Garbology: A Study of How Java Objects Die

Raoul L. Veroy
Computer Science
Tufts University
rveroy@cs.tufts.edu

Samuel Z. Guyer
Computer Science
Tufts University
sguyer@cs.tufts.edu

Abstract

In this paper we present a study of how Java programs dispose of objects. Unlike prior work on object demographics and lifetime patterns, our goal is to precisely characterize the actions that cause objects to become unreachable. We use a recently-developed tracing tool, called Elephant Tracks, which can localize object deaths within a specific method and tell us the proximal cause. Our analysis centers around *garbage clusters*: groups of connected objects that become unreachable at precisely the same time due to a single program action. We classify these clusters using traditional features, such as size, allocation site, and lifetime, and using new ones, such as death site and cause of death.

We present results for a set of standard benchmarks including SPECJVM98, SPECjbb, and DaCapo. We identify several patterns that could inform the design of new collectors or tuning of existing systems. Most garbage clusters are small, suggesting that these programs almost always dispose of large structures in a piecemeal fashion. In addition, most clusters die in one of only a dozen or so places in the program. Furthermore, these death sites are much more stable and predictable than object lifetimes. Finally we show that this information could inform a new kind of garbage collection algorithm, which we call *cluster-aware garbage collection*, and we evaluate its potential using a GC simulator.

CCS Concepts •**Software and its engineering → Garbage collection; Dynamic analysis;**

Keywords Memory management, Dynamic analysis, Java

ACM Reference format:
Raoul L. Veroy and Samuel Z. Guyer. 2017. Garbology: A Study of How Java Objects Die. In *Proceedings of Onward!'17, Vancouver, BC, Canada, October 25–27, 2017,* 12 pages.
DOI: 10.1145/3133850.3133854

1 Introduction

Many of the dramatic performance improvements in garbage collection over the last 20 or 30 years have been driven by observations about common patterns of heap memory use. The best known, and perhaps most successful is generational collection, which exploits the weak generational hypothesis that most objects die young (Ungar 1984). Generational collectors are widely deployed in practice, and have given rise to a large family of related collection schemes, such as age-based garbage collection (Stefanović et al. 1999) and pretenuring (Blackburn et al. 2001), as well as the use of specialized memory spaces for large objects and immortal objects. A nagging problem, however, is how to improve performance beyond generational collectors, particularly for programs with non-generational behavior (e.g., software caches) or with very large heaps that defy simple lifetime classifications.

An exciting trend that has been gaining momentum is to develop collectors that support application-specific policies and mechanisms. Recent promising examples include cluster collection (Cutler and Morris 2015), data structure aware garbage collection (Cohen and Petrank 2015), and connectivity-based garbage collection (Hirzel et al. 2003). In order to be effective, however, these new techniques rely on detailed information about how a program creates, connects, and ultimately disposes of objects. Prior work on computing this information includes a number of empirical studies of object demographics, heap structure, and GC behavior(Blackburn et al. 2004; Dieckmann and Hölzle 1999; Hirzel et al. 2002; Jones and Ryder 2008).

Notably absent from these studies, however, is an analysis of exactly how objects become garbage: *What program events create garbage and where does that happen in the program?* Part of the reason is that until recently the tools for analyzing heap memory behavior (GC tracing tools) were not precise enough to discover this information. The original work on the Merlin tracing algorithm (Hertz et al. 2006), for example, uses a fairly coarse notion of time, called *allocation time*, that only advances at object allocations. As a result, recorded program events (such as object deaths) cannot be localized any more precisely than the nearest allocation site.

In this paper we use the Elephant Tracks GC tracing tool to perform a detailed study of how and where objects become garbage (Ricci et al. 2013). Elephant Tracks measures time in method calls/returns, which is both more precise and more informative than allocation time. It allows us to determine the exact location in the program where each object become garbage, which in turn allows us to identify the program action that caused it. For example, we can compute how many objects die as a result of losing a stack reference (versus losing a heap reference), and we can determine in which methods these events occur.

Onward!'17, Vancouver, BC, Canada
2017. 978-1-4503-5530-8/17/10…$15.00
DOI: 10.1145/3133850.3133854

Knowing both where objects are allocated *and* where they die gives us the full dynamic extent of their lifetime in the program. Program scope is qualitatively different from any measurement of lifetime. In particular, it is much less arbitrary: it is defined only by the program actions that directly impact an object. We find, for example, that scopes are stable and predictable. In most cases, objects allocated at a particular program point only die in a small number of possible places; often only one or two. These objects can nonetheless have wildly different lifetimes because different code paths – in many cases irrelevant to the objects in question – are executed between their allocation and death.

Our study centers around *garbage clusters*: groups of connected objects that become unreachable together as a result of a single program action. For each of our benchmark programs, we analyze garbage clusters in a variety of ways, including size, scope, connectivity, and cause of death. In addition, we compute *memory flows*: collections of clusters with same dynamic extent. We present this data for a set of well-known Java benchmarks: SPECJVM98, DaCapo, and SPECjbb. Some of our key findings include:

- Most garbage clusters are small. With the exception of one benchmark, clusters are rarely larger than 10 objects, suggesting that Hayes's notion of a *key object* does not exist at any significant scale in these programs.
- Most clusters die in one of only 10 to 20 different methods.
- Memory flows are highly predictable. That is, most of the objects created at a given allocation site die at one of only two or three program points (in many cases, only one). This observation holds even when the objects vary wildly in their lifetimes.
- Some clusters contain cyclic garbage (one of the major challenges for reference counting collectors), but the vast majority of these cycles are small (one to four objects) and are entirely contained within a single class (i.e., all objects are instances of one class and/or any nested classes it contains).

We believe that detailed information about patterns of object disposal will enable a new class of application-specific collection algorithms that exploit these patterns. We present one possible algorithm, called *Cluster Aware Garbage Collection* (CAGC), based on an incremental marking scheme that exploits our knowledge of garbage clusters. We implement our oracular algorithm in a simulator to show its potential. We believe that much more is possible.

