Concurrent Replicating Garbage Collection

James O'Toole and Scott Nettles

Abstract

We have implemented a concurrent copying garbage collector that uses replicating 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 simplifies the programmers task and 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 that 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 that are unacceptable to many applications that might otherwise utilize GC.

Incremental collection addresses the problem of pause times by allowing the collector and application to proceed in tandem. We have previously demonstrated that *replicating garbage collection* can be used to build incremental collectors that limit these pauses sufficiently to allow applications such as mouse tracking to use GC [10]. In this work we show how the same technique can be used to build collectors that 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 next section we introduce the basic idea of replicating garbage collection. Then we describe our implementation (Section 3) and present measurements of its performance (Section 4). The results show that pause times are mostly eliminated and that 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. The survey by Wilson [15] should be useful to readers unfamiliar with the area.

2 Concurrent Replicating 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 by 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. Replicating 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.

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

This research was sponsored by the Wright Research and Development Center, Aeronautical Systems Division under Contract F33615-90-C-1465, Arpa Order No. 7597, by the Air Force Systems Command and the Advanced Research Projects Agency (ARPA) under Contract F19628-91-C-0128, and by the Department of the Army under Contract DABT63-92-C-0012.

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

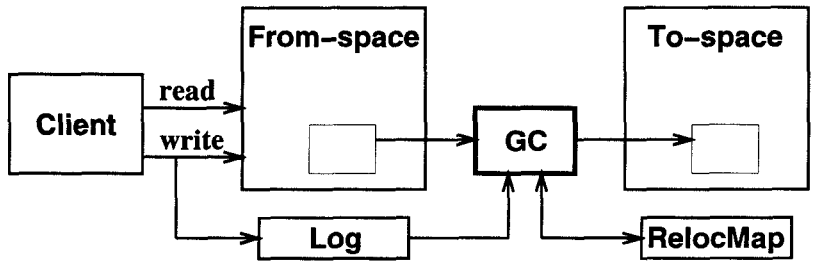


Figure 1: Replication and The Mutation Log

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 replicating collection differs from the standard approach is that the copying of objects 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 describe here the client accesses only the original object. We call this 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 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 [12, 14].

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 all the objects that are accessible to the client. The collector creates replicas of the objects pointed to by the client's roots. The collector also scans replicas in to-space to find pointers to from-space objects 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 and are up-to-date. The replicas contain only to-space pointers because to-space has been scanned. 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 that read and write the objects that 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, which is 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 that point from older objects to newer objects.

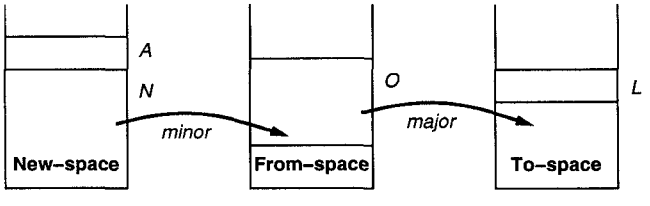
3 Implementation

Our concurrent replicating collector is based on a version of Standard ML of New Jersey (SML/NJ) that has been extended to support multiprocessors [9]. The collector uses a separate thread for garbage collection work. Scanning and replication work are done concurrently, but the current prototype processes mutation log entries only while the client is 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. The size of the new-space is controlled by the runtime parameter N. When new-space fills, a minor collection is initiated to copy the live data into old-space. Old-space is divided into from-space and to-space. 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.



O	Memory copied into From-space before major gc.
N	Initial allocation limit in New-space before minor gc.
A	Minimum additional allocation permitted per gc pause.
L	Maximum amount of data copied while client paused.

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. We modified the SML/NJ compiler and all appropriate runtime system operations so that all mutations are recorded in the storelist. In previous work [10], we measured the runtime cost of the additional logging to be 0–5% of total execution time for the benchmarks described in this paper. No logging is required for allocation operations because newly allocated objects cannot yet have been copied by the garbage collector.

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 the collector overwrites the object header word with the forwarding pointer.

As shown in Figure 3, the client operation that 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.

