Design and Implementation of a Concurrent Garbage Collector for Rotor

Archana Ravindar, Y. N. Srikant
archana@csa.iisc.ernet.in, srikant@csa.iisc.ernet.in

IISc-CSA-TR-2005-2
http://archive.csa.iisc.ernet.in/TR/2005/2/

Computer Science and Automation
Indian Institute of Science, India

February 2005

Design and Implementation of a Concurrent Garbage Collector for Rotor

Archana Ravindar, Y. N. Srikant
archana@csa.iisc.ernet.in, srikant@csa.iisc.ernet.in

Abstract

Introducing concurrency into the garbage collection process is an attractive proposition. It improves the pause times considerably making garbage collection feasible for interactive and real-time systems. This report describes the design and implementation of a concurrent collector which we developed for *Rotor*, Microsoft's shared source implementation of .NET. The concurrent collector is found to decrease the pause time by 36.24% on an average, while increasing the elapsed time by 1.75%.

1 Introduction

As programs grew in size and became more complex, managing memory manually turned out to be taxing and error prone. It was hence essential to abstract the problem of memory management from the programmer into a separate module, making automatic dynamic memory management or *Garbage Collection* a necessity.

However traditional garbage collectors were associated with huge overheads which made them unusable in practical systems. Two advances namely the concept of generations [Lieberman 83] and the introduction of concurrency [Steele 75] were responsible for making garbage collection feasible in practical systems.

Typical Garbage collectors (Gcs) paused the program threads during the entire duration of the collection process. However this made garbage collection unacceptable to interactive and real time systems, where response time is a critical parameter. Concurrent garbage collectors were introduced to decrease the pause times associated with garbage collection. These collectors carry out most of the collection process concurrently with the program, pausing the program threads briefly for very short periods of time.

In this report we describe a design and implementation of a concurrent generational collector, that can also run in the stop-the-world mode. The collector is fitted into *Rotor*, Microsoft's shared source implementation of .NET. We also describe how *Rotor* was modified to work with the concurrent collector.

We use an adapted version of the concurrent replication collection scheme described in [Toole 93]. The concurrent replicating collector uses a methodology called *Fromspace invariant* that achieves synchronization between the collector and the program thread at a low cost.

The concurrent collector is found to improve the pause times by 36.24% while increasing the average elapsed times by 1.75%. When run in the stop-the-world mode the collector increases the average pause time by 2.14%. The new collector is found to reduce the scavenge time by 27.5%.

The rest of the report is organized as follows: Section 2 describes in detail, the process and the issues involved in a concurrent collector. This is followed by a brief overview of the concurrent replication collection scheme described in Section 3. In Section 4, we describe how the concurrent replication collector was adapted to build the new collector. Section 5 describes how the collector was fitted into rotor. We present our experimental methodology and results in Section 6 and 7 respectively. Finally we conclude in Section 8 with possible avenues of future work.

2 Concurrent Collection

A concurrent garbage collector is one that carries out most of the collection work concurrently while the program is executing. Schemes like reference counting naturally suit the concurrent gc model. However reference counting is associated with many issues [Wilson 92] that makes it difficult to implement an efficient concurrent version. The other alternative is to make tracing collectors concurrent.

In a tracing collector, the cost of tracing objects dominates the total garbage collection cost, if the number of live objects are not too small. If we can concurrently trace objects as the program executes, the pause times are bound to reduce considerably.

A concurrent tracing collector normally pauses the program threads only when it has to access root information from critical system structures like the runtime stack. In the rest of the report we will look at concurrent collection from a generational tracing collector's view point.

There are three main entities involved in concurrent tracing.

- The *Live object graph*, involving reachable objects
- The *Program*, that keeps changing the graph, while the application is executing. Hence it is known as the *mutator* in gc terminology.
- The *Collector*, that has to trace the live object graph.

As tracing progresses, the program might mutate the live object graph. So the gc has to be alert to these changes in the graph and trace them too. Hence it appears that some kind of synchronization mechanism has to be in place to make sure that the gc traces all the live objects correctly.

2.1 Synchronization between the program and the collector

The kind of synchronization required depends on the tracing mechanism used. The mark and sweep collector introduces a multiple readers, single writer coherence problem [Wilson 92]. Both the collector and the mutator in this case read the live object graph. But only the mutator can write into the live object graph whereas the collector can write only to the mark bits associated with each object.

The copying scheme introduces a more complicated coherence problem- multiple readers, multiple writers problem since both the collector and mutator write into the pointer fields.[1] However an accurate view need not be maintained by both the mutator and the collector. As long as the collector is able to trace the actual live objects correctly it doesn't matter whether it treats some additional objects that are in reality dead, as live. The trade-off here is between cheaper synchronization and conservatism that introduces floating garbage, as described in the next section.

