Oldest-First Garbage Collection

Computer Science Technical Report 98-81

Darko Stefanović J. Eliot B. Moss Kathryn S. McKinley

Department of Computer Science

University of Massachusetts

Revised, April 1998

__Note.__ Some of the information in this document is superseded by later results. In particular, we now view the oldest-first collection described in this report as a special case of renewal-age oldest-first collection. We also believe that excess promotion of objects can be a significant factor for performance in practice, but it is not modelled in this study. This circumstance diminishes the utility of mathematical analyses and simulations based solely on object lifetimes. April 1998.

Abstract. An oldest-first generational garbage collector leaves intact the most recently allocated object space, and instead collects the remaining, older objects. Because these older objects have had more time to die, an oldest-first copying collector will generally do less copying than a traditional generational collector (which operates youngest-first), a non-generational collector, and even Clinger and Hansen's non-predictive collector (which does wait a little while for objects to die). To explore the performance of oldest-first collection, we present a mathematical analysis, simulation results for a variety of mature object lifetime distributions, and simulation results for mature object lifetimes drawn from real programs and for real mature object traces. These results demonstrate that oldest-first collection does perform significantly better than youngest-first or non-generational collection for mature objects. Although some previous work pointed in this direction, it provided very little evidence of our conclusion. We also find that oldest-first collection works well for lifetime distributions that satisfy the generational hypothesis, which suggests we should also consider oldest-first collection in the young object space.

1 Introduction

Garbage collection, automatic dynamic memory management, frees programmers from explicitly managing data allocation and reclamation. Its software engineering benefits are well known, but its influence on performance is a subject of debate. To date, the best-performing garbage collectors in wide use are generational copying collectors. According to common wisdom, these collectors rely on the fact that the youngest objects die quickly, and concentrate their collection efforts on them. For longer-lived objects, there is little evidence that the younger ones die more quickly than the older ones. Nevertheless, we show that, if collection cost is determined by object lifetimes, then a non-standard generational "oldest-first" collector can efficiently collect these longer-lived, mature objects, and in general, objects that do not die quickly.

Figure 1 illustrates a collector that divides objects into young and old, using a traditional generational collector for the young space, and a mature space collector. The collector promotes objects that survive the oldest generation of young space into the mature space. In this paper, we consider three alternative organizations for the mature space collector: (1) non-generational, (2) youngest-first, and (3) oldest-first. The non-generational collector simply collects the entire space every time it fills up. It serves as a base line for comparison. The youngest-first collector is a traditional two-generation copying collector. It collects the young objects every time the heap fills up. The oldest-first collector instead collects the old objects every time the heap fills up.

To explore these alternatives, we study a variety of object allocation traces which exhibit different lifetime distributions. We analyze the three collectors mathematically and through simulation using widely different object lifetime distributions, and show that the oldest-first collector is superior to the others. We take the lifetime traces for mature objects from a number of analytical distributions and distributions built from actual programs, and we simulate the collectors on actual mature object traces. We are not aware of any other work that reports accurate object lifetime information for mature objects.

The remainder of the paper is organized as follows. We first discuss related work that prompted our effort. Section 3 then defines the object lifetime terminology. Section 4 provides a simply motivating example for why oldest-first should perform well. We then present a set of analytical object lifetime distributions we use to study the three collectors in Section 5. The remaining sections describe collection space and time costs, and provide mathematical and simulation analyses of these costs for the three collectors, including their behavior on actual mature object traces.

2 Related work

The primary related work is a recent study by Clinger and Hansen [Clinger and Hansen, 1997]. They considered the behavior of heaps under the exponential distribution (radioactive decay model), and proposed a non-predictive collector. Rather than collecting the youngest data like a traditional generational collector, the non-predictive collector processes the *youngest* portion of the

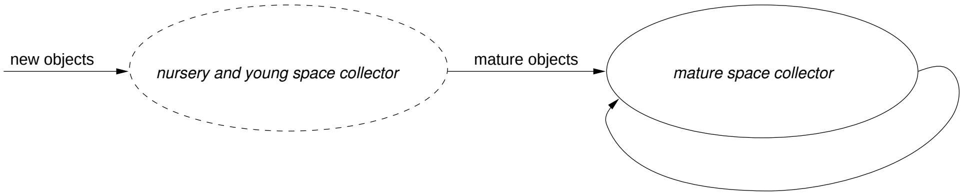


Figure 1: Structure of collection.

older generation, so it differs from our oldest-first collector, which processes the *oldest* portion. Because oldest-first copies the oldest portion it will do less copying than non-generational, youngest-first, or Clinger and Hansen's non-predictive collector, for *any object survival function*. Clinger and Hansen presented their results as being confined to the exponential distribution.

Because the analysis is simpler, Clinger and Hansen only explored values of $g < 0.5$, where g is the fraction of the heap *not* collected at the next collection. We extend the analysis to the full range $0 < g < 1$ and find that values of g close to 1 perform well. Other ways in which we go beyond Clinger and Hansen's work are: we analyze youngest-first collection; we consider other analytical distributions; we simulate collector behavior with mature object distributions determined from actual traces; and we simulate collector behavior on actual mature object traces.

There have been a number of papers about object lifetimes and the generational hypotheses; Clinger and Hansen offer a good overview of these [Clinger and Hansen, 1997, Section 9 (pp. 106–107)]. We could not find any recent studies of mature object lifetime distributions that go beyond anecdotal reports, with the signal exception of Hayes' data [Hayes, 1993].

One of us has worked on collection techniques directed at mature objects, the Mature Object Space (MOS) collector, also called the Train Algorithm [Hudson and Moss, 1992]. The objective of that work, though, is not so much the minimization of total garbage collection cost as avoidance of disruptive pauses. Seligmann and Grarup [Seligmann and Grarup, 1995] implemented MOS and found it to work quite well. It both keeps pauses short and keeps heap usage very close to the amount of live data. What makes MOS relevant here is that if we ignore the fact that it will tend to reorganize objects according to their reachability, it is an oldest-first collector for mature objects. Thus the present work provides additional support for the MOS approach.

3 Background and definitions

There is no consensus on terminology for garbage collection analysis, so we define the terms as we use them. The *time* between two events as seen by a collector is expressed as the amount of data allocated in between the events and managed by that collector. For a mature space collector, the time reflects the amount of data allocated *into the mature space*, i.e., promoted out of the young space. The *lifetime* of an object is the time elapsed between the object's allocation and when the object becomes unreachable. Given a sequence of new or mature objects, we characterize some statistical properties of the sequence using survival-related functions.

