View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Washington University St. Louis: Open Scholarship

Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research
Computer Science and Engineering

Report Number: WUCSE-2003-40

2003-04-15

Contaminated Garbage Collection

Dante J. Cannarozzi

We describe a new method for determining when an object can be garbage collected. The method does not require marking live objects. Instead, each object X is dynamically associated with a stack frame M, such that X is collectable when M pops. Because X could have been dead earlier, our method is conservative. Our results demonstrate that the method nonetheless identifies a large percentage of collectable objects. The method has been implemented in Sun's Java Virtual Machine interpreter, and results are presented based on this implementation.

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

Recommended Citation
Cannarozzi, Dante J., "Contaminated Garbage Collection" Report Number: WUCSE-2003-40 (2003). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1085

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

Washington University in St. Louis

School of Engineering & Applied Science

Short Title: Contaminated Garbage Collection Cannarozzi, M.Sc. 2003

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CONTAMINATED GARBAGE COLLECTION

by

Dante John Cannarozzi

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

CONTAMINATED GARBAGE COLLECTION

by Dante John Cannarozzi

ADVISOR: Dr. Ron K. Cytron

May, 2003
Saint Louis, Missouri

We describe a new method for determining when an object can be garbage collected. The method does not require marking live objects. Instead, each object X is *dynamically* associated with a stack frame M, such that X is collectable when M pops. Because X could have been dead earlier, our method is conservative. Our results demonstrate that the method nonetheless identifies a large percentage of collectable objects. The method has been implemented in Sun's Java$^{\mathrm{TM}}$ Virtual Machine interpreter, and results are presented based on this implementation.

To my family

Contents

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

Acknowledgments . . . . **vii**

1 Introduction **1**

1.1 Background 3

2 Approach **5**

2.1 Example 5

2.2 Summary 9

3 Implementation **11**

3.1 Data Structures and Modifications 12

3.1.1 Objects 12

3.1.2 Frames 14

3.1.3 Static Variables 14

3.1.4 Tainted Objects 15

3.2 Interpreter-Generated Static References 15

3.3 Multiple Threads and Native Code 16

3.4 A Static Optimization 17

3.5 Shrinking the CG Handle Size 18

3.6 Resetting CG Structures During Traditional Garbage Collection 18
3.7 Recycling of CG Objects 19

4 Experiments **21**
4.1 Collectable Objects 21
4.2 Static Objects 22
4.3 Thread Behavior 24
4.4 Size and Age of the Equilive Blocks 26
4.5 Performance and Overhead 28
4.6 Larger SPEC Runs 29
4.7 Resetting Results 31
4.8 Recycling Results 32

5 Generally Related Work **35**
5.1 The Train Algorithm 36

6 Conclusion and Future Work **38**

Appendix A Data **41**

References **46**

Vita **49**

List of Figures

2.1 Frames that keep objects live 6

2.2 Instructions affect object lifetime 7

3.1 Two threads sharing an object 17

4.1 Percentage of objects collectable by our approach, without and with the optimization described in Section 3.4 22

4.2 Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 1) 23

4.3 Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 10) 24

4.4 Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 100) 25

4.5 Distribution of block sizes 26

4.6 Age at death of objects we collect 27

4.7 Timing results. The rightmost column shows the speedup of over traditional collector in the JDK 1.1.8 system for size 1 28

4.8 Timing results. The rightmost column shows the speedup of over traditional collector in the JDK 1.1.8 system for size 10 29

4.9 SPEC benchmarks, large runs 30

4.10 Speedup of our approach over JDK 1.1.8. For the large run, `mpegaudio` and `compress` took over an hour to complete with either system 30
4.11 SPEC benchmarks, small runs 32
4.12 Recycle timing, small runs 33
4.13 Number of objects recycled, small runs 33

A.1 Percentage of objects that we treat as static (live for the program's duration) due to sharing among threads 41
A.2 Object breakdown, small runs 42
A.3 Object breakdown, medium runs 42
A.4 Object breakdown, large runs 42
A.5 SPEC benchmarks, small runs 43
A.6 SPEC benchmarks, medium runs 44
A.7 SPEC benchmarks, large runs 45

Acknowledgments

I would like to thank my advisor, Dr. Ron K. Cytron for his guidance with this work and my committee, Dr. Kenneth Goldman and Dr. Aaron Stump. I would like to thank I thank Guy Steele Jr. and Sun Microsystems for access to their Java Virtual Machine. This work was funded by the National Science Foundation under grant ITR–0081214.

I would also like to thank my parents Dante and Elayne and my sister Melissa for their support and encouragement throughout my academic career.

I am indebted to the current and past members of the DOC group. Without their help, it would have taken me far longer to complete this work. Especially helpful were Morgan Deters, Steve Donahue, and Matt Hampton, as they were constantly available to answer my questions.

I would like to again thank Morgan Deters for his diligent reads and rereads of my thesis for errors. Without his graciously donated editing time, this thesis would have taken much longer.

Dante John Cannarozzi

Washington University in Saint Louis
May 2003

Chapter 1

Introduction

Garbage collection is the recycling of objects (memory) that is know to be "dead" (no longer in use) for the duration of the program. There are several different kinds of garbage collection algorithms including generational and mark and sweep (MSA).

Generational collection is based on the idea that recently created objects are more likely to die than older objects and that as objects age, their chances of dying decreases. The algorithm works by associating objects with generations. Newly created objects are added to the "youngest" generation and can progress to older generations (if they don't die).

Mark and sweep takes a different approach. The first phase (marking) traverses all the reachable (live) objects and marks them. This phase starts with what are called the "roots of computation", which refers to the variables on the stack. Then the sweep phase recycles any memory that isn't already marked as well as attempts to compact memory so that it is contiguous.

In education, research, and industry, use of garbage-collected languages such as Java[4] and ML[14] remains strong. However, despite many advances, the cost of automatic garbage collection continues to be prohibitive in some areas, notably embedded, real-time, and scientific applications.

- CPU cycles must be devoted to collecting the garbage. Incremental systems amortize the cost, and extra processors can hide the cost if those processors have nothing better to do.

- The need for collection can occur at unpredictable and inopportune times.

- Storage becomes fragmented unless objects are moved, but object relocation fools most underlying storage systems. For example, an object can be in cache, but known by its former address. Access of the object at the new address results in a fault followed by a fetch from slower storage.

- Exact garbage collectors mark live objects. While generational collection can limit such marking to a subset of a program's live objects, the marking phase pollutes the cache as the live objects are touched.

In this thesis, we propose and evaluate the performance of a new scheme, the *contaminated garbage (*CG*) collector*. This new collector has the following properties:

- It can operate in concert with a traditional collector, decreasing the frequency with which the traditional collector must be called.

- It does not require a "marking" phase, so that data caches remain valid even as objects are collected.

- It is incremental, doing constant work after every method finishes

- It collects a reasonable percentage (on average, 53%) of dead objects.

- It correctly identifies dead objects, but objects that it thinks are live may in fact be dead.

To elaborate on the last point, CG collection is "conservative", though not in the traditional sense of that term. Conservative collection has been proposed for languages (such as C)

in which reference variables cannot be precisely determined; such collectors are conservative because they may be forced to treat a value as a pointer [5]. The CG collector is conservative in a different way and for different reasons, as we explain in chapter 2.

1.1 Background

Wilson presents an excellent survey of storage allocation [23] and collection [22] techniques. All known methods for exact garbage collection require marking live objects to some extent. Generational collection limits the scope of the marking phase to a set of objects that are believed mostly to be dead.

One way of comparing our work is to examine how various approaches view the notion of a *generation*.

- Traditional generational collection[3] defines a generation by the longevity of its objects. This separates newer from older objects, so that garbage collection can concentrate on the newer (presumably shorter-lived) objects. More recently, it has been proposed to focus on other than the youngest generation [18].

- The train algorithm[17], discussed below, views objects not only in terms of their longevity, but also in terms of their interconnection. Objects that reference each other tend to be clustered in the same generation. This nicely accommodates cyclic data structures, as they become free at the same time.

- Our algorithm attempts to cluster objects, not in terms of their longevity, but in terms of their expected expiration. When they must die—not how long they have lived—is our key concern. We dynamically compute the time at which a cluster of objects must be dead, based on the references among the objects.