[1] Replication copying scheme however introduces a much simpler problem- multiple readers, single writer coherence.

2.1.1 Tricolor abstraction

The notion of tricolor abstraction introduced by [Dijkstra 78] is meant to ensure that the state of the live object graph is communicated to the GC and the program. The idea is to associate with every object, a color that indicates its state in the tracing process. Several incremental algorithms have used the tricolor abstraction in different ways to suit their purposes. We will discuss one such method here.

Initially all objects are colored white. This signifies that the collector has not yet visited the object while tracing. If an object including its direct children has been traced, it is colored black. If an object has been traced but its children are not, then it is colored gray. The gray objects form a boundary between the already traced objects and the yet to be traced ones.

The correctness of tracing lies on the basis of an invariant: *A black object will never point to a white object.* If the mutator does create a pointer from a black object to a white object it should inform the collector in some way. To understand how important the invariant is in maintaining a consistent view between the mutator and the collector, lets look at the following example taken from [Wilson 92].

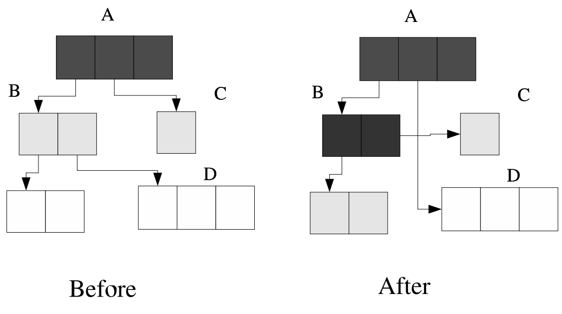

Figure 1: Illustration of Dijkstra's Tricolor Abstraction.

Referring to Figure 1, assuming object A is completely scanned. The collector has just traced objects B and C. After tracing B completely, now suppose that the program makes A point to D instead and B point to C, thus violating the invariant. Now since A has been thought of as completely traced, the collector would not be able to trace D unless it is specifically informed of that fact.

This means that there should be some additional bookkeeping done either while reading or writing pointers to convey the information to the collector. Read barrier methods involve bookkeeping during pointer read, likewise there are write barrier methods that do the same during pointer write.

2.2 Floating garbage

One side effect of concurrent tracing is the creation of *floating garbage*, that is depicted using the following example. Figure 2 shows object A linked with object B. The collector traces A and B and their children as shown in Figure 3. But during program execution A is now linked with C destroying the pointer connecting A and B, deeming B garbage, as shown in Figure 4. Now the gc has to trace C as well, since it is now live, but the tracing of B cannot be undone. Hence B remains in the heap till the next collection cycle as shown in Figure 5 and is termed as floating garbage in gc literature.

3 Concurrent Replication Collection

3.1 Basic scheme of operation

In the concurrent replication scheme [Toole 93], the collector scavenges the live objects concurrently while the program is executing. The main attraction of this scheme is that there is no low level synchro-

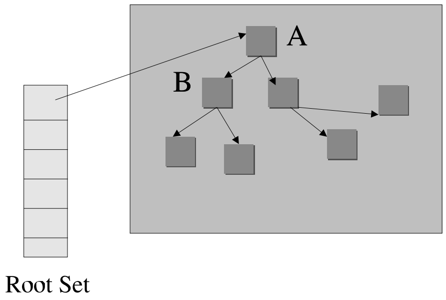

Figure 2: Before tracing.

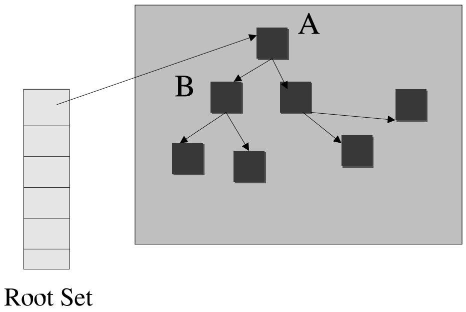

Figure 3: After tracing.

nization required between the program and the collector. Periodically both are brought into sync.

3.1.1 Heap structure

The basic collection model that [Toole 93] use, is that of a copying semispace collector. The heap is a generational one, the young generation is known as the *Newspace*. All new allocations happen in this region. The old generation is further divided into a *Fromspace* and a *Tospace*.