Following standard practice [Cox and Oakes, 1984, Elandt-Johnson and Johnson, 1980], we describe the distribution of lifetimes using the *survivor function*, the *density function*, and the *mortality (or hazard) function*. If we view the object lifetime as a random variable $\mathbf{T}$, then the survivor function is $s_{\mathbf{T}}(t) = \wp\{\mathbf{T} \geqslant t\}$, and it expresses the fraction of original allocation that is still live after an interval t. It is a monotone non-increasing function. The density function is $f_{\mathbf{T}}(t) = -s'_{\mathbf{T}}(t)$, and it expresses the distribution of object lifetimes. The mortality function is $m_{\mathbf{T}}(t) = \frac{f_{\mathbf{T}}(t)}{s_{\mathbf{T}}(t)} = -\frac{d}{dt}\log s_{\mathbf{T}}(t)$, and it expresses the age-specific death rate. (Subscripts $\mathbf{T}$ are dropped in the following.)

4 Why oldest-first is better: a simple analysis

Consider an arbitrary survivor function $s(t)$, as illustrated in Figure 2. The only certain property of $s(t)$ is that it is a monotone non-increasing function. Suppose that space V is available for allocating new objects, and that following a period of allocation into this space we collect the live objects, copying them elsewhere. The volume of objects collected is $\int_0^V s(t)dt$, or the area under the curve $s(t)$. Collections occur at regular intervals of length V, so the copying cost of collection is proportional to $\frac{\int_0^V s(t)dt}{V}$. If instead we look at only one half of the allocated space, namely

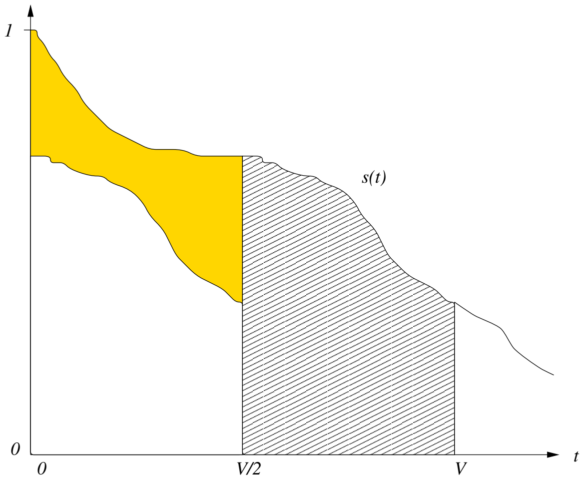


Figure 2: Fundamentals of collection: survivor function.

that allocated earlier, i.e, with greater current age, then the volume collected is $\int_{V/2}^{V} s(t)dt$, or the area hatched in the figure. We can implement this strategy by collecting twice as frequently, and each time collecting not the area $V/2$ just allocated, but the one allocated in the previous cycle. The copying cost of collection is then proportional to $\frac{2\int_{V/2}^{V} s(t)dt}{V}$. Now, because $s(t)$ is non-increasing, it will always be the case that $\int_{V/2}^{V} s(t)dt \leqslant \int_{0}^{V/2} s(t)dt$, and $2\int_{V/2}^{V} s(t)dt \leqslant \int_{0}^{V/2} s(t)dt + \int_{V/2}^{V} s(t)dt = \int_{0}^{V} s(t)dt$. This difference is indicated by the shaded region on the top of the left half of the figure. Collecting twice as frequently a region half as big, but *with postponement*, makes the copying cost lower (or in the worst case does not change it), *regardless* of object lifetimes.

The foregoing reasoning can be generalized to configurations in which a portion smaller than one-half of the volume is considered each time, and postponement is by more than one collection cycle. These configurations promise even lower copying costs, and we use the term *oldest-first collection* for all of them. Hence, we can summarize by saying that oldest-first collection copies less and is thus better than non-generational collection. By similar arguments it is also better than traditional two-generational collection, which repeatedly examines the most recently allocated region, and can thus be termed youngest-first. We now examine how much better it is, and refine the analysis to include also the cost of invoking the collector.

5 Object lifetimes

Generational collection has traditionally been justified and explained using certain hypotheses about the distribution of object lifetimes *in real systems*. The *weak generational hypothesis* states that young objects die fast, i.e., that mortality $m(t)$ is high for small t. The *strong generational hypothesis* states furthermore that even among older objects, the relatively younger ones die faster, i.e., that $m(t)$ is a monotone decreasing function. In the context of these hypotheses, the exponential distribution,[1] for which the mortality is constant, is the boundary case between distributions favorable to generational collection, with $m(t)$ decreasing, and those unfavorable to it, with $m(t)$ increasing [Baker, 1993].

We show that these assumptions are not intrinsically necessary for the efficient operation of a generational collector, but that the form of the distribution does affect the best organization of the collector.

We were interested in analyzing distributions reflective of actual mature object lifetime distributions, but could find no previous studies of mature object lifetimes. Therefore we gathered object allocation traces from 25 long-running programs in Smalltalk and in SML/NJ, and extracted traces of allocation into mature space. Some of the object lifetime distributions satisfy the generational hypotheses and some do not. To cover the space of possible shapes of the mortality function, we chose three representative analytical distributions: the exponential distribution (mortality is constant), the square-root-exponential (mortality decreases with age), and the square-exponential (mortality increases with age).[2]

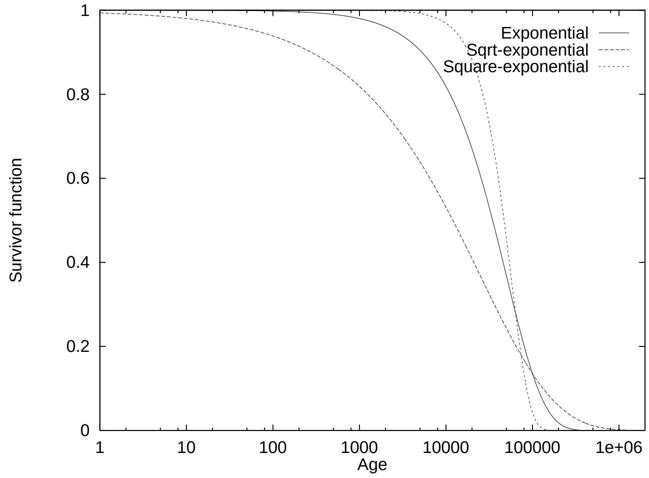
Exponential survival: The survivor function is $s(t) = e^{-\lambda t}$, mortality is $m(t) = \lambda$, and the probability density function is $f(t) = \lambda e^{-\lambda t}$. The mortality is constant for all ages. For this distribution, we also have analytical solutions for collector performance. We used them to validate the simulation procedure.