Our thesis organized as follows: Chapter 2 explains our approach using a simple example. Chapter 3 describes an implementation of a CG collector, along with complications

that arise from multiple threads and native code. Chapter 5 compares our approach with previous work. Chapter 4 presents experiments based on this implementation. Chapter 6 presents conclusions and ideas for future work in this area.

Chapter 2

Approach

Our idea is based on the following property of single-threaded programs (multiple threads are addressed in Chapter 3). Each object X in the heap is live due to references that ultimately begin in the program's runtime stack and static areas.[1] When the set of frames containing direct or indirect references to X is popped, then X is no longer live and it can be collected.

Moreover, owing to the nature of a stack, the set of frames that keep X live must contain some frame M that is last-to-be-popped (oldest) among the set's frames. The liveness of X can thus be tied to frame M: when frame M pops, X can be collected.

2.1 Example

We illustrate the CG collector using the example shown in Figure 2.1. The stack frames are shown numbered from 0 to 5; frame 5 is youngest frame, and frame 0 is not popped until the program finishes. Each frame corresponds to a method invocation, and the local variables for each method reside within the method's frame. The objects, labeled with letters A through F, reside in the heap. Arrows in Figure 2.1 depict the references from the

[1] We view static references as stemming from a program's initial stack frame.

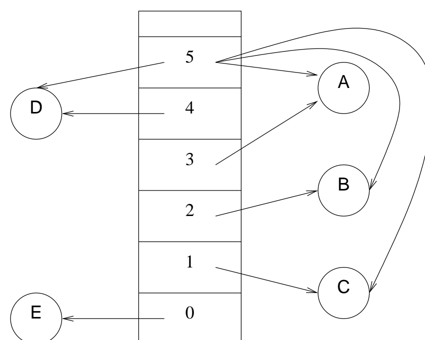

Figure 2.1: Frames that keep objects live.

methods' local variables to the heap objects. Though not shown in Figure 2.1, we assume each object X has a field x that is capable of referencing any other object. Also, we assume in this example that any method can access the program's static variables.

Given the frame references shown in Figure 2.1, the liveness of the objects is as follows.

Object	Referencing Frames	Earliest Frame
A	3, 5	3
B	2, 5	2
C	1, 5	1
D	4, 5	4
E	0	0

Although A is referenced by two frames, the object is live until frame 3 is popped. This illustrates an important property of our approach. With each object X, we associate a single frame M such that when M is popped, X is known to be dead—we then say that X's life *depends* on frame M, or that M is X's *dependent* frame.

As a special case, we associate frame 0 with objects that are referenced by *static* variables. Thus, CG collection determines that variables such as E in Figure 2.1 appear to

$$\begin{array}{ll} B.b \leftarrow A & \Leftarrow \boxed{1} \\ C.c \leftarrow B & \Leftarrow \boxed{2} \\ D.d \leftarrow C & \Leftarrow \boxed{3} \\ E.e \leftarrow D & \Leftarrow \boxed{4} \\ E.e \leftarrow \bot & \Leftarrow \boxed{5} \end{array}$$

Figure 2.2: Instructions affect object lifetime.

be live for the duration of the program. Frame 0 also serves to represent objects for which we (currently) cannot determine a dependent frame, as discussed in Chapter 3.

With the situation shown in Figure 2.1, it is clear that D could be collected when frame 4 pops. However, programs can cause one object to reference another, which has the effect of changing an object's dependent frame. We next examine the liveness of each object as the program shown in Figure 2.2 executes statements that cause one object to reference another. All of these statements are executed within Figure 2.1's frame 5— the frame of the currently active method. For our example, we assume this method has access to all objects as follows. Objects A through D are referenced using frame 5's parameters (locals in the JVM); object E is static and globally accessible.

The effects of the program's steps on the liveness of the heap objects are described as follows.

$\boxed{1}$ B now references A. With this reference established, A can be collected no earlier than B. Thus, A's dependent frame is changed from 3 to 2.

We say that B has *contaminated* A by touching (referencing) it.

$\boxed{2}$ C now contaminates B which still references A. Thus, the liveness of both B and A must be adjusted, so that they are now dependent on frame 1.

$\boxed{3}$ Although D now contaminates C, D depends on frame 4, which will be popped before C is dead. Thus, the dependent frames of A, B, and C are not changed—those objects all depend on frame 1.

However, D now has access to those objects. If D's liveness changed, then the liveness of those objects might also be affected. Our algorithm tracks such information efficiently, though conservatively.

4 Sure enough, E now contaminates D, which makes *all* the objects take on its liveness. Thus, all objects become dependent on frame 0.

5 Although E has contaminated D, E no longer references it. Ideally, this should revert the actual liveness of A–D to the situation after 4. For example, A can be collected when frame 1 pops.

In our approach, however, contamination cannot be undone. Once E contaminates the other variables (indirectly, by contaminating D), they become dependent on frame 0. Their dependence cannot be improved to a younger frame.

An extreme example of this is the "static finger of liveness". Suppose a static variable references *every* heap object. At each contamination, the affected object becomes dependent on frame 0, which isn't popped until the program finishes. As shown in Chapter 4, actual programs have better manners.

An unresolved issue from the above discussion concerns how to track the effects of a program's future behavior after 3. The problem is that D doesn't change any object's lifetime by referencing C. However, future changes to D's dependent frame may affect objects that can be referenced from D.

We accommodate this problem by asserting that contamination is symmetric, affecting *both* X and Y when X references Y. Thus, in the above example, D's dependent frame becomes synonymous with C's, so that future changes to D are correctly accommodated. Unfortunately, this conservatively makes D dependent on frame 1 after 3 executes.

2.2 Summary

In summary, the CG collector operates as follows.

- We maintain an *equilive* equivalence relation over a program's heap-allocated objects. Objects in the same block of the induced partition are viewed as having the same lifetime and are dependent on the same frame.

  Equilive sets grow through union operations; an equilive set's dependent frame can change as the program executes, but always by moving to an older frame.

- When a frame M pops, all equilive sets associated with M contain objects that must be dead. Such objects can be safely collected when M pops. If the objects are already in some kind of list L, then the objects can be returned to the available storage pool by joining L to the free-storage list. This can be accomplished with two storage accesses, which should not disrupt the effectiveness of the data cache.

- Two blocks $\mathcal{A}$ and $\mathcal{B}$ of the relation are merged (by a union operation) when objects $A \in \mathcal{A}$ and $B \in \mathcal{B}$ contaminate each other. This could happen because A references B, or because B references A.

  An exception to this policy occurs in an optimization described in Section 3.4.

- When a new block is formed by merging two existing blocks, the new block is dependent on the older (lower-numbered) of the existing blocks' dependent frames.

- The liveness of an object X, and therefore X's block, is affected if a method returns X to its caller. The liveness of X's block must be adjusted so that its dependent frame is popped no sooner than its caller's.

The reflexive and transitive aspects of equilive are accurate. However, the symmetric property introduces conservativeness, as illustrated with the example of D above.

Our approach is therefore conservative—though not because we can't tell what is a reference and what is not [5]. The CG collector may overestimate the lifetime of an object. For such objects, traditional garbage collection may collect the object when we would not. We therefore evaluate our approach in Chapter 4 by showing the percentage of objects that are collectable using CG.

Our approach does have the following advantages over traditional collection.

- Traditional collection requires marking live objects. While some generational collectors [22, 12] can limit themselves to marking a subset of the live objects, this phase of garbage collection pollutes the cache (and more distant virtual memory components) with objects that are not referenced actively by the running program [11].

- Maintaining the equilive relation can be accomplished efficiently if the disjoint sets of objects are maintained using Tarjan's *union by rank* and *path compression* heuristics [9]. The resulting overhead is a (nearly) constant amount of work per storage reference.

Chapter 3

Implementation

We implemented our approach in the context of Sun's Java system, Java Development Kit (JDK) 1.1.8. Our changes were confined to those portions of the Java Virtual Machine (JVM) [13] that deal with object creation, frame creation (in response to method calls), method return, and the base (traditional) garbage collection. Sun's 1.1.8 system offers the following JVM interpreters.[1]

- The reference interpreter is written entirely in C.
- A more efficient interpreter implements the most frequently executed portions in (Sparc) assembly language.