When the allocation reaches a particular threshold in the newspace, a *minor* collection is triggered

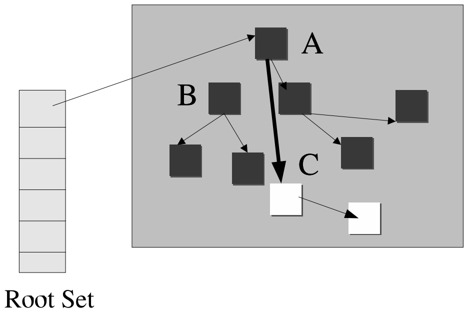

Figure 4: A case of pointer overwrite.

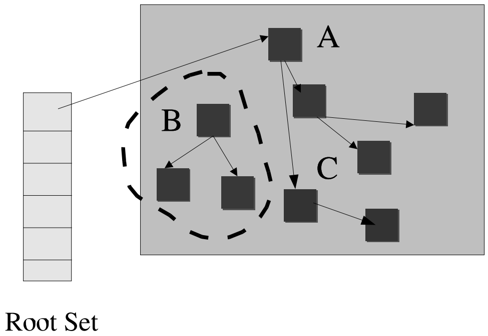

Figure 5: Floating garbage.

where in all the live objects from the newspace are copied onto the fromspace region as shown in Figure 6. The newspace is now entirely reclaimed and is ready for new allocation. This goes on until the allocation in fromspace crosses a particular threshold. When this happens, a *major* collection is invoked wherein the entire heap is collected and all the live objects are transferred into the tospace region. The roles of fromspace and tospace are reversed. The idea is that whenever a major collection is invoked, the tospace should be kept empty.

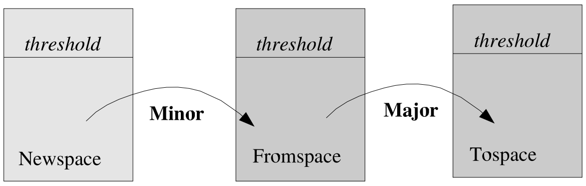

Figure 6: The Heap in the Concurrent Replicating collector.

3.1.2 Synchronization between the program and the collector

The replication collector achieves low synchronization costs by using a methodology called the *Fromspace invariant* as shown in Figure 7. This means, that the program can access (for read and write) only the original live object graph in the fromspace. However the collector tries to replicate the live object graph into the Tospace and hence it accesses the live object graph only for read purposes.

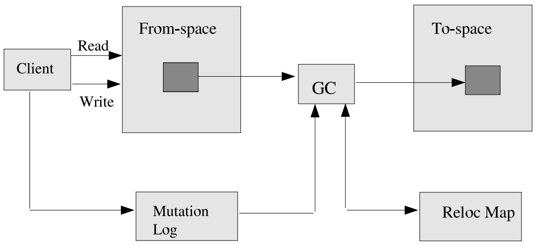

Figure 7: The Collector Model in the Concurrent Replicating collector.

The only situations where synchronization might be actually required would be - The program accesses the live object graph for write and the collector accesses the graph for read. This can introduce a WAR hazard. Suppose the program changes or mutates a part of the live object graph, which was already replicated then these new changes have to get reflected in the replica as well. Otherwise there is no consistency between the view of the live object graph that the program sees and what the collector sees.

3.1.3 Mutation log

To take care of such situations, the replication scheme uses a mutation log. The program enters all the changes that it makes to the live object graph into the mutation log. The collector refers to this log and makes the same changes into the replica. The mutation log is processed periodically and hence the original live object graph and its replica are brought into sync from time to time.

The mutation log and the write barrier required to track intergenerational pointers might be combined in the implementation, since tracking intergenerational pointers is also a kind of bookkeeping.

3.1.4 Object model

Replication copying doesn't destroy the original objects. It stores the new address of every object in its header as shown in Figure 8 to ensure that the collector is able to replicate them correctly and can update all the roots to their corresponding new addresses.

4 Design

The collector that we have developed is similar to the replicating collector in its basic scheme of operation, but there are certain points of difference between the two. In this section we describe the design of the new collector in detail, comparing it with the design of the replicating collector.

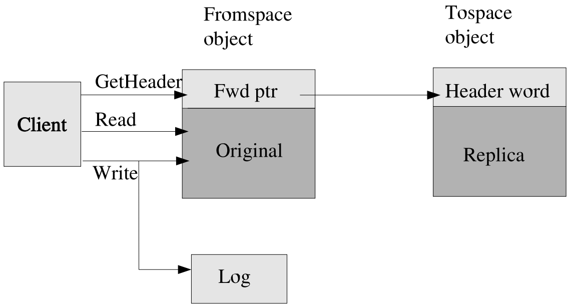

Figure 8: The Object Model in the Concurrent Replicating collector.

4.1 Heap structure and scheduling

The heap structure and the scheduling criteria of the new collector is the same as that of the replicating collector.

