Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research
Computer Science and Engineering

Report Number: WUCSE-2003-31

2003-04-28

Using Contaminated Garbage Collection and Reference Counting Garbage Collection to Provide Automatic Storage Reclamation for Real-Time Systems

Matthew P. Hampton

Language support of dynamic storage management simplifies the application programming task immensely. As a result, dynamic storage allocation and garbage collection have become common in general purpose computing. Garbage collection research has led to the development of algorithms for locating program memory that is no longer in use and returning the unused memory to the run-time system for late use by the program. While many programming languages have adopted automatic memory reclamation features, this has not been the trend in Real-Time systems. Many garbage collection methods involve some form of marking the objects in memory. This marking requires time... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Hampton, Matthew P., "Using Contaminated Garbage Collection and Reference Counting Garbage Collection to Provide Automatic Storage Reclamation for Real-Time Systems" Report Number: WUCSE-2003-31 (2003). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1077

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1077

Using Contaminated Garbage Collection and Reference Counting Garbage Collection to Provide Automatic Storage Reclamation for Real-Time Systems

Matthew P. Hampton

Complete Abstract:

Language support of dynamic storage management simplifies the application programming task immensely. As a result, dynamic storage allocation and garbage collection have become common in general purpose computing. Garbage collection research has led to the development of algorithms for locating program memory that is no longer in use and returning the unused memory to the run-time system for late use by the program. While many programming languages have adopted automatic memory reclamation features, this has not been the trend in Real-Time systems. Many garbage collection methods involve some form of marking the objects in memory. This marking requires time proportional to the size of the head to complete. As a result, the predictability constraints of Real-Time are often not satisfied by such approaches. In this thesis, we present an analysis of several approaches for program garbage collection. We examine two approximate collection strategies (Reference Counting and Contamination Garbage Collection) and one complete collection approach (Mark and Sweep Garbage Collection). Additionally, we analyze the relative success of each approach for meeting the demands of Real-Time computing. In addition, we present an algorithm that attempts to classify object types as good candidates for reference counting. Our approach is conservative and uses static analysis of an application's type system. Our analysis of these three collection strategies leads to the observation that there could be benefits to using multiple garbage collectors in parallel. Consequently we address challenges associated with using multiple garbage collectors in one application.

Washington University in St. Louis

School of Engineering & Applied Science

Short Title: Real-Time Garbage Collection Hampton, M.Sc. 2003

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

USING CONTAMINATED GARBAGE COLLECTION AND REFERENCE COUNTING GARBAGE COLLECTION TO PROVIDE AUTOMATIC STORAGE RECLAMATION FOR REAL-TIME SYSTEMS

by

Matthew P. Hampton, B.S.

Prepared under the direction of Dr. Ron K. Cytron

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of
Master of Science

May, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

USING CONTAMINATED GARBAGE COLLECTION AND REFERENCE COUNTING GARBAGE COLLECTION TO PROVIDE AUTOMATIC STORAGE RECLAMATION FOR REAL-TIME SYSTEMS

by Matthew P. Hampton,

ADVISOR: Dr. Ron K. Cytron

May, 2003
Saint Louis, Missouri

Language support for dynamic storage management simplifies the application programming task immensely. As a result, dynamic storage allocation and garbage collection have become common in general purpose computing. Garbage collection research has led to the development of algorithms for locating program memory that is no longer in use and returning that unused memory to the run-time system for later use by the program.

While many programming languages have adopted automatic memory reclamation features, this has not been the trend in Real-Time systems. Many garbage collection methods involve some form of marking the objects in memory. This marking requires time proportional to the size of the heap to complete. As a result, the predictability constraints of Real-Time are often not satisfied by such approaches.

In this thesis, we present an analysis of several approaches for program garbage collection. We examine two approximate collection strategies (Reference Counting and Contaminated Garbage Collection) and one complete collection approach (Mark and Sweep Garbage Collection). Additionally, we analyze the relative success of each approach for meeting the demands of Real-Time computing.

In addition, we present an algorithm that attempts to classify object types as good candidates for reference counting. Our approach is conservative and uses static analysis of an application's type system.

Our analysis of these three collection strategies leads to the observation that there could be benefits to using multiple garbage collectors in parallel. Consequently, we address challenges associated with using multiple garbage collectors in one application.

Contents

List of Figures . . . . **v**

Acknowledgments . . . . **vii**

1 Introduction **1**

2 Related Work **4**

2.1 Scoped Memory 4

2.2 Real-Time Copying Garbage Collection 5

2.3 Mostly Non-Copying Real-Time Garbage Collection 6

3 Garbage Collection Background **7**

3.1 Mark and Sweep Garbage Collector 7

3.2 Reference Counting Garbage Collector 9

3.3 Contaminated Garbage Collector 11

3.4 Approach Comparison 13

4 Garbage Collection Experiments **14**

4.1 Performance Metrics 14

4.1.1 Object-Collection Effectiveness 15

4.1.2 Rot-Time 15

4.1.3 Overall Execution Time 16

4.1.4 RT Readiness Ratio 17
4.2 Experiments 17
4.2.1 Experiment Overview 17
4.2.2 Object Collection Statistics 19
4.2.3 Rot-Time Analysis 21
4.2.4 Overall Execution Time 26
4.2.5 Minimum Heap Size Comparison 29
4.2.6 Real-Time Readiness Analysis 31

5 RCGC Classification Algorithm 37
5.1 Algorithm Description 37
5.2 Results 39

6 Concurrent Execution of RCGC and CGC 42
6.1 Problem Overview 42
6.2 Inter-Class Object References 43
6.2.1 A CGC Object References an RCGC Object 43
6.2.2 An RCGC Object References a CGC Object 43
6.3 Proof of Correctness for Reference Counting of CGC Objects 44
6.3.1 Correctness of Summing Reference Counts for Union Operations 44

7 Conclusions and Future Work 46

Appendix A Support Data for Experiments 48

References 53

Vita 55

List of Figures

3.1 MSA Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, and the arrows are references to those objects. 8

3.2 RCGC Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, the arrows are references to those objects, and the numbered boxes represent object reference counts. 10

3.3 CGC Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, the arrows are references to those objects, and the sources of the dashed arrows are the stack frames associated with a given equilive set 12

4.1 SPEC98 Benchmark Properties with MSA Execution Times 18

4.2 Object Collection Statistics for CGC and RCGC (Size-1) 19

4.3 Object Collection Statistics for CGC and RCGC (Size-10) 20

4.4 Min/Max/Avg Percentage of Objects Collected 21

4.5 Average Rot-Times for CGC and RCGC (Size-1) 22

4.6 Average Rot-Times for CGC and RCGC (Size-10) 23

4.7 Average Rot-Times for CGC and RCGC Adjusted by the MSA Period (Size-1) 24

4.8 Average Rot-Times for CGC and RCGC Adjusted by the MSA Period (Size-10) 25

4.9 Object Rot-Time Distribution for CGC and RCGC (Size-10) 26

4.10 CGC and RCGC Slowdown over MSA (Size-1) 27
4.11 CGC and RCGC Slowdown over MSA (Size-10) 28
4.12 CGC Speedup over RCGC (Size-1 and 10) 29
4.13 RCGC and CGC Slowdown over MSA with Smaller Heap Sizes (Size-1) . 30
4.14 RCGC and CGC Slowdown over MSA with Smaller Heap Sizes (Size-10) 31
4.15 CGC Speedup over RCGC with Smaller Heap Sizes (Size-1 and 10) . . . 32
4.16 Heap Sizes Used for Experimental Runs 32
4.17 RT Readiness Ratio: Maximum Allocation Time/Average Allocation Time 33
4.18 RT Readiness Ratio: Maximum Allocation Time/Average Allocation Time over Differing Heap Sizes (`jack`, Size-1) 34

5.1 Percentage of Objects Determined to be Reference Countable 40

A.1 Object Collection Effectiveness Data Table 49
A.2 Average Rot-Time Data Table 49
A.3 Rot-Time Distribution Data Table (Size-10) 50
A.4 Speedup/Slowdown Data Table 50
A.5 Speedup/Slowdown with Smaller Heaps Data Table 51
A.6 Real-Time Readiness Ratio Data Table 51
A.7 Real-Time Readiness Ratio Data Table (`jack` Size-1 52
A.8 RCGC Classification Algorithm Results Data Table 52

Acknowledgments

I would very much like to thank my adviser, Dr. Ron K. Cytron, for his support of my graduate study financially and more importantly intellectually through our discussions and collaborative efforts. I would also like to acknowledge the other members of my committee, Dr. Chris Gill and Dr. Aaron Stump, who are offering their time and expertise to assist in the evaluation of my research.

I would like to thank Dante Cannarozzi, Morgan Deters, Steve Donahue, Jennifer Duemler, Mike Henrichs, Christine Julien, Jamie Payton, and Richard Souvenir who have always been willing to lend assistance throughout my progression through the master's degree program. As friends and colleagues, they have always been willing to share their time and talents to help me achieve my goals.

I extend my gratitude to the members of the DOC group who have provided a venue for lively discussion and the sharing of insights which have bolstered the research efforts of us all.

I appreciate very much the efforts of the department faculty who provide an intellectually stimulating environment in which to pursue a graduate education.

I would also like to acknowledge Peggy Fuller, Jean Grothe, Myrna Harbison, and Sharon Matlock, whose efforts ensure that the department functions effectively. In addition, I send many thanks to the staff of Computing Technology and Services, Mark Bober, Mitchell Henderson, Samantha Lacy, Josh Lawrence, and Allen Rueter, whose willingness to help and level of commitment has been nothing short of extraordinary .

I would especially like to thank my parents, James K. and Kathleen Hampton, for all of their love and support throughout the previous 23 years.

The research represented in this thesis was sponsored by the NSF under grant ITR-0081214.

Matthew P. Hampton,

Washington University in Saint Louis
May, 2003

Chapter 1

Introduction

Real-Time (RT) applications generally require that their performance exhibit predictable behavior. Indeed, RT application developers are often willing to sacrifice some overall performance and efficiency for the ability to predict bounds for certain program characteristics (*e.g.*, execution time or storage requirements). In the past, RT performance was ensured in part by carefully implementing applications at the assembly-language level. With such implementations, memory management for example was performed explicitly as part of the application code. More recently however, RT applications are being implemented in higher level languages such as Java^{TM} [6], C, and C++. When programming RT applications in these languages, operations such as memory allocation and reclamation must be implemented in a manner that is cognizant of the needs of the RT environment.

Most modern programming languages provide some form of *dynamic storage management* [11, 10]. For example the Java^{TM} and C languages offer the primitives `new` and `malloc` respectively as a means for allocating storage from the heap. Java^{TM} also provides a *Garbage Collector (GC)* [10] program for automatic reclamation of memory no longer used whereas C and C++ offer the primitives `free` and `delete` for manual memory deallocation. Most GC implementations use a *Mark and Sweep*

Algorithm (MSA) [10] to detect objects that are no longer used by an application. While this method works well in general, barring extensive modification or knowledge abour a bound on the amount of storage used, it is not suitable for RT applications. This is because the MSA must examine every object in the heap, and thus it is difficult to place a reasonable bound on the execution time of a GC cycle. In this case, provisioning to ensure bounded execution time for GC would be prohibitively expensive; therefore, MSA is not a suitable solution for RT implementations. Were it possible to calculate a bound on storage used, such an approach would have limited application because such bounds are often difficult to determine for programs in general.

Research in the field of dynamic storage management has yielded several alternatives to the MSA approach that provide explicit GC execution-time bounds. In this context, algorithms such as the *Contaminated Garbage Collector (CGC)* [3] and *Reference Counting Garbage Collector (RCGC)* [10] are promising solutions. By adding extra *per-instruction/per-object* bookkeeping, these methods manage to reclaim unused storage without searching the entire object space. As a result, a bound can be placed on the *worst-case* execution time, thus allowing an RT scheduler to make reasonable provisions for GC resource requirements.

This thesis explores the merits of each of these alternative bounded execution time GC methods (*i.e.*, CGC and RCGC) relative to each other and relative to the standard MSA. Four performance metrics are defined and a set of experiments is performed using a group of selected benchmark programs. The data are presented to determine the effectiveness of CGC and RCGC. The performance metrics are as follows:

- **Object Collection Effectiveness:** This refers to the completeness of each method in collecting objects that are no longer used. It is quantified as the percentage of objects collected.

- **Rot-Time:** This is a measure of the amount of time that passes between the point at which an object is no longer used by a program and the time of its collection. The larger the collection delay, the more likely a given collector would be unable to satisfy the memory requirements of a program.

- **Overall Execution Time:** This refers to the overall execution times for the applications for each of the GC schemes.

- **RT Readiness Ratio:** This ratio is defined as the ratio of the time required for worst-case execution of a given program or code segment to that of the time required for the average-case execution of that same component. Calculating this ratio offers a means to determine the degree to which resources are wasted when worst-case performance is assumed in all cases as in the context of RT.

In the chapters that follow, we offer analysis of the relative effectiveness of our approximate garbage collection approaches. Chapter 2 provides an overview of some related work. In Chapter 3, we provide an overview of MSA, RCGC, and CGC. In Chapter 4, we present data that compare the operation of RCGC and CGC to each other and to MSA. In Chapter 5 we offer an algorithm that determines statically if an object is a suitable candidate for RCGC. Chapter 6 discusses an approach to address the concurrent use of multiple garbage collection strategies during program execution. Chapter 7 presents some conclusions and directions for future work.

Chapter 2

Related Work

2.1 Scoped Memory

The Real-Time Specification for Java (RTSJ) [2] avoids the issue of automatic garbage collection through a scoped memory approach. That is, the programmer is faced with the task of creating regions of memory from which objects can be allocated. These regions are tied to execution scopes and are reclaimed at the point of exit of their associated scope. At any given point in the program, the programmer can select from which of the available memory scopes he or she wishes to allocate objects. While this approach addresses the predictability concerns that might be raised by dynamic storage management, it does make the task of application programming more challenging. In order to ensure memory safety, there are rules regarding how objects from different memory scopes can reference each other. Therefore, the programmer must not only have an idea of the program memory structure at the point of allocation but also an understanding of the overall memory structure of the program. Having an alternative to the Mark and Sweep Algorithm (MSA) which also meets the predictability needs of an RT system would remove some of this memory management burden from programmers.

