Real-Time Replication Garbage Collection

Scott Nettles and James O'Toole

Appears in PLDI '93

Abstract

We have implemented the first copying garbage collector that permits continuous unimpeded mutator access to the original objects during copying. The garbage collector incrementally replicates all accessible objects and uses a mutation log to bring the replicas up-to-date with changes made by the mutator. An experimental implementation demonstrates that the costs of using our algorithm are small and that bounded pause times of 50 milliseconds can be readily achieved.

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

1 Introduction

Garbage collector pauses are always annoying, but for many applications they are intolerable. For example, smoothly tracking a mouse in an interactive graphics application requires pause times of 50 milliseconds or less[5]. For garbage collection to be useful in applications with real-time constraints, pause times must be bounded. We have implemented a garbage collector which provides real-time collection using a new technique. This technique is efficient and can provide such short bounded pause times.

The key method used by real-time collectors is *incremental* collection, in which garbage collection work is interleaved with mutation. For incremental collection to be possible, the garbage collector must sometimes suspend its work and permit the mutator to run, even though the collection algorithm has not completed.

Previous work on incremental collection has focused on techniques that required either special hardware or operating system support [12, 1], or in which the extra overhead for the mutator was potentially very high [4, 16]. These algorithms are variants of Baker's algorithm[2] which uses a to-space invariant. The to-space invariant requires that the mutator use only pointers into to-space. The cost of enforcing this restriction leads to the need for special hardware or operating system support.

Instead of a to-space invariant, our method uses a from-space invariant which requires that the mutator use only the original from-space objects. The garbage collector incrementally builds a consistent replica of the accessible objects. The modified collector invariant decouples the execution of the garbage collector from the mutator, and permits the collector great flexibility in scheduling its replication activity.

An early prototype of our implementation[14] demonstrated that replication can be used for incremental collection but did not provide real-time response. It also did not allow for a careful comparison of performance with stop-and-copy collection. To demonstrate that our technique is practical and feasible for real-time collection, we have implemented several variants of this technique for Standard ML of New Jersey (SML/NJ). Our experimental collectors provide excellent performance with little runtime overhead. The real-time collector provides bounded pause times within the limits needed by interactive applications.

In the sections that follow, we introduce our general approach, based on the new invariant. We provide a high-level

Authors' addresses: nettles@cs.cmu.edu, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213. (412)268-3617
otoole@lcs.mit.edu, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139. 617-253-6018

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, by the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0128, and by the Department of the Army under Contract DABT63-92-C-0012.

explanation of our method and its fundamental correctness conditions. We then discuss the details of our experimental implementation and its real-time performance goals. We present experimental results that show that the cost of the technique is low in practice and that pause times are well controlled. Finally, we discuss possible improvements to the implementation and suggest areas for further work. We assume that the reader is familiar with the basics of copying and generational garbage collection, a survey may be found in Wilson[20].

2 Real-Time Replication Garbage Collection

Incremental collectors permit the mutator to resume execution before the collection has completed. The operations of the collector and the mutator may be interleaved. Thus the effects of the garbage collector must not be observable by the language primitives used by the mutator.

The standard technique used by copying garbage collectors to copy an object destroys the original object by overwriting it with a forwarding pointer. Therefore, incremental collectors that use the standard copying technique require the mutator to use only the relocated copy of an object. Enforcing this to-space invariant typically requires a "read-barrier", as shown in figure 1. The implementation of read-barriers has consequently been the focus of much effort in incremental garbage collection work.

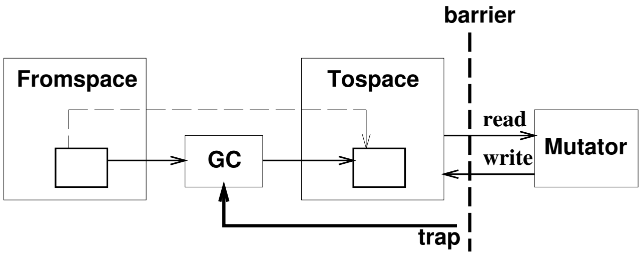

Figure 1: A Read Barrier Protecting Tospace

In contrast, our technique requires the collector to replicate live objects without destroying the original objects. The mutator is able to continue accessing the original objects. This allows us to eliminate the read-barrier and modify the to-space invariant. However, our method requires a write-barrier because the mutator may continue to modify objects after they have been replicated. A write-barrier is much less costly to implement than a read-barrier[10].

Conceptually, the standard *Copy* operation can be made non-destructive by reserving an extra word in the object which is not observable by the mutator and which is used to store the forwarding pointer. In our algorithm, the goal of the collector is to successfully replicate all live objects which are present in "from-space" by creating corresponding objects in "to-space". When this task is complete, the collector replaces the roots of the mutator with pointers to their corresponding replicas in to-space, discards from-space, and terminates.