4.2 Copying algorithm and object model

The object model of the new collector resembles that of the concurrent replicating collector. We use a depth-first scheme for traversing the live object graph, since it has better impact on locality than other schemes [Moon 84].

4.3 Synchronization between the program and the collector

The replicating collector uses the mutation log to ensure that the live object graph is consistent with the replica graph. However we perform the synchronization operation differently. Our approach for synchronizing the program and the collector closely follows the Dijkstra's tricolor scheme. We associate each object with a color which is used to indicate object state information.

1. *White*: Initially all objects are colored white. This indicates that the object has not yet been considered by the collector.

2. *Gray*: When the collector starts copying an object into the destination space, it colors it gray. The color gray indicates to the program that the collector is right now accessing the object for read.

3. *Black*: When the collector has finished replicating an object into the destination space, it colors it black. The color black indicates to the program that the objcet has already been processed by the collector.

4. *Red*: When the program mutates an already copied object then it is colored red. This indicates to the collector that the live object graph has been changed and the same changes have to be reflected on to the replica graph. At the end of scavenging the collector processes the red objects and re-scavenges them back to the destination area so that the live object graph and the replica are the same.

As shown in Figure 9, lets assume the collector has finished copying object A and is now copying object B. And it goes ahead and copies the entire structure into the destination area as shown in Figure 10. Now the object A is written by the program such that it now points to object C instead of B as shown in Figure 11. Since A and its children were already copied to the destination area, it is colored red. All such red objects are eventually processed by the collector at the end of scavenge phase, making the final heap look like Figure 12.

So this way the coloring scheme substitutes the mutation log in ensuring synchronization. Two bits are needed in the object header to store the color information apart from a single bit used to store the generation and a word to store the new address.

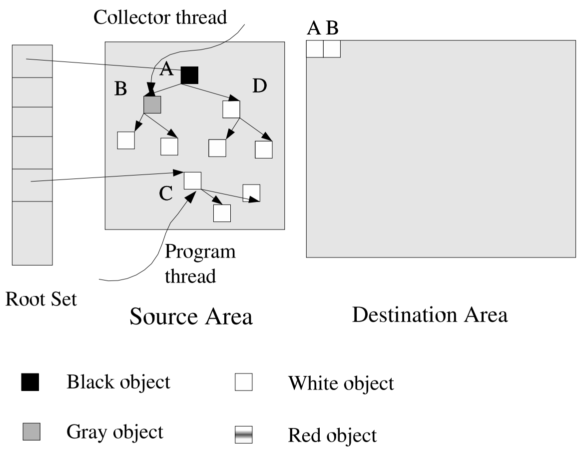

Figure 9: In the midst of tracing object structure 'A'

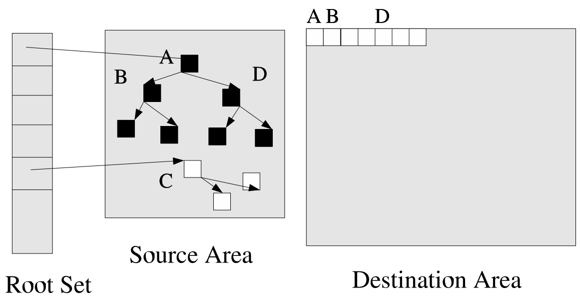

Figure 10: After tracing object structure 'A'

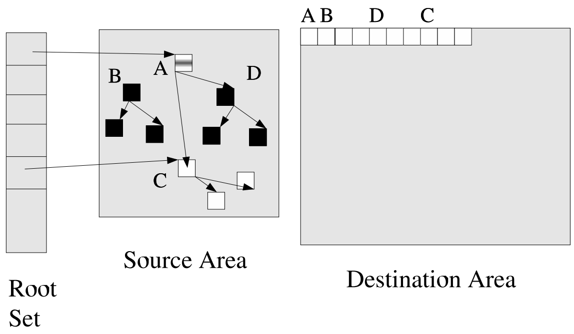

Figure 11: coloring 'A' red

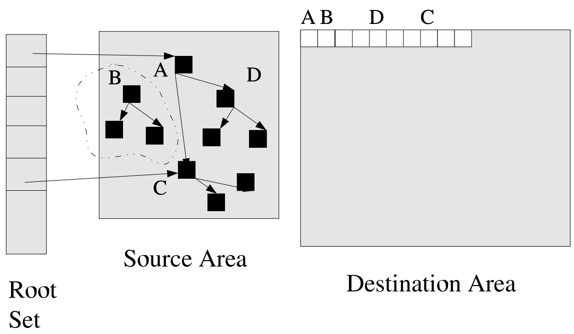

Figure 12: Handling red objects

4.4 Card Tables

The replicating collector uses the simple store list scheme to track intergenerational pointers. But we take the card tables approach [Hosking 92].