The rest of this paper is organized as follows. First, we review prior work in object demographics and garbage collection tracing tools. Second, we explain how we analyze traces to produce the data for our study. Next, we present results for the individual benchmarks as well as aggregated results. We then present our prototype CAG collector simulator and the results of running the simulator on our data. Finally, we discuss future directions for both the analysis and garbage collector design.

2 Background and related work

Our study builds on prior work in several ways. First, we continue a line of research that aims to characterize the memory workloads of object-oriented programs and provide information that can assist in the further memory management research. Second, we use a dynamic analysis tool to collect traces of the memory behavior of programs. The tool we use builds on existing techniques and algorithms, but provides a new level of precision. Thirdly, we develop a novel garbage collection algorithm that exploits the knowledge gained in the dynamic analysis. We evaluate the GC algorithm in a simulator.

2.1 Object demographics

Many garbage collection research papers use object demographics data to motivate the design of the collector. Notable examples include the original work on generational scavenging (Ungar 1984) and work on pretenuring (Blackburn et al. 2001). For the most part these papers exploit coarse-grained patterns in the lifetimes of objects – for example, distinguishing long-lived from short-lived objects – and precision is not crucial. In some cases they use precise information about the allocation sites of objects, since this information is readily available. Since object death information has not been easy to compute few collectors have been designed around this information.

Connectivity-based garbage collection is a substantially different approach that uses static analysis to organize objects into groups with similar lifetimes, allowing the runtime system to focus on groups most likely to be garbage (Hirzel et al. 2003). This work was supported by a study showing that connected objects tend to die together (Hirzel et al. 2002). The specific analysis computed the probability that two objects have the same lifetime if they are connected by some chain of references in the heap.

Several well-known papers have been devoted entirely to the empirical study of object demographics. The most recent study by Jones and Ryder provides an in-depth analysis of object lifetimes. It presents age data for different categories of objects – application vs library vs virtual machine objects – across different input sizes (Jones and Ryder 2008). It also shows the correlation between lifetime and allocation context (allocation site in a calling context). As with prior studies, however, lifetimes are characterized in a fairly coarse way, and the underlying data is only precise for object allocation and heap update events. Our goal is to complement these studies with precise information about object deaths.

2.2 Collecting GC traces

Early work on analyzing programs for memory management involved running the garbage collector more frequently in order to compute object death times more precisely (Stefanovic 1999). This technique is extremely limited, though, because each collection is so expensive. The Merlin algorithm provided a breakthrough in performance by allowing precise death times to be computed during regularly-scheduled garbage collections (Hertz et al. 2006). The key idea in this algorithm is to timestamp objects when they are used, so that when they are found to be garbage by the collector, the last timestamp indicates approximately when they were last reachable. The challenge is figuring out when to timestamp objects reachable from the stack (local variables). The strategy in the original paper is to timestamp everything on the stack at each allocation site, which provides a balance between precision and performance. The major downside is that since time is tied to allocation sites, it is not possible to precisely locate events in the program structure.

3 Trace collection and analysis

In this section, we first describe the raw trace information provided by the Elephant Tracks tracing tool; then we discuss how we analyze this information systematically to identify interesting memory use patterns.

3.1 Trace entries

An Elephant Tracks trace consists of a sequence of heap-related events recorded during execution of a Java program. Each kind of event includes information about the objects involved:

- An **object allocation** event includes the object type and object size, and number of elements in the case of arrays.
- A **pointer update** event includes the source and target objects, the field or array element being updated, and the old target (if there was one).
- A **root** event indicates when an object reference is used (a witness to the object being reachable from the stack).
- A **death** event identifies when an object becomes unreachable (the earliest point it could be collected).
- **Method entry and exit** events.

Every object allocated is assigned a unique identifier, so subsequent events can refer to the exact object instances involved. At this level of detail the resulting traces are enormous – in the 100s of gigabytes for the larger benchmarks.

3.2 Method time

The Merlin algorithm, which we employ, relies on timestamping in order to place object death events in their proper place in the trace. The exact definition of "trace time" is arbitrary, but the more fine-grained it is, the more precisely object deaths can be placed among the other events (which are simply recorded in the order they occur).

Garbage collection researchers have traditionally measured time in terms of bytes allocated, since it places events relative to the consumption of memory (Jones et al. 2011). This notion of time, called *allocation time*, is useful for studying GC algorithms because it produces a trace that is equivalent to running the collector at every allocation – the most frequent collection schedule that would make sense. The problem with allocation time is that it is too coarse to use for studying other kinds of events. For example, an object that becomes garbage when a method returns will appear to have died at the next allocation site, even if it is in a completely different part of the code.

Elephant Tracks, on the other hand, ticks the trace clock at every method entry and exit. This notion of time, called *method time* is much more precise than allocation time: method boundaries are encountered 10-20 times more frequently than allocations. More importantly, however, each discrete time corresponds to a specific dynamic method invocation (or part of a method). This level of precision allows us to identify the exact program event that causes an object (or set of objects) to become unreachable. Furthermore, by matching entries and exits, we can reconstruct the full calling context of any event. Later in this section we describe an example of how this information is used.

3.3 Cause of death

The first step in our garbology analysis is to classify objects according to their cause of death. At the most fundamental level, objects can only die in two ways: by losing a stack reference, or by losing a heap reference. We refer to these categories as *dying by stack* and *dying by heap*. We can compute this information easily from an Elephant Tracks trace, but it is more informative to break down these categories based on circumstances.

Many objects die when a program terminates, and these "immortal" objects do not fall cleanly into either of our categories. While this information might be useful for some kinds of collectors (Blackburn et al. 2001), we exclude immortal objects from our analysis. The focus of this study is on understanding how programs explicitly dispose of objects in their steady state, which is crucial for long-running programs, such as server applications.

Quite often, the die-by-stack case is incidental, and the real cause of death is a heap store. Consider a `remove` function for a container data structure: it might use a stack reference to point to the element to be removed while it modifies the structure. The direct cause of death of the removed object is the loss of that stack reference when the `remove` method returns, but the proximate cause is really the overwriting of the associated heap references in the container. We remove these cases from *die by stack* and place them in their own

category called *dying by stack after heap*: when an object first loses its last heap reference, then its last stack reference.