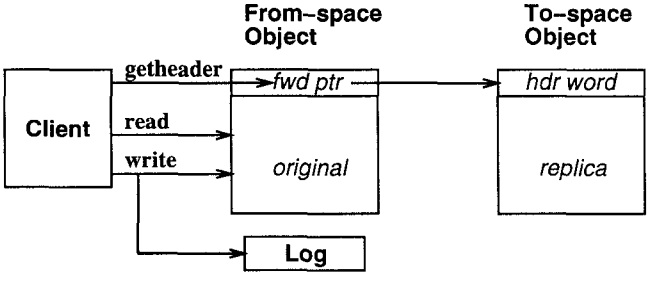


Figure 3: Getheader Operations Follow the Forwarding Word

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. This 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 that can be allocated by the client. The allocation limit is initially set to the size of new-space (N). Whenever the client is about to exceed its current allocation limit it traps into the garbage collection module. In our current implementation, the client then synchronizes with the collector. The garbage collector processes the mutation log, initiates a concurrent garbage collection, and then permits the client to continue execution.

However, to continue executing the client must allocate more memory. Yet, the client must also be prevented from allocating live data more rapidly than the collector can copy it, or else the collection will never be completed. Therefore, our implementation sometimes slows down the client by not permitting it to continue execution immediately. If the client exceeds its allocation limit while a garbage collection is already in progress, then the client is paused until the collector has performed some minimum amount of replication work (see Section 3.4).

The collector calculates how much additional memory allocation will be granted to the client using the parameter A (see Figure 2). When the client exhausts its allocation limit, the limit is advanced by $(last\_amount + A)/2$ units of additional memory, where $last\_amount$ is the previous amount of memory given to the client. Using this formula the allocation increment eventually decays to A. Thus, the A parameter specifies the minimum amount of memory the client must be permitted to allocate each time it is delayed by interacting with the garbage collector.

3.4 Controlling GC Activity

When the client is allocating very aggressively, it will be paused by the garbage collector when it exceeds its allocation limit. The collector also pauses the client when it has established the completion condition, in order to attempt a flip. We believe the collector should pause the client and attempt to flip immediately upon establishing the completion condition, because otherwise the client will allocate more memory. To control how long the client will be halted during these pauses we use a parameter L. The L parameter limits the amount of memory the collector will copy while the client is paused.

When the collector has paused the client to attempt a flip, it processes the mutation log and performs some replication work. If the completion condition can be established without exceeding the 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 collector and the client is permitted to resume execution while the collector continues to perform replication work.

3.5 Client/Collector Synchronization

In version 0.75 of SML/NJ, the client always transfers control to the garbage collector using an arithmetic trap that causes a Unix signal. Also, our collector uses a Unix signal mechanism taken from the SML/NJ MP system by Morrisett and Tolmach [9] to interrupt the client when it has established the completion condition. Our implementation also requires the client to asynchronously pause the collector. This is now implemented by having the collector poll periodically to detect a synchronization request from the client.

Unix signals are an expensive way for the client to synchronize with the garbage collector. Version 0.93 of SML/NJ replaces the client-initiated trap with a goto. However, even with this improvement our implementation will need to asynchronously halt the client in a known garbage collectible state. This is because in general there may be more than one client thread, all of which must synchronize to perform the flip. We are investigating better solutions to this problem.

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 another processor for concurrent 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.

- *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 10 megabytes per second), 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 compiler is a large optimizing compiler 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 compilation phase.
- *Sort* is a sorting program based on futures that are implemented using SML threads. *Sort* does more mutation than typical SML programs and creates a large amount of live data. The high mutation and survival rates make this a challenging gc benchmark.

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 primary data caches are write-through caches and there is a five word deep write buffer between each primary cache and its associated secondary cache. The secondary data caches are write-back caches and are kept consistent via a shared memory bus watching protocol. 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 2000 kilobytes 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 1000 kilobytes 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 result contrasts with 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 that makes such repeatability almost impossible to achieve.