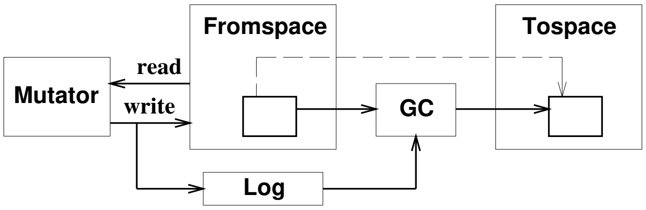

Figure 2: Replication and The Mutation Log

2.1 Mutations are Logged

After the collector has replicated an object, the original object may be modified by the mutator. If this happens, then the same modification must also be made to the replica before the mutator switches to using the replica. Therefore, our algorithm requires the mutator to record all mutations in a "mutation log", as shown in figure 2. The collector must use the mutation log to ensure that all replicas reach a consistent state by the time the collection terminates. The collector does this by processing the log entries and applying mutations to the replicas.

When the collector modifies a replica which has already been scanned, it rescans the replica to ensure that any object referenced as a result of the mutation is also replicated in to-space. After a log entry has been processed in this way it may be discarded. The cost of logging and of processing the mutation log depends on the application, and also the implementation of logging. Mutation logging works best when mutations are infrequent or can be recorded without mutator cooperation. Mutation logging is also attractive whenever writes are already expensive or a mutation log is required for other reasons. For example, generational collectors, persistent data, and distributed systems usually make mutation operations more expensive[13].

2.2 The Collector Invariant

The invariant maintained by the replication-based garbage collector is that the mutator can only access from-space objects, that all previously scanned objects in to-space contain only to-space pointers, and that all to-space replicas are up-to-date with respect to their original from-space objects, unless a corresponding mutation is recorded in the mutation log.

This from-space invariant differs from standard collector invariants because it requires the mutator to continue using the original from-space objects. The from-space invariant permits the replicated objects to be in an inconsistent state, as long as the inconsistencies are recorded in the mutation log. It is because of these inconsistencies that the mutator must continue to use only the from-space objects until the collection algorithm completes.

2.3 The Completion Condition

The collector has completed a collection when the mutation log is empty, the mutator's roots have been scanned, and all of the objects in to-space have been scanned. When these conditions have been met, the invariant ensures that all objects reachable from the roots have been replicated in to-space, are up-to-date, and contain only to-space pointers. When the collector has established this completion condition, it atomically updates all roots of the mutator to point at their corresponding to-space replicas, discards the from-space, and renames to-space as from-space.

2.4 Limiting Pause Times

In order to guarantee that the garbage collector will only pause the mutator for a bounded time, the collection algorithm must somehow limit its execution. If the algorithm has not completed when the maximum pause time has passed, the collector must stop work and permit the mutator to continue executing.

The replication-based algorithm described here can suspend execution at any time, and is suitable for concurrent implementation (see section 6). However, the actual mechanisms that can be used to control the duration of garbage collection pauses are implementation dependent, and are discussed in section 3.3.

2.5 Optimization Opportunities

The from-space invariant used by this algorithm is very weak, in the sense that the collector never needs to work on any particular task in order to allow the application to execute. The collector only needs to replicate all the live data soon enough to terminate and reuse the memory in from-space before the application runs out of memory.

In the algorithm of Appel, Ellis, and Li[1], the application may frequently be blocked waiting for the collector to copy the objects that it must use. We believe that the flexibility of our invariant offers potentially important optimization opportunities to any replication-based implementation. For example, the collector can copy objects in essentially any desired order.

This freedom in copying order could be used to increase locality of reference or to change the representation of objects stored in a cache[17]. Another way that copying order freedom can be exploited is by concentrating early replication work on objects reachable from particular roots. Particular roots may be more likely to change than others, so copying them later could reduce the amount of latent garbage copied by the collector.

Also, if no mutable objects have been replicated then the collector need not apply mutations to replicas. The collector could choose to concentrate early replication effort on only immutable objects, and thereby delay the need to process the log until the last possible moment. The actual copying of an object can be delayed until the object is scanned using an optimization suggested by Ellis[9]. The collector could replicate mutable objects into a segregated portion of the to-space, and delay copying and scanning mutable objects as long as possible. Mutation log entries created before the first mutable object was actually copied could be discarded.

3 Implementation

To test the practicality of our new approach, we implemented a real-time garbage collector using the replication-based algorithm. The collector is designed to show that pause times can be limited and to permit accurate comparison with an existing stop-and-copy collector. The experimental collector operates in the runtime system of Standard ML of New Jersey, which uses a two-level generational heap design. The collector uses the replication algorithm for both minor-incremental and major-incremental collections, which share the implementation of object replication and mutation logging.

The major and minor collectors differ in when collections are initiated and how their execution is controlled. The real-time collector can be operated with the incremental algorithm enabled for one or both of the two generations present in the original SML/NJ collector. The experimental results presented in section 4 use results from various configurations to quantify the costs of the replication method and the pause time behavior for several benchmarks.