2.2 Real-Time Copying Garbage Collection

Nettles and O'Toole [8] use a copying mark and sweep collector to provide garbage collection support for RT programming. Their approach avoids long execution pauses induced by GC through incremental execution of MSA. They limit the amount of time that is allotted to GC execution in order to meet RT constraints. By making a copy of the live objects in the heap as MSA executes, the program can continue operation. As the live objects of the heap are marked, duplicates of those objects are placed in a separate memory region. In order to ensure data consistency, their approach maintains a mutation log, which tracks changes to data references as the program executes. Once a GC cycle is complete and the mutations have been addressed, an atomic operation that switches the program object space to the GC created copy is performed, and the memory for the original heap can be deallocated.

This approach offers garbage collection that meets the needs of RT systems. However, it is important to note that there is memory overhead required to keep a separate copy of the object heap. In addition to memory overhead, one might conceive of a scenario in which a large number of objects were reachable from a program through a relatively small number of references from the program stack. That is, a very large portion of the heap might be reachable from an application, but only a very small number of objects would be referenced directly from the stack. This scenario would be problematic for the incremental mark and sweep collector. This is because it might spend a great deal of time making a copy of what appear to be numerous live heap objects when a relatively small number of changes in stack references might render most of the heap objects collectible, thus requiring another GC cycle.

2.3 Mostly Non-Copying Real-Time Garbage Collection

Bacon et al [1] present another version of the incremental MSA collector which reduces the need for copying. Their approach makes use of size segregated free-lists. As the collector executes the sweep phase, dead objects can be returned to the approriate list. As a result, copying the objects during collection becomes necessary only when memory fragmentation has become a significant concern. When memory fragmentation is low, there is no need to relocate objects to satisfy a storage request. Therefore, the collector need not create a copy of the live heap as it executes. The authors claim that such fragmentation is rare, and that as a result, their approach "induces lower overhead and offers more consistent utilization than other RT GCs."

There are certainly applications whose execution patterns do actually fragment the memory heap. In the face of fragmentation, the above approach suffers from the same limitations associated with the simple copying collector. That said, a more general solution for RT garbage collection that does not depend on the absence of memory fragmentation would be beneficial.

Chapter 3

Garbage Collection Background

This section describes the operation of each of the Mark and Sweep Algorithm (MSA), Reference Counting Garbage Collector (RCGC), and Contaminated Garbage Collector (CGC). We begin by presenting how each GC method handles the same allocation-garbage collection portion of a program's execution. This is followed with an introductory comparison of the strengths and weaknesses of the methods.

3.1 Mark and Sweep Garbage Collector

The MSA is an exact collector in that it collects all of the objects in memory that are unreachable from a running program. It succeeds in doing this by performing an exhaustive search of the objects in the object heap and marking those objects as still reachable from the executing program. References to unmarked objects are removed from the heap. These tasks are achieved in two phases:

1. The first phase iterates over the object space (*i.e.*, the circles in Figure 3.1) and marks objects that are still live (*i.e.*, shades the objects in Figure 3.1 to indicate a marking).

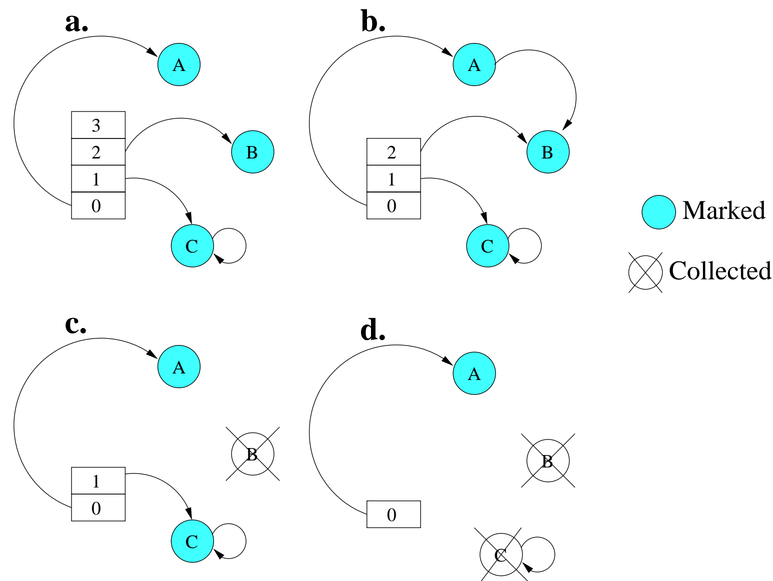

Figure 3.1: MSA Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, and the arrows are references to those objects.

2. The second phase reclaims the memory occupied by objects that were not marked during the execution of phase one.

Figure 3.1 shows an example of how the MSA operates. In each section of Figure 3.1, the program stack is represented by the numbered, vertically stacked rectangles. The object heap is composed of individual objects that are shown as circles containing letters. The stack frames are numbered from earliest (in time) to most recent with stack frame 0 being the initial stack frame. Further, for the purposes of illustration, we may assume that the MSA executes as often as necessary to immediately detect objects the moment they are no longer alive.

Figure 3.1 indicates that there are three objects in the heap, *A, B,* and *C* spawned from stack frames *0, 2,* and *1* respectively. Initially, all of the heap objects are marked (shaded) because they are still *live* (Figure 3.1a). At some point in time after stack frame 3 pops, illustrated in Figure 3.1b, a reference is made from object *A* to object *B*. Executing the MSA at this time again causes each of the objects to be

marked since they can all be referenced by the program. In Figure 3.1c, notice that the reference to object B from A has been removed, and because stack frame 2 has popped, B is no longer reachable from the program. As a result, the collector will mark only objects A and C, and the memory associated with object B is reclaimed. Figure 3.1d shows the point where object C is no longer being referenced from the stack. Although there is still a self-reference to C, the MSA is able to determine that C is no longer reachable from the program because it performs an exhaustive search of the object space. Thus, as the collector executes, only object A is marked and the memory associated with object C is collected.

Since on each execution the MSA must check each object in the heap, detecting the live objects has a complexity of $\Theta(n + e)$, where n is the number of live objects in the heap and e is the number of live references. However, the number of objects is very difficult to predict prior to execution, particularly in situations where program execution is data dependent. Thus, it is difficult to bound the running time of the garbage collection. This is a critical drawback of the MSA in RT situations where RT response to external stimuli requires *a priori* knowledge of the time, or bounds on time, associated with program execution. This is the primary motivation behind exploring alternative garbage collection techniques.

3.2 Reference Counting Garbage Collector

RCGC operates by keeping track of the number of references to a specific object from either the program stack or other objects. Thus, as long as a given object is still being referenced by some entity in the program, the collector will assume that the object is alive. Once an object's reference count reachers zero, it can be placed on a list of objects to be returned to the heap at a specified time. For our purposes, this reclamation was implemented at a stack frame boundary. That is, at each point

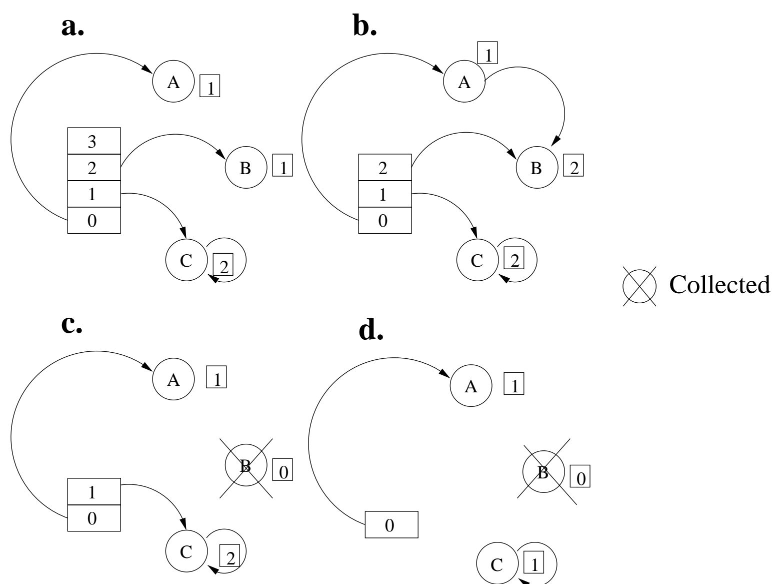

Figure 3.2: RCGC Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, the arrows are references to those objects, and the numbered boxes represent object reference counts.

where a stack frame is popped, dead objects are handed off to the run-time system to be reclaimed. As we will demonstrate shortly, the efficacy of reference counting is limited by the extent to which objects reference each other.

Figure 3.2 depicts the same program as Figure 3.1. However, the RCGC is utilized to reclaim memory. In each stage of execution, we show the reference counts as seen from the collector's perspective. The initial reference counts of the objects are as shown in Figure 3.2a. In Figure 3.2b, notice that the reference from object A to object B causes the reference count for B to be incremented. In Figure 3.2c the reference count for B drops to 0 because stack frame *2* has popped and A no longer points to B. Therefore, B can be collected. The RCGC is unable to detect the death of C in Figure 3.2d, because its reference count is still greater than zero due to the self-reference. This exemplifies the limitation of the reference counting approach. That is, objects involved in reference cycles are never collected by the

reference counting collector because their reference counts will not drop to zero even though they may no longer be reachable from the program.

Unlike the MSA, it is possible to bound the execution time of RCGC to detect live objects. This is because RCGC executes incrementally over the lifetime of the program. RCGC can simply hand a pointer to a list of free objects to the run-time system at the point of collection; hence, RCGC has a collection time complexity of O(1). For each memory reference, RCGC must determine how to adjust reference counts, but this is also a constant time operation. As a result, there is a constant amount of overhead induced on each instruction.

3.3 Contaminated Garbage Collector

The fundamental principle behind CGC is that we can partition the heap into sets of objects that are all regarded as equally live. These sets are called *equilive* sets. That is, if two objects are in the same equilive set, then they will be collected at the same time. Initially, an object is in its own set and is associated with the stack frame in which it is allocated. As the program executes and references are made from one object to another, CGC joins these equilive sets. As stack frames are popped (as with the reference counting approach), a list of objects whose memory can be freed is passed to the memory management subsystem. While CGC is unaffected by cycles that might occur among objects, it does not break apart equilive sets. Sets are not split because the correctness of the approach is based on the assumption that an object in a given set is equally live as other objects in that set. That is, no element of an equilive set will have a shorter or longer lifespan than any other element of that same set. Thus, objects which might reference each other for only a short time remain tied to each other from the collector's point of view.

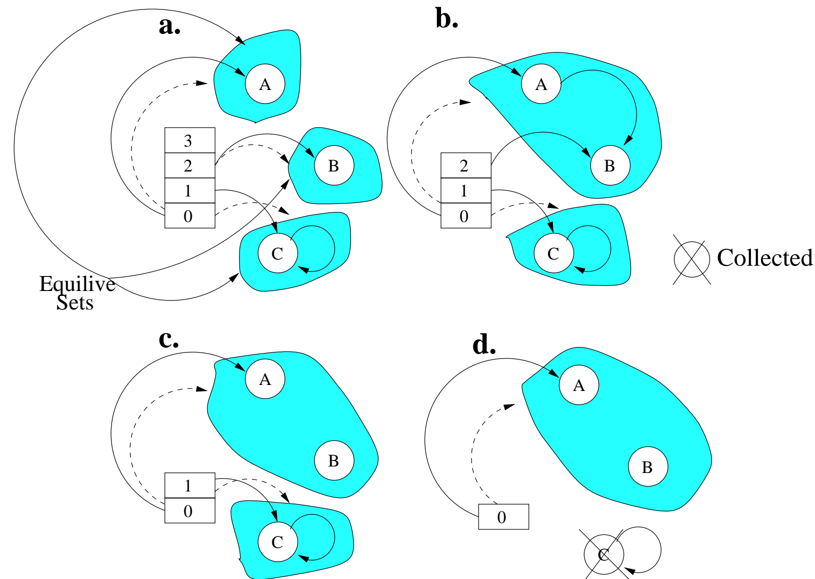

Figure 3.3: CGC Operation: where the vertically stacked rectangles are stack frames, the circles are heap objects, the arrows are references to those objects, and the sources of the dashed arrows are the stack frames associated with a given equilive set .

Figure 3.3 shows the same example program previously considered. The shaded areas represent equilive sets; the dashed arrows show with which stack frame a given equilive set is associated. As we can see from Figure 3.3a, initially all of the objects in the heap are contained in their own equilive sets. However, as the reference from A to B is made in Figure 3.3b, the equilive sets for A and B are unioned. In Figure 3.3c, note that while B is no longer live, CGC fails to detect it. Because A and B were placed in the same equilive set, the collector is not able to detect the death of B. Since A was associated with an earlier stack frame than was B, B cannot be collected until after stack frame 0 pops. While this is a problem for CGC, the issue of cycles (a problem with RCGC) is not. As illustrated in Figure 3.3d, object C, which contains a self-reference, is collected.

As with RCGC, it is possible to bound the execution time requirements of CGC for detecting live objects. It executes incrementally over the lifetime of the

program. The point of collection is a stack frame pop, and at that time, CGC simply hands a pointer to a list of free objects to the run-time system. Thus, as is the case with RCGC, collection with CGC has a complexity of O(1) in terms of the number of objects being collected. Also as with RCGC, overhead is incurred for each instruction. When one object references another, their equilive sets must be merged. CGC uses The *Union by Rank and Path Compression* [4] data structure. As a result, the time required to merge two equilive sets has an amortized complexity of O(α(m,n)) where m is the number of set operations performed, n is the total number of objects in the system, and α is the Inverse Ackermann Function [4]. α grows so slowly that for all reasonable paremeter values, it is less than equal to four. Consequently, for all practical purposes, CGC performs a constant amount of work at each instruction.

3.4 Approach Comparison

Clearly, there are limitations faced by both CGC and RCGC that lead to difficulties in certain situations. That is, because these two methods are approximate collectors, it is possible that they will be incapable of collecting enough objects to provide applications with the memory resources they need. However, we present data that show both methods do reasonably well in terms of object collection.