4.5 Write Barrier

The write barrier in the baseline collector is used for two purposes. It monitors the object write and ensures synchronization by performing color checks and it records possible intergenerational pointers. The details of how the barrier functions is explained as follows.

1. Synchronization:

   - A write to a white object is ignored; no synchronization required.
   - A write to a gray object waits until the object becomes black; the program waits for the collector to finish copying the object completely before it writes onto it.
   - A write to a black object colors it red; for further processing by the collector.

- A write to a red object is ignored; Anyway the object is going to be processed by the collector.

2. Intergenerational pointers:
   - A write to a young object is ignored.
   - A write to an old object causes the corresponding card table entry to be set; one could always perform an additional comparison to check whether the pointer being written into the old object is a young pointer. But that increases the cost of the write barrier.

4.6 Collection Process

The SML/NJ runtime system described in [Toole 93] has no runtime stack. Hence the heap is extensively used. But the runtime system on which the new collector runs has a stack. The collection process hence needs to access the runtime stack for roots gathering and update.

The new collector operates in two modes which can be set by the user. The collector can run in the stop-world mode where the program thread is stopped for the entire duration of the collection. It can also run in the concurrent mode where the program is stopped only for brief periods of time when critical structures like the runtime stack etc are accessed.

5 Implementation

5.1 Heap Structure

The heap is organized into a set of three semispaces. The young generation comprising of newspace and the old generation comprising of two semispaces-fromspace and the tospace. Each of the semispaces are implemented as a contiguous array of bytes. Since we use copying collection, the allocation methodology is very simple. Allocation involves just incrementing the pointer by the size required.

5.2 Object Model

The collector has to adapt to the existing object model since we are plugging in the new collector into *Rotor* and the object model is widely used in all the other layers of *Rotor*. The object header in *Rotor* stores the method table address and a slot that stores the relocated address of an object. This slot at the same time stores an integer known as the syncblock index, used for object synchronization. The garbage collector in *Rotor* is of the stop-the-world kind. Hence it can afford to use the same word for two purposes. During the collection time, the slot is used to store the new address of the copied object. Once the collection process is over and all the objects are updated to the new addresses, it resumes the function of storing the syncblock index.

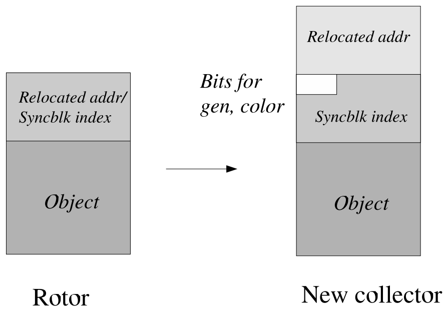

Figure 13: The Object Model in the new collector.

The new collector is a concurrent one, both the relocated address and the syncblock index might be required for use at the same time. Hence we need to have an additional header word to store the relocated address. The size of the object header for the new collector increases by one word (4 bytes) for the new collector. To store the generation and color we make use of the free bits located in the slot for syncblock index. The object model in rotor and the new collector is as shown in Figure 13.

5.3 Roots Scan

For roots gathering and updation, *Rotor* calls two important functions *Promote* and *Relocate* whose main purpose is to mark references in the stack into the root set and update them to their new addresses respectively. We modify these functions to work accordingly with the new collector.

5.4 Card Table

The card table is implemented as a bitmap. Each bit in the card table represents 128 [2] bytes of area in the heap. If the entry is set it indicates that there is an old to young pointer from an object lying in that region. It is supported by a card object header table whose entries correspond to those of the card table. Each entry in the card object header table is an address of the first object occuring in the region corresponding to that entry of the card table. Both these together help in tracking intergenerational pointers.

5.5 Write Barrier

All pointer writes in *Rotor* go through the write barrier which calls the function *setCardTableEntryInterlocked* to track intergenerational pointers. We modify this function to track old to young pointers and to perform synchronization as described in section 4.5.

[2] This value is configurable and is known as the card size.

5.6 Finalization

Finalization can be thought of as an extension of garbage collection. If an object is finalized, certain additional operations are performed automatically when it is reclaimed. Generally objects that manage resources like files etc are ideal candidates for finalization. Finalization allows a resource to gracefully clean itself up when it is being collected [Wilson 92]. Objects that are finalizable are registered in a separate data structure. Once the collection process is over it becomes clear which objects are unreachable. And the unreachable ones that are finalizable get their corresponding finalization operations invoked.