3.1 The SML/NJ Runtime System

We chose SML/NJ (version 0.75) for our work primarily because it has a good compiler and a simple generational garbage collector. Since the runtime system has no stack, heavy demands are placed on the storage allocation and reclamation system. Providing real-time garbage collection is therefore challenging. However, the SML language encourages a mostly functional programming style, so mutations are rare. This is advantageous to our technique.

In the SML/NJ collector, there are two generations, old and new. Objects are allocated in new-space. When new-space fills, a minor collection is initiated which copies the live data into old-space. The size of the new-space is controlled by the runtime parameter N. Another parameter, O, controls the initiation of a major collection. When the amount of memory copied into the old space by minor collections exceeds O, a major collection occurs, copying all live data into to-space and then exchanging the roles of to-space and old-space.

3.2 Replication and Logging

Generational collectors must identify mutations that might create pointers from old-space into new-space. The SML/NJ

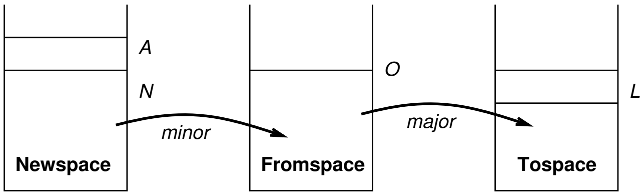

Figure 3: SML/NJ Heaps with GC Parameters

collector uses a log called the "storelist" to track such mutations. The replication-based algorithm needs to log all mutations to the contents of a previously replicated object. We modified the SML/NJ compiler so that all mutations are recorded in its storelist.

The most straightforward implementation of non-destructive copying is to store a forwarding pointer to the replica in an extra word in each object. However, measurements of the SML/NJ system suggest that most objects are only three words long, including the object header word used to store certain type and length information. This means that the overhead of allocating an extra word per object would be prohibitive. Therefore in our implementation we overwrite the object header word with the forwarding pointer.

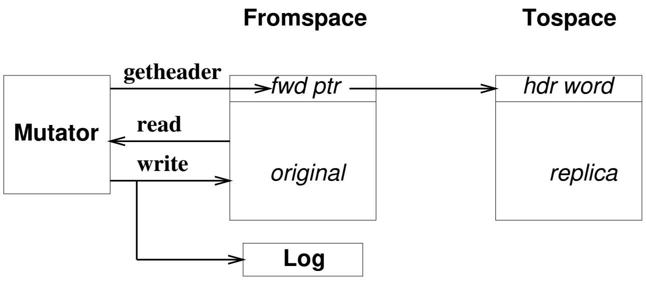

Figure 4: Getheader Operations Follow the Forwarding Word

Our implementation must ensure that the replacement of the object header word is not observable by the mutator. The mutator accesses the object header word only during the polymorphic equality operator and certain type-specific length operations. We modified the compiler to implement these operations by checking for the forwarding pointer and reading the object header word from the replica when necessary. This slows down these operations without imposing an overhead on normal read access to object contents. We also modified all runtime system call operations which modify ML data (e.g. I/O primitives) to perform appropriate logging.

3.3 Controlling Pause Duration

In SML/NJ the pauses due to major collections are the longest and most disruptive. The real-time collector uses the incremental algorithm to eliminate these pauses. The incremental algorithm is enabled when the O parameter first triggers a major collection. Each time a minor generation collection occurs, the major-incremental collector performs some work after the minor collector terminates. This approach slightly increases the pause times for minor collections and completely eliminates the more disruptive major pause times.

In order to control the total pause time caused by the combined minor and major collections, the incremental algorithm restricts the amount of work it does using a parameter, L. The L parameter limits the total amount of memory copied by the collections during a single pause. However, only the incremental algorithm respects the work limit L.

If the minor collection has exceeded the copy limit L, then the major-incremental algorithm processes the mutation log, but does not perform any additional replication work. Therefore, when L is very small, the major-incremental collection may not terminate. There is an implementation-dependent lower bound for L that will guarantee termination, but such a conservative completion strategy increases the total cost of garbage collection[14].

3.4 The Real-Time Collector

Minor collection pauses are usually short, but may not be bounded by L. Therefore, to bound these collection pauses, the real-time collector uses the incremental algorithm for minor collections as well as for major collections. When the work limit L is exceeded during a minor collection, the collector suspends execution and returns control to the mutator. In this case, new-space must be expanded in order for the mutator to allocate more objects. Currently the implementation expands new-space by a parameter A, whenever any incremental collection is awaiting completion.

For a minor collection the log contains pointers into the old heap which are roots. Our technique requires that all roots be atomically updated at the time of a flip. For minor flips this requires an additional traversal of the log to update the roots in the old heap. At this point the log has been filtered so that it includes only the pointer related mutations as only these entries are roots.

