Concurrent Replication Garbage Collection: An Implementation Report

James O'Toole[†] and Scott Nettles

April 1993

CMU-CS-93-138

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Submitted to POPL'94.
Also appears as MIT-LCS-TR-570.

This research was sponsored by the Avionics Laboratory, 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, by the Defense Advanced Research Projects Agency (DARPA) under Contract F19628-91-C-0168, and by the Department of the Army under Contract DABT63-92-C-0012.

Keywords: storage management, garbage collection, concurrent garbage collection, incremental garbage collection, copying garbage collection, replication garbage collection

Abstract

We have implemented a concurrent copying garbage collector that uses replication garbage collection. In our design, the client can continuously access the heap during garbage collection. No low-level synchronization between the client and the garbage collector is required on individual object operations. The garbage collector replicates live heap objects and periodically synchronizes with the client to obtain the client's current root set and mutation log. An experimental implementation using the Standard ML of New Jersey system on a shared-memory multiprocessor demonstrates excellent pause time performance and moderate execution time speedups.

1 Introduction

As programs have become larger and more complex the use of dynamic storage allocation has increased. Increased use of object oriented and functional programming techniques further exacerbates this trend. These same trends also make automatic management of dynamic storage or garbage collection (GC) increasingly necessary. GC both simplifies the programmers task and more importantly increases the robustness and safety of programs that use it.

The traditional objections to GC are primarily performance related. It has often been considered too expensive for use in practical applications. Recent studies by Zorn[16] of applications which make heavy use of dynamic storage suggest that in fact explicit storage management may be as costly as GC. However many garbage collectors stop the application during collection, creating pauses which are unacceptable to many applications which might otherwise utilize GC.

Incremental collection addresses the problem of pause times by allowing the collector and application to proceed in tandem. We have recently demonstrated that *replication garbage collection* can be used to build incremental collectors which limit these pauses sufficiently to allow applications such as mouse tracking to use GC. In this work we show how the same technique can be used to build collectors which are concurrent. Because most of the collection work can be done concurrently we are able to demonstrate both much shorter pauses and speedups compared to our previous work.

In the sections that follow, we first introduce the basic idea of replication garbage collection. Then we discuss our implementation which takes the form of modifications to the Standard ML of New Jersey (SML/NJ) collector. We then discuss the details of our implementation and present measurements which show that pause times are mostly eliminated and elapsed execution times are reduced. 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. For those unfamiliar with the area an excellent survey may be found in Wilson[15].

2 Concurrent Replication GC

Concurrent garbage collectors permit the client to execute while the garbage collection is in progress. The operations of the client and the collector may be interleaved in any order, yet the effects of the garbage collector must not be observable to the client. In many previous concurrent garbage collection designs, the interactions between the client and the collector may lead to complex and expensive synchronization requirements. Replication garbage collection requires that the collector replicate live objects without modifying the original objects. Interactions with the client are minimized, making this design attractive for use in a concurrent collector.

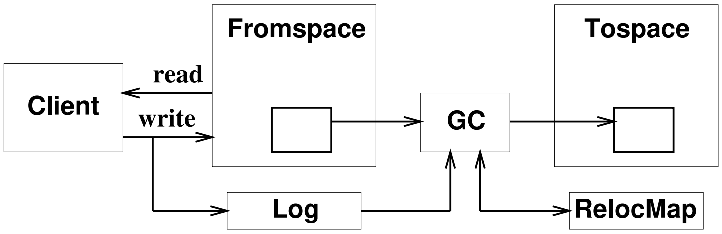

Figure 1: Replication and The Mutation Log

2.1 The Client Uses From-Space

The standard technique used by copying garbage collectors to copy an object destroys the original object by overwriting it with a forwarding pointer. Therefore, concurrent collectors using this technique must ensure that the client uses only the relocated copy of an object. This requirement is referred to as the to-space invariant.

The primary way in which replication copying differs from the standard approach is that the copy is performed non-destructively. Conceptually, whenever the collector replicates an object it stores a relocation record in a relocation map, as shown in Figure 1. In general the client may access the original object or the relocated objects and is oblivious to the existence or contents of the relocation map. In the implementation we consider here the client accesses only the original object. We refer to this as the from-space invariant.