In RT systems, in addition to the number of objects collected, performance predictability is of concern. Clearly, in this domain, both of the approximate approaches offer a feasible solution. Since MSA must search the entire object space each time it executes, there is no reasonable bound on the computational time required by the MSA. However, as indicated earlier, both CGC and RCGC have constant time complexity thus making them suitable for RT applications. That said, we need to assess the trade-offs associated with each collection approach empirically. The following chapter focuses on a performance comparison of the collectors under study.

Chapter 4

Garbage Collection Experiments

4.1 Performance Metrics

Before presenting the data, let us further examine the performance metrics that will be used and discuss their importance. As noted in Chapter 1, we use four metrics to compare the garbage collection schemes:

Object-Collection Effectiveness This is the percentage of collectible objects collected by the GC algorithm. The Mark and Sweep Algorithm (MSA) theoretically collects 100% of the collectible objects and effectiveness is thus relative to MSA operation.

Rot-Time Rot-Time is the time delay between an object's death and detection of that death by the MSA.

Overall Execution Time This metric is time required for the program to complete the application using one of the GC strategies.

RTReadiness Ratio This is the ratio of worst-case to average-case performance for a given operation.

4.1.1 Object-Collection Effectiveness

Clearly, an important measure of the effectiveness of any garbage collector is the degree to which it is able to collect dead objects. This metric is especially important for the approximate collectors that we have discussed: if their ability to collect objects is too limited (*i.e.*, they don't collect a sufficiency of dead objects or fail to collect them in a timely fashion), then gains in predictability would not be enough to warrant their use. Obviously, the mark and sweep approach can collect all objects that are no longer reachable from a running program. [1] As a result, the MSA is the gold standard for collection effectiveness. The Reference Counting Garbage Collector (RCGC) and the Contaminated Garbage Collector (CGC) are thus compared to the MSA.

4.1.2 Rot-Time

Another important metric for a garbage collection algorithm is the amount of time that passes between the point in the program at which an object is no longer reachable and the point at which the GC is able to collect the object. Rot-Time is a useful metric because it gives us the ability to quantify the amount of extra time dead objects remain in the heap utilizing memory resources. As a collector's Rot-Time increases, the likelihood that memory resources will be exhausted before the collector can reclaim unused storage also increases. A collector which results in objects with excessively large Rot-Times would offer little use even if it were capable of collecting all of the unreachable objects. This is because these objects would stay in the system utilizing memory resources for long periods of time thus mitigating any benefits resulting from their collection. Consequently, as object collection is delayed further, the

[1] The Java^{TM} implementation of MSA approximates stack references. That is, it is possible for the collector to see a primitive on the stack whose value happens to correspond to a valid object reference. Therefore, it is possible the collector will mark objects that should not be.

memory footprint of a given program can only increase since unused memory remains uncollected for a more lengthy period of time.

We present data that compare the relative collection delays associated with CGC and RCGC to each other. We examine two aspects of Rot-Time. We begin our analysis by comparing the average Rot-Time associated with CGC and RCGC. We are able to calculate that value using the following,

$$Average.Rot.Time_{CGC} = \frac{\sum_{i=1}^{n}(T_i^{CGC} - T_i^{MSA})}{n}$$

$$Average.Rot.Time_{RCGC} = \frac{\sum_{i=1}^{n}(T_i^{RCGC} - T_i^{MSA})}{n}$$

where n is the number of objects collected by both RCGC and CGC. The T_i^{MSA}, T_i^{CGC}, and T_i^{RCGC} terms represent the collection time of the ith object by MSA, CGC, and RCGC respectively. In addition to our comparison of the average Rot-Times, we also examine the distribution of the Rot-Times for each of CGC and RCGC. These data show how the collection delay is distributed over all of the objects being considered.

4.1.3 Overall Execution Time

The third metric in our comparison relates to the overall program execution time associated with each collection scheme. Obviously, if it is prohibitively expensive to use a specific GC, predictability advantages aside, an RT application or any other for that matter, would be loathe to take on significant overhead. We present data that compare the relative effects of the different GCs on program execution time.

4.1.4 RT Readiness Ratio

The fourth metric we use is the RT Readiness Ratio. We use the metric to compare how MSA affects the object allocation time during program execution. This ratio is extremely valuable because it provides a means to compare how much overprovisioning might occur under an RT scenario. Because RT systems require predictability, they assume worst-case performance for a given program or code block. As a result, the larger the ratio of worst-case to average-case behavior, the greater the degree to which resources will likely be wasted. We present this ratio for the execution time of object allocation.

4.2 Experiments

In this section, we present experimental data in which we compare the operation of RCGC and CGC with MSA. We begin by providing an overview of our experimental approach. Results and data analysis follow.

4.2.1 Experiment Overview

Our experiments were conducted using the Sun 1.1.8 Java Virtual Machine (JVM) [7] with the JVM modified to provide support for both CGC and RCGC. To obtain information related to object collection effectiveness and Rot-Time, the JVM was modified to allow all three GCs (*i.e.,* MSA, RCGC, and CGC) to execute concurrently. Our current implementations of CGC and RCGC do not allow objects to be collected when any two or more of the collectors run concurrently. Thus, we did not allow any one collector to reclaim objects; instead, we simply took a time stamp at the point a given collector detected an object to be dead. In order to facilitate this operation, we

Name	Description	Lines of Source	Objects Created		MSA Execution Time (sec)	
			Size-1	Size-10	Size-1	Size-10
compress	Modified Lempel-Ziv	6,396	5,148	5,270	334	416
jess	Expert System	570	46,143	106,653	8	57
raytrace	Ray Tracer	3,750	277,055	277,055	44	115
db	Database Manager	1,020	8,354	124,177	1	52
javac	Java Compiler	9,485	26,132	211,109	5	48
mpegaudio	MPEG-3 decompressor	N/A	7,581	9,131	40	417
mtrt	Ray Tracer, threaded	3,750	276,112	276,112	44	151
jack	PCCTS tool	N/A	410,484	410,484	198	394

Figure 4.1: SPEC98 Benchmark Properties with MSA Execution Times

ran each benchmark with sufficient memory to allow the program to execute without actual garbage collection.

In addition, because the Rot-Time calculations require that we have some information about when an optimal garbage collection scheme would collect an object, we used MSA to run continuously to provide optimal data for object collection times. As an approximation to ideal MSA operation, the MSA collector was set to execute on a periodic basis (every 10,000 JVM instructions). Executing the MSA more frequently would have required excessive computation and would not have been reasonable for our experiments. [2] For the execution-time data, each of the garbage collection algorithms was run separately and the times were measured for each of the benchmark applications.

Figure 4.1 contains the SPEC98 [5] benchmarks used in our experiments. It is important to note that two of them (`mpegaudio` and `compress`) are computational in nature and thus do not allocate many objects. We omit the data for the `mtrt` (multi-threaded raytrace) benchmark because of the benchmark's similarity to the single-threaded version, `raytrace`. Each of the SPEC98 benchmarks has three sizes (1, 10,

[2]Executing MSA every 10,000 JVM instructions resulted in each experiment taking on the order of a week.

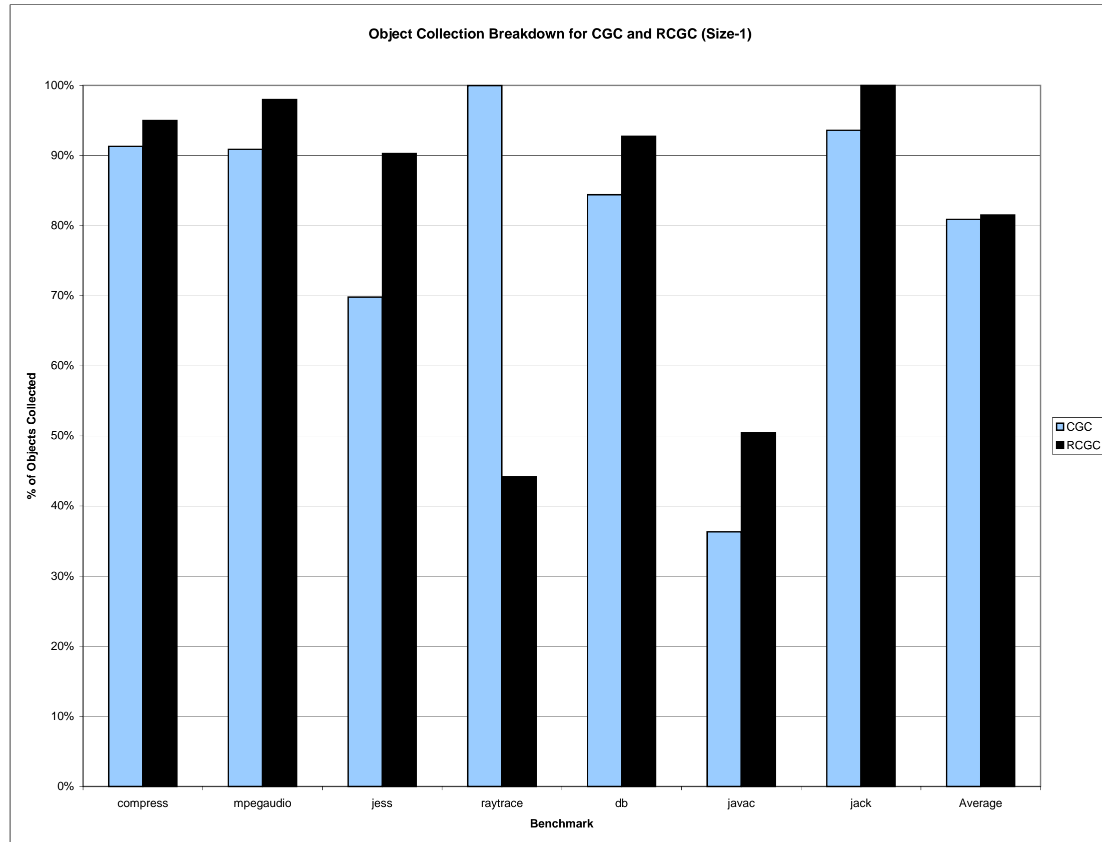

Figure 4.2: Object Collection Statistics for CGC and RCGC (Size-1)

100). A larger size implies a longer execution time and generally the allocation of more objects. For our tests, benchmarks with sizes 1 and 10 were used. We elected not to use the size-100 benchmarks due to time constraints. We selected these benchmarks because their usage is common in the field, and they provide a point of reference when compared with the results seen by others.

4.2.2 Object Collection Statistics

Figure 4.2 and Figure 4.3 depict the number of collectible objects that are detected and reclaimed by our two approximate garbage collection methods for benchmarks

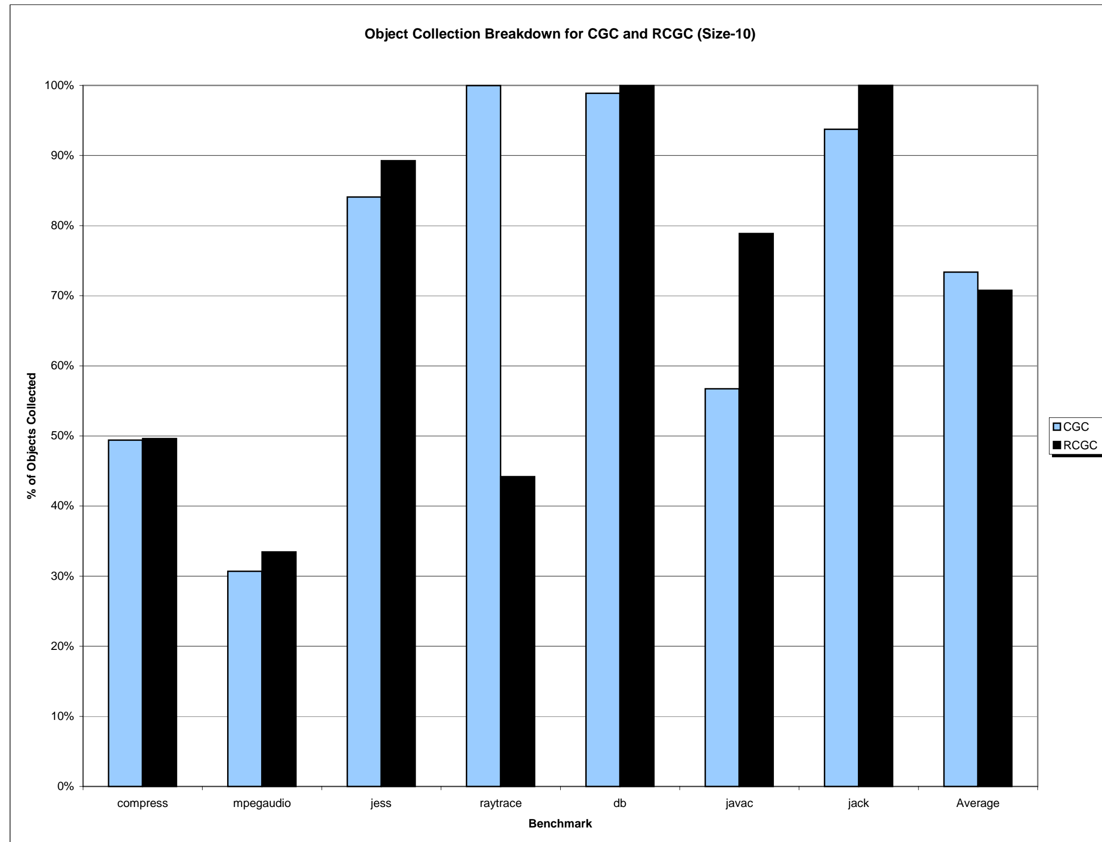

Figure 4.3: Object Collection Statistics for CGC and RCGC (Size-10)

sizes 1 and 10 respectively. The percentages were calculated by dividing the number of objects collected by each of CGC and RCGC by the number of objects collected by MSA multiplied by 100%. The objects collected via MSA represent the total number of objects that could be collected during the execution of the program. Figure 4.2 and Figure 4.3 show that both CGC and RCGC manage to collect a substantial number of the collectible objects ranging from 30% to 100% with a mean of 81.52%. However, at size 10, neither approach does particularly well for `compress` or `mpegaudio`. In fact, the collectors collect fewer than 50% of the collectible objects. Unfortunately, the benchmarks at sizes 1 and 10 are black boxes from our point of view, and there is

	RCGC		
	Min	Max	Avg
Size-1	44.21%	100.00%	81.52%
Size-10	44.21%	100.00%	70.77%
	CGC		
Size-1	36.33%	99.97%	80.90%
Size-10	30.69%	99.97%	73.36%

Figure 4.4: Min/Max/Avg Percentage of Objects Collected

not sufficicient information to determine what differences between the two benchmark sizes might cause the performance degradation. In addition, notice that the reference counting and contaminated approaches are fairly comparable for most of the benchmarks, but the `raytrace` application shows relatively poor RCGC performance. The results are aggregated in Figure 4.4. [3]

While it is obvious that there are applications for which the approximate collectors might not be appropriate solutions (*e.g.*, `compress, mpegaudio`), these results show that there are many applications for which reference counting or the contaminated collector work well from the perspective of object collection. In addition, we can see from the data that as a general rule, RCGC is slightly more effective at object collection than CGC because it tends to collect a higher percentage of the collectible objects for most of the benchmarks.