Our real-time collector does not yet offer an absolute guarantee of bounded pause times. First, the implementation makes no attempt to shield the mutator from page faults, I/O, swapping, and other system effects. Second, the current implementation does not incrementally copy a single large object, nor does it incrementally process the mutation log. Therefore, these operations can exceed the work limit L. If necessary, these operations can easily be implemented so that they are performed incrementally and respect the work limit.

4 Performance

The goals of the performance study were to demonstrate that pause times are bounded and to measure the overheads imposed by our technique. The measured performance is good; the real-time collector achieves short pause times with an acceptable overhead. In addition to the basic measurements of pause and execution times, we also undertook a series of experiments to quantify the contributions of various factors to the overhead.

4.1 Benchmarks

Three benchmarks were used to test our implementation. Each was chosen because it stressed the memory management system in a different way. All benchmarks require several minutes to execute and require many major and minor garbage collections during execution. See [6] for more details about these benchmarks.

- *Primes* is a prime number sieve implemented in a simple lazy language which is in turn interpreted by an SML program. It allocates memory at a very high rate (approximately 10Mb/sec), but few objects survive garbage collection. It is typical of compute-bound programs in SML/NJ.

- *Comp* is the SML/NJ compiler compiling a portion of itself. This is the most realistic benchmark; the SML/NJ compiler is a large optimizing compiler and is in daily, production use. *Comp* does not allocate as much data as *Primes*, but more of it survives collections. The amount of live data fluctuates depending on the phase of the compilation.

- *Sort* is a sorting program based on futures which are in turn implemented using SML threads. *Sort* does more mutation than a typical SML program and it creates a large amount of live data. Both the large mutation rate and the substantial survival rate make this a challenging example for our technique.

All benchmarks were executed on a Decstation 5000/200 with 64 Mb of physical memory running the Mach 2.5 operating system. The system has a 25MHz clock and separate 64Kb instruction and data caches. For the pause time measurements the system clock resolution was set to 4ms.

4.2 Parameter Settings

To test our system we chose values for the parameters N, O, L and A. For O we used the values 5Mb and 1Mb. The larger value is typical for running SML/NJ in our environment, while the lower setting was chosen to emphasize overheads present in major collections. For N we chose 1Mb and 0.2Mb. Again, the larger value is typical for use with the stop-and-copy collector. The lower setting was chosen because it allowed us to achieve pause times of 50 milliseconds yet still have the collector terminate. We chose 50 milliseconds as our target pause time because this is the maximum pause time which will allow an interactive program to smoothly track a mouse[5].

When N is 0.2Mb we set L to 0.1Mb and when N is 1Mb we set L to 0.5Mb. The value of 0.1Mb was chosen because that is approximately how much data the collector can copy in 50 milliseconds, while the value of 0.5Mb was chosen somewhat arbitrarily. A was chosen to be L/2. This guarantees that the collector will make progress when an incremental collection is active.

We also ran our benchmarks with other values of L but those results are not particularly illuminating and have been omitted due to lack of space (see [19] for more details). In general, as L increases, pause times increase and duration of collections decrease. Any overheads that are related to collection duration decrease.

In this study we are concerned with quantifying the overheads of adding replication-based collection to the system, rather that studying what polices should be used to control such a collector. Since the choice of policy can strongly influence performance we controlled for it in the following way. Using the parameters above, the real-time collector was run in such a way as to produce a script of exactly when it flipped and how much new allocation space it returned. These scripts were then used to replay these policy decisions for all benchmark runs. This ensures that the differences we measured were those imposed by our mechanism rather than variations caused by different policy decisions. We measured the overhead caused by this replay method and found it to be smaller than the margins of error (approximately 2%) typical in our benchmark runtimes.

4.3 Pause Times

The primary motivation for using a real-time garbage collector is to provide bounded collector pause times. In this section we present the measurement results for our benchmarks.

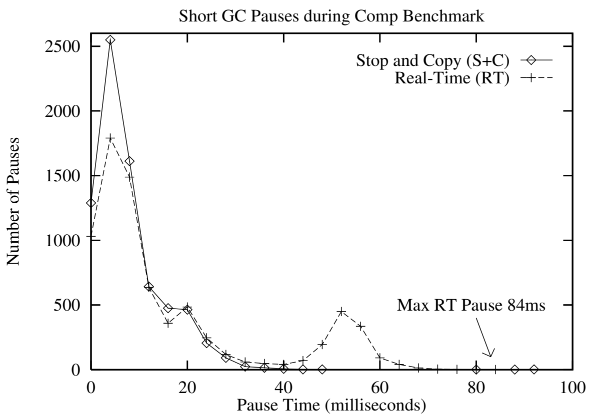

Figure 5: Compiler Benchmark (N=0.2Mb, O=1Mb)