2.2 Mutations are Logged

After the collector has replicated an object, the original object may be modified by the client. In this case, the same modification must also be made to the replica before the client can safely use the replica. Therefore, our algorithm requires the client to record all mutations in a "mutation log", as shown in Figure 1. The collector uses the log to ensure that all replicas are in a consistent state when the collection terminates. The collector does this by reading the log entries and applying the mutations to the replicas.

The cost of logging and of processing the mutation log varies depending on the application and the logging technique. Mutation logging works best when mutations are infrequent or can be recorded without client cooperation. Mutation logging is also attractive whenever a log is already required for other reasons, such as in generational collectors, distributed applications, and transactional storage systems[14, 12].

2.3 The Collector Invariant

The invariant maintained by the collector is that the client can only access from-space objects and that all to-space replicas are up-to-date with respect to their original from-space objects, unless a corresponding mutation is described in the mutation log.

2.4 The Completion Condition

While the collector executes, it endeavors to replicate of all objects accessible to the client in from-space in to-space. The collector creates replicas of the objects pointed to by the client's roots, and continues scanning objects in to-space to find pointers to objects in from-space and replace them with pointers to corresponding replicas in to-space.

The collector has completed a collection when the mutation log is empty, the client 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 halts the client, atomically verifies the completion condition, updates the client's roots to point at the corresponding to-space replicas, discards the from-space, and renames to-space as from-space.

2.5 Client Interactions

Although the garbage collector executes concurrently with the client, the from-space invariant ensures that there is no low-level interaction between the collector and client. The client executes machine instructions which read and write the objects which reside in from-space. The collector reads the objects in from-space and writes the objects in to-space. Conceptually, the relocation map shown in Figure 1 is used only by the collector.

The collector does interact with the client via the mutation log and the client's roots. The collector must occasionally obtain an up-to-date copy of the client's roots in order to continue building the to-space replica. Also, the collector reads the mutation log being written by the client. These interactions may be asynchronous and do not require the client to be halted.

However, when the collector has established the completion condition, it must halt the client in order to atomically verify the completion condition and update the client's roots. After the roots have been updated, the client can resume execution. The duration of this pause in the client's execution depends on the synchronization delay due to interacting with the client thread and also on the size of the root set. In a generational collector the root set may include the set of cross-generational pointers which point from older objects to newer objects.

3 Implementation

Our concurrent replication copying collector is based on a version of Standard ML of New Jersey (SML/NJ) which has been extended to support multiprocessors. The collector uses a separate gc thread to perform scanning and replication work, but the current prototype processes mutation log entries only while the clients are paused. The concurrent collector can be enabled for one or both of the two generations present in the original SML/NJ collector.

3.1 The SML/NJ Runtime System

SML/NJ (version 0.75) has a good compiler and a simple generational garbage collector. The runtime system has no stack and therefore places heavy demands on the memory management system. Providing efficient garbage collection in this environment is challenging because of SML/NJ's high allocation rates. However, the SML language encourages a mostly functional programming style, so mutations are rare.

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. Old-space is divided into From-space and To-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 from-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 from-space. The spaces and associated parameters are shown in Figure 2.

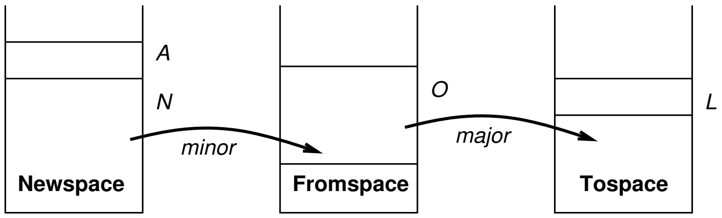

Figure 2: SML/NJ Heaps with GC Parameters

3.2 Logging and Replication

Generational collectors must identify mutations that might create pointers from older spaces into younger spaces. The SML/NJ collector uses a log called the "storelist" to track such mutations. In our previous work on replication garbage collection, we modified the SML/NJ compiler and all appropriate runtime system operations so that all mutations are recorded in the storelist.

