{
  ExecEnv *ee = EE();
  PARTIALLIST *pList = &chunkvar.partialLists[ iList ];
  int objSz  = chkconv.binSize[ iList ];
  int maxObj = chkconv.binToObjectsPerBlock[ iList ];
  int status, count;
  BlkAllocHdr *ph, *nextPh;
  BLKOBJ *freeList;

  *pFreeBlocks = 0;
  *pFreeBytes = 0;

  _lockPartialList( pList, ee);

  ph = pList->firstBlock;
  while (ph) {
    (*pFreeBlocks)++;
    status = bhGet_status( ph );
    mokAssert( status == PARTIAL );
    freeList = (BLKOBJ*)ph->freeList;
    if (freeList) {
      mokAssert( OBJBLOCKHDR(freeList) == ph );
      count = (int)freeList->count;
      mokAssert( count<=maxObj && count>0 );
      *pFreeBytes += count;
    }
    ph = ph->nextPartial;
  }
  _unlockPartialList( pList, ee );
  *pFreeBytes *= objSz;
}

GCEXPORT void chkGetPartialBlocksStats( int freeBlocks[], int freeBytes[])
```

```
{
  int i;
  for (i=0; i<N_BINS; i++)
    _getPartialListStats( i, &freeBlocks[i], &freeBytes[i] );
}


GCEXPORT int chkCountPartialBlocks(void)
{
  int n=0, i;
  for (i=0; i<N_BINS;i++)
    n += chunkvar.nBlocksInPartialList[i];
  return n;
}

/*************************************************************
************* Mutual Services ********************************
*************************************************************/

/*****************************************
*
* Initialize conversion tables.
*
*/
static void _initChunkConv( void )
{
  int target,i, j;

  i=0;

  chkconv.binSize[ i++ ] = 8;
  chkconv.binSize[ i++ ] = 16;
  chkconv.binSize[ i++ ] = 24;
  chkconv.binSize[ i++ ] = 32;
  chkconv.binSize[ i++ ] = 40;
  chkconv.binSize[ i++ ] = 48;
  chkconv.binSize[ i++ ] = 56;
  chkconv.binSize[ i++ ] = 64;
  chkconv.binSize[ i++ ] = 80;
  chkconv.binSize[ i++ ] = 96;
  chkconv.binSize[ i++ ] = 112;
  chkconv.binSize[ i++ ] = 128;
  chkconv.binSize[ i++ ] = 160;
  chkconv.binSize[ i++ ] = 192;
  chkconv.binSize[ i++ ] = 224;
  chkconv.binSize[ i++ ] = 256;
  chkconv.binSize[ i++ ] = 320;
  chkconv.binSize[ i++ ] = 384;
  chkconv.binSize[ i++ ] = 448;
  chkconv.binSize[ i++ ] = 512;
  chkconv.binSize[ i++ ] = 640;
  chkconv.binSize[ i++ ] = 768;
  chkconv.binSize[ i++ ] = 1024;
  chkconv.binSize[ i++ ] = 1280;
  chkconv.binSize[ i++ ] = 2048;
  chkconv.binSize[ i++ ] = 4096;
  chkconv.binSize[ i++ ] = 8192;

  mokAssert( i == N_BINS );

  j = 0;
  for (i=0; i<=N_BINS; i++) {
    target = chkconv.binSize[i];
    for (; j<=target; j++) {
      chkconv.szToBinIdx[ j ] = i;
      chkconv.szToBinSize[ j ] = target;
    }
  }

  for (i=0; i<N_BINS; i++) {
    chkconv.binToObjectsPerBlock[i] = BLOCKSIZE / chkconv.binSize[i];
#ifdef RCDEBUG
    chunkvar.nBlocksInPartialList[i] = 0;
#endif /* RCDEBUG */
  }
}


/*************************************************************
******************** COLLECTION *****************************
*************************************************************/


/************************************************
*
* Adds a block to a partial list.
*
* A block is added to the partial list by a
* collector when it finds that it's in the
* VOIDBLK state.
*
* The state is changed and the block is added to
```

```
* the appropriate list.
*
*
* Locks taken:
*       the partial list lock
*
* Competing operations:
*       mutators executing _getPartialBlock
*
* state changes:
*       VOIDBLK ---> PARTIAL.  No contention.
*/
static void _addPageToPartialList( BlkAllocHdr* ph )
{
  BlkAllocHdr *head;
  int idx = bhGet_bin_idx(ph);
  PARTIALLIST *pList = &chunkvar.partialLists[ idx ];

  mokAssert( bhGet_status(ph) == VOIDBLK );
  bhSet_status(ph, PARTIAL );

  _lockPartialList( pList, gcvar.ee );
  head = pList->firstBlock;
  ph->nextPartial = head;
  ph->prevPartial = (BlkAllocHdr*)pList;
  if (head)
    head->prevPartial = ph;
  pList->firstBlock = ph;
#ifdef RCDEBUG
  chunkvar.nBlocksInPartialList[ idx ] ++;
#endif /* RCDEBUG */
  _unlockPartialList( pList, gcvar.ee );
}


/***********************************************
*
* Flush the buffers that contains block headers
* which have observed to be full.
*
* Each partial list is locked and the buffer
* corresponding to it is examined.
*
* Each element has been already observed to be
* entirely free may have undergone many changes
* since:
*
* 1. It could have been reallocated and now
*     it is either OWNED or VOIDBLK.
*
* 2. If it turned into VOPIDBLK then the collector
*    could have already freed it.
*
* We protect against each of these possibilities
* by checking that the block is indeed full, and
* in the original partial list where it was observed.
*
* Additionally, we mark such a block as DUMMYBLK in
* order not to free it twice.
*
* When the candidates for freeing are verifired, the
* array of truly deletable blocks is passed to the
* block manager.
*
* Locks taken:
*     1. the partial list lock.  Each at a time.
*     2. Afterwards, the block manager lock.
*
* Competing operations:
*     mutators executing _getPartialBlock.
*
* State changes:
*     PARTIAL ---> Block Mgr states.  Contention resolved
*     by block mgr lock.
*/
static void _flushObservedFull(void)
{
  int  listIdx, status, count, maxObj, currentListIdx;
  int   blockIdx;
  PARTIALLIST *pList;
  BlkAllocHdr *ph;

  chunkvar.nTrulyFull = 0;


  for (listIdx = 0; listIdx<N_BINS; listIdx++) {
    pList = &chunkvar.partialLists[ listIdx ];
    maxObj = chkconv.binToObjectsPerBlock[listIdx] ;

    _lockPartialList( pList, gcvar.ee);

    for (blockIdx=0; blockIdx<pList->nObservedFull; blockIdx++) {
```

```
      ph = pList->observedFull[ blockIdx ];

      /* Did some mutator took it ? */
      status = bhGet_status(ph);
      if (status != PARTIAL) { /* yep */
        continue;
      }

      /*
       * Is it in the original partial list
       * where it was observed to be full ?
       */
      currentListIdx = bhGet_bin_idx(ph);
      if (currentListIdx != listIdx ) /* nop */
        continue;

      /**
       * Is it still fully free ?
       */
      if (!ph->freeList) /* nop */
        continue;

      count = ph->freeList->count;

      mokAssert( count>=0 && count<=maxObj );

      if (count < maxObj) /* nop */
        continue;

      /*
       * Protect against extracting a single block
       * mutiple times.
       */
      bhSet_status( ph, DUMMYBLK );

      /* extract the page */
      ph->prevPartial->nextPartial = ph->nextPartial;
      if (ph->nextPartial)
        ph->nextPartial->prevPartial = ph->prevPartial;
#ifdef RCDEBUG
      chunkvar.nBlocksInPartialList[ listIdx ] --;
#endif /* RCDEBUG */

      chunkvar.trulyFull[ chunkvar.nTrulyFull++ ] = ph;
    }

    _unlockPartialList( pList, gcvar.ee );

    pList->nObservedFull = 0; /* reset the list specific counter */
  }
  /* reset global counter */
  chunkvar.nObservedFull = 0;

  /* return blocks to the block manager */
  blkFreeSomeChunkedBlocks( chunkvar.trulyFull, chunkvar.nTrulyFull );
}

/*************************************************************************
*
* Take a note that a block has been observed to be fully free.
*
* For each partial list we keep a buffer and a counter of blocks that
* were observed as full.  Additonally, we keep a global counter of
* all the blocks in all the partial lists that were observed to be full.
*
* If either the list specific counter or the global counter crosses a
* threshold, the lists are flushed using _flushObservedFull()
*
*
* Locks taken:
*       the call to _flushObservedFull() may lock partial lists and/or
*       the block manager (one at a time).
*/
static void _handleFullPartialBlock( PARTIALLIST *pList, BlkAllocHdr* ph )
{
  pList->observedFull[ pList->nObservedFull++ ] = ph;
  chunkvar.nObservedFull++;
  if (pList->nObservedFull >= MAX_OBSERVED_FULL_PER_LIST ||
      chunkvar.nObservedFull >= MAX_OBSERVED_FULL)
    _flushObservedFull();
}




/************************************************
**************  Allocation **********************
************************************************/
```

```
/************************************************
*
* Moves all the items in a page's free list into
* the allocation list passed as a parameter.
*
* This function is called by a mutator which is
* the owner of this block.  It is invoked for
* a page which has just been extracted from a
* partial list so it's clear that the free
* list is non-empty.
*
* Locks taken:
*    The page's lock
*
* Competing operations:
*    _flushRecycledListEntry().  Contention is
*    resolved by the page's lock.
*/
static void _stealFreeList( ALLOCLIST *allocList )
{
  BlkAllocHdr *ph = allocList->allocBlock;
  BLKOBJ *prev, *head;

  mokAssert( allocList->binIdx == bhGet_bin_idx( ph ) );
  mokAssert( bhGet_status(ph) == OWNED );

  bhLock( ph );
  (volatile BLKOBJ*)prev = ph->freeList;
  ph->freeList = NULL;
  bhUnlock(ph);

  mokAssert( prev );

  head = prev->next;

  prev->next = ALLOC_LIST_NULL;

  allocList->head = head;
}

/************************************************
*
* Tries extracting a block from a partial list.
*
* If the partial list corresponding to the allocation
* list is non-empty then the first element is extracted.
*
* While the partial list lock is held, the state of the
* block is changed to OWNED.  This protects against
* freeing the block by the collector back to the block
* manager.
*
* The partial list lock is then released.
*
* Then the blocks free list is stolen (i.e., moved onto the
* allocation list) which entails locking the block.
*/
static BOOL _getPartialBlock( ALLOCLIST *allocList, ExecEnv *ee )
{
#ifdef RCDEBUG
  static int deltaMax = -1;
  int delta = GetTickCount();
#endif

  BlkAllocHdr *ph;
  PARTIALLIST *pList = &chunkvar.partialLists[ allocList->binIdx ];

  _lockPartialList( pList, ee );
  ph = pList->firstBlock;
  if ( !ph ) {
    _unlockPartialList( pList, ee );
#ifdef RCDEBUG
    delta = GetTickCount() - delta;
    if (delta > deltaMax) {
      deltaMax = delta;
      jio_printf(" ***1 ALLOC_PARTIAL delta=%d\n", delta );
      fflush( stdout );
    }
#endif
    return FALSE;
  }
  else {
    BlkAllocHdr *next = ph->nextPartial;
    pList->firstBlock = next;
    if (next)
      next->prevPartial = (BlkAllocHdr*)pList;
  }
  bhSet_status( ph, OWNED );
#ifdef RCDEBUG
  chunkvar.nBlocksInPartialList[ allocList->binIdx ] --;
#endif /* RCDEBUG */
  _unlockPartialList( pList, ee );
```

```
  allocList->allocBlock = ph;

  _stealFreeList(allocList);

  mokAssert( allocList->head );
  mokAssert( allocList->head->count );

#ifdef RCDEBUG
  delta = GetTickCount() - delta;
  if (delta > deltaMax) {
    deltaMax = delta;
    jio_printf(" ***2 ALLOC_PARTIAL delta=%d\n", delta );
    fflush( stdout );
  }
#endif
  return TRUE;
}


/*********************************************************
*
* Tries allocating object from the allocation list or from
* the block which is currently owned by it.
*
* If the allocation list is non-empty, then the first element
* is extracted and returned (no locking required).
*
* Otherwise, if the allocation list has no allocation block
* associated with it, then the function fails.
*
* Othetwise, the page is locked and its free list is probed.
* If the free list is empty then the page is transformed into
* a VOIDBLK block, the block is disassociated with the
* allocation list and the fucntion fails.
*
* Otherwise, the free list is stolen and merged into the
* allocation list.  The first element is extracted and
* returned.
*/
static BLKOBJ *_allocFromOwnedBlock( ALLOCLIST* allocList )
{
  BLKOBJ *head = allocList->head;

  if (head != ALLOC_LIST_NULL) {

#ifdef RCDEBUG
    {
      BLKOBJ *firstObj;

      mokAssert( allocList->allocBlock );
      mokAssert( bhGet_status( allocList->allocBlock) == OWNED );
      firstObj = BLOCKHDROBJ(allocList->allocBlock);
      if ((char*)firstObj <  blkvar.heapStart ||
          (char*)firstObj >= blkvar.heapTop ||
          (char*)head < blkvar.heapStart ||
          (char*)head >= blkvar.heapTop ) {
        jio_printf(
                   "Blk=%x first=%x head=%x\n",
                   allocList->allocBlock,
                   firstObj,
                   head );
        fflush( stdout );
        mokAssert( 0 );
      }
      mokAssert( (((word)head) & ((word)firstObj)) == ((word)firstObj));
      if (allocList->head)
        mokAssert(
                  ((((int)allocList->head) -
                    ((int)head)) % chkconv.binSize[ allocList->binIdx ]) == 0  );
    }
#endif
    allocList->head = head->next;
    return head;
  }

  {
#ifdef RCDEBUG
    static int deltaMax = -1;
    int delta = GetTickCount();
#endif

    BlkAllocHdr *ph = allocList->allocBlock;
    if (!ph) return NULL;

    /* see if there is something on the free list */
    bhLock( ph );
    (volatile BLKOBJ*)head = ph->freeList;
    if (head) {
      /* copy and clear */
      ph->freeList = NULL;
      bhUnlock(ph);
```

```
      {
        BLKOBJ *ret = head->next;

        head->next = ALLOC_LIST_NULL;
        allocList->head = ret->next;
        return ret;
      }
    }

    /* OK, we have to abandon the page, i.e.,
     * transfrom it into a VOIDPG page
     */
    bhSet_status(ph, VOIDBLK );
    bhUnlock( ph );
    allocList->allocBlock = NULL;

#ifdef RCDEBUG
    delta = GetTickCount() - delta;
    if (delta > deltaMax) {
      deltaMax = delta;
      jio_printf(" ***3 ALLOC_OWNED delta=%d\n", delta );
      fflush( stdout );
    }
#endif
  }
  return NULL;
}

/*******************************************************
*
* Allocate a single block from the block manager and
* chunk it into the given allocation list.
*/
static bool _getBlkMgrBlock( ALLOCLIST* allocList, ExecEnv *ee )
{
#ifdef RCDEBUG
  static int deltaMax = -1;
  int delta = GetTickCount();
#endif

  BlkAllocHdr *ph = blkAllocBlock( ee );
  int sz;
  int count;
  BLKOBJ *start, *curr, *next;

  if (!ph) {
#ifdef RCDEBUG
    delta = GetTickCount() - delta;
    if (delta > deltaMax) {
      deltaMax = delta;
      jio_printf(" ***4 ALLOC_BLK delta=%d\n", delta );
      fflush( stdout );
    }
#endif
    return false;
  }

  sz = chkconv.binSize[ allocList->binIdx ];
  count = chkconv.binToObjectsPerBlock[ allocList->binIdx ];

  mokAssert( count >= 2 );

  count--;

  start = curr = BLOCKHDROBJ(ph);

  for ( ;count>0; count--) {
    next = (BLKOBJ*)(((word)curr) + sz );
    curr->next = next;
    curr = next;
  }
  curr->next = ALLOC_LIST_NULL;

  allocList->head = start;
  allocList->allocBlock = ph;
  ph->nextPartial = ph->prevPartial = NULL;
  ph->freeList = NULL;
  ph->StatusLockBinidx = (OWNED << 24) | allocList->binIdx;

#ifdef RCDEBUG
  delta = GetTickCount() - delta;
  if (delta > deltaMax) {
    deltaMax = delta;
    jio_printf(" ***5 ALLOC_BLK delta=%d\n", delta );
    fflush( stdout );
  }
#endif
  return true;
}
```

```
/*********************************************************************
*********************************************************************
******************  Exported Functions  *****************************
*********************************************************************
*********************************************************************/

/********************** Collection ********************************/

#ifdef RCDEBUG
GCFUNC void chkPreCollect(BLKOBJ *o)
{
  word blockid;
  RLCENTRY *rlce;
  BLKOBJ *head;

  blockid = OBJBLOCKID(o);
  rlce = &chunkvar.rlCache[blockid % chunkvar.nCacheEntries];
  head = rlce->recycledList;

  /**
   * Is the cache entry currently owned by this block ?
   */
  if ((((word)head) ^ ((word)o)) < BLOCKSIZE) {

    mokAssert( OBJBLOCKID(head)==blockid );
    {
      int binIdx = bhGet_bin_idx( OBJBLOCKHDR(o) );
      int objSize = chkconv.binSize[ binIdx ];
      int maxObjs = chkconv.binToObjectsPerBlock[ binIdx ];

      /**
       * since some but not all BLKBOJs of the block are linked
       * the following should hold.
       */
      mokAssert( head->count>0 && head->count<maxObjs );
    }
    o->next  = head->next;
    head->next = o;
    head->count ++;
    return;
  }

  if (head)
    chkFlushRecycledListEntry( rlce );

  /* now the entry is vacant and we can use it */
  o->count = 1;
  o->next  = o;
  rlce->recycledList = o;
}
#endif /* RCDEBUG */

/*********************************************************************
*
* Flush an entry in the recycled lists cache.
*
*
* First of, the block is locked then its state is read, the free list
* is merged with the recycled list and then the lock is released.
*
* -- If the block is in the VOIDBLK state:
*
* a. The free list must be empty.
* b. If the free list now contains all elements in the block then the
*    block is returned directly to the block manager (without going
*    through the "observed full" set).  Otherwise, the state is changed
*    to PARTIAL  (no lock is taken).  Then the corresponding partial list
*    is locked and the block is added to it.
*
* -- Additional action for PARTIAL
* a. If the block is now fully freed, then it is marked as "observed full"
*    which may lead to the flushing of the "observed full" set.
*
* Note: free lists and recycled lists are circular.
*
*/
GCFUNC void chkFlushRecycledListEntry(RLCENTRY *rlce)
{
  BlkAllocHdr *ph;
  int nFree, nRecycled;
  BLKOBJ *recycledList, *freeList;
  unsigned status;

  recycledList = rlce->recycledList;
  ph = OBJBLOCKHDR( recycledList );

  mokAssert( recycledList ); /* or else it woudn't be in the cache */
  mokAssert( recycledList->next ); /* it's a circular list */

  nRecycled = recycledList->count;

  mokAssert( nRecycled ); /* or else it woudn't be in the cache */
```

```
  bhLock( ph );

  status = bhGet_status(ph);
  mokAssert( status==PARTIAL || status==OWNED || status==VOIDBLK);

  (volatile BLKOBJ*)freeList = ph->freeList;

  if (freeList) {
    BLKOBJ *t;
    mokAssert( freeList->count );
    nFree = freeList->count + recycledList->count;
    t = recycledList->next;
    recycledList->next = freeList->next;
    freeList->next = t;
  }
  else {
    nFree = recycledList->count;
    freeList = recycledList;
  }

  freeList->count = nFree;
  ph->freeList = freeList;

  bhUnlock( ph );

  if (status == PARTIAL) {
    /*
     * Have we freed all chunks on a
     * partial page ?
     */
    int binIdx = bhGet_bin_idx( ph );
    PARTIALLIST *pList = &chunkvar.partialLists[ binIdx ];
    int maxChunks = chkconv.binToObjectsPerBlock[ binIdx ];
    if (maxChunks == nFree)
      _handleFullPartialBlock( pList, ph );
  }
  else if (status == VOIDBLK) {
    /**
     * either put the VOIDBLK page into the partial list or
     * return it to the block manager.
     */
    int binIdx = bhGet_bin_idx( ph );
    int maxChunks = chkconv.binToObjectsPerBlock[ binIdx ];
    if (maxChunks==nFree) {
      blkFreeChunkedBlock(ph);
    }
    else {
      _addPageToPartialList(ph);
    }
  }
  rlce->recycledList = NULL;
}

GCFUNC void chkFlushRecycledListsCache( void )
{
  int i;
  RLCENTRY *rlce = chunkvar.rlCache;
  for (i=chunkvar.nCacheEntries; i>0; i--, rlce++)
    if (rlce->recycledList)
      chkFlushRecycledListEntry( rlce );
}

GCFUNC void chkSweepChunkedBlock( BlkAllocHdr *ph, int status)
{
  int binidx = bhGet_bin_idx( ph );
  int objsz = chkconv.binSize[ binidx ];
  int nobj = chkconv.binToObjectsPerBlock[ binidx ];
  GCHandle *h = (GCHandle*)BLOCKHDROBJ(ph);
  RLCENTRY rlce;
  int count = 0;

  while (nobj>0) {
    nobj--;
    if (gcGetHandleRC(h)==0 && !h->logPos) {
      BLKOBJ *o = (BLKOBJ*)h;
      o->next = o;
      rlce.recycledList = o;
      count = 1;
      goto __scan_with_list;
    }
    h = (GCHandle*)(objsz + (char*)h);
  }

  return; /* found nothing */

 __scan_with_list:
  /* here recycled list is non-empty */

  h = (GCHandle*)(objsz + (char*)h);
  while (nobj>0) {
    nobj--;
```

```
    if (gcGetHandleRC(h)==0 && !h->logPos) {
      BLKOBJ *o = (BLKOBJ*)h;
      count++;
      o->next = rlce.recycledList->next;
      rlce.recycledList->next = o;
      goto __scan_with_list;
    }
    h = (GCHandle*)(objsz + (char*)h);
  }

#ifdef RCDEBUG
  gcvar.dbg.nFreedInCycle += count;
  gcvar.dbg.nBytesFreedInCycle += count*objsz;
#endif
  rlce.recycledList->count = count;
  chkFlushRecycledListEntry( &rlce );
}

/****************** Allocation *******************************/

GCEXPORT BLKOBJ *chkAllocSmall(ExecEnv* ee, unsigned binIdx )
{
  int retries;
  ALLOCLIST *allocList = & ee->gcblk.allocLists[ binIdx ];
  BLKOBJ* ores;

  ores = _allocFromOwnedBlock( allocList );
  if (ores) {
    return ores;
  }

  /* now is a good time to cooperate ! */
  //  if (ee->gcblk.stage != gcvar.stage)
  // gcThreadCooperate(ee);

  for (retries=0; retries<3; retries++) {
    if (_getPartialBlock( allocList, ee )) {
      ores = _allocFromOwnedBlock( allocList );
      mokAssert( ores );
      return ores;
    }
    if (_getBlkMgrBlock( allocList, ee )) {
      ores = _allocFromOwnedBlock( allocList );
      mokAssert( ores );
      return ores;
    }
    /* Sync GC */
    if (gcvar.initialized) {
      gcvar.memStress = true;
      gcRequestSyncGC();
    }
    else
      break;
  }
  OutOfMemory();
  return NULL;
}

/****************** Initialization *******************************/
GCFUNC void chkInit(unsigned nMB)
{
  unsigned      sz;
  unsigned      nPages;

  /* init conversion tables */
  _initChunkConv();

  /* Allocate page headers cache, ZEROED OUT */
  nPages = nMB << (20 - BLOCKBITS);
  chunkvar.nCacheEntries = nPages / RLCACHE_RATIO;
  if (chunkvar.nCacheEntries < 117)
    chunkvar.nCacheEntries = 117;
  sz = chunkvar.nCacheEntries * sizeof(RLCENTRY);
  chunkvar.rlCache = (RLCENTRY*)mokMemReserve( NULL, sz );
  mokMemCommit( chunkvar.rlCache, sz, true );
}
```

End of file source listing

D.5 rcgc.c

rcgc.c contains the code for the reference counting and tracing garbage collection algorithms.

Source listing for file rcgc.c

```
/*
 * File:    rcgc.c
```

```
 * Author:  Mr. Yossi Levanoni
 * Purpose: implementation of the garbage collector
 */