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 that achieved the best absolute performance. In general we see

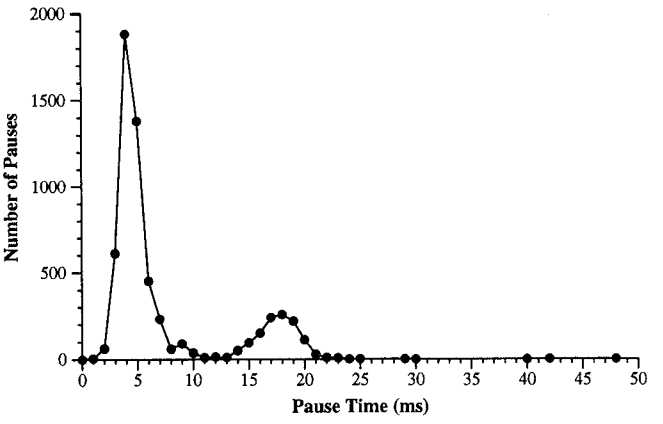


Figure 4: Primes Benchmark Pause Times

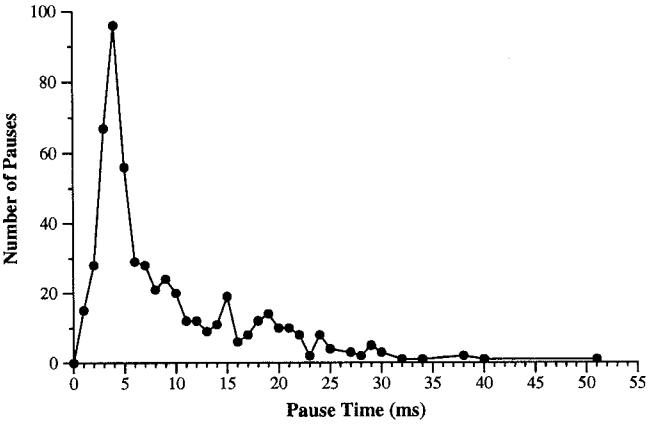


Figure 5: Compiler Benchmark Pause Times

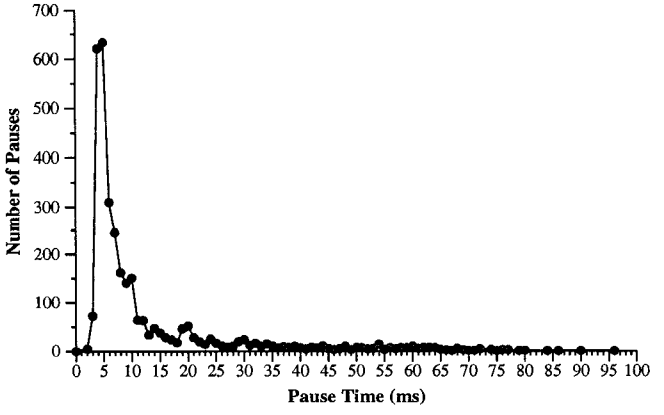


Figure 6: Sort Benchmark Pause Times

N	O	Stop+Copy			Major-Only-Concurrent				Concurrent			
Kb	Kb	Elapsed	Major	Minor	Elapsed	Speedup	Major	Minor	Elapsed	Speedup	Major	Minor
1000	100	119.36	10.3%	15.7%	112.84	5.5%	4.3%	24.5%	133.46	-11.8%	4.5%	31.4%
1000	2000	108.71	1.2%	17.7%	109.33	-0.6%	1.6%	19.4%	132.33	-21.7%	4.4%	33.9%
500	100	132.68	16.8%	17.9%	116.79	12.0%	4.8%	24.8%	135.09	-1.8%	4.8%	29.7%
500	2000	112.20	2.2%	22.2%	114.16	-1.7%	2.1%	22.6%	132.19	-17.8%	5.8%	35.0%