Square-root-exponential survival: The survivor function is $s(t) = e^{-\sqrt{\beta t}}$, $m(t) = \frac{\sqrt{\frac{\beta}{t}}}{2}$, and $f(t) = \frac{\sqrt{\frac{\beta}{t}}e^{-\sqrt{\beta t}}}{2}$. The mortality is a decreasing function of age. This distribution satisfies the generational hypotheses and corresponds well to the lifetime distributions of young objects in real systems.

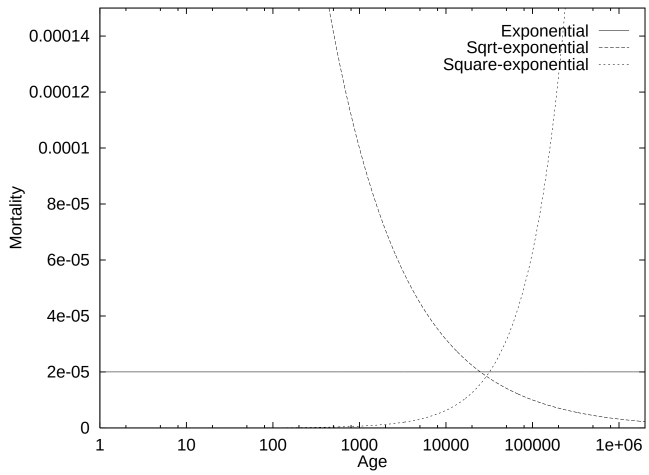
Square-exponential (semi-normal) survival: The survivor function is $s(t) = e^{-(\beta t)^2}$, $m(t) = 2\beta^2 t$, and $f(t) = 2\beta^2 t e^{-(\beta t)^2}$. The mortality is an increasing function of age.

[1] Also known as the radioactive decay model [Baker, 1993, Clinger and Hansen, 1997].

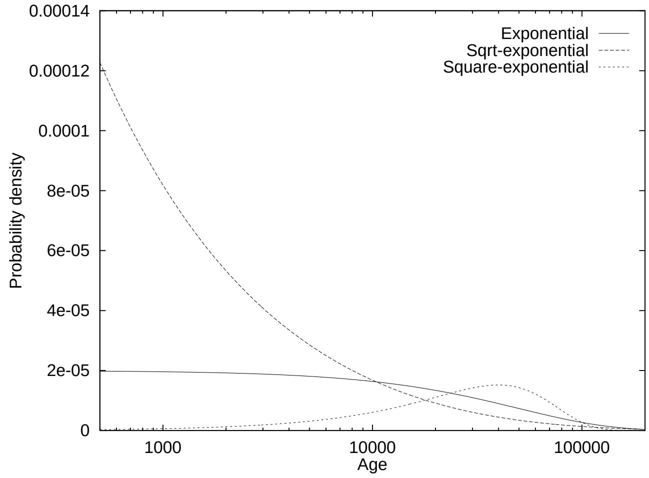
[2] The questions of modelling the observed lifetime distributions using particular analytical distributions are beyond the scope of this paper.



(a) Survivor function



(b) Mortality function



(c) Density function

Figure 3: Analytical distributions.

On opposite sides of the exponential distribution we have: the square-root-exponential with mortality decreasing for all ages, so it should be ideal for youngest-first collection; and the square-exponential, with mortality starting at 0 and increasing for all ages. Figure 3 illustrates these survivor functions, and the dramatic differences in their mortality functions.

In the remainder of the paper, we develop and compare the total cost of garbage collection for each scheme (non-generational, youngest-first, and oldest-first), for the analytical distributions, for real distributions, and for real traces. We make the comparison of the collection cost overheads of different schemes fair by allowing each scheme an equal amount of space overhead. We first develop mathematical formulae describing the time and space cost of collection (Section 6). We continue by describing how each collector works in Section 7 and giving a summary of the mathematical analysis for the exponential distribution in Section 8. We offer simulation results for the analytical distributions and traces synthesized using real distributions in Section 9.1, with results from actual traces in Section 9.2. These results, based on object lifetimes, show that for a broad range of operating conditions, which we expect to be representative both of real workloads and of real collector implementation costs, the oldest-first organization is superior to the youngest-first organization. However, the results based on heap pointer structure show a different and ambiguous picture, as we discuss in Section 10; we present our final conclusions in Section 11.

6 Time and space overheads of collection

Time. The main cost of copying garbage collection is the cost of copying live objects, whether they are copied elsewhere (pure copying) or compacted within the region (sliding). This cost partly depends on the *number* of objects copied, but it is roughly proportional to the total *volume* copied. The *mark/cons ratio*, defined as $\mu = \frac{\text{Volume copied}}{\text{Volume allocated}}$, is a good measure of the copying cost and we use it to compare copying costs. It represents the average number of times an object is copied, and thus measures the time "wasted" by the collector. If a program allocates an amount A, and the cost of *copying* one word is c_c times the cost of *allocating* one word, where c_c is a small number, perhaps in the range 1-3, then the copying cost for the program is $c_c \mu A$.

In an earlier study, we measured the cost of one garbage collection in a deployed system (SML/NJ v0.93) to be $C = 5183 + 81.2w$, where C is the cost in cycles and w is the amount copied in words, and the second term always dominates. The first term, which we call C_S, is the cost of an invocation of the garbage collector. The number of invocations is inversely proportional to the amount of allocation between two collections, i.e., the amount F freed by each collection. The total garbage collection startup cost for a program that allocates an amount A is $C_S \frac{A}{F}$; the total cost is $c_c \mu A + C_S \frac{A}{F}$; and the cost per word allocated is $c = c_c \mu + C_S \frac{1}{F}$.

In addition to these costs, which we model, there are some costs that are beyond the scope of this study. In both analysis and simulation, we assume that perfect knowledge of object reachability is available. In practice, that knowledge is derived by the collector itself by means of tracing. There is clearly an up-front cost of the tracing operation, but in a copying collector that cost can be accounted for as part of the copying cost by suitably increasing c_c. However, tracing in a collector with multiple heap regions depends on the availability of *remembered sets*—records of pointers that cross region boundaries, in particular, pointers into the region collected. The maintenance cost of these sets is paid in part by the collector and in part by the mutator. In addition, whenever a region is not collected, the collector assumes that all data in it are live, and that all pointers emanating from that region and into the region collected are root pointers. Thus, the collector assumes that more data are live than is actually the case, and the copying cost is correspondingly higher than it would be given perfect reachability knowledge.

Space. We use V to denote the volume available for the heap. It is possible in real collectors to vary this size to adapt to the workload, and since using more space tends to reduce the time overhead, in our study we fix V to compare different schemes equitably by giving each the same space constraints. We note that pure copying collection has higher peak space demands than sliding compaction. Here we consider only sliding collection.