3.4 Garbage clusters and key objects

One problem with the analysis described so far is that it does not properly account for objects that die indirectly when the object(s) pointing to them become garbage. For example, a single variable going out of scope could cause a group of connected objects to *all* become garbage. Clearly, the object directly referred to by the variable dies by stack, but what about the objects it points to? One could argue that they die by heap, since heap references are going away, but that does not seem like a useful classification. Instead, we atrribute the same cause of death to all objects in the group.

We define a *garbage cluster* as a set of objects that die at the same time as a result of a single action by the program. For heap actions, there is only ever one heap reference that is overwritten. The classification is more subtle though for stack actions: there are often multiple stack references that go out of scope when a method exits. Rather than lump together potentially unrelated objects, we consider each stack reference as a separate event. Occasionally, there are multiple variables that point to the same object, or even worse, that share structure. Our analysis chooses the cause by determining which incoming reference is older. This design decision is in line with the spirit of our analysis because an object intuitively belongs to a garbage cluster if it has been alive with that cluster the longest.

Garbage clusters are closely related to the idea of *key object opportunism*, first proposed by Hayes (Hayes 1991). He suggested that there are special objects that control the lifetimes of many other objects. When those key objects die, many other objects die as a results. Hayes proposed a collector that schedules collections around the lifetimes of these objects. It was never clear, however, whether key objects actually control a significant amount of memory resources. We can use our garbage clusters analysis to check this claim. The key object in each cluster is the object that is most directly affected by the cause of death (e.g., the object whose last reference the program removes).

3.5 Program contexts

Our analysis uses Elephant Tracks to identify exactly where an object dies in the program. The source trace allows us to determine the full calling context of every event, including object death. Using the full calling context though, generates too much information and results in an infeasible analysis.

In order to make the analysis more practical, we use the static location of the death event. That is, we use the top of the stack as the object's death location. Quite often, however, the top of the stack is just a library method. As programmers, we might be more interested in the application method that made the call further down the stack. Our analysis therefore also computes the nearest non-library method in the full calling context. The static location together with the nearest non-library method are used instead of the full calling context.

Our analysis identifies the following packages as libraries:

- `java.*`
- `sun.*`
- `com.sun.*`
- `com.ibm.*`

Any class from those packages are considered library classes with library methods. Thus, all other classes and methods are defined to be non-library classes and methods.

3.6 Properties of garbage clusters

Once we have aggregated all of the object deaths into garbage clusters, we can analyze their properties, including allocation and death contexts of each cluster, the age in both allocation time and method time, number of objects in the cluster, and total size of all objects in bytes.

While all objects in a garbage cluster die at the same time, they are all allocated at different times. In order to present a summary of the allocation properties of clusters, we classify them according to the oldest object in the cluster. In many cases, this object also happens to be the key object. We also compute the range of ages of the objects relative to the age of the cluster as a whole.

3.6.1 Cycles in garbage clusters

Finally, we analyze the connectivity of objects in each cluster to determine if there are any cycles. The precision of our analysis allows us to determine exactly how often cyclic garbage occurs, and compute properties of the cycles, including their size and the types of the objects involved. This information is interesting for reference counting collectors, which cannot collect cyclic garbage without a separate algorithm (typically some form of tracing). Using the trace information, we can reconstruct the connectivity of objects after they are dead and apply a standard strongly-connected components analysis to find cycles (Tarjan 1972). Note that many programs create cyclic structures of varying sizes during execution, but we are interested only in the cycles that still exist when the objects die.

3.7 Analysis implementation

Our heap analysis is implemented in two stages. The first stage is a trace processor written in C++ that reads the raw Elephant Tracks traces and simulates the events in order, allowing us to reconstruct the intermediate states of the program. The trace processor models the objects in the heap and the pointers between them, and keeps track of when they become garbage. In addition, it uses the method entry and exit log entries to keep track of the calling context and label each heap event with this information. When the trace processor finds a group of objects with the same death time, it inspects the recent history and calling context information

to identify the cause. Since it has a full model of the heap, it can determine which objects are in connected groups.

The second stage of the analysis is written in Python. This part of the analysis does further aggregation and computation on the data generated from the C++ simulator. This part of the analysis computes the following: (i) The top allocation-death context pairs; (ii) overall statistics of the precise action that caused objects to die; (iii) garbage cluster membership.

The analysis was performed on a 64-core, 2.5 GHz Opteron Processor 6380 system with 126 GB of memory, running Red Hat Enterprise Linux Server release 6.8. While we had multiple cores available, the simulator currently runs in a single thread only. The analysis in its current form needs a lot of memory to analyze the Elephant Tracks trace. Thus the simulator is unable to finish running on the larger benchmarks: `eclipse`, `jython`, `h2`, and `sunflow`. We plan to improve the performance of the simulator so that we are able to successfully analyze these larger benchmarks.

4 Cluster-aware Garbage Collection

Our proposed GC algorithm relies on the intuition that program behavior is clearly related to program structure. Since objects die in clusters, we propose that garbage clusters can be the basic unit of garbage collection. While the granularity might be too fine for some applications, the cluster grouping surprisingly has good results for some applications as we will show in the results.

4.1 CAGC simulator design

The CAGC simulator is based on an idealized mark-sweep scheme. To simplify the experiment, the collector is implemented as non-generational. Using the same Elephant Tracks traces used in the Garbology analysis and a given heap size, the simulator models the allocation and collection of all objects in the trace.

The heap is separated into two regions: (i) a *regular region*; and (ii) the special *deferred region*. The CAGC simulator then takes a predetermined number of garbage clusters from the previous analysis to treat as a designated group of objects. This special group identifies which objects will be allocated into the deferred region. All other objects are allocated into the regular region.

Simulated garbage collection is triggered when the simulator heap cannot fulfill an allocation request. The CAG collector can perform two kinds of collections. First, the collector will attempt a regular collection which targets only the regular region. If enough free memory is reclaimed to satisfy the request, the collector ends the collection phase. If not, a full collection involving the deferred region is performed. Again, if enough free memory is reclaimed, the collector ends the collection and transfers control back to the application. If not enough memory is available, an out of memory error is generated because the heap size configuration cannot satisfy the program requirements.