To facilitate our implementation, we based our work on the C version. However, the changes we made are compatible with the architecture of the (speedier) assembly version.

We next sketch our basic implementation and describe how we accommodate interpreter-generated static references and the more conceptually demanding characteristics of the JVM—namely, multiple threads and native code.

[1]This JVM does not provide Just-In-Time (JIT) compilation, though later versions do.

3.1 Data Structures and Modifications

Sun's JVM interpreter manages objects using *handles*. Each handle contains a pointer to the object's current location as well as a reference to an appropriate method table for (virtual) method-lookup. References between objects indirect through the handles. Thus, if objects are relocated (during garbage collection, for example), then only the handle's pointer to the object needs to be updated.

The interpreter offers a standard treatment of method-call and method-return. Each activation record is pushed onto a thread-specific stack [1].

To implement our approach, we modified Sun's JDK 1.1.8 system as described in the following sections.

3.1.1 Objects

We augmented each object handle with fields to accommodate union/find of the equilive sets. We also added fields to maintain the list of sets as well as for each set. We used a parent pointer and an integer rank for union/find. To traverse each equilive set, we maintain a next pointer as well as a last pointer, which points to the last handle in the equilive set. Each set is on a doubly linked list on a frame and has a pointer to the previous and next set. To allow for easy access to a handle's associated frame, we have also included a pointer back to its frame. To deal with multiple threads, we use a pointer to the thread where each allocation occured. Although not required for our implementation, we also added a unique integer ID as well as a birth depth (of the stack), so that we could track at what depth an object was allocated.

Union/Find on Disjoint Sets

Our approach uses union/find on disjoint sets to organize its data. The basic idea is to have two operations: `union` and `find`. The design of the algorithm assures that objects are

in exactly one set. Each set has an associated rank that is used to determine the parent set during the union operation. During a union, which ever set has the highest rank is chosen to be the parent, if both ranks are equal, a set is chosen. Path compression is used to improve the performance of the algorithm. Whenever a find operation is executed, if the parent of the set is not the root, find is recursively called and the result is stored as the parent of the current set. Every object that find is called on has its parent updated to be the root and we avoid degenerate structures. For more details on the algorithm and Tarjan's analysis, see [9]).

A straightforward implementation would require one "ancestor" field and one integer field to represent the rank. Of course, "primitive" objects (such as integers) do not use handles and thus do not incur any overhead.

A more clever representation can be achieved by noting that the lower bits of JVM object pointers are already reserved, and are therefore assumed to be zero. The equilive sets can then be maintained so that the rank never exceeds a predetermined threshold. Thus, the union/find algorithm can be implemented with one additional word per object handle.

Our approach requires the ability to determine any object's dependent frame. In a straightforward implementation, this can be achieved simply by introducing a pointer into the handle, such that the pointer references the the object's dependent frame. This pointer can be eliminated if each equilive set's representative element points to the dependent frame for the entire set.

In summary, the results reported in this thesis were obtained by introducing an additional eight words (thirty-two bytes) into what was formerly a two word (eight byte) handle. The implementation also has an additional six words that are used for other garbage collection schemes. As will be discussed in Section 3.5, the handle size can squeezed in half.

The original interpreter divided the heap up into two parts, one for handles and one for the objects. This division was 20% for handles and 80% for objects. To account for

our overhead, we adjust the space allocated for the handles so that it is eight times the original. This maintains the original object space (since the object size has not changed), but expands the handle space proportional to the space we added in the handle.

Arrays An array is treated as just another object—we do not differentiate an array's elements. Thus, any object stored into an array causes the array and the object to contaminate each other, as described later in this section.

3.1.2 Frames

When a frame is popped, the equilive objects that depend on the frame can be collected. Thus, each frame is equipped with a reference to a list of its dependent equilive blocks. We also gave each frame a unique ID number.

3.1.3 Static Variables

We maintain a list of objects that are dependent on our "frame 0". Such variables are never collected by our approach.

Essentially, the JVM interpreter must take action for those JVM instructions that cause one object to refer to another. The JVM instruction set conveniently separates these by whether the referencing object is static.

- When an object is created, it is associated with the frame of the currently active method.
- The `areturn` instruction causes a method to return an object to its caller. The object's equilive block is adjusted to depend on the caller's frame, unless the object is already dependent on an older frame.

- The `putfield` instruction causes object X to reference Y. If Y is not null, then X and Y contaminate each other, as described earlier.

  In the special case where Y is already static, the optimization described in Section 3.4 avoids contaminating X.

- The `putstatic` instruction can cause a static variable to reference an object. If so, the referenced object's equilive block is added to the list of frame-0 dependent blocks.

3.1.4 Tainted Objects

We also maintain a list of objects which we know to be dead. Keeping these objects on a list allows us to check future references and helps to assure the correctness of our algorithm. Anytime we find an object to be dead, we add it to this list.

We began with almost no familiarity of Sun's JVM interpreter. Nonetheless, it took only six weeks to implement our initial approach in that system. While this is a tribute to the interpreter's design, it also underscores the simplicity of our approach. Similarly, the code generator of a native-code compiler could easily be modified to emit the necessary code to maintain our structures. Subsequent modifications described in sections 3.6 and 3.7 have occured since the original implementation.

3.2 Interpreter-Generated Static References

For our approach to work, it must be able to take action when one object references another. For code written in Java, this requirement poses no problem. However, the interpreter can itself generate references to objects, and we had to integrate such references into our garbage collector.

A good example of this kind of problem is the `intern()` method of the `String` class. A program could generate multiple `String` objects, each with the same contents. The `intern()` method maps any `String` to a unique occurrence with its contents. Thus, given any two strings, equality of their contents can be tested using "==" once the strings are mapped using `intern()`. JDK 1.1.8 implements `intern()` using a hash table—internal to the interpreter—to maintain references to the unique occurrences of any `String` mapped via `intern()`. The references from the hash table are essentially static, since a `String` must map to the same reference via `intern()` for the duration of a program.

Because this activity is not part of the JVM instruction stream, we had to insert calls in the `String` class to tell our collector that any `String` mapped via `intern()` is static.

The class loader and JNI[2]-processing components were other sources of static references to the heap. Most likely, any implementation of JVM will maintain such references. To use our approach, these need to be identified and proper calls to our collector must be inserted.

3.3 Multiple Threads and Native Code

The discussion so far has been limited to single threads and Java-source programs. In this section, we describe our currently simple treatment of multiple threads and native code. More sophistication is possible, but that is a subject of future work.

Our assumption that an object is dependent for its life on a *single* stack frame does not hold if a program shares such an object among multiple threads, as shown in Figure 3.1. Within Thread 1, A is dependent on frame 3; however, Thread 2 can also access A until its frame 1 is popped.

[2] Java Native Interface

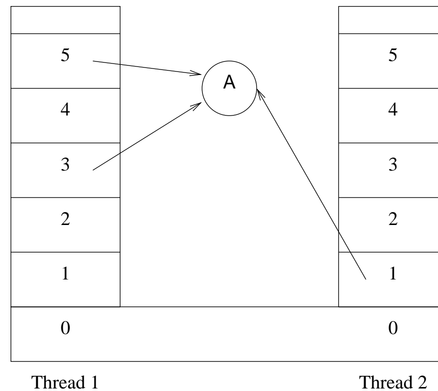

Figure 3.1: Two threads sharing an object.

For the purposes of this thesis, we dynamically discover objects that are accessed by multiple threads and we treat their equilive blocks as static—dependent on the program's frame 0.

Sun's JVM system allows native (e.g., C) code to be interspersed with Java code—each can call the other. A mechanism (object pinning) is already provided so native code can rely on an object's address. However, when C code calls Java methods, it is possible that objects are created and returned, perhaps briefly, to the native caller. To be conservative, we catch such allocations and treat the equilive blocks as if they were static.

3.4 A Static Optimization

While the approach described in Section 2 is correct, Plezbert [6] identified a situation for which we can offer a better treatment. Consider the results of the assignment

$$A.a \leftarrow S$$

where S is static—associated with the last-to-be-popped stack frame. As described in Chapter 2, our approach would union the equilive blocks containing A and S. As a result, A would also be regarded as static, existing for the lifetime of the program. However, in this case, such action is unnecessarily conservative. The object S is already determined not

to be collectable until the program is over. No further action can cause S to be regarded as more live than that. Thus, if S is believed to last for the duration of the program, there is no reason to join A's equilive bock with S's when A references S.