4.2.3 Rot-Time Analysis

We now offer experimental data that provide a means to compare CGC and RCGC on the basis of how quickly they are able to determine an object to be dead. For this

[3] The support data for Figure 4.2, Figure 4.3, and Figure 4.4 can be found in Figure A.1.

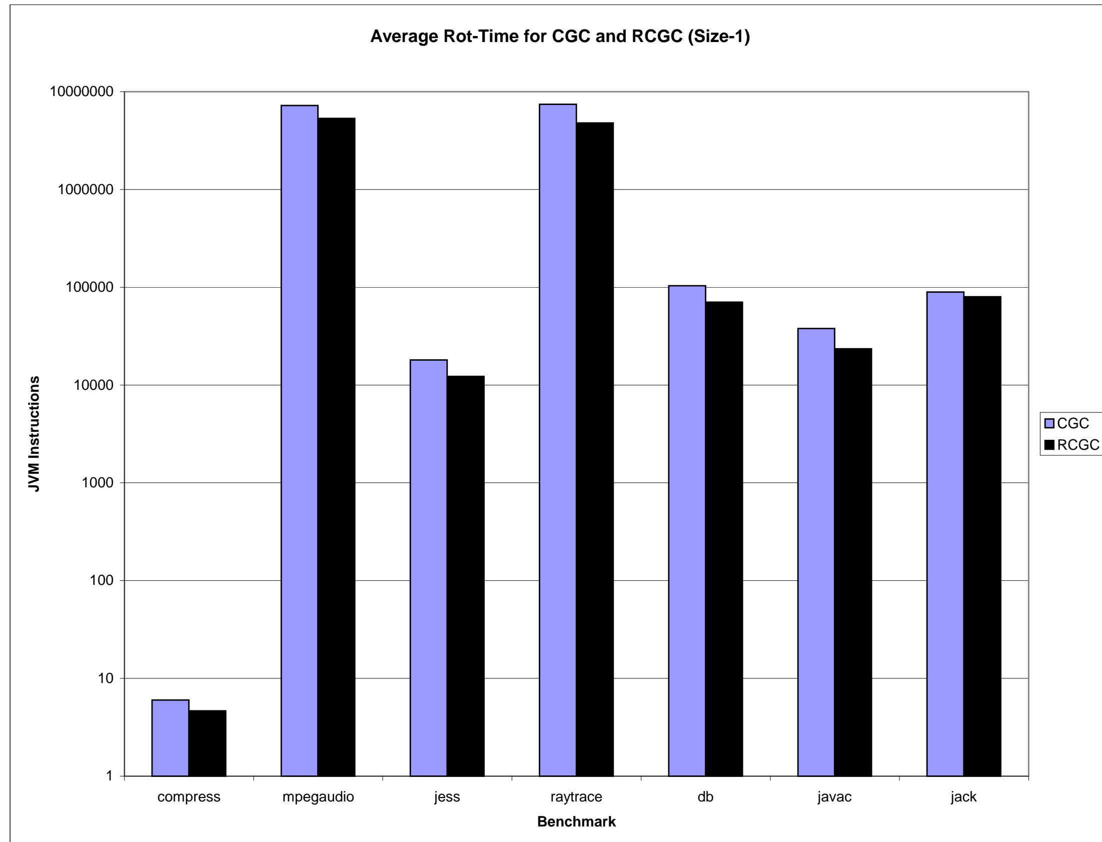

Figure 4.5: Average Rot-Times for CGC and RCGC (Size-1)

analysis, we use the term Rot-Time, which is measured as the number of JVM instructions executed between an object's collection by the MSA and that same object's collection by either CGC or RCGC.

Average Rot-Time

Figure 4.5 and Figure 4.6 show the average Rot-Time for both RCGC and CGC for sizes 1 and 10 respectively. A log scale is used to make the data visually discernable. We can see that the average Rot-Time for CGC is larger than that of RCGC for all of the various benchmarks. For the `db` behchmark at size 10, the CGC average is quite

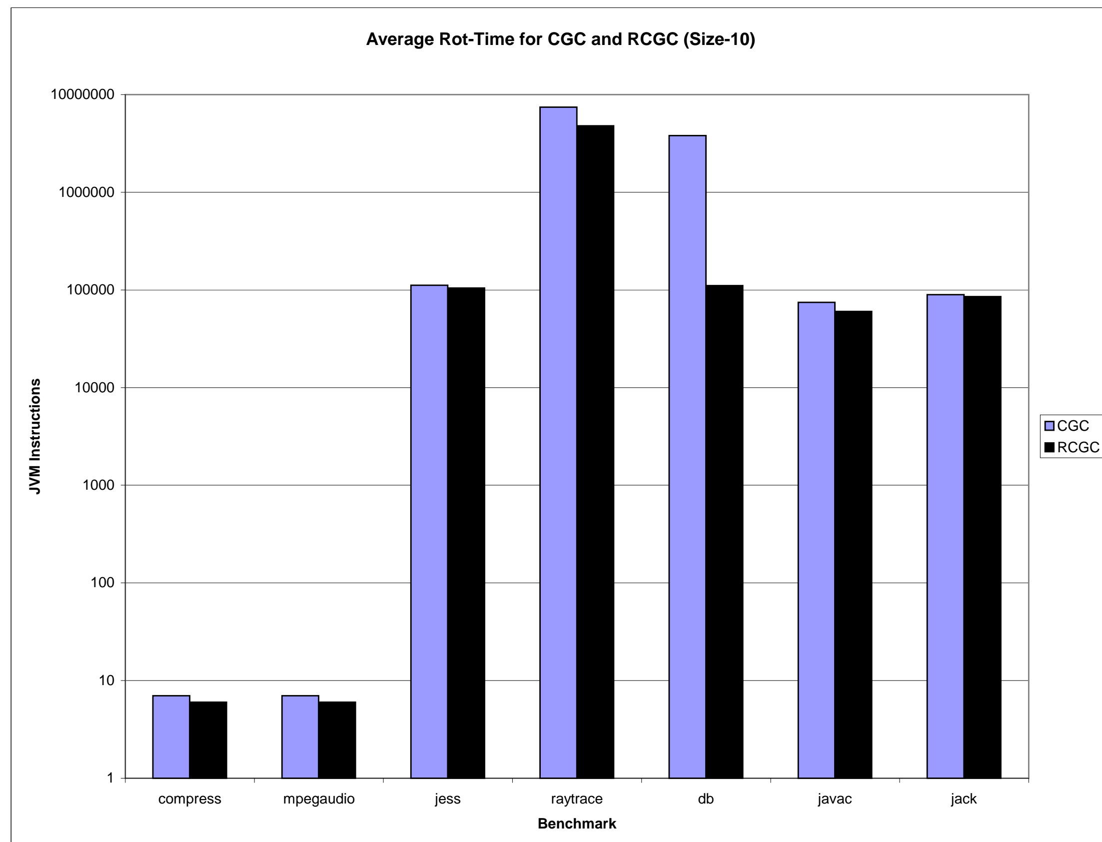

Figure 4.6: Average Rot-Times for CGC and RCGC (Size-10)

large. It is likely there are a high number of short-lived inter-object references. This would explain such a large CGC collection delay.

Because we execute the MSA every 10,000 JVM instructions and not after every JVM instruction, it is possible that our Rot-Time measurements are underestimated by the period of MSA execution. For example, let us assume that we execute MSA at instruction i and an object o dies at instruction $i+1$. We would detect $o's$ death at instruction $i+10,000$ thereby underestimating the Rot-Time by $10,000$ instructions.

Figure 4.7 and Figure 4.8 show the average Rot-Times as they would be were the Rot-Time of every object under consideration increased by 10,000 instructions.

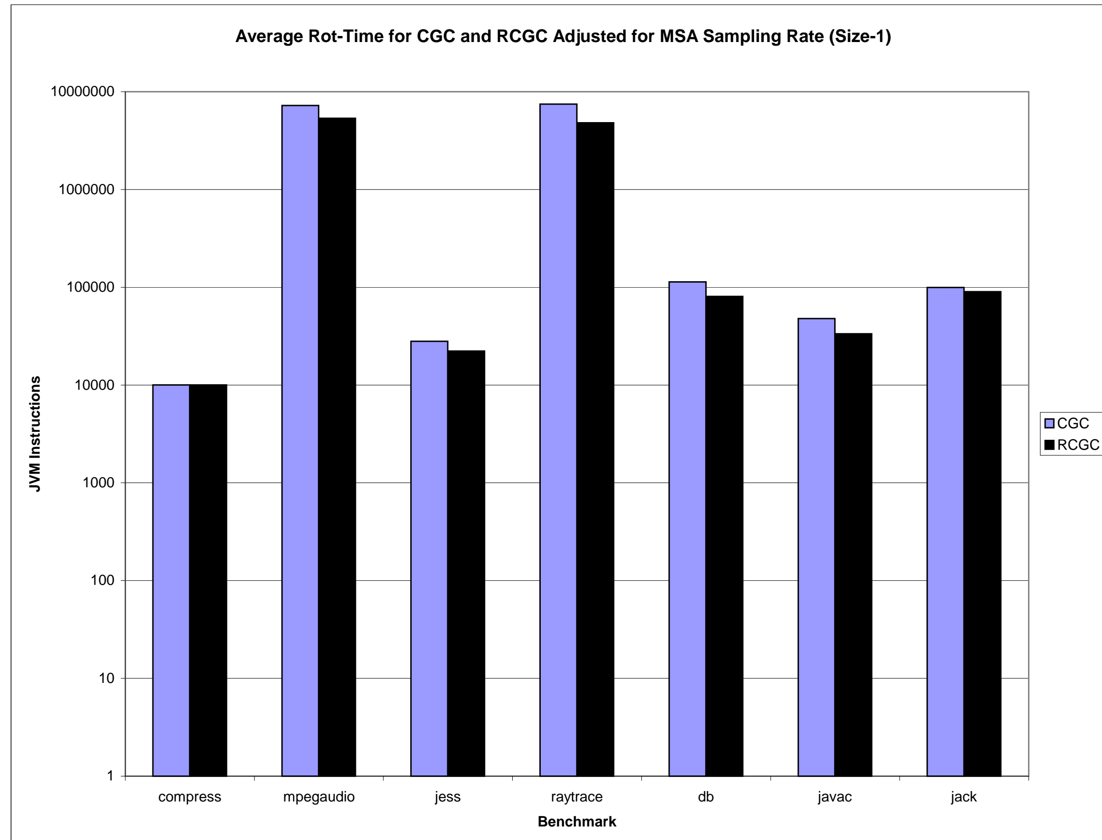

Figure 4.7: Average Rot-Times for CGC and RCGC Adjusted by the MSA Period (Size-1)

Again, we use a log scale for clarity. Even under these worst-case assumptions, the trend mentioned above seems to hold. That is, for most of the benchmarks, the average Rot-Time for RCGC is slightly lower than that of CGC. This suggests that RCGC reclaims unused memory slightly faster than CGC. [4]

Rot-Time Distribution

Figure 4.9 presents the distribution of the objects' Rot-Times aggregated over all the size-10 benchmarks. The same histogram for the size-1 benchmarks are omitted because the data collected for the smaller size yield no new information. We can

[4] The source data for Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8 can be found in Figure A.2.

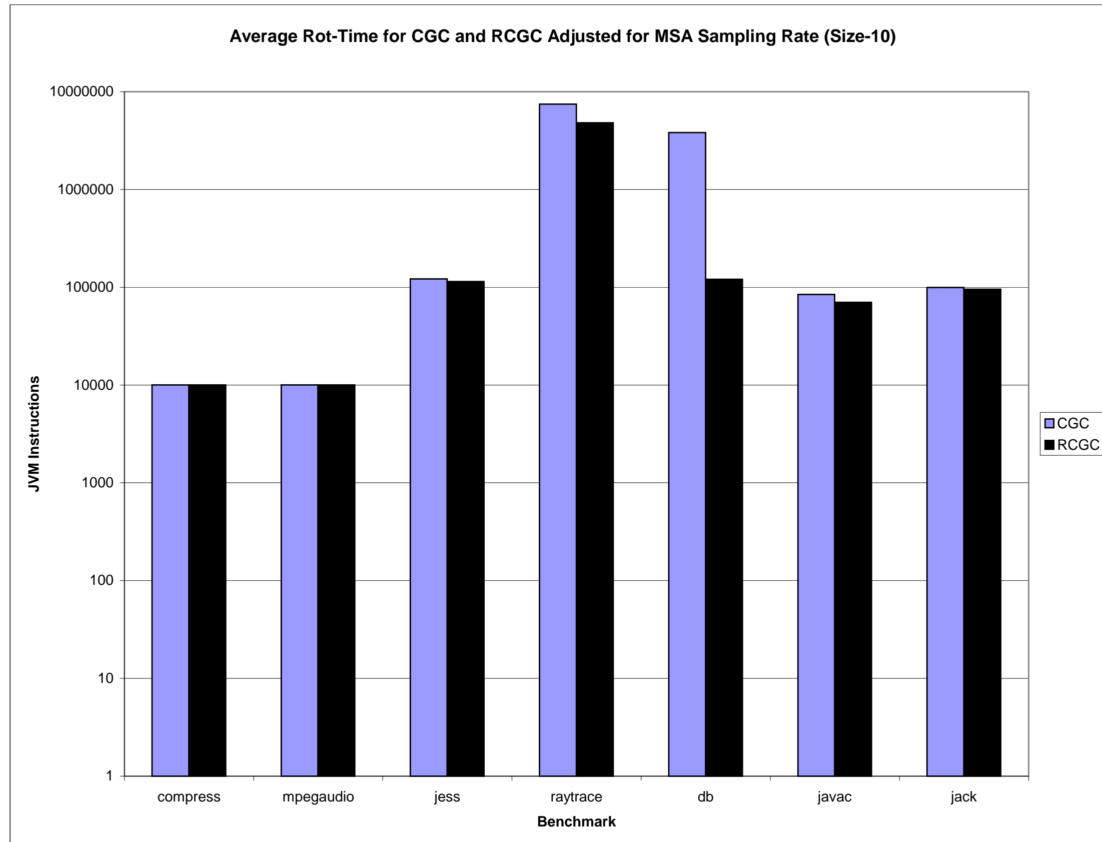

Figure 4.8: Average Rot-Times for CGC and RCGC Adjusted by the MSA Period (Size-10)

see very clearly that the distribution of the Rot-Times confirms what we see with the average Rot-Time statistics presented previously. That is, the distribution for RCGC shows that objects are collected slightly earlier than they are using CGC. Furthermore, for both CGC and RCGC, most (over 60%) of the objects are collected within 10,000 JVM instructions of their creation. This seems reasonable because most programs allocate objects for short-term usage.

