Rethinking age-based garbage collectors

(White Paper for Graduate Student Workshop, DARPA Information Technology Office, July 1998)

Darko Stefanović
(joint work with J. Eliot B. Moss and Kathryn S. McKinley)
Department of Computer Science
University of Massachusetts
Amherst MA 01003

June 29, 1998

1 Introduction

Garbage collection has been around for a long time, and is accepted as a customary element for dynamic memory management in run-time systems of modern programming languages, for reasons of safety and for its clear software engineering benefits. The wide-spread acceptance of object-oriented languages makes it more important than ever to have garbage collection also work as fast as possible.

The state of the art in garbage collection on stock hardware are multi-generational copying collectors. We have ourselves developed a flexible (configurable) and efficient language-independent garbage collection toolkit based on this paradigm, and have used it in Smalltalk, Standard ML, and Java.

But we recently discovered better ways to collect than generational collection, approaching in some cases optimal performance according to our simulation-based study of programs from object-oriented languages.

2 Current beliefs

There are two real advantages for which generational collectors are preferred over original non-generational collectors: pause times are shorter (on average), and total effort (copying cost) is lower.

There are, however, three tenets that have become commonly held as explanations why generational collectors have these advantages, and that have discouraged exploration of alternative strategies.

The first has to do with a property, empirically confirmed in object-based systems, that young object die fast, more precisely, that mortality is a decreasing function. The popular wisdom is that generational collectors exploit this property by concentrating effort on collecting young objects.

Whenever only a part of the heap is collected at once, the remainder must be assumed live for the purposes of the collection, and all pointers from outside the region collected into it are roots of the collection. This approximation is conservative, and finds more data live than is actually the case, and thus copying cost is greater. The second tenet is that this excess copying effort is small in generational collection because pointers from outside the region and into it are, by construction, pointers from older objects to younger objects, and such pointers are few. The third tenet is that for the same reason it is easy to maintain the write-barrier (actions taken during the execution of the program when a pointer is stored) and remembered sets (the sets of pointers crossing into a region).

Taken together, these three tenets do more than explain the good performance of generational collection. They also rule out using any strategy that collected older objects in preference to younger—it would "concentrate effort" on regions of lower mortality. It would also have many incoming pointers from outside, thus from younger objects, and have both too much excessive copying and high maintenance cost for remembered sets. Our study, however, shows that the three tenets are not necessarily always valid, and that alternatives including older-first collection are feasible.

3 Current beliefs questioned

First, the intuition that effort should go to regions with high mortality is countered by the intuition that within any given area, its older half has less live data than its younger half, so it it better to go for the older half. And indeed, in the beginning of our work we discovered that a variant of oldest-first collection worked better than generational (youngest-first) collection. We should point out that, even though a decreasing mortality is attested in all measured systems for relatively young objects, the behavior for relatively older objects, and certainly for very long-lived objects in long-running programs, remains unexplored.

Second, the excess copying: we empirically examined the difference in copying cost between a collector which assumes that everything outside the collected region is live, and one which has an oracle to tell apart the live from the dead. We found that, as long as the schemes preserved the objects in a logical order of allocation (thus always having data of a contiguous age group in the region collected), the excess copying was not excessive. We then measured the statistics of pointer distances (difference in age between the object containing the pointer and the object pointed to) and found that, on the one hand, neither older-to-younger nor younger-to-older direction dominates across all studied traces (Smalltalk and Java). But, more important, we found that most observed distances are very small indeed, be they positive or negative. Notice that while these properties work in favor of generational collection they also work in favor of any other age-order-preserving collection scheme.

Third, the implementation overheads: although the ultimate test must be in implementation, we can argue that the overheads of an oldest-first scheme need not be greater than a youngest-first, generational, scheme. Current state-of-the-art collectors implement the write-barrier using card marking—a bit-map indicates which block of memory has had a pointer stored into it. It is most efficient to keep the write-barrier code as simple as possible, record all pointer stores in the card table, and later scan the table and inspect pointers to find if they actually cross generation boundaries. Even if younger-to-older pointers predominate, the card-marking write-barrier itself is unaffected. As for scanning the card table, generational collectors have to scan the cards corresponding to older generations, typically a large part of the heap space, therefore scanning the parts corresponding to younger regions is feasible just as well. Lastly, the sizes of remembered sets need not be enormous. The foreseeable implementations would group contiguously allocated or copied objects into blocks, and since blocks will always be wholly within or wholly outside the region collected, there is no need explicitly to represent the pointers that do not cross block boundaries. As long as objects in a block are of a similar age, most of their pointers will remain within the block, as our pointer distance measurements show, thus only the remaining few need to be stored as inter-block remembered sets.

4 Towards new collection schemes

The above results motivated us to explore other collection schemes among those that preserve the relative order of objects. We chose to undertake simulation studies before an actual implementation, because thus we have greater flexibility in choosing parameters, and we can even explore the limits of achievable performance using oracles of varying strength.

We examined the obvious opposite of youngest-first collection, which is a strict oldest-first: choosing a fixed amount of oldest data. It sometimes works well, but if the amount is small, the collector works poorly. This is similar to a strict youngest-first collector, which always chooses a fixed amount of youngest data (i.e., the nursery). It too works poorly, or even fails.

These results led us to explore strategies that allow the collected region to be chosen more flexibly, so that both older and younger regions are visited. A deferred older-first collector does it in the simplest possible manner, by moving a window of collection across the heap, but looking at older objects first. We found it to perform remarkably well, often close to the performance of a locally-optimal scheme (one in which the best region to collect is chosen by trying all possible regions, not something one would or could do in implementation). The intuitive reason for this good performance is also remarkably simple: as the window of collection moves towards younger objects, it finds itself at some point corresponding to an age at which objects commonly die; and, since few objects survive in the window, the window does not move very fast. Thus, this collector homes in on the region where it is most profitable to collect, and stays there for a long time. A welcome property is that the best configurations observed in our experiments are those with a small window size, and, since the amount copied per collection is limited by window size, we see that low total copying cost is accompanied by predictably short pause times.

5 Basic Bibliography

Richard Jones and Rafael Lins: ”Garbage Collection: Algorithms for Automatic Dynamic Memory Management”, John Wiley, Chichester, 1996.
Antony L. Hosking, J. Eliot B. Moss and Darko Stefanovic: ”A Comparative Performance Evaluation of Write Barrier Implementations”, in OOPSLA, 1992.