Figure 5 shows a plot of pause times for both the stop-and-copy collector and the real-time collector running *Comp* with O=1Mb and N=200Kb. The real-time collector has a maximum pause of 84ms and the peak at 50ms represents

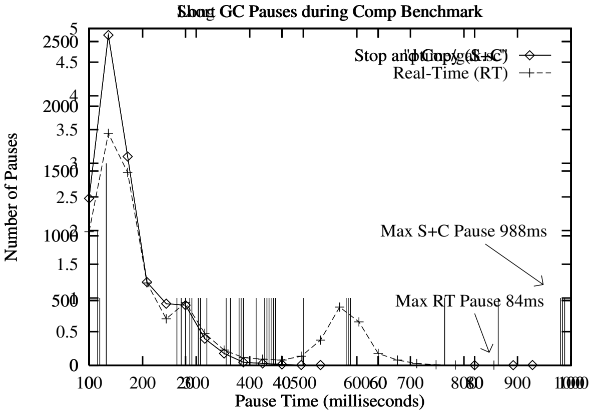

Figure 6: Compiler Benchmark (N=0.2Mb, O=1Mb)

the result of truncating the longer stop-and-copy pauses to that value. Figure 6 shows the longer pauses of the stop-and-copy collector which our technique eliminated. Note that during this 245 second long run the stop-and-copy collector causes a pause longer than 0.5 seconds approximately every 20 seconds.

Table 1 summarizes the rest of the pause time data. The table shows the the median pause time, the 99% percentile pause time, and the maximum pause time. These measures show that the real-time collector is successful at bounding the pauses, and that in exchange the duration of many shorter pauses increase slightly.

		Primes					
O	N	Stop+Copy			Real-Time		
Mb	Mb	50%	99%	Max	50%	99%	Max
1	0.2	12	96	106	12	52	66
1	1.0	28	96	102	32	136	142
5	0.2	12	16	106	12	20	58
5	1.0	28	40	102	32	44	158
		Comp					
O	N	Stop+Copy			Real-Time		
Mb	Mb	50%	99%	Max	50%	99%	Max
1	0.2	8	36	990	12	64	86
1	1.0	28	148	934	36	292	314
5	0.2	12	36	778	12	60	74
5	1.0	32	120	450	36	260	294
		Sort					
O	N	Stop+Copy			Real-Time		
Mb	Mb	50%	99%	Max	50%	99%	Max
1	0.2	24	232	342	36	72	78
1	1.0	108	316	350	116	328	342
5	0.2	28	44	338	32	68	74
5	1.0	100	220	314	108	320	334

Table 1: Garbage Collection Pause Times (msec)

4.4 Elapsed Times

The real-time collector is clearly successful at providing bounded pause time, but at what cost in performance? A diagram of the component costs of elapsed time in our implementation is shown in figure 7. Several of the costs, such as latent garbage and the generational scan of pointer mutations, are shared by any incremental and/or generational collector, and are not peculiar to replication gc. These overheads will be explored in some detail in the following section.

To determine total overhead, we measured our benchmarks using a variety of configurations: the full real-time collector, the real-time collector with only minor collections done incrementally, the real-time collector with only major collections done incrementally, the stop-and-copy collector with the compiler changes for real-time collection and the stop-and-copy collector without those modifications.

Figures 8, 9, and 10 summarize this data. In general the overhead for the most realistic benchmark, *Comp*, is under 10%. We consider this overhead acceptable. Even *Sort*, the most demanding benchmark, shows overheads under 25%. Note that the cost of doing minor-only incremental is essentially the full cost of real-time collection. We do not have a good explanation of why in some cases this cost is larger than for the full real-time collector, but in later sections we