/* forward declarations */
static void _snoopThreadLocals( sys_thread_t* t );
static void _incrementHandleRC( void * h);
static void _traceSetup(void);
static void _freeHandle(GCHandle* h);

/************** Debug Prints ********************/
static FILE *fDbg;


#ifdef RCDEBUG
static void dbgprn(int level, char *fmt, ...)
{
  char buff[1000];
  if (level <= 2) {
    va_list args;

    va_start( args, fmt );
    if (fDbg==NULL)
      fDbg = fopen("test.txt", "wt" );
    vfprintf( fDbg, fmt, args );
    vsprintf( buff, fmt, args );
    jio_printf( "%s", buff );
    fflush( stdout );
    va_end( args );
  }
}
#endif


/************************************************************/
/* atomic op */


// int ___compare_and_swap(unsigned *addr, unsigned oldv, unsigned newv);

#pragma optimize( "", off )

GCFUNC void gcSpinLockEnter(volatile unsigned *p, unsigned id)
{
  int i;

  for(i=0; i<N_SPINS; i++) {
    if (*p) continue;
    if (___compare_and_swap((unsigned*)p, 0, id))
      // jio_printf("gcSpinLockEnter ended (1)\n");
      return;
  }
  i = 1;
  for (;;) {
    mokSleep( i/1000 );
    if (___compare_and_swap((unsigned*)p, 0, id)) {
      return;
    }
    i *= 2;
  }
}

GCFUNC void gcSpinLockExit(volatile unsigned *p, unsigned id)
{
#ifdef RCDEBUG
  bool res;
#endif
  mokAssert( *p == id );
#ifdef RCDEBUG
  res =  ___compare_and_swap((unsigned*)p, id, 0);
  mokAssert( res );
#else
  ___compare_and_swap((unsigned*)p, id, 0);
#endif /* RCDEBUG */
}

#pragma optimize( "", on )

/***************  BUFFER MANAGEMENT ***********************/

static uint* buffList = NULL;
static uint  pad_against_false_sharing1[256];
static uint  buffListLock;
static uint  pad_against_false_sharing2[256];

void _buffListLockEnter(uint ee)
{
  gcSpinLockEnter( &buffListLock, (unsigned)ee );
}

void _buffListLockExit(uint ee)
{
```

```
  gcSpinLockExit( &buffListLock, (unsigned)ee );
}

static uint* _allocFreshBuff(void)
{
  uint *bf;

  bf = (uint*)mokMemReserve( NULL, BUFFSIZE );
  mokMemCommit( bf, BUFFSIZE, false );
  if (!bf) {
    jio_printf("YLRC: out of log buffers space\n");
    fflush( stdout );
    exit(-1);
  }
#ifdef RCDEBUG
  bf[USED_IDX] = Im_used;
#endif
  return bf;
}

static uint* _allocBuff(ExecEnv *ee)
{
  uint *bf;

  if (buffList==NULL) {
    bf = _allocFreshBuff();
    _buffListLockEnter( (unsigned)ee );
    gcvar.nAllocatedChunks++;
    gcvar.nUsedChunks++;
    mokAssert( gcvar.nFreeChunks+gcvar.nUsedChunks == gcvar.nAllocatedChunks );
    _buffListLockExit( (unsigned)ee );
    goto checkout;
  }
  _buffListLockEnter( (unsigned)ee );
  bf = buffList;
  if (!bf) {
    gcvar.nUsedChunks++;
    gcvar.nAllocatedChunks++;
    mokAssert( gcvar.nFreeChunks+gcvar.nUsedChunks == gcvar.nAllocatedChunks );
    _buffListLockExit( (unsigned)ee );
    bf = _allocFreshBuff();
  }
  else {
    gcvar.nUsedChunks++;
    gcvar.nFreeChunks--;
    mokAssert( gcvar.nFreeChunks+gcvar.nUsedChunks == gcvar.nAllocatedChunks );
#ifdef RCDEBUG
    mokAssert( bf[USED_IDX] == Im_free );
    bf[USED_IDX] = Im_used;
#endif // RCDEBUG
    buffList = (unsigned*)bf[LINKED_LIST_IDX];
    _buffListLockExit( (unsigned)ee );
  }
 checkout:
  if (ee != gcvar.ee) {
    gcvar.nChunksAllocatedRecentlyByUser++; // allow inaccuracy due to race condition
    if (gcvar.nChunksAllocatedRecentlyByUser >= gcvar.opt.userBuffTrig
        && gcvar.initialized
        && !gcvar.gcActive) {
#ifdef RCVERBOSE
      jio_printf("ALLOC BUFF used=%d TRIGERRING ASYNC RC\n", gcvar.nUsedChunks );
      fflush( stdout );
#endif
      gcRequestAsyncGC( );
    }
  }
  return bf;
}

static void _freeBuff( ExecEnv *ee, uint* buff)
{
  mokAssert( ee == gcvar.ee );

#ifdef RCDEBUG
  mokAssert( buff[USED_IDX] == Im_used );
#endif

  _buffListLockEnter( (unsigned)ee );
  buff[LINKED_LIST_IDX] = (uint)buffList;
  buffList = buff;
  gcvar.nFreeChunks++;
  gcvar.nUsedChunks--;
  mokAssert( gcvar.nFreeChunks+gcvar.nUsedChunks == gcvar.nAllocatedChunks );
#ifdef RCDEBUG
  buff[USED_IDX] = Im_free;
#endif // RCDEBUG
  _buffListLockExit( (unsigned)ee );
}

static void _initBuffReservedSlots( ExecEnv* ee, uint *newbuff )
{
  newbuff[LINKED_LIST_IDX]           = 0;
```

```
  newbuff[REINFORCE_LINKED_LIST_IDX] = 0;
  newbuff[NEXT_BUFF_IDX]             = 0;
  newbuff[LAST_POS_IDX]              = 0;
#ifdef RCDEBUG
  newbuff[ALLOCATING_EE]             = (uint)ee;
  newbuff[LOG_CHILDS_IDX]            = 0;
  newbuff[LOG_OBJECTS_IDX]           = 0;
  newbuff[USED_IDX]                  = Im_used;
#endif
}

GCEXPORT void gcBuffAllocAndLink(ExecEnv* ee, BUFFHDR *bh)
{
  uint i;
  uint *newBuff = _allocBuff( ee );

  _initBuffReservedSlots( ee, newBuff );

  /* backword link */
  newBuff[N_RESERVED_SLOTS] = ((uint)bh->pos) | BUFF_LINK_MARK;

  /* forward link */
  /* from the current position to the new chunk */
  *bh->pos = ((uint)&newBuff[N_RESERVED_SLOTS]) | BUFF_LINK_MARK;
  /* from the beginning of the current buffer to the next buffer */
  bh->currBuff[NEXT_BUFF_IDX] = (uint)newBuff;


  /* update record */
  bh->pos = &newBuff[N_RESERVED_SLOTS+1];
  bh->limit = newBuff + BUFFSIZE/sizeof(uint);
  bh->currBuff = newBuff;

  /*
   * Reserve place for"
   * 1. the handle and forward pointer (2 words).
   * 2. and a reserved place for a snooped object.
   */
  bh->limit -= 3;
}

static void buffInit(ExecEnv *ee, BUFFHDR *bh)
{
  int i;
  bh->start = _allocBuff(ee);

  _initBuffReservedSlots( ee, bh->start );

  /* backword link */
  bh->start[N_RESERVED_SLOTS] = ((unsigned)NULL) | BUFF_LINK_MARK;
  bh->pos = &bh->start[N_RESERVED_SLOTS+1];
  bh->limit = bh->start + BUFFSIZE/sizeof(uint);
  bh->limit -= 3; /* for the handle, forward pointer and reserved snoop */
  bh->currBuff = bh->start;
}

#define buffIsModified(bh) ((bh)->pos  != &(bh)->start[N_RESERVED_SLOTS+1])


#pragma optimize( "", off )
GCEXPORT void gcBuffSlowConditionalLogHandle(ExecEnv* ee, GCHandle *h)
{
  int avail;
  GCHandle **objslots;
  GCHandle **p;
  ClassClass *cb;
  BUFFHDR *bh;

#ifdef RCDEBUG
  uint nLoggedChilds = 0;
#endif // RCDEBUG

  bh = &ee->gcblk.updateBuffer;


  if (obj_flags(h)==T_NORMAL_OBJECT) {
    cb = obj_classblock(h);
    mokAssert( cb != classJavaLangClass);
    { /* OK, it's a non-class object */
      unsigned short *offs = cbObjectOffsets(cb);
      int nrefs = unhand(cb)->n_object_offsets;
      objslots = (GCHandle**)(((char*)unhand(h))-1);

      mokAssert( objslots && h && bh && ee && offs && nrefs>0);
      p = (GCHandle**)bh->pos;
      avail = bh->limit - (uint*)p;
      if (nrefs > avail) {
        ee->gcblk.cantCoop = false;
        gcBuffAllocAndLink( ee, bh );
        p = (GCHandle**)bh->pos;
#ifdef RCDEBUG
        avail = bh->limit - bh->pos;
```

```
        mokAssert( nrefs <= avail );
#endif /* RCDEBUG */
        ee->gcblk.cantCoop = true;
      }
      for (;;) {
        unsigned short slot = *offs;
        GCHandle *child;

        if (slot==0) break;

        child = *(GCHandle**)(slot + (char*)objslots);
        if (child) {
          *p = child;
          p++;
#ifdef RCDEBUG
          nLoggedChilds++; // increment counter of logged slots
          mokAssert( nrefs > 0 );
          nrefs--;
#endif // RCDEBUG
        }
        offs++;
      }
    }
  }
  else {
    register long n = obj_length(h);
    GCHandle **body = (GCHandle**)(((ArrayOfObject*)gcUnhand(h))->body);

    mokAssert( obj_flags(h) == T_CLASS);     /* an array of classes */
    mokAssert( n > 0 );

    p = (GCHandle**)bh->pos;
    avail = bh->limit - (uint*)p;
    if (n > avail) {
      ee->gcblk.cantCoop = false;
      gcBuffAllocAndLink( ee, bh );
      p = (GCHandle**)bh->pos;
#ifdef RCDEBUG
      avail = bh->limit - bh->pos;
      mokAssert( n <= avail );
#endif /* RCDEBUG */
      ee->gcblk.cantCoop = true;
    }
    while (--n >= 0) {
      GCHandle *child = *body;
      body++;
      if (child) {
        *p = child;
        p++;
#ifdef RCDEBUG
        nLoggedChilds++; // increment counter of logged slots
#endif // RCDEBUG
      }
    }
  }

  /* commit ? or discard ? */
  if (!h->logPos) { /* commit */
    *p = (GCHandle*)(BUFF_HANDLE_MARK | (unsigned)h);
    /*
     * actually the order of instructions here
     * should be reversed in order to enable
     * async reading of buffers.
     */
    h->logPos = (uint*)p;
    bh->pos = (unsigned*)(p+1);
#ifdef RCDEBUG
    // increment counters of logged slots
    bh->start[LOG_CHILDS_IDX] += nLoggedChilds;
    bh->start[LOG_OBJECTS_IDX] ++;
#endif // RCDEBUG
  }
}
#pragma optimize( "", on )


#ifdef RCNOINLINE

GCEXPORT void gcBuffConditionalLogHandle(ExecEnv* ee, GCHandle *h)
{
  if (!h->logPos)
    gcBuffSlowConditionalLogHandle( ee, h);
}


GCEXPORT void gcBuffLogWordUnchecked(ExecEnv *ee, BUFFHDR *bh, uint w)
{
  *bh->pos = w;
  bh->pos++;
#ifdef RCDEBUG
  // increment counter of logged objects
  bh->start[LOG_OBJECTS_IDX] ++;
```

```
#endif // RCDEBUG
}

GCEXPORT void gcBuffReserveWord(ExecEnv *ee, BUFFHDR *bh)
{
  mokAssert( bh && ee );
  if ( bh->pos >= bh->limit) {
    gcBuffAllocAndLink( ee, bh );
  }
}

GCEXPORT void gcBuffLogWord(ExecEnv *ee, BUFFHDR *bh, uint w)
{
  mokAssert( w && bh && ee );

  gcBuffReserveWord( ee, bh );
  gcBuffLogWordUnchecked( ee, bh, w );
}

GCEXPORT void gcBuffLogNewHandle(ExecEnv *ee, GCHandle *h)
{
  BUFFHDR *bh;

  mokAssert( ee );

  bh = &ee->gcblk.createBuffer;

  ee->gcblk.cantCoop = true;
  *bh->pos = (uint)h;
  h->logPos = bh->pos;
  bh->pos++;
#ifdef RCDEBUG
  // increment counter of logged objects
  bh->start[LOG_OBJECTS_IDX] ++;
#endif // RCDEBUG
  mokAssert( gcGetHandleRC(h)==0 );
  ee->gcblk.cantCoop = false;
  gcBuffReserveWord( ee, bh );

  mokAssert( gcNonNullValidHandle(h) );
}

#endif /* RCNOINLINE */



/******************** VALIDATION ****************************/

GCFUNC bool _isHandle(void *h)
{
  BlkAllocHdr *bah;
  int status;

  if ((byte*)(h) <blkvar.heapStart) return false;
  if ((byte*)(h) >= blkvar.heapTop) return false;
  if ((((unsigned)h) & OBJMASK) != (unsigned)h) return false;
  if ((byte*)unhand((JHandle*)h) != (byte*)gcUnhand((JHandle*)h)) return false;
#ifdef RCDEBUG
  if (((GCHandle*)h)->status != Im_used) return false;
#endif

  bah = OBJBLOCKHDR(h);
  status = bhGet_status( bah );

  if (status==ALLOCBIG) {
    if ( ((uint)h & BLOCKMASK) == 0)
      return true;
    return false;
  }

  if (status<OWNED || status>PARTIAL)
    return false;

#ifdef RCDEBUG
  {
    int bin_idx = bhGet_bin_idx( bah );
    mokAssert( (((uint)h &  BLOCKMASK) % chkconv.binSize[bin_idx]) == 0);
  }
#endif

  /* check if on same page or ALLOC_LIST terminator */
  if ((uint)((GCHandle*)h)->logPos == (uint)ALLOC_LIST_NULL) return false;
  if ( ((uint)h ^ (uint)((GCHandle*)h)->logPos) < BLOCKSIZE )
    return false;

#ifdef RCDEBUG
  {
    uint val;
    uint *pos = ((GCHandle*)h)->logPos;
    if (pos) {
      val = *pos;
      if ( (val & ~3) != (uint)h ) {
```

```
        /*
         * This is a problem only if we're the collctor,
         * this means that someone has garbaled the log, the
         * logPos pointer or both.
         *
         * If we're a mutator then this is not an error since
         * the log could have already been freed by the collector.
         */
        if (gcvar.ee == EE()) {
          mokAssert(0);
        }
      }
    }
  }
#endif

  return true;
}


/*****************************************************************/

/**************** ZCT + RC ***********************************/

#ifndef RCDEBUG
#define _putInNextZCT(h)\
do { \
  gcBuffLogWord( gcvar.ee, (&gcvar.nextZctBuff), (uint)h );\
} while(0)
#else
static void _putInNextZCT(void *h)
{
  gcBuffLogWord( gcvar.ee, (&gcvar.nextZctBuff), (uint)h );
  gcvar.dbgpersist.nPendInCycle++;
}
#endif


#define _markInZCT(h) H1BIT_Set( gcvar.zctBmp.entry, (unsigned)h )

#define _markNotInZCT(h)  H1BIT_Clear( gcvar.zctBmp.entry, (unsigned)h )

static bool _isInZCT(GCHandle *h)
{
  bool res;
  H1BIT_GetInlined( gcvar.zctBmp.entry, (unsigned)h, res );
  return res;
}


GCFUNC uint gcGetHandleRC( GCHandle *h)
{
  uint res;
  H2BIT_GetInlined( gcvar.rcBmp.entry, (unsigned)h, res  );
  return res;
}

static void _incrementHandleRC( void  * h)
{
  H2BIT_Inc( gcvar.rcBmp.entry, (unsigned)h );
}

static uint _incrementHandleRCWithReturnValue( void * h)
{
  uint res;
  H2BIT_IncRVInlined( gcvar.rcBmp.entry, (unsigned)h, res );
  return res;
}

static void _decrementHandleRCInUpdate( void * h)
{
  uint prevRC;
  H2BIT_DecInlined( gcvar.rcBmp.entry, (unsigned)h, prevRC );
  if (prevRC==1 && !_isInZCT(h)) {
    _markInZCT( h );
    gcBuffLogWord( gcvar.ee, &gcvar.zctBuff, (uint)h );
#ifdef RCDEBUG
    gcvar.dbg.nInZct++;
    gcvar.dbg.nUpdate2ZCT++;
#endif // RCDEBUG
  }
}


static void _enlargeZctStack(void)
{
  GCHandle **p;
  uint sz = ((char*)gcvar.zctStackTop)-((char*)gcvar.zctStack);

  mokAssert( gcvar.zctStackSp == gcvar.zctStackTop );
  p = (GCHandle**)mokMemReserve( gcvar.zctStack, sz );
  if (p) {
```

```
    mokAssert( p == gcvar.zctStack );
    mokMemCommit( p, sz, false );
    gcvar.zctStackTop = (GCHandle**)(sz + (char*)gcvar.zctStackTop);
  }
  else {
    uint newsz = sz*2;
    GCHandle **oldstack = gcvar.zctStack;
    gcvar.zctStack = (GCHandle**)mokMemReserve( NULL, newsz );
    gcvar.zctStackTop = (GCHandle**)(newsz + (char*)gcvar.zctStack);
    gcvar.zctStackSp = (GCHandle**)( sz + (char*)gcvar.zctStack );
    mokMemCommit( (char*)gcvar.zctStack, newsz, false );
    CopyMemory( gcvar.zctStack, oldstack, sz );
    mokMemDecommit( (char*)oldstack, sz );
    mokMemUnreserve( (char*)oldstack, sz );
  }
}


static void _decrementHandleRCInDeletion(void *child)
{
  uint prevRC;
  H2BIT_DecInlined( gcvar.rcBmp.entry, (unsigned)child, prevRC );
  mokAssert( !_isInZCT(child) );
  mokAssert( prevRC > 0 );
  if (prevRC==1) {
#ifdef RCDEBUG
    gcvar.dbg.nRecursiveDel++;
    _freeHandle( child );
#else
    if (gcvar.zctStackSp == gcvar.zctStackTop) {
      _enlargeZctStack();
    }
    *gcvar.zctStackSp++ = child;
#endif // RCDEBUG
  }
}

static void _putInMarkStack(void *h)
{
  if (gcvar.zctStackSp == gcvar.zctStackTop) {
    _enlargeZctStack();
  }
  *gcvar.zctStackSp++ = (GCHandle*)h;
}


static void _decrementLocalHandleRC(void *h)
{
  uint prevRC;
  H2BIT_DecInlined( gcvar.rcBmp.entry, (unsigned)h, prevRC );
  mokAssert( !_isInZCT(h) );
  mokAssert( prevRC > 0 );
  if (prevRC==1) {
    _markInZCT(h);
    _putInNextZCT( h );
  }
}

/*********************  Local Marks *************************/

static bool _isLocal(void *h)
{
  uint res;
  H1BIT_GetInlined( gcvar.localsBmp.entry, (unsigned)h, res );
  return res;
}

static void _setLocal(void *h)
{
  if (!_isLocal(h)) {
    H1BIT_Set( gcvar.localsBmp.entry, (unsigned)h);
    _incrementHandleRC(h);
    gcBuffLogWord( gcvar.ee, (&gcvar.uniqueLocalsBuff), (uint)h );
#ifdef RCDEBUG
    gcvar.dbg.nLocals++;
#endif
  }
}


static void _unsetLocal(void *h)
{
  /* This also resets the local mark of near by objects,
   * but we don't care since we're turning everybody
   * off.
   */
  H1BIT_ClearByte( gcvar.localsBmp.entry, (unsigned)h);
}


/******************** COLLECTION !!!!! **********************/
```

```
/****************** HS1 ***********************************/


static int _setSnoopFlagHelper(sys_thread_t * thrd, void *dummy)
{
  ExecEnv *ee = SysThread2EE( thrd );

  mokAssert( ee != gcvar.ee );
  ee->gcblk.snoop = true;
  return SYS_OK;
}

static int _HS1Helper(sys_thread_t *thrd, bool *allOK)
{
  ExecEnv *ee;

  ee = SysThread2EE( thrd );

  mokAssert( ee != gcvar.ee );

  if (ee->gcblk.stage == GCHS1) return SYS_OK;
  if (ee->gcblk.cantCoop) {
    *allOK = false;
    return SYS_OK;
  }

  while( gcvar.nPreAllocatedBuffers < 2) {
    buffInit( gcvar.ee, &gcvar.preAllocatedBuffers[gcvar.nPreAllocatedBuffers] );
    gcvar.nPreAllocatedBuffers++;
  }

  mokThreadSuspendForGC( thrd );
  mokAssert(ee->gcblk.stage==GCHS4);
  if (ee->gcblk.cantCoop) {
    mokThreadResumeForGC( thrd );
    *allOK = false;
    return SYS_OK;
  }
#ifdef RCDEBUG
  gcvar.dbg.nHS1Threads++;
  gcvar.dbg.nUpdateObjects += ee->gcblk.updateBuffer.start[LOG_OBJECTS_IDX];
  gcvar.dbg.nUpdateChilds += ee->gcblk.updateBuffer.start[LOG_CHILDS_IDX];
  gcvar.dbg.nCreateObjects += ee->gcblk.createBuffer.start[LOG_OBJECTS_IDX];
#endif // RCDEBUG

  /* now steal the buffers (if they were modified) */

  if (buffIsModified(&ee->gcblk.createBuffer)) {
    /* make sure that the last word in the buffer is NULL */
    *ee->gcblk.createBuffer.pos = 0;
    /* make sure the second entry in the buffer points to
     * the last entry
     */
    ee->gcblk.createBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.createBuffer.pos;
    /* the first entry is the linked list pointer */
    ee->gcblk.createBuffer.start[LINKED_LIST_IDX] = (uint)gcvar.createBuffList;
    gcvar.createBuffList = ee->gcblk.createBuffer.start;
    /* give the thread new buffers to play with */
    gcvar.nPreAllocatedBuffers--;
    ee->gcblk.createBuffer = gcvar.preAllocatedBuffers[gcvar.nPreAllocatedBuffers];
  }
#ifdef RCDEBUG
  else {
    mokAssert( ee->gcblk.dbg.nBytesAllocatedInCycle==0 );
    mokAssert( ee->gcblk.dbg.nRefsAllocatedInCycle==0 );
  }
#endif

  if (buffIsModified(&ee->gcblk.updateBuffer)) {
    /* do the same for the update buffer */
    *ee->gcblk.updateBuffer.pos = 0;
    ee->gcblk.updateBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.updateBuffer.pos;
    ee->gcblk.updateBuffer.start[LINKED_LIST_IDX] = (uint)gcvar.updateBuffList;
    gcvar.updateBuffList = ee->gcblk.updateBuffer.start;
    gcvar.nPreAllocatedBuffers--;
    ee->gcblk.updateBuffer = gcvar.preAllocatedBuffers[gcvar.nPreAllocatedBuffers];
  }

#ifdef RCDEBUG
  gcvar.dbg.nBytesAllocatedInCycle += ee->gcblk.dbg.nBytesAllocatedInCycle;
  gcvar.dbg.nRefsAllocatedInCycle += ee->gcblk.dbg.nRefsAllocatedInCycle;
  gcvar.dbg.nNewObjectUpdatesInCycle += ee->gcblk.dbg.nNewObjectUpdatesInCycle;
  gcvar.dbg.nOldObjectUpdatesInCycle += ee->gcblk.dbg.nOldObjectUpdatesInCycle;

  ee->gcblk.dbg.nBytesAllocatedInCycle = 0;
  ee->gcblk.dbg.nRefsAllocatedInCycle = 0;
  ee->gcblk.dbg.nNewObjectUpdatesInCycle = 0;
  ee->gcblk.dbg.nOldObjectUpdatesInCycle = 0;
#endif

  /* restart the thread */
```

```
  ee->gcblk.stage = GCHS1;
  mokThreadResumeForGC( thrd );
#if 0
  ee->gcblk.gcSuspended = true;
#endif
  return SYS_OK;
}

#pragma optimize( "", off )
static void _Initiate_Collection_Cycle(void)
{
  bool allOK;

  mokAssert( gcvar.stage == GCHS4);