Ther results presented in Section 4 include this optimization, except for one column in Figure 4.1 which is designed to show the benefits of the optimization.

3.5 Shrinking the CG Handle Size

We also have another implementation in which we squeezed our sixteen word handle down to eight words. This was accomplished through the combining of the rank and parent structure. This can be done because of two factors. First, we know that, in general, the rank does not exceed ten (for the SPECjvm98 benchmarks) and that the handles are aligned based on their size. In our case, the handles are aligned on an eight word boundary, so we can guarantee that the bottom four bits will never be used for a handle address. The rank was stored in the parent pointer and we use a mask to set and restore it. This has the obvious benefit that it takes half as much memory for each handle as our previously discussed approach.

3.6 Resetting CG Structures During Traditional Garbage Collection

While CG works well, it is also useful to consider using it in addition to the traditional collector. If CG operates in concert with the traditional collector, it would be advantageous to have it reset our structures while running. The use of equilive sets makes our method more conservative, and since the traditional collector starts with the roots of computation and follows all the references to other objects, we could take advantage of this time to

update our information. To implement this resetting, we need to modify the collection phase. The collector starts with each stack frame and marks each object that's live and then any object it references. Before each stack frame is considered, we remove all the equilive sets from the frame. As each object is considered by the traditional collector, we reset it to be associated with the current frame. Each object is then searched to find any objects that it points to. At this point, we union the objects' sets together and have our updated information. The benefits of this approach are that, ideally, after each call to the collector, CG would have the same liveness for all objects. Unfortunately, due to the additive nature of our approach, this is only approximately true. The traditional collector also benefits because we are able to free more objects and the collector is called less often.

3.7 Recycling of CG Objects

Another possible improvement on Chapter 2 deals with what we do with the objects once they have been identified as dead. Upon each method return, we have a list of equilive sets of objects which we know to be dead. Normally we iterate through this list, visiting each object and freeing it. A better approach would be to defer the freeing of the object and recycle these objects so that they can be used during successive allocations.

To accomplish this task, we disconnect the list of equilive sets from the frame after it is popped and prepend it to a list of recycled objects. Then, during allocation, we traverse this list and do a "first-fit" search for an object of the correct size. Now, instead of having to free each object in every equilive set after a method return, we only update a pointer. This lowers CG overhead by deferring the "freeing" until allocation. Allocation from our free list happens when the allocator fails to allocate an object, before it tries to run MSA.

We hypothesized that this optimization would be beneficial in several ways. We have discovered that, in general, most of the objects in Java programs are of the same size (16 bytes). The default allocator in JDK 1.1.8 does a linear search through the object pool

to find the first object that is at least as big as requested (and also tries to coalesce two contiguous objects to make a block big enough). We can guarantee that every time the JVM (during allocation) looks at our list of dead objects, it will be looking at a free object. The allocator keeps track of the last location where it allocated an object from, so it would not seem to be much of an improvement while the heap has space available. Once the heap has filled (or the first attempt at allocation fails) the allocator has to rescan the heap to find free object and recycling would seem most useful.

Chapter 4

Experiments

We implemented our approach as described in Section 3.4. We then conducted experiments on the approach using the programs described in Figure 4.1. We used the SPEC benchmark suite [10]. The suite has eight different benchmarks that implement various tests that can be run for different sizes (1, 10, 100). [1]

4.1 Collectable Objects

For each benchmark, Figure 4.1 shows the number of objects created during its run. The right two columns show the percentage of all objects that were collected by our method. The rightmost column shows the percentage of collectable objects when the optimization described in Section 3.4 is enabled; this is of course the preferred implementation. For comparison purposes, we also show the percentage of objects collectable without the optimization. All other objects were treated by our method as *static*—live until the end of the program. Given our approach, such objects are either declared static or else they are referenced indirectly by a static object.

[1]The `mtrt` program is a multithreaded version of `raytrace`; however, multiple threads are required for computation only for the larger problem sizes. Thus, our results for these two programs are very similar.

benchmark	description	lines of source	objects created	collectable no opt	collectable with opt
compress	Modified Lempel-Ziv	920	5123	9%	11%
jess	Expert System	570	45867	35%	61%
raytrace	Ray Tracer	3750	276960	98%	98%
db	Database Manager	1020	7608	18%	36%
javac	Java Compiler	9485	26116	23%	24%
mpegaudio	MPEG-3 decompressor	N/A	7550	6%	7%
mtrt	Ray Tracer, threaded	3750	276084	98%	98%
jack	PCCTS tool	N/A	393742	69%	89%

Figure 4.1: Percentage of objects collectable by our approach, without and with the optimization described in Section 3.4.

The ray-tracing, path-navigating, and `jack` programs were over 90% collectable using the CG collector. The `mpegaudio` and `compress` programs do not generate many objects; the objects that are generated are fairly long-lived. Thus, we did not collect much for those programs, but neither would an exact approach. For the other benchmarks, we are from 7%–60% successful. Although those numbers may seem low, even if we are only 50% successful, this means that the traditional collector would be called half as often as without our approach.

4.2 Static Objects

Objects that we believe to be static live for the duration of the program. One metric to measure the effectiveness of our approach is the number of static objects versus the total number of collectable objects. Figure 4.2 shows that, for small runs (size 1), `compress`, `db`, and `mpegaudio` have a large number of static objects. We observe that `jess` and `javac` have about the same number of static as collectable objects. Otherwise, the number of static objects is relatively small. For larger sizes (10 and 100), Figure 4.3 and Figure 4.4, show significant improvement with the exception of `compress` and `mpegaudio`, which

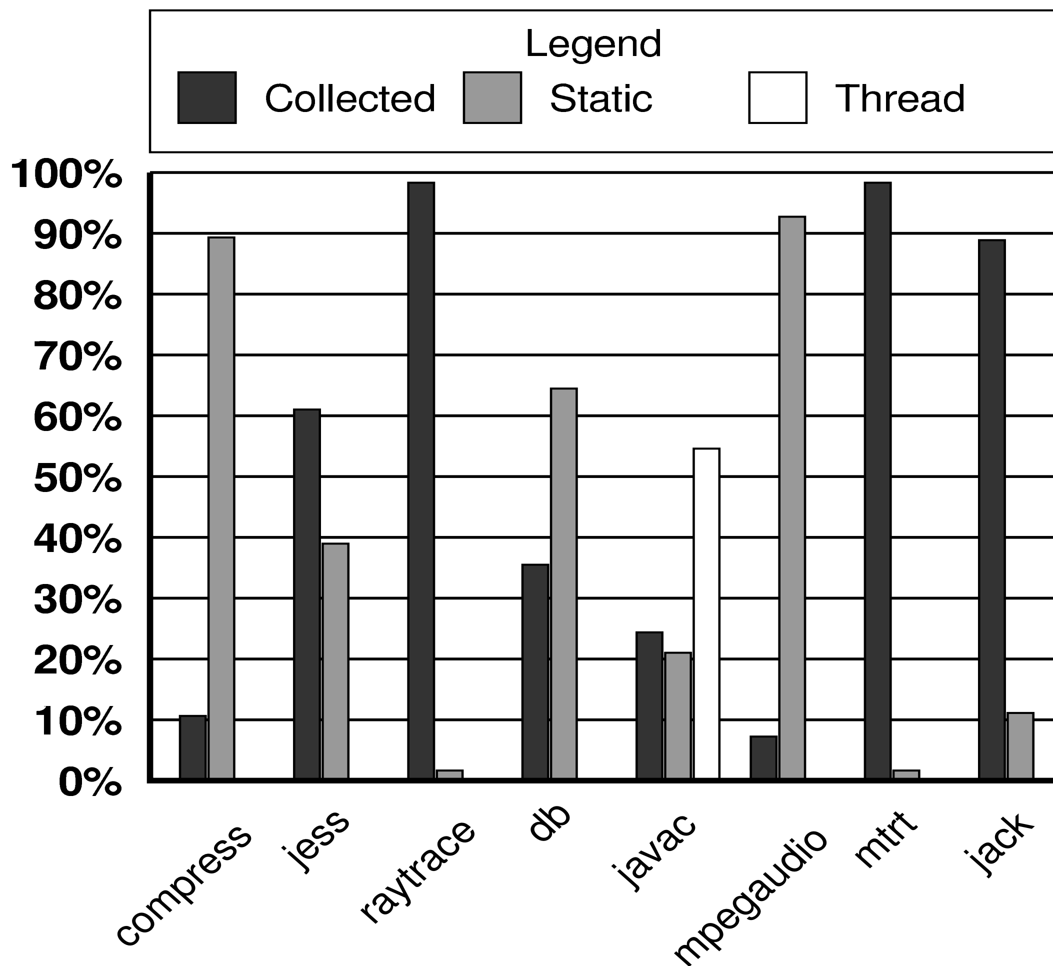

Figure 4.2: Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 1).

allocate only a few objects and do mostly compuatartion. For the other benchmarks, the number of static objects increases by a small amount and the number of collectable objects shows a dramatic increase. These results lead us to believe that our approach would be useful in longer-running benchmarks and applications. Servers and web based servlets are examples of such programs that might benefit from our approach.