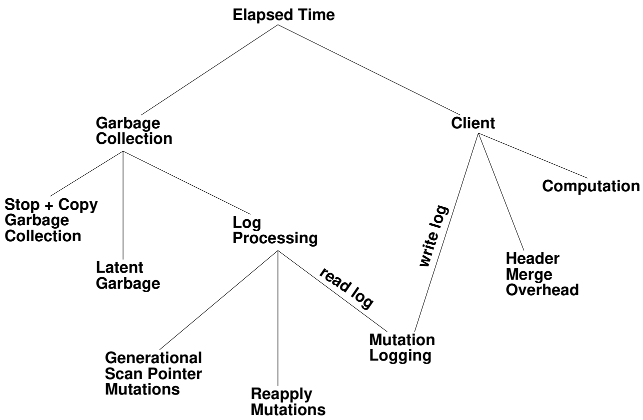

Figure 7: Components of Execution Time

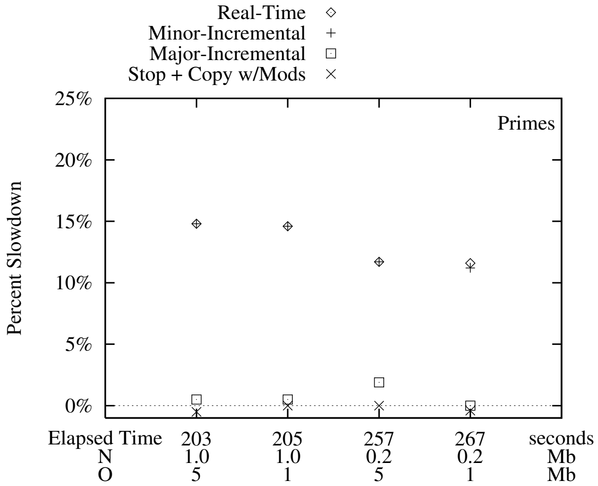

Figure 8: Primes Benchmark: Elapsed Times

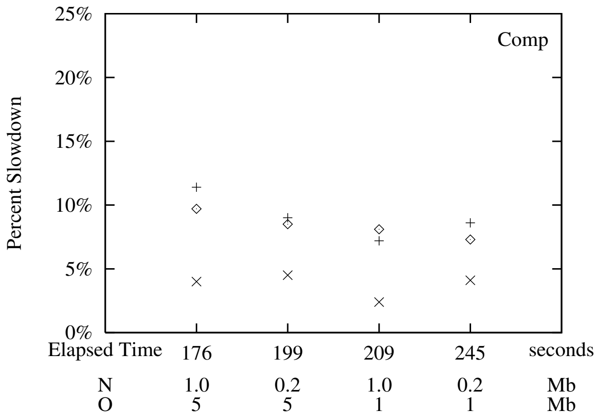

Figure 9: Compiler Benchmark: Elapsed Times

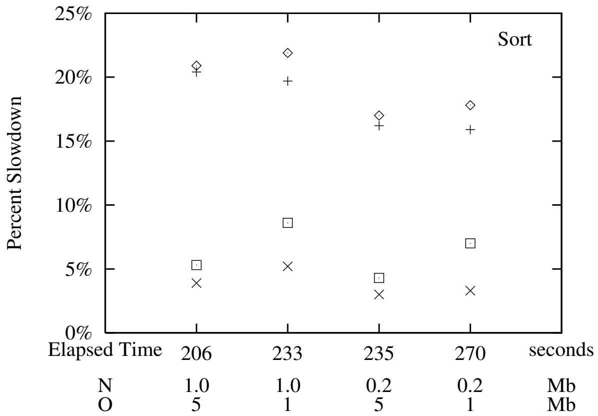

Figure 10: Sort Benchmark: Elapsed Times

will see why it should be essentially the same.

4.5 Overheads Due to Compiler Modifications

The compiler modifications which were needed to support our technique impose two overheads on the compiler: testing headers for forwarding words and adding extra records to the log. The extra log records also impose an overhead on the garbage collector. To measure these overheads we ran the stop-and-copy collector with each of these modifications enabled separately. We were unable to measure the cost of testing for the presence of forwarding pointers, leading us to conclude that it is negligible, or at most a few percent. The cost of the extra log entries accounted for essentially all of the overhead due to the compiler modifications.

Examining the entries in figures 8, 9,and 10, we see that the overhead of these additional records is essentially 0% for *Primes* and near 5% for *Comp* and *Sort*. This is easily explained by following observations. *Primes* does almost no mutations and so should see no overhead. *Sort* mostly mutates integer references which the compiler normally does not place on the log, while *Comp* contains many mutations to byte data which is likewise normally not logged. Perhaps the overhead due to these non-pointer mutations could be reduced either by forwarding them to to-space during mutation instead of logging, or by using other logging techniques. We have not yet made separate measurements of the mutator and garbage collector costs.

		Primes			
O (Mb)	N (Mb)	CR (secs)	%CR	CF (secs)	%CF
1	0.2	7.0	2.3	14.1	4.7
1	1.0	6.4	2.7	21.2	9.0
5	0.2	5.7	2.0	14.7	5.1
5	1.0	6.1	2.6	21.4	9.2

		Comp			
O (Mb)	N (Mb)	CR (secs)	%CR	CF (secs)	%CF
1	0.2	3.3	1.3	3.7	1.4
1	1.0	3.0	1.3	4.6	2.0
5	0.2	3.1	1.4	3.9	1.8
5	1.0	2.5	1.3	4.7	2.4

		Sort			
O (Mb)	N (Mb)	CR (secs)	%CR	CF (secs)	%CF
1	0.2	5.3	1.7	10.2	3.2
1	1.0	4.8	1.7	14.2	5.0
5	0.2	4.3	1.6	10.0	3.6
5	1.0	4.6	1.8	13.9	5.6

Table 2: Log processing costs

4.5.1 Processing the Mutation Log

Two costs of our technique which are not shared by other incremental or generational collectors are the costs of reapplying mutations to to-space and of atomically updating roots found in the log during a minor flip. To measure this cost we repeatedly processed the mutation log in order to increase the overhead to a measurable level, both with and without the reapplication of mutations enabled. This allowed us to distinguished the two cases described above.

