Contaminated Garbage Collection [*]

Dante J. Cannarozzi, Michael P. Plezbert, and Ron K. Cytron
Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

Abstract

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^{TM} Virtual Machine interpreter, and results are presented based on this implementation.

1 Introduction

In education, research, and industry, use of garbage-collected languages such as Java and ML 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 storage systems. 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.
- Traditional 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.

[*]Sponsored by the National Science Foundation, under grant NCR 9628218; contact author cytron@cs.WUSTL.edu

In this paper, 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 collects a reasonable percentage 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 [3]. The CG collector is conservative in a different way and for different reasons, as we explain shortly.

Our paper is organized as follows: Section 2 explains our approach using a simple example. Section 3 describes an implementation of a CG collector, along with complications that arise from multiple threads and native code. Section 4 compares our approach with previous work. Section 5 presents experiments based on this implementation. Section 6 presents conclusions and ideas for future work in this area.

2 Approach

Our idea is based on the following property of single-threaded programs (multiple threads are addressed in Section 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.

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

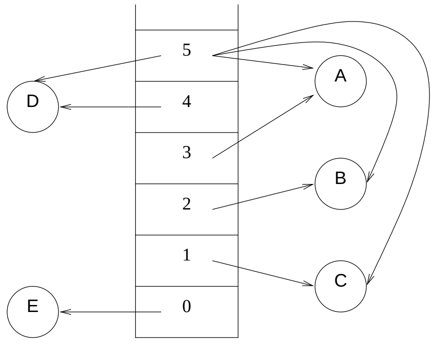

Figure 1: Frames that keep objects live.

2.1 Example

We illustrate the CG collector using the example shown in Figure 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 1 depict the references from the methods' local variables to the heap objects. Though not shown in Figure 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 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 1 appear to 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 Section 3.

With the situation shown in Figure 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 following program executes statements that cause one object to reference another. All of these statements are executed within Figure 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.

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

Figure 2: Instructions affect object lifetime.

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.

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

$\boxed{5}$ Although E has contaminated D, E no longer references it. Ideally, this should revert the actual liveness of A–D to the situation after $\boxed{4}$. 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 death". 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 Section 5, 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 $\boxed{3}$. 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 $\boxed{3}$ 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 [3]. 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 Section 5 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 generational collectors [18, 10] 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 [9].

- 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 [7]. The resulting overhead is a (nearly) constant amount of work per storage reference.

3 Implementation

We implemented our approach in the context of Sun's Java system, JDK 1.1.8. Our changes were confined to those portions of the Java Virtual Machine (JVM) [11] 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.

- 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.

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 follows.

objects: We augmented each object handle with fields to accommodate union/find of the equilive blocks.

A straightforward implementation would require one "ancestor" field and one integer field to represent the rank (for details on Tarjan's algorithm, see [7]). 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 paper were obtained by introducing four 32-bit words into what was formerly a 64-bit object header. Although good results were obtained, such overhead is excessive, especially for small objects. Reduction of this overhead using the above ideas is the subject of future work.

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.

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.

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.

We began with almost no familiarity of Sun's JVM interpreter. Nonetheless, it took only 6 weeks to implement our 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.

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-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

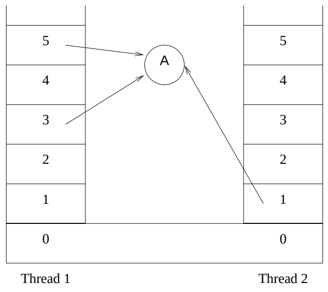

Figure 3: Two threads sharing an object.

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. Within Thread 1, A is dependent on frame 3; however, Thread 2 can also access A until its frame 1 is popped.

For the purposes of this paper, 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 An Optimization

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

$A.a = S$

where S is static—associated with the last-to-be-popped stack frame. As described in Section 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 5 include this optimization, except for one column in Figure 4 which is designed to show the benefits of the optimization.

4 Previous work

Wilson presents an excellent survey of storage allocation [19] and collection [18] 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 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 [15].

- The train algorithm, 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.

4.1 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 [5, 13, 20] 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 [12], 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* [17, 16]. Regions are perhaps the closest in nature to the ideas expressed in this paper. 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 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.

4.2 The train algorithm

Our approach is influenced by the train algorithm [10, 14]. 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 [10], 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.

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 Section 6.

5 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.

- The first program is a student's compiler project (the parsing and semantic analysis phases), written by a Java novice. No thought whatsoever was expended on efficient use of objects. The size of this program is inflated: 6088 of the lines were generated by scanner- and parser-generating tools.