  // if (gcvar.
  /* raise snoop flags */
  QUEUE_LOCK( gcvar.sys_thread );
  mokThreadEnumerateOver( _setSnoopFlagHelper, NULL );
  QUEUE_UNLOCK( gcvar.sys_thread );

#ifdef RCDEBUG
  memset( &gcvar.dbg, 0, sizeof(gcvar.dbg) );
  gcvar.dbg.nInZct = gcvar.dbgpersist.nPendInCycle;
  gcvar.dbgpersist.nPendInCycle =0;
#endif // RCDEBUG

  /* do first handshake */
  QUEUE_LOCK( gcvar.sys_thread );

  gcvar.stage = GCHS1;
  mokAssert( gcvar.createBuffList == NULL );
  mokAssert( gcvar.updateBuffList == NULL );

  gcvar.createBuffList = gcvar.deadThreadsCreateBuffList;
  gcvar.deadThreadsCreateBuffList = NULL;
  gcvar.updateBuffList = gcvar.deadThreadsUpdateBuffList;
  gcvar.deadThreadsUpdateBuffList = NULL;

#ifdef RCDEBUG
  gcvar.dbg.nUpdateObjects = gcvar.dbgpersist.nDeadUpdateObjects;
  gcvar.dbgpersist.nDeadUpdateObjects = 0;

  gcvar.dbg.nUpdateChilds = gcvar.dbgpersist.nDeadUpdateChilds;
  gcvar.dbgpersist.nDeadUpdateChilds = 0;

  gcvar.dbg.nCreateObjects = gcvar.dbgpersist.nDeadCreateObjects;
  gcvar.dbgpersist.nDeadCreateObjects = 0;
#endif

  for(;;) {
    allOK = true;
    mokThreadEnumerateOver( _HS1Helper, &allOK );
    if (allOK) break;
    mokSleep( 10 );
  }

  QUEUE_UNLOCK( gcvar.sys_thread );
}
#pragma optimize( "", on  )

/*********************  HS2 & HS3 *************************/

static void _clearFlagsInUpdateBuffer(uint *p)
{
  uint *ptr;
  uint type;
#ifdef RCDEBUG
  uint *first_entry = p+N_RESERVED_SLOTS;
#endif

  mokAssert( p );


  p = (uint*)p[LAST_POS_IDX];
  mokAssert( ! *p );
  p--;
  mokAssert( *p );

  for (;;) {
    type = *p & 3;
  next_entry:
    ptr = (uint*)(*p & ~3);
#ifdef RCDEBUG
    /*
     *the one and only entry which
     * is supposed to be NULL is the
     * last one.
     */
    if (p==first_entry)
      mokAssert( *p == BUFF_LINK_MARK );
    if (*p == BUFF_LINK_MARK)
```

```
      mokAssert(p == first_entry );
#endif
    switch (type) {

    case BUFF_DUP_HANDLE_MARK: {
#ifdef RCDEBUG
      gcvar.dbg.nActualCyclesBroken++;
      gcvar.dbg.nActualUpdateObjects++;
      /*
       * can happen becuase of deletion
       * cycle breaking.
       */
      dbgprn( 3, "\t\tclear:up:broken %x\n", ptr );
#endif
      for (;;) {
        p--;
        type = *p & 3;
        if (type) goto next_entry;
#ifdef RCDEBUG
        gcvar.dbg.nActualUpdateChilds++;
#endif
      }
    }

    case 0: {/* Logged slot entry */
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( gcNonNullValidHandle(h) );
      p--;
#ifdef RCDEBUG
      dbgprn( 4, "\t\tclear:up:slot %x\n", ptr );
      gcvar.dbg.nActualUpdateChilds++;
#endif
      break;
    }

    case BUFF_LINK_MARK: {
      if (!ptr) {
#ifdef RCDEBUG
        mokAssert(p==first_entry);
#endif
        return;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; // skip forward pointer
      break;
    }

    case BUFF_HANDLE_MARK: { /* Containing object entry */
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
#ifdef RCDEBUG
      dbgprn( 4, "\t\tclear:up:hand %x\n", ptr );
      /* is this entry cycle closing ?
       * we assume that the striking majority
       * of entries are, so we modify
       * only those which are duplicates.
       */
      gcvar.dbg.nActualUpdateObjects++;
#endif
      if (h->logPos == p) { /* yep */
        mokAssert( gcNonNullValidHandle(h) );
        h->logPos = NULL; /* clear dirty flag */
      } else {
        *p = BUFF_DUP_HANDLE_MARK | (uint)h;
#ifdef RCDEBUG
        gcvar.dbg.nUpdateDuplicates++;
#endif
      }
      p--;
      break;
    }
    }
  }
}

static void _clearFlagsInUpdateBufferList(void)
{
  uint *buffList = gcvar.updateBuffList;
  while (buffList) {
    _clearFlagsInUpdateBuffer( buffList );
    buffList = (uint*)buffList[LINKED_LIST_IDX];
  }
}


static void _clearFlagsInCreateBuffer(uint *p)
{
#ifdef RCDEBUG
  uint *last_entry = (uint*)p[LAST_POS_IDX];
#endif

  mokAssert( p );
```

```
  p += N_RESERVED_SLOTS;
  p++; /* skip the first back pointer */

  for (;;) {
    uint *ptr = (uint*)(*p & ~3);
    uint type = *p & 3;
    mokAssert( type != BUFF_HANDLE_MARK);
    mokAssert( type != BUFF_DUP_HANDLE_MARK);
#ifdef RCDEBUG
    /*
     * the one and only entry which
     * is supposed to be NULL is the
     * last one.
     */
    if (p==last_entry)
      mokAssert( *p == 0 );
    if (!*p)
      mokAssert(p == last_entry );
#endif
    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
#ifdef RCDEBUG
      dbgprn( 4, "\t\tclear:cr: %x\n", ptr );
#endif
      if (!h) return;
      mokAssert( gcValidHandle(h)  );
      /* In the create buffer all entries
       * are cycle closing since there is
       * no contention for these objects.
       */
      mokAssert( h->logPos == p );
      h->logPos = NULL; /* clear dirty mark */
#ifdef RCDEBUG
      gcvar.dbg.nActualCreateObjects++;
#endif
      p++;
    }
    else { /*type==BUFF_LINK_MARK*/
      mokAssert( ptr );
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      mokAssert( (LOWBUFFMASK & (uint)ptr) == N_RESERVED_SLOTS*sizeof(uint));
      p = ptr+1;
    }
  }
}



static void _clearFlagsInCreateBufferList( void )
{
  uint *buffList = gcvar.createBuffList;
  while (buffList) {
    _clearFlagsInCreateBuffer( buffList );
    buffList = (uint*)buffList[LINKED_LIST_IDX];
  }
}


static void _Clear_Dirty_Marks(void)
{
#ifdef RCDEBUG
  DWORD start, end;

  start = GetTickCount();
  dbgprn( 0, "_Clear_Dirty_Marks(begin) time=%d\n", start);
#endif

  _clearFlagsInCreateBufferList(  );
  _clearFlagsInUpdateBufferList(  );

#ifdef RCDEBUG
  end = GetTickCount();

  dbgprn( 2, "\tnHS1Threads=%d\n", gcvar.dbg.nHS1Threads );
  dbgprn( 2, "\tnUpdateObjects=%d\n",  gcvar.dbg.nUpdateObjects );
  dbgprn( 2, "\tnUpdatdChilds=%d\n", gcvar.dbg.nUpdateChilds );
  dbgprn( 2, "\tnActualUpdateObjects=%d\n", gcvar.dbg.nActualUpdateObjects );
  dbgprn( 2, "\tnActualUpdateChilds=%d\n", gcvar.dbg.nActualUpdateChilds );
  dbgprn( 2, "\tnFreeCyclesBroken=%d\n", gcvar.dbgpersist.nFreeCyclesBroken );
  dbgprn( 2, "\tnCreateObjects=%d\n",  gcvar.dbg.nCreateObjects );
  dbgprn( 2, "\tnActualCreateObjects=%d\n",  gcvar.dbg.nActualCreateObjects );

  if (gcvar.dbg.nUpdateDuplicates) {
    dbgprn( 1, "\tnUpdateDuplicates=%d\n", gcvar.dbg.nUpdateDuplicates );
  }

  if (gcvar.dbgpersist.nFreeCyclesBroken) {
    dbgprn( 1, "\tnFreeCyclesBroken=%d\n", gcvar.dbgpersist.nFreeCyclesBroken );
  }

  mokAssert( gcvar.dbg.nActualUpdateObjects == gcvar.dbg.nUpdateObjects );
```

```
  mokAssert( gcvar.dbg.nActualUpdateChilds == gcvar.dbg.nUpdateChilds );
  mokAssert( gcvar.dbg.nActualCyclesBroken == gcvar.dbgpersist.nFreeCyclesBroken );
  mokAssert( gcvar.dbg.nActualCreateObjects == gcvar.dbg.nCreateObjects );

  gcvar.dbgpersist.nFreeCyclesBroken = 0;
  dbgprn( 0, "_Clear_Dirty_Marks(end) time=%d delta=%d\n", end, end-start );
#endif // RCDEBUG
}


static int _HS2Helper(sys_thread_t *thrd, bool *allOK)
{
  ExecEnv *ee;

  ee = SysThread2EE( thrd );

  mokAssert( gcvar.ee != ee );

  if (ee->gcblk.stage == GCHS2) return SYS_OK;
  if (ee->gcblk.cantCoop) {
    *allOK = false;
    return SYS_OK;
  }

  mokThreadSuspendForGC( thrd );
  mokAssert( ee->gcblk.stage == GCHS1 );
  if (ee->gcblk.cantCoop) {
    mokThreadResumeForGC( thrd );
    *allOK = false;
    return SYS_OK;
  }

  /* mark current position in the buffer */
  ee->gcblk.updateBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.updateBuffer.pos;
  /*
   * link the buffer into the reinforce buff
   * list.  Note that the buffer stays at the
   * mutator.
   *
   * We link the buffers instead of going again
   * through the thread ring in order not to
   * lock it when we really do the reinforce
   * stage.
   */
  ee->gcblk.updateBuffer.start[REINFORCE_LINKED_LIST_IDX] =
    (uint)gcvar.reinforceBuffList;
  gcvar.reinforceBuffList = ee->gcblk.updateBuffer.start;

#ifdef RCDEBUG
  {
    uint *pos = ee->gcblk.updateBuffer.pos;
    /*
     * i.e., we never point to the reserved area:
     */
    mokAssert( (((uint)pos)&LOWBUFFMASK) >= N_RESERVED_SLOTS );
    /*
     * If there is something in the current chunk, then
     * the last entry is a containing handle entry.
     * i.e., we don't see partial entries.
     */
    mokAssert( (((uint)pos)&LOWBUFFMASK) >= (N_RESERVED_SLOTS+1)*sizeof(int) );
    if ( (((uint)pos)&LOWBUFFMASK) > (N_RESERVED_SLOTS+1)*sizeof(int)) {
      mokAssert( (((uint)*(pos-1))&3) == BUFF_HANDLE_MARK );
    }
    /*
     * Otherwise, this should be a back-pointer to the
     * previous chunk.
     */
    else {
      mokAssert( (((uint)*(pos-1))) == BUFF_LINK_MARK );
    }
    gcvar.dbg.nHS2Threads++;
    gcvar.dbg.nReinforceObjects += ee->gcblk.updateBuffer.start[LOG_OBJECTS_IDX];
    gcvar.dbg.nReinforceChilds += ee->gcblk.updateBuffer.start[LOG_CHILDS_IDX];
  }
#endif /* RCDEBUG */

  /* restart the thread */
  ee->gcblk.stage = GCHS2;
  mokThreadResumeForGC( thrd );
  return SYS_OK;
}

static void _reinforceUpdateBuffer( uint *p, uint *limit )
{
  mokAssert( p );

  p += N_RESERVED_SLOTS;
  p++; /* skip the first back pointer */

  for (;;) {
    uint *ptr = (uint*)(*p & ~3);
```

```
    uint type = *p & 3;
#ifdef DEBUG
    if (!ptr)
      mokAssert( p ==  limit);
#endif DEBUG
    if (p==limit)
      return;
    mokAssert( type != BUFF_DUP_HANDLE_MARK );
    switch (type) {
    case BUFF_LINK_MARK: {
      mokAssert( ptr );
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      mokAssert( (LOWBUFFMASK & (uint)ptr) == N_RESERVED_SLOTS*sizeof(uint));
      p = ptr+1; /* skip backward pointer */
      break;
    }

    case BUFF_HANDLE_MARK: {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
                                /* reinforce, if needed */
      if (!h->logPos)
        h->logPos = p;
      p++;
#ifdef RCDEBUG
      gcvar.dbg.nActualReinforceObjects ++;
#endif // RCDEBUG
      break;
    }

    case 0: {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
#ifdef RCDEBUG
      gcvar.dbg.nActualReinforceChilds ++;
#endif // RCDEBUG
      p++;
      break;
    }
    }
  }
}

static void _HS3Cooperate(ExecEnv *ee)
{
  bool res = gcCompareAndSwap( &ee->gcblk.stageCooperated, GCHSNONE, GCHS3 );
  mokAssert( res );
#ifdef RCDEBUG
  gcvar.dbg.nHS3CoopThreads++;
#endif /* RCDEBUG */
}

static int _HS3Helper(sys_thread_t *thrd, bool *allOK)
{
  ExecEnv *ee;
  bool res;

  ee = SysThread2EE( thrd );

  mokAssert( gcvar.ee != ee );

  /* already moved to the next state? */
  if (ee->gcblk.stage == GCHS3) return SYS_OK;

  /* only the collector advances the stage field */
  mokAssert( ee->gcblk.stage == GCHS2 );

  /* did the thread cooperate voluntarily? */
  res = gcCompareAndSwap( &ee->gcblk.stageCooperated, GCHS3, GCHSNONE);
  if (res) {
    ee->gcblk.stage = GCHS3;
#ifdef RCDEBUG
    gcvar.dbg.nHS3Threads += 100;
#endif /* RCDEBUG */
    return SYS_OK;
  }

  /* OK, we will suspend the thread, but only
   * if it's in cooperative mode.
   *
   * Pesimistic check:
   */
  if (ee->gcblk.cantCoop) {
    *allOK = false; /* try later */
    return SYS_OK;
  }

  /* Suspend the thread */
  mokThreadSuspendForGC( thrd );
```

```
  /*
   * Now we have to check cantCoop again.
   */
  if (ee->gcblk.cantCoop) {
    mokThreadResumeForGC( thrd );
    *allOK = false; /* try later */
    return SYS_OK;
  }

  mokAssert( ee->gcblk.stageCooperated == GCHS3 ||
             ee->gcblk.stageCooperated == GCHSNONE );

  ee->gcblk.stageCooperated = GCHSNONE;
  ee->gcblk.stage = GCHS3;
  mokThreadResumeForGC( thrd );

#ifdef RCDEBUG
  gcvar.dbg.nHS3Threads++;
#endif /* RCDEBUG */

  return SYS_OK;
}

static void _Reinforce_Clearing_Conflict_Set(void)
{
  bool allOK;

#ifdef RCDEBUG
  uint start, end;

  start = GetTickCount();
  dbgprn( 0, "_Reinforce_Clearing_Conflict_Set(begin) time=%d\n", start);
#endif

  /* do second handshake */
  mokAssert( gcvar.reinforceBuffList == NULL );

  QUEUE_LOCK( gcvar.sys_thread );
  gcvar.stage = GCHS2;
  /*
   * Link for reinforcemenr buffers of threads who
   * died between HS1 and HS2
   */
  gcvar.reinforceBuffList = gcvar.deadThreadsReinforceBuffList;
  gcvar.deadThreadsReinforceBuffList = NULL;

#ifdef RCDEBUG
  gcvar.dbg.nReinforceObjects = gcvar.dbgpersist.nDeadReinforceObjects;
  gcvar.dbgpersist.nDeadReinforceObjects = 0;

  gcvar.dbg.nReinforceChilds  = gcvar.dbgpersist.nDeadReinforceChilds;
  gcvar.dbgpersist.nDeadReinforceChilds = 0;
#endif

  /*
   * Link update buffers of live threads
   */
  for(;;) {
    allOK = true;
    mokThreadEnumerateOver( _HS2Helper, &allOK );
    if (allOK) break;
    mokSleep( 10 );
  }
  QUEUE_UNLOCK( gcvar.sys_thread );

  while ( gcvar.reinforceBuffList ) {
    uint *p = gcvar.reinforceBuffList;
    uint *limit = (uint*)gcvar.reinforceBuffList[LAST_POS_IDX];
    _reinforceUpdateBuffer( p, limit );
    gcvar.reinforceBuffList = (uint*)p[REINFORCE_LINKED_LIST_IDX];
  }

  /* do third handshake */
  QUEUE_LOCK( gcvar.sys_thread );
  gcvar.stage = GCHS3;
  for(;;) {
    allOK = true;
    mokSleep( 10 );
    mokThreadEnumerateOver( _HS3Helper, &allOK );
    if (allOK) break;
  }
  QUEUE_UNLOCK( gcvar.sys_thread );

#ifdef RCDEBUG
  end = GetTickCount();

  dbgprn( 2, "\tnHS2Threads=%d\n", gcvar.dbg.nHS2Threads );
  dbgprn( 2, "\tnHS3Threads=%d\n", gcvar.dbg.nHS3Threads );
  dbgprn( 2, "\tnHS3CoopThreads=%d\n", gcvar.dbg.nHS3CoopThreads );

  if (gcvar.dbg.nReinforceObjects || gcvar.dbg.nReinforceChilds) {
    dbgprn( 1, "\tnReinforceChilds=%d\n", gcvar.dbg.nReinforceChilds );
```

```
    dbgprn( 1, "\tnReinforceObjects=%d\n",  gcvar.dbg.nReinforceObjects );
  }

  mokAssert( gcvar.dbg.nActualReinforceObjects == gcvar.dbg.nReinforceObjects );
  mokAssert( gcvar.dbg.nActualReinforceChilds == gcvar.dbg.nReinforceChilds );

  dbgprn(
         0,
         "_Reinforce_Clearing_Conflict_Set(end) time=%d delta=%d\n",
         end,
         end-start );
#endif // RCDEBUG
}


static void _markHandlesInSnoopBufferAsLocal(uint *buff)
{
  uint *ptr, type, *p;

  mokAssert( buff );

  /* go backwards */
  p = (uint*)buff[LAST_POS_IDX];
  mokAssert( p );
  mokAssert( *p==0 );
  p--;
  mokAssert( *p );


  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_HANDLE_MARK );
    mokAssert( type != BUFF_DUP_HANDLE_MARK );
#ifdef DEBUG
    if (!ptr)
      mokAssert( buff+N_RESERVED_SLOTS == p);
    if ( buff+N_RESERVED_SLOTS == p)
      mokAssert( *p == BUFF_LINK_MARK);
#endif
    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
      _setLocal( h );
#ifdef RCDEBUG
      gcvar.dbg.nActualSnooped++;
#endif // RCDEBUG
      p--;
    }
    else { /*type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      /* free the more recent buffer */
      _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
      if (!ptr)
        return;
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }

  }
}

static void _markSnoopedAsLocal(void)
{
  uint *buff = gcvar.snoopBuffList;
  while (buff) {
    uint *nextBuff = (uint*)buff[0];
    _markHandlesInSnoopBufferAsLocal(buff);
    buff = nextBuff;
  }
  gcvar.snoopBuffList = NULL;
}


/************************** HS4 ***************************************/

#define SAFETY_MARGINE 20

static void _snoopExactHandle(JHandle *h)
{
  if (!h) return;
  mokAssert( _isHandle(h) );
  _setLocal( h );
}

static void _snoopHandleOrScalar(JHandle *h)
{
  if (_isHandle(h))
    _setLocal(h);
}
```

```
static void _snoopHandleOrObjectOrScalar(JHandle *h)
{
  if (_isHandle(h))
    _setLocal(h);
  else {
    JHandle *obj = gcRehand(h);
    if (_isHandle(obj)) {
      _setLocal( obj );
    }
  }
}

static void _snoopJavaFrame(JavaFrame *frame, stack_item *top_top_stack)
{
  stack_item *ssc, *limit;
  JHandle *ptr;
  JavaStack *javastack;
  struct methodblock *mb = frame->current_method;

  limit = top_top_stack;
  javastack = frame->javastack;

  /* Scan the operand stack. */
  /*CONSTCOND*/
  while (1) {
    int is_first_chunk = IN_JAVASTACK((stack_item *)frame, javastack);
    for (ssc = is_first_chunk ? frame->ostack : javastack->data;
         ssc < limit; ssc++) {
      ptr = ssc->h;
      _snoopHandleOrScalar( (JHandle*)ptr ); /* Never an object pointer */
    }
    if (is_first_chunk)
      break;
    javastack = javastack->prev;
    limit = javastack->end_data;
  }

  /* Nothing more to do for pseudo and JIT frames. */
  if (mb == 0 || IS_JIT_FRAME(frame)) {
    mokAssert( !IS_JIT_FRAME(frame) ); /* YLRC -- don't support JIT ... */
    return;
  }

  if (mb->fb.access & ACC_NATIVE) {
    /* For native frames, we scan the arguments stored at the top
       of the previous frame. */
    JavaFrame *prev_frame = frame->prev;
    if (prev_frame == 0)
      return;
    ssc = prev_frame->optop;
    limit = ssc + mb->args_size;
  } else {
    /* Scan local variables in Java frame */
    ssc = frame->vars;
    if (ssc == 0)
      return;
    limit = (stack_item *)frame;
  }

  for (; ssc < limit; ssc++) {
    ptr = ssc->h;
    _snoopHandleOrScalar(ptr); /* Never an object pointer */
  }
}

static void _snoopThreadLocals( sys_thread_t *t )
{
  ExecEnv *ee = SysThread2EE(t);
  JHandle *tobj = ee->thread;
  unsigned char **ssc, **limit;
  void *base;

  mokAssert( EE2SysThread(ee) != sysThreadSelf());

  if (ee->initial_stack == NULL) {
    /* EE already destroyed. */
    return;
  }

  /* Mark thread object */
  if (tobj) {
    mokAssert( gcNonNullValidHandle((GCHandle*)tobj) );
    _snoopExactHandle( tobj );
  }
  {
    long *regs;
    int nregs;

    /* Scan the saved registers */
    regs = sysThreadRegs(t, &nregs);
    for (nregs--; nregs >= 0; nregs--) {
      _snoopHandleOrObjectOrScalar( (JHandle*)regs[nregs] );
```

```
    }

    base = ee->stack_base;
    ssc  = sysThreadStackPointer(t);
  }

  if (ssc == 0 || base == 0 || (ssc == base)) {
    /*
     * If the stack does not have a top of stack pointer or a base
     * pointer then it hasn't run yet and we don't need to scan
     * its stack.  When exactly each of these data becomes available
     * may be system-dependent, but we need both to bother scanning.
     */
    goto ScanJavaStack;
  }

  /* Align stack top, important on Windows 95. */
  if ((long)ssc % sizeof(void *)) {
    ssc = (unsigned char **)((long)(ssc) & ~(sizeof(void *) - 1));
  }

  limit = (unsigned char **) base;

  mokAssert(ssc != limit);

  /*
   * The code that scans the C stack is assuming that the current
   * stack pointer is at a lower address than the limit of the stack.
   * Obvioulsy, this is only true for downward growing stacks.  For
   * upward growing stack, we exchange ssc and limit before we start
   * to scan the stack.
   */

#if defined(STACK_GROWS_UP)
  {
    unsigned char  **tmp;

    tmp = limit;
    limit = ssc;
    ssc = tmp;
  }
#endif /* STACK_GROWS_UP */


  while (ssc < limit) {
    register unsigned char *ptr = *ssc;
    _snoopHandleOrObjectOrScalar( (JHandle*)ptr );
    ssc++;
  }

  /*
   * Whether or not we scan the thread stack, we decide independently
   * whether to scan the Java stack.  Doing so should be more robust
   * in the face of partially-initialized or partially-zeroed threads
   * during thread creation or exit, or changes to any of that code.
   */
 ScanJavaStack:
  {
    JavaFrame *frame;

    /*
     * Because of the Invocation API, the EE may not be on the C
     * stack anymore.
     */
    _snoopExactHandle( ee->exception.exc );