Overall, the Rot-Time data show that RCGC collects objects slightly faster than CGC. This reduced collection delay allows unused memory to be returned to

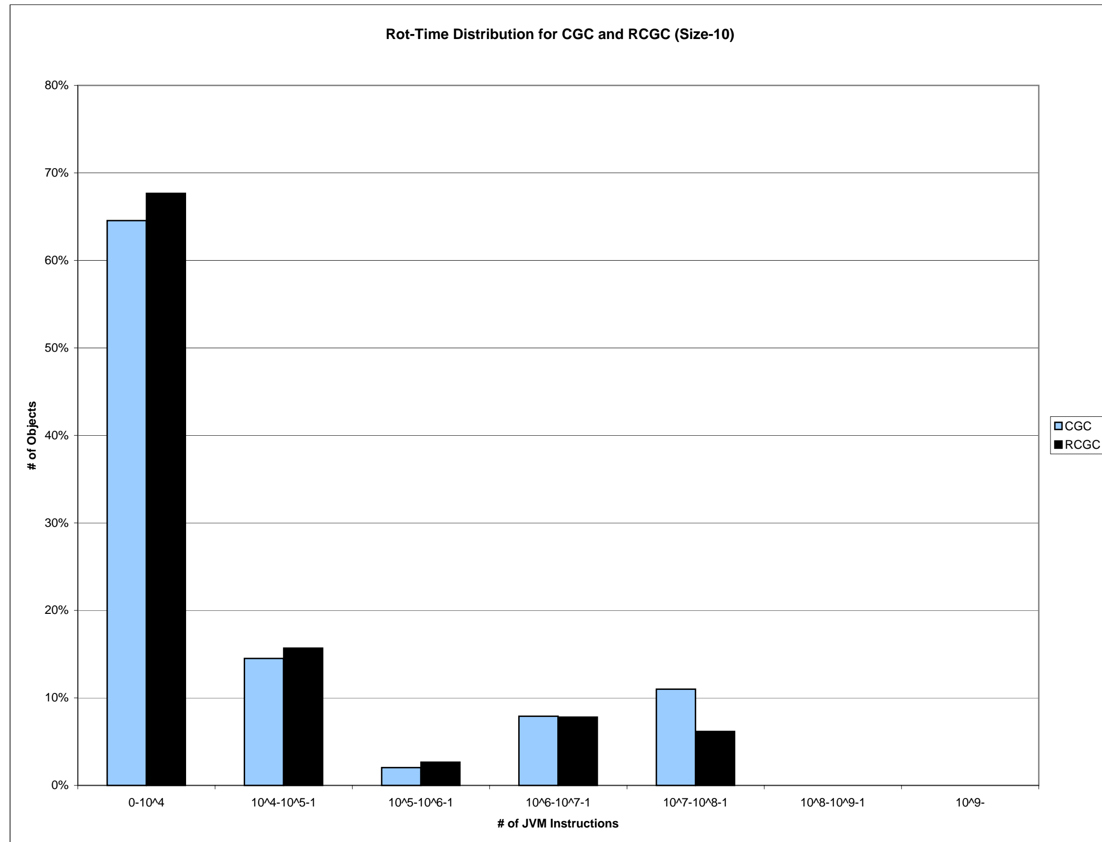

Figure 4.9: Object Rot-Time Distribution for CGC and RCGC (Size-10)

the program at an earlier point in its execution. As a result, the memory footprint required for an application can be reduced by returing dead objects more quickly. [5]

4.2.4 Overall Execution Time

In Figure 4.10 and Figure 4.11, we present a comparison of the execution time requirements for CGC and RCGC with MSA. For the size-1 benchmarks, the use of RCGC and CGC cause no more than a 20% slowdown. The principal exception is the `raytrace` benchmark in which RCGC has a more than 90% slowdown. We see similar

[5] The source data for Figure 4.9 are found in Figure A.3.

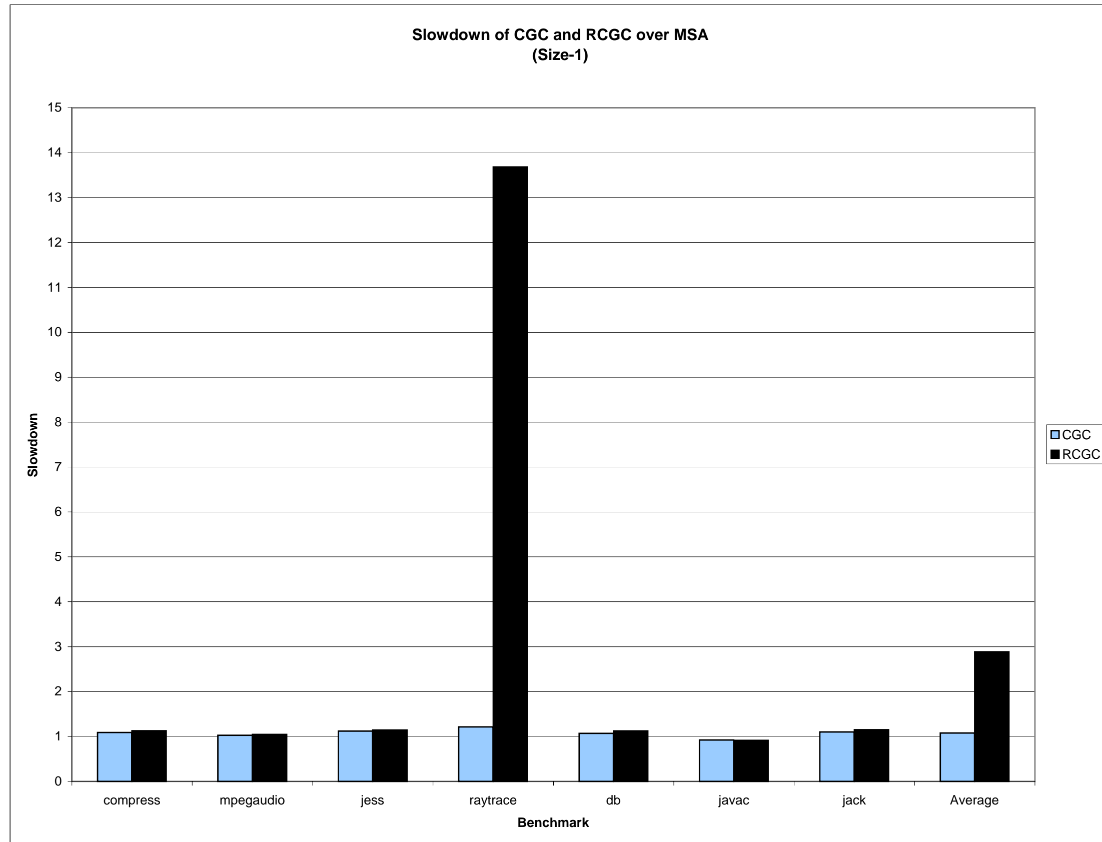

Figure 4.10: CGC and RCGC Slowdown over MSA (Size-1)

behavior with the larger size `raytrace` experiments. This appears to be a result of the poor object collection capability of RCGC with this application (see Figure 4.2 and Figure 4.3). However, the exact cause requires further investigation. For the size-10 benchmarks, Figure 4.11 shows that the slowdown for RCGC and CGC is no more than 20% with notable exceptions for `db` and `raytrace`.

Our experiments were conducted using relatively large heaps to ensure that that mark and sweep collector would not need to execute constantly. In fact, we saw at most two garbage collection cycles during the execution of all of the benchmarks. Thus, even under the best of circumstances for the mark and sweep collector, we see

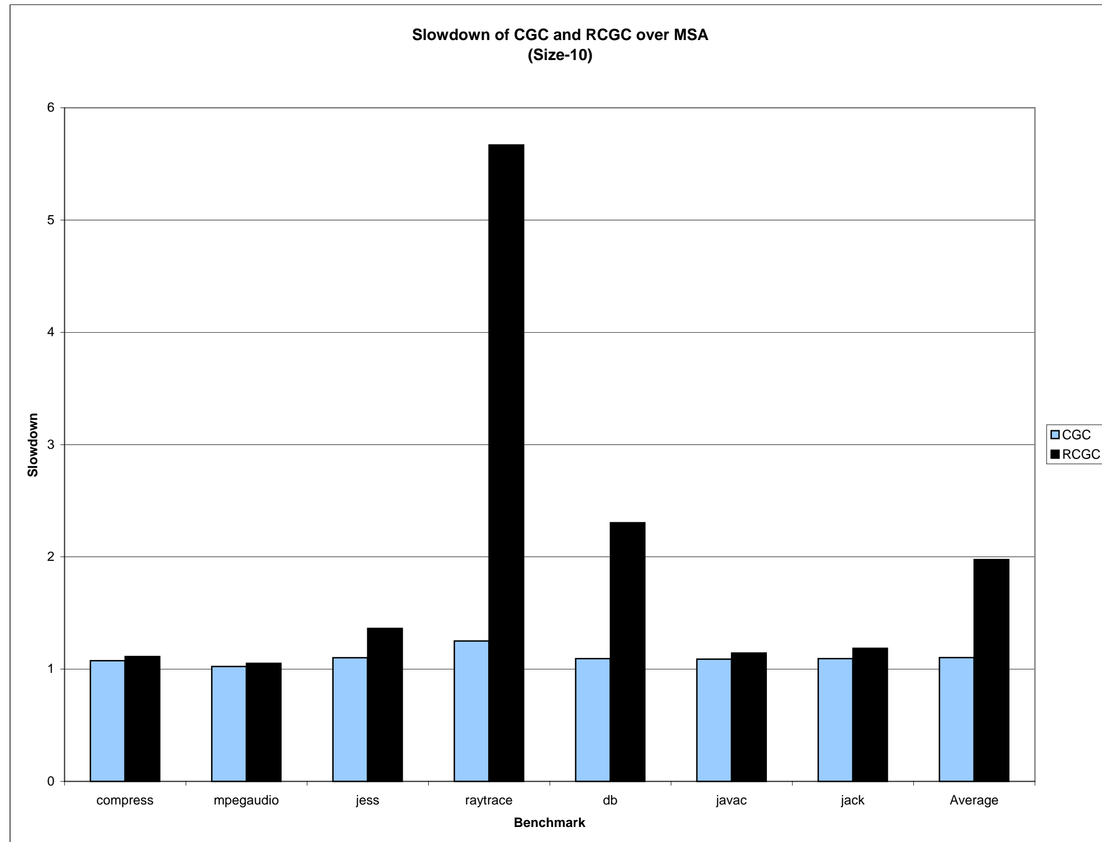

Figure 4.11: CGC and RCGC Slowdown over MSA (Size-10)

that both reference counting and contaminated garbage collection are fairly competitive (within 20% for the most part). Figure 4.12 shows the speedup for CGC over RCGC. It is clear that CGC is faster in general than RCGC. [6] This suggests that the overhead required to update reference counts is larger than that required to manage the equilive sets.

[6]The source data for Figure 4.10, Figure 4.11, and Figure 4.12 can be found in Figure A.4.

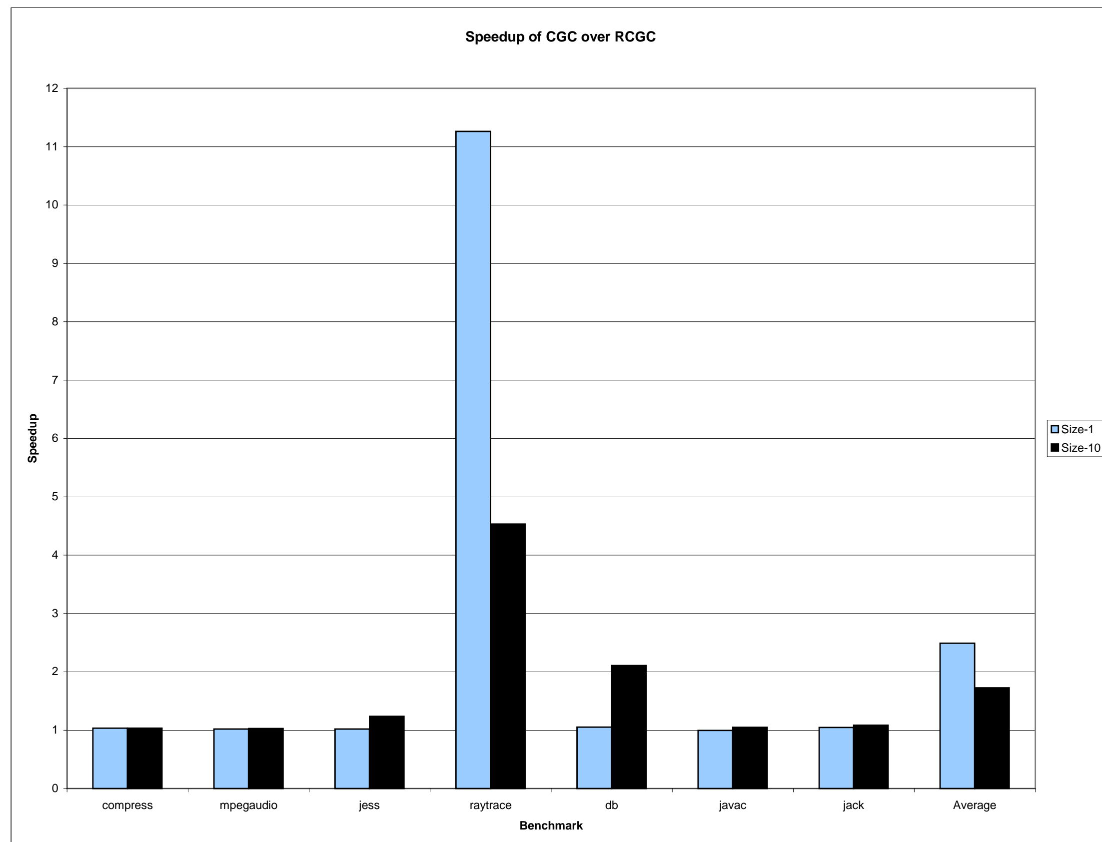

Figure 4.12: CGC Speedup over RCGC (Size-1 and 10)

4.2.5 Minimum Heap Size Comparison

In addition to the data presented above that compare the amount of execution time overhead induced by using CGC and RCGC, we also performed a series of experiments that show what would happen when the benchmarks are run with smaller heaps to force the MSA to execute more frequently. For each of our approximate GC strategies, we found the minimum heap size for which a given benchmark could execute successfully. We then compared the execution times for each benchmark when using either RCGC or CGC with that of MSA at the appropriate heap size.