The *Cluster Aware GC* (CAGC) simulator is a trace processor written in C++, which shares some code with the Garbology analysis trace processor. The CAGC simulator can be configured with the following parameters:

- The maximum heap size.
- The garbage clusters to allocate to the deferred region. These clusters are selected from the initial Garbology analysis.
- The full program Elephant Tracks (ET) trace.

The reliance on the garbage cluster information produced by the Garbology analysis is precisely what makes this implementation of the CAG collector oracular. We recognize that there may be no easy way to determine garbage cluster membership to the precision that we are able to in the Garbology analysis, since the analysis needs a complete ET trace. That is, it is highly impractical to want to run an application to the end, before actually running it using the garbage cluster information. We do believe however that this oracular simulation should lead to future work where garbage cluster membership may be determined with high probability.

Since the ET trace contains all allocation and reference update events, the simulator can precisely model the program heap. Program execution is modeled by maintaining the heap according to the allocation policies in the CAGC design. If an object is part of the special set of garbage clusters, then that object is allocated in the deferred region. Otherwise, the object is allocated in the regular region.

The ET trace also contains all reference updates, which allows the simulator to model precise object connectivity by maintaining information on all reference edges. Note that if we ignore the edge's direction, there are 3 ways an edge can be located with respect to the regions. An edge's endpoints can be one of three possible configurations:

- Both source and target are in the regular region.
- Both source and target are in the deferred region.
- The source and target are different regions. While this may further be broken down into two separate cases, we ignore the edge direction because the direction adds nothing to our analysis.

Such a classification enables us to calculate GC costs for the CAGC algorithm.

We use the *mark/cons ratio* as our measure of GC cost. *Mark* is a measure of GC activity which includes tracing and copying. We do not include other effects like cache misses that can also affect GC costs in order to simplify the experiment. For our purposes we use *number of objects traced* as a simple proxy for the *mark* part of the ratio. *Cons*, on the other hand, is a measure of the application mutation activity. We use the number of bytes allocated to represent the *cons* part of the ratio. We use this simplified mark/cons ratio to measure the performance characteristics of our CAG collector.

Since the Elephant Tracks trace allows the simulator to determine precisely when an object dies, the simulator is able to correctly simulate all garbage collection activities. When a garbage collection is invoked, the simulator keeps track of all marking costs. Thus we can compare the performance of the CAG collector to a naive mark-sweep collector.

4.2 Experimental methodology

How big should the simulated heap be? This is an important experimental parameter since heap size will affect the collector performance. Blackburn et al showed that you can decrease the number garbage collections for any program if you give it a large enough heap (Blackburn et al. 2004). We take a range of possible heap sizes to study the relationship between heap size and CAGC performance.

First, we determine the smallest heap size that we will run the simulation on. For this, we use the program's *maximum live size*. The *live size* of a program is the size of the part of the heap that's currently reachable from the program roots at a given time. The *maximum live size* can then be determined over a complete program run. In our simplified simulator world, we take a program's maximum live size as the starting point since this would be the minimum heap size that a program needs to run.[1]

We then generate the sequence of heap sizes using some percentage of the maximum live size. This percentage can range from 25% to 100%. For the maximum heap size, we simply run on the next available heap size until no garbage collections are incurred for that heap size. While achieving zero collections isn't necessary, this ensures that our data points are complete. Each heap size data point generates an associated *mark/cons* ratio that we can compare to the *mark/cons* ratio of a straightforward marking collector.

5 Results

In this section we present results from our analysis of GC traces, and we show the potential of using this information in a new kind of garbage collector.

5.1 Benchmarks

For our experiments we analyze two common benchmark suites, the SPECjvm98 (spe 1998) and DaCapo (Blackburn et al. 2006) benchmarks, as well as SPECjbb (spe 2005). The benchmark programs were run on the J9 JVM with the Elephant Tracks (Ricci et al. 2013) tracing tool enabled.

5.2 Overall demographics by death cause

In this section we present empirical results of our Garbology analysis, including some detailed analysis of individual benchmarks as well as overall patterns across all of them. In general, we find that garbage clusters are typically small

[1]We actually take the *maximum live size* and round it up to the nearest 4K.

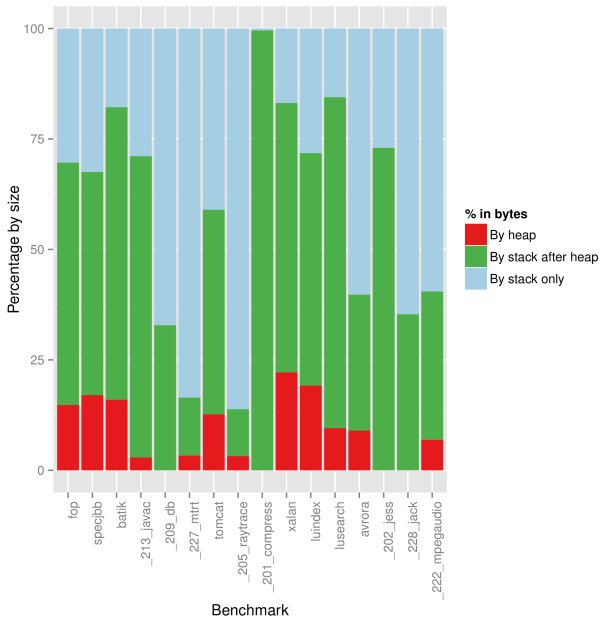

Figure 1. Objects' cause of death classified in percentage of memory size. We show objects that died by stack action but were previously pointed at by the heap as a separate category.

– mostly 1 to 10 objects – and that their death is largely predictable.