Table 1: Primes Benchmark Elasped Times

N	O	Stop+Copy			Major-Only-Concurrent				Concurrent			
Kb	Kb	Elapsed	Major	Minor	Elapsed	Speedup	Major	Minor	Elapsed	Speedup	Major	Minor
1000	100	87.86	34.8%	10.9%	60.45	31.2%	2.1%	16.1%	60.92	30.7%	3.2%	12.9%
1000	2000	58.96	4.0%	16.8%	57.11	3.1%	1.3%	20.1%	57.82	1.9%	3.4%	16.1%
500	100	102.42	43.7%	11.9%	63.76	37.7%	2.7%	15.6%	64.91	36.6%	4.0%	12.6%
500	2000	60.21	4.9%	20.1%	58.99	2.0%	2.2%	22.8%	60.16	0.1%	4.8%	18.1%

Table 2: Compiler Benchmark Elasped Times

N	O	Stop+Copy			Major-Only-Concurrent				Concurrent			
Kb	Kb	Elapsed	Major	Minor	Elapsed	Speedup	Major	Minor	Elapsed	Speedup	Major	Minor
1000	100	80.60	26.3%	30.1%	81.73	-1.4%	1.2%	56.5%	72.24	10.4%	8.3%	38.3%
1000	2000	61.90	3.0%	39.4%	81.78	-32.1%	1.7%	78.4%	73.37	-18.5%	11.0%	50.0%
500	100	94.50	33.1%	31.0%	82.77	12.4%	2.3%	48.9%	79.57	15.8%	12.7%	37.2%
500	2000	65.44	4.5%	45.2%	82.55	-26.1%	3.4%	70.5%	79.17	-21.0%	18.1%	53.2%

Table 3: Sort Benchmark Elasped Times

that the pauses are very short, around 5 milliseconds. The pause times are generally an order of magnitude shorter than the delays due to virtual memory when page faults must access the disk.

We are concerned about the long tail of longer pause times that appear in these results, although they make up only a tiny fraction of the pauses. Although we don't yet have good explanations for the longer pauses, our traces show that most of those pauses have much larger elapsed wallclock time (shown here) than user cpu time. This indicates that uncontrolled operating system effects such as processor scheduling or page faults are definitely contributing to the longer pauses. 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 process the log concurrently or account for log processing under the work limit parameter L. We hope to fix this in our next implementation.

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, which are often one second or more.

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 can be a small component of total execution time, and concurrent collection introduces many short client/collector interactions, such speedups are difficult to achieve and hard to measure.

Tables 1, 2, and 3 contain the elapsed time performance results for the three benchmarks. Each table contains a section for the stop-and-copy collector, our new collector performing only major collections concurrently, and our concurrent collector in use for both minor and major collections. The columns shown are the total elapsed time, speedup relative to the stop-and-copy collector, and percentage of time spent doing major and minor collection work.

All percentages are given relative to the original elapsed time using stop-and-copy. The speedup provides the reduction in elapsed time as a percentage of the original elapsed time, so a negative speedup indicates a slowdown. Each row corresponds to a different choice of the parameter values controlling how much allocation takes place before collections are initiated.

The improvement in total elapsed time achieved by the concurrent collector ranges from a speedup of 47% to a slowdown of 21%. Comparing the concurrent collector's speedup to the major-only-concurrent speedup reveals that the performance improvements are mostly due to concurrent major collections. From the center sections of the table, we note that when major collections are performed concurrently, the time attributed to minor collections rises. This is party due to the cost of log processing, which is charged entirely to the minor collector because the minor and major log processing code is intermingled. Concurrent major collections also may increase minor collection costs because interrupting the client to complete a major collection causes a minor garbage collection.

However, we still observe that the total time devoted to minor and major collections is generally lower for the major-only-concurrent collector than for the stop-and-copy collector. Both concurrent collectors achieve better speedup results when the O parameter setting is small. When O is small the major collections occur much more often and therefore consume a larger fraction of the stop-and-copy execution time.

Although using the concurrent collector for major collections produced good speedups, it appears that concurrent minor collections were less successful. In fact, comparing the major-only-concurrent and the concurrent collector sections of the tables reveals that the fully concurrent configuration often performed worse. Only the *Sort* benchmark performed better when minor collections were concurrent, because only the *Sort* benchmark spends a large percentage of elapsed time on minor collections in the stop-and-copy configuration. Even in this benchmark, the fully concurrent configuration shows more elapsed time in the minor collector than the stop-and-copy configuration does.