    _snoopExactHandle(ee->pending_async_exc);

    if ((frame = ee->current_frame) != 0) {
      struct methodblock *prev_current_method = 0;
      while (frame) {
        struct methodblock *current_method = frame->current_method;
        /*
         * If the previous frame was a transition frame from C back
         * to Java (indicated by prev_current_method == NULL), then
         * this new frame might not have set its optop.  We must be
         * conservative.   Otherwise, we can use the optop value.
         *
         * Also permit two consecutive frames with NULL current
         * methods, in support of JITs.  See bug 4022856.
         */
        stack_item *top_top_stack =
          (prev_current_method == 0 && current_method != NULL &&
           ((current_method->fb.access & ACC_NATIVE) == 0))
          ? &frame->ostack[frame->current_method->maxstack]
          : frame->optop;
        _snoopJavaFrame(frame, top_top_stack);
        frame = frame->prev;
        prev_current_method = current_method;
      }
    }
  }
```

```
}


static int _HS4Helper( sys_thread_t *thrd, bool *allOK )
{
  ExecEnv *ee;

  ee = SysThread2EE( thrd );

  mokAssert( gcvar.ee != ee );

  if (ee->gcblk.stage == GCHS4) return SYS_OK;
  if (ee->gcblk.cantCoop) {
    *allOK = false;
    return SYS_OK;
  }

  while(gcvar.nPreAllocatedBuffers < 1) {
    buffInit( gcvar.ee, &gcvar.preAllocatedBuffers[gcvar.nPreAllocatedBuffers] );
    gcvar.nPreAllocatedBuffers++;
  }

  mokThreadSuspendForGC( thrd );
  mokAssert( ee->gcblk.stage == GCHS3 );
  if (ee->gcblk.cantCoop) {
    mokThreadResumeForGC( thrd );
    *allOK = false;
    return SYS_OK;
  }

  ee->gcblk.snoop = false;

  /* put into the snooped object set
   * all of the locally reachable objects
   */
  _snoopThreadLocals( thrd );

  /* now steal the snooped objects set */
  if (buffIsModified(&ee->gcblk.snoopBuffer)) {
    *ee->gcblk.snoopBuffer.pos = 0;
    ee->gcblk.snoopBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.snoopBuffer.pos;
    ee->gcblk.snoopBuffer.start[LINKED_LIST_IDX] = (uint)gcvar.snoopBuffList;
    gcvar.snoopBuffList = ee->gcblk.snoopBuffer.start;

#ifdef RCDEBUG
    gcvar.dbg.nSnooped += ee->gcblk.snoopBuffer.start[LOG_OBJECTS_IDX];
#endif // RCDEBUG

    /* give the thread a new snoop buffer to play with */
    gcvar.nPreAllocatedBuffers--;
    ee->gcblk.snoopBuffer = gcvar.preAllocatedBuffers[gcvar.nPreAllocatedBuffers];
  }

#ifdef RCDEBUG
  gcvar.dbg.nHS4Threads++;
#endif // RCDEBUG

  /* restart the thread */
  ee->gcblk.stage = GCHS4;

  mokThreadResumeForGC( thrd );
  return SYS_OK;
}

static void _snoopClass(ClassClass *cb)
{
  /* We must be extra careful in scanning the internals of a class
   * structure, because this routine may be called when a class
   * is only partially loaded (in createInternalClass).
   */
  /*
   * YLRC --
   *
   * No need to recursively trace super classes as we mark all
   * classes anyway.  This also holds for classes referred
   * to from the constant pool.
   *
   */
  JHandle *h;

  if (cbConstantPool(cb) &&
      cbConstantPool(cb)[CONSTANT_POOL_TYPE_TABLE_INDEX].type) {
    union cp_item_type *constant_pool = cbConstantPool(cb);
    union cp_item_type *cpp =
      constant_pool+ CONSTANT_POOL_UNUSED_INDEX;
    union cp_item_type *end_cpp =
      &constant_pool[cbConstantPoolCount(cb)];
    unsigned char *type_tab =
      constant_pool[CONSTANT_POOL_TYPE_TABLE_INDEX].type;
    unsigned char *this_type =
      &type_tab[CONSTANT_POOL_UNUSED_INDEX];
```

```
    for ( ; cpp < end_cpp; cpp++, this_type++) {
      if (*this_type == (CONSTANT_String|CONSTANT_POOL_ENTRY_RESOLVED)) {
        _snoopExactHandle( (JHandle*)(*cpp).p );
      }
    } /* loop over constant pool*/
  }

  /* Scan class definitions looking for statics */
  if (cbFields(cb) &&
      (cbFieldsCount(cb) > 0)) { /* defensive check */
    int i;
    struct fieldblock *fb;
    for (i = cbFieldsCount(cb), fb = cbFields(cb); --i >= 0; fb++) {
      if (fieldsig(fb) &&  /* Extra defensive */
          (fieldIsArray(fb) || fieldIsClass(fb)) && (fb->access & ACC_STATIC)) {
        JHandle *sub = *(JHandle **)normal_static_address(fb);
        _snoopExactHandle( sub );
      }
    }
  }

  h = (JHandle *)cbClassname(cb);
  _snoopExactHandle( h );

  h = (JHandle *)cbLoader(cb);
  _snoopExactHandle( h );

  h = (JHandle *)cbSigners(cb);
  _snoopExactHandle( h );

  h = (JHandle *)cbProtectionDomain(cb);
  _snoopExactHandle( h );
}

static void _snoopBinClasses(void)
{
  ClassClass **pcb;
  int i;

  BINCLASS_LOCK( sysThreadSelf() /*gcvar.sys_thread*/ );
  pcb = binclasses;
  for (i = nbinclasses; --i >= 0; pcb++) {
    ClassClass *cb = *pcb;
    _snoopExactHandle( (JHandle*)cb );
    _snoopClass( cb );
  }
  BINCLASS_UNLOCK( sysThreadSelf() /*gcvar.sys_thread*/ );
}

static void _snoopPrimitiveClasses(void)
{
  static ClassClass **primitive_classes[] = {
    &class_void, &class_boolean, &class_byte, &class_char, &class_short,
    &class_int, &class_long, &class_float, &class_double, NULL
  };
  ClassClass ***cbpp = primitive_classes;

  while (*cbpp) {
    ClassClass *cb = **cbpp;
    _snoopExactHandle( (JHandle*)cb );
    _snoopClass( cb );
    cbpp++;
  }
}

static void _snoopMonitorCacheHelper(monitor_t *mid, void *cookie)
{
  JHandle *h = (JHandle*) mid->key;
  if (_isHandle(h) && sysMonitorInUse(sysmon(mid)) ) {
    _snoopExactHandle( h );
  }
}

static void _snoopMonitorCache(void)
{
  CACHE_LOCK( sysThreadSelf() /*gcvar.sys_thread*/ );
  monitorEnumerate( _snoopMonitorCacheHelper, 0);
  CACHE_UNLOCK( sysThreadSelf() /*gcvar.sys_thread*/ );
}

static void  _snoopJNIGlobalsRefs( void )
{
  _snoopJavaFrame(globalRefFrame, globalRefFrame->optop);
}

static void _snoopInternedStrings(void);

static void _snoopGlobals(void)
{
  _snoopBinClasses(  );
  _snoopPrimitiveClasses(  );
```

```
  _snoopMonitorCache(  );
  _snoopInternedStrings( );
  _snoopJNIGlobalsRefs(  );
}


static void _Consolidate( void )
{
  bool allOK;
#ifdef RCDEBUG
  uint start, end;

  start = GetTickCount();
  dbgprn( 0, "_Consolidate(begin) time=%d\n", start);
#endif

  if (gcvar.collectionType == GCT_TRACING)
    _traceSetup();

  /* init buffer of local objects */
  buffInit( gcvar.ee, &gcvar.uniqueLocalsBuff );

  /* snoop global objects */
  _snoopGlobals( );

#ifdef RCDEBUG
  gcvar.dbg.nGlobals = gcvar.dbg.nLocals;
  gcvar.dbg.nLocals = 0;
#endif

  /* do fourth handshake */
  QUEUE_LOCK( gcvar.sys_thread );

  gcvar.stage = GCHS4;
  mokAssert( gcvar.snoopBuffList == NULL );

  /* add snoop buffers of dead threads and
   * clear the list
   */
  gcvar.snoopBuffList = gcvar.deadThreadsSnoopBuffList;
  gcvar.deadThreadsSnoopBuffList = NULL;


#ifdef RCDEBUG
  gcvar.dbg.nSnooped = gcvar.dbgpersist.nDeadSnooped;
  gcvar.dbgpersist.nDeadSnooped = 0;
#endif

  /* now add the threads buffers */
  for(;;) {
    allOK = true;
    mokThreadEnumerateOver( _HS4Helper, &allOK );
    if (allOK) break;
    mokSleep( 10 );
  }

  QUEUE_UNLOCK( gcvar.sys_thread );

  /* process thread buffers */
  _markSnoopedAsLocal();


#ifdef RCDEBUG
  end = GetTickCount();

  dbgprn( 2, "\tnHS4Threads=%d\n", gcvar.dbg.nHS4Threads );
  dbgprn( 2, "\tnSnooped=%d\n",  gcvar.dbg.nSnooped );
  dbgprn( 4, "\tnActualSnooped=%d\n", gcvar.dbg.nActualSnooped );
  dbgprn( 2, "\tnLocals=%d\n", gcvar.dbg.nLocals );
  dbgprn( 2, "\tnGlobals=%d\n", gcvar.dbg.nGlobals );

  mokAssert( gcvar.dbg.nActualSnooped == gcvar.dbg.nSnooped );
  dbgprn( 0, "_Consolidate(end) time=%d delta=%d\n", end, end-start);

#endif // RCDEBUG
}


/*****************************************************************/
/************************  UPDATE PHASE  *************************/
/*****************************************************************/

/************************  Updating Counters *********************/

static void _determineHandleContents(GCHandle *h)
{
  uint *p;

 start:
  p = h->logPos;

  if (p) {
```

```
    mokAssert( h == (GCHandle*)(*p^BUFF_HANDLE_MARK) );
#ifdef RCDEBUG
    gcvar.dbg.nUndetermined++;
#endif // RCDEBUG
    p--;
    while (1) {
      GCHandle *hSon = (GCHandle*)*p;
      uint type = 3 & *p;
      mokAssert( hSon );
      if (type) return;
      _incrementHandleRC( hSon );
      p--;
    }
  }

  {
    GCHandle **tempbuff = gcvar.tempReplicaSpace;
    register GCHandle *child;
    register GCHandle **objslots;

    switch (obj_flags(h)) {
    case T_NORMAL_OBJECT:{
      register ClassClass *cb = obj_classblock(h);
      register unsigned short  offset;
      register unsigned short *object_offsets ;

      if (cb == classJavaLangClass || unhand(cb)->n_object_offsets==0) {
#ifdef RCDEBUG
        gcvar.dbg.nDetermined++;
#endif
        return;
      }

      object_offsets = cbObjectOffsets(cb);
      objslots = (GCHandle **)(((char *)unhand(h)) - 1);
      while ((offset = *object_offsets++)) {
        child = *(GCHandle **) ((char *) objslots + offset);
        if (child) {
          tempbuff++;
          *tempbuff = child;
        }
      }
      break;
    }

    case T_CLASS: {     /* an array of classes */
      register long n = obj_length(h);
      GCHandle **body = (GCHandle**)(((ArrayOfObject*)gcUnhand(h))->body);
      while (--n >= 0) {
        child  = body[n];
        if (child) {
          tempbuff++;
          *tempbuff = child;
        }
      }
      break;
    }
    }
    if (h->logPos) {
      goto start;
    }
    /* OK, the replica we have at this point is valid
     * so use it as the reference to the objects'
     * contents.
     */
#ifdef RCDEBUG
    gcvar.dbg.nDetermined++;
#endif // RCDEBUG
    while( tempbuff > gcvar.tempReplicaSpace) {
      child = *tempbuff;
      _incrementHandleRC( child );
      tempbuff--;
    }
  }
}

static void _updateRCofSingleUpdateLog(uint *buff)
{
  uint *ptr, type, *p;

  mokAssert( buff );

  /*
   * go backwards since its better to
   * first increment and only then decrement
   * (it will cause less entries in the ZCT)
   * so we want to first see the handle and
   * only then its contents.
   */
  p = (uint*)buff[LAST_POS_IDX];

  mokAssert( p );
```

```
  mokAssert( *p==0 );
  p--;
  mokAssert( *p );


  for (;;) {
    type = *p & 3;
  next_round:
    ptr = (uint*)(*p & ~3);
    mokAssert( type != 0 );
    switch (type) {
    case BUFF_LINK_MARK: {
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
                                /* free the more recent buffer */
      _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
        return;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
      break;
    }

    case BUFF_HANDLE_MARK: {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
      _determineHandleContents( h );
#ifdef RCDEBUG
      gcvar.dbg.nUpdateRCObjects++;
#endif // RCDEBUG
      for(;;) {
        GCHandle *h;
        p--;
        h = (GCHandle*)*p;
        type = ((uint)h) &3;
        if (type) goto next_round;
        mokAssert( gcNonNullValidHandle(h) );
        _decrementHandleRCInUpdate( h );
#ifdef RCDEBUG
        gcvar.dbg.nUpdateRCChilds++;
#endif // RCDEBUG
      }
    }

    case BUFF_DUP_HANDLE_MARK: {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
#ifdef RCDEBUG
      gcvar.dbg.nUpdateRCObjects++;
      gcvar.dbg.nUpdateRCDuplicates++;
#endif // RCDEBUG
      for(;;) {
        p--;
        type = *p & 3;
        if (type) goto next_round;
#ifdef RCDEBUG
        gcvar.dbg.nUpdateRCChilds++;
#endif // RCDEBUG
      }
    }
    }
  }
}


static void _updateRCofSingleCreateLog(uint *buff)
{
  uint *ptr, type, *p;

  mokAssert( buff );

  p = (uint*)buff[LAST_POS_IDX];
  mokAssert( p );
  mokAssert( *p == 0 );
  p--;
  mokAssert( *p );

  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_HANDLE_MARK );
    mokAssert( type != BUFF_DUP_HANDLE_MARK );

    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
      _determineHandleContents( h );
#ifdef RCDEBUG
      gcvar.dbg.nCreateRCObjects++;
```

```
#endif // RCDEBUG
      p--;
    }
    else { /* type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
        return;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }
  }
}

static void _updateRCofUpdateLog( void )
{
  uint *log = gcvar.updateBuffList;
  while (log) {
    uint *nextLog = (uint*)log[0];
    _updateRCofSingleUpdateLog( log );
    log = nextLog;
  }
  gcvar.updateBuffList = NULL;
}

static void _updateRCofCreateLog( void )
{
  uint *log = gcvar.createBuffList;
  while (log) {
    _updateRCofSingleCreateLog( log );
    log = (uint*)log[0];
  }
}


static void _Update_Reference_Counters( void )
{
#ifdef RCDEBUG
  uint start, end;

  mokAssert( gcvar.zctBuff.start[LOG_OBJECTS_IDX] == gcvar.dbg.nInZct );

  start = GetTickCount();

  dbgprn( 0, "__Update_Reference_Counters(begin) time=%d\n", start);
#endif // RCDEBUG

  _updateRCofUpdateLog();
  _updateRCofCreateLog();

#ifdef RCDEBUG
  end = GetTickCount();

  dbgprn( 3, "\tnUpdateRCObjects=%d\n",  gcvar.dbg.nUpdateRCObjects );
  dbgprn( 3, "\tnUpdateRCChilds=%d\n",  gcvar.dbg.nUpdateRCChilds );
  dbgprn( 3, "\tnUpdateRCDuplicates=%d\n",  gcvar.dbg.nUpdateRCDuplicates );
  dbgprn( 3, "\tnCreateRCObjects=%d\n",  gcvar.dbg.nCreateRCObjects );
  dbgprn( 2, "\tnDetermined=%d\n", gcvar.dbg.nDetermined );
  dbgprn( 2, "\tnUndetermined=%d\n", gcvar.dbg.nUndetermined );
  dbgprn( 2, "\tnInZct=%d\n", gcvar.dbg.nInZct );

  mokAssert( gcvar.dbg.nDetermined+gcvar.dbg.nUndetermined ==
             gcvar.dbg.nUpdateObjects + gcvar.dbg.nCreateObjects -
             (gcvar.dbg.nUpdateDuplicates + gcvar.dbg.nActualCyclesBroken) );
  mokAssert( gcvar.dbg.nUpdateRCObjects == gcvar.dbg.nUpdateObjects);
  mokAssert( gcvar.dbg.nUpdateRCChilds == gcvar.dbg.nUpdateChilds);
  mokAssert( gcvar.dbg.nUpdateRCDuplicates ==
             gcvar.dbg.nUpdateDuplicates +gcvar.dbg.nActualCyclesBroken);
  mokAssert( gcvar.dbg.nCreateRCObjects == gcvar.dbg.nCreateObjects );
  mokAssert( gcvar.zctBuff.start[LOG_OBJECTS_IDX] == gcvar.dbg.nInZct );

  dbgprn( 0, "_Update_Reference_Counters(end) time=%d delta=%d\n", end, end-start );
#endif // RCDEBUG
}

/*********************  Reclamation ***********************************/

static void _throwNonZerosFromCurrentZCT( BUFFHDR *tmpZCT )
{
  uint *ptr, type, *p, *buff;

#ifdef RCDEBUG
  uint nOld = 0, nDel = 0, nThrown =0, nPend=0;
  uint start, end;

  start = GetTickCount();
  dbgprn( 0, "_throwNonZerosFromCurrentZCT(start) time=%d\n", start );
#endif

  buff = gcvar.zctBuff.start;
```

```
  mokAssert( (((uint)buff) & LOWBUFFMASK) == 0);
  mokAssert( buff );

  p = gcvar.zctBuff.pos-1;

  mokAssert( p );
  mokAssert( *p );

  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_HANDLE_MARK );
    mokAssert( type != BUFF_DUP_HANDLE_MARK );

    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
#ifdef RCDEBUG
      nOld++;
#endif
      mokAssert( h );
      mokAssert( gcNonNullValidHandle(h) );
      mokAssert( _isInZCT(h) );
      if (gcGetHandleRC(h) > 0) {
        _markNotInZCT(h);
#ifdef RCDEBUG
        nThrown++;
#endif // RCDEBUG
      }
      else {
#ifdef RCDEBUG
        nDel++;
#endif
        gcBuffLogWord( gcvar.ee, tmpZCT, (unsigned)h );
      }
      p--;
    }
    else { /*type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      /* free the more recent buffer */
      _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
        goto __end;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }
  }
 __end:;
#ifdef RCDEBUG
  end = GetTickCount();

  mokAssert( gcvar.tmpZctBuff.start[LOG_OBJECTS_IDX] == nDel );
  mokAssert( nThrown+nDel+nPend == nOld );

  gcvar.dbg.nInZct = gcvar.tmpZctBuff.start[LOG_OBJECTS_IDX];
  gcvar.dbgpersist.nPendInCycle= nPend;

  dbgprn( 2, "\tnOld=%d\n",  nOld  );
  dbgprn( 2, "\tnDel=%d\n",  nDel  );
  dbgprn( 2, "\tnPend=%d\n", nPend );
  dbgprn( 2, "\tnThrown=%d\n", nThrown );
  dbgprn( 2, "\tnInZct=%d\n", gcvar.dbg.nInZct );
  dbgprn( 2, "\tnInNextZct=%d\n", gcvar.dbgpersist.nPendInCycle );
  dbgprn( 2, "_throwNonZerosFromCurrentZCT(end) time=%d delta=%d\n", end, end-start );
#endif
}


static void _processCreateBuffsIntoZCT( void )
{
#ifdef RCDEBUG
  uint nCreate = 0, nDel = 0, nThrown=0, nPend=0;
  uint nAlreadyInZct=0;
  uint start, end;
#endif

  uint *ptr, type, *p;
  uint *buff = gcvar.createBuffList, *nextBuff;
  BUFFHDR *tmpZCT = &gcvar.tmpZctBuff;


#ifdef RCDEBUG
  start = GetTickCount();
  dbgprn( 0, "_processCreateBuffIntoZCT(start)  time=%d\n", start );
#endif

  while (buff) {
    nextBuff = (uint*)buff[0];

    mokAssert( (((uint)buff) & LOWBUFFMASK) == 0);
    mokAssert( buff );
```

```
    p = (uint*)buff[LAST_POS_IDX];
    mokAssert( p );
    mokAssert( *p == 0 );
    p--;
    mokAssert( *p );

    for        (;;) {
      ptr = (uint*)(*p & ~3);
      type = *p & 3;
      mokAssert( type != BUFF_HANDLE_MARK );
      mokAssert( type != BUFF_DUP_HANDLE_MARK );
      if (type==0) {
        GCHandle *h = (GCHandle*)ptr;
#ifdef RCDEBUG
        nCreate++;
#endif
        mokAssert( h );
        mokAssert( gcNonNullValidHandle(h) );
        if (gcGetHandleRC(h) == 0) {
          if (!_isInZCT(h)) {
            _markInZCT( h );
#ifdef RCDEBUG
            nDel++;
#endif
            gcBuffLogWord( gcvar.ee, tmpZCT, (unsigned)h );
          }
#ifdef RCDEBUG
          else {
            nAlreadyInZct++;
          }
#endif // RCDEBUG
        }
#ifdef RCDEBUG
        else {
          nThrown++;
        }
#endif // RCDEBUG
        p--;
      }
      else { /* type==BUFF_LINK_MARK*/
        mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
                                  /* free the more recent buffer */
        _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
        if (!ptr) {
          mokAssert( buff+N_RESERVED_SLOTS == p);
          goto __end_chunk;
        }
        mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
        p = ptr-1; /* skip forward pointer */
      }
    }
  __end_chunk:
    buff = nextBuff;
  }
  gcvar.createBuffList = NULL;

#ifdef RCDEBUG
  end = GetTickCount();

  mokAssert( gcvar.tmpZctBuff.start[LOG_OBJECTS_IDX] == nDel + gcvar.dbg.nInZct );
  gcvar.dbg.nInZct = gcvar.tmpZctBuff.start[LOG_OBJECTS_IDX] ;

  gcvar.dbgpersist.nPendInCycle += nPend;

  mokAssert( gcvar.dbg.nCreateObjects == nCreate );
  mokAssert( nThrown+nDel+nPend+nAlreadyInZct == nCreate );

  gcvar.dbg.nCreateDel = nDel;

  dbgprn( 2, "\tnCreate=%d\n", nCreate  );
  dbgprn( 2, "\tnDel=%d\n",  nDel  );
  dbgprn( 2, "\tnPend=%d\n", nPend );
  dbgprn( 2, "\tnThrown=%d\n", nThrown );
  dbgprn( 2, "\tnInZct=%d\n", gcvar.dbg.nInZct );
  dbgprn( 2, "\tnInNextZct=%d\n",  gcvar.dbgpersist.nPendInCycle );
  dbgprn( 0, "_processCreateBuffIntoZCT(end) time=%d delta=%d\n", start, end-start );
#endif
}

#pragma optimize( "", off )
static void _freeHandle(GCHandle* h)
{
  for (;;) {
    unsigned *p;
    BlkAllocBigHdr *bh;
    int status;

    mokAssert( h );
    mokAssert( gcNonNullValidHandle(h) );
    mokAssert( gcGetHandleRC(h)==0 );
```