The easiest way to implement the relocation map is to store a forwarding pointer in an extra word in each replicated object. However, most objects in the SML/NJ runtime system are only three words long, so the forwarding words would be relatively costly in space. Therefore in our implementation we overwrite the object header word with the forwarding pointer.

As shown in Figure 3, the client operation which reads the header word was modified to follow the forwarding word. Our previous results showed that the runtime cost to the client due to this change was not significant[10]. However, in the presence of concurrency, this change creates a potential read-write conflict between the collector and the client. If the client is reading the header word at the same time the collector is installing a forwarding pointer, we must make sure that the client gets the correct header word.

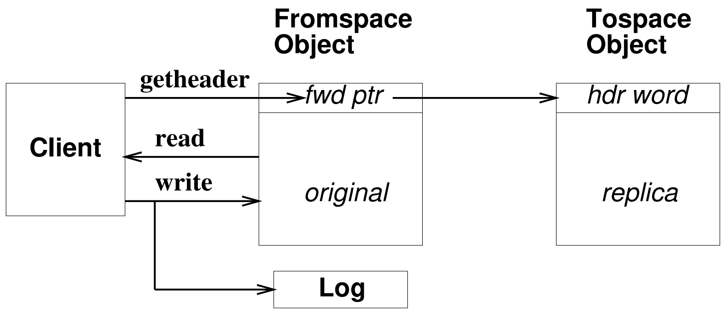

Figure 3: Getheader Operations Follow the Forwarding Word

The code sequences used by the client and the collector for these operations were designed to avoid any possible race condition. The client reads the from-space header word only once and then dereferences the value obtained if it is a forwarding pointer. The collector replaces the from-space header word with the forwarding pointer only after storing the correct header word in the to-space replica. method works provided that the memory system performs single-word write operations atomically and that several write operations issued from a single processor are performed in the order issued.

3.3 Controlling Client Allocation

The SML/NJ system uses an allocation limit to control the amount of memory which can be allocated by the client. The allocation limit is initially set to the size of newspace (N). Whenever the client is about to exceed its current allocation limit, it traps into the garbage collection module and triggers a garbage collection. In our current implementation, when the garbage collection is first triggered the client processes its mutation log, awakens the gc thread, and immediately resumes execution.

In order to continue execution, the client's allocation limit must be increased. Another parameter (A) is used to control the minimum amount additional allocation which will be granted to the client (see figure 2). Our implementation provides the client with $(last\_amount + A)/2$ amount of additional memory each time the client exhausts its current allocation limit, where *last_amount* is the size of the previous amount of memory given to the client. This formula was chosen so that if the client is repeatedly consuming its allocation limit the allocation increment will decay to A, the minimum acceptable amount.

In the worst case, the client threads may be allocating more live data per unit time than the gc thread can copy. In such a case, the client must be slowed down enough that the gc thread can keep up. Currently we do this by pausing the client thread to perform replication work if it exceeds its allocation increment and a garbage collection is already active.

In order to control the pause time, the algorithm restricts the amount of work it does using a parameter L. The L parameter limits the total amount of memory copied by the collector and directly controls the pause times. This policy essentially makes the collector incremental rather than concurrent when the client is allocating very aggressively.

Controlling GC Activity

The gc thread also interrupts the client asynchronously whenever it has established the completion condition. The gc thread does this using a Unix signal mechanism taken from the SML/NJ MP system by Morrisett and Tolmach[9]. The signal causes the client to enter the garbage collection module and attempt to complete the collection. The client processes the mutation log and scans to-space to perform a limited amount of replication work.

If the completion condition can be established without exceeding the work limit L, then the client's roots are updated and the from-space and to-space exchange roles. Otherwise, the client's roots are copied into a shadow root set used by the concurrent gc thread, and the clients resume execution using from-space while the gc thread continues to perform collection work.

