A Note on the Implementation of Replication-Based Garbage Collection for Multithreaded Applications and Multiprocessor Environments

Alain Azagury* Elliot K. Kolodner† Erez Petrank‡

Department of System Technology
IBM Haifa Research Lab
MATAM, Haifa 31905, Israel.

August 12, 1998

Abstract

Replication-based incremental garbage collection [5, 4] is one of the more appealing concurrent garbage collection algorithms known today. It allows continuous operation of the application (the mutator) with very short pauses for garbage collection. There is a growing need for such garbage collectors suitable for a multithreaded environments such as the Java Virtual Machine. Furthermore, it is desirable to construct collectors that also work on multiprocessor computers.

We begin by pointing out an important, yet subtle point which arises when implementing the replication based garbage collector for a multithreaded environment. We first show that a simple and natural implementation of the algorithm may lead to an incorrect behavior of multithreaded applications. We then show that another simple and natural implementation eliminates the problem completely. Thus, the contribution of this part is in stressing this warning to future implementors.

Next, we address the effects of the *memory coherence model* on this algorithm. We show that even when the algorithm is properly implemented with respect to our first observation, a problem might still arise when a multiprocessor system is used. Adopting a naive solution to this problem results in very frequent (and expensive) synchronization. We offer a slight modification to the algorithm which eliminates the problem and requires little synchronization.

Keywords: Parallel algorithms, Memory management, Garbage collection.

1 Introduction

Replication-based garbage collection [5, 4] is one of the more elegant solutions known today for non-disruptive concurrent garbage collection with minimal synchronization between the collector and the application program (the mutator). We expect this algorithm to prove useful for applications which require fast response, and even for real time applications.

*E-mail: azagury@haifa.vnet.ibm.com.
†E-mail: kolodner@haifa.vnet.ibm.com.
‡E-mail: erezp@haifa.vnet.ibm.com.

Environments, which support multithreaded applications are becoming increasingly popular in particular, as part of the Java runtime [2]. Furthermore, systems with more than one processor are becoming more common in modern machines, especially in server systems, but also on the desktop.

In this note, we address the problems of implementing replication based garbage collection for a multithreaded language such as Java, and the problems of implementing this algorithm on a system with multiprocessor computer.

In this short note, we do not elaborate on the motivation for garbage collection in general, nor on the need for non-disruptive collection. We also do not discuss the advantages of multithreaded applications. The reader is referred to [1] for a good overview on garbage collection algorithms and motivations. The reader may find motivation and experimental results for the specific replication garbage collection algorithm in the original papers [5, 4].

1.1 A subtle implementation point for multithreaded application

We begin with the case of multithreaded applications. Namely, there are several mutator threads (and one additional collector thread). We first note that one natural implementation of the algorithm introduces a subtle problem which may foil the correctness of the running application. We then suggest a solution which is a different implementation of the algorithm. This other implementation is as simple and natural as the problematic implementation. We feel that since the problem is subtle, this issue deserves a clear and explicit discussion which should serve as a warning to future implementors.

In the original papers [4, 5] there is no discussion of this issue. The algorithm is discussed for the case of a single mutator thread, for which both possible implementations are fine. We remark that in their original implementation the authors did choose to implement the algorithm in the manner that is suitable for multi-mutator-threads operation [3].

1.2 Care required in a multiprocessor system

Next, we consider the behavior of the algorithm in modern multiprocessor environments, e.g., Power-PC, Sparc, Alpha, and Intel. In these environments there is a typical problem of memory coherence. Namely, the order of updates executed by Processor P_1 is not necessarily the order viewed by Processor P_2. This problem brings back to life the misbehavior of Subsection 1.1 even with the proper implementation. Furthermore, any update performed by the mutators might be "forgotten" because of the garbage collection activity. Thus, in this case a slight modification is needed in order to make the algorithm fit into a multiprocessor environment. The naive solution, of synchronizing the views of the various processors whenever needed, requires very frequent synchronization is thus unacceptably expensive. Our solution requires only little synchronization.

1.3 Organization

In Section 2 we briefly recall the algorithm of replication based garbage collection [5, 4]. In Section 3 we discuss the multithreaded environment. We first explain the implementation which is problematic in Subsection 3.1, next, in Subsection 3.2 we present an example of a program that may go astray due to garbage collection activity. In subsection 3.3 we point out the correct implementation that eliminates the problem. In section 4 we discuss the problematic effects of multiprocessor environments on the algorithm and suggest a modification of the algorithm which eliminates the problem.

