Short Presentation: Incremental Copying Collection with Pinning (Progress Report)

Daniel Spoonhower
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217
spoons@cs.cmu.edu

Guy Blelloch
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217
blelloch@cs.cmu.edu

Robert Harper
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217
rwh@cs.cmu.edu

Abstract

Inspired by work in semi-conservative collection, we have implemented a mostly-copying collector for an object-oriented language, including support for object pinning. Our collector efficiently manages fragmentation by measuring *page residency* and determining where copying can be most effective. This work in progress will form the foundation of our ongoing work in real-time collection.

1 Introduction

While garbage collection is an accepted mechanism for memory management, the performance of a collector remains highly dependent on the collection strategy employed in its implementation. A fundamental decision is the choice between copying and non-copying collection, and this decision is constrained by both semantic and performance requirements. Taken in their pure forms, these two techniques complement each other in their benefits and drawbacks.

Copying collectors provide compaction, generally improve memory locality, and offer both better asymptotic running time and a means to bound pause times. However, pure copying collectors are not particularly well-suited to large or long-lived objects and may not be applicable for all languages (*e.g.* those with ambiguous pointers or object pinning).

In contrast, non-copying collectors reduce fragmentation during allocation but make no effort to improve locality, and because they must scan the entire heap for free space, they make it difficult or impossible to bound pause times. They are biased toward larger and longer-lived objects and are compatible with language features such as object pinning. They are also able to operate with a smaller memory footprint.

A number of attempts have been made to bridge the gap between these two extremes. The most familiar examples are generational collectors [11, 1]. Generational collectors segregate objects by age in order to reduce pause times and to avoid copying long-lived objects. Collectors may also maintain separate heaps [12], applying the most suitable collection technique to each heap. More recent work [2] suggests that the collector should copy objects only when fragmentation exceeds some threshold.

We observe that *mostly-copying* collection [3, 9] is an appropriate framework for addressing these trade-offs. While previous work using this technique has been limited to semi-conservative collection, it is equally applicable to the other semantic and performance issues raised above. This framework captures the behaviors of the approaches described above by offering flexibility in how to promote heap objects. In addition, we believe that by measuring *page residency*, our mostly-copying collector will more readily adapt to changing program usage patterns.

We have implemented a mostly-copying collector for C#, an object-oriented programming language [8], that provides uniform support for pinned and large objects and that uses an estimate of page residency to limit fragmentation. Our preliminary results show that our collector offers performance comparable to a generational collector. Mostly-copying collection will provide a foundation for our continuing work in real-time collection.

2 Algorithm

Copying collection can be used to give hard bounds on the length of pause times in terms of the amount of live data [7]. However, pure copying collectors are incompatible with ambiguous roots and with the semantics of pinned objects. In extending previous work in real-time copying collection, we applied a technique used in conservative collection to support both object pinning and a limited form of copying. This technique can then be extended to other circumstances where copying collection performs poorly, notably for large and long-lived objects.

Mostly-Copying Collection Bartlett introduced *mostly-copying* collection [3] as a mechanism for semi-conservative collection. While previous work in conservative collection has focused on purely non-copying techniques [6], mostly-copying collectors relocate objects whenever possible, reserving non-copying techniques for those roots whose types are unknown.

Unlike pure copying collectors, Bartlett's mostly-copying collector divides the heap into a large number of pages instead of two semi-spaces. The distinction between the *from-space* and the *to-space* is then made logically rather than as a fixed range of addresses. Seen abstractly as sets of pages, the *from-* and *to-spaces* are not necessarily contiguous regions of memory and may change over time.

Individual objects may be promoted (as in an ordinary semi-space collector) by copying the associated data and updating any references to the objects, or an entire page may be promoted by removing the page from one set and adding it to the other. Roots whose types are unknown are promoted using the latter technique, and therefore roots that are not pointers will not be erroneously changed during collection.

Pinned Objects To support a robust foreign function interface, many modern languages require that objects passed beyond the scope of the garbage collector are *not* moved during collection. Objects may also be *pinned* in languages (such as C#) that support address arithmetic.

In C#, only root objects are pinned, and these pinned roots may be enumerated at the beginning of each collection. We expect pinned roots to be rare and recognize that pathological use of pinning would certainly defeat our efforts for efficient collection.