The Unix signals used by the current implementation seem to be a very expensive way to get the client into the garbage collection module. In particular, in version 0.75 the client always transfers control to the garbage collector using an arithmetic trap which causes a Unix signal. SML/NJ version 0.93 uses a goto for this purpose, and this may decrease the cost of client/collector synchronization, but our implementation may still need to asynchronously signal the client. We are investigating other solutions to this problem.

Our implementation uses a fixed value for the parameter L, but we would like to investigate whether the amount of inline garbage collection work done by the client should start small and slowly increase to L, depending on the number of times the client has trapped into the collector.

4 Performance

The goals of the performance study were to demonstrate that pause times are significantly shorter than those for the incremental version of the algorithm and to measure the speedup provided by the use of a concurrent thread for garbage collection work. The measured performance is good; the concurrent collector achieves pause times in the neighborhood of 5 milliseconds and eliminates most of the garbage collection work from the elapsed execution time.

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 many major and minor garbage collections during execution. See [5] 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 Silicon Graphics 4D/340 equipped with 192 megabytes of physical memory. The clock resolution on this system is approximately 1 millisecond. The machine contains four MIPS R3000 processors clocked at 33 megahertz. Each processor has a 64 kilobyte instruction cache, a 64 kilobyte primary data cache, and a 256 kilobyte secondary data cache. The secondary data caches are kept consistent via a shared memory bus watching protocol and there is a five word deep store buffer between the primary and the secondary caches. Because of the store buffers, processors can observe out-of-date values. The average copying rate achieved by the garbage collector while running the benchmarks on this hardware platform was between 1 and 2 megabytes per second.

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 2 megabytes and 100 kilobytes. 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 1 megabyte and 500 kilobytes. Again, the larger value is typical for use with the stop-and-copy collector, while the lower value showed good performance with our system. Unlike in our previous work on incremental collection, small values of N are not important for providing short pause times.

In all cases we set L to 3 kilobytes. The L parameter determines how long the client might remain in the garbage collector. Choosing a low setting allows us to achieve maximum speedups and short pauses. This results also contrast to our incremental collector, where short pause time conflicted with good elapsed time performance. Empirically, the 3 kilobyte limit appears to be a good compromise between greater overhead and larger pause times. We arbitrarily chose A to be 10 kilobytes in all cases. Our studies showed that performance was not strongly coupled to the choice of A in our current implementation.

Unfortunately, trying to compare SML/NJ's stop-and-copy collector to our concurrent collector is difficult. Ideally we would like each collector to do the same amount of work and for this amount of work to be repeatable. Unfortunately concurrency introduces a degree of nondeterminism which makes such repeatability almost impossible to achieve.

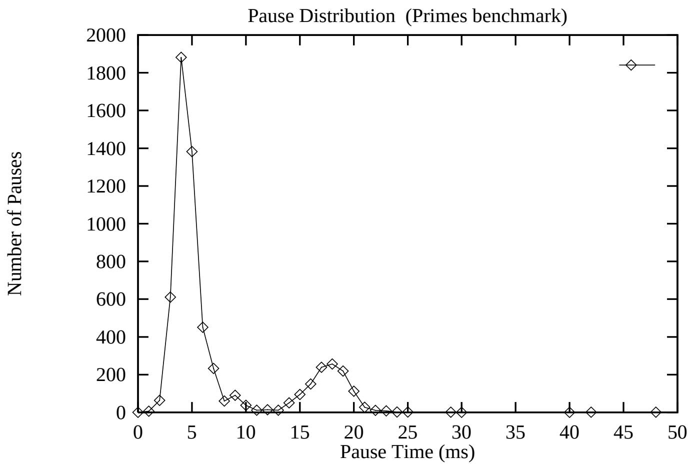

Figure 4: Primes Benchmark Pause Times

4.3 Pause Times

One motivation for using a concurrent garbage collector is to eliminate the pause times normally experienced by the client while the garbage collector executes. In this section we report on the pause times achieved by our collector.