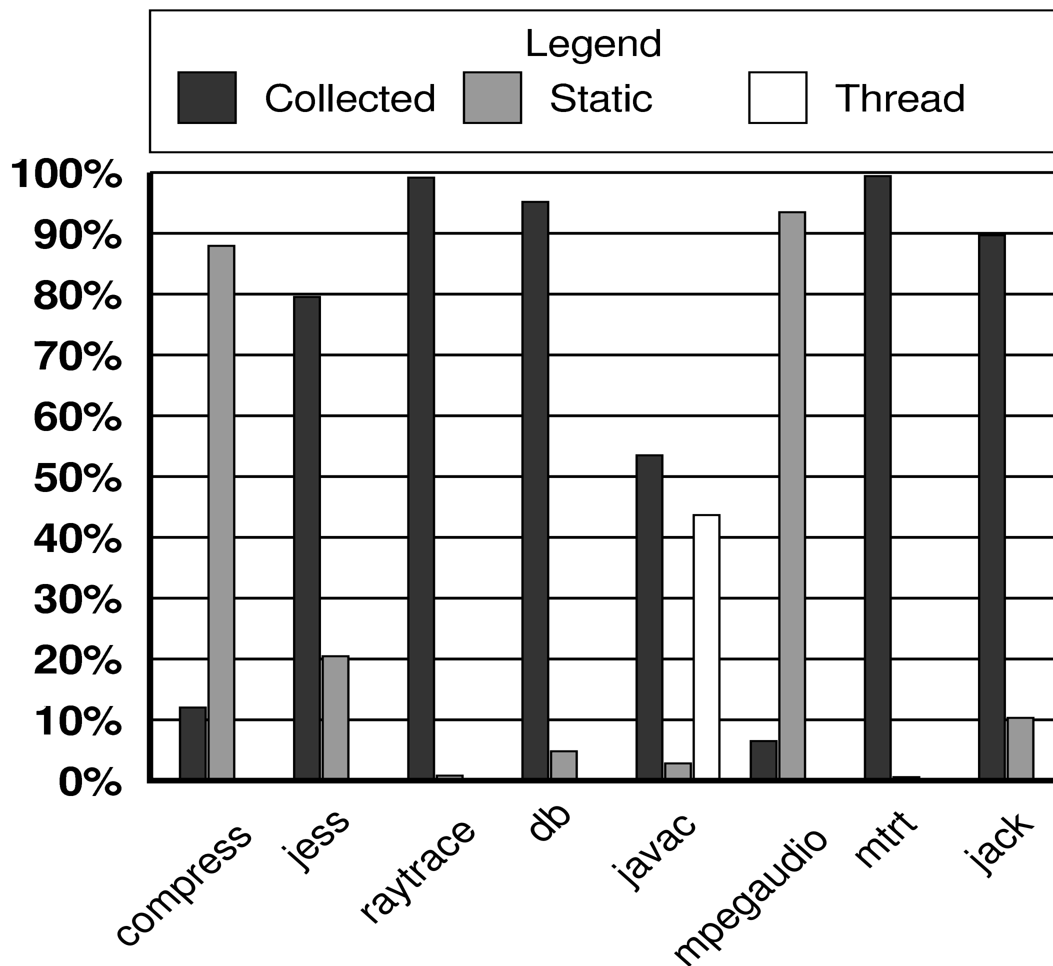

Figure 4.3: Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 10).

4.3 Thread Behavior

Because we treat multiple threads conservatively, we measured the number of objects that were forced into the static set when they were accessed by multiple threads. Recall that objects in the static set are treated by our approach as live for the program's duration. Figure 4.2 shows that most of our benchmarks had very few thread-shared objects. The `mtrt` and `raytrace` programs are equipped to run multithreaded, but showed only a very small percentage of their objects being shared across multiple threads. The `javac`

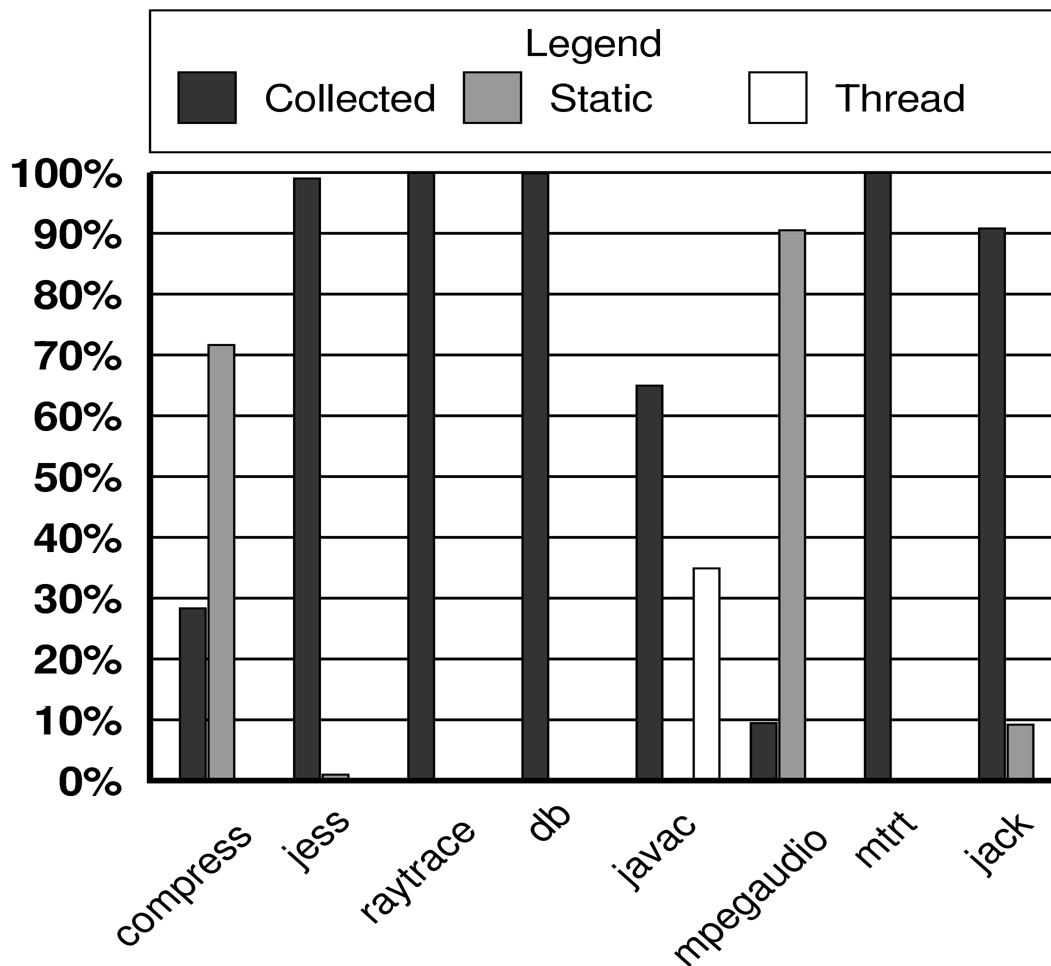

Figure 4.4: Percentage of objects that we treat as static (live for the program's duration) and due to sharing among threads (size 100).

benchmark had the largest number of thread-shared objects (over 72% of the total objects and more than two times the collectable or static objects)). As the size of the benchmarks increase (to 10 and 100), we see improved effectiveness of our approach, as shown in Figure 4.3 and Figure 4.4. In a similar fashion to static objects, the relative number of thread-shared objects increases slowly, while the number of collectable objects increases quickly. The larger runs show `javac` having almost twice as many collectable objects as thread-shared objects.

benchmark	total collectable	number of blocks of size 1	2	3	4	5	6–10	>10	percent exact
compress	5123	176	65	31	7	2	0	2	3%
jess	45867	3050	7193	3156	20	68	43	18	7%
raytrace	276960	40415	9446	1503	1834	3	4277	2	15%
db	7608	330	93	319	5	2	0	3	4%
javac	26111	2792	526	337	177	142	1	1	11%
mpegaudio	7550	177	63	37	9	2	0	1	2%
mtrt	276084	40290	9321	1474	1800	2	4277	2	15%
jack	393742	119252	85418	13515	4720	30	26	1	30%

Figure 4.5: Distribution of block sizes.