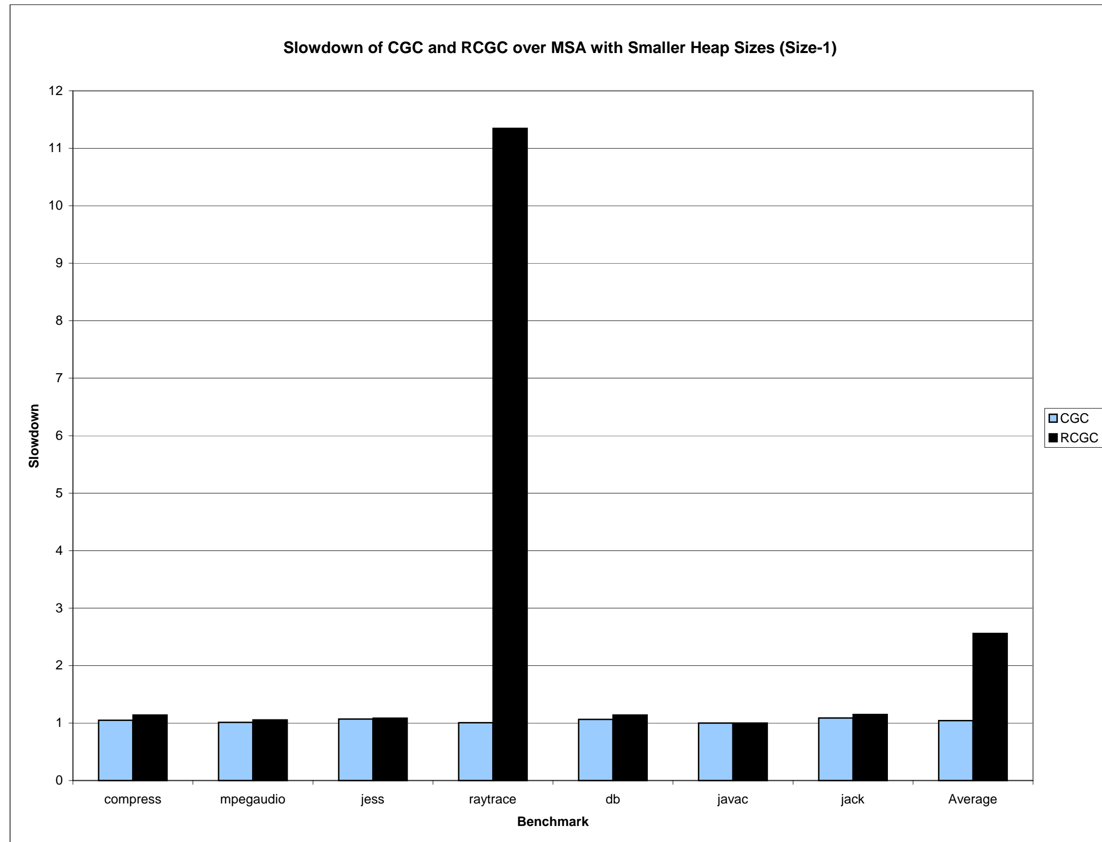

Figure 4.13: RCGC and CGC Slowdown over MSA with Smaller Heap Sizes (Size-1)

Figure 4.13 and Figure 4.14 present data for the experiments described above. The smaller heap sizes forced the MSA to execute more frequently. This reduces the impact of the overhead associated with RCGC and CGC In comparison with the data presented in Figure 4.10 and Figure 4.11 in Section 4.2.4, we can see that there is a slight decrease in the slowdown of CGC and RCGC for several of the benchmarks with the smaller heap. While the effects of the execution of the MSA are small, they are not negligible. We can see that with the smaller heaps, the slowdown of RCGC and CGC is reduced by two to three percentage points. Figure 4.15 compares the execution time of RCGC and CGC directly. The speedup numbers are very similar

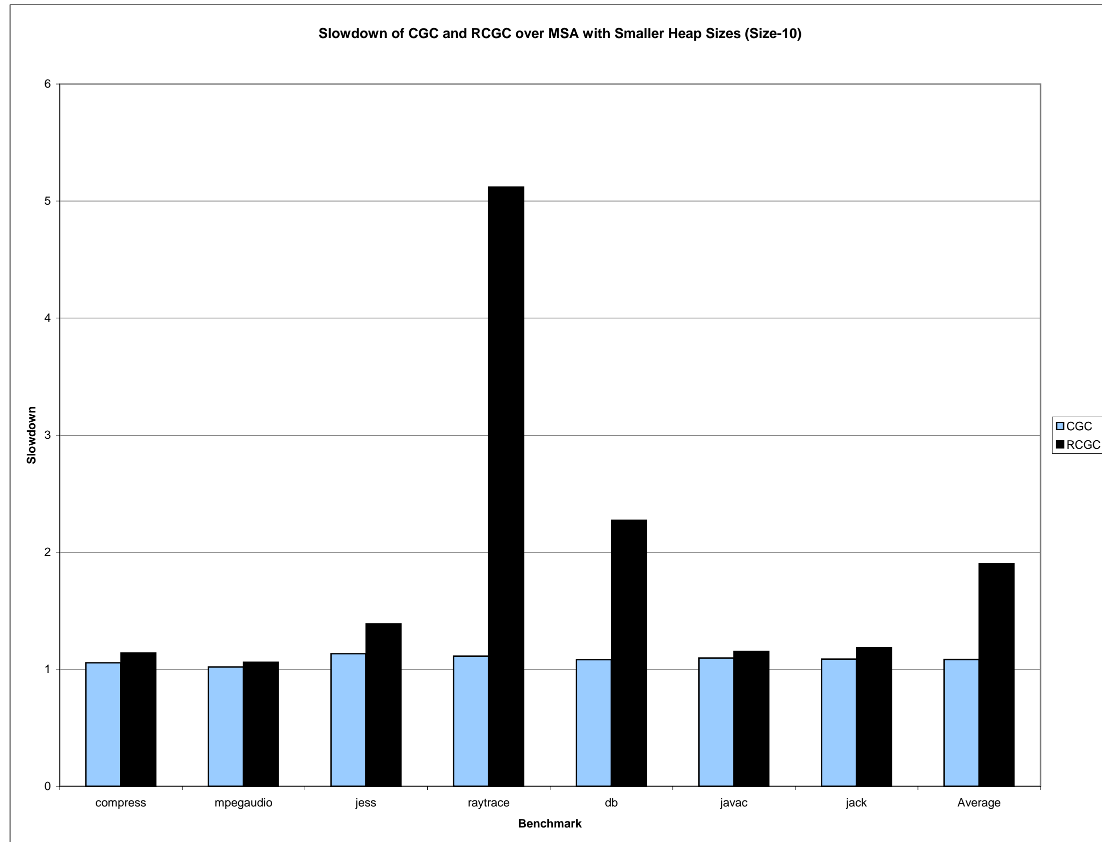

Figure 4.14: RCGC and CGC Slowdown over MSA with Smaller Heap Sizes (Size-10)

to those seen in Figure 4.12 which makes sense given the fact that the work done by both RCGC and CGC is not affected by the heap size. [7] Figure 4.16 displays the heap sizes calculated for each of the benchmarks when using one of RCGC or CGC.

4.2.6 Real-Time Readiness Analysis

In this section, we present data that indicate that the approximate collectors offer a solution for RT systems and that MSA is not appropriate, primarily due to the inability to bound its performance. For any RT environment, a fundamental concern

[7] The source data for Figure 4.13, Figure 4.14, and Figure 4.15 are located in Figure A.5.

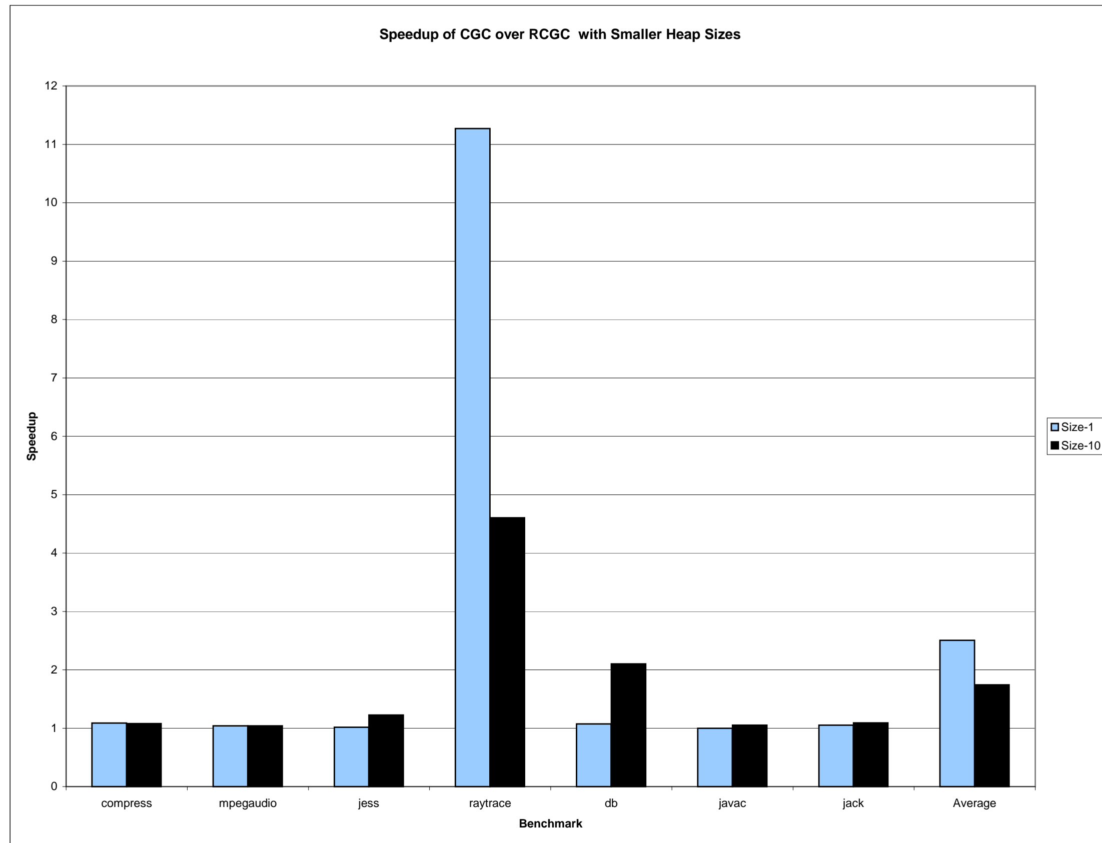

Figure 4.15: CGC Speedup over RCGC with Smaller Heap Sizes (Size-1 and 10)

	Heap Size (KB)			
Bencehmark	**RCGC**	**CGC**	**RCGC**	**CGC**
	Size 1		**Size 10**	
compress	9,697	9,697	10,981	10,981
mpegaudio	507	507	607K	628
jess	724	1,207	2,807	3,307
raytrace	6,884	6,884	7,884	7,884
db	451	551	3,901	4,651
javac	894	975	4,417	6,405
jack	5,141	8,317	5,411	11,340

Figure 4.16: Heap Sizes Used for Experimental Runs

Benchmark	Size-1			Size-10		
	RCGC	CGC	MSA	RCGC	CGC	MSA
compress	100.37	107.74	4,717.97	110.44	200.29	7,871.04
mpegaudio	111.66	141.28	174.19	117.93	245.94	4,250.46
jess	166.21	561.03	2,601.72	1,067.19	652.12	34,084.65
raytrace	47.32	41.38	139,693.02	1,457.26	6,127.74	112,265.75
db	77.19	189.54	2,347.14	3,848.38	280.69	10,715.12
javac	144.84	190.15	13,672.10	1,808.78	190.15	46,064.62
jack	7,284.40	3,835.24	52,408.51	8,078.32	10,835.13	56,760.00
Average	1,133.14	723.05	30,802.09	2,355.47	2,647.44	38,430.23

Figure 4.17: RT Readiness Ratio: Maximum Allocation Time/Average Allocation Time

is predictability in the operation of entities within that environment. As a result, a program that performs quite efficiently on average but with distrastrous worst-case behavior is less preferable than another program with worse average-case performance but with worst-case behavior closer to its average-case. This is because the timing requirements of RT systems require the assumption that the programs operate under worst-case conditions resulting in overprovisioning certain resources, such as time.

One way to measure the degree to which resources might be underutilized due to overprovisioning is to calculate the ratio of the worst-case to average-case performance of a given program or code segment. Obviously, the closer that ratio is to one, the less likely it is that computing resources will be wasted. We use this analysis to compare our garbage collection schemes' fitness for RT.