- The programs `trav` and `corners` compute navigation information (based on shortest-path), using US Census road descriptions [4]. These programs were written by a Java expert, hand-optimized to minimize the need for garbage collection. `Ttrav` is a multithreaded version of `trav`, designed to operate as an applet. Similarly, `Tcorners` is a multithreaded version of `corners`. The applet versions load the data in a separate thread, under control of applet buttons that can suspend or resume the loading.

- The bottom set of programs are the SPEC suite [8]. Here, they were run on their smallest problem sizes.[2]

[2]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.

Name	Description	Lines of source	Objects created	Collectable No opt	Collectable With opt
jmm	Compiler from a course	7552	20634	78%	79%
corners	Finds road intersections	4378	177477	36%	99%
Tcorners	Threaded version of above	same	187450	36%	38%
trav	Solves TSP, gives dir'ns	5747	542933	61%	79%
Ttrav	Threaded version of above	same	552936	61%	61%
compress	Modified Lempel-Ziv	920	5144	11%	14%
jess	Expert System	570	46129	36%	42%
raytrace	Ray Tracer	3750	277052	98%	98%
db	Database Manager	1020	8088	24%	41%
javac	Java Compiler	9485	26127	23%	30%
mpegaudio	MPEG-3 decompressor	N/A	7578	8%	9%
mtrt	Ray Tracer, threaded	3750	276108	98%	98%
jack	PCCTS tool	N/A	410479	70%	90%

Figure 4: Benchmarks; Percentage of objects collectable by our approach, without and with the optimization described in Section 3.4.

5.1 Collectable Objects

For each benchmark, Figure 4 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.

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 30%–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.

5.2 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 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.

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 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 `jack` benchmark, almost all 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 4, static approaches may serve well here.

5.3 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 7 shows that most of our benchmarks had very few thread-shared objects. The `mtrt` and `raytrace` programs are equipped to run multithreaded, but did not in fact use multiple threads for the data sizes we supplied. On the other hand, the applet-versions of `corners` and `trav` generate a graph in one thread that is used by another. The graph itself is treated statically because of this thread sharing. Also, any object referenced by the graph, or referencing the graph, becomes unioned with the graph's objects. All such objects are treated as live for the program's duration in our approach. Such harsh treatment of thread-shared objects is unnecessary, as described in Section 6.

Name	Total	Number of blocks of size							Percent
	Collectable	1	2	3	4	5	6–10	> 10	Exact
jmm	16283	2038	1070	2676	953	53			13%
Tcorners	71510	10320	4849	14413	572	1193			14%
corners	174901	10315	4848	11988	514	1193		1	6%
Ttrav	336894	48571	6682	81260	3521	757	482	38	14%
trav	429132	141846	6681	80772	3463	757	482	45	33%
compress	709	206	110	32	7	2	1	2	29%
jess	19604	3380	1829	3158	19	68	10	31	17%
raytrace	272552	40516	9491	1503	1835	3	4278	2	15%
db	3323	819	138	321	4	2	1	3	25%
javac	7768	3413	640	340	176	142	72	2	44%
mpegaudio	717	214	108	39	8	2	1	1	30%
mtrt	271622	40323	9366	1476	1799	2	4278	2	15%
jack	366818	136001	85463	13517	4719	30	27	1	37%

Figure 5: Distribution of block sizes.

Name	Distance from birth to death frames.						
	0	1	2	3	4	5	> 5
jmm	4574	3842	4336	2204	1327		
Tcorners	15149	11501	16297	17460	9655	1448	
corners	15146	9056	13824	14925	15598	4040	102312
Ttrav	49483	11515	10339	87609	86003	83029	8916
trav	21029	5766	1354	823	2761	4	6
compress	174	193	158	84	64	25	11
jess	6152	3679	6709	503	62	18	2481
raytrace	35806	23823	29071	13876	11449	6383	152144
db	667	620	1207	603	72	22	129
javac	3602	1954	1340	221	406	88	157
mpegaudio	181	204	173	74	56	18	11
mtrt	35581	23759	29003	13685	11260	6287	152047
jack	79977	263071	19495	2521	1717	22	15

Figure 6: Age at death of objects we collect.

Name	Total num of static objects	Percentage of static objects due to threads
jmm	4347	0%
Tcorners	115940	98%
corners	2546	0%
Ttrav	216042	98%
trav	113801	0%
compress	4434	0%
jess	26523	0%
raytrace	4444	1%
db	4763	0%
javac	18357	0%
mpegaudio	6859	0%
mtrt	4430	1%
jack	43659	0%

Figure 7: Percentage of objects that we treat as static (live for the program's duration) due to sharing among threads.