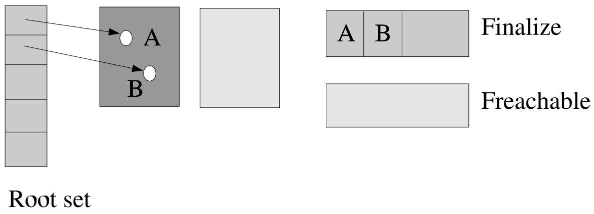

Figure 14: Registering finalizable objects

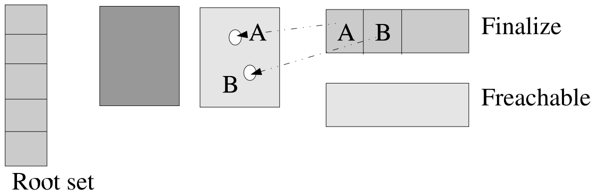

Figure 15: After collection

Finalizable objects are determined by checking whether the object's type contains the finalize method. All such finalizable objects are placed in

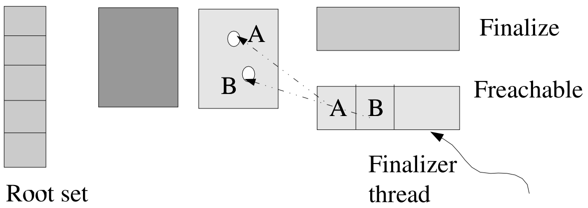

Figure 16: Processing unreachable finalizable objects

a *finalize queue*. The new collector contains two queues to support the finalization feature.

For instance objects A and B are finalizable as shown in Figure 14. Once they are determined to be unreachable in a subsequent collection, they are now shifted to the *freachable* queue as shown in Figure 16. Although the object has become garbage it is still reachable from one of the finalize queues. Hence the name f-reachable. Since these objects are yet to get their finalize methods called, they are promoted to the older generation. Associated with the freachable queue, is a finalizer thread that keeps monitoring the freachable queue for any new entries and empties the queue by calling the finalize methods on such objects.

5.7 Interior Pointers

Pointers that point to a particular field of an object are said to be interior pointers. Although interior pointers are a useful feature [Stutz 03], they complicate the design of the collector. Because a root may no longer be a whole object having a proper header, but could just be an interior pointer. To locate the parents of interior pointers in the young generation we use a bitmap that records all allocation in that area. To locate the parents of interior pointers in the old generation we use the card object header table.

While the collector accesses the stack to gather roots it makes sure that it gets whole objects to scan by invoking a call to get the pointer's parent. The parent of a whole object is itself. Similarly when it accesses the runtime stack for roots update, it makes sure that it works only on whole objects.

5.8 Large Objects

Some collectors provide support for large objects. If the size of an object exceeds a particular limit, it is allocated in a remote area and processed separately. This is desirable because large sized objects may give rise to early collection causing several other young objects to get promoted prematurely.

The new collector however doesn't support large objects allocation. All such objects are allocated in the heap. *Rotor* on startup creates a certain number of large objects. So we add large object allocation support, just enough to handle these set of fixed objects.

5.9 Collection Sequence

The collection sequence in the new collector can be summarized as follows.

1. **Stop the world:** The program threads are stopped.
2. Gather roots.
3. (Only for concurrent mode): **Restart World.**
4. Scavenge all the live objects into the destination area.
5. (Only for concurrent mode): **Stop the world.**
6. (Only for concurrent mode): Scavenge red objects if any into the destination space. [3]

[3]This operation is performed while the world is stopped to avoid the of creation of an indefinite amount of red objects.

7. Scavenge the finalize queue: the finalization process is described in the following subsection.
8. Update roots.
9. **Restart world.**

6 Methodology

6.1 Experimental Platform

This study was conducted on *Rotor* version 1.0 [Rotor 01]. We ran the programs on an Intel pentium III 450 Mhz processor with 128MB of main memory and a 512KB cache, running Free BSD 4.5.

6.2 Benchmarks

To carry out this study, we used the C# versions of the java programs from Spec JVM98 [Spec 98], Java olden [Jolden], Java grande [Jgrande] and the gc test suite provided with *Rotor* [Rotor 01]. The benchmarks and their runtime parameters are summarized in Table 1.

7 Results

In this section we compare the performance of the new collector with *Rotor*'s garbage collector w.r.t pause, collection, elapsed and scavenge times. The programs were run with heap sizes such that both the new collector and the *Rotor*'s collector has the same number of collections for a fair comparison.