```
#ifdef RCDEBUG
    {
      unsigned obj_type = obj_flags(h);
      if (obj_type == T_NORMAL_OBJECT) {
        register ClassClass *cb = obj_classblock(h);
        gcvar.dbg.nRefsFreedInCycle += unhand(cb)->n_object_offsets;
      }
      else if (obj_type == T_CLASS) { /* an array of references */
        long n = obj_length(h);
        gcvar.dbg.nRefsFreedInCycle += n;
      }
    }
#endif // RCDEBUG

    p = h->logPos;
    if (p) {
#ifdef RCDEBUG
      dbgprn( 1, "\t\tfree:dirty: %x\n", h);
      mokAssert( h == (GCHandle*)(*p^BUFF_HANDLE_MARK) );
      h->logPos = NULL;
      gcvar.dbgpersist.nFreeCyclesBroken++;
#endif
      *p = *p | BUFF_DUP_HANDLE_MARK;
      p--;
      while (1) {
        GCHandle *child = (GCHandle*)*p;
        uint type = 3 & *p;
        mokAssert( child );
        if (type) break;
#ifdef RCDEBUG
        dbgprn( 3, "\t\tfree:dirty:dec %x\n", child);
#endif
        _decrementHandleRCInDeletion( child );
        p--;
      }
    }
    else {
      register GCHandle  *child;
      register char      *objslots;
      unsigned obj_type = obj_flags(h);

      if (obj_type == T_NORMAL_OBJECT) {
        register ClassClass *cb = obj_classblock(h);
        unsigned short *object_offsets;
        int offset;

        mokAssert( cb != classJavaLangClass);

        object_offsets = cbObjectOffsets(cb);
        if (object_offsets) {
          objslots = ((char *)gcUnhand(h)) - 1;
          while ((offset = *object_offsets++)) {
            child =  *((GCHandle **) (((char *)objslots) + offset));
            if (child) {
              mokAssert( gcNonNullValidHandle(child) );
              _decrementHandleRCInDeletion( child );
            }
          }
        }
      }
      else if (obj_type == T_CLASS) { /* an array of references */
        register long n = obj_length(h);
        GCHandle **body;

        body = (GCHandle**)(((ArrayOfObject *)gcUnhand(h))->body);
        while (--n >= 0) {
          child = body[n];
          if (child) {
            _decrementHandleRCInDeletion( child );
          }
        }
      }
    }
#ifdef RCDEBUG
    gcvar.dbg.nFreedInCycle++;
    h->status = Im_free;
#endif
    bh = (BlkAllocBigHdr *)OBJBLOCKHDR(h);
    status = bhGet_status( bh );

    mokAssert( status==ALLOCBIG ||
               status==VOIDBLK ||
               status==PARTIAL ||
               status==OWNED );
    mokAssert( ALLOCBIG < OWNED );
    mokAssert( OWNED < VOIDBLK );
    mokAssert( VOIDBLK < PARTIAL );

    if (status == ALLOCBIG) {
#ifdef RCDEBUG
      gcvar.dbg.nBytesFreedInCycle +=
        ((BlkAllocBigHdr *)OBJBLOCKHDR(h))->blobSize * BLOCKSIZE;
```

```
#endif
      blkFreeRegion( (BlkAllocBigHdr *)OBJBLOCKHDR(h) );
    }
    else {
#ifdef RCDEBUG
      gcvar.dbg.nBytesFreedInCycle +=
        chkconv.binSize[ bhGet_bin_idx( (BlkAllocHdr*)bh ) ];
#endif
      chkPreCollect( (BLKOBJ*)h );
    }
    if (gcvar.zctStackSp == gcvar.zctStack)
      return;
    gcvar.zctStackSp--;
    h = *gcvar.zctStackSp;
  }
}
#pragma optimize( "", on )

static void _freeHandlesOnTempZCT(BUFFHDR *tmpZCT)
{
  uint *buff = tmpZCT->start;
  uint *ptr, type, *p;

#ifdef RCDEBUG
  uint start, end;
  uint nInZCT = 0;

  start = GetTickCount();
  dbgprn( 0, "_freeHandlesOnTempZCT(start) time=%d\n", start );
#endif // RCDEBUG


  mokAssert( (((uint)buff) & LOWBUFFMASK) == 0);
  mokAssert( buff );

  p = tmpZCT->pos - 1;
  mokAssert( p );
  mokAssert( *p );

  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_DUP_HANDLE_MARK );
    mokAssert( type != BUFF_HANDLE_MARK );

    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( _isInZCT(h) );
      mokAssert( gcNonNullValidHandle(h) );
      mokAssert( gcGetHandleRC(h)==0 );
      _freeHandle( h );
      _markNotInZCT(h);
#ifdef RCDEBUG
      nInZCT++;
#endif // RCDEBUG
      p--;
    }
    else { /* type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      /* free the more recent buffer */
      _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
#ifdef RCDEBUG
        mokAssert( nInZCT == gcvar.tmpZctBuff.start[LOG_OBJECTS_IDX] );
#endif // RCDEBUG
        goto __end;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }
  }
 __end:;
#ifdef RCDEBUG
  end = GetTickCount();
  dbgprn( 2, "\tnFreedInCycle=%d\n", gcvar.dbg.nFreedInCycle );
  dbgprn( 2, "\tnRecursiveDel=%d\n", gcvar.dbg.nRecursiveDel );
  dbgprn( 2, "\tnRecursivePend=%d\n", gcvar.dbg.nRecursivePend );
  dbgprn( 0, "_freeHandlesOnTempZCT(start) delta=%d\n", end-start );
#endif
}

static void _processLocalsIntoNextZCT( void)
{
  uint *buff = gcvar.uniqueLocalsBuff.start;
  uint *ptr, type, *p;

#ifdef RCDEBUG
  uint start, end;
  start = GetTickCount();
  dbgprn( 0, "_processLocalsIntoNextZCT(start) time=%d\n", start );
```

```
#endif // RCDEBUG


  mokAssert( (((uint)buff) & LOWBUFFMASK) == 0);
  mokAssert( buff );

  /* allocate buffer for next ZCT */
  buffInit( gcvar.ee, &gcvar.nextZctBuff );

  p = gcvar.uniqueLocalsBuff.pos - 1;
  mokAssert( p );
  mokAssert( *p );

  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_DUP_HANDLE_MARK );
    mokAssert( type != BUFF_HANDLE_MARK );

    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( h );
      mokAssert( _isHandle( h ) );
      mokAssert( !_isInZCT(h) );

      _unsetLocal(h);
      _decrementLocalHandleRC( h );

      p--;
    }
    else { /* type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      /* free the more recent buffer */
      _freeBuff( gcvar.ee, p - N_RESERVED_SLOTS);
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
#ifdef RCDEBUG
        gcvar.uniqueLocalsBuff.pos = NULL;
#endif
        goto checkout;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }
  }
 checkout:;
#ifdef RCDEBUG
  end = GetTickCount();
  dbgprn( 2, "\tnPendInCycle=%d\n", gcvar.dbgpersist.nPendInCycle );
  dbgprn( 0, "_processLocalsIntoNextZCT(start) delta=%d\n", end-start );
#endif
}

static void _Reclaim_Garbage(void)
{
  buffInit( gcvar.ee, &gcvar.tmpZctBuff );

  _throwNonZerosFromCurrentZCT( &gcvar.tmpZctBuff );

  _processCreateBuffsIntoZCT( );

  _freeHandlesOnTempZCT( &gcvar.tmpZctBuff );

  chkFlushRecycledListsCache( );
}


/************************************************************
************* Tracing Cycle Stuff **************************
************************************************************/

static void _freeListOfBuffers( uint* buff )
{
  while (buff) {
    uint *next;
    next = (uint*)buff[NEXT_BUFF_IDX];
    _freeBuff( gcvar.ee, buff );
    buff = next;
  }
}


static void _freeListOfListsOfBuffers( uint *buff)
{
  while (buff) {
    uint *next;
    next = (uint*)buff[LINKED_LIST_IDX];
    _freeListOfBuffers( buff );
    buff = next;
  }
}
```

```
static void _traceSetup( void )
{
  _freeListOfListsOfBuffers( gcvar.createBuffList );
  gcvar.createBuffList = NULL;

  _freeListOfListsOfBuffers( gcvar.updateBuffList );
  gcvar.updateBuffList = NULL;

  *gcvar.zctBuff.pos = 0;
  gcvar.zctBuff.start[ LAST_POS_IDX ] = (int)gcvar.zctBuff.pos;
  _freeListOfBuffers( gcvar.zctBuff.start );

  /* Decommit the "zct" bmp */
  mokMemDecommit( gcvar.zctBmp.bmp, gcvar.zctBmp.bmp_size );

  /*  Clear the "rc" bmp */
  mokMemDecommit( gcvar.rcBmp.bmp, gcvar.rcBmp.bmp_size );
  mokMemCommit( gcvar.rcBmp.bmp, gcvar.rcBmp.bmp_size, true );
}

static void _scanHandle(GCHandle *h)
{
  int prevRC = _incrementHandleRCWithReturnValue( h );
  if (prevRC == 0)
    _putInMarkStack( h );
}


static void _markHandleSons(GCHandle *h)
{
  uint *p;

 start:
  p = h->logPos;

#ifdef RCDEBUG
  gcvar.dbg.nTracedInCycle++;
#endif // RCDEBUG
  if (p) {
#ifdef RCDEBUG
    gcvar.dbg.nUndetermined++;
#endif // RCDEBUG
    if ( ((*p) & 3) == 0) { /* newly created object */
      /*
       * must be called directly from _traceFromLocals
       */
      mokAssert( _isLocal(h) );
      return;
    }
    mokAssert( h == (GCHandle*)(*p^BUFF_HANDLE_MARK) );
    p--;
    while (1) {
      GCHandle *hSon = (GCHandle*)*p;
      uint type = 3 & *p;
      mokAssert( hSon );
      if (type) return;
      _scanHandle( hSon );
      p--;
    }
  }

  {
    GCHandle **tempbuff = gcvar.tempReplicaSpace;
    register GCHandle *child;
    register GCHandle **objslots;

    switch (obj_flags(h)) {
    case T_NORMAL_OBJECT:{
      register ClassClass *cb = obj_classblock(h);
      register unsigned short  offset;
      register unsigned short *object_offsets ;

      if (cb == classJavaLangClass || unhand(cb)->n_object_offsets==0) {
#ifdef RCDEBUG
        gcvar.dbg.nDetermined++;
#endif
        return;
      }

      object_offsets = cbObjectOffsets(cb);
      objslots = (GCHandle **)(((char *)unhand(h)) - 1);
      while ((offset = *object_offsets++)) {
        child = *(GCHandle **) ((char *) objslots + offset);
        if (child) {
          tempbuff++;
          *tempbuff = child;
        }
      }
      break;
    }
```

```
    case T_CLASS: {     /* an array of classes */
      register long n = obj_length(h);
      GCHandle **body = (GCHandle**)(((ArrayOfObject*)gcUnhand(h))->body);
      while (--n >= 0) {
        child  = body[n];
        if (child) {
          tempbuff++;
          *tempbuff = child;
        }
      }
      break;
    }
    }
    if (h->logPos) {
      goto start;
    }
    /* OK, the replica we have at this point is valid
     * so use it as the reference to the objects'
     * contents.
     */
#ifdef RCDEBUG
    gcvar.dbg.nDetermined++;
#endif // RCDEBUG
    while( tempbuff > gcvar.tempReplicaSpace) {
      child = *tempbuff;
      _scanHandle( child );
      tempbuff--;
    }
  }
}


static void _emptyMarkStack( void )
{
  for (;;) {
    GCHandle *h;

    if (gcvar.zctStackSp == gcvar.zctStack)
      return;
    gcvar.zctStackSp--;
    h = *gcvar.zctStackSp;

#ifdef RCDEBUG
    mokAssert( _isHandle(h) );
    mokAssert( gcGetHandleRC(h) > 0);
    {
      /*
       * Check that if we see an object nested in
       * another one then this object cannot be
       * a one created since the beginning of the
       * cycle.
       */
      uint *p = h->logPos;
      if (p) {
        mokAssert( h == (GCHandle*)(*p^BUFF_HANDLE_MARK) );
      }
    }
#endif
    _markHandleSons( h );
  }
}


static void _traceFromLocals( void)
{
  uint *buff = gcvar.uniqueLocalsBuff.start;
  uint *ptr, type, *p;

  mokAssert( (((uint)buff) & LOWBUFFMASK) == 0);
  mokAssert( buff );

  p = gcvar.uniqueLocalsBuff.pos - 1;
  mokAssert( p );
  mokAssert( *p );

  for (;;) {
    ptr = (uint*)(*p & ~3);
    type = *p & 3;
    mokAssert( type != BUFF_DUP_HANDLE_MARK );
    mokAssert( type != BUFF_HANDLE_MARK );

    if (type==0) {
      GCHandle *h = (GCHandle*)ptr;
      mokAssert( _isHandle(h) );
#ifdef RCDEBUG
      {
        int rc = gcGetHandleRC( h );
        mokAssert( rc >= 1 );
      }
#endif
      _markHandleSons( h );
      _emptyMarkStack();
```

```
      p--;
    }
    else { /* type==BUFF_LINK_MARK*/
      mokAssert( (LOWBUFFMASK & (uint)p) == N_RESERVED_SLOTS*sizeof(uint));
      if (!ptr) {
        mokAssert( buff+N_RESERVED_SLOTS == p);
        return;
      }
      mokAssert( *ptr == BUFF_LINK_MARK|(uint)p );
      p = ptr-1; /* skip forward pointer */
    }
  }
}


static void _Trace( void )
{
#ifdef RCDEBUG
  uint start, end;
  start = GetTickCount();
  dbgprn( 0, "_Trace(start) time=%d\n", start );
#endif

  _traceFromLocals();

#ifdef RCDEBUG
  end = GetTickCount();
  dbgprn( 2, "\tnTracedInCycle=%d\n", gcvar.dbg.nTracedInCycle );
  dbgprn( 0, "_Trace(end) delta=%d\n", end-start );
#endif
}



static void _Sweep( void )
{
#ifdef RCDEBUG
  uint start, end;
  start = GetTickCount();
  dbgprn( 0, "_Sweep(start) time=%d\n", start );
#endif

  blkSweep();

#ifdef RCDEBUG
  end = GetTickCount();
  dbgprn( 2, "\tnFreedInCycle=%d\n", gcvar.dbg.nFreedInCycle );
  dbgprn( 0, "_Sweep(end) delta=%d\n", end-start );
#endif
}

/***************** GC Driver Func **************/
#if 0
static int _ResumeHelper( sys_thread_t *thrd, bool *allOK )
{
  ExecEnv *ee;

  mokAssert( gcvar.sys_thread != thrd );
  ee = SysThread2EE( thrd );
  if (ee->gcblk.gcSuspended)
    mokThreadResumeForGC( thrd );
  return SYS_OK;
}
#endif /* 0 */

#ifdef RCDEBUG
static void _printStats(void)
{
  float avg, avgs;
  dbgprn( 1, " __________  THIS CYCLE STATS _____________:\n");
  dbgprn( 1, "STORE: new=%d old=%d\n",
          gcvar.dbg.nNewObjectUpdatesInCycle,
          gcvar.dbg.nOldObjectUpdatesInCycle );
  dbgprn( 1, "UPDATE: updated=%d logged-slots=%d\n",
          gcvar.dbg.nUpdateObjects, gcvar.dbg.nUpdateChilds );
  if (gcvar.dbg.nCreateObjects) {
    avg = (float)gcvar.dbg.nBytesAllocatedInCycle/gcvar.dbg.nCreateObjects;
    avgs = (float)gcvar.dbg.nRefsAllocatedInCycle/gcvar.dbg.nCreateObjects;
  }
  else {
    avg =-1;
    avgs = -1;
  }

  dbgprn( 1, "CREATE: objects=%d bytes=%d avg=%f refs=%d avg=%f\n",
          gcvar.dbg.nCreateObjects, gcvar.dbg.nBytesAllocatedInCycle, avg,
          gcvar.dbg.nRefsAllocatedInCycle, avgs);
  dbgprn( 1,
          "RECLAIM: objects=%d   bytes=%d\n",
          gcvar.dbg.nFreedInCycle,
          gcvar.dbg.nBytesFreedInCycle );
  dbgprn( 1, "STUCK: %d\n", gcvar.dbg.nStuckCountersInCycle );
```

```
  gcvar.dbgpersist.nLoggedUpdates += gcvar.dbg.nUpdateObjects;
  gcvar.dbgpersist.nLoggedSlots += gcvar.dbg.nUpdateChilds;

  gcvar.dbgpersist.nObjectsAllocated += gcvar.dbg.nCreateObjects;
  gcvar.dbgpersist.nBytesAllocated += gcvar.dbg.nBytesAllocatedInCycle;
  gcvar.dbgpersist.nRefsAllocated += gcvar.dbg.nRefsAllocatedInCycle;
  gcvar.dbgpersist.nObjectsFreed +=  gcvar.dbg.nFreedInCycle;
  gcvar.dbgpersist.nBytesFreed += gcvar.dbg.nBytesFreedInCycle;
  gcvar.dbgpersist.nRefsFreed += gcvar.dbg.nRefsFreedInCycle;
  gcvar.dbgpersist.nNewObjectUpdates += gcvar.dbg.nNewObjectUpdatesInCycle;
  gcvar.dbgpersist.nOldObjectUpdates += gcvar.dbg.nOldObjectUpdatesInCycle;
  gcvar.dbgpersist.nStuckCounters += gcvar.dbg.nStuckCountersInCycle;



  dbgprn( 1, " __________  ACCUMULATING STATS _____________:\n");
  dbgprn( 1, "STORE: new=%d old=%d\n",
          gcvar.dbgpersist.nNewObjectUpdates,
          gcvar.dbgpersist.nOldObjectUpdates );
  dbgprn( 1, "UPDATE: updated=%d logged-slots=%d\n",
          gcvar.dbgpersist.nLoggedUpdates, gcvar.dbgpersist.nLoggedSlots );
  if (gcvar.dbgpersist.nObjectsAllocated) {
    avg = (float)gcvar.dbgpersist.nBytesAllocated / gcvar.dbgpersist.nObjectsAllocated;
    avgs = (float)gcvar.dbgpersist.nRefsAllocated / gcvar.dbgpersist.nObjectsAllocated;
  }
  else {
    avg = -1;
    avgs = -1;
  }
  dbgprn( 1, "CREATE: objects=%d bytes=%d avg=%f refs=%d avg=%f\n",
          gcvar.dbgpersist.nObjectsAllocated,
          gcvar.dbgpersist.nBytesAllocated,
          avg,
          gcvar.dbgpersist.nRefsAllocated,
          avgs );
  dbgprn(
         1,
         "RECLAIM: objects=%d bytes=%d\n",
         gcvar.dbgpersist.nObjectsFreed,
         gcvar.dbgpersist.nBytesFreed );
  dbgprn( 1, "STUCK: %d\n", gcvar.dbgpersist.nStuckCounters );
  {
    int nAllocated =  gcvar.dbgpersist.nBytesAllocated - gcvar.dbgpersist.nBytesFreed;
    int nFree = blkvar.heapSz - nAllocated;
    dbgprn( 1, "USAGE:   free=%10d   used= %10d\n", nFree, nAllocated );
  }
  blkPrintStats();
  dbgprn( 1, "PARTIAL: %d\n", chkCountPartialBlocks() );
}
#endif /* RCDEBUG */

GCFUNC void gcCheckGC(void)
{
  int nFreeBlocks = FREE_BLOCKS();
  if (nFreeBlocks < gcvar.gcTrigHigh)
    gcRequestAsyncGC();
}

static int _recommendCollectionMethod(void)
{
  int    nSamples, i, t, m;
  float  norm, avg[2], prob[2], r;

  if (gcvar.opt.recommendOnlyRCGC)
    return GCT_RCING;

  for (t=0; t<2; t++) {
    nSamples = 0;
    avg[t] = 0;
    for (i=0; i<N_SAMPLES; i++) {
      if (gcvar.runHist[t][i]) {
        avg[t] += gcvar.runHist[t][i];
        nSamples++;
      }
      else break;
    }
    avg[t] = nSamples ? avg[t]/nSamples : 0;
  }

  printf( "***  _recommendCollectionMethod trace=%f rc=%f\n",
          avg[GCT_TRACING], avg[GCT_RCING] );

  if (avg[GCT_TRACING] < 0.001) return GCT_TRACING;
  if (avg[GCT_RCING] < 0.001) return GCT_RCING;

  /*
   * Normalize so that prob ~ 1/avg
   * and prob[0]+prob[1] == 1
   */
  norm = (avg[0] * avg[1]) / ( avg[0] + avg[1] );
```

```
  prob[0] = norm / avg[0];
  prob[1] = norm / avg[1];

  printf( "p[0]=%f p[1]=%f sum=%f\n", prob[0], prob[1], prob[0]+prob[1] );
  r = (float)rand() / (float)RAND_MAX;

  if (r < prob[0]) m = 0;
  else m = 1;

  printf("r=%f --> m=%d\n", r , m );
  return m;
}

static void _updateRunHist(int runTime)
{
  int i;
  int t = gcvar.collectionType;

  for (i=N_SAMPLES-2; i>=0; i--)
    gcvar.runHist[t][i+1] =  gcvar.runHist[t][i];

  gcvar.runHist[t][0] = runTime;
}

static void _gc(void)
{
  uint  delta, end, start;
  int   nWasFree;

  start = GetTickCount();
  gcvar.gcActive = true;
  gcvar.collectionType = gcvar.nextCollectionType;
  gcvar.nextCollectionType = GCT_RCING;
  if (gcvar.usrSyncGC) {
    gcvar.collectionType = GCT_TRACING;
    gcvar.usrSyncGC = false;
  }
  if (gcvar.memStress) {
    gcvar.memStress = false;
    gcvar.collectionType = GCT_TRACING;
  }
  if (gcvar.opt.useOnlyTracingGC)
    gcvar.collectionType = GCT_TRACING;
  if (gcvar.opt.useOnlyRCGC)
    gcvar.collectionType = GCT_RCING;


  nWasFree = FREE_BLOCKS();

#ifdef RCVERBOSE
  jio_printf("----------------- start gc(%d--%s)  time=%d  -----\n",
             gcvar.iCollection,
             gcvar.collectionType == GCT_TRACING ? "TRACING" : "RC",
             start );
  fflush( stdout );
#endif
  _Initiate_Collection_Cycle();
  _Clear_Dirty_Marks();
  _Reinforce_Clearing_Conflict_Set();
  _Consolidate();
  if (gcvar.collectionType == GCT_RCING) {
    _Update_Reference_Counters( );
    _Reclaim_Garbage( );
  }
  else {
    _Trace();
    _Sweep();
    /* re-commit the "zct" bmp */
    mokMemCommit( gcvar.zctBmp.bmp, gcvar.zctBmp.bmp_size, true );
  }

  _processLocalsIntoNextZCT();
  gcvar.zctBuff = gcvar.nextZctBuff;
  gcvar.nextZctBuff.pos = NULL;

  end = GetTickCount();
  delta = end - start;

  _updateRunHist( delta );

#ifdef RCDEBUG
  if (gcvar.collectionType == GCT_RCING) {
    gcvar.dbgpersist.nPendInCycle = gcvar.nextZctBuff.start[LOG_OBJECTS_IDX];
    mokAssert( gcvar.dbg.nFreedInCycle == gcvar.dbg.nInZct + gcvar.dbg.nRecursiveDel );
  }
#endif //RCDEBUG

  /*
   * OK, now see where we stand and set the strategy for the
   * next cycle.
   */
  {
```

```
    int nNowFree, nLowMark;
    int prevTrig;
    bool  failed, gotIntoSync;

    nNowFree = FREE_BLOCKS();
    nLowMark = gcvar.gcTrigHigh + (gcvar.opt.lowTrigDelta * blkvar.nBlocks)/100;

    failed = nNowFree < nLowMark;
    gotIntoSync = gcvar.memStress;

    jio_printf("**** high=%d low=%d free=%d was=%d failed=%d sync=%d\n",
               gcvar.gcTrigHigh,
               nLowMark,
               nNowFree,
               nWasFree,
               failed,
               gotIntoSync
               );
    fflush( stdout );

    prevTrig = gcvar.gcTrigHigh;

    if (gcvar.collectionType == GCT_TRACING) {
      if (gotIntoSync && failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
        gcvar.gcTrigHigh -= (gcvar.opt.raiseTrigInc * blkvar.nBlocks)/100;
      }
      else if (gotIntoSync && !failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
        gcvar.gcTrigHigh += (gcvar.opt.lowerTrigDec * blkvar.nBlocks)/100;
      }
      else if (!gotIntoSync && failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
        gcvar.gcTrigHigh -= (gcvar.opt.raiseTrigInc * blkvar.nBlocks)/100;
      }
      else /* (!gotIntoSync && !failed) */ {
        gcvar.nextCollectionType =  _recommendCollectionMethod();
      }
    }
    else /*(gcvar.collectionType == GCT_RCING)*/ {
      if (gotIntoSync && failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
      }
      else if (gotIntoSync && !failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
      }
      else if (!gotIntoSync && failed) {
        gcvar.nextCollectionType =  GCT_TRACING;
      }
      else /* (!gotIntoSync && !failed) */ {
        gcvar.nextCollectionType =  _recommendCollectionMethod();
      }
    }

    jio_printf("**** prevTrig=%d currTrig=%d curCycle=%s nextCycle=%s\n",
               prevTrig,
               gcvar.gcTrigHigh,
               gcvar.collectionType == GCT_RCING ? "RC" : "TRACING",
               gcvar.nextCollectionType == GCT_RCING ? "RC" : "TRACING"
               );
    fflush( stdout );
  }