Figure 1 shows all of the benchmarks ordered by maximum live size in descending order. Here we show the general classification of death cause as a percentage of memory size (see Section 3.3 for a detailed description of the categories). We include immortal objects in a separate category. Notice that all benchmarks have a significant percentage of objects dying by stack (i.e., their last reference is a stack reference). Some benchmarks (`specjbb`, `xalan`, and `luindex` to name a few), have a relatively higher proportion of objects that die by heap (i.e., they become unreachable when a heap pointer moves). All in all, though, this graph shows a surprisingly high proportion of *die by stack* objects.

As mentioned in Section 3.3, we often find that *die by stack* objects have been recently removed from a heap-based data structure prior to becoming garbage. To account for this behavior, we reanalyzed the data, looking for cases where objects die immediately after losing a heap reference. We reclassified these objects as *die by stack after heap*. The resulting graph is shown in figure 1, which fits more closely with our intuition about how these programs work.

This result also hints at a more important pattern of behavior: objects often die as a result of a specific, intentional action by the program (e.g., removal from a data structure) that is strongly predictive. This kind of information could be used in a number of ways to improve memory management. For GC algorithms that rely on estimating tenancy, such

	Garbage cluster sizes												
benchmark	1	2	3	4	5	6	7	8	9	10	11	12	13+
_201_compress	2,983	1,205	156	4	2	1	0	0	0	0	0	0	0
_202_jess	3,733,934	2,091,909	3,420	99	18	0	2	0	0	0	0	0	0
_205_raytrace	5,839,977	235,636	15,797	3,112	68	1	2	0	0	0	0	0	0
_209_db	2,907,822	112,389	171	5	15,546	1	2	0	0	0	0	0	0
_213_javac	2,859,266	898,705	157,208	66,884	13,922	6,591	3,101	1,121	393	230	187	124	3,276
_227_mtrt	6,000,607	268,224	25,106	4,270	96	2	2	0	0	0	0	0	0
_228_jack	3,632,399	432,625	124,296	2,166	628	1,374	35	16	672	16	16	0	0
avrora	1,397,610	161,820	105,905	220	29	12	11	7	0	1	0	0	0
batik	730,811	74,379	28,456	7,612	3,550	134	38	165	1	6	0	1	4
fop	1,433,033	573,648	87,696	7,944	3,569	461	187	97	18	25	0	1	5
luindex	267,693	58,206	8,939	85	33	20	9	1	0	1	0	1	0
lusearch	6,905,703	1,516,410	677,400	289	131,156	53,267	6	6	2	2	0	1	3
specjbb	3,785,434	568,349	6,951	16	1	0	0	0	0	0	0	0	1
tomcat	8,660,718	925,172	483,161	11,280	2,372	7,778	154	83	235	22	1	21	4
xalan	5,737,589	399,461	60,187	16,352	4,220	63	9	3	0	4	0	1	0

Table 1. Number of clusters of each size (columns) for each benchmark (rows). With the exception of `javac` most programs dispose of objects in small groups at any given time.

as the garbage-first collector, the program could be instrumented with hints to let the collector know that a group of objects is likely to be garbage.

5.3 Most clusters are small

In table 1 we show the distributions of garbage cluster sizes across all the benchmarks. For each benchmark, we count the number of clusters that consist of one object, of two objects, etc., up to 12 object clusters. Any clusters with 13 or more objects are lumped into a single category. One of the most notable patterns we found is that garbage clusters are mostly small – almost all are less than 10 objects, and the vast majority have four or fewer.

The `213_javac` benchmark is a notable exception. The `javac` benchmark appears to create large structures consisting of 700 to 1000 objects that represent types and instructions. These structures disappear all at once when the information is written to the compiled class file.

The prevalence of small garbage clusters suggests that for most Java programs, *key objects* of any significant magnitude, do not exist. Instead, most programs dispose of large data structures in a piecemeal fashion, dismantling and processing them bit by bit. This style of disposal has certain advantages, though. For one, it keeps memory drag fairly low (that is, programs do not have significant amounts of reachable, but dead data in memory). It also reinforces the observation above that object disposal is an intentional act by the program, even though there is no explicit delete operation.

5.4 The number of death sites is small

Like many heap profiling tools before ours, we can label clusters according to where they are allocated. More importantly, though, we can also identify them according to where in the program they become garbage. One of the unique features of our analysis is that we can compute the exact calling contexts for both the allocation and the death of every object. Unfortunately, this information is much too detailed, and it is impossible to present in a compact and useful way. Instead, we look only at the method containing the allocation and death sites, with one slight twist. In many cases, objects are created and die inside Java library methods. For example, if a program calls *clear()* on a container, all of the elements appear to die inside the container's *clear()* method. The real question, however, is where *clear()* was called. So, for this analysis, when computing the allocation and death methods, we walk up the call stack to find the nearest non-library method.

Table 2 shows the top site, by volume, where clusters of objects become garbage. One notable feature of these results is that for most programs, the top death sites account for a large fraction of all allocated memory. In fact, we found that 90% or more of all memory dies at one of the top 20 death sites, depending on the program. A few of the benchmarks, such as `batik` and `tomcat` have more death sites accounting for 90% or more of the total garbage as shown in table 3.

We can also aggregate clusters by their allocation site, and look at how many death sites they potentially reach. Table 4 shows these results. In some cases, clusters of objects allocated at one site only die at a small number of sites, suggesting that the fate of these objects is highly predictable.

Benchmark	Site	% of total allocation	Garbage/ max livesize ratio	number of allocation sites
_201_compress	spec/benchmarks/_201_compress/Input_Buffer.readbytes	85.64	12.68	2
_202_jess	spec/benchmarks/_202_jess/jess/Node2.runTests	56.60	109.57	9
_205_raytrace	spec/benchmarks/_205_raytrace/Point.GetZ	58.05	20.19	18
_209_db	spec/benchmarks/_209_db/Entry.equals	66.60	7.20	4
_213_javac	spec/benchmarks/_213_javac/Type.tClass	17.40	3.47	79
_227_mtrt	spec/benchmarks/_205_raytrace/Point.GetZ	39.96	7.75	26
_228_jack	spec/benchmarks/_228_jack/RunTimeNfaState.Move	46.50	71.68	20
avrora	avrora/sim/radio/Medium$Receiver.earliestNewTransmission	29.80	10.72	3
batik	org/apache/batik/bridge/BridgeContext.finalize	13.77	0.75	222
fop	org/apache/xmlgraphics/ps/PSGenerator.formatDouble	24.49	1.25	118
luindex	.../lucene/analysis/standard/StandardTokenizerImpl.yyreset	23.78	1.76	1
lusearch	.../lucene/queryParser/QueryParserTokenManager.ReInit	40.29	130.66	3
specjbb	spec/jbb/StockLevelTransaction.process	22.56	2.47	2
tomcat	org/dacapo/tomcat/Page.stringDigest	23.69	15.64	29
xalan	org/apache/xalan/transformer/TransformerImpl.transform	20.40	32.20	49

Table 2. Each benchmark's top death site by total garbage measured in bytes.