We believe that the increased fraction of elapsed time in the minor collector is caused by high synchronization costs and the log processing work. Earlier measurements [10] of these benchmarks indicate that the log processing costs are small, but they do explain some of the increase in minor gc time observed here. More significantly, the delay imposed when the client waits for the gc thread to detect a synchronization request appears to be about 2 milliseconds on average. (The gc thread was polling to detect a synchronization request from the client after copying 3 kilobytes.) This delay is not required by our design and can be eliminated by using fine-grained locking within the garbage collection module to control access to particular collector state variables. We therefore expect further experience with the collector to allow us to improve this aspect of its performance.

4.5 Future Benchmarking Plans

All of the benchmarks we have measured so far use single-threaded client programs, but the implementation does support multithreaded clients. 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 investigating performance questions about the collector that are not answered by this paper. We expect to be able to measure the trapping and synchronization costs in the current implementation. These and other measurements might answer the policy questions raised in Sections 3.3 and 3.4.

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 objects. 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 that 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 that 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. Doligez and Leroy [7] implemented a concurrent collector that uses a mixed strategy to provide collection for a multithreaded version of CAML. Huelsbergen and Larus [8] implemented a concurrent collector for SML/NJ that uses replicating collection. Both of these collectors depend heavily on the fact that ML implementations can distinguish mutable from immutable data. Our technique does not depend on this feature of ML and is therefore more generally applicable.

In Doligez and Leroy's system immutable objects are allocated in private heaps and collected by a replicating stop-and-copy collector. The collector copies live immutable objects into a shared heap. To avoid inconsistent mutable values, all mutable objects are allocated in the shared heap. The shared heap is collected by a concurrent mark-and-sweep algorithm based on Dijkstra [5]. When a mutation causes an immutable object to become reachable from the shared heap, then it is

immediately copied into the shared heap. The use of replicating collection allows the original owner of immutable objects to continue to access the private copy.

The critical difference between their approach and ours is that they do not use replicating collection 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 area size to 32 kilobytes. 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. Doligez and Gonthier further characterize how a multiprocessor garbage collector can be more unobtrusive [6]. We believe this is an important advantage and are attempting to understand how to achieve it in our system.

Huelsbergen and Larus's collector uses an invariant that 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, so it is less efficient than 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 have described elsewhere [11] and obtain a direct comparison.

The work described in this paper extends our previous work on incremental and real-time collection [10, 11] by supporting concurrency among multiple clients and the garbage collector. We use this concurrent collector together with a transaction manager for a persistent heap in which the mutation log also serves as the transaction log [14]. In that work [14] we used replicating garbage collection to demonstrate the first implementation of a concurrent compacting garbage collector for a persistent heap. We showed how to provide good performance for a transactional heap but discussed the concurrent garbage collection algorithm only at a high level. In contrast, this paper explains the concurrent collector design and implementation in detail and explores some of the policy issues that are relevant to providing better control over garbage collection pauses.

6 Future Work

We plan to make additional performance measurements and test various control policies for the concurrent collector (see Section 4.5). Another area that requires study is how to schedule the work of the concurrent garbage collector opportunistically 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. Also, 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 therefore a natural area for future work.

7 Conclusions

We have implemented a simple concurrent garbage collector using replicating 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.

Acknowledgments

We would like to thank the DEC Systems Research Center for support as summer interns in 1990, when we first explored the idea of replicating collection. We thank Sally McKee for showing us how to use jgraph. We also thank an anonymous referee for detailed and useful comments.

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] 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.

[6] D. Doligez and G. Gonthier. Portable Unobtrusive Garbage Collector for Multiprocessor Systems. In *Proceedings of the 1994 ACM Symposium on Principles of Programming Languages*, January 1994.

[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*, pages 217–226. 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] S.M. Nettles and J.M. Wing. Persistance + Undoability = Transactions. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 2, pages 832–843. IEEE, January 1992.

[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 O'Toole, Scott Nettles, and David Gifford. Concurrent Compacting Garbage Collection of a Persistent Heap. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. ACM, SIGOPS, December 1993.

[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.