7.1 Pause Time

The main objective for choosing a concurrent gc algorithm for the new collector was the reduction in pause times. Almost all the programs report significant reductions in pause times for the concurrent mode of the baseline collector as compared to the rotor GC except for *Raytrace* which reports an increase of 4.62%.

The average reduction in pause times for the concurrent mode is 36.24%. The new collector when run in the stop-the-world mode reports pause time improvements in only 5 out of 11 programs. The average increase in the stop-the-world pause times is 2.14%.

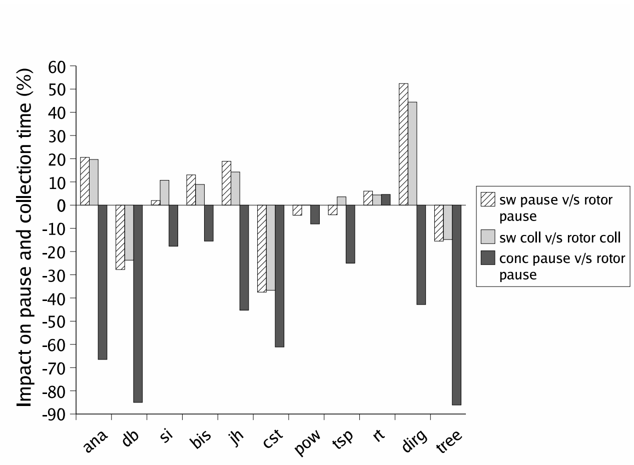

Figure 17: Performance of the new collector w.r.t pause and collection Times.

7.2 Collection Time

We present collection times for only stop-the-world mode, since in the concurrent mode collection time includes the program execution time as well since both the threads are active during collection. We observe in Figure 17, that the graph for stop-the-world collection times and pause times closely follow each other. That is because in the stop-the-world mode there is very little difference between the pause time and the collection time, since the program thread is

Source	Program	Runtime parameters
Rotor gc test suite	directedgraph	No. of vertices=100
Spec JVM98	_208_cst	No of iterations=1, speed=1
	_209_db	No. of iterations=1, Speed=10
	_211_anagram	Speed=1
	_210_si	Speed=10
Java Olden	bisort	No of nodes=4, size=2500
	jhealth	MaxLevel=5, MaxTime=100, seed=23
	power	No of feeders= 5, No of laterals= 10, No of branches= 3, No of leaves= 5
	tsp	Size= 600
	treeadd	No of levels=16
Java Grande	raytrace	Width= 25, height= 25

Table 1: Set of benchmarks used and their configuration

paused for the entire duration of collection. The a
erage increase in stop-the-world collection times
2.79%.

7.3 Elapsed Time

The baseline collector introduces a very small ov
head of 1.11% when run in the stop-the-world mo
However the overhead is slightly worse in the co
current mode. That is because of the additional sy
chronization code that gets executed. The avera
overhead on the elapsed time is 1.75% for the co
current mode.

It can be observed that inspite of a substantial improvement in the pause time, the elapsed times do not change by much. That is because the collection time constitutes a very small portion of the elapsed time.

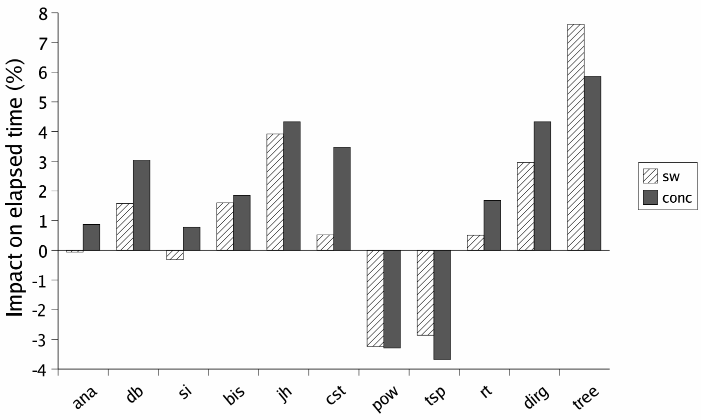

Figure 18: Performance of the new collector w.r.t elapsed time.

7.4 Scavenge Time

We consider the scavenge times only for the stop-the-world mode (depicted in Figure 19), since scavenging is done concurrently for the concurrent mode. The average scavenge time is found to decrease by 27.5%. If we observe the graph as shown in Figure 20, almost all the programs copy the same number of objects with *Rotor*'s own gc and the new garbage collector. However in some programs the number of objects copied in the new collector is more than that in the case of *Rotor*. Thats because the new collector doesnt support large objects. As a result the collector has to scavenge these additional objects as well.

