Program Structure Aware Garbage Collection

Raoul Veroy
Graduate student

rveroy@cs.tufts.edu / Tufts University
Advisor: Samuel Guyer
ACM Student number: 3748596

1. Problem and Motivation

This work seeks to improve garbage collectors by using knowledge about how and where objects become garbage to inform our decision to perform a collection. With the explosion of *Big Data*, the size of the heap is increasing to the point that full collections can be catastrophic for the running application. Because current modern collectors are typically implemented as generational tracing systems, this forces the collector to do a full collection on a large heap. The resulting pause time from a full collection can be unacceptable for the user. The ever increasing size of the live heap opens an opportunity for a collector that can trade throughput for reduced pause times. The *program structure aware* (PSA) collector attempts to do this by incurring more incremental collections so as to lessen the work for full collections. The important question is: how can we schedule these additional incremental collections so as not to accidentally degrade overall performance of the user application?

2. Background and Related Work

Many of the performance improvements in garbage collection have resulted from adapting garbage collection algorithms to various program attributes. One of the most successful examples of this is generational collection, which exploits the weak generational hypothesis that most objects die young (Lieberman and Hewitt 1983; Ungar 1984). Clustered garbage collection attempts to reduce pause times and total collector work by exploiting the connectivity of object clusters (Cutler and Morris 2015). Data structure aware garbage collection uses hints from the programmer to inform liveness decisions (Cohen and Petrank 2015).

There have been prior work on immediate reclamation of garbage. A straightforward implementation of reference counting reclaims garbage as soon as it becomes unreachable (Collins 1960). Free-me used a static analysis to determine where to safely insert calls to free newly formed garbage (Guyer et al. 2006).

In a previous paper, we studied how and where objects become garbage (Veroy and Guyer 2017 - under review) by determining the exact program action that caused the objects to become unreachable. Objects can die in three ways: by losing a heap reference (*dying by heap action*), by losing a stack reference (*dying by stack action*) or when a program terminates (*dying at program end*). Thus, we know which objects die as a result of losing a stack reference (versus losing a heap reference), and in which methods these events occur.

My goal for *Program Structure Aware Garbage Collection* will exploit the following program tendencies (Veroy and Guyer 2017 - under review):

1. Most objects die by stack action. See figure 1.
2. For most programs, objects die in only a handful of methods.

The PSA collector will exploit these tendencies by using function exits as a sign to start a collection.

Reference counting collectors represent the one end of the garbage collection spectrum where garbage is reclaimed immediately in a naive implementation. On the other end, modern extensions to the naive reference counting algorithm delays reclamation of garbage to improve performance. Most modern tracing collectors wait for an allocation failure to trigger a collection. The PSA collector proposes to offer a tradeoff where collections happen more frequently, but pause times are reduced. By amortizing the costs of collection over the run of the program, the chances of a catastrophic full collection will be greatly reduced.

3. Approach and Uniqueness

Most tracing collector implementations trigger a collection only when there isn't enough space for an allocation request. Even generational collectors, which can perform an incremental collection on the young nursery, will only do so when the nursery can not fulfill the current allocation request. If the nursery collection is unable to satisfy the allocation request, generational collectors will perform a full collection.

The question I would like to answer is this: Given what we've discovered about how objects tend to become garbage through some stack action, is there a better point in time in the program to trigger a collection? That is, would it be beneficial for some programs to trigger the collection before the heap is exhausted? I propose that there are more useful program events that can signal when to start a collection.

Previously, stack allocation-based collectors have attempted to exploit the stack discipline nature of programs (Gay and Steensgaard 2000; Choi et al. 2003; Shankar et al. 2008). These implementations usually used escape analysis to identify objects that can be safely allocated on a stack instead of the heap. On the other hand the key idea in the PSA collector is to trigger collection using function exits, so there is no need for stack allocation. While there is no need for stack allocation, the PSA collector would be compatible with the mentioned static escape analyses.

[Copyright notice will appear here once 'preprint' option is removed.]