5.4 Performance and Overhead

Finally, we examine the run-time overhead of our approach in Figure 8. We began with Sun's JDK 1.1.8 (call this the *base* system) and modified it to use our CG algorithm. The rightmost column of Figure 8 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 from 4%–24% 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.

The `corners` and `trav` programs are not improved by our approach. However, these were hand-optimized by an expert to avoid garbage generation. Thus, the overhead surfaces but without the benefits for these programs.

To isolate the overhead of maintaining the equilive sets, we ran the base system with the "`-noasyncgc`" flag—and gave it plenty of storage—so that it never ran garbage collection. Thus, the middle column of numbers in Figure 8 shows the speedup (typically slowdown) of our approach over the base system when the base system never needs to collect. In a few cases, we still beat the base system, probably because the cache performance is better for CG than for a system that never collects (and thus needs a lot of primary storage).

5.5 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 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` went from 41% collectable in the small run to 99% collectable in the large run. Similarly, the number of objects that we can collect exactly (because they were uncontaminated) mostly improved in the large runs, except for `db`.

Name	Our time (seconds)	Speedup over JDK -noasyncgc	Speedup over JDK
jmm	4.77	.98	1.14
Tcorners	57.0	.87	.91
corners	55.0	.91	.89
Ttrav	142.0	.89	.89
trav	140.0	.90	.89
compress	772.34	.99	1.05
jess	11.5	1.00	1.24
raytrace	63.9	1.00	1.20
db	3.2	1.00	1.14
javac	7.9	1.00	1.17
mpegaudio	86.3	.99	1.04
mtrt	62.2	.98	1.20
jack	157.5	1.00	1.10

Figure 8: Timing results. The rightmost column shows the speedup of our method over the traditional collector in the JDK 1.1.8 system. The middle column is explained in the text.

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 9: Spec benchmarks, large runs.

Finally, we compare execution times for the SPEC benchmarks in Figure 10. The "small" speedups are reprised from Figure 8; included also are the speedups (and slowdowns) of our method for the medium- and large-scale runs of the benchmarks.

Our approach worked well for the small runs, and it should be noted that even the "small" runs take substantial time. As we move to the medium- and large-runs, our approach starts to lose ground. We believe this is happening for the following reasons.

- Figure 9 shows the statistics for how well we can collect objects for the large runs. Although a large percentage are collectable, it may be the case that we collect them too late to do the long-running programs any good. Indeed, we found that we had to allocate more storage to the long runs. In doing so, the programs hardly collected at all when run with traditional garbage collection.
- Figure 9 shows that when the `mpegaudio` and `compress` programs run longer, they do not allocate more objects. Our approach continues to incur overhead but this is never offset by any real collection of objects for these programs.

name	Small	Medium	Large
compress	1.05	.95	
jess	1.24	.88	.66
raytrace	1.20	.85	.68
db	1.14	.83	.48
javac	1.17	.86	.70
mpegaudio	1.04	.97	
mtrt	1.20	1.19	.67
jack	1.10	1.10	.69

Figure 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.

- Because our method is conservative, and owing the nature of contamination, the preciseness of our collector only degrades with time.
- When a frame pops, we return all of its dependent objects to the heap. Currently, our data structures do not mesh well with the freelist of the heap-allocator. As a result, we return objects one-at-a-time to the heap. By reconciling our data structure with the freelist's, we can return all objects with a single operation.

Based on our experimentation, we next present ideas for future work.

6 Conclusions

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 and Figure 9).
- Of those objects that are CG-collectable, most occur in blocks with three or fewer objects (Figure 5).
- For some programs (such as `jack` and `jess`), most objects that we can collect are collected within one or two frames of their birth (Figure 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 well for the small runs of the SPEC benchmarks, performance is lost on the longer runs.

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 [6, 9] 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 [5, 16] 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.

Acknowledgements

We thank Guy Steele Jr. and Sun Microsystems for access to their Java interpreter. We also thank the PLDI program committee for their review of our work and their helpful comments. In particular, we are grateful to Mads Tofte for showing us the important connection of our approach with "regions".

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] 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.*

[4] US Census. Tiger mapping service, the "coast to coast" digital map database. Technical report, US Census Bureau, 1999. http://tiger.census.gov/.

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

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

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

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

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

[10] 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.

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

[12] 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.

[13] 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.

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

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

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

[17] 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.

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

[19] 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.

[20] 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.