2 Replication based garbage collection

In this section, we shortly recall the replication based garbage collection algorithm of [5, 4]. In this short note we only briefly recall the replicating algorithm. We assume the reader is familiar with copying garbage collections in general. For details, the reader is referred to [5, 4]. In a simple copying garbage collection the heap is divided into two spaces: *From-space* and *to-space*. During normal operation the application uses *to-space*. When garbage collection is needed, the application (mutator) threads are stopped, the roles of *from-space* and *to-space* are flipped, and the collector copies all live objects from the *from-space* area to *to-space*. After doing that, the collector reclaims the *from-space* area and lets the mutator threads resume. We skip the details.

The idea of replicating garbage collection is to let the collector work in parallel to the mutators[1]. The replication based collector starts a collection cycle by switching the names of the semi-spaces *from-space* and *to-space*. But then, the mutator threads are not stopped. While the mutators keep running and operating on *from-space*, the collector replicates the live objects from the *from-space* area into the *to-space* area. Finally, the mutator threads are stopped and their roots are updated to point to the replicated objects in the *to-space* area.

The problem is that while the replication is executed, objects in *from-space* keep on changing and this has to be reflected in the to-space replica. In order to make the replica consistent, the mutators log all modifications to a *mutation-log*. The collector updates the replicas according to the mutation log. Once the mutation log is cleared (i.e., all its records were applied by the collector on the replica and modified pointers are rescanned), the collector may stop the mutator threads for a short pause in which the collector clears the mutation log again (additional entries may have been appended until the mutators stopped) and updates the mutators roots. The pause for the final update (flip) is short.

3 Multithreaded applications

We start by presenting an issue that was not adequately addressed in the original papers and show how an implementation that does not address the issue can be incorrect. We describe the problematic implementation. Then, we present two applications which may operate incorrectly with this problematic implementation. Finally, we describe a correct implementation.

3.1 The problematic implementation

The problem arises when the mutation log is implemented in the standard "databases approach". Namely, each record in the mutation-log contains the relevant mutation details: The address to be updated and the value to put in the address. This seems like a natural approach which, as we fear, might be adopted for an implementation of the algorithm.

The problem occurs when there is a race on updates between the mutator threads. Namely, when two or more mutator threads try to update the same address. One of them "wins" and the value that it stores actually prevails in the race. The other loses and its update does not survive the race. We stress that this is a legitimate happening which is even useful for goals such as synchronization or locking. Let us describe why races cause a problem.

The problem is that the race on the updates on the given address is followed by a race on the mutation log. Each mutator updates the address and then inserts a mutation record into the

[1] This is not the first algorithm that allows parallel copying collection. The reader is referred to [1] for a balanced report on the various existing concurrent copying garbage collection algorithms.

mutation log. But it may happen that the race on the updates is settled inconsistently with the race on the mutation log. As a consequence, the *from-space* area contains a value in some address a which is inconsistent with the output of executing the mutation-log records one by one. Thus, the value in address a will be modified at some point (the "flip" point) without the mutators intention of modifying it again. In other words, the collection changes the content of the memory. In the next section, we discuss specific applications which may fail because of this inconsistency.

3.2 A counter example

Let us explain the problem of the implementation in Subsection 3.1 on a multithreaded application simply by constructing two counterexamples. In our examples, a correct program becomes incorrect due to garbage collection activity.

3.2.1 Locking

The simplest example is a "locker" object. Namely, an object that implements a lock on a critical resource (or section). We explain how the collection algorithm may foil the correctness of the lock, thus, creating chaos in the system synchronization. The reader may envision similar examples for various synchronization protocols. The key ingredient in the example is that the threads settle a decision problem by allowing a race on updates of a predetermined object.

Think of a lock which is implemented by a byte. The lock is locked when the byte value is 1, and released when it is 0. Access to the critical section is started by a thread performing compare and swap on this byte. If the initial value was 0, then the swap writes 1 to this byte. Only the writing mutator has access to the critical section. After executing the critical section, the mutator writes 0 to the lock to free it.

Now let's think of this operation under the replication based garbage collection. Say Mutator A finished executing the critical section and is now setting Lock=0. On the other hand, Mutator B that was waiting for the lock to be released is executing compare and swap, setting the lock to 1 again. By the collection algorithm, the two mutators must update the mutation log in order to keep the *to-space* replica updated. Thus, Mutator A writes a record `lock=1` in the mutation log, whereas Mutator B writes a record `lock=0`. Clearly, in *from-space* Mutator A had to be first. Otherwise B would not have been able to set the lock to 1. However, in the mutation-log, it is not clear which mutation-record appears first. In particular, B may beat A in the race and in this case the mutation log will contain the command `lock=1` before the command `lock=0`.