4.4 Size and Age of the Equilive Blocks

Recall that blocks containing objects A and B are merged when A references B (or B references A). For the following reasons, we were curious about the number of objects that accrue in each block prior to the block's collection using CG.

- Blocks that contain a single object are exact: no unions are performed and so we can return such objects at the next method-return.
- If most blocks are size 1, then an approach that looks only for such blocks might work well without the overhead of our more general approach.
- Recalling our example from Section 2, we were forced to overestimate D's lifetime when it was merged with C. Our approach could be improved by keeping track of dependent frames per-object instead of per-block. However, this would be unreasonable if there were many objects per block.

Figure 4.5 shows the size of the collectable blocks created during the runs of our benchmarks. Although most blocks contain more than one object, the majority of blocks do contain three or fewer objects.

benchmark	Distance from birth to death frames						
	0	1	2	3	4	5	> 5
compress	146	151	140	69	29	10	0
jess	5,824	8,926	12,272	640	249	79	1
raytrace	35,707	36,590	29,183	1,050	11,285	6,368	152,133
db	180	613	1,200	554	30	74	50
javac	3,445	1,477	930	203	305	4	2
mpegaudio	146	163	153	58	24	3	0
mtrt	35,550	36,526	28,990	861	11,221	6,272	152,036
jack	63,230	263,574	20,992	1,961	168	7	4

Figure 4.6: Age at death of objects we collect.

Next, we measured the distance to die for objects that we were able to collect. Suppose an object X is born in frame M. When X is finally collected, it must depend on a frame at least as old as M. The singleton blocks mentioned earlier—for which our information is exact—may not die in their allocating frame, because a frame can return a result to its caller. Figure 4.6 shows the age, in frame distance, of objects when they die.

Objects that are collected in the 0 column never escape the frame in which they were allocated. Many collectable objects fall into that category. However, most are associated with older frames. For the `javac` benchmark, a significant portion of objects allocated in a frame are detected collectable when that frame's caller returns.

For those objects that die in their birth frames, it may be worth considering how such objects could be collected sooner than their dependent frame pops. The singleton sets can be collected once it can be shown that no local variable references the object. As described in Section 1.1, static approaches may serve well here. It is interesting to note static escape analysis [21] is practically limited to analysis of two frames, while our approach can detect an arbitrary number of frames.[2]

[2]Personal communication with Martin Rinard

benchmark	CG	JDK	speedup
compress	318.9908	292.6376	0.92
jess	5.7176	5.1144	0.89
raytrace	35.217	27.8904	0.79
db	0.692	0.6558	0.95
javac	3.335	3.7172	1.11
mpegaudio	34.3276	33.3924	0.97
jack	70.6476	64.2296	0.91

Figure 4.7: Timing results. The rightmost column shows the speedup of over traditional collector in the JDK 1.1.8 system for size 1.

4.5 Performance and Overhead

Finally, we examine the run-time overhead of our approach in Figure 4.7. We began with Sun's JDK 1.1.8 (call this the *base* system) and modified it to use our CG algorithm. All tests were run on a Sun Ultra Sparc 5 workstation with 128MB of real memory and 1.6GB of virtual memory. The processor is a UltraSparc-IIi running at 400MHz. The rightmost column of Figure 4.7 shows the speedup obtained by CG. Recall that our approach incurs overhead for maintaining the equilive blocks. Also, action is taken at each `store` and `return` operation. The base system does not incur such overhead, but does pause to garbage collect when its heap becomes relatively full.

The rightmost column shows at best, an 11% improvement in execution time using CG. This represents an absolute savings of time using our approach over the base system, even though we perform extra work at every `store` operation. Thus, the savings can be attributed to avoidance of the traditional garbage collector. Moreover, we set up the runs to avoid heap compaction. Thus, the savings stems primarily from avoiding the marking phase of garbage collection. In general, though we do show a slowdown, we are within 10%–20% of the base system's execution time.

To isolate the overhead of maintaining the equilive sets, we ran the base system with the asynchronous GC disabled as well as giving is plenty of storage. This allowed the

benchmark	CG	JDK	speedup
compress	372.7318	346.6596	0.93
jess	54.4978	49.4348	0.91
raytrace	97.9964	78.0694	0.80
db	43.1086	39.3572	0.91
javac	31.9388	29.3018	0.92
mpegaudio	354.9622	345.6264	0.97
jack	140.8294	128.9202	0.92

Figure 4.8: Timing results. The rightmost column shows the speedup of over traditional collector in the JDK 1.1.8 system for size 10.

benchmarks to run without running the MSA collector. Thus, the middle column of numbers in Figure 4.7 and Figure 4.8 shows the speedup (typically slowdown) of our approach over the base system for sizes 1 and 10, repectively. In general, we are within 10% of the the base system.

4.6 Larger SPEC Runs

We next examined the performance of our approach on the "larger" SPEC benchmarks. These are really the same programs used previously, but with longer running times. As shown in Figure 4.9, most of the benchmarks generated substantially more objects. The exceptions to this are `compress` and `mpegaudio`, which are computational in nature. Interestingly, our approach worked only better in terms of the percentage of collectable objects. Notably, `db` and `javac` went from 41% and 24% collectable in the small run to 91% and 99% collectable in the large run. Similarly, the number of objects that we can collect exactly mostly improved in the large runs, except for `db`.

Name	Objects Created	Collectable With opt	Exactly Collectable
compress	6,959	28%	27%
jess	7,924,661	41%	42%
raytrace	6,346,978	99%	82%
db	3,211,531	99%	0%
javac	5,879,703	91%	12%
mpegaudio	7,582	9%	30%
mtrt	6,585,974	99%	80%
jack	6,863,344	90%	37%

Figure 4.9: SPEC benchmarks, large runs.

benchmark	size 1	size 10	size 100
compress	0.97	0.97	0.98
jess	0.93	0.96	3.18
raytrace	0.87	0.85	1.71
db	0.95	0.94	0.94
javac	1.14	0.96	2.77
mpegaudio	1.00	1.00	1.30
jack	0.93	0.94	1.98

Figure 4.10: Speedup of our approach over JDK 1.1.8. For the large run, `mpegaudio` and `compress` took over an hour to complete with either system.

Finally, we compare execution times for the SPEC benchmarks in Figure 4.10. The "small" speedups are reprised from Figure 4.7; included also are the speedups (and slowdowns) of our method for the medium- and large-scale runs of the benchmarks. We show a slight improvement in the size 10 runs and a significant jump in size 100.

Our approach worked well for the small runs, and it should be noted that even the "small" runs take substantial time.

4.7 Resetting Results

In Section 3.6 we describe how a normal MSA pass for garbage collection can reset object information for CG. In this section, we present results from resetting CG structures, showing the effects of resetting on the quality of object collection under CG.

Suppose an object is actually dead at time t. That object's collection under CG falls generally into one of the following three categories:

1. Within a boundable amount of time after t, a frame pop occurs and CG collects the object. This is the best case for our approach.