When considering a heap in equilibrium, in which the rate of allocation equals the rate of object demise, we use v to denote the steady-state amount of live data.[3] The ratio $L = \frac{V}{v}$ expresses the space overhead of the collector configuration: how many times the available heap space is gretaer than the amount of live data. The smallest possible heap, with no space overhead, would have $V = v$ and $L = 1$. But for any collector organization, as L is made to approach 1, the copying cost μ tends to infinity. In order not to deal with such large values of μ for small L, it is often convenient when visualizing and comparing copying costs to normalize with respect to a non-generational collector, and we define the *relative mark/cons ratio* $\rho = \mu/\mu_{\text{non-generational}}$. The value of this metric will turn out to be just $\rho = \mu(L-1)$, in other words it measures the *time-space product overhead*.

We denote by f the *fraction* of the heap a collection frees: $f = \frac{F}{V}$. The total cost of collection is thus $c = c_c \mu + C_S \frac{1}{fLv}$. Normalizing, we can use the cost $c' = \mu + C_S \frac{1}{c_c fLv} = \mu + \frac{X}{fL}$. Here $X = \frac{C_S}{c_c v}$ captures the cost of collection startup relative to the cost of copying. C_S and c_c depend on garbage collector implementation, and v is specific to the application program. The number of collector invocations differs in the collector organizations we consider, thus relative collector performance depends on the cost of collection startup. We therefore present comparison results with X as parameter for a wide range of values of X. But we must then ask—what are reasonable values of X that one could encounter in practice? Consider our earlier measurements: the cost of copying one word is about 80 cycles, but the cost of allocating one word is about 2 cycles, so $c_c \approx 40$; the cost of startup C_S is about 5000 cycles. Then for a program with a smallish steady-state live amount of 50,000 words, $X \approx 0.0025$. Our comparison results will show that for reasonable values of X and L the best configuration of the oldest-first collector has lower cost than the best configuration of the youngest-first collector.

7 Collector details

We now describe each collector in more detail. These descriptions correspond to the way in which we simulated the collectors' behavior.

Non-generational collector: The total space available is V and after collection i, $\varepsilon_i V$ data is live and $f_i V$ is free. Initially the whole heap is empty: $\varepsilon_0 = 0$ and $f_0 = 1$. Each collection processes the entire space V. It does not matter in which direction objects are compacted or in which direction they are allocated.

Youngest-first collector: The total space available is V. It is divided into a young generation of size gV and an older generation of size $(1-g)V$. Allocation proceeds in the young generation until it is exhausted, which trig-

[3] The notion of a steady state is convenient for mathematical treatment of heap organizations. Moreover, real programs that do reach a quasi-steady state can run long and go through a large number of collections, so the effects of the fortuitous choice of collection instants are reduced.

gers a *minor collection*. The minor collection processes the young generation, copying survivors into the old generation. Eventually the old generation fills up, and the next time the young generation fills, the entire heap is collected, with survivors put back into the old generation. We assume that g is chosen such that the old generation exactly fills up in the steady state. That is, if the old generation "overfills", we increase V to hold the additional survivors, while keeping the young generation's *size* constant (implying g is smaller than first assumed).

Oldest-first collector: We proceed circularly through the space, which we visualize here as proceeding from left to right. Just after collection $i-1$, there is $f_{i-1}V$ space available for allocation (see Figure 4). We allocate from left to right until that space fills. We then choose to collect some part of the area of very old objects just ahead (to the right) of the allocation pointer. In the steady state, that space is $(1-g)V$. In any case, we collect it, sliding $\varepsilon_i V$ survivors to the left and creating $f_i V$ free space. We set the allocation pointer to the left end of that space and we can resume allocation. Note that we will cycle through the entire space in $K = \frac{V}{(1-g)V}$ collections. Note that K need not be an integer, so analysis is intricate.

8 Mathematical analysis for the case of the exponential distribution (summary)

We summarize here the results of the mathematical analysis of the three collectors described in the preceding section.

Non-generational collector: The mark/cons ratio is $\mu_{\text{non-generational}} = \frac{1}{L-1}$ regardless of distribution [Appel, 1987, Jones and Lins, 1996].

Youngest-first collector: Under exponential distribution, the number of minor collections per major collection cycle is $\zeta = \frac{L(1-g)-e^{-Lg}}{1-e^{-Lg}}$. The mark/cons ratio is $\mu_{\text{youngest-first}} = \frac{1-g}{\zeta g}$. The full derivation is given in Appendix A.

Oldest-first collector: Even under exponential distribution the analysis is quite complicated, and the details of the general solution are given in Appendix B. In the simplest case, when K from previous section is an integer, the mark/cons ratio is $\mu_{\text{oldest-first}} = \frac{1-g-f}{f}$, where f is found as the solution of the nonlinear equation: $(1-Kf)L\left(1-e^{-KfL}\right) = Ke^{-KfL}\left(e^{fL}-1\right)$.

Under exponential distribution, and for any configuration, $\mu_{\text{oldest-first}} < \mu_{\text{non-generational}} < \mu_{\text{youngest-first}}$.

9 Simulation results

We produced synthetic object allocation traces, with object lifetimes as independent identically distributed random variables. We normalized the steady-state live amount in the heap, i.e., the expected value of lifetime, to 50,000, and used traces of at least 1,000,000 objects allocated, so that a steady-state is clearly entered. For the three distributions reported here, we have: for exponential, $\lambda = 2\cdot 10^{-5}$, and generation of the synthetic trace lifetimes $\mathbf{T}$ by inversion from a uniformly-distributed random variate $\mathbf{U}$ according to formula $\mathbf{T} = -\frac{1}{\lambda}\ln \mathbf{U}$ (see Ref. [Devroye, 1986, pp. 27ff]); for square-root-exponential, $\beta = 4\cdot 10^{-5}$, and $\mathbf{T} = \frac{1}{\beta}(\ln \mathbf{U})^2$; for square-exponential, $\beta = 1.772454\cdot 10^{-5}$, and $\mathbf{T} = \frac{1}{\beta}\sqrt{-\ln \mathbf{U}}$.

We built simulators for both youngest-first and oldest-first collection, according to the description given in Section 7. Under the assumption that the allocation trace enters a steady state, the simulator reports the steady-state value of the live amount of data, of the amounts collected, mark/cons ratio, frequency of collection, etc. We validated the simulators against the mathematical model for the exponential distribution, by calculating the mark/cons ratio for each collector configuration L, g used in simulation. The average relative error is $e = \sqrt{\sum(\frac{\mu_{\text{simulated}}}{\mu_{\text{calculated}}}-1)^2}$, where the summation is over all configurations. For youngest-first collection, $e = 0.0048$ over 6630 configurations. For oldest-first collection, $e = 0.0076$ over 10587 configurations. With agreement within 1%, we are satisfied that the simulator is accurate.