Figure 2 presents these results. CR is the cost of reapplying the mutations in seconds and %CR is the percentage cost relative to that of the real-time collector. CF is the cost of atomically flipping the roots in seconds and %CF is percent relative to real-time collection. The cost of actually forwarding the stores is generally small and is always smaller than the cost of the atomic flip. If the minor collection need not be incremental then the flip cost may be avoided. It is also possible that improving the data structures used to represent the mutation log would reduce this cost. This data, along with the measurements of overheads due to compiler modifications also reveals why the minor-incremental collector has essentially the same performance as the real-time collector. The minor-incremental collector shares all these costs with the real-time collector, and these costs dominate.

4.5.2 Latent garbage

One potential overhead for any incremental collection algorithm is that data considered live by the collector may die before the collector terminates. This latent garbage increases the overhead of collection since it must be copied and it leaves less free memory to be used by the allocator. To measure this effect we compared the amounts of data copied by the stop-and-copy collector and the real-time collector. Because the flips and allocation amounts were exactly synchronized the difference between the two is the latent garbage.

Table 3 shows our measurements of latent garbage both in Kb (G) and as percentage of the true live data (%G). We also estimate the cost of copying this much data in seconds (CG). Our cost estimates are based on measurements of the rate at which the collector copied data. These measurements show a typical copying rate of approximately 2Mb/sec. This correlates well with the fact that L = 100Kb gives 50 millisecond pause times.

We see that the amount of latent garbage is generally a small fraction of the total amount of data copied. The absolute amount of latent garbage goes down both with increasing N and O. For O this is because there are fewer collections to create latent garbage. For N it is because for these measurements increasing N increases L. When L increases the incremental algorithm terminates more quickly and there is less latent garbage. These measurements suggest that latent garbage is not an important contributor to the overheads in our current tests. However, different policies with respect to when to begin collection and how rapidly to complete it may make this effect more important.

		Primes		
O (Mb)	N (Mb)	G (Kb)	%G	CG (secs)
1	0.2	739	0.5	0.4
1	1.0	0	0.0	0.0
5	0.2	159	0.1	0.1
5	1.0	0	0.0	0.0

		Comp		
O (Mb)	N (Mb)	G (Kb)	%G	CG (secs)
1	0.2	7556	3.6	3.9
1	1.0	6247	4.1	3.2
5	0.2	5561	1.6	2.8
5	1.0	1723	1.9	0.9

		Sort		
O (Mb)	N (Mb)	G (Kb)	%G	CG (secs)
1	0.2	5561	1.6	2.8
1	1.0	4115	1.5	2.1
5	0.2	1237	0.4	0.6
5	1.0	998	0.4	0.5

Table 3: Latent garbage amounts

5 Related Work

The real-time copying collector by Baker[2] first proposed the condition that object accesses somehow be redirected to the relocated copy of the object. The work of Ellis, Li, and Appel[1] exemplifies the use of virtual memory traps and other operating system support to implement similar conditions. A method due to Brooks[4], and later implemented by North[16], requires the mutator to follow a forwarding pointer which leads to the relocated object. Nilsen[15] describes a software implementation of Baker's algorithm which is designed for an environment in which strings are heavily used. The overhead of his technique seems to be prohibitive in a more general context.

Recent work by Boehm, Demers and Shenker [3] on a concurrent mark-and-sweep collector promises real-time performance. As in our algorithm, a form of mutation logging is used by the collector to track changes made by the mutator. The mutation log is implemented by periodically sampling the dirty page bits maintained by the virtual memory system. Live objects are not relocated, but rather are marked non-destructively. Therefore, GC efforts can be interleaved freely with mutator operations, but the compaction possible in copying collectors is unavailable. The authors observed the possibility of using a from-space invariant for a copying collector.

Two recent collectors for ML are quite closely related to ours. However, both depend on the semantics of ML more closely than our work.

Doligez and Leroy[8] have implemented a concurrent collector which uses a mixed strategy to provide collection for a multithreaded version of CAML. Immutable objects are allocated in private heaps which are collected by a replication-based stop-and-copy collector. This collector copies values into a shared heap which is collected using a concurrent mark-and-sweep algorithm based on Dijkstra[7]. To avoid the issue of inconsistent mutable values all such objects are allocated in the shared heap. If mutations to such a value cause other values which currently reside in a private heap to become reachable from the shared heap, these values are copied into the shared heap at the time of mutation. The use of replication-copying allows the original owner of these values to continue to access the copy in the private heap.