2. The object is collected in the same frame as above, but the method spends an unboundable amount of time prior to the frame pop. Since we associate the liveness of objects with frames, we only can only tell if objects are dead when a frame pops. If an object becomes dead during a method, our approach will not discover it dead until the next frame pop. For exmaple, the method may contain a loop that prevents us from collecting the object expeditiously.

3. In the above two cases, CG is accurate to the next frame pop. That is, if an object dies while frame f is active, then we collect the object when frame f pops. Thus, the last case for our method is when CG associates the object with a frame longer-lived than f. For example, the object could be associated with the static set. This happens most often when a static object touches another object and then points away. Our approach would be to put the object being referenced into the static set and it would live "forever". The optimization metioned in Section 3.4 helps to avoid this situation.

During the mark phase of MSA, we verify and update our CG structures as described in section 3.6. The net effect of such an update is to correct the approximation errors introduced by our approach.

name	collected by MSA	less live	GC cycles
compress	227	1	9554
jess	13210	41	90
raytrace	232003	13	540
db	2258	1	24
javac	10359	1	66
mpegaudio	206	1	1165
mtrt	231654	14	531
jack	38215	2	1005

Figure 4.11: SPEC benchmarks, small runs.

We instrumented the JVM to run garbage collection after a certain number of instructions had been executed. For these results, we ran MSA every 100,000 JVM instructions. Figure 4.11 shows the results we found. The “collected by” column shows that most objects were determined to be unreachable and therefore dropped out of our structures and were collected by the sweep phase of MSA. A small number of objects were determined to be “less live” than our static set. Those objects that did move from the static set only moved a few frames. The nonstatic objects showed no movement between frames at all. Experimentation with larger sizes showed no improvement and those results are not shown.

4.8 Recycling Results

Figure 4.12 shows that the benefits of recycling objects are almost as good as predicted. In general we are within 4% of the original timings, with speedups happening more often than not. `compress` is the best performer, with a 3% increase. Figure 4.13 shows the number of objects that we recycle versus the total number of objects allocated. The `compress`, `db`, and `mpegaudio` benchmarks all show a small number of objects recycled, while the other benchmarks show 10% to 60% of objects being recycled. In the benchmarks that recycled a large number of objects, we see a smaller speedup. Since the JVM allocator

Name	CG time seconds	CG with recycling	speedup using recycling
compress	311.53	303.38	1.03
jess	7.50	7.57	0.99
raytrace	43.27	44.60	0.97
db	0.89	0.88	1.01
javac	4.40	4.42	0.99
mpegaudio	35.22	34.60	1.02
mtrt	45.15	44.36	1.02
jack	171.69	171.40	1.00

Figure 4.12: Recycle timing, small runs.

Name	objects recycled	percent of total
compress	308	6.01
jess	13728	29.93
raytrace	32175	11.62
db	702	9.23
javac	5701	21.83
mpegaudio	313	4.15
mtrt	31432	11.38
jack	222344	56.47

Figure 4.13: Number of objects recycled, small runs.

progresses sequentially through the heap and remembers the location of its last allocation, it can quickly find the next free object. This style of allocation only works the first time through the heap. With longer running benchmarks, the heap will become full and the allocator will be forced to start its search at the beginning of the heap. Searching becomes more difficult because it has to find free space among the objects.

Our list of equilive sets is not ordered and we have to search through each set (doing a first fit) every time, leading to a worst case of $O(n)$ for n objects in our recycle list. Note that because of first fit, we expect to do better than the average case of $O(n/2)$, as we only have to find an object at least as big as requested. This could be further improved by

keeping the objects sorted on the equilive sets, but that would make our CG approach even slower. Another possibility would be to keep the sets organized by type, so that we could merely look for a specific type of object, and reset its structures.

Chapter 5

Generally Related Work

Appel [2] has observed that stack-allocated storage (i.e., local variables) can be managed more efficiently using the (more general) heap. Instead of reclaiming each frame individually upon its method's return, multiple frames are collected when garbage collection transpires. In summary, Appel proposes to treat stack-allocated objects as heap-allocated. We are essentially trying the dual of that approach: we model heap-allocated objects as if they were allocated in a stack frame, but we continually revise *which* stack frame holds a heap-allocated object.

Static analysis techniques [7, 16, 24] attempt to determine the lifetime of objects, by finding environments from which such objects cannot escape. The representation for such environments can be a stack frame [15], so that objects are directly associated with a "deeper" stack frame than the method in which they are instantiated.

Also, the notion of an environment-escape has been generalized to that of a *region* [20, 19]. Regions are perhaps the closest in nature to the ideas expressed in this thesis. As with our approach, regions can decrease the need for mark-based garbage collection. A region essentially introduces a stack-based pair of allocation and deallocation sites for an object, where the sites are determined by static analysis and not by a program's syntax.

The distinguishing feature between regions and our work is that regions are determined statically, while our approach operates dynamically.

It is not clear that regions are better or worse than our approach.

- Our approach continually enlarges the "region" associated with an object, when the object is referenced by objects with longer lifetimes. For example, the instruction sequence shown in Figure 2.2 leaves all objects dependent on frame 0 in our approach. Static analysis (such as proposed in the "regions" work) could easily show that A could be collected when frame 1 pops.

- Because static methods must accommodate any path through a program, it is possible that our approach can fare better because it adjusts the expected expiration of objects dynamically, as determined by actual execution paths in a program. Thus, we might determine that an object can be released at a point prior to that which static analysis can show that the object is free.

The integration of our method with static approaches is the subject of future work.

5.1 The Train Algorithm

Our approach is influenced by the train algorithm [12, 17]. That algorithm continually reorganizes the heap so that objects that reference each other are clustered at the time that such objects are dead. In the jargon of the train algorithm [12], our approach can be expressed as follows. Each stack frame is associated with a train. When the stack frame is popped, all cars of the frame's train are known to be free, so we simply return those objects to the heap. The train algorithm moves objects between cars of trains during garbage collection, with the goal of clustering objects that reference each other. Instead of moving individual objects, our approach essentially joins two trains, leaving them attached to the appropriate stack frame. We are less precise than the train algorithm, because we deal with

objects only in terms of their containing trains. Also, once trains are joined, we do not consider separating them unless we have resetting of our structures enabled, as described in section 3.6.

The train algorithm is more precise, but—like all generational approaches—it requires keeping track of certain kinds of references. In summary, our approach does not supplant the train algorithm. Both approaches are incremental: objects that are dead may go uncollected for some time. Our approach avoids marking, and storage is returned as method frames are popped. The integration of our method with the train algorithm is the subject of future work, as discussed in Chapter 6.

Chapter 6

Conclusion and Future Work

We have presented a simple but conservative approach for tracking an object's dependent frame. Our experiments show the following.

- A reasonable percentage of objects are collectable by our approach (Figure 4.1 and Figure 4.9).
- Of those objects that are CG-collectable, most occur in blocks with three or fewer objects (Figure 4.5).
- For some programs (such as `javac` and `jack`), most objects that we can collect are collected within one or two frames of their birth (Figure 4.6). For other programs (such as `raytrace` and `mpegaudio`), a majority of objects are collected more than 5 frames past their birth frame.
- Although our approach performs reasonably well for the small runs of the SPEC benchmarks, we see improvements on the more practical sizes of 10 and 100.

In response to these observations, our plans for the future include the following.

To gain better insight into when and how well objects *can* be collected, we plan to identify the point at which

- an object becomes collectable
- traditional (exact) garbage collection collects it
- CG collects it

While it appears that a large number of objects can be reclaimed efficiently by our approach, our results suggest the following possibilities for future work.

- The operations needed to maintain the equilive sets are sufficiently simple that they might be incorporated directly into a storage architecture.
- The equilive singleton sets could be maintained "by type". Thus, when a frame is popped, there would be a collection of free objects of a given type. Instead of returning such objects to a general free-storage pool, they could be *recycled* the next time objects of that type are needed. For languages like Java, where objects of a given type always take the same size (except for arrays), such object recycling could have a big payoff.

  Moreover, this could improve the reference locality of a program. Others [8, 11] have suggested using garbage collection as a time to reorganize (live) storage to improve locality. If CG can recycle the dead storage, then the next instantiation of an object type may have its data already in cache.
- On its own, our approach never improves the dependent frame of an equilive block. However, it may be possible that such information could be reset when traditional collection is performed. Such fresh starts may give our approach more latitude in finding dead objects.
- Because many objects appear to be collectable when their birth frame pops, it is worth considering how such objects could be collected sooner. In particular, an object in a

size-1 set can be collected once its dependent frame no longer references the object. This could happen well before the executing method's frame pops.