#ifdef RCDEBUG
  _printStats();
#endif


  gcvar.gcActive = false;

#ifdef RCVERBOSE
  jio_printf(
             "----------------- end gc(%d)  delta=%d ---------\n",
             gcvar.iCollection,
             end-start );
  fflush( stdout );
#endif
}

HANDLE hGCEvent, hMutEvent;

void gcThreadFunc(void *param)
{
  gcvar.ee = EE();
  gcvar.sys_thread = EE2SysThread ( gcvar.ee );

#ifdef RCDEBUG
  dbgprn(
         0,
         "GC Thread starting ... ee=%x sys_thread=%x\n",
         gcvar.ee,
```

```
        gcvar.sys_thread );
#endif
  gcvar.initialized = true;

  for(;;) {
    PulseEvent( hMutEvent );
#ifdef RCDEBUG
    dbgprn( 0, " ********** GC -- sleeping (%d)\n", gcvar.iCollection );
#endif
    WaitForSingleObject( hGCEvent, INFINITE );
#ifdef RCDEBUG
    jio_printf( " ********** GC -- wokeup (%d)\n", gcvar.iCollection );
    fflush( stdout );
#endif
    gcvar.nChunksAllocatedRecentlyByUser = 0;
    _gc();
#ifdef RCDEBUG
    dbgprn( 0, " ********** GC -- done (%d)\n", gcvar.iCollection );
#endif
    gcvar.iCollection++;
  }
}

/************************************************/
/********** USER REQUESTS ******************/
/************************************************/

GCEXPORT void gcRequestSyncGC(void)
{
  sys_thread_t *self = sysThreadSelf();
  int wasPhase = gcvar.iCollection;
  int waitT = 100;

#ifdef RCVERBOSE
  jio_printf("SYNC GC thread=%x (iCollection=%d) stress=%d\n",
             self,
             wasPhase,
             gcvar.memStress);
  fflush( stdout );
#endif
  gcvar.usrSyncGC = true;
  SetEvent( hGCEvent );
  while (wasPhase == gcvar.iCollection) {
    WaitForSingleObject( hMutEvent, waitT );
    waitT *= 2;
#ifdef RCDEBUG
    dbgprn( 0,
            "SYNC GC thread=%x GOT GC LOCK (iCollect=%d)\n",
            self,
            gcvar.iCollection );
#endif
  }
#ifdef RCDEBUG
  dbgprn( 0, "SYNC GC thread=%x DONE (iCollect=%d)\n", self,  gcvar.iCollection );
#endif
}

GCEXPORT void gcRequestAsyncGC(void)
{
  if (!gcvar.gcActive) {
    SetEvent( hGCEvent );
  }
}

/*------------------------ Init ----------------------------*/

static void gcInit(int __nMegs)
{
  DWORD HEAP_SIZE = __nMegs << 20;
  DWORD  ZCT_SIZE = HEAP_SIZE/0x100;

  FILE *f;

  DWORD TimeAdjustment;       // size of time adjustment
  DWORD TimeIncrement;        // time between adjustments
  BOOL  TimeAdjustmentDisabled; // disable option

  hGCEvent  = CreateEvent( NULL, FALSE, FALSE, NULL );
  hMutEvent = CreateEvent( NULL, FALSE, FALSE, NULL );

  GetSystemTimeAdjustment(
                          &TimeAdjustment,       // size of time adjustment
                          &TimeIncrement,        // time between adjustments
                          &TimeAdjustmentDisabled // disable option
                          );
#ifdef RCDEBUG
  dbgprn( 0, "TimeAdjustment=%d, TimeIncrement=%d, TimeAdjustmentDisabled=%d\n",
          TimeAdjustment,       // size of time adjustment
          TimeIncrement,        // time between adjustments
          TimeAdjustmentDisabled // disable option
          );
#endif
```

```
  f = fopen( "gcopt.txt", "r" );
  if (!f) {
    jio_printf( "GCOPT.txt could not be opened\n");
    exit(-1);
  }
  for (;;) {
    char buff[200];
    char opt[100];
    int  val;

    if (! fgets( buff, sizeof(buff), f) ) break;
    if (buff[0]=='#') continue; /* remark line */
    if (2 != sscanf( buff, "%s %d", opt, &val )) {
      jio_printf("Error reading GCOPT.TXT\n");
      exit(-1);
    }
#define CHECKGCOPT(optname) if (strcmp(opt, #optname)==0) {\
                              gcvar.opt. optname = val;\
                              jio_printf("GCOPT set: %s = %d\n", #optname, val);\
                              continue;\
                            } else do {} while(0)
    CHECKGCOPT(recommendOnlyRCGC);
    CHECKGCOPT(useOnlyTracingGC);
    CHECKGCOPT(useOnlyRCGC);
    CHECKGCOPT(listBlkWorth);
    CHECKGCOPT(userBuffTrig);
    CHECKGCOPT(initialHighTrigMark);
    CHECKGCOPT(lowTrigDelta);
    CHECKGCOPT(raiseTrigInc);
    CHECKGCOPT(lowerTrigDec);
    CHECKGCOPT(uniPrio);
    CHECKGCOPT(multiPrio);
    jio_printf("GCOPT unknown option %s\n", opt );
    exit(-1);
  }
  fclose( f );

  /* Init blocks manager */
  blkInit( HEAP_SIZE >> 20 );

  /* Init chunks manager */
  chkInit( HEAP_SIZE >> 20 );

  gcvar.stage = GCHS4;
  gcvar.createBuffList = NULL;
  gcvar.updateBuffList = NULL;
  gcvar.snoopBuffList = NULL;
  gcvar.deadThreadsCreateBuffList = NULL;
  gcvar.deadThreadsUpdateBuffList = NULL;
  gcvar.deadThreadsSnoopBuffList = NULL;
  gcvar.reinforceBuffList = NULL;

  gcvar.tempReplicaSpace = (GCHandle**)mokMemReserve( NULL, BUFFSIZE );
  mokMemCommit( (char*)gcvar.tempReplicaSpace, BUFFSIZE, false );

  gcvar.zctStack = (GCHandle**)mokMemReserve( NULL, ZCT_SIZE );
  mokMemCommit( (char*)gcvar.zctStack, ZCT_SIZE, false );
  gcvar.zctStackTop = (GCHandle**)(ZCT_SIZE + (char*)gcvar.zctStack);
  gcvar.zctStackSp = gcvar.zctStack;

  H1BIT_Init( &gcvar.localsBmp, (uint*)blkvar.heapStart, HEAP_SIZE );
  H2BIT_Init( &gcvar.rcBmp, (uint*)blkvar.heapStart, HEAP_SIZE );
  H1BIT_Init( &gcvar.zctBmp, (uint*)blkvar.heapStart, HEAP_SIZE );

  buffInit( gcvar.ee, &gcvar.zctBuff );

  gcvar.gcMon = (sys_mon_t*)sysMalloc(sysMonitorSizeof());
  gcvar.requesterMon = (sys_mon_t*)sysMalloc(sysMonitorSizeof());

  sysMonitorInit(  gcvar.gcMon );
  sysMonitorInit(  gcvar.requesterMon );

  gcvar.collectionType = GCT_RCING;

  gcvar.gcTrigHigh = (gcvar.opt.initialHighTrigMark * blkvar.nBlocks)/100;
}

GCEXPORT void gcStartGCThread(void)
{
  int priority;

  /*
   * If we're on an MP then the GC thread should be alloted a processor
   * of its own when it needs it.  So we select the priority to be
   * 10 which is translated in threads_md.c into win32 time critical
   * priority.
   *
   * Otherwise, we choose priority==9 which translates into win32
   * "highest priority"
   */
  if (sysGetSysInfo()->isMP)
```

```
    priority = gcvar.opt.multiPrio;
  else
    priority = gcvar.opt.uniPrio;
  createSystemThread("YLRC Garbage Collector (YEH!)", 9, 10*1024, gcThreadFunc, NULL);
}

GCEXPORT void gcThreadCooperate(ExecEnv *ee)
{
  int gcStage;

  mokAssert( !ee->gcblk.cantCoop );

  ee->gcblk.cantCoop = true;
  gcStage = gcvar.stage;
  if (ee->gcblk.stage == gcStage) goto __exit;
  if (ee->gcblk.stageCooperated == gcStage) goto __exit;
  mokAssert( ee->gcblk.stageCooperated == GCHSNONE );
  switch (gcStage) {
  case GCHS1:
    mokAssert( ee->gcblk.stage == GCHS4 );
    goto __exit;

  case GCHS2:
    mokAssert( ee->gcblk.stage == GCHS1 );
    goto __exit;

  case GCHS3:
    mokAssert( ee->gcblk.stage == GCHS2 );
    _HS3Cooperate( ee );
    goto __exit;

  case GCHS4:
    mokAssert( ee->gcblk.stage == GCHS3 );
    goto __exit;
  }

 __exit:
  ee->gcblk.cantCoop = false;
}


GCEXPORT void gcThreadAttach(ExecEnv* ee)
{
  int i, stage;
  sys_thread_t *self = EE2SysThread( ee );

#ifdef RCDEBUG
  dbgprn( 0, "gcThreadAttach starting for ee=%x thread=%x\n", ee, self);
#endif

  ee->gcblk.cantCoop = false;

  buffInit( ee, &ee->gcblk.updateBuffer );
  buffInit( ee, &ee->gcblk.createBuffer );
  buffInit( ee, &ee->gcblk.snoopBuffer );

#ifdef RCDEBUG
  dbgprn( 2, "QUEUE_LOCK %x\n", self );
#endif
  QUEUE_LOCK( self );
#ifdef RCDEBUG
  dbgprn( 2, "QUEUE_LOCK %x took the lock\n", self );
#endif

  {
    SAVEDALLOCLISTS *sal = gcvar.pListOfSavedAllocLists;

    if (sal) {
      gcvar.pListOfSavedAllocLists = sal->pNext;
      memcpy( ee->gcblk.allocLists, sal->allocLists, sizeof(sal->allocLists) );
      sysFree( sal );
    }
    else {
      for (i=0; i<N_BINS; i++) {
        ee->gcblk.allocLists[i].binIdx = i;
        ee->gcblk.allocLists[i].head = ALLOC_LIST_NULL;
      }
    }
  }

  stage = gcvar.stage;
  ee->gcblk.stageCooperated = GCHSNONE;
  ee->gcblk.stage = stage;
  if (ee->gcblk.stage != GCHS4)
    ee->gcblk.snoop = true;
  else
    ee->gcblk.snoop = false;
  ee->gcblk.gcInited = true;
  QUEUE_UNLOCK( self );
#ifdef RCDEBUG
  dbgprn( 0, "gcThreadAttach ee=%x stage=%d\n", ee, stage);
  dbgprn( 0, "gcThreadAttach ended for ee=%x self=%x\n", ee, self);
```

```
#endif
}

GCEXPORT void gcThreadDetach(ExecEnv* ee)
{
  sys_thread_t *self = EE2SysThread( ee );
  SAVEDALLOCLISTS *sal;

  sal = (SAVEDALLOCLISTS*)sysMalloc(  sizeof(SAVEDALLOCLISTS) );

  mokAssert( sizeof(sal->allocLists) == sizeof(ee->gcblk.allocLists) );
  mokAssert( sizeof(sal->allocLists) == sizeof(ALLOCLIST)*N_BINS );

  memcpy( sal->allocLists, ee->gcblk.allocLists, sizeof( ee->gcblk.allocLists) );

  QUEUE_LOCK( self );

  sal->pNext = gcvar.pListOfSavedAllocLists;
  gcvar.pListOfSavedAllocLists = sal;

#ifdef RCDEBUG
  gcvar.dbgpersist.nDeadUpdateObjects +=
    ee->gcblk.updateBuffer.start[LOG_OBJECTS_IDX];
  gcvar.dbgpersist.nDeadUpdateChilds  +=
    ee->gcblk.updateBuffer.start[LOG_CHILDS_IDX];
  gcvar.dbgpersist.nDeadCreateObjects +=
    ee->gcblk.createBuffer.start[LOG_OBJECTS_IDX];
  gcvar.dbgpersist.nDeadSnooped       +=
    ee->gcblk.snoopBuffer.start[LOG_OBJECTS_IDX];
#endif

  /* link the create buffer into a list for dead threads */
  *ee->gcblk.createBuffer.pos = 0;
  ee->gcblk.createBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.createBuffer.pos;
  ee->gcblk.createBuffer.start[LINKED_LIST_IDX] =
    (uint)gcvar.deadThreadsCreateBuffList;
  gcvar.deadThreadsCreateBuffList = ee->gcblk.createBuffer.start;

  /* do the same for the update buffer */
  *ee->gcblk.updateBuffer.pos = 0;
  ee->gcblk.updateBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.updateBuffer.pos;
  ee->gcblk.updateBuffer.start[LINKED_LIST_IDX] =
    (uint)gcvar.deadThreadsUpdateBuffList;
  gcvar.deadThreadsUpdateBuffList = ee->gcblk.updateBuffer.start;

  /* do the same for the snoop buffer */
  *ee->gcblk.snoopBuffer.pos = 0;
  ee->gcblk.snoopBuffer.start[LAST_POS_IDX] = (uint)ee->gcblk.snoopBuffer.pos;
  ee->gcblk.snoopBuffer.start[LINKED_LIST_IDX] = (uint)gcvar.deadThreadsSnoopBuffList;
  gcvar.deadThreadsSnoopBuffList = ee->gcblk.snoopBuffer.start;


  /* If we're between HS1 & HS2 then also link the update buffer
   * into the dead threads reinforce list
   */
  if (ee->gcblk.stage == GCHS1) {
#ifdef RCDEBUG
    gcvar.dbgpersist.nDeadReinforceObjects +=
      ee->gcblk.updateBuffer.start[LOG_OBJECTS_IDX];
    gcvar.dbgpersist.nDeadReinforceChilds  +=
      ee->gcblk.updateBuffer.start[LOG_CHILDS_IDX];
#endif
    ee->gcblk.updateBuffer.start[REINFORCE_LINKED_LIST_IDX] =
      (uint)gcvar.deadThreadsReinforceBuffList;
    gcvar.deadThreadsReinforceBuffList = ee->gcblk.updateBuffer.start;
  }

  ee->gcblk.gcInited = false;

  QUEUE_UNLOCK( self );
}


void gcDo_gcupdate(ExecEnv *ee, void *_h, void *_slot, void *_newval )
{
#ifdef RCDEBUG
  static int deltaMax = -1;
  int delta = GetTickCount();
#endif

  GCHandle *h = (GCHandle*)_h;
  GCHandle **slot = (GCHandle**)_slot;
  GCHandle *newval = (GCHandle*)_newval;

#ifdef RCDEBUG
  sysAssert( h );
  sysAssert( ValidHandle(h) );
  sysAssert( !*slot || ValidHandle(*slot) );
  sysAssert( !newval || ValidHandle(newval) );

  {
    uint *p = h->logPos;
```

```
    if (p) {
      uint val = *p;
      uint type = val&3;
      sysAssert( (val&~3) == (uint)h );
      if (type==0) { // create log
        ee->gcblk.dbg.nNewObjectUpdatesInCycle++;
      }
      else {
        ee->gcblk.dbg.nOldObjectUpdatesInCycle++;
      }
    }
  }
#endif // RCDEBUG

  ee->gcblk.cantCoop = true;
  if (!h->logPos) {
    gcBuffSlowConditionalLogHandle( ee, (GCHandle*)h );
  }
  *slot = newval;
  if (newval && ee->gcblk.snoop) {
    BUFFHDR *bh = &ee->gcblk.snoopBuffer;
    gcBuffLogWordUnchecked( ee, bh, (uint)newval );
    ee->gcblk.cantCoop = false;
    gcBuffReserveWord( ee, bh );
  }
  else {
    ee->gcblk.cantCoop = false;
  }

#ifdef RCDEBUG
  delta = GetTickCount() - delta;
  if (delta > deltaMax) {
    deltaMax = delta;
    dbgprn( 0, " *** UPDATE(offset=%d) delta=%d\n", (char*)slot - (char*)h, delta );
  }
#endif
}

void gcDo_gcupdate_array(ExecEnv *ee, void *_arrayh, void* _slot, void *_newval )
{
  gcupdate( ee, _arrayh, _slot, _newval );
}

void gcDo_gcupdate_jvmglobal(ExecEnv* ee, void* _global, void *_newval )
{
#ifdef RCDEBUG
  static int deltaMax = -1;
  int delta = GetTickCount();
#endif

  GCHandle **slot = (GCHandle**)_global;
  GCHandle *newval = (GCHandle*)_newval;
  sysAssert( !newval || ValidHandle(newval) );

  ee->gcblk.cantCoop = true;
  *slot = newval;
  if (newval && ee->gcblk.snoop) {
    BUFFHDR *bh = &ee->gcblk.snoopBuffer;
    gcBuffLogWordUnchecked( ee, bh, (uint)newval );
    ee->gcblk.cantCoop = false;
    gcBuffReserveWord( ee, bh );
  }
  else {
    ee->gcblk.cantCoop = false;
  }

#ifdef RCDEBUG
  delta = GetTickCount() - delta;
  if (delta > deltaMax) {
    deltaMax = delta;
    dbgprn( 0, " *** UPD_GLOBAL delta=%d\n", delta );
  }
#endif
}

void gcDo_gcupdate_class(ExecEnv*  ee, ClassClass* cb, void *_slot, void *_newval )
{
  GCHandle **slot = (GCHandle**)_slot;

  sysAssert( ValidHandle(cb) );
  sysAssert( !*slot || ValidHandle(*slot) );

  gcupdate_jvmglobal( ee, slot, _newval );
}

void gcDo_gcupdate_static(
  ExecEnv* ee,
  struct fieldblock* fb,
  void *_slot,
  void* _newval
)
{
```

```
  GCHandle **slot = (GCHandle**)_slot;
  char isig = fieldsig(fb)[0];
  if (isig == SIGNATURE_CLASS || isig == SIGNATURE_ARRAY) {
    sysAssert( !*slot || ValidHandle(*slot) );
    gcupdate_jvmglobal( ee, slot, _newval );
  }
  else {
    *slot = (GCHandle*)_newval;
  }
}

GCEXPORT void gcPutstatic(ExecEnv *ee, struct fieldblock *fb, JHandle *val)
{
  sysAssert( fb );
  sysAssert( ValidHandle(fb->clazz) );
  gcupdate_static( ee, fb, &fb->u.static_value, val );
}

GCEXPORT void gcPutfield(ExecEnv *ee, JHandle *h, int offset, JHandle *val)
{
  Classjava_lang_Class *ucb;
  JHandle **slot;
  GCHandle *_h;

#ifdef RCDEBUG
  {
    Classjava_lang_Class *ucb;

    mokAssert( h );
    mokAssert( isHandle(h) );

    ucb = unhand(obj_classblock(h));

    mokAssert( ucb->is_reference[offset] );
    mokAssert( !val || isHandle(val) );
  }
#endif

  slot = (JHandle**)(((uint*)unhand(h)) + offset);
  gcupdate( ee, h, slot, val );
}


GCEXPORT void gcAastore(ExecEnv *ee, ClassArrayOfObject *arr, int offset, JHandle *val)
{
  JHandle **slot;
  JHandle *arrh;
#ifdef RCDEBUG
  ClassClass *cb;
  long n;
#endif


  arrh = gcRehand( arr );

#ifdef RCDEBUG
  mokAssert( arr );
  mokAssert( arrh );
  mokAssert( isHandle(arrh) );
#endif

  slot = &arr->body[offset];

#ifdef RCDEBUG
  mokAssert( !*slot || isHandle(*slot) );
  mokAssert( !val || isHandle(val) );

  mokAssert( obj_flags(arrh) == T_CLASS );

  n = obj_length(arrh);

  mokAssert( offset < n );
  mokAssert( offset >=0 );

  cb = (ClassClass*)arr->body[n];

  mokAssert( cb );
  mokAssert( isHandle(cb) );
#endif

  gcupdate_array( ee, arrh, slot, val );
}
```

End of file source listing

D.6 rcgc.h

rcgc.h contains declarations and macros which are needed by the rest of the JVM. In particular, it defines the GC blocks which are associated with threads, layout of objects and page headers and the definition of frequently used functions that were turned into macros.

Source listing for file rcgc.h

```
/*
 * File:    rcgc.h
 * Author:  Mr. Yossi Levanoni
 * Purpose: Publicly visible interface to garbage collection and allocation.
 */
/****************** Initialization ******************************/
#ifndef __RCGC__
#define __RCGC__

#include <assert.h>
#include <stdio.h>
#include <windows.h>

#include "monitor.h"

//#ifdef DEBUG
#define RCDEBUG
//#endif

#define RCVERBOSE

#define RCNOINLINE

#define GCEXPORT
#define GCFUNC static

#ifdef RCDEBUG
#define RCDEBUGVAR 1
#else
#define RCDEBUGVAR 0
#endif

/*****************************************************************************
*
* Forward declarations for external structures
*/
#define DECSTRUCT(T) struct T; typedef struct T T;

DECSTRUCT(BUFFHDR);
struct execenv;
typedef struct execenv ExecEnv;
typedef bool_t bool;

typedef struct GCHandle {
  unsigned  *obj;
  struct methodtable *methods;
  unsigned *logPos;
#ifdef RCDEBUG
  unsigned status;
#endif
} GCHandle ;

#define false FALSE
#define true  TRUE

/*****************************************************************************
*
* Atomic operatrions
*
*
*/
#define N_SPINS 4000

/*****************************************************************
*
* Some primitive data structures.
*/
typedef unsigned         word;
typedef unsigned         uint;
typedef unsigned char    byte;
typedef unsigned short   PAGEID;
typedef unsigned short   PAGECNT;

/*****************************************************************
*
* An object (chunk of memory) as the chunk manager sees it.
*/

typedef struct BLKOBJtag BLKOBJ;

struct BLKOBJtag {
```

```
  int      count;
  int      unused;
  BLKOBJ   *next;
};

/*****************************************************************
*
* Object and page sizes.
*
* We assume that objects are at least 8 bytes aligned.  This leaves 3
* bits for playing.
*
* The minimal object size is 16 bytes because we have (at least) two
* words overhead per object: class pointer and log pointer.  In the
* "handled" JVM we have a third extra poiner.
*/
#define OBJGRAIN  8
#define OBJBITS   3
#define MINOBJ    16
#define OBJMASK   (~(OBJGRAIN-1))
/*
 * Minimal size of a page for the design to work: 256 bytes.
 * The reason for this is that we sometimes (in BLKLIST blocks) keep
 * a block identifier as a 24 bit entity.  Thus, a block has to be at
 * least 8 bits wide in order to allow 4GB regions.
 *
 * Since in pracrice we use blocks which are at least 4KB big, this is
 * not a problem.
 *
 * Additionally, we store the size of chunks in a block on a 16 bit
 * entity.  Thus, a block cannot be much bigger than 64KB or we'll
 * have to encode this field etc.
 */
#define MINBLOCKBITS        8
#define MAXNONBLOCKBITS     (32-MINBLOCKBITDS)

#define MAXBLOCKBITS        16


#define MAXOBJPERBLOCK      (BLOCKSIZE/MINOBJ)

/*
 * Actual block size.  This coincides with the PC page size.
 */
#define BLOCKBITS           (14)
#define NONBLOCKBITS        (32-BLOCKBITS)
#define BLOCKSIZE           (1<<BLOCKBITS)
#define BLOCKMASK           ((1<<BLOCKBITS)-1)

/*
* Size of maximal chunk.  Allocations larger than this size
* are given full blocks.
*/
#define MAX_CHUNK_ALLOC     (BLOCKSIZE/2)

/* address of first object on the block */
#define OBJPAGE(o)       ((OBJECT*)(((unsigned)o) & (~BLOCKMASK)))

/* offset of object in the block */
#define OBJOFFSET(o)     (((unsigned)(o)) & BLOCKMASK)

/* number of block relative to address 0 */
#define OBJBLOCKID(o)    (((unsigned)(o))>>BLOCKBITS)

/* Object's block header */
#define OBJBLOCKHDR(o)   (&blkvar.blockHeaders[ OBJBLOCKID(o)])

/* convert from block header to the block's address */
#define BLOCKHDROBJ(ph)  ((BLKOBJ*)(((ph)-blkvar.blockHeaders)<<BLOCKBITS))


/*****************************************************************************/
/*****************************************************************************/
/******                                                                 *****/
/******                        BLOCK MANAGER                            *****/
/******                                                                 *****/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************
 */

/*
* Page States
*/

#define   BLK           1 /* In the block manager */
#define   BLKLIST       2 /* ---   " " -----------*/
#define   CHUNKING      3 /* Just out of the block manager, going to be OWNED */
#define   ALLOCBIG      4 /* Multiple-blocks object */
#define   INTERNALBIG   5 /* In the middle of ALLOCBIG, only in DEBUG */
#define   OWNED         6 /* Chunked block which is owned by some thread */
#define   VOIDBLK       7 /* Chunked block, allocation exhausted. */
```

```
#define   PARTIAL       8 /* Chunked block, sitting in a partial blocks list */
#define   DUMMYBLK      9 /* Temporary state */

#define   LASTMGRSTATE  BLKLIST

/*
Page header format for: OWNED, VOIDPG, PARTIAL.