Figures 4, 5, and 6 show plots of pause times for each of the benchmarks. The plots shown are for the setting which achieved the best absolute performance. In general we see that the pauses are very short, around 5 milliseconds. The pause times are generally an order of magnitude shorter than the delays caused by virtual memory when page faults access the disk.

We are concerned about the long tail of longer pause times which appear in these results, although they make up only a tiny fraction of the pauses. We don't yet have a good explanation for them. Another interesting anomaly is the second peak of longer pause times occurring in the primes benchmark (see Figure 4). This is due to processing the mutation log. The implementation does not account for log processing under the work limit parameter L. We hope to fix this.

The measurements show that the concurrent collector is largely successful at eliminating the pauses. Its pauses are minuscule in comparison to those produced by the stop-and-copy collector.

4.4 Elapsed Times

The other primary motivation for using a concurrent garbage collector is to reduce the elapsed time of the client program by allowing the collection work to be performed concurrently. Because garbage collection time is a relatively small component of total execution time such speedups are difficult to achieve and hard to measure.

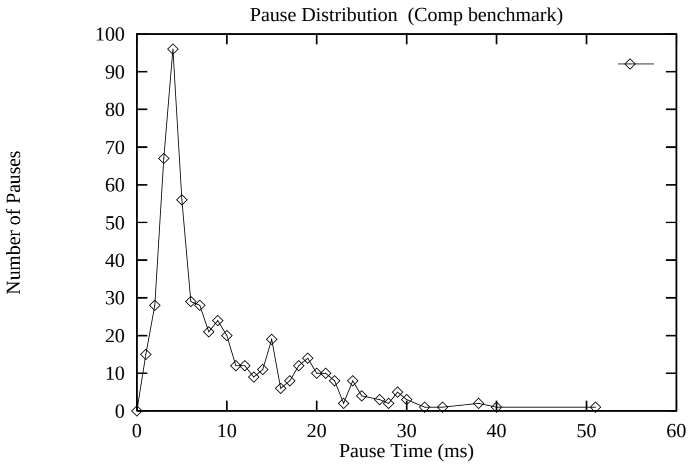

Figure 5: Compiler Benchmark Pause Times

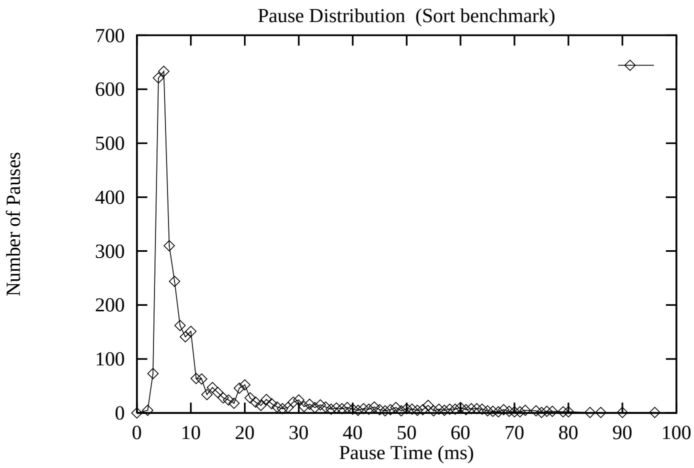

Figure 6: Sort Benchmark Pause Times

O Kb	N Kb	Stop+Copy Elapsed	Stop+Copy Major	Stop+Copy Minor	Concurrent Elapsed	Concurrent Speedup	Concurrent Major	Concurrent Minor	Major-Only Elapsed	Major-Only Speedup	Major-Only Major	Major-Only Minor
1000	100	119.36	10.3%	15.7%	133.46	111.8%	4.5%	31.4%	112.84	94.5%	4.3%	24.5%
1000	2000	108.71	1.2%	17.7%	132.33	121.7%	4.4%	33.9%	109.33	100.6%	1.6%	19.4%
500	100	132.68	16.8%	17.9%	135.09	101.8%	4.8%	29.7%	116.79	88.0%	4.8%	24.8%
500	2000	112.20	2.2%	22.2%	132.19	117.8%	5.8%	35.0%	114.16	101.7%	2.1%	22.6%