Unlike previous applications of mostly-copying collection, our collector has complete knowledge of the types of values on the program stack. However, each pinned root must be held at a fixed address in the heap for one or more collection cycles. Just as Bartlett's collector promoted pages where the type of a root was unknown, ours does so to promote pinned objects (logically) without changing their addresses (physically). Our collector provides fast allocation, as well as support for pinned objects while limiting heap fragmentation.

Note that for programs that do not utilize object pinning, a collector with this design behaves as an ordinary semi-space copying collector would behave.

Large Objects Many collectors distinguish large objects both because they are more expensive to copy and because they are expected to survive longer. Collectors may more aggressively tenure large objects or manage them a separate region of memory with a different mechanism [12].

Previous implementations of conservative mostly-copying collection recognized that there was little or no benefit in copying objects that consumed most or all of a page. Note that the survival rates of large objects are less important than their impact on fragmentation. Collectors should avoid copying large objects, not only because they are expensive to copy, but because there is little or nothing to be gained in doing so. This trade-off between fragmentation and the cost of copying that led us to a more general application of mostly-copying collection, described below.

Page Residency Pages containing large objects are exemplary candidates for page promotion because the collector can determine *a priori* that there is little risk of fragmentation in doing so. Ideally, the collector would promote only those pages where the cost of copying reachable objects outweighs the otherwise resulting fragmentation.

The term *residency* is used to describe the density of reachable objects, and in the context of mostly-copying collection, we will consider *page residency*. Using this terminology, we can say that if the residency of a page is sufficiently high, the page should be promoted. If the residency is not high enough then any reachable objects on the page should be relocated.

Seen in this light, we can understand that large objects should generally remain in fixed positions in the heap, not simply because they are large, but because they occupy pages with high residency.

Our collector cannot, however, determine the residency of a page in general without first traversing the entire heap. Instead, it estimates the residency of each page using the following heuristic: it approximates the current residency by using the residency at the end of the *previous* collection cycle. (We assume that the residency of any page allocated by the mutator between the current and the previous collection cycles is zero.)

Implementation Our collector maintains sets of *free* and *used* pages using linked lists and performs allocation by removing pages from the *free* list and then incrementing a free pointer (as in Smith and Morrisett's implementation [9]). During collection, the *from-* and *to-spaces* are maintained similarly.

Objects are copied using a Cheney scan. In addition to the familiar pair of pointers, our collector maintains a list of gray pages (the gray portion of the heap need not be contiguous). In the case of page promotion, objects are both marked and added to a mark stack. For promoted pages, we tally the residency as we set each mark bit. In the current implementation, pages with an estimated residency less than 75% are scavenged.

We have implemented our collector as part of the Shared Source Common Language Runtime (SSCLI or "Rotor"). In addition to pinning, our collector supports a number of other features required by the C# run-time, including finalizers, weak references and interior pointers.

3 Analysis

As noted above, if the collector *never* promotes pages then it will behave exactly as a semi-space copying collector. Conversely, if the collector promotes *all* pages, it mimics the behavior of a mark-and-sweep collector.

Comparison to Generational Collection Bartlett [4] showed how mostly-copying collection could be extended to perform generational collection with the addition of a remembered set [11].

The configuration of a generational collector (the number and sizes of generations) not only controls the length of minor collections but also determines how the collector will respond to different distributions of object lifetimes. For example, choosing a small nursery results in shorter minor collections. However, decreasing the size of the nursery also increases the risk that the collector promotes objects to the older generation immediately before they become unreachable. In a generational collector, there is no mechanism to retract the decision to promote an object; all promoted objects are treated uniformly. This risk can be mitigated by increasing the number of generations or by dividing each generation into *increments* [5].

These increments are reminiscent of our pages, but are still constrained by a linear ordering based upon age. If the behavior of the mutator results in significant shifts in memory usage (where many, but not all older objects become unreachable), generational collectors will be required to perform full collections. Using residency, we believe our collector will adapt to changing program behavior by scavenging only those pages which are sparsely populated (regardless of age).

Worst-Case Behavior It should be clear that when our estimate of residency is accurate, our collector performs well, copying from pages that have become fragmented over time. In the worst case, all but a small number of objects on a page survive exactly one collection cycle and become unreachable immediately after that cycle completes; though this results in a high error in the residency estimate, our algorithm naturally recovers in one round.