Benchmark	Site	% of total allocation	Garbage/ max livesize ratio	number of death sites
_201_compress	spec/benchmarks/_201_compress/Compress.spec_select_action	88.46	13.10	10
_202_jess	spec/benchmarks/_202_jess/jess/Node2.appendToken	59.01	114.25	40
_205_raytrace	spec/benchmarks/_205_raytrace/OctNode.Intersect	32.89	11.44	4
_209_db	spec/benchmarks/_209_db/Entry.equals	66.01	7.14	1
_213_javac	spec/benchmarks/_213_javac/Scanner.bufferString	15.18	3.03	13
_227_mtrt	spec/benchmarks/_205_raytrace/OctNode.Intersect	31.08	6.03	103
_228_jack	spec/benchmarks/_228_jack/RunTimeNfaState.Move	46.50	71.68	3
avrora	avrora/sim/radio/Medium$Receiver.earliestNewTransmission	29.07	10.40	1
batik	org/dacapo/harness/DacapoClassLoader.loadClass	11.47	0.63	30
fop	org/apache/xmlgraphics/ps/PSGenerator.formatDouble	18.99	0.97	22
luindex	org/apache/lucene/demo/FileDocument.Document	25.94	1.92	7
lusearch	org/apache/lucene/queryParser/QueryParser.parse	26.46	85.81	10
specjbb	spec/jbb/StockLevelTransaction.process	22.56	2.47	3
tomcat	org/dacapo/tomcat/Page.stringDigest	21.04	13.89	6
xalan	org/apache/xpath/VariableStack.reset	21.20	33.46	35

Table 4. Each benchmark's top allocation site by total allocated.

For example, objects from `specjbb`'s top allocation site die in only three sites.

Using allocation sites as predictors is nothing new, as it was studied extensively by Jones and Ryder (Jones and Ryder 2008) where they related allocation sites to object lifetimes. On the other hand, we have shown how some allocation sites are highly predictable with respect to the corresponding death site. Note that while this isn't true of all allocation sites, the existence of a few highly predictable allocation-death site pairs means that there is predictable behavior we can exploit in a tuned garbage collection implementation.

5.5 Flow of memory

We aggregate the information for all clusters with the same pair of allocation and death sites, which describe a kind of *memory flow* from a source (allocation) to a sink (death site). We characterize a memory flow by the total amount of memory it accounts for and the number of objects. We also show the range of object ages in each flow. Suppose a garbage cluster `G` dies at a given site pair. Using allocation time as the basis, our analysis takes the oldest member of the cluster and uses this for the minimum age and maximum age attribute of the site pair. The minimum age for a site pair

	Number of sites	
Benchmark	90%	95%
_201_compress	3	5
_202_jess	6	7
_205_raytrace	9	14
_209_db	3	4
_213_javac	37	56
_227_mtrt	19	29
_228_jack	13	19
avrora	9	14
batik	63	111
fop	45	77
luindex	27	43
lusearch	3	9
specjbb	15	20
tomcat	81	148
xalan	44	90

Table 3. Number of death sites that account for at least 90% to 95% of total garbage.

then, is the minimum among all the oldest members of every garbage cluster belonging to the allocation-death site pair. The maximum age is calculated in the same way.

Table 7 shows all the benchmarks using only the top allocation-death site pair by total size (shown in MB). We can see that a large number of garbage clusters follow one of only a handful of flows through the program. For most of the programs, the top pair accounts for 10% or more of all the memory allocated by the program. Notice that in a number of cases, the allocation site and the death site are the same method. These cases indicate structures that are essentially local to those methods. Also notice that in many cases the objects involved have a wide range of ages in allocation time, even though their lifetimes are highly stable from the standpoint of memory flow. This result highlights the problem with using allocation time as metric for studying object demographics: even thought strong patterns exist, they are not readily apparent from the object ages.

Taken together, these results suggest that we should focus on developing garbage collection algorithms that can take advantage of knowing:

- Which objects form garbage clusters. That is, which objects tend to die together.
- Where in the program objects tend to die.

5.6 Cyclic garbage

Most of our benchmarks create cycles of objects, some of which die together as cycles. Somewhat surprisingly, however, we have found that by the time these cycles become garbage they are typically very small – only one to four objects. Table 6 shows a summary of this data aggregated across all of the benchmarks. The way we group cycles for this table is by the types that make up the cycle – we refer to this set of types as the *type signature* of the cycle. Each row in the table represents a particular combination of object types. The reason we aggregate across benchmarks is that they have many kinds of cycles in common; typically, cycles created as part of the standard library data structures, such as linked lists and hash tables. Another finding is that many cycles consist of nested types within a single class. That is, the cycle is entirely "contained" in the class that builds it.

It is important to note that only the strongly connected components themselves are small. For most cycles, there are many objects reachable from the cycle that are not part of the strongly connected component. In other words, failing to collect these small cycles would leave significant garbage in memory.

We are planning future work that can exploit the predictability of the type signature, along with the small number of objects in a cycle. We envision a reference counting algorithm that reduces the need for the backup tracing collector by preemptively breaking up the cyclic garbage.

5.7 Cluster aware garbage collection performance

We ran the CAGC simulator on most of the benchmark suites listed in section 5.1. As mentioned in section 3.7, we were not able to finish the Garbology analysis for the following larger benchmarks: `eclipse`, `jython`, `h2`, and `sunflow`.