We compare the ratio mentioned above for object allocation times when using our three garbage collection methods. Because both CGC and RCGC collect continuously, we would expect the ratio observed when using those methods to be relatively small compared to that seen when using MSA. This is because there may be an allocation failure which causes MSA to execute thereby delaying that allocation significantly.

Heap Size	**MSA**	**RCGC**	**CGC**
2MB	34,684.78		
4MB	38,756.67		
6MB	67,803.07		
8MB	82,670.82		
10MB	119,140.45	947.61	527.06
12MB	136,620.71	52.77	76.16
14MB	149,514.84	79.87	43.26
16MB	1,185.33	46.57	53.82
18MB	1,149.45	50.21	47.51
20MB	1,142.92	57.48	44.36

Figure 4.18: RT Readiness Ratio: Maximum Allocation Time/Average Allocation Time over Differing Heap Sizes (`jack`, Size-1)

We obtained this data by timing each object allocation and calculating the average of those times. We then took the ratio of the maximum value observed to the calculated average. Figure 4.17 shows the worse-case to average-case execution time ratio for object allocation in our benchmarks. We can see quite clearly that the ratio for MSA is significantly larger than that for either CGC or RCGC in all cases. Figure 4.17 provides clear evidence that the the unpredictability of MSA is not suitable for the RT environment. On average the ratios are around 27 and 16 times worse for MSA vs. RCGC at sizes 1 and 10 respectively. The ratios are approximately 42 and 14 times worse for MSA vs. CGC at sizes 1 and 10 respectively.

Although the ratios associated with CGC and RCGC are not insignificant, it is likely this phenomenon is related to object size and Java^{TM}'s use of an unsorted freelist allocator. The time required to allocate an object is partially affected by the time required to find free memory for that object and initialize the memory. Therefore, increasing object sizes likely yields longer search and initialization times. Such phenomena are avoidable. The use of an allocation scheme which offers a constant time bound for example would reduce these ratios.

The data were collected from benchmark executions in which the heap sizes were the same as those calculated for the experiments presented in Section 4.2.5. The heap sizes used are found in Figure 4.16. For the MSA experiments, the smaller of the two heap sizes (*i.e.*, the minimum of the RCGC or CGC heap sizes) was used. As a result, these ratios represent a best-case scenario for MSA. This is true because running a given benchmark with a larger heap may result in fewer allocations interrupted by a collection cycle. However, those collection cycles would have to process more objects since the program would be able to allocate more objects without needing a collection. Thus, assuming there is insufficient memory to execute the program without any collection at all, the MSA ratio can only grow as heap size grows.

In Figure 4.18, we track the effects of heap size on the maximum to average allocation time ratio for the `jack` benchmark at size-1. We can see quite clearly that as the heap size increases, the allocation time ratio for MSA also increases. As noted previously, we see this behavior because increasing the heap size allows the program to execute for a longer period of time with garbage collection. As a result, with a larger heap, at the point when collection occurs, there will be more objects to process. That pattern holds as long as there is insufficient memory to run the `jack` program without any garbage collection. The ratios for both CGC and RCGC are not present for the heap sizes of 2,4,6, and 8MB because those heap sizes were not sufficient for the benchmark to complete execution using our approximate collectors. Figure 4.18 shows a large drop in the ratios for both CGC and RCGC as the heap size is increased from 10MB to 12MB. This is likely due to the behavior of the Java^{TM} allocator itself. Because it uses an unsorted freelist to maintain free blocks of memory, larger heaps will tend to reduce the search time required to find a free memory block. Since 10MB is close to the minimum heap size that can be used, it is likely that there are

allocations for which the list search time is very long. While Figure 4.18 corresponds to only the `jack` benchmark, we believe similar patterns would hold for the rest of the benchmarks.

It is clear from Figure 4.17 and Figure 4.18 that MSA is less appropriate for RT systems than is either RCGC or CGC. [8] The nature of MSA is such that bounding its execution and therefore its effects on the execution of object allocation is exceedingly difficult. Furthermore, even if a bound were calculated, the resource requirements called for would likely render an RT schedule infeasible.

[8] The source data for Figure 4.17 and Figure 4.18 are contained in Figure A.6 and Figure A.7 respectively.

Chapter 5

RCGC Classification Algorithm

We have shown that both the Contaminated Garbage Collector (CGC) and Reference Counting Garbage Collector (RCGC) offer effective collection solutions for the RT environment. Further, we have explored the weaknesses of each garbage collection approach. Having seen the cases in which our collectors fail to perform, it seems logical to explore the possibility of determining *a priori* whether one garbage collection scheme might work better for a given object than another. For example, were it possible to know all of the objects involved in a reference cycle, CGC could be used to manage the collection of those objects since RCGC cannot collect objects involved in cycles. Making such an ideal determination is not possible at compile-time given the information available. That said, a more conservative approach that examines the type-system of a given program and determines which types *might* become involved in a reference cycle is presented below.

5.1 Algorithm Description

The fundamental concept behind our approach is the observation that objects may become involved in reference cycles via direct or indirect references. That is, a given

object might have a field that allows it to refer directly to itself or through a chain of references spanning multiple objects. Given this observation, the task we undertake is determining which types could become involved in a cycle. In order to discover the elements of this set, we analyze the types of the fields for each of the object types in the program.

If we view the referencing capabilities of an application's type system as a directed graph, then the problem of finding cycles of object references can be formulated as the well-known problem of finding *Strongly Connected Components* (SCCs) [9]. Thus, our approach first builds a directed graph representing the possible referencing patterns of a given type set; it then finds the SCCs.

The algorithm we describe below is conservative in that it may omit classes that could be reference countable but appear statically to be unsuitable. There is no harm in viewing *any* class as reference countable, except for the overhead in maintaining reference-counting information for objects that cannot be collected using such information.

Our approach is to bulid a graph whose nodes represent instantiable classes and whose edges indicate potential references between classes. An edge is placed between classes x and y if an object of (actual runtime) type x could reference an object of (actual runtime) type y. The liveness of any object *not* involved in a cycle of such a graph can be determined using reference counting.

- A graph is constructed with a vertex for every class type.
- For a field variable of declared type x, let $CouldBe(x)$ denote the set of actual runtime types that could be referenced by the variable of type x. We dscribe the computation of this set below.

- For an actual class of type c, let $HasA(c)$ represent the set of declared variable types in

  $$c, super(c), super(super(c)), \ldots, \text{Object}$$

  This set represents the (declared) types of objects that could be referenced from an instance of c.

We then perform the following computation:

- For each class c
  - For each type t in $HasA(c)$
    - For each type $u \in CouldBe(t)$ place an edge in the graph from node c to u.

Finally, the computation of $CouldBe(t)$ is the *fixed point* of the following:

- $t \in CouldBe(t)$
- If class $c \in CouldBe(t)$ then so is every subclass of c.
- If interface $i \in CouldBe(t)$ then so is every class that implements i.
- If interface $i \in CouldBe(t)$ then so is every interface that extends i.

By repeating the above rules until nothing is added to $CouldBe(t)$ we arrive at a fix point answer.

5.2 Results

In this section, we present results from the execution of our algorithm on the Java^{TM} benchmarks. Figure 5.1 shows the proportion of object types for each benchmark

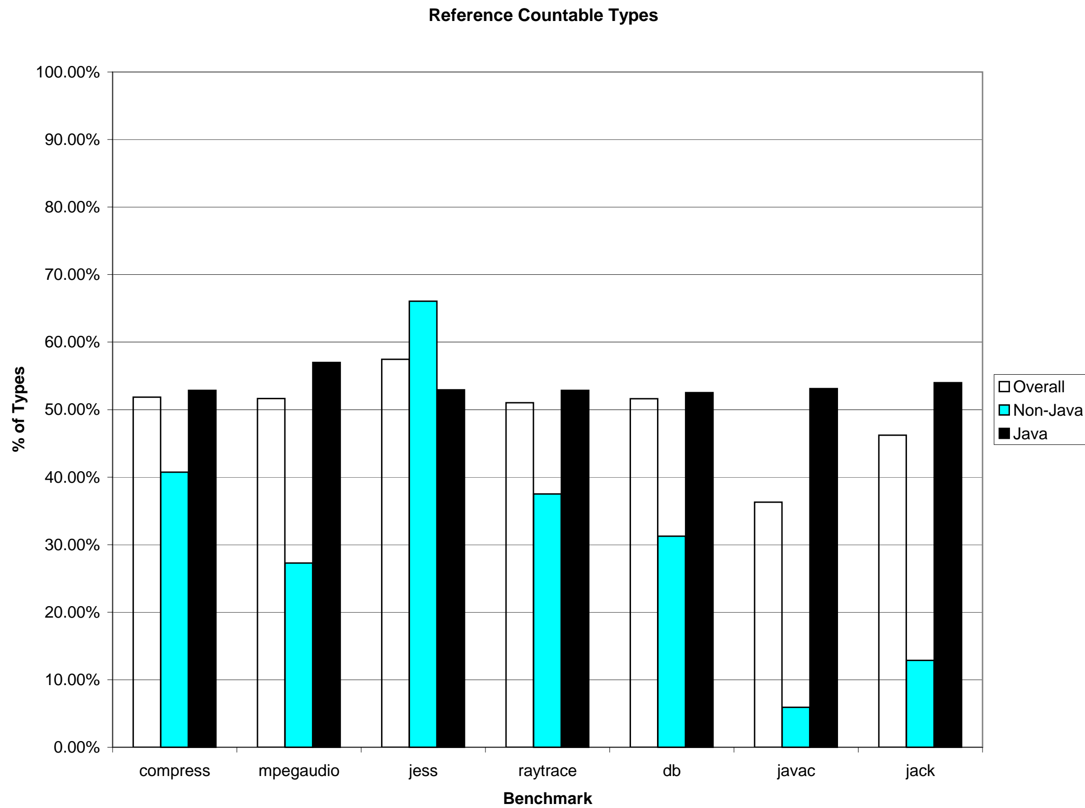

Figure 5.1: Percentage of Objects Determined to be Reference Countable

determined to be reference countable. For each benchmark, there are three data items. There is a value computed for the overall percentage of reference countable object types. The other two percentages were calculated over the object types built-in to Java^{TM} and those object types that are benchmark specific. We can see from Figure 5.1 that a large percentage of the object types used in each benchmark are good candidates for reference counting. Over half of the object types used by the benchmarks are reference countable according to our algorithm. While the benchmark specific results are less impressive, it is important to remember that our algorithm is

conservative. As a result, it is possible that classes not detected to be be reference countable may well never become involved in a reference cycle. [1]

[1] The source data for Figure 5.1 are found in Figure A.8.

Chapter 6

Concurrent Execution of RCGC and CGC

The data presented in previous chapters have shown that there might be some advantage to being able to select the GC mechanism on a per-object basis. To achieve this goal, we must consider the problems associated with using multiple GCs concurrently.

6.1 Problem Overview

Given a program, we wish to separate the objects referenced by that program into two classes: one class of objects is managed by the Reference Counting Garbage Collector (RCGC), and the other by the Contaminated Garbage Collector (CGC). There is one central issue that must be addressed:

- How do we handle references that occur between objects managed by the different collectors? That is, what action should be taken when an object managed by RCGC references an object managed by CGC and *vice versa*?

6.2 Inter-Class Object References

Given an object type space that is divided into two classes (one class managed by RCGC and the other by CGC), there are four possible referencing patterns. However, references that occur between objects in the same set are not a concern since the GC responsible for a given set has a facility for dealing with those references already. Thus, for the purposes of this discussion, we consider only references that occur between objects managed by different collection strategies. In the following, we argue that handling such references is a relatively straightforward operation.

6.2.1 A CGC Object References an RCGC Object

Let us consider an object X, managed by CGC, and an object Y, managed by RCGC. Let us assume that X references Y at some point during program execution. Under these circumstances, nothing needs to be done. The reference counting semantics for Y will ensure its correct collection behavior. As for X, there is no need to perform a union since Y is not part of any equilive set. Further, we need not worry about Y being collected before X. The only way Y can be collected before X is if the reference from X to Y were removed and there were no other references to Y, causing its reference count to drop to 0. In this case, there is no problem because the reference from X to Y no longer exists.

6.2.2 An RCGC Object References a CGC Object

Let us now consider the case in which Y refers to X. In order to ensure the proper collection behavior for X, we simply modify its collection strategy slightly by adding a reference count field to X. This field will keep track of the number of RCGC object references for a given CGC object. Given the operation of the CGC, we can simplify

this strategy by keeping only one reference count per equilive set. That is, the representative object of an equilive set will have a reference count field that will be the number of RCGC objects with live references to members of that set. As a result, we will only detect an equilive set to be collectible if the CGC algorithm itself finds the set to be dead and the reference count for that set is 0. Furthermore, upon a union, the representative element of the newly created set will have a reference count equal to the sum of the reference counts of the two original sets. We now follow with a proof that adding the reference count will not result in improper operation.

6.3 Proof of Correctness for Reference Counting of CGC Objects

Let us begin with the following: let us assume there is an object y managed by CGC that is part of an equilive set S_y. Futher, let there be a set of references of size r from objects not managed by CGC whose targets are elements of S_y. We can define the reference count of y as follows.

$rc(y) \equiv rc(S_y) = \sum_{i=1}^{r}(1)$

As mentioned above, should two equilive sets be merged, we need only add their respective reference counts.

6.3.1 Correctness of Summing Reference Counts for Union Operations

We now argue that the reference count of an equilive set created by the union of two smaller sets is the sum of the reference counts of the two smaller sets. Consider

two equilive sets ES_1 and ES_2 with reference counts of x_1 and x_2 respectively. Let us assume that these two sets are unioned to form a single set ES'. Based on the operation of CGC, we know that ES_1 and ES_2 must be disjoint sets. Therefore, there can be no overlap in the pool of non-CGC object references to ES_1 and ES_2. As a result, the reference count for ES' must be equal to $x_1 + x_2$.