Word 0:  <---------------------- nextPartial(32) -------------------------->
Word 1:  <---------------------- prevPartal(32) -------------------------->
Word 2:  <---------------------- freeList(32) -------------------------->
Word 3:  <-- status(8) --><-- lock(8) --><-------- binidx(16)  --------->

In this case, the second word in the object pointed by "freeList"
contains the number of objects in the list.  recycledList is cached
(see below), the number of elements is held in the same manner at the
second word of the first element of the list.


&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

Page header format for ALLOCBIG:

Word 0:  <---------------------- AllocInProgress(32) ------------------->
Word 1:  <---------------------- unused(32) ---------------------------->
Word 2:  <---------------------- size(32) ---------------------------->
Word 3:  <-- status(8) --><---------------- unused(24) ----------------->

"AllocInProgress" is true in the interval between the changing of the
state from BLKxxx to ALLOCBIG till the object is logged in the allocating
thread create log.  This prevents sweep from reclaiming such an object
just after it has been allocated.

"size" is the size of this large object, in blocks.


&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

Page header format for INTERNALBIG:

Word 0:  <---------------------- startBlock(32) ------------------------->
Word 1:  <---------------------- unused(32) ---------------------------->
Word 2:  <---------------------- unused(32) ---------------------------->
Word 3:  <-- status(8) --><--------------unused(24) --------------------->

Where "start page" is the address where this large object begins.

THIS FORMAT IS GUARANTEED ONLY IN DEBUG MODE.


&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

  Page header format for BLK:

Word 0:  <---------------------- nextRegion(32) ------------------------->
Word 1:  <---------------------- prevRegion(32) ------------------------->
Word 2:  <---------------------- size(32) ---------------------------->
Word 3:  <-- status(8) --><--------------unused(24) --------------------->



Next and prev are linked list pointers.  size is the size in pages of the
regions.


&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&


  Page header format for BLKLIST:

Word 0:  <---------------------- firstRegion ---------------------------->
Word 1:  <---------------------- nextList (32) ------------------------->
Word 2:  <---------------------- size (32) ---------------------------->
Word 3:  <-- status(8) --><------------ prevListIDX (24)  -------------->


"firstRegion" is a pointer to a BLK block, the first on a linked list
of regions with the same size.

"nextList" points to the next list header (of type BLKLIST).  The pointer
to the previous list is encoded in the field "prevListIDX" as an index
into the allocatedPageHeaders array.

"size" is the size of the region.  Each element in the list has this size.

**************************************************************************/


/*
* Field selectors
*/
#define STATUSMAK     0xff000000
```

```
#define LOCKMASK      0x00ff0000
#define BINIDXMASK    0x0000ffff
#define PREVLISTMASK  0x00ffffff


typedef struct BlkAllocHdrTAG          BlkAllocHdr;
typedef struct BlkAllocBigHdrTAG       BlkAllocBigHdr;
typedef struct BlkAllocInternalHdrTAG  BlkAllocInternalHdr;
typedef struct BlkRegionHdrTAG         BlkRegionHdr;
typedef struct BlkListHdrTAG           BlkListHdr;
typedef struct BlkAnyHdrTAG            BlkAnyHdr;



struct BlkAllocHdrTAG {
  BlkAllocHdr   *nextPartial;
  BlkAllocHdr   *prevPartial;
  volatile  BLKOBJ        *freeList;
  volatile  word          StatusLockBinidx;
};

struct BlkAllocBigHdrTAG {
  volatile  word   allocInProgress;
  word   unused2;
  volatile  int    blobSize;
  volatile  word   StatusUnused;
};

struct BlkAllocInternalHdrTAG {
  BlkAllocBigHdr   *startBlock;
  word             unused1;
  word             unused2;
  volatile  word             StatusUnused;
};

struct BlkListHdrTAG {
  BlkRegionHdr    *nextRegion;
  BlkListHdr      *nextList;
  volatile  int             listRegionSize;
  volatile  word            StatusPrevListID;
};

struct BlkRegionHdrTAG {
  BlkRegionHdr   *nextRegion;
  BlkRegionHdr   *prevRegion;
  volatile  int            regionSize;
  volatile  word           StatusUnused;
};

struct BlkAnyHdrTAG {
  volatile word w0;
  volatile word w1;
  volatile word w2;
  volatile union {
    volatile byte b[4];
    volatile unsigned short s[2];
    volatile word w;
  } u;
};


/*
 * Utility macros
 */

/*
 * p is a pointer to AllocPgHdr.  Set and get the chunk size
 */
#define bhGet_bin_idx(p)      ((int)(((p)->StatusLockBinidx)&BINIDXMASK))
#define bhSet_bin_idx(p,idx)  do {\
  word v; \
  mokAssert( (idx)< N_BINS ); \
  v = p->StatusLockBinidx; \
  v = v & ~BINIDXMASK; \
  v = v | idx; \
  p->StatusLockBinidx = v; \
} while(0)


/*
 * p is a pointer to BlkRegionHdr. Set and get the previous list IS.
 */
#define bhGet_prev_region_list(p)  \
    ((BlkListHdr*)&blkvar.allocatedBlockHeaders[(p)->StatusPrevListID & PREVLISTMASK])

#define bhSet_prev_region_list(p,pBlkListHeader)  \
do {\
  word idx; \
  word v; \
  idx = (pBlkListHeader) - (BlkListHdr*)blkvar.allocatedBlockHeaders; \
  mokAssert (idx < (word)(blkvar.nBlocks+2)); \
  v = p->StatusPrevListID; \
```

```
  v = v & ~PREVLISTMASK; \
  v = v | idx; \
  (p)->StatusPrevListID = v; \
} while(0)



/*
 * Set and get the status of any page
 */
#define bhGet_status(p)          (((BlkAnyHdr*)p)->u.b[3])
#define bhSet_status(p,s)        do{ bhGet_status(p)=(s); }while(0)


/************************************************************************
*
* Block manager structure
*
*/
#define N_QUICK_BLK_MGR_LISTS    5

struct BLKVAR {
  BlkListHdr*    pRegionLists;
  BlkRegionHdr*  quickLists[ N_QUICK_BLK_MGR_LISTS ];
  byte*          heapStart;
  byte*          heapTop;
  BlkRegionHdr*  heapTopRegion;
  BlkRegionHdr*  wildernessRegion;
  word           heapSz;
  word           nBlocks;
  BlkAllocHdr    *blockHeaders;
  BlkAllocHdr*   allocatedBlockHeaders;
  sys_mon_t*     blkMgrMon;
  int            nWildernessBlocks;
  int            nListsBlocks;
  int            nAllocatedBlocks;
};


#define FREE_BLOCKS() \
  (((blkvar.nListsBlocks*gcvar.opt.listBlkWorth)/100)+blkvar.nWildernessBlocks)

/************************************************************************
 * Block manager exports
 */
GCEXPORT  BlkAllocBigHdr*   blkAllocRegion( unsigned nBytes, ExecEnv *ee );


/************************************************************************/
/************************************************************************/
/******                                                          *****/
/******                   CHUNK MANAGEMENT                       *****/
/******                                                          *****/
/************************************************************************/
/************************************************************************/
/************************************************************************/

/************************************************************************
*
* Recycled lists cache.
*
* The cache is simply an array of pointers to blocks.  The blocks are
* linked in a circular list with the first element holding the number
* of elements in the list.
*
* Collisions are treated by flushing an entry.  Meaning: adding the
* list to the block's free list.
*/

/*
* this ration defines the number of blocks per recycled lists cache
* entry.
*/
#define RLCACHE_RATIO               10

typedef struct RLCacheEnteryTAG RLCENTRY;

struct RLCacheEnteryTAG {
  BLKOBJ       *recycledList;
};

/************************************************************************
*
* Partial Lists to Block Manager evacuation thresholds.
*
*/
#define MAX_OBSERVED_FULL_PER_LIST  2
#define MAX_OBSERVED_FULL           4

/************************************************************************
*
* Allocation lists
```

```
*
* These structures are embedded in the threads EE for fast allocation.
* Each thread has an allocation list per bin size.
*
*/

typedef struct AllocListTAG ALLOCLIST;

#define ALLOC_LIST_NULL ((BLKOBJ*)0x12baab21)

struct AllocListTAG {
  BLKOBJ*        head;
  BlkAllocHdr*   allocBlock;
  int            binIdx;
};


#define OutOfMemory()   mokAssert(0)
#define ALLOC_RETRY     (20)


/************************************************************************
*
* Bins conversion tables.
*
*/
#define N_BINS  (27)

struct CHKCONV {
  int szToBinIdx[ BLOCKSIZE ];
  int szToBinSize[ BLOCKSIZE ];
  int binSize[ N_BINS ];
  int binToObjectsPerBlock[ N_BINS ];
};


/************************************************************************
*
* Partial lists.
*
* A partial list is a list of blocks which have some free chunks on them.  The
* pages are linked in a doubly linked list whose head is in this structure.
*
* There is a list per each bin size.
*
* The list also contains a remembered set of blocks which have been observed to
* be full.
*
* Finally the list contains a lock and therefore it is padded to a total size
* of 256 bytes (assuming this is bigger or equal to the contention granule)
* in order to prevent false sharing with other partial lists.
*/
struct PARTIALLISTtag {
  BlkAllocHdr  *firstBlock;
  word         lock;
  int          nObservedFull;
  BlkAllocHdr  *observedFull[ MAX_OBSERVED_FULL_PER_LIST ];
  word         pad[64 - (MAX_OBSERVED_FULL_PER_LIST +3) ];
};


typedef struct PARTIALLISTtag PARTIALLIST;


/************************************************************************
*
* Chunk manager structure.
*
*/
struct CHUNKVAR {
  PARTIALLIST   partialLists[ N_BINS ];
  int           nBlocksInPartialList[ N_BINS ];
  int           nCacheEntries;
  RLCENTRY      *rlCache;
  int           nObservedFull;
  int           nTrulyFull;
  BlkAllocHdr*  trulyFull[ MAX_OBSERVED_FULL ];
};


/************************************************************************
*
* Chunk Manager exports
*
*/
GCEXPORT  int      chkCountPartialBlocks(void);
GCEXPORT  BLKOBJ*  chkAllocSmall(ExecEnv* ee, unsigned binIdx);
GCEXPORT  void     chkReleaseAllocLists( ExecEnv *ee);

#ifndef RCDEBUG

#define chkPreCollect(__o) \
```

```
do{\
  word blockid;\
  RLCENTRY *rlce;\
  BLKOBJ *head;\
  BLKOBJ *o = (BLKOBJ*)(__o);\
\
  blockid = OBJBLOCKID(o);\
  rlce = &chunkvar.rlCache[blockid % chunkvar.nCacheEntries];\
  head = rlce->recycledList;\
        \
  if ((((word)head) ^ ((word)o)) < BLOCKSIZE) {\
    o->next  = head->next;\
    head->next = o;\
    head->count ++;\
    goto __chkPreCollect_done_;\
  }\
  if (head) \
    chkFlushRecycledListEntry( rlce );\
        \
  o->count = 1;\
  o->next  = o;\
  rlce->recycledList = o;\
__chkPreCollect_done_:;\
} while(0)

#define _allocFromOwnedBlockInlined( allocList, __res )\
do {\
   BLKOBJ *head = allocList->head;\
   if (head != ALLOC_LIST_NULL) {\
      allocList->head = head->next;\
      (BLKOBJ*)__res = head;\
   }\
   else {\
      __res = NULL;\
   } \
} while (0)


#define chkAllocSmallInlined(ee, binIdx, __res)\
do {\
   ALLOCLIST *allocList = & (ee)->gcblk.allocLists[ (binIdx) ];\
   _allocFromOwnedBlockInlined( allocList, __res);\
   if (!__res) {\
      (BLKOBJ*)__res = chkAllocSmall( ee, binIdx);\
   }\
} while (0)

#else /* RCDEBUG */

#define chkAllocSmallInlined( ee, binIdx, __res)\
do {\
   (BLKOBJ*)__res = chkAllocSmall( ee, binIdx );\
} while(0)

#endif /* ! RCDEBUG */


/*************************************************************************/
/*************************************************************************/
/******                                                           *****/
/******                  BITMAPS                                  *****/
/******                                                           *****/
/*************************************************************************/
/*************************************************************************/
/*************************************************************************/


/************************************************************************
*
* 1 Bit per handle BMP
*/
typedef struct H1BIT_BMP H1BIT_BMP;
struct H1BIT_BMP {
  byte *entry;
  byte *bmp;
  byte *rep_addr;
  unsigned bmp_size;
};

/************************************************************************
*
* 2 Bits per handle BMP
*/
typedef struct H2BIT_BMP H2BIT_BMP;
struct H2BIT_BMP {
  byte *entry;
  byte *bmp;
  byte *rep_addr;
  unsigned bmp_size;
};

/*
```

```
*
* Include inline vertions of bmp functions:

void H1BIT_Set(byte* entry, unsigned h);
void H1BIT_Clear(byte* entry, unsigned h);
void H1BIT_Put(byte* entry, unsigned h, unsigned val);
byte H1BIT_Get(byte* entry, unsigned h);
void H1BIT_Init(H1BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size );

void H2BIT_Put(byte* entry, unsigned h, unsigned val);
void H2BIT_Clear(byte* entry, unsigned h);
void H2BIT_Stuck(byte* entry, unsigned h);
byte H2BIT_Get(byte* entry, unsigned h);
void H2BIT_Inc(byte* entry, unsigned h);
byte H2BIT_IncRV(byte* entry, unsigned h);
byte H2BIT_Dec(byte* entry, unsigned h);
void H2BIT_Init(H2BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size );

Functions that have a return value have "Inlined" appended to their name
e.g H1BIT_GetInlined( entry, h, __res_var) where __res_var is the *name*
of the variable onto which the result should be stored.
*/

#ifdef RCNOINLINE

#define H1BIT_GetInlined( entry, h, __res_var)\
do {\
        __res_var = H1BIT_Get(entry, h );\
} while (0)

#define H2BIT_GetInlined( entry, h, __res_var)\
do {\
        __res_var = H2BIT_Get(entry, h );\
} while (0)

#define H2BIT_IncRVInlined( entry, h, __res_var)\
do {\
        __res_var = H2BIT_IncRV(entry, h );\
} while (0)

#define H2BIT_DecInlined( entry, h, __res_var)\
do {\
        __res_var = H2BIT_Dec(entry, h );\
} while (0)


#else /* ! RCNOINLINE */

#include "rcbmp_inline.h"

#endif /*  RCNOINLINE */


/*****************************************************************************/
/*****************************************************************************/
/******                                                                 *****/
/******                  GC Data Structures                             *****/
/******                                                                 *****/
/*****************************************************************************/
/*****************************************************************************/
/*****************************************************************************/


/*************************************************************************
*
* Buffer mgmnt.
*
*/
#define BUFFBITS   18
#define BUFFSIZE   (1<<BUFFBITS)
#define BUFFMASK   (BUFFSIZE-1)
#define LOWBUFFMASK ((1<<16)-1)

#define BUFF_LINK_MARK            1U
#define BUFF_HANDLE_MARK          2U
#define BUFF_DUP_HANDLE_MARK      3U

#ifdef RCDEBUG
#define N_RESERVED_SLOTS 8
#else
#define N_RESERVED_SLOTS 4
#endif //RCDEBUG

#define LINKED_LIST_IDX            0
#define REINFORCE_LINKED_LIST_IDX  1
#define NEXT_BUFF_IDX              2
#define LAST_POS_IDX               3
#ifdef RCDEBUG
#define ALLOCATING_EE              4
#define LOG_CHILDS_IDX             5
#define LOG_OBJECTS_IDX            6
#define USED_IDX                   7
```

```
#endif

typedef struct BUFFHDR BUFFHDR;
struct BUFFHDR {
  uint *pos;
  uint *limit;
  uint *start;
  uint *currBuff;
};

GCEXPORT void gcBuffConditionalLogHandle(ExecEnv *ee, GCHandle *h);
GCEXPORT void gcBuffLogWord(ExecEnv *ee, BUFFHDR *bh, uint w);
GCEXPORT void gcBuffLogNewHandle(ExecEnv *ee, GCHandle *h);


/*****************************************************************************
*
* Thread specific GC block
*
* It conatains the create, uodate and snoop buffers.
*
* Also it contains the thread GC state and allocation lists.
*/
struct GCTHREADBLK {
  bool      gcInited;
  bool      gcSuspended;
  bool      cantCoop;
  bool      snoop;
  int       stage;
  int       stageCooperated;
  BUFFHDR   updateBuffer;
  BUFFHDR   createBuffer;
  BUFFHDR   snoopBuffer;

  ALLOCLIST allocLists[ N_BINS ];
#ifdef RCDEBUG
  struct {
    int nBytesAllocatedInCycle;
    int nRefsAllocatedInCycle;
    int nNewObjectUpdatesInCycle;
    int nOldObjectUpdatesInCycle;
  } dbg;
#endif // RCDEBUG
};

typedef struct SAVEDALLOCLISTS {
  struct SAVEDALLOCLISTS *pNext;
  ALLOCLIST allocLists[ N_BINS ];
} SAVEDALLOCLISTS;

/*****************************************************************************
*
* Global GC block
*
* GCHS4 is defined as zero so that the GC is in this state when the system
* is initialized.
*/
enum GCSTAGE { GCHS1=1, GCHS2=2, GCHS3=3, GCHS4=0, GCHSNONE=0x12345678};

#define N_GC_STAGES 4

enum GCTYPE { GCT_TRACING=0, GCT_RCING=1 };

#define N_SAMPLES 4

struct GCVAR {
  bool           initialized;
  bool           gcActive;
  int            iCollection;
  int            requestPhase;
  int            collectionType;
  int            nextCollectionType;

  // triggering
  bool           memStress;
  bool           usrSyncGC;
  int            gcTrigHigh;
  int            runHist[2][N_SAMPLES];

  ExecEnv*       ee;
  sys_thread_t*  sys_thread;
  int            stage;
  uint*          createBuffList;
  uint*          updateBuffList;
  uint*          snoopBuffList;
  uint*          deadThreadsCreateBuffList;
  uint*          deadThreadsUpdateBuffList;
  uint*          deadThreadsSnoopBuffList;
  uint*          deadThreadsReinforceBuffList;
  uint*          reinforceBuffList;
  GCHandle**     tempReplicaSpace;
  H1BIT_BMP      localsBmp;
```

```
  H2BIT_BMP      rcBmp;
  H1BIT_BMP      zctBmp;
  BUFFHDR        zctBuff;
  BUFFHDR        nextZctBuff;
  BUFFHDR        tmpZctBuff;
  BUFFHDR        uniqueLocalsBuff;
  BUFFHDR        preAllocatedBuffers[2];
  int            nPreAllocatedBuffers;
  GCHandle**     zctStack;
  GCHandle**     zctStackSp;
  GCHandle**     zctStackTop;
  sys_mon_t*     gcMon;
  sys_mon_t*     requesterMon;
  SAVEDALLOCLISTS *pListOfSavedAllocLists;

  // chunk mgmt
  uint nAllocatedChunks;
  uint nChunksAllocatedRecentlyByUser;
  uint nUsedChunks;
  uint nFreeChunks;

  // settable options
  struct {
    int recommendOnlyRCGC;
    int useOnlyRCGC;
    int useOnlyTracingGC;
    int listBlkWorth;
    int userBuffTrig;
    int initialHighTrigMark;
    int lowTrigDelta;
    int raiseTrigInc;
    int lowerTrigDec;
    int uniPrio;
    int multiPrio;
  } opt;

#ifdef RCDEBUG

  struct {
    // running totals
    uint nObjectsAllocated;
    uint nObjectsFreed;

    uint nBytesAllocated;
    uint nBytesFreed;

    uint nRefsAllocated;
    uint nRefsFreed;

    uint nOldObjectUpdates;
    uint nNewObjectUpdates;
    uint nLoggedUpdates;
    uint nLoggedSlots;
    uint nStuckCounters;

    // from prev to curr cycle
    uint nPendInCycle;
    uint nFreeCyclesBroken;
    uint nDeadUpdateObjects;
    uint nDeadUpdateChilds;
    uint nDeadCreateObjects;
    uint nDeadReinforceObjects;
    uint nDeadReinforceChilds;
    uint nDeadSnooped;
  } dbgpersist;

  struct {
    uint nHS1Threads;
    uint nHS2Threads;
    uint nHS3Threads;
    uint nHS4Threads;

    uint nHS1CoopThreads;
    uint nHS2CoopThreads;
    uint nHS3CoopThreads;
    uint nHS4CoopThreads;

    // update logs
    uint nUpdateObjects;
    uint nUpdateChilds;
    uint nActualUpdateObjects;
    uint nActualUpdateChilds;
    uint nUpdateDuplicates;
    uint nUpdate2ZCT;
    uint nActualCyclesBroken;

    // update logs, for reinforcement
    uint nReinforceObjects;
    uint nReinforceChilds;
    uint nActualReinforceObjects;
    uint nActualReinforceChilds;
```

```
    // create logs
    uint nCreateObjects;
    uint nActualCreateObjects;
    uint nCreateDel;

    // same checks, during RC updating
    uint nUpdateRCObjects;
    uint nUpdateRCChilds;
    uint nUpdateRCDuplicates;
    uint nCreateRCObjects;

    // more RC updating...
    uint nDetermined;
    uint nUndetermined;

    // roots
    uint nLocals;
    uint nGlobals;
    uint nSnooped;
    uint nActualSnooped;

    // freeing
    uint nInZct;
    uint nRecursiveDel;
    uint nFreedInCycle;
    uint nRecursivePend;
    uint nBytesAllocatedInCycle;
    uint nBytesFreedInCycle;
    uint nRefsAllocatedInCycle;
    uint nRefsFreedInCycle;

    // tracing stuff
    uint nTracedInCycle;

    // counters
    uint nStuckCountersInCycle;

    // updates
    int nNewObjectUpdatesInCycle;
    int nOldObjectUpdatesInCycle;
  } dbg;
#endif // RCDEBUG
};


/*****************************************************************************
*
* GC Exports
*/
GCEXPORT void  gcGetInfo( uint *pUc, uint *pFc, uint *pAc, int *iGc );
GCEXPORT void  gcBuffSlowConditionalLogHandle( ExecEnv *ee, GCHandle *h);
GCEXPORT void  gcBuffAllocAndLink( ExecEnv *ee, BUFFHDR *bh);
GCEXPORT void  gcRequestSyncGC(void);
GCEXPORT void  gcRequestAsyncGC();
GCEXPORT void  gcInit(int nMegs);
GCEXPORT void  gcInstallBlk(ExecEnv* ee);
GCEXPORT void  gcUninstallBlk(ExecEnv* ee);
GCEXPORT bool  gcNonNullValidHandle( GCHandle *h);
GCEXPORT bool  gcValidHandle( GCHandle *h);
GCEXPORT void  gcThreadAttach(ExecEnv *ee);
GCEXPORT void  gcThreadDetach(ExecEnv *ee);
GCEXPORT void  gcThreadCooperate(ExecEnv *ee);

extern struct BLKVAR     blkvar;
extern struct CHKCONV    chkconv;


#endif /* __RCGC__ */
```

End of file source listing

D.7 rcbmp.c and rcbmp_inline.h

These two files contain the declaration and implementation of a 1-bit-per-word and 2-bit-per-word data structures which are used extensively by the (e.g., for the ZCT and reference counters). Since the declarations are repeated in the definition, we bring here only the listing of rcbmp.c.

Source listing for file rcbmp.c

```
/*
 * File:    rcbmp.c
 * Aurhor:  Yossi Levanoni
 * Purpose: 1 bit per word and 2 bit per word bitmap implementation.
 */
#ifdef RCNOINLINE
```