From the runs, we were able to determine the performance of the CAG collector as compared to a straightforward marking collector using the *mark/cons* ratio. A few benchmarks showed significant performance improvements at the lower end of the heap size range. The most marked improvements were for the `luindex`, `lusearch`, and `_205_raytrace` benchmarks, as shown in table 5. The *CAGC mark cost* column shows how the CAG collector improves the *mark/cons* ratio.

All other benchmarks did not show any significant improvements. There are two possibilities here. First, the garbage clusters chosen may not have significantly aided the CAG collector. If the objects that end up in the region are short-lived, the generational effect would then overwhelm the cluster effect. A second possible reason is that the program may not be amenable to the CAGC algorithm. We expect that there will be a class of programs that CAG collection will work well with. And we totally expect that a large class of programs would not. We leave it up to future work on determining the nature of such programs.

On the other hand, the CAG collector performed the best on the `lusearch` benchmark which has the lowest heap size among the benchmarks presented in table 5. In spite of the help from the oracle, the CAG collector was not able to significantly improve the performance of the benchmarks with larger heaps. Notwithstanding this early result, we believe that our simulated results indicate the future directions for improving and applying the CAG collector.

Benchmark	mark/cons ratio CAGC	mark/cons ratio Normal	CAGC mark cost	Heap size(MB)	Number of collections
luindex	0.12	0.14	89.9%	6.25	76
luindex	0.0268	0.0298	89.9%	7.82	16
lusearch	0.17	0.38	83.2%	4.13	31,988
lusearch	0.44	0.47	92.4%	8.26	5,034
lusearch	0.48	0.50	96.2%	12.39	2,762
_205_raytrace	10.27	12.44	78.9%	7.43	121,025
_205_raytrace	4.06	4.11	98.6%	14.86	33

Table 5. We present here the CAGC simulator performance numbers for select benchmarks. All other benchmarks are not shown here if the performance gain is less than 1%. A benchmark may be shown with multiple heap sizes. Cluster aware garbage collection (CAGC) refers to our idealized GC algorithm using garbage cluster knowledge. Mark costs are relative to a simple mark-sweep collector.

Types in the cycle	Number of cycles	Number of objects		
		Minimum	Maximum	Median
`LinkedList$Link`	106,917	1	2	2
`ElemContext`	41,898	1	1	1
`ConcurrentLinkedQueue$Node`	9,230	1	1	1
`HashMap, HashMap$1`	5,979	2	2	2
`NativeMethodAccessorImpl, DelegatingMethodAccessorImpl`	3,907	2	2	2
`LinkedBlockingQueue$Node`	3,003	1	1	1
`HashSet, HashMap, HashMap$Entry, HashMap$Entry`	2,173	4	4	4
`PythonTree, ArrayList, Object`	1,988	4	4	4
`DTMDefaultBaseTraversers$AncestorTraverser, SAX2DTM, DTMAxisTraverser`	1,300	3	3	3
`HashMap, HashMap$2`	704	2	2	2
`FileChannelImpl, AbstractInterruptibleChannel$1`	132	2	2	2
`NativeConstructorAccessorImpl, DelegatingConstructorAccessorImpl`	131	2	2	2
`SAX2DTM, DTMDefaultBaseTraversers$ChildTraverser, DTMAxisTraverser`	102	3	3	3
`Cleaner`	86	1	1	1
`CSSLexicalUnit$SimpleLexicalUnit, CSSLexicalUnit$IntegerLexicalUnit`	72	2	2	2
`TypeVariableBinding, MethodBinding, TypeVariableBinding`	71	3	3	3
`ConcurrentHashMap, ConcurrentHashMap$EntrySet`	70	2	2	2
`SocketChannelImpl, AbstractInterruptibleChannel$1`	65	2	2	2
`XMLNSDocumentScannerImpl, XMLDocumentScannerImpl$TrailingMiscDispatcher`	62	2	2	2
`BufferUnderflowException`	40	1	1	1
`Label, Frame`	24	2	2	2
`ConcurrentHashMap, ConcurrentHashMap$KeySet`	13	2	2	2

Table 6. Summary of all garbage cycles across all benchmarks. Notice that by the time cycles become garbage they are typically small – only one to four objects. Large cyclic garbage is rare in our benchmark suite.

First, we can target the CAGC algorithm at programs that run in limited memory situations. Modern Java VMs will throw an error if too much time is spent doing collections.[2] We propose that there's a need for GC algorithm improvements in low memory situations that Garbology inspired algorithms plan to fill.

Secondly, we note that the benchmarks suites used by the memory management community have short runtimes. We ran a casual test of the DaCapo benchmarks using OpenJDK 1.7 on an AMD Opteron 6380 at 2.5 GHz, and most benchmarks ran in less than 10 seconds. The `eclipse` benchmark had the longest runtime at roughly 45 seconds. We believe that the standard benchmark suites used by the research

[2]From *3.2 Understand the OutOfMemoryError Exception* in (jav 2016)

Benchmark	Allocation site	Death site	Size (MB)	%	Min age(kB)	Max age(kB)
fop	PSGenerator.formatDouble	PSGenerator.formatDouble	26.32	13%	0.03	17,594
specjbb	StockLevelTransaction.process	StockLevelTransaction.process	77.65	22%	0.00	46
batik	DacapoClassLoader.loadClass	DacapoClassLoader.loadClass	14.07	10%	0.00	49,040
_213_javac	Type.tClass	Type.tClass	44.20	13%	0.00	<1
_209_db	Database.set_index	Database.remove	33.93	20%	15.17	1,061
_227_mtrt	PolyTypeObj.Intersect	Point.GetZ	38.84	14%	0.00	<1
tomcat	Page.stringDigest	Page.stringDigest	190.11	20%	0.00	3,077
_205_raytrace	PolyTypeObj.Intersect	Point.GetZ	60.59	23%	0.00	<1
_201_compress	Decompressor.<init>	Decompressor.decompress	45.34	42%	71.91	83
xalan	VariableStack.reset	VariableStack.reset	105.53	9%	319.37	27,840
luindex	FileDocument.Document	StandardTokenizerImpl.yyreset	10.14	21%	20.03	799
lusearch	QueryParser.parse	QueryParserTokenManager.ReInit	354.20	26%	0.00	4,205
avrora	Medium$Receiver .earliestNewTransmission	Medium$Receiver .earliestNewTransmission	31.91	28%	0.00	1
_202_jess	Node2.appendToken	Node2.runTests	280.70	56%	0.05	50,659
_228_jack	RunTimeNfaState.Move	RunTimeNfaState.Move	122.18	46%	0.00	<1