If this is indeed the case, and while B is working on the critical section, the collector finishes updating the *to-space* area and makes the flip. Now, all threads view the lock as being released, while B still believes that it has the lock on the critical section. In other words, the program correctness is foiled by the garbage collection operation.

3.2.2 Leader election

Another possible example of an application which may be foiled by the collector activity is the following implementation of space-efficient leader election.

Suppose we have a few active mutators out of n existing mutators, and we want to select one of them to lead a parallel algorithm (elect a leader). Suppose, also, that we choose to implement this protocol with the following space efficient algorithm: The mutators use one shared byte on which each mutator writes its id. The simultaneous writing creates a race between all threads that ends

when one of the updates "wins" in the race and its id remains written on the shared byte.[2]

This algorithm is, perhaps, not the nicest or most elegant solution to the problem of leader election. But this is irrelevant. The algorithm is "legitimate" and some user somewhere may choose to use it. Now, we consider the effects of replication based garbage collection on this algorithm.

Note that the race on the update of the shared byte is settled independently of the other race that occurs on the update of the mutation-log. Thus, the last update of the original cell does not have to be the last update of the mutation-log. Until the collector updates the roots to point into *to-space*, one mutator may be the leader, but as the collector updates the roots, the leader may change! This foils the assumption that a single leader was chosen and may cause mutators to decide on two different leaders, depending on the time they read the shared byte. Mutator M_i may be the leader for the mutators that read the shared byte before the flip, and a different mutator M_j may seem the leader for the mutators that read the shared byte after the flip, thus, foiling the correctness of the application program.

3.3 The solution

Note first, that one way to solve the problem is to ask that the mutation-log update be associated with the update of the memory. Namely, we *lock* the access to the object and then we update the object and the mutation-log before we release the object lock. It is easy to check that this indeed solves the problem. However, the price of synchronization on each update is unacceptable.

The solution is extremely simple. Instead of logging the mutation on the mutation log, we log only the address on which the update was made. To update the replica, the collector must therefore read the original space again and copy the relevant value to the replica.

Let us say a few words on why this implementation of the mutation-log is safe with respect to the problem discussed above. Suppose that there is a race on the update of the content of address a. Then although we do not know which mutator won the race, we know that *after* it won the race, it wrote the address to the mutation log. Thus, after the final value was written to address a, the address will be written into the log, and *after* the address will be written into the log, the collector will take care of copying the value (which by this time is final) from address a to the replicated space. Note that the other racing mutator threads may have written the address a into the mutation log before or after the thread that won the race. But we only care about the latest value being noted by the collector, and this is guaranteed.

4 Multiprocessor environments

In this section we argue that even with the proper implementation, in a multiprocessor environment the user program may still fail due to garbage collection activity. Furthermore, in this case, the problem arises even when there is no race condition in the application program. Any simple update of the application program may fail due to the garbage collector activity. We then suggest a slight modification of the original replication based garbage collection algorithm which solves the problem.

4.1 The problem

Let us start by presenting the problem in a multiprocessor environment. We do this by recalling why the proper implementation worked well in our previous scenario. We argued that keeping the order of the following four operations implies the correct behavior of the application:

[2]We do not discuss the question of timing, i.e., how the mutators know that the updates are over and they can read the leader name. One may assume a synchronous system, or a more relaxed assumption of a time-out mechanism.

1. The mutator updates address a.
2. The mutator adds address a to the mutation-log.
3. The collector reads a from the log, and finally,
4. The collector copies the content of address a to the proper address in the *to-space* replica.

The order of operations is enforced as follows: the second operation follows the first since this is the mutator program. The fourth follows the third since this is the collector program. And the third operation is a consequence of the second operation. Hence the order between them must be kept, and in Operation (4) the collector copies the value that was previously properly updated by the mutator in Operation (1).

The problem originates from the *memory coherence model* provided by a multiprocessor. Most architectures. e.g., IBM Power-PC, Sun Sparc, Digital Alpha, and Intel, do not guarantee that the operations executed by Processor P_1 appear in the same order in the view of Processor P_2. Thus, returning to the above 4 operations, it is possible that the collector will see the update of address a only after it reads the update to the mutation log. From the collector point of view, this means that it might copy the content of address a before the new value actually appears in his view. Thus, the new replica in *to-space* will contain an outdated value of the content of address a. Furthermore, this value will never be updated.

In what follows, we suggest a modification of the garbage collection algorithm that eliminates the above problem.