Chapter 7

Conclusions and Future Work

For a programmer, the advantages of automatic garbage collection are well-known. As a result, the use of garbage collection is commonplace in general purpose computing. However, the often unpredictable nature of memory usage during program execution complicates matters significantly in the RT domain. As a result, direct language memory management support has been avoided by RT application developers leaving the task of explicitly allocating and deallocating objects or memory regions (as in the RTSJ) to the programmer. This forces the programmer to maintain at least a partial view of the overall memory structure of a given program at every allocation point. While such an approach addresses RT predictability concerns, it complicates application development.

From the preceding work, we can effectively compare our alternative garbage collection approaches to the Mark and Sweep Algorithm (MSA) and to each other. Additionally, we can comment on the degree to which these collection mechanisms are suitable for RT. It is clear from the data that neither RCGC nor CGC collects all of the objects collected by MSA. However, both methods are reasonably successful in terms of object collection. Further, while both approximate collectors induce overhead, it is not so large an overhead as to mitigate their benefits. Most importantly, whereas

bounding the operation of MSA is impractical, the incremental nature of both RCGC and CGC makes them ideal candidates for garbage collection when RT constraints must be met.

Our work also shows that RCGC tends to perform better in general, however, there are cases in which CGC is a more appropriate choice. Each collector has limitations in the realm of object collection; the fact that these limitations are complementary provides direction for future work. One area of future work might be to determine how we can combine these garbage collection approaches to increase their overall collection effectiveness. For example, we might wish to use RCGC to collect objects containing only primitive data since they cannot become involved in reference cycles. However, we would use CGC to collect objects representing the nodes of a doubly-linked list since those objects may well become part of a reference cycle. In addition, finding a means of using MSA with RCGC and CGC in a limited fashion while still meeting RT demands would be advantageous.

Appendix A

Support Data for Experiments

The data that follow are the source data for the figures presented in the preceding thesis. Figure A.1 is the support data for Figure 4.2, Figure 4.3, and Figure 4.4. The source data for Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8 are displayed in Figure A.2. Figure A.3 contains the source data for Figure 4.9. Figure A.4 depicts the support data used to create Figure 4.10, Figure 4.11, and Figure 4.12. Figure A.5 provides the source data for Figure 4.13, Figure 4.14, and Figure 4.15. Figure A.6 and Figure A.7 contain the the source data used by Figure 4.17 and Figure 4.18 respectively. Finally, Figure A.8 contains the source data used for Figure 5.1 [1].

[1] User types refer to those defined by the benchmarks themselves, Java types are those built-in to Java.

Benchmark	Size-1						
	CGC	RCGC	RCGC&CGC	MSA	Collectible	% CGC	% RCGC
compress	20	45	600	14	679	91	95
mpegaudio	0	49	628	14	691	91	98
jess	0	8,297	28,299	3,942	40,538	70	90
raytrace	151,948	56	120,350	19	272,373	100	44
db	0	319	3,227	277	3,823	84	93
javac	0	2,501	6,434	8,775	17,710	36	50
jack	0	25,044	366,727	14	391,785	94	100
	Size-10						
compress	60	63	656	670	1,449	49	50
mpegaudio	0	62	686	1,487	2,235	31	33
jess	0	5,230	84,913	10,848	100,991	84	89
raytrace	151,948	56	120,350	19	272,373	100	44
db	0	1,313	118,336	20	119,669	99	100
javac	0	44,214	113,184	42,156	199,554	57	79
jack	0	24,444	366,727	14	391,185	94	100

Figure A.1: Object Collection Effectiveness Data Table

Benchmark	Size-10							
	Average Rot-Time				log_{10}(Average Rot-Time)			
	Observed		Adjusted		Observed		Adjusted	
	CGC	RCGC	CGC	RCGC	CGC	RCGC	CGC	RCGC
compress	6	5	10,006	10,005	0.78	0.70	4.00	4.00
mpegaudio	7,224,410	5,746,902	7,234,410	5,756,902	6.86	6.76	6.86	6.76
jess	18,021	15,846	28,021	25,846	4.26	4.20	4.45	4.41
raytrace	7,445,949	4,793,783	7,455,949	4,803,783	6.87	6.68	6.87	6.68
db	103,447	77,499	113,447	87,499	5.01	4.89	5.05	4.94
javac	37,809	32,649	47,809	42,649	4.58	4.51	4.68	4.63
jack	89,541	85,398	99,541	95,398	4.95	4.93	5.00	4.98
	Size-10							
compress	7	6	10,007	10,006	0.85	0.78	4.00	4.00
mpegaudio	7	6	10,007	10,006	0.85	0.78	4.00	4.00
jess	111,675	104,479	121,675	114,479	5.05	5.02	5.09	5.06
raytrace	7,445,948	4,793,783	7,455,948	4,803,783	6.87	6.68	6.87	6.68
db	3,802,600	110,631	3,812,600	120,631	6.58	5.04	6.58	5.08
javac	74,437	60,230	84,437	70,230	4.87	4.78	4.93	4.85
jack	89,540	85,398	99,540	95,398	4.95	4.93	5.00	4.98

Figure A.2: Average Rot-Time Data Table

Benchmark	Size-10							
		$0\text{-}10^4$	$10^4\text{-}10^5\text{-}1$	$10^5\text{-}10^6\text{-}1$	$10^6\text{-}10^7\text{-}1$	$10^7\text{-}10^8\text{-}1$	$10^8\text{-}10^9\text{-}1$	10^9-
compress	**CGC**	522	50	47	0	0	0	37
	RCGC	562	17	47	0	0	0	30
mpegaudio	**CGC**	508	92	47	0	0	0	39
	RCGC	564	44	47	0	0	0	31
jess	**CGC**	75,309	5,855	2,880	750	80	39	0
	RCGC	75,298	5,907	2,844	750	80	34	0
raytrace	**CGC**	38,625	3,438	3,754	31,738	42,795	0	0
	RCGC	44,512	8,743	6,985	31,750	28,360	0	0
db	**CGC**	3,488	95,350	1,755	1,914	15,829	0	0
	RCGC	20,374	95,355	1,185	1,271	151	0	0
javac	**CGC**	110,312	2,320	139	173	240	0	0
	RCGC	109,946	2,779	129	140	190	0	0
jack	**CGC**	329,785	23,520	6,607	6,772	11	32	0
	RCGC	330,138	23,551	6,437	6,565	9	27	0
Total	**CGC**	597,772	134,310	18,841	73,117	101,748	71	76
	RCGC	626,527	145,253	24,630	72,253	57,150	61	61
%	**CGC**	64.56	14.51	2.03	7.90	10.99	0.01	0.01
	RCGC	67.66	15.69	2.66	7.80	6.17	0.01	0.01

Figure A.3: Rot-Time Distribution Data Table (Size-10)

	Size-1					
	Execution Time (s)			Slowdown over MSA		Speedup
Benchmark	**CGC**	**RCGC**	**MSA**	**CGC**	**RCGC**	**CGC over RCGC**
compress	318.96	329.53	292.63	1.09	1.13	1.03
mpegaudio	34.28	34.95	33.39	1.03	1.05	1.02
jess	5.71	5.83	5.10	1.12	1.14	1.02
raytrace	33.75	380.08	27.78	1.21	13.68	11.26
db	0.69	0.73	0.65	1.07	1.13	1.05
javac	3.33	3.32	3.62	.92	.92	.99
jack	70.63	73.94	64.20	1.10	1.15	1.05
	Size-10					
compress	372.72	385.87	346.69	1.08	1.11	1.04
mpegaudio	353.63	363.22	345.53	1.02	1.05	1.03
jess	54.49	67.46	49.50	1.10	1.36	1.24
raytrace	97.62	442.40	78.05	1.25	5.67	4.53
db	43.02	90.68	39.36	1.09	2.30	2.11
javac	31.91	33.53	29.32	1.09	1.14	1.05
jack	140.87	152.72	128.90	1.09	1.18	1.08

Figure A.4: Speedup/Slowdown Data Table

	Size-1					
	Execution Time (s)			Slowdown over MSA		Speedup
Benchmark	**CGC**	**RCGC**	**MSA**	**CGC**	**RCGC**	**CGC over RCGC**
compress	288.34	328.89	292.94	1.05	1.14	1.09
mpegaudio	33.14	34.95	33.35	1.01	1.05	1.04
jess	5.40	5.86	5.44	1.07	1.09	1.02
raytrace	33.51	380.24	31.12	1.01	11.35	11.27
db	0.64	0.73	0.65	1.06	1.14	1.07
javac	3.32	3.32	3.20	1.00	1.00	1.00
jack	64.53	74.06	64.58	1.09	1.15	1.05
	Size-10					
compress	338.95	386.05	345.51	1.05	1.14	1.08
mpegaudio	341.33	361.75	345.11	1.02	1.06	1.04
jess	47.66	66.12	50.08	1.13	1.39	1.22
raytrace	86.24	441.47	83.41	1.11	5.12	4.60
db	39.84	90.56	47.72	1.08	2.27	2.10
javac	29.22	33.64	28.68	1.09	1.15	1.05
jack	128.97	152.71	129.11	1.09	1.18	1.09

Figure A.5: Speedup/Slowdown with Smaller Heaps Data Table

	Size-1								
	MSA			RCGC			CGC		
Benchmark	**Avg**	**Max**	**Ratio**	**Avg**	**Max**	**Ratio**	**Avg**	**Max**	**Ratio**
compress	33.4	157,580.35	4,717.97	2.67	268	100.37	2.84	291.78	102.74
mpegaudio	2.37	412.82	174.19	2.34	261.28	111.66	2.65	374.38	141.28
jess	9.69	25,210.64	2,601.72	2.77	460.4	166.21	2.66	1,492.34	561.03
raytrace	12.42	1,734,987.27	139,693.02	3.79	179.34	47.32	2.5	103.44	41.38
db	5.65	13,261.34	2,347.14	2.44	188.35	77.19	2.64	500.38	189.54
javac	7.97	108,966.62	13,672.10	2.63	380.93	144.84	2.88	547.63	190.15
jack	3.56	186,574.29	52,408.51	2.17	15,807.14	7,284.40	2.44	9,357.99	3,835.24
	Size-10								
compress	36.47	177,646.76	4,871.04	2.78	307.02	110.44	2.85	570.83	200.29
mpegaudio	4.3	18,276.96	4,250.46	2.44	287.76	117.93	2.72	668.95	245.94
jess	5.31	180,989.47	34,084.65	2.59	2,764.03	1,067.19	2.71	1,767.24	652.12
raytrace	12.03	1,350,556.97	112,265.75	2.99	4,357.22	1,457.26	2.41	14,767.85	6,127.74
db	70.14	751,558.54	10,715.12	3.06	11,776.04	3,848.38	2.39	670.84	280.69
javac	4.37	201,302.4	46,064.62	2.36	4,268.71	1,808.78	2.88	547.63	190.15
jack	3.86	219,093.59	56,760.00	2.18	17,610.73	8,078.32	2.39	25,895.96	10,835.13

Figure A.6: Real-Time Readiness Ratio Data Table

	MSA			RCGC			CGC		
Heap Size	**Avg**	**Max**	**Ratio**	**Avg**	**Max**	**Ratio**	**Avg**	**Max**	**Ratio**
2MB	5.71	198,119.49	34,684.78						
4MB	5.84	226,338.97	38,756.67						
6MB	4.09	277,314.56	67,803.07						
8MB	4.48	370,365.28	82,670.82						
10MB	3.77	449,159.48	119,140.45	2.49	2,359.54	947.61	2.64	1,391.43	527.06
12MB	3.84	524,623.52	136,620.71	2.53	133.51	52.77	2.66	202.59	76.16
14MB	4.11	614,506.00	149,514.84	2.53	202.06	79.87	2.65	114.63	43.26
16MB	2.59	3,070.01	1,185.33	2.48	115.50	46.57	2.62	141.00	53.82
18MB	2.65	3,045.79	1,149.35	2.49	125.03	50.21	2.62	124.47	47.51
20MB	2.65	3,028.75	1,142.92	2.54	146.01	57.48	2.64	117.11	44.36

Figure A.7: Real-Time Readiness Ratio Data Table (`jack` Size-1

	# of RC Types		# of Types		% RC Types		
Benchmark	**User**	**Java**	**User**	**Java**	**Overall**	**User**	**Java**
compress	11	158	27	299	51.84	40.74	52.84
mpegaudio	18	171	66	300	51.64	27.27	57.00
jess	107	163	162	308	57.45	66.05	52.92
raytrace	15	158	40	299	51.03	37.50	52.84
db	5	156	16	297	51.60	31.25	52.53
javac	10	162	169	305	36.29	5.92	53.11
jack	9	162	70	300	46.22	12.86	54.00

Figure A.8: RCGC Classification Algorithm Results Data Table

References

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298. ACM Press, 2003.

[2] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java.* Addison-Wesley, 2000.

[3] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, pages 264–273, 2000.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT, 1990.

[5] SPEC Corporation. Java SPEC benchmarks. Technical report, SPEC, 1999. Available by purchase from SPEC.

[6] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[8] Scott Nettles and James O'Toole. Real-time replication garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 217–226, 1993.

[9] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1,2:146–160, September 1972.

[10] Paul R. Wilson. Uniprocessor garbage collection techniques (Long Version). Submitted to ACM Computing Surveys, 1994.

[11] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In Henry Baker, editor, *Proceedings of International Workshop on Memory Management*, volume 986 of *Lecture Notes in Computer Science*, Kinross, Scotland, September 1995. Springer-Verlag.

Vita

Matthew P. Hampton,

Date of Birth June 3, 1979

Place of Birth Cape Girardeau, Missouri

Degrees B.S. *Summa Cum Laude*, Applied Science (Computer Science), May 2001,
from Washington University.

Publications Steven M. Donahue, Matthew P. Hampton, Ron K. Cytron, Mark Franklin, and Krishna M. Kavi. ”Hardware Support for Fast and Bounded Time Storage Allocation” in *Proceedings of the Workshop on Memory Processor Interfaces (WMPI)* in conjunction with the International Symposium of Computer Architecture, Anchorage, Alaska, May 2002.

Steven M. Donahue, Matthew P. Hampton, Morgan Deters, Jonathan M. Nye, Ron K. Cytron, and Krishna M. Kavi. “Storage Allocation for Real-Time, Embedded Systems” in *Proceedings of the First International Workshop on Embedded software*, Washington , D.C., May 2001.

May, 2003