Huelsbergen and Larus[11] have recently built a concurrent collector for SML/NJ which uses replication-based copying. They use a to-space invariant and a consistency protocol which requires that the mutator read and write the to-space version if it exists. Our previous work[14] considers this protocol, the from-space invariant and other consistency options for replication garbage collection. In addition to maintaining a to-space invariant, their collector has a number of other differences from our own. Their collector is not generational which leads to a slow down relative to the original SML/NJ collector (despite the use of multiple processors) and makes it difficult to directly assess the overhead of their technique. Less important their implementation does not merge forwarding pointers with header words and thus has a substantial space penalty. Also their implementation is more closely tied to the semantics of mutable values in SML and to the details of their processor memory consistency model. We hope to implement their technique along with others from[14] in the context of a concurrent version of the collector described here. This will allow a quantitative comparison of these options.

6 Future Work and Conclusions

We are actively extending the current work in several directions. Our experimental implementation is perfectly suited for use as a concurrent collector. The replication primitive can be interleaved freely with mutator activity, as long as the memory system provides single-word memory atomicity. Synchronization between the collector and the mutator is only required for transferring the mutation log and updating the roots. The concurrent version of this implementation is working and initial performance measurements can be found in [18].

Replication-based copying is also a promising approach for use in heap based transaction systems. In addition to the advantages concurrent collection has for such systems a further advantage is that such systems also must log all mutation to transactional data. Replication-copying is thus even more attractive. We are currently working on extending the concurrent collector implementation to support a transactional persistent heap[13].

An area which we have not yet explored is what policies are best suited for use with our collector. For example in an interactive system, our technique would allow collection to proceed while the system was waiting for input. If such pauses are long enough or frequent enough collection may become essentially free. On the other hand when running compute bound jobs it may make no sense to pay even the small cost of incremental collection since such jobs already introduce lengthy pauses.

Conclusions

We have designed and implemented a real-time garbage collector using a new replication-based invariant. This invariant eliminates the need for a read-barrier, and therefore enables real-time garbage collection on stock hardware with low mutator overhead. We have examined various overhead costs in an implementation that relies on mutator cooperation for logging. Our experimental implementations show that controlled pause times of 50 milliseconds can be readily achieved in practice; this satisfies the requirements for most interactive applications.

Acknowledgments

Thanks to the DEC Systems Research Center for support as summer interns in 1990, at which time this idea was originally conceived, to David Pierce and Nicholas Haines for their contributions to the original implementation, and to John Reppy for his suggestion to merge the forwarding pointer and header word. Plenty of thanks goes to the facilities staff at CMU, in particular Alessandro Forin and Dale Moore, for providing timely assistance adjusting the clock resolution. Thanks also to our readers: Andrew Myers, Mark Day, Greg Morrisett, Nicholas Haines, Richard Lethin, John Keen, Amer Diwan, David Tarditi, Brian Milnes, Ali-Reza Adl-Tabatabai and David Gifford.

References

[1] 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.

[2] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280–294, 1978.

[3] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly Parallel Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, 1991.

[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*, 1984.

[5] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[6] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 43–52, June 1992.

[7] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly Garbage Collection:An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[8] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ml. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 113–123, January 1993.

[9] John R. Ellis, Kai Li, and Andrew W. Appel. Real-time Concurrent Garbage Collection on Stock Multiprocessors. Technical Report DEC-SRC-TR-25, DEC Systems Research Center, February 1988.

[10] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. A comparative performance evaluation of write barrier implementations. In *Proceedings of the 1992 Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992.

[11] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the 1993 ACM Symposiym on Principles and Practice of Parallel Programming*, 1993. To appear.

[12] 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.

[13] Scott M. Nettles, James W. O'Toole, and David K. Gifford. Concurrent garbage collection of persistent heaps. Technical Report MIT-LCS-TR-569 and CMU-CS-93-137, Massachusetts Institute of Technology and Carnegie Mellon University, 1993. Submitted to 14th Symposium on Operating Systems Principles.

[14] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-Based Incremental Copying Collection. In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992. Also available as Carnegie Mellon University Technical Report CMU-CS-93-135.

[15] K. Nilsen. Garbage Collection of Strings and Linked Data Structures in Real-time. *Software–Practice and Experience*, 18(7):613–640, July 1988.

[16] 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.

[17] James W. O'Toole. Garbage Collecting an Object Cache. Technical Report MIT/LCS/TM-485, Massachusetts Institute of Technology, April 1993. To appear.

[18] James W. O'Toole and Scott M. Nettles. Concurrent Replication Garbage Collection. Technical Report MIT-LCS-TR-570 and CMU-CS-93-138, Massachusetts Institute of Technology and Carnegie Mellon University, 1993.

[19] James W. O'Toole and Scott M. Nettles. Real-Time Replication GC: An Implementation Report. Technical Report MIT-LCS-TR-568 and CMU-CS-93-136, Massachusetts Institute of Technology and Carnegie Mellon University, 1993.

[20] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, pages 1–42. ACM, Springer-Verlag, September 1992.