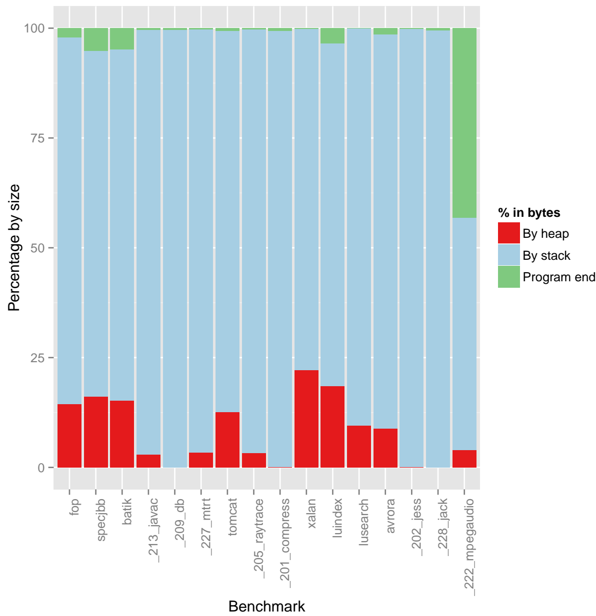

Figure 1. Objects' cause of death classified, in percentage of memory size, as dying by heap, dying by stack action, and dying at program end. Benchmarks are sorted from largest maximum live size to smallest.

3.1 Program Structure Aware Garbage Collection

The program structure aware (PSA) garbage collector aims to take advantage of the tendency of most objects to die because of a function returning. My current research on the PSA collector can be divided into the following steps:

1. Using the dynamic analysis in our previous work, we can identify the top functions where a lot of objects die at the end of the function (Veroy and Guyer 2017 - under review). The PSA collector will use these functions to trigger a collection. I plan to implement this in the Jikes RVM for the mark-sweep and copying collectors (Alpern et al. 2005).
2. If objects die at the end of the same function, then the PSA collector can incrementally threaten only those objects that may die in that function. This suggests that allocating these objects into the same region will improve collector performance. Therefore, in addition to the Jikes RVM collectors, I also plan to test the PSA hypothesis in the Garbage First collector for the Hotspot JVM (Detlefs et al. 2004).
3. Reference counting collectors could also be improved using the PSA hypothesis. All reference counting collectors need a backup system to detect cycles. These backup systems tend to run periodically in a manner similar to tracing algorithms. As such, the backup cycle detectors can also be triggered like the tracing collectors.

3.1.1 Design Points

I plan to run experiments acorss the following design points in the PSA collector:

Trigger points I plan to identify the best functions to use for the PSA collection trigger points. These should be the functions where most objects die.

Collection frequency Triggering on every function return for the chosen trigger points may negatively affect program performance. PSA collection frequency should be tuned depending on the application.

Note that these design points do not have to be statically defined. The collector may employ heuristics to tune the design points dynamically while the user application is running.

4. Results and Contributions

I plan to run the PSA collector on well-known Java benchmark suites: SPECJVM98, DaCapo, and SPECjbb. I also plan to run the PSA collector on large ($> 64GB$) heaps. While the key metric will be pause times, I fully expect the PSA collector to improve overall runtimes for programs with large heaps.

While initial experiments will be run on Jikes RVM mark-sweep collector, the PSA paradigm should be applicable to both generational and region collectors. Thus I expect to be able to experiment on the Jikes generational reference counting collector and the Hotspot garbage first collector.

The essential contribution of this work will be to add a new technique to the garbage collection toolbox. I envision this to be useful to both researchers and industry implementors.

References

B. Alpern, S. Augart, S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen, J. E. B. Moss, T. Ngo, and V. Sarkar. The jikes research virtual machine project: Building an open-source research community. *IBM Syst. J.*, 44 (2):399–417, Jan. 2005. ISSN 0018-8670. doi: 10.1147/sj.442.0399. URL `http://dx.doi.org/10.1147/sj.442.0399`.

J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 25(6):876–910, 2003.

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

G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, 1960.

C. Cutler and R. Morris. Reducing pause times with clustered collection. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15. ACM, 2015.

D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04, pages 37–48, New York, NY, USA, 2004. ACM. ISBN 1-58113-945-4. doi: 10.1145/1029873.1029879. URL `http://doi.acm.org/10.1145/1029873.1029879`.

D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction*, pages 82–93. Springer, 2000.

S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *ACM SIGPLAN Notices*, volume 41, pages 364–375. ACM, 2006.

H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. *ACM Sigplan Notices*, 43(10):127–142, 2008.

D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, Apr. 1984.

R. L. Veroy and S. Z. Guyer. Garbology: A study of how java objects die. 2017 - under review.