Replication-Based Incremental Copying Collection

Scott Nettles James O'Toole[†] David Pierce
Nicholas Haines

April 1993

CMU-CS-93-135

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[†]Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Also appears in *Proceedings of the SIGPLAN International Workshop on Memory Management*

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597 and by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0168.
The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Keywords: replication, garbage collection, incremental garbage collection, concurrent garbage collection, real-time garbage collection

Abstract

We introduce a new *replication-based* copying garbage collection technique. We have implemented one simple variation of this method to provide incremental garbage collection on stock hardware with no special operating system or virtual memory support. The performance of the prototype implementation is excellent: major garbage collection pauses are completely eliminated with only a slight increase in minor collection pause times.

Unlike the standard copying algorithm, the replication-based method does not destroy the original replica when a copy is created. Instead, multiple copies may exist, and various standard strategies for maintaining consistency may be applied. In our implementation for Standard ML of New Jersey, the mutator continues to use the from-space replicas until the collector has achieved a consistent replica of all live data in to-space.

We present a design for a concurrent garbage collector using the replication-based technique. We also expect replication-based gc methods to be useful in providing services for persistence and distribution, and briefly discuss these possibilities.

1 Introduction

Copying garbage collection (GC) is an important memory management technique, but its application has been largely limited to situations that can tolerate GC pauses. There have been numerous schemes for incremental or concurrent copying collectors that are "real time," i.e. that limit GC pauses to small bounded intervals. Real-time collectors interleave garbage collection with program execution, thus spreading out the copying work so that the individual interruptions are unobtrusive. These incremental collectors fall into one of two groups: those that require special hardware [6], and those that use virtual memory protection [2].

The disadvantage of techniques which use special hardware is that they are not portable. Techniques which use other operating system support such as the ability to control the virtual memory system are often not portable, and can be prohibitively costly due to the cost of trap handling or similar operations. We propose a new technique for implementing incremental and concurrent copying collectors that requires no special support from either hardware or operating system. In addition, it promises to be useful for other algorithms that use copying to provide features such as persistent data and distributed computing.

We first introduce our general approach, based on nondestructive copying or replication. Next we outline our experimental implementation and present preliminary performance measurements which demonstrate its excellent real-time behavior. Finally we discuss the application of the replication-based technique to concurrent collection, and suggest other applications.

2 The General Method

Copying collection works by copying all of the valid data from one region (from-space) to another (to-space), leaving the garbage behind. We assume the reader is familiar with the basic technique of copying collection as well as the notion of generational collection. The key operations of copying collection are as follows:

- *Copy* an object from from-space into to-space, leaving a forwarding pointer in the original from-space object.
- *Forward* a from-space pointer into to-space, if necessary copying the object it references, and redirecting the pointer to the to-space copy.
- *Scan* a to-space object, forwarding all of the object's pointers.

The mutator can perform the following operations on objects: read a field, write a field, and compare pointers for equality. Incremental GC requires that these operations be interleavable with the operations of the garbage collector outlined above. (Concurrent GC has much stricter requirements, discussed in section 5 below.)

Since the standard copying technique overwrites from-space objects with forwarding pointers in the *Copy* operation, most incremental collectors require that the mutator use only the to-space copy of an object. To maintain this invariant, the collection algorithm must rely on low-level hardware support. (E.g. hardware support for following forwarding pointers or trapping all attempts to access the unscanned portions of to-space.)

In contrast, our technique simply *replicates* the from-space object in to-space. A forwarding pointer is placed in a special word reserved at the head of the from-space object. Since the original object is not destroyed by the copying operation, any use of the object may continue to reference the original object. However, because multiple copies of an object may exist, read and write operations must adhere to one of several consistency protocols.

If reads are permitted to access either copy, write operations must modify both to-space and from-space replicas. Also, pointer-based equality tests must follow the forwarding pointers in order to ensure that only to-space (or only from-space) pointers are compared. In more sophisticated systems, where copying is used for purposes other than GC and there may be more than two replicas of an object, the mutator must modify *all* replicas (for this purpose we can make the forwarding chain circular by having a 'reversing pointer' in the newest replica). In this system, read operations can be freely interleaved with any of the GC operations, but under some consistency protocols the write operations may require synchronization with the collector, and care may be required to ensure that the mutator does not write from-space pointers into previously scanned to-space replicas.