Table 7. Top allocation-death context pair (by total size) for each benchmark.

community do not reflect an important class of server programs that are designed to run indefinitely. This is how the `lusearch` benchmark, which performs a search over a text corpus, represents the promise of algorithms that exploit garbage clusters. A real world text search program would be expected to do multiple searches over a longer period of time, whereas the `lusearch` benchmark runs in less than 2 seconds.

Additionally, we note that the CAGC simulator only used a single region for simplicity. Given that any program will have numerous garbage clusters, any practical design of the CAGC algorithm should use multiple regions.

5.8 Practical CAG collectors

The improvements were achieved in large part because we used oracular foreknowledge from the Garbology analysis, which is clearly impractical. We believe that Garbology can help develop practical heuristics that can approximate garbage cluster membership. Previously, information about the objects' allocation site and type have been used successfully as predictors for GC algorithms (Blackburn et al. 2001; Jones and Ryder 2008). Consequently, we present some possible avenues for future work on practical heuristics for identifying garbage clusters.

In our Garbology demographics of the DaCapo and SpecJVM benchmark suites, we show that a significant number objects die because of stack action. We believe that future algorithms should exploit this fact. Clearly, stack action is directly correlated to program structure, whether through loops or function calls. By focusing on garbage clusters that die because of stack action, we need to identify the static points in the program that cause garbage clusters to die. Table 2 shows that a small number of death sites is responsible for a non-trivial number of objects. We are currently studying the possibility of exploiting these death sites for the CAG algorithm.

6 Conclusion and future work

Exploiting predictable program behavior has been effective in the design of garbage collection algorithms. Developing new and creative ways to understand program heap behavior is therefore essential for the progress of these algorithms. We have shown how the study of garbage clusters shifts the emphasis from allocation to disposal.

We strongly believe that the following key findings in this paper will form the basis for new garbage collection algorithms:

- Most garbage clusters are small, thus reducing the significance of key objects.
- Most garbage clusters die in relatively few methods.
- Memory flows are highly predictable. That is, most of the objects created at a given allocation site die at one of only two or three program points (in many cases, only one).
- Most cyclic garbage clusters are composed of a small number of types and objects.
- Most objects and garbage clusters die by stack action. The majority of these objects were reachable from the heap before dying.

We believe that collectors that exploit these findings should be able to achieve performance gains over more traditional collectors that only exploit the allocation side of the program. The oracular CAG collector is such an example, where

CAG collection showed improvements in low memory situations for some benchmarks. Through further refinement of the design, we hope to achieve a practical design that can be implemented in a real JVM. Another avenue for future work involves improving reference counting algorithms by exploiting the cyclic tendencies of programs.

References

1998. SPECjvm 1998 benchmark. http://www.spec.org/jvm98. (1998).

2005. SPECjbb 2005 benchmark. http://www.spec.org/jbb2005. (2005).

2016. Java Platform, Standard Edition Troubleshooting Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/troubleshoot/memleaks002.html. (2016). [Online; accessed 22-April-2017].

Stephen M. Blackburn, John Cavazos, Sharad Singhai, Asjad Khan, Kathryn S. Mckinley, J. Eliot, B. Moss, and Sara Smolensky. 2001. Pretenuring for Java. In *In Proceedings of SIGPLAN 2001 Conference on Object-Oriented Programming, Languages, and Applications*. ACM Press, 342–352.

Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. 2004. Myths and realities: The performance impact of garbage collection. *ACM SIGMETRICS Performance Evaluation Review* 32, 1 (2004), 25–36.

Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, and others. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, Vol. 41. ACM, 169–190.

Nachshon Cohen and Erez Petrank. 2015. Data structure aware garbage collector. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. ACM, 28–40.

Cody Cutler and Robert Morris. 2015. Reducing Pause Times with Clustered Collection. In *Proceedings of the 2015 International Symposium on Memory Management*. 131–142.

Sylvia Dieckmann and Urs Hölzle. 1999. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *ECOOPfi99fiObject-Oriented Programming*. Springer, 92–115.

Barry Hayes. 1991. Using Key Object Opportunism to Collect Old Objects. *SIGPLAN Not.* 26, 11 (Nov. 1991), 33–46.

Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. 2006. Generating Object Lifetime Traces with Merlin. *ACM Transactions on Programming Languages and Systems* 28, 3 (May 2006), 476–516.

Martin Hirzel, Amer Diwan, and Matthew Hertz. 2003. Connectivity-Based Garbage Collection. In *OOPSLA*. ACM Press, 359–373.

Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. 2002. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management*. 36–49.

Richard Jones, Antony Hosking, and Eliot Moss. 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management* (1st ed.). Chapman & Hall/CRC.

Richard E Jones and Chris Ryder. 2008. A study of Java object demographics. In *Proceedings of the 7th international symposium on Memory management*. ACM, 121–130.

Nathan P Ricci, Samuel Z Guyer, and J Eliot B Moss. 2013. Elephant tracks: portable production of complete and precise gc traces. *ACM SIGPLAN Notices* 48, 11 (2013), 109–118.

Darko Stefanovic. 1999. *Properties of age -based automatic memory reclamation algorithms*. Ph.D. Dissertation. The University of Massachusetts Amherst.

Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. 1999. Age-based Garbage Collection. *SIGPLAN Not.* 34, 10 (Oct. 1999), 370–381.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

David Ungar. 1984. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *SIGPLAN Not.* 19, 5 (April 1984), 157–167.