Table 1: Elasped Times (Primes)

O Kb	N Kb	Stop+Copy Elapsed	Stop+Copy Major	Stop+Copy Minor	Concurrent Elapsed	Concurrent Speedup	Concurrent Major	Concurrent Minor	Major-Only Elapsed	Major-Only Speedup	Major-Only Major	Major-Only Minor
1000	100	87.86	34.8%	10.9%	60.92	69.3%	3.2%	12.9%	60.45	68.8%	2.1%	16.1%
1000	2000	58.96	4.0%	16.8%	57.82	98.1%	3.4%	16.1%	57.11	96.9%	1.3%	20.1%
500	100	102.42	43.7%	11.9%	64.91	63.4%	4.0%	12.6%	63.76	62.3%	2.7%	15.6%
500	2000	60.21	4.9%	20.1%	60.16	99.9%	4.8%	18.1%	58.99	98.0%	2.2%	22.8%

Table 2: Elasped Times (Comp)

O Kb	N Kb	Stop+Copy Elapsed	Stop+Copy Major	Stop+Copy Minor	Concurrent Elapsed	Concurrent Speedup	Concurrent Major	Concurrent Minor	Major-Only Elapsed	Major-Only Speedup	Major-Only Major	Major-Only Minor
1000	100	80.60	26.3%	30.1%	72.24	89.6%	8.3%	38.3%	81.73	101.4%	1.2%	56.5%
1000	2000	61.90	3.0%	39.4%	73.37	118.5%	11.0%	50.0%	81.78	132.1%	1.7%	78.4%
500	100	94.50	33.1%	31.0%	79.57	84.2%	12.7%	37.2%	82.77	87.6%	2.3%	48.9%
500	2000	65.44	4.5%	45.2%	79.17	121.0%	18.1%	53.2%	82.55	126.1%	3.4%	70.5%

Table 3: Elasped Times (Sort)

Tables 1, 2, and 3 contain the elapsed time performance results for the three benchmarks. The columns shown in the tables are the total elapsed time using all three collector configurations, the speedup of the concurrent configurations relative to the stop-and-copy collector, and the percentage of time spent doing major and minor collection work. All percentages are shown as percentage of the original elapsed time using stop-and-copy. Each row of the table corresponds to a different choice of parameter values which control how much allocation takes place before collections are triggered.

In general we are successful at reducing the time spent in major collections and the cases in which we achieve reasonable speed ups are those in which the major collection overhead is high. Unfortunately we have so far been unsuccessfully at improving minor collection times. In fact we have been more successful at increasing them. We believe this is in part due to the high costs of synchronization with the mutator as well as the fact that minor collections are brief but frequent. We believe that further experience with the collector will allow us to improve this aspect of our performance.

4.5 Future Benchmarking Plans

All of the benchmarks we have measured so far use single-threaded client programs, but our implementation does support multithreaded clients. We hope to obtain measurements of multithreaded client programs soon. We have heard from Tolmach that the speedups achieved on the benchmarks in his work with Morrisett[9] may have been limited because the garbage collector was stop-and-copy and single-threaded. It is possible that those speedups would be closer to linear using a concurrent collector.

We are also interested in feedback concerning performance questions about the collector which aren't answered by this paper. We expect to be able to measure the trapping and GC synchronization costs in the current implementation. These and other measurements might answer the policy questions raised in Section 3.3.

5 Related Work

There is a long history of incremental and concurrent copying collectors dating back to Baker[2]. Essentially all of these collectors require the client to access the to-space version of an object during collections. The technique of Ellis, Li, and Appel[1] enforces this restriction by using virtual memory protection to force clients to use only to-space values. Our technique does not require any unusual operating system or hardware support and it imposes smaller demands on the client than software versions of Bakers algorithm. To-space methods also constrain the order in which objects are copied. We believe that the ability to freely choose the order in which objects are copied and traversed is especially important in a system which may need to optimize access to the disk.

The idea of a separate forwarding pointer word first appeared in the context of to-space methods. Brooks' technique[4], later implemented by North and Reppy[13], requires the client to follow a forwarding pointer which leads to the relocated object. This eliminated a test in favor of extra space and an indirection.