This general protocol of reading any copy and writing all copies is a standard one used for maintaining replicated data, so we use the term "replication-based copying". Another possibility is to have write operations modify only the newest version of an object, in which case the read operations for *mutable* objects must always read the newest version. In section 5, we discuss this possibility, which may be preferable for concurrent applications.

Note that these operations are distinct from that of updating the 'root set', that set of pointers directly visible to the mutator (registers, the stack, etc.). At some point in the GC process, these pointers must be updated. In a standard incremental collector, this is done immediately after the 'flip' by a simple 'forward' operation to start the GC. With a replication-based algorithm, it is possible to delay this step until just before the flip, after copying all live data into to-space. By using this technique, the collector can ensure that the mutator uses only from-space objects. In this case, there is no need for the collector to synchronize with the mutator except very briefly at flip time. Notice that this variation is not fully general, as it does not provide for more sophisticated uses of copying.

The advantage of the above technique is that it allows for incremental collection with no special hardware or OS support, but what are the disadvantages? First, it requires one extra word per object for the forwarding pointer. Fortunately, this extra word can often be absorbed into other object header words which are already present. The second disadvantage is that the consistency protocol may make writes (and possibly reads of mutable objects) more expensive. For some languages this would be unsatisfactory because mutations are common. However, for applicative languages like SML, in which side effects are less frequent and mutable objects are clearly distinguished by a type system, this runtime cost is probably not a problem. The third disadvantage is that of copying latent garbage, but this is an inevitable cost of any incremental method, and all such garbage is discarded by the next collection. The final disadvantage is that tests of pointer equality become more expensive. This may be a serious disadvantage for Lisp family languages where the use of `eq` is common. It is probably less important for SML, because equality testing is already expensive, and not as frequently used.

3 Implementation

We have built a prototype implementation of a replication-based incremental collector for SML/NJ (version 66). In order to quickly test the utility of the replication-based method, we chose to implement a simple variation of the general replication algorithm. In this variation, the mutator uses only the from-space replicas. Therefore, the mutator need not adhere to a consistency protocol, and so only one small change to the SML/NJ compiler was required. The rest of the implementation work required modifications to the standard SML/NJ garbage collector.

SML/NJ uses a simple generational copying collector [1], with two generations known as new-space and old-space. The new-space is used for newly allocated data, and the old-space contains data which has survived at least one collection. When the new-space fills, a 'minor' collection is performed, copying data from the new-space to the old-space. The compiler keeps a record (the 'store list') of all writes to mutable objects so that references from the old-space into the new-space can be found during minor collection. When the old-space fills, a 'major' copying collection is performed. Minor collections are typically short and non-disruptive, but major collections are often lengthy.

Our implementation leaves minor collections as they are, but makes the major collections incremental, doing some portion of the major collection at each minor collection. There are several reasons for this choice. First, it avoids having the allocator allocate the forwarding word; instead it is added when objects are copied from new to old. This avoids a change to the compiler backend's allocation primitives. Second, since the GC is in control during a minor collection, it is convenient and cheap to do incremental work at that time. By limiting the amount of incremental work done at each minor collection, we can keep pauses brief, within a factor of, say, three times as long as for a minor collection alone.

We use the strategy, described above, of only updating the root set when the GC is complete. The mutator can therefore only see from-space objects. We use the store list during each GC increment to update to-space versions and rescan them if necessary. The SML/NJ compiler version 66 keeps a log of all mutations which store pointers, for use by the generational collection algorithm. We modified the mutation log to include all mutations, so that the incremental collector can update to-space. This avoided the need to modify the compiler to add a write-all-replicas protocol.

In order to ensure that the garbage collector terminates, we must guarantee that all live data will be replicated in to-space before from-space overflows with new data copied by the minor collections. We want to restrict the amount of GC work done in each increment, but still ensure that a 'flip' takes place before from-space is full. Otherwise, when from-space fills, the incremental collector will have to perform a large amount of remaining gc work,

	#minor pauses	mean pause	modal pause	max. pause	90% below	#major pauses	mean pause	max. pause	total GC
orig	5422	17ms	15ms	734ms	45ms	48	2.2s	5.0s	201s
incr	5422	57ms	46ms	499ms	93ms	—	—	—	312s

Table 1: Pause timings for stop-and-copy vs. incremental collectors.

which will be tantamount to a major garbage collection pause.