9.1 Simulation results for analytical distributions

We first present the results of the simulations of analytical distributions varying the configuration parameters L (heap space overhead) and g (fraction considered young). For better readability, we use the mark/cons ratio ρ, normalized to the non-generational collector. We show in Figure 5 the copying cost of youngest-first collection on the left, and that of oldest-first collection on the right. Each plot has the configuration parameter g along the horizontal axis, and includes several curves for selected values of the space overhead parameter L: 1.3, 1.5, 1.7, 2.0, 2.5, 3.0, and 4.0. Note first that the scale on the vertical axis is different on the left and on the right: the curves for youngest-first lie mostly above 1, and those for oldest-first mostly below 1: youngest-first is mostly worse than non-generational, and oldest-first is mostly better. It is only for the square-root-exponential distribution, and only for

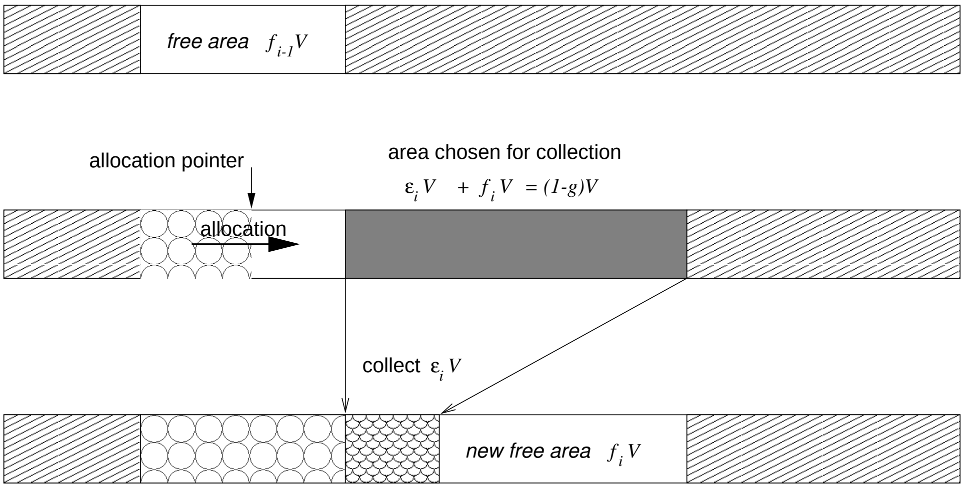


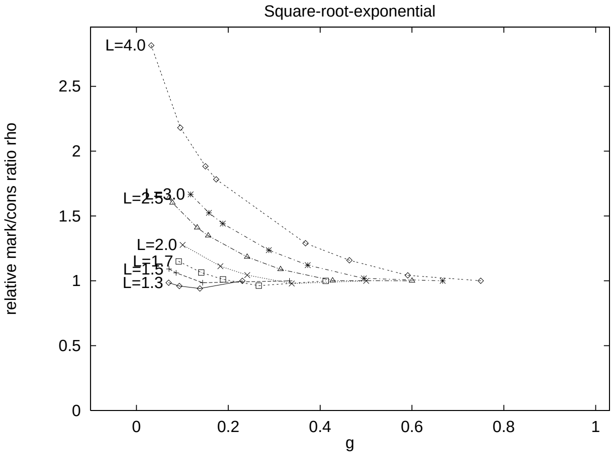
Figure 4: Oldest-first collection.

small values of L and of g that the opposite is true. Second, the relative advantage of oldest-first over non-generational collection grows with more space L. The relative disadvantage of youngest-first against non-generational collection also grows with more space L. Third, for youngest-first collection the cost diminishes with increasing g. As g increases, the number ζ of minor collections between major collections decreases, and the collector behaves more like a non-generational one. Fourth, for oldest-first collection the cost has a downward trend with increasing g; it decreases monotonically for the exponential but non-monotonically for the two other distributions. (The case-analysis in Appendix B provides an intuition for this pattern.) Fifth, and quite surprising, the variance of cost of youngest-first collection for the three distributions is not great. This variance grows with L, but for a modest value $L = 1.5$ for example, the cost on the square-root-exponential distribution is at least 82% of the cost on the exponential distribution (achieved with $g \approx 0.1$), and the cost on the square-exponential distribution is at most 106% (with $g \approx 0.16$).

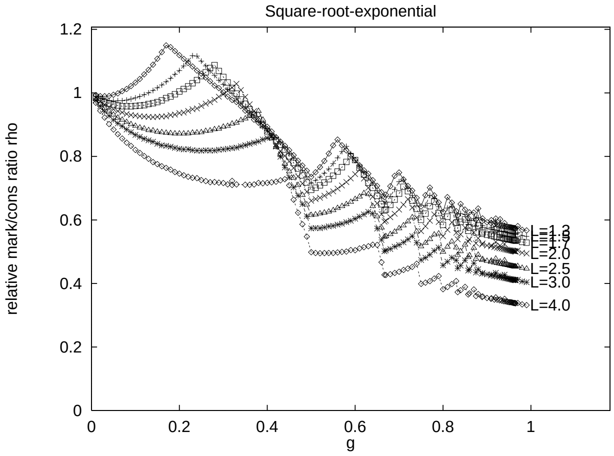
We see that the mark/cons ratio of a collector is heavily dependent on the configuration parameter g. Let us assume, however, that a collector is capable of adapting to the workload by changing its configuration, within a fixed amount of heap space. In other words, we are interested in the best choice of g for a given L. The optimization is done with respect to the total cost measure $c' = \mu + \frac{X}{fL}$. If both the youngest-first and the oldest-first collector achieve their best configuration, then we can fairly compare them, by asking which one has the lower cost for the same space overhead (measured by L) and the same relative cost of collector invocation (measured by X). For the three distributions, the answer is presented in Figure 6. In the region indicated by shading, where the cost of invocation X is high, and in the region of extremely tight heaps where L is close to 1, the youngest-first collector is better (but the absolute performance of either scheme is poor). However, for reasonable values of L, and reasonable values of X, as discussed previously, the oldest-first collector is better. To see how much better, observe the contour lines drawn at 20% increments of the cost ratio $\frac{c'_{\text{youngest-first}}}{c'_{\text{oldest-first}}}$. The advantage of the oldest-first collector increases with diminishing X and with increasing L. However, for the square-root-exponential distribution on the left, the contours are widely spaced, which indicates a shallow grade, and only modest improvement in the advantage of the oldest-first collector. For the exponential distribution and even more so for the square-exponential distribution, this improvement is rapid. The break-even contour also shifts a bit with the change in distribution.