Remark: We would like to remark shortly on compiler optimization. The main issue here is that the process that reads the updated value must indeed get the value after the update. One thing that may foil the correct operation here is that the optimizing compiler decides to delay the actual update. The implementor must make sure that precautions are taken to make the optimizer "behave" properly, e.g., in C declare the appropriate variables volatile. We do not elaborate further on this point.

4.2 Our solution

We concentrate on the case in which we have multiple mutator threads and one collector thread. The more general case of a parallel collector, i.e., multiple *collector* threads is not discussed here. There are no "standard" algorithms that perform such a collection, and the synchronization problems, which arise, depend very much on the specific way chosen to implement the parallel collector. However, we believe that for any reasonable extension of the replication based collection to run with multiple threads, our solution can be naturally extended.

Recall that the main problem is to make the view of the various processors consistent. We do not want to discuss a specific architecture, but each has its own synchronization commands which enforce a consistent view between the processors. For example - the command `sync` on the IBM Power-PC, the command `membar` on SPARC, and the command `wbinvd` on Intel Pentium. Let us use the Power-PC `sync` operation to describe our solution. A similar solution can be implemented on any other platforms. The command `sync` on the Power-PC has the following effect: if Processor P_1 executes a `sync`, then all other processors view any operation that was executed by P_1 before the `sync` as being executed before any operation that was executed after the `sync`. In other words, an update performed by P_1 before the `sync` cannot be switched in the view of any processor with an update performed by P_1 after the `sync` operation.

The most naive solution is to perform such a `sync` to synchronize the view of the processors after each update. Namely, when performing Operation (1) and (2) of Subsection 4.1 above, the mutator must perform a `sync` operation between the two commands. Although this solution is correct, the cost is unacceptable. Synchronization operations are usually very expensive and we cannot allow a synchronization to take place that often.

Thus, we suggest using a local buffering system. Each processor uses a local buffer to store its mutation records, rather than storing them directly into the mutation log. The mutation log will not contain mutation records anymore, but will be a queue of buffers, and each buffer will contain mutation records. A mutator uses its buffer to store its mutation records and once a buffer is filled with records, the mutator adds the buffer to the mutation log. Just before adding the buffer to the log, a `sync` operation is executed. Let us now look at the collector. The collector reads the buffers from the mutation log one by one and performs the needed updates on the replica as dictated by the buffer records. Since the `sync` operation was executed before the buffer was appended to the mutation log, it is guaranteed that after the buffer is seen by the collector, all the updates that are mentioned in the buffer, are already viewed by the collector and therefore the collector copies the right values to the *to-space* replica.

Note that the actual access to the mutation log must be synchronized since the log is updated and read concurrently by several threads. This synchronization is also an expensive operation. Our buffering solution also reduces this cost. Instead of synchronizing once for every update, we synchronize once for every full buffer.

Finally, let us explain how the process ends. Once the collector finishes working on all buffers in the queue, it stops all mutator threads. The collector performs the updates that appear in the unfinished buffers of each thread, and only afterwards it finishes the collection cycle by performing the flip: Updating the mutators roots to point to the *to-space* part of the memory, and reclaiming the *from-space* area. Care must be taken in this case as well. For example, if a thread is stopped just after updating the memory and before updating the mutation log, then we face the same problem as above: the update is "lost" during the final stage of the collection.

A standard approach for copying garbage collection (in which the roots of the mutators are changed while they sleep) is to use a *safe-point* mechanism. A safe point is a place in the execution of a program at which it may be stopped for collection. When the garbage collection algorithm has to stop the threads, it waits until each thread arrives at a safe point. In our case, we would define the point between updating the memory and updating the mutation log as not being a safe point. Thus, a mutator thread will never be stopped at that point for collection.

We also remark that stopping the threads is usually done using some interrupt or trap mechanism. Furthermore, it is usually guaranteed that before the interrupt or trap handler is invoked, the memory gets updated and all processors can view all previous updates of the interrupted processor. Thus, this solves our coherence problem as well.

5 Acknowledgments

We thank Scott Nettles for his helpful comments and for answering our questions on the details of the implementation in [4].

References

[1] R. E. Jones and R. D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley & Sons, July 1996.

[2] J. Gosling, B. Joy and G. Steele. The Java Language Specification. Addison-Wesley.

[3] S. Nettles. Private communications.

[4] S. Nettles and J. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of ACM SIGPLAN Notices, Albuquerque, New Mexico, June 1993. ACM Press.

[5] S. Nettles, J. O'Toole, D. Pierce and N. Haines. Replication-Based Incremental Copying Collection. In Bekkers and Cohen, editors. Proceedings of *International Workshop on Memory Management*, volume 637 of Lecture Notes in Computer Science, St Malo, France, 16-18 September 1992. Springer-Verlag.