In the prototype implementation, we guarantee that this will not happen by requiring the incremental collector to copy more objects into to-space than were added to from-space by the minor collection. Therefore, the duration of the incremental collector's pauses can be controlled by adjusting the size of the new-space and the amount of additional incremental copying done.

4 Measurements

The initial performance measurements for our prototype implementation are shown in table 1. The table describes the garbage collector pauses which occurred during a single test case. The test case compiled a significant part of the SML/NJ compiler, and was run without paging activity on a DECstation 5000/200 equipped with 64 Mb of main memory. The incremental collector completely eliminates the major collection pauses of 2 to 5 seconds with which every SML/NJ user is aggravatedly familiar.

The minor pauses measured for the original collector represent the delay caused by a collection of old-space into new-space. The minor pause time for the incremental collector includes the generational collection of old-space into new-space and also the work done by the incremental algorithm transporting objects in the from-space (old-space) to the to-space.

The statistical distribution of the minor pause times are both unimodal, with pronounced modes at at a pause time of less than 50ms, but with a long tail to several hundred milliseconds. Our collector increases the mode, but its performance appears to be interactive enough to remain acceptable to users.

The measured mean pause time for our collector is 57 milliseconds. We expect to reduce that figure to 50ms or less by varying the control parameters of our implementation. Reducing the size of the new-space and the fraction of incremental work done will shorten these pauses. Because our collector is incremental, we can also cut short the incremental collection activity if it becomes too lengthy.

The total garbage collection time is increased by more than 50% relative to version 66 of the SML/NJ. We anticipate being able to reduce this to approximately 10% by simple optimizations of our existing code (we believe most of this increase is due to the fact that

	#objects copied	total size	overhead bytes	overhead % heap
all objects	27M	344Mb	108Mb	24%
mutable only	1.76M	18Mb	7Mb	2%

Table 2: Space overhead of forwarding words for incremental collector.

the prototype implementation performs a 'flip' operation twice as often as the standard algorithm. There is no mutator time overhead in the current implementation.

Table 2 shows the total space overhead of our system. The total size measurements given in the table do not include the overhead for forwarding words, and the percentage figure measures the amount of overhead bytes as a percentage of the total heap size, including overhead. The prototype implementation uses a separate forwarding word for every object, which results in a very high space overhead of 24% because a majority of objects are two-word records ('cons cells') with a header word. However, we can reduce the space overhead by storing the forwarding pointer and the header information in the same word. In this scheme, a replicated object has header information on only the newest copy. Any operation which needs the header information must follow forwarding pointers to locate the newest copy of the object. In the write-newest protocol, this optimization can be applied to all objects, eliminating the space overhead entirely.

However, in the write-all consistency protocol, even the newest replicas of mutable objects require 'backwarding pointers', so this optimization cannot be applied to them. In this case the space overhead would be reduced to just 2% of the heap, as shown in the table. Certain operations such as `size` would need to follow the forwarding pointer chain, as well as other low-level run-time operations such as tag checks.

5 Concurrent Collection

The same technique is applicable to a concurrent system, in which the collector and the mutator run in parallel, as separate threads of a single process. This is only an advantage in multi-processor systems, when the collector may be running on one processor while the mutator (or mutators) is running on the others—in single-processor systems one is merely sacrificing control over when the collector runs, which is pointless.

In a concurrent system, not only must the semantic operations of the collector and mutator be independent, as discussed above, but the individual machine instructions of each must be interleavable. This is a much stronger condition, but it is not hard to satisfy in a concurrent version of the incremental collector described above.

First consider whether running our prototype incremental collector concurrently with the mutator would produce read/write conflicts. The mutator only reads or writes from-space replicas. The collector reads from-space replicas, but writes only to-space replicas. The collector also writes the forwarding words of from-space replicas, which the mutator does not access. Thus the collector will not interfere with the mutator. If the forwarding word and the header word are merged, then the collector and the mutator could conflict while accessing this word. However, as long as the collector can atomically update the header word to install the forwarding pointer, there is no danger. The mutator will either read the from-space replica's header word before it is overwritten, or follow the forwarding pointer to the to-space replica.