9.2 Simulation results for real traces

In this section, we consider copying costs for real programs. We instrumented three systems to record object lifetimes: a Smalltalk virtual machine [Hosking et al., 1992], a custom version of the SML/NJ compiler [Stefanović and Moss, 1994], and a Java virtual machine. We use our language-independent garbage collector toolkit [Hudson et al., 1991] to record object allocation and to report the demise of objects at each collection. We configured the collector to collect very frequently (each 40,000 words of allocation for Smalltalk, and 125,000 words for SML) and to collect the whole heap each time. This setup enables accurate (in relation to trace length) measurement of object lifetimes. We

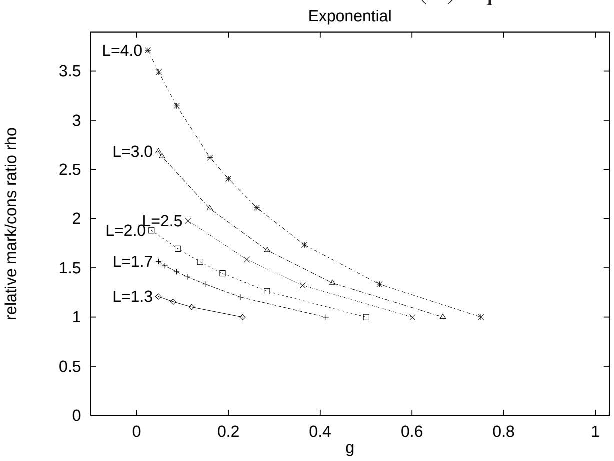


Youngest-first

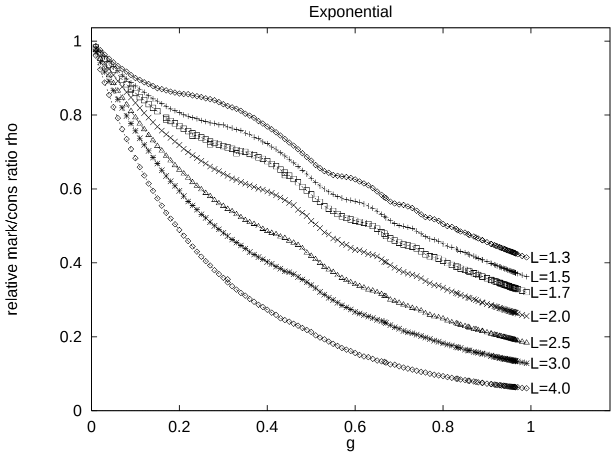


Oldest-first

(A) Square-root-exponential distribution

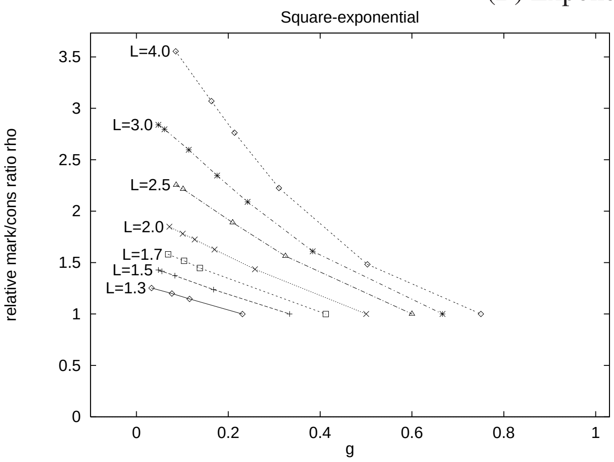


Youngest-first

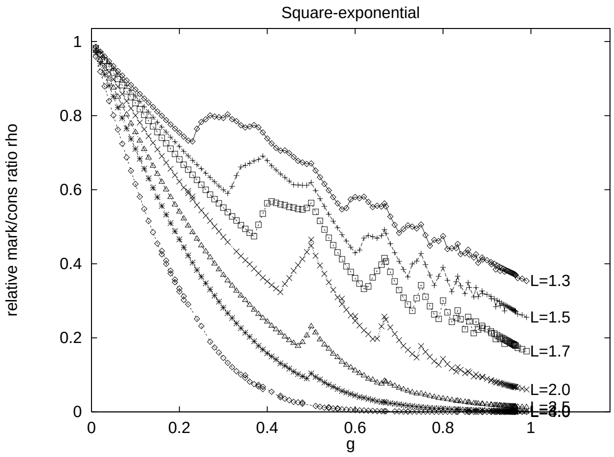


Oldest-first

(B) Exponential distribution



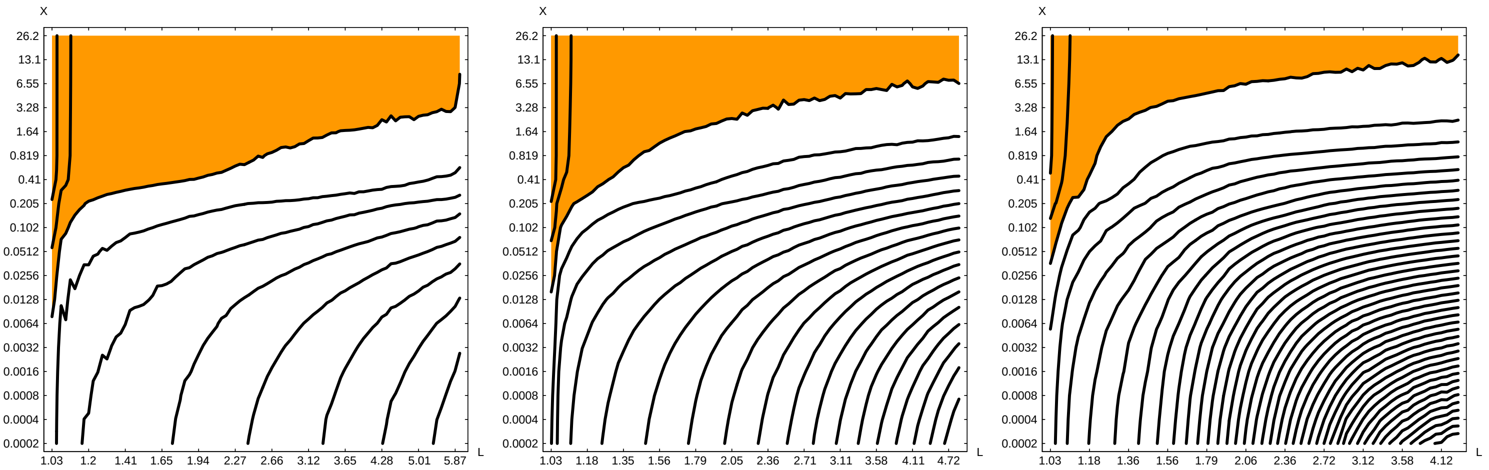
Youngest-first



Oldest-first

(C) Square-exponential distribution

Figure 5: Copying costs of different distributions.