Static analysis [7, 19] may help determine where such variables die. Also, it is possible that an efficient dynamic scheme could detect that such variables are dead.

- Static analysis might also help by determining the conditional liveness of objects. If object X can be shown to be as live as object Y, and we can tell that X is dead, then Y must also be dead.

- Our treatment of thread-shared objects is to consider them live for the program's duration. Instead, a set of dependent stack frames could be associated with an equilive block. Further investigation is needed to explore the expense and benefits of a more general approach.

- Our approach could compliment the train algorithm by collecting objects when methods return. Exact collection might be required less frequently. Also, the train algorithm could update our structures when it does run, sharpening the effectiveness of our approach.

Appendix A

Data

benchmark	Total num of static objects	Percentage of static objects due to threads
compress	4578	0%
jess	17876	0%
raytrace	4644	1%
db	4907	0%
javac	19745	72%
mpegaudio	7003	0%
mtrt	4628	1%
jack	43806	0%

Figure A.1: Percentage of objects that we treat as static (live for the program's duration) due to sharing among threads.

benchmark	popped	static	thread
compress	545	4576	2
jess	27991	17874	2
raytrace	272316	4599	45
db	2701	4905	2
javac	6366	5490	14255
mpegaudio	547	7001	2
mtrt	271456	4583	45
jack	349936	43804	2

Figure A.2: Object breakdown, small runs.

benchmark	popped	static	thread
compress	629	4602	2
jess	84723	21788	2
raytrace	554204	4599	99
db	116123	5889	2
javac	112789	5972	92060
mpegaudio	591	8493	2
mtrt	794189	4585	116
jack	699794	80525	2

Figure A.3: Object breakdown, medium runs.

benchmark	popped	static	thread
compress	1862	4708	2
jess	7846779	77882	2
raytrace	6342111	4455	414
db	3206483	5048	2
javac	3806149	8464	2045161
mpegaudio	718	6864	2
mtrt	6582100	4445	431
jack	6232144	631184	2

Figure A.4: Object breakdown, large runs.

Benchmark	CG	JDK
compress	319.022	292.779
compress	319.515	292.628
compress	318.993	292.277
compress	318.564	292.493
compress	318.86	293.011
jess	5.76	5.114
jess	5.726	5.072
jess	5.706	5.118
jess	5.692	5.062
jess	5.704	5.206
raytrace	33.726	27.639
raytrace	41.12	27.729
raytrace	33.71	27.686
raytrace	33.82	28.461
raytrace	33.709	27.937
db	0.688	0.636
db	0.689	0.639
db	0.689	0.637
db	0.699	0.664
db	0.695	0.703
javac	3.326	3.609
javac	3.333	3.605
javac	3.33	3.64
javac	3.336	3.619
javac	3.35	4.113
mpegaudio	34.018	33.284
mpegaudio	34.393	33.343
mpegaudio	34.426	33.361
mpegaudio	34.028	33.5
mpegaudio	34.773	33.474
jack	70.554	64.187
jack	70.729	64.086
jack	70.493	64.203
jack	70.866	64.222
jack	70.596	64.45

Figure A.5: SPEC benchmarks, small runs.

Benchmark	CG	JDK
compress	372.657	346.708
compress	372.731	347.075
compress	372.729	346.671
compress	372.701	346.164
compress	372.841	346.68
jess	54.5	49.725
jess	54.431	49.759
jess	54.575	49.103
jess	54.433	49.678
jess	54.55	48.909
raytrace	99.598	78.017
raytrace	97.524	77.937
raytrace	97.706	78.198
raytrace	97.574	77.816
raytrace	97.58	78.379
db	42.925	39.352
db	42.916	39.338
db	43.571	39.353
db	43.166	39.363
db	42.965	39.38
javac	31.794	29.454
javac	31.648	29.185
javac	31.807	29.208
javac	32.673	29.16
javac	31.772	29.502
mpegaudio	355.008	345.737
mpegaudio	352.475	346.363
mpegaudio	353.407	345.186
mpegaudio	361.457	345.33
mpegaudio	352.464	345.516
jack	140.567	128.599
jack	140.969	128.852
jack	140.979	128.719
jack	140.946	129.304
jack	140.686	129.127

Figure A.6: SPEC benchmarks, medium runs.

Benchmark	CG	JDK
compress	5536.811	5444.246
compress	5546.523	5458.166
compress	5540.937	5460.112
compress	5544.716	5461.799
compress	5529.341	5456.450
jess	1107.007	3668.475
jess	1268.469	3628.811
jess	1135.807	3750.79
jess	1114.949	3635.839
jess	1113.318	3563.147
raytrace	1425.78	2321.214
raytrace	1394.654	2339.595
raytrace	1417.778	2340.03
raytrace	1385.052	2618.47
raytrace	1354.019	2344.222
db	3227.322	3043.220
db	3231.903	3039.893
db	3237.756	3055.134
db	3229.218	3057.607
db	3228.809	3042.432
javac	1755.543	4949.161
javac	1776.483	4990.91
javac	1799.675	4931.495
javac	1835.242	5006.536
javac	1799.431	4931.756
mpegaudio	3600.563	3914.408
mpegaudio	3599.813	3873.521
mpegaudio	3606.237	3701.674
mpegaudio	3597.167	8363.869
mpegaudio	3683.987	3654.349
jack	1323.306	2771.095
jack	1338.635	2761.07
jack	1471.035	2700.231
jack	1333.17	2669.262
jack	1343.598	2610.093

Figure A.7: SPEC benchmarks, large runs.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] Andrew Appel. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[3] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[4] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, Boston, 2000.

[5] Hans-Juergen Boehm. Space efficient conservative garbage collection. *SIGPLAN Notices*, 28(6):197–206, June 1993. *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*.

[6] Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron. Contaminated garbage collection. *Programming Language Design and Implementation*, pages 264–273, 2000.

[7] D. R. Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Dept. of Computer Sci., Rice U., Houston, TX, August 1987.

[8] Trishul Chilimbi and James Larus. Using generational garbage collection to implement cache-conscious data placement. *Proceedings of the International Symposium on Memory Management*, 1998.

[9] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

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

[11] Scott Haug. Automatic storage optimization via garbage collection. Master's thesis, Washington University, 1999.

[12] Richard L. Hudson, Ron Morrison, J. Eliot B. Moss, and David S. Munro. Garbage collecting the world: One car at a time. In *OOPSLA'97 ACM Conference on Object-Oriented Systems, Languages and Applications — Twelth Annual Conference*, volume 32(10) of *ACM SIGPLAN Notices*. ACM Press, October 1997.

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

[14] Robin Milner, Mads Tofte, and Robert W. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[15] Alastair Reid, John McCorquodale, Jason Baker, Wilson Hsieh, and Joseph Zachary. The need for predictable gc. *Proceedings of the Second Workshop on Compiler Support for System Software*, 1999.

[16] Cristina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, California, January 1988.

[17] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. *Proceedings of ECOOP '95*, pages 235–252, 1995.

[18] Darko Stefanovic. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, Amherst, 1999.

[19] Mads Tofte. A brief introduction to regions. *Proceedings of the International Symposium on Memory Management (ISMM)*, pages 186–195, 1998.

[20] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[21] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. In *Proceedings of the SIGPLAN '01 Conference on Program Language Design and Implementation*, Snowbird, Utah, June 2001.

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

[23] 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.

[24] Kwang Keun Yi and Williams Ludwell Harrison. Interprocedural data flow analysis for compile-time memory management. Technical Report CSRD 1244, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Urbana, IL 61801, USA, August 1992.

Vita

Dante John Cannarozzi

Date of Birth	May 11, 1979
Degrees	B.S. 2001, from Washington University, St. Louis, Missouri.
Professional Societies	Association for Computing Machinery
Publications	Dante J. Cannarozzi, Michael P. Plezbert, Ron K. Cytron. "Contaminated Garbage Collection" *Programming Language Design and Implementation*, Vancouver, Canada, 2000.

May 2003