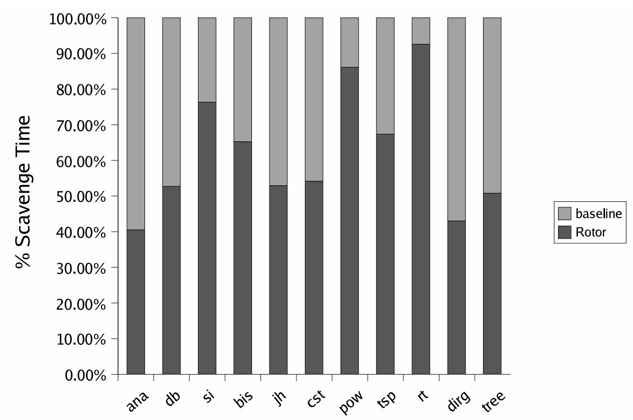

Figure 19: Performance of the new collector w.r.t scavenge time.

8 Conclusions and Future Work

Concurrent collection is fairly successful in bringing down the pause times. Improvement in pause times do not directly translate into improvement in elapsed times, since the collection time occupies a very small portion of the overall elapsed time. Infact it may even worsen the elapsed time, since additional synchronization code gets executed.

There is a limit to the performance improvement that one can achieve working with *object traceability* alone. Certain other aspects of object behavior could prove useful in making collections more efficient. Future work will concentrate on exploiting one such object property known as *object clustering* to do better garbage collection, with the help of techniques like compile time analysis and profiling.

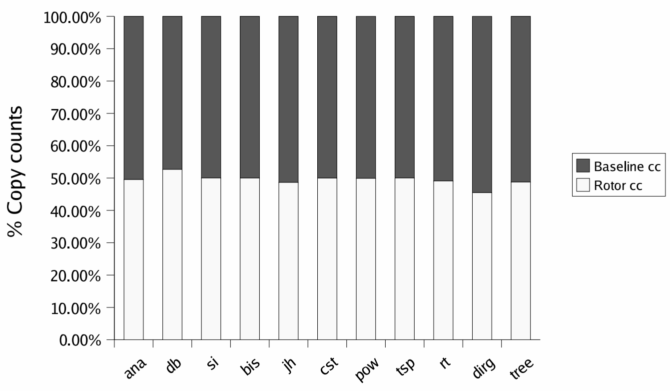

Figure 20: Number of objects copied by the Rotor gc and the new collector.

References

[Boehm 91] H. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, pages 157-164, Toronto, Canada, June 1991.

[Dijkstra 78] Edsger W. Dijkstra, Leslie Lamport, A.J. Martin, C.S.Scholten and E.F.M.Steffens. On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM, 21(11):966-975, November 1978.

[Diwan 92] Amer Diwan, Eliot Moss and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In ACM Conference on Programming Languages Design and Implementation, 1992.

[Hosking 92] Antony L. Hosking, J. Eliot B. Moss and Darko Stefanovic, A comparative performance evaluation of write barrier implementations. In ACM

Conference on Object-Oriented Programming Systems, Languages, and Applications, pages 92-109, 1992

[Jgrande] http://www.epcc.ed.ac.uk/javagrande

[Jolden] Brenden Cahoon, Java Olden benchmarks, http://www.cs.utexas.edu/users/cahoon/

[Jones 96] R. Jones and R. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management. John Wiley and Sons, July 1996.

[Lieberman 83] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. In Communications of the ACM, 26(6):419-29,1983.

[Moon 84] David A. Moon, Garbage collection in a large LISP system. In Steele, editor. Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming, Austin, Texas, August 1984. ACM Press, pages 235-246.

[Printezis 00] Tony Printezis and David Detlefs. A Generational Mostly concurrent Garbage Collector. In the International Symposium on Memory Management, pages 143-155, Minneapolis, 2000.

[Rotor 01] http://www.sscli.net.

[Spec 98] http://www.spec.org/osg/jvm98.

[Steele 75] Guy L.Steele. Multiprocessing Compactifying Garbage Collection. Communications of the ACM, 18(9):495-508. September 1975.

[Stutz 03] David Stutz, Ted Neward and Geoff Shilling. Shared Source CLI Essentials.

[Toole 93] James O' Toole and Scott Nettles. Concurrent Replicating Garbage Collection. In ACM Symposium on LISP and Functional Programming.

[White 98] Derek White and Alex Garthwaite. The GC Interface in the EVM. Technical Report TR-98-67, Sun Microsystems Labs, December 1998.

[Wilson 92] P. Wilson. Uniprocessor Garbage Collection Techniques. In International Workshop on Memory Management, number 637 in Lecture Notes in Computer Science, pages 1-42, St. Malo, France, September 1992. Springer-Verlag.