Square-root-exponential Exponential Square-exponential

Figure 6: Comparison of youngest-first and oldest-first collection.

obtained traces for a number of long-running benchmark programs and interactive sessions (17 in Smalltalk, 8 in SML, 19 in Java). The longest trace allocates 310 million words and comes from an hour-long Smalltalk session.

Since our emphasis is on collection in mature space, we used a filtering mechanism to extract subtraces of only mature objects. In practice, mature objects will be those that are promoted out of a young-space collector. Although the objects promoted out of the young-space collector vary in age, depending on the construction of the collector, it can safely be assumed for the purposes of our evaluation that they are of the same age. This threshold age can be chosen somewhat arbitrarily; we made several choices for each original object trace, making sure that the thresholds were beyond the first knee of the survivor function—beyond the region of very young objects with very high mortality. [4] We then focused on those mature object traces that approach a quasi-steady state. Here we report on two SML programs, Tomcatv and Swim, which are adaptations of two SPEC95 matrix calculation benchmarks, and one Smalltalk heap-intensive benchmark program, Tree-replace, which repeatedly replaces subtrees of a large tree by newly allocated subtrees.

We show in Figure 7(a,b,c) the comparison of oldest-first and youngest-first collection for three real program traces of mature space objects. Plots of their survivor functions are in Figure 7(e). The interpretation of the contour graphs is as in the preceding section: oldest-first collection is preferred in the lower region of the plot, for smaller X. However, recall that $X = \frac{C_S}{c_c v}$. For real traces the value of the live amount v in the quasi-steady state is known. Thus, with the suggested realistic values of C_S and c_c, we have actual values for X as indicated underneath the figures. For these values of X, oldest-first is better, regardless of L. Why then provide the plots for a range of X values? C_S and c_c values are different among collector implementations, and so X could be somewhat higher or lower. Note, however, that $\frac{C_S}{c_c}$ would have to be at least 1000 times higher than we have estimated for the operating points to move up into the regions where youngest-first is preferred.

We never expected simulations of real program traces to produce smooth results like those for synthetic traces of analytical distributions; after all they do not enter a truly steady state. Nevertheless, we thought the plots in the top row of Figure 7 to be very jagged. We considered two possible sources of the irregularity: first, that the distribution of lifetimes may be far from smooth (with sharp peaks in mortality at critical ages); and second, that there is significant correlation between lifetimes of successively allocated objects. Both effects can be expected in a trace of any iterative algorithm.

To eliminate the second effect, we generated a synthetic trace having the same distribution of object lifetimes as the real trace, but with lifetimes drawn independently. We show in Figure 7(d) the outcome for such a synthetic trace based on the Tomcatv benchmark. The appearance is still slightly jagged, so this effect must be caused by the distribution itself. The contours change very little, but in the relatively flat region at the top of the plot, the bound-

[4] It is not correct, however, simply to reject objects with lifetimes under the threshold: the lifetimes of the remaining objects must be transformed to correspond to the allocation amounts as seen by mature space.

ary shifts somewhat downward. We cannot claim, however, that the correspondence between the real trace and its synthetic version is close for all traces, and this question awaits further investigation.

10 Simulation results for real traces with pointer structure

11 Discussion and conclusions

We found that oldest-first collection almost always outperforms youngest-first generational collection and non-generational collection. Now for the caveats! The reported analyses are for sliding compaction; pure copying may exhibit some differences. However, since oldest-first does less copying, it presumably will also need less additional space for pure copying. Our comparisons are for a cost model that includes copying and collector startup costs, but the model ignores remembered set maintenance and write-barrier issues, and further assumes that the memory hierarchy affects all collectors equally.

Changing the relationships between the regions collected and not collected might have large effects on remembered set sizes and their processing times. On the other hand, if locality of reference between objects is correlated with original allocation time, then the effects might not be large. We observe that the oldest-first collector effectively inserts new survivors between batches of old survivors, so it may *decrease* locality. This subject demands experimental measurement, and so does cache behavior.

Our simulations assumed perfect knowledge of reachability, too, and actual generational collectors will tend to copy a bit more because they rely on the remembered sets, which are an upper bound on live pointers into the region to be collected, but are not necessarily precise. A related concern is that straightforward oldest-first collection might fail to collect cycles of garbage. The Mature Object Space collector [Hudson and Moss, 1992] addresses that concern, but necessarily reorganizes objects.

We also observe that real systems often have long-lived data, not likely to be discarded, and special techniques are needed to prevent copying such data repeatedly. An example are class files for heavily used language or library classes in Java. On the other hand, classes dynamically loaded by a net browser should perhaps be subject to collection as the user changes attention to different tasks.

Another obvious area for future exploration is applying the oldest-first strategy to collection of young objects. In any case, we conclude that oldest-first collection is quite promising, and look forward to evaluating it *in vivo*.

References

[Appel, 1987] Appel, A. W. (1987). Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279.

[Baker, 1993] Baker, H. G. (1993). 'Infant Mortality' and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57.

[Clinger and Hansen, 1997] Clinger, W. D. and Hansen, L. T. (1997). Generational garbage collection and the radioactive decay model. *SIGPLAN Notices*, 32(5):97–108. *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*.

[Cox and Oakes, 1984] Cox, D. R. and Oakes, D. (1984). *Analysis of Survival Data*. Chapman and Hall, London.

[Devroye, 1986] Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag, New York.

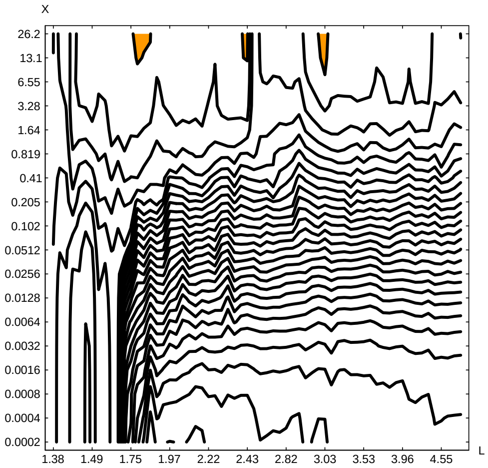
[Elandt-Johnson and Johnson, 1980] Elandt-Johnson, R. C. and Johnson, N. L. (1980). *Survival Models and Data Analysis*. Wiley, New York.

[Hayes, 1993] Hayes, B. (1993). *Key Objects in Garbage Collection*. PhD thesis, Stanford University, Stanford, California.