Now consider whether the mutator will interfere with the collector. It can only interfere by writing a word the collector is reading. But at worst this would cause the collector to copy the wrong value to to-space and at some point this mistake would be corrected in the process of updating to-space to reflect mutator writes. Thus the mutator does not interfere with the collector.

Almost all of the synchronization needed to make our prototype incremental collector concurrent is already present in the incremental collector, because the effects of mutator stores are communicated to the collector indirectly through the store list. Implementing a concurrent collector is simply a matter of managing the handoff of the current roots and the store list, and synchronizing to forward the root pointer set when the collection terminates.

6 Related Works

Real-time incremental or concurrent garbage collection has been the goal of many research projects in the past. Recent work includes that by Ellis, Li, and Appel [2], which exemplifies the use of the virtual-memory system to control the GC behavior, and Halstead [5], using hardware improvements. The first real-time copying collector, by Baker [3] requires special hardware, and paved the way for many other such systems. Some existing algorithms work on stock hardware without operating systems support, such as those by Brooks [4] and later North [8], but none of these show such small time and space overheads as our technique.

7 Future Work

Since the overhead for this new technique appears to be acceptable, we believe it will be useful when applied to several other interesting GC-related algorithms. These other algorithms can all make use of copying to achieve some useful end other than collecting garbage, and may be able to share some runtime and/or storage costs with the garbage collector.

One such algorithm is used to implement persistent storage. One of us has implemented a persistent storage system based on copying objects from the heap into a persistent heap [7]. A major performance bottleneck is the need to scan the entire heap for pointers to objects which have been copied. Nondestructive copying will eliminate this scan.

We are also interested in using copying to implement mechanisms for distributed computing, such as those required by object repositories. In these distributing computing systems, data which will be replicated at a remote machine is copied into a message buffer, linearizing it for transmission purposes. Again nondestructive copying will greatly lessen the overhead of such copies. Also, we anticipate a simple interface between the local GC described here and the global (distributed) GC required in such a system.

A final possibility is the technique of delayed hash consing. Here the system tries to detect if two (immutable) objects are identical. If they are then they can be merged. This merge can be implemented by nondestructively adding a forwarding pointer from one object to the other. This technique may greatly reduce the amount of heap space needed.

We are extending our implementation in these directions and exploring some ideas for "opportunistic" GC [9], in which the timing of garbage collections is chosen to minimize disruptiveness. We are investigating triggering GC within the user-interaction loop, immediately before prompting for input, and after long waits for input. As a start, we are adding some very simple code to disable the incremental technique when the mutator is compute-bound, reverting to the more efficient stop-and-copy collection, the pauses of which will not be noticed during the compute delay.

8 Conclusions

We have introduced a promising new copying GC technique, replication-based copying. This technique is especially well suited to languages like SML where mutations are rare. We have implemented a simple incremental GC for SML/NJ based on this technique and have obtained preliminary data showing our idea to be workable. We are continuing work to make related algorithms equally practical.

Acknowledgments

Scott Nettles and James O'Toole would like to thank DEC's Systems Research Center for support as summer interns, during which time this idea was originally conceived. Scott Nettles and David Pierce would like to thank Peter Lee for support with the implementation. Thanks also to John Reppy for his suggestion to merge the forwarding pointer and header word. Greg Morrisett provided many hours of helpful conversation. Thanks to Penny Anderson, Mark Sheldon, Ellen Siegel and the Venari group for proofreading.

References

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

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[3] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, 1978.

[4] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 256–262, 1984.

[5] Robert H. Halstead, Jr. Implementation of multilisp: LISP on a multiprocessor. In *ACM Symposium on LISP and Functional Programming*, pages 9–17, 1984.

[6] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246. ACM, August 1984.

[7] Scott M. Nettles and J.M. Wing. Persistence + Undoability = Transactions. Technical Report CMU-CS-91-173, Carnegie Mellon University, August 1991.

[8] S. C. North and J.H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 113–133. Springer-Verlag, 1987.

[9] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *Proceedings of ACM SIGPLAN 1989 Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1989.