Taken in the extreme, we could assume that the same fate befalls

Table 1. End-to-end time (in seconds). The left column shows times for a generational collector, the right for our mostly-copying collector using an estimate of page residency.

Benchmark	Rotor	Mostly-Copying
huffman	15.891	17.563
xml	39.906	39.829
splay	74.141	72.922
jsc	4.641	4.656

all pages in the heap: despite high residency at the end of the first collection, all pages contain a small number of survivors at the beginning of the second collection. Note that in the second cycle, the collector is required to perform very little work: it must only traverse the memory graph, marking live objects. During this second cycle, the collector discovers that each of these pages should be scavenged in the third cycle, a cycle that will also terminate relatively quickly (as copying collection takes time proportional only to the amount of live data).

We noted that residency could be measured precisely by making an additional traversal of the heap, but rejected this possibility, claiming that it was too expensive. The collector will sometimes simulate this behavior *but only in situations where its estimate is wrong.* That is, the second and third cycles, taken together, perform a step in which the collector recognizes the errors in its estimates and corrects them.

Measurements To motivate further investigation of our ideas, we have performed several simple performance measurements. We measured the total application and collection time consumed by several benchmarks for the original Rotor collector and our mostly-copying collector (using estimated residency). Each test was performed five times; the best times are shown in Table 1.

The Rotor collector is a generational collector and copies data from the nursery on minor and major collections, while using a mark-and-sweep technique on older objects in major collections only. At this stage in our implementation, a comparison with a generational collector is an inadequate measure of performance. Our collector currently performs no incremental collections, delaying collection as long as possible. Our collections are less frequent but more time-consuming, since the entire memory graph is traversed during each collection.

The left column of Table 2 shows the percentage of surviving data that our collector determines is stable and suitable for page promotion. The right column shows the accuracy of the residency estimate; larger percentages indicate greater changes to the stable set over time.

The splay tree benchmark consists of a series of random root insertions on a splay tree, interleaved with truncations of the tree. During each truncation, we remove any nodes greater than some fixed depth d. Object lifetimes in this example follow a lognormal distribution (a distribution observed in many applications [10]) and the shape of this distribution varies with d. The other benchmarks include Huffman encoding an image, parsing an XML document, and compiling a series of 37 JScript files. The XML parser uses a significant number of pinned roots.

4 Future Work

Our strategy closely parallels the work of Bacon *et al.*[2]. As we have pointed out above and as they have noted, some form of compaction is necessary in any real-time collector, but it must be applied sparingly to avoid recopying objects unnecessarily. One focus of our future work will be a more detailed comparison with this work.

Table 2. Effectiveness of promotion strategy. Percentage of surviving data promoted by page promotion and the error in the estimate of page residency (as a percentage of page promoted data).

Benchmark	Page Promoted	Estimate Error
huffman	90.03%	0.04%
xml	51.89%	10.36%
splay	70.25%	11.86%
jsc	24.95%	*

*Only one collection cycle occurs

In addition to performing a more thorough analysis and carrying out a more complete set of experiments, we plan to extend our current collector to support incremental collection with bounds on the amount of time and space consumed by the collector. We also plan to demonstrate the sensitivity of different collector algorithms to their configuration parameters using programs such as our splay tree benchmark.

5 References

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

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–298. ACM Press, 2003.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, DEC Western Research Laboratory, February 1988.

[4] J. F. Bartlett. A generational compacting garbage collector for C++. In *ECOOP/OOPSLA Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Canada, 1990.

[5] S. M. Blackburn, R. E. Jones, K. S. McKinley, and J. E. B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation, PLDI'02, Berlin, June, 2002*, volume 37(5) of *ACM SIGPLAN Notices*. ACM Press, June 2002.

[6] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, 1988.

[7] P. Cheng and G. Blelloch. A parallel, real-time garbage collector. In *Proc. 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 125–136, June 2001.

[8] A. Hejlsberg, S. Wiltamuth, and P. Golde. *The C# Programming Language*. Addison-Wesley, 2003.

[9] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. *ACM SIGPLAN Notices*, 34(3):68–78, 1999.

[10] D. Stefanovic, K. S. McKinley, and J. E. B. Moss. On models for object lifetime distributions. In *International Symposium on Memory Management*, October 2000.

[11] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167. ACM Press, 1984.

[12] D. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 1–17. ACM Press, 1988.