```
#include <stdio.h>

#include "rcgc.h"


#define PAGE_SIZE 4096
#define ROUND_PAGE(u) (((u)&~(PAGE_SIZE-1))+PAGE_SIZE)

/*
 * BIT FIELD MANIPULATION
 */
#define MAKE_MASK(shift,length)           (((1<<(length))-1)<<(shift))
#define GET_BIT_FIELD(w,shift,length)     (((w)&MAKE_MASK(shift,length))>>shift)

#define OR_BIT_FIELD(w,v,shift)           do{ (w) = (w) | ((v)<<(shift)); \
                                          }while(0)

#define CLEAR_BIT_FIELD(w,shift,length)   do{ (w) = (w) & (~MAKE_MASK(shift,length)); \
                                          }while(0)

#define SET_BIT_FIELD(w,v,shift,length)   do{CLEAR_BIT_FIELD(w,shift,length);\
                                          OR_BIT_FIELD(w,v,shift);\
                                          }while(0)

/*
 * Specify (log) allignment of handles.
 */
#define H_GRAIN_BITS      3
/*
 * Field selector bits.  The next 3 bits select
 * the bit inside the bmp word.  there
 * are 8 options.
 */
#define H1B_FS_BITS       3
/*
 * The rest of the bits handle selects
 * the bmp byte inside the bitmap.
 */
#define H1B_BS_BITS       (32-(H_GRAIN_BITS+H1B_FS_BITS))
#define H1B_NON_BS_BITS   (H_GRAIN_BITS+H1B_FS_BITS)


#define H1BIT_BYTE(entry,h)   (byte*)(((uint)h>>H1B_NON_BS_BITS) + (byte*)entry)

void H1BIT_Set(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H1BIT_BYTE(entry, h);
  byte v = *bbmp;
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS, H1B_FS_BITS );
  OR_BIT_FIELD(v, 1, field_selector );
  *bbmp = v;
}


void H1BIT_Clear(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H1BIT_BYTE(entry, h);
  byte v = *bbmp;
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS, H1B_FS_BITS );
  CLEAR_BIT_FIELD(v, field_selector, 1 );
  *bbmp = v;
}

void H1BIT_ClearByte(byte* entry, unsigned h)
{
  byte *bbmp = H1BIT_BYTE(entry, h);
  *bbmp = 0;
}

void H1BIT_Put(byte* entry, unsigned h, unsigned val)
{
  mokAssert( val <= 1);
  if (val==0)
    H1BIT_Clear(entry, h);
  else
    H1BIT_Set(entry, h);
}

byte H1BIT_Get(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte  *bbmp = H1BIT_BYTE(entry, h);
  byte  v = *bbmp;
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS, H1B_FS_BITS );
  uint res = GET_BIT_FIELD(v, field_selector, 1 );
  return res;
}

/*
```

```
 * Create a new 1-bit per handle BMP with the handles starting
 * at address ‘rep_addr’ and the handles area being ‘rep_size’
 * bytes long.
 */
void H1BIT_Init(H1BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size )
{
  /* each bit in the bimtap represents a handle, which
   * takes 2^H_GRAIN_BITS bytes.  So a byte in the
   * bitmap repreesnts 2^(H_GRAIN_BITS+3) bytes in the
   * handle space.
        */
  bmp->bmp_size = rep_size >> (H_GRAIN_BITS+3);
  bmp->bmp_size = ROUND_PAGE( bmp->bmp_size );
  bmp->bmp = (byte*)mokMemReserve( NULL, bmp->bmp_size );
  mokMemCommit( bmp->bmp, bmp->bmp_size, true );
  bmp->rep_addr = (byte*)rep_addr;
  bmp->entry = bmp->bmp - (((unsigned)rep_addr)>>H1B_NON_BS_BITS);
}


/*********************************************************************

Implementation of a 2 bit per handle BMP.

Layout of a handle:

| 31 --------- 5 | 4 -- 3 | 2 - 0 |
|      BS        |   FS   |  Z    |

Where:

-- Z: these bits are always zero (because handles are 8-byte aligned).
-- FS: Field Select.  Selects a 2-bit field in a byte of the
       bitmap.  The selector is 4 bits wide cause there are 4
       possibilies.
-- BS: Word selector, relatively to the beginning of the heap, this is
       the bitmap word selector.

*********************************************************************/

/*
 * Field selector bits.  There are 16 options.  If the
 * selector value is s (with 0<=s<=15), then the field
 * begins at bit s*2.
 */
#define H2B_FS_BITS       2
/*
 * The rest of the handle selects
 * the bmp word inside the bitmap.
 */
#define H2B_BS_BITS       (32-(H_GRAIN_BITS+H2B_FS_BITS))
#define H2B_NON_BS_BITS   (32-H2B_BS_BITS)

#define H2BIT_BYTE(entry,h)   ((((uint)h)>>H2B_NON_BS_BITS) + entry)


void H2BIT_Put(byte* entry, unsigned h, unsigned val)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte v = *bbmp;
  /* we include the third least bit in the selector (it is always zero). */
  /* to get selection of 0,2,4,...,30, and not 0,1,...15.                */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );
  mokAssert( val <= 3);

  SET_BIT_FIELD(v, val, field_selector, 2);
  *bbmp = v;
}


void H2BIT_Clear(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte v = *bbmp;
  /* we indlude the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  CLEAR_BIT_FIELD(v, field_selector, 2);
  *bbmp = v;
}


void H2BIT_Stuck(byte* entry, unsigned h)
{
```

```
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte v = *bbmp;
  /* we indlude the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  OR_BIT_FIELD(v, 3, field_selector );
  *bbmp = v;
}

byte  H2BIT_Get(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte v = *bbmp;
  byte  res;
  /* we indlude the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  res = GET_BIT_FIELD(v, field_selector, 2 );
  return res;
}

#ifdef RCDEBUG
#pragma optimize( "", off )
void _forceIncSanityCheck(byte *entry, unsigned h, int f)
{
  int nextF = (f==3) ? 3 : f+1;

  mokAssert( H2BIT_Get(entry,h) == nextF );
  if (f==2) {
    gcvar.dbg.nStuckCountersInCycle++;
  }
}
#pragma optimize( "", on )
#endif

void H2BIT_Inc(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte val = *bbmp;
  uint f;
  /* we indlude the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  f = GET_BIT_FIELD(val, field_selector, 2);
  mokAssert( f<= 3 );
  if (f<3) { /* STUCK remains STUCK */
    SET_BIT_FIELD( val, f+1, field_selector, 2);
    *bbmp = val;
  }
#ifdef RCDEBUG
  _forceIncSanityCheck( entry, h, f);
#endif
}


byte H2BIT_IncRV(byte* entry, unsigned h)
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte val = *bbmp;
  uint f;
  /* we indlude the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  f = GET_BIT_FIELD(val, field_selector, 2);
  mokAssert( f<= 3 );
  if (f<3) { /* STUCK remains STUCK */
    SET_BIT_FIELD( val, f+1, field_selector, 2);
    *bbmp = val;
  }
#ifdef RCDEBUG
  _forceIncSanityCheck( entry, h, f);
#endif
  return f;
}

byte H2BIT_Dec(byte* entry, unsigned h)
```

```
{
  /* entry address into the bitmap.*/
  byte *bbmp = H2BIT_BYTE(entry, h);
  byte val = *bbmp;
  uint f;
  /* we include the upper zero in the selector */
  uint field_selector = GET_BIT_FIELD( h, H_GRAIN_BITS-1, H2B_FS_BITS+1 );

  mokAssert( field_selector%2 == 0);
  mokAssert( field_selector <= 30 );

  f = GET_BIT_FIELD(val, field_selector, 2);
  mokAssert( f<= 3 );
  mokAssert( f>= 1 ); /* we should never go below zero */
  if (f<3) { /* STUCK remains STUCK */
    SET_BIT_FIELD( val, f-1, field_selector, 2);
    *bbmp = val;
    mokAssert( H2BIT_Get(entry,h)== f-1 );
  }
  return f;
}

/*
 * Create a new 2-bit per handle BMP with the handles starting
 * at address ‘rep_addr’ and the handles area being ‘rep_size’
 * bytes long.
 */
void H2BIT_Init(H2BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size )
{
  /* each 2 bits in the bimtap represents a handle, which
   * takes 2^H_GRAIN_BITS bytes.  So a byte in the
   * bitmap repreesnts 2^(H_GRAIN_BITS+2) bytes in the
   * handle space.
        */
  bmp->bmp_size = rep_size >> (H_GRAIN_BITS+2);
  bmp->bmp_size = ROUND_PAGE( bmp->bmp_size );
  bmp->bmp = (byte*)mokMemReserve( NULL, bmp->bmp_size );
  mokMemCommit( bmp->bmp, bmp->bmp_size, true );
  bmp->rep_addr = (byte*)rep_addr;
  bmp->entry = bmp->bmp - (((unsigned)rep_addr)>>H2B_NON_BS_BITS);
}

char * write_bits(unsigned x)
{
  char *s = (char *)mokMalloc(33, false);
  unsigned i = 1<<31;
  int j=0;
  for (;j<32;j++) {
    s[j] = x&i ? '1' : '0';
    i >>= 1;
  }
  s[j] = '\0';
  return s;
}

void testBitFields(void)
{
  int shift, length;
  unsigned m=0, val;

  while (1) {
    jio_printf("Enter shift length val, please: ");
    scanf("%d %d %x", &shift, &length, &val );
    SET_BIT_FIELD(m, val, shift, length);
    jio_printf("m=(%x)%s field=(%x)%s\n", m, write_bits(m),
           GET_BIT_FIELD(m, shift, length), write_bits(GET_BIT_FIELD(m, shift, length)) );
  }
}


typedef struct HandleTAG { unsigned h1, h2; } Handle;

H2BIT_BMP Bmp;

#define N_HANDLES 10000

void test2BitBmp(void)
{
  int i,j;
  Handle* handleSpace = (Handle*)mokMalloc( N_HANDLES*sizeof(Handle), false );
  H2BIT_BMP *bmp = &Bmp;

  H2BIT_Init( bmp, (unsigned*)handleSpace, N_HANDLES*sizeof(Handle) );
  for (i=0; i<2 ;i++) {
    for (j=0; j<N_HANDLES; j++) {
      uint v = H2BIT_Get( bmp->entry, (unsigned)&handleSpace[j] );
      if (v != (uint)i)
        jio_printf("Bad RC for j=%d, val=%x\n", j, v );
      else
        jio_printf("Good RC for j=%d, val=%x\n", j, v );
      H2BIT_Inc(  bmp->entry, (unsigned)&handleSpace[j] );
    }
```

```
  }
  for (i=2; i>=0 ;i--) {
    for (j=0; j<N_HANDLES; j++) {
      uint v = H2BIT_Get( bmp->entry, (unsigned)&handleSpace[j] );
      if (v != (uint)i )
        jio_printf("Bad RC for j=%d, val=%x exoect=%i\n", j, v, i );
      else
        jio_printf("Good RC for j=%d, val=%x\n", j, v );
      H2BIT_Dec(  bmp->entry, (unsigned)&handleSpace[j] );
    }
  }
}


#endif /* RCNOINLINE */
/**/
```

End of file source listing

D.8 rcgc_internal.h

rcgc_internal.h contains declarations which are needed internally by the collector and allocator (forward declarations etc.)

Source listing for file rcgc_internal.h

```
/*
 * File:    rcblkmgr.h
 * Author:  Mr. Yossi Levanoni
 * Purpose: Header for internal use of the collector/allocator.
 */
#ifndef __RCGC_INTERNAL__
#define __RCGC_INTERNAL__

GCFUNC  bool     gcCompareAndSwap( unsigned *addr, unsigned oldv, unsigned newv);
GCFUNC  void     gcSpinLockEnter(volatile unsigned *p, unsigned id);
GCFUNC  void     gcSpinLockExit(volatile unsigned *p, unsigned id);
GCFUNC  void     gcCheckGC(void);

GCFUNC  void              blkInit( unsigned nMB );
GCFUNC  BlkAllocHdr*      blkAllocBlock( ExecEnv *ee );
GCFUNC  void              blkFreeChunkedBlock( BlkAllocHdr *ph );
GCFUNC  void              blkFreeSomeChunkedBlocks( BlkAllocHdr **pph, int nBlocks );
GCFUNC  void              blkFreeRegion( BlkAllocBigHdr *ph );
GCFUNC  void              blkSweep(void);


GCFUNC    void     chkFlushRecycledListEntry( RLCENTRY *rlce );
GCFUNC    void     chkFlushRecycledListsCache( void );
GCFUNC    void     chkSweepChunkedBlock( BlkAllocHdr *ph, int status);
GCFUNC    void     chkInit(unsigned nMB);

#ifdef RCDEBUG
GCFUNC void chkPreCollect(BLKOBJ* o);
#endif /* RCDEBUG */

#ifdef RCNOINLINE

GCFUNC void H1BIT_Set(byte* entry, unsigned h);
GCFUNC void H1BIT_Clear(byte* entry, unsigned h);
GCFUNC void H1BIT_ClearByte(byte* entry, unsigned h);
GCFUNC void H1BIT_Put(byte* entry, unsigned h, unsigned val);
GCFUNC byte H1BIT_Get(byte* entry, unsigned h);
GCFUNC void H1BIT_Init(H1BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size );

GCFUNC void H2BIT_Put(byte* entry, unsigned h, unsigned val);
GCFUNC void H2BIT_Clear(byte* entry, unsigned h);
GCFUNC void H2BIT_Stuck(byte* entry, unsigned h);
GCFUNC byte H2BIT_Get(byte* entry, unsigned h);
GCFUNC void H2BIT_Inc(byte* entry, unsigned h);
GCFUNC byte H2BIT_IncRV(byte* entry, unsigned h);
GCFUNC byte H2BIT_Dec(byte* entry, unsigned h);
GCFUNC void H2BIT_Init(H2BIT_BMP* bmp, unsigned* rep_addr, unsigned rep_size );

#endif /*  RCNOINLINE */

GCFUNC   uint  gcGetHandleRC(GCHandle* h);

/*************************************************************
 * System utilities layer (MOK)
 *
 */
#define mokSleep Sleep

/*
 * Memory
```

```
 */
/* Advanced */
GCFUNC void* mokMemReserve(void *starting_at_hint, unsigned sz );
GCFUNC void  mokMemUnreserve( void *start, unsigned sz );
GCFUNC void* mokMemCommit( void *start, unsigned sz, bool zero_out );
GCFUNC void  mokMemDecommit( void *start, unsigned sz );

/* C style */
GCFUNC void* mokMalloc( unsigned sz, bool zero_out );
GCFUNC void  mokFree( void *);

/* zero out */
GCFUNC void  mokMemZero( void *start, unsigned sz );

#define mokAssert sysAssert
#define gcAssert  sysAssert

#ifdef RCDEBUG
#define Im_used   0x1badbad1
#define Im_free   0x12344321
#endif

int x86CompareAndSwap(unsigned *addr, unsigned oldv, unsigned newv);

#define ___compare_and_swap  x86CompareAndSwap
#define gcCompareAndSwap     x86CompareAndSwap


/*
 * p is a pointer to BlkAllocHdr.  Lock and unlock the page
 */
#pragma optimize( "", off )
static void bhLock(BlkAllocHdr *p)
{
  volatile word *ptr = (volatile word*)&p->StatusLockBinidx;
  for (;;) {
    volatile word oldv, newv;
    oldv =  *ptr;
    oldv = oldv & ~LOCKMASK;
    newv = oldv | LOCKMASK;
    if (gcCompareAndSwap( (word*)ptr, oldv, newv))
      goto ___do_bh_lock_end;
  }
 ___do_bh_lock_end:;
}

static bhUnlock(BlkAllocHdr* p)
{
  for (;;) {
    volatile word *ptr = (volatile word*)&p->StatusLockBinidx;
    word oldv, newv;
    oldv =  *ptr;
    if (!(oldv & LOCKMASK )) {
      __asm { int 3 }
    }
    newv = oldv & ~LOCKMASK;
    if (gcCompareAndSwap( (word*)ptr, oldv, newv))
      goto ___do_bh_unlock_end;
  }
 ___do_bh_unlock_end:;
}
#pragma optimize( "", on )


#define gcNonNullValidHandle  _isHandle
#define gcValidHandle(h)      ((h)==NULL || _isHandle((h)))

#endif /* __RCGC_INTERNAL__ */
```

End of file source listing

D.9 rchub.c

This file simply includes the block manager, chunk manager and collector into a single translation unit.

Source listing for file rchub.c

```
/*
 * File:    rcbmp.c
 * Aurhor:  Yossi Levanoni
 * Purpose: Includes all of the allocator and collector into a single
 *          translation unit.
 */
#define GCINTERNAL
```

```
#define gcUnhand(h)   ((JHandle**)(((char*)h)+sizeof(GCHandle)))
#define gcRehand(obj) ((JHandle*)(((char*)obj)-sizeof(GCHandle)))

#include "rcgc.h"
#include "rcgc_internal.h"
#include "../../../win32/hpi/include/threads_md.h"

struct BLKVAR   blkvar;
struct CHKCONV  chkconv;

static struct CHUNKVAR   chunkvar;
static struct GCVAR      gcvar;

#include "mok_win32.c"
#include "rcbmp.c"
#include "rcblkmgr.c"
#include "rcchunkmgr.c"
#include "rcgc.c"
```

End of file source listing

D.10 ylrc_protocol.h

This file (the name of which stands for "The Yossi Levanoni's Reference Counting Protocol") defines the write barrier that must be adhered to when changing references. i.e., this is the declaration of the write barrier.

Source listing for file ylrc_protocol.h

```
/*
 * File:    ylrc_protocol.h
 * Author:  Mr. Yossi Levanoni
 * Purpose: Definition of the write barrier
 */
#ifndef YLRC

#define YLRC

struct execenv;
typedef struct execenv ExecEnv;

void gcDo_gcupdate(ExecEnv *ee, void *_h, void *_slot, void *_newval );
void gcDo_gcupdate_array(ExecEnv *ee, void *_arrayh, void* _slot, void *newval);
void gcDo_gcupdate_class(ExecEnv*  ee, ClassClass* cb, void *_slot, void *_newval );
void gcDo_gcupdate_jvmglobal(ExecEnv* ee, void* _global, void *_newval );
void gcDo_gcupdate_static( ExecEnv* ee, struct fieldblock* fb, void* slot, void* _newval );


#define gcupdate(ee,_h,_slot,_newval ) \
  gcDo_gcupdate(ee,_h, _slot,_newval )

#define gcupdate_array(ee,_arrayh,_slot,newval) \
  gcDo_gcupdate_array(ee,_arrayh, _slot,newval)
#define gcupdate_class(ee,cb,_slot,_newval ) \
  gcDo_gcupdate_class(ee,cb,_slot,_newval )
#define gcupdate_jvmglobal(ee,_global,_newval ) \
  gcDo_gcupdate_jvmglobal(ee, _global,_newval )
#define gcupdate_static(ee,fb,slot,_newval ) \
  gcDo_gcupdate_static(ee,fb,slot,_newval )

#endif /* ! YLRC */
```

End of file source listing

D.11 gc.c

This file contains code mostly from the original JVM. Most importantly, this file includes rchub.c and defines the entry point for allocation code.

Due to the author's non-disclosure agreement with Javasoft, only those parts of the file which are new to the collector are listed below.

Source listing for file gc.c

```
.
.
.
```

```
#include "rchub.c"
.
.
.
/*****************************************************************
 *************                                 ******************
 *************  Allocation Cache (degenerated)  ******************
 *************                                 ******************
 *****************************************************************/

HObject * cacheAlloc(ExecEnv *ee, struct methodtable *mptr, long size)
{
#ifdef RCDEBUG
  static int deltaMax = -1;
  int delta = GetTickCount();
#endif

  GCHandle *h;
  JHandle  *_h;
  uint     *obj;
  int      bin;

  uint nbytes = sizeof(GCHandle) + size;

  if (nbytes <= MAX_CHUNK_ALLOC) {
    bin = chkconv.szToBinIdx[ nbytes ];

    chkAllocSmallInlined( ee, bin, _h );

    if (!_h) return NULL;

#ifdef RCDEBUG
    ee->gcblk.dbg.nBytesAllocatedInCycle += chkconv.binSize[ bin ];
#endif

    h = (GCHandle*)_h;
    obj = (uint *)(h + 1);
    if (size > 0)
      memset( obj, 0, size );

#ifdef RCDEBUG
    h->status = Im_used;
#endif
    h->methods = mptr;
    h->obj     = obj;

    gcBuffLogNewHandle(ee, h);

#ifdef RCDEBUG
    delta = GetTickCount() - delta;
    if (delta > deltaMax) {
      deltaMax = delta;
      printf( " *** CACHE(small, nbytes=%d) delta=%d\n", nbytes, delta );
    }
#endif
  }
  else {
    BlkAllocBigHdr *ph;
    int i;
    for(i=0; i<3; i++) {
      ph = blkAllocRegion( nbytes, ee  );
      if (ph) goto __good;
      gcvar.memStress = true;
      gcRequestSyncGC();
    }

    return NULL;

  __good:
    h = (GCHandle*)BLOCKHDROBJ((BlkAllocHdr*)ph);

#ifdef RCDEBUG
    ee->gcblk.dbg.nBytesAllocatedInCycle += ph->blobSize * BLOCKSIZE;
#endif


    obj = (uint *)(h+1);
    ZeroMemory( obj, size );
#ifdef RCDEBUG
    h->status = Im_used;
#endif
    h->methods =  mptr;
    h->obj     =  obj;

    gcBuffLogNewHandle(ee, h);

    ph->allocInProgress = 0;

#ifdef RCDEBUG
    delta = GetTickCount() - delta;
    if (delta > deltaMax) {
      deltaMax = delta;
```

```
      printf( " *** CACHE(big, nbytes=%d) delta=%d\n", nbytes, delta );
    }
#endif
  }
  sysAssert( h );

  return (HObject*)h;
}



/*****************************************************************
 *************                                  ******************
 *************           Heap Meters            ******************
 *************                                  ******************
 *****************************************************************/
.
.
.
int64_t
TotalObjectMemory(void)
{
  return blkvar.heapSz;
}

int64_t
FreeObjectMemory(void)
{
  int freePartialBytes[N_BINS], freePartialBlocks[N_BINS];

  int nBlockBlocks = blkvar.nWildernessBlocks + blkvar.nListsBlocks;
  int nBlockBytes, nPartialBytes, nPartialBlocks, nBytes, i;
  float avgRes;

  printf("****************** FreeObjectMemory statistics(begin)\n");
  nBlockBytes = nBlockBlocks*BLOCKSIZE;
  printf("BlkMgr blocks=%d MB=%d\n", nBlockBlocks, nBlockBytes>>20 );


  chkGetPartialBlocksStats( freePartialBlocks, freePartialBytes );
  printf("Partial:\n");
  printf("binsz\tblocks\tMB\n");

  nPartialBytes = 0;
  nPartialBlocks = 0;

  for (i=0; i<N_BINS; i++) {
    printf("%d\t%d\t%d\n",
           chkconv.binSize[i],
           freePartialBlocks[i],
           freePartialBytes[i]>>20 );
    nPartialBlocks += freePartialBlocks[i];
    nPartialBytes += freePartialBytes[i];
  }

  if (nPartialBlocks)
    avgRes =  (float)nPartialBytes /((float)BLOCKSIZE*(float)nPartialBlocks);
  else
    avgRes = -1;

  printf("Total partial: blocks=%d MB=%d avg-res=%f\n",
         nPartialBlocks,
         nPartialBytes>>20,
         avgRes
         );
  nBytes = nBlockBytes + nPartialBytes;
  printf("Total free MB=%d\n", nBytes>>20 );
  printf("****************** FreeObjectMemory statistics(end)\n");

  return nBytes;
}

int64_t
TotalHandleMemory(void)
{
  return 0;
}

int64_t
FreeHandleMemory(void)
{
  return 0;
}
.
.
.
/*
 * User interface to synchronous garbage collection.  This is called
 * by an explicit call to GC.
 */
void
gc(unsigned int free_space_goal)
```

```
{
  gcRequestSyncGC();
}
.
.
.
bool_t isHandle(void *p)
{
  return _isHandle(p);
}

bool_t isObject(void *p)
{
  GCHandle *h = (GCHandle* )(((char*)p)-sizeof(GCHandle));
  return _isHandle(h);
}
.
.
.
bool_t isValidHandle(JHandle *h)
{
  return _isHandle(h);
}
.
.
.
```

End of file source listing