Work by Boehm, Demers and Shenker [3] on a concurrent mark-and-sweep collector uses mutation logging to track changes made by the client. The mutation log is implemented by periodically sampling the dirty page bits maintained by the virtual memory system. 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 and employ variations of the replication idea. Both depend heavily on the fact that ML implementations can distinguish mutable from immutable data. Our technique does not and is thus more generally applicable.

Doligez and Leroy[7] 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[6]. To avoid the issue of inconsistent mutable values, all mutable 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.

The critical difference between their approach and ours is of course that they do not use replication copying to implement the concurrent collector. They also avoid the issue of mutable object consistency by not replicating mutable objects. Their approach has several disadvantages when compared to ours. First, the need to allocate mutable objects in the shared heap makes such allocation expensive. Second, the need to copy values assigned to mutable value may lead to unnecessary overhead. If the same location is overwritten before the next collection then extra copying will be done. Finally, the use of a stop-and-copy collector for minor collections means such collections are bounded in duration only by the size of the new area. They deal with this problem by limiting the new size to 32Kb. This is acceptable for their byte-code interpreter, but would not be for SML/NJ. Their technique has one important advantage over ours. In their collector each thread can perform its minor collection independently of every other thread and in general no global synchronization is needed between the clients and the collector. We believe this to be an important advantage and are attempting to understand how to achieve it in our system.

Huelsbergen and Larus[8] describes a concurrent collector for SML/NJ which uses replication copying. Their collector uses an invariant which requires the client to use the to-space version of a mutable object if it exists. Because the client sometimes uses to-space objects, all operations on mutable objects must suffer some additional overhead due to synchronization with the collector. As a result, their implementation is more closely tied to the semantics of mutable values in SML and to the details of their processor memory consistency model.

In addition, their collector is not generational, which makes it less efficient that the original SML/NJ collector despite the use of multiple processors. This also makes it difficult to assess the overhead of their technique. Less importantly, their implementation does not merge forwarding pointers with header words and thus has a substantial space penalty. We hope to implement their invariant along with some of the others we described in [11] and obtain a direct comparison.

The present work extends our previous work on incremental and real-time collection[11, 10] by supporting concurrency among multiple clients and the garbage collector and exploring the policy issues which are central to providing hard real time bounds, We also use this concurrent collector in conjunction with a transaction manager for a persistent heap, where the mutation log does double-duty as the transaction log[14].

6 Future Work and Conclusions

We plan to do additional performance measurements and test various control policies for the concurrent collector (see section 4.5). Another area which requires further study is how to most opportunistically schedule the work of the concurrent gc thread so as to minimize its impact on overall client performance. In an interactive or disk-bound system, collection work could be scheduled to coincide with i/o activity. The resources consumed by the garbage collection thread in a multiprocessor system are not free. Understanding the collector's impact on overall system performance is a natural area for future work.

Conclusions

We have implemented a simple concurrent garbage collector using replication garbage collection. The from-space invariant permits the collector and the client to operate concurrently without imposing low-level synchronization delays on individual heap operations. The client communicates to the collector via a mutation log. We have examined various synchronization costs in an implementation that relies on client cooperation for logging. Our prototype implementation shows moderate speedups and excellent pause time performance for applications with bounded allocation rates.

Acknowledgements

Thanks to the DEC Systems Research Center for support as summer interns in 1990, at which time this idea was originally conceived.

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] 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.

[6] 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.

[7] 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.

[8] 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.

[9] J. Gregory Morrisett and Andrew Tolmach. Procs and locks: A portable multiprocessing platform for standard ml of new jersey. In *Proceedings of the 1993 ACM Symposiym on Principles and Practice of Parallel Programming*, pages 198–207, 1993.

[10] Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*. ACM, June 1993.

[11] 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.

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

[13] 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.

[14] James W. O'Toole, Scott M. Nettles, and David K. Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, SIGPLAN, December 1993. To appear.

[15] 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.

[16] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.