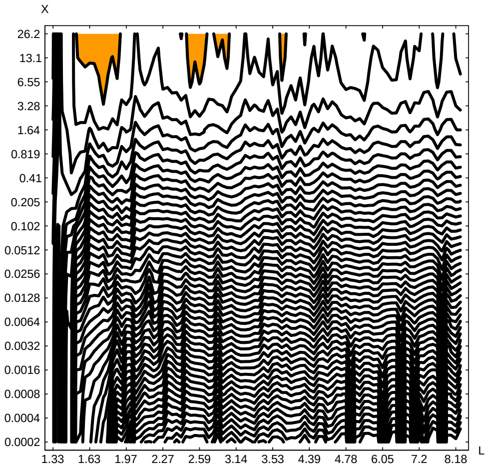
[Hosking et al., 1992] Hosking, A. L., Moss, J. E. B., and Stefanović, D. (1992). A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 92–109, Vancouver, Canada. *SIGPLAN Notices 27*, 10 (October 1992).

[Hudson and Moss, 1992] Hudson, R. L. and Moss, J. E. B. (1992). Incremental collection of mature objects. In Bekkers, Y. and Cohen, J., editors, *International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 388–403, St. Malo, France. Springer-Verlag.

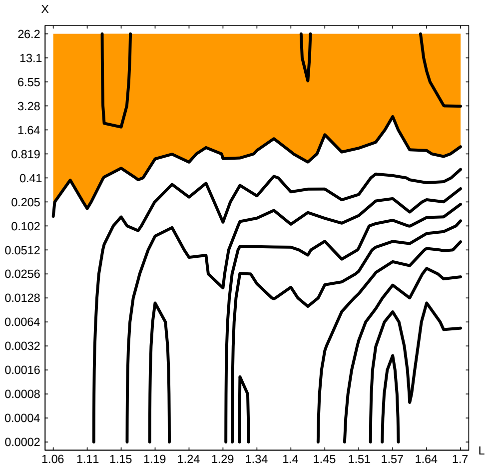
[Hudson et al., 1991] Hudson, R. L., Moss, J. E. B., Diwan, A., and Weight, C. F. (1991). A language-independent garbage collector toolkit. Technical Report 91-47, University of Massachusetts, Amherst.



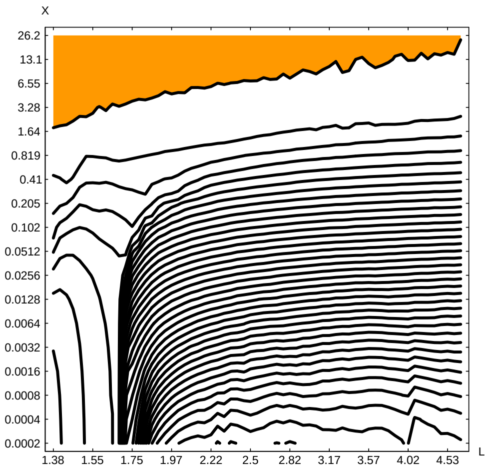
(a) SML SPEC101-Tomcatv
($v \approx 13396, X \approx 0.01$)



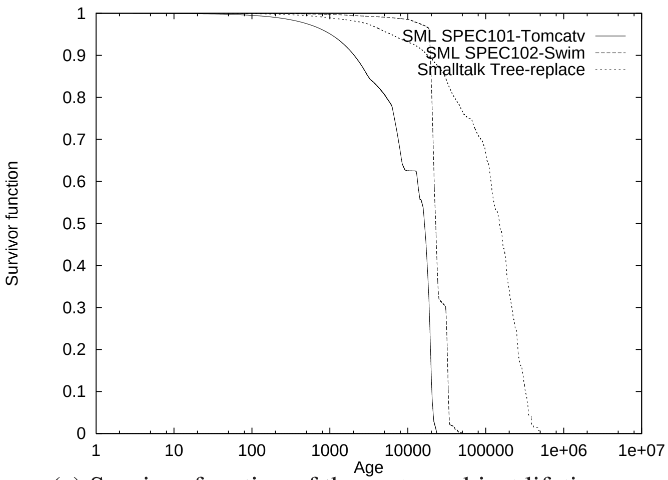
(b) SML SPEC102-Swim
($v \approx 25094, X \approx 0.005$)



(c) Smalltalk Tree-replace
($v \approx 187966, X \approx 0.0007$)



(d) Synthetic trace with distribution as in SML SPEC101-Tomcatv



(e) Survivor function of the mature object lifetime distribution

Figure 7: Comparison of youngest-first and oldest-first collection (real traces).

[Jones and Lins, 1996] Jones, R. and Lins, R. (1996). *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, Chichester.

[Seligmann and Grarup, 1995] Seligmann, J. and Grarup, S. (1995). Incremental mature garbage collection using the train algorith. In Nierstras, O., editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, University of Aarhus. Springer-Verlag.

[Stefanović and Moss, 1994] Stefanović, D. and Moss, J. E. B. (1994). Characterisation of object behaviour in Standard ML of New Jersey. In *1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida.

Appendix

A Copying cost in the youngest-first collector

Let the younger generation (0) occupy gV and older generation (1) occupy $(1-g)V$.

The exact time between two minor collections is $\tau_0 = gV$. The expected live amount in the younger generation at collection is $\varepsilon_0 V = \int_0^{\tau_0} s_o(t)dt = \frac{1}{\lambda}\left(1-e^{-\lambda gV}\right)$. Live data from generation 0 are promoted into generation 1.

Upon some number $\zeta - 1$ of minor collections, the older generation fills up. On the next minor collection it becomes necessary to collect the older generation as well in a major collection. Let $\varepsilon_1 V$ be the expected live amount in both generations at the time of major collection. The expected available space in generation 1 between collections is $\alpha_1 V = (1-g)V - \varepsilon_1 V$. The expected number of minor collections between two major collections is $\zeta - 1 = \frac{\alpha_1}{\varepsilon_0}$. For the purposes of analysis, we take the values of the free parameters g and L such that ζ is an integer. (It can be shown that this assumption is more than just a convenience for analysis.) At collection time, the younger generation contains an area of size $\varepsilon_1 V$ that was last established live at time $\zeta\tau_0$ ago, and $\zeta - 1$ areas of size $\varepsilon_0 V$ that were last established live at times $(\zeta-1)\tau_0, \ldots, \tau_0$ ago. Generation 0 also contains new live data, as at any generation 0 collection. Owing to the memoryless property of the object lifetime distribution, the expected amount of live data in generation 1 is then:

$$\varepsilon_1 V = \varepsilon_1 V s_0(\zeta\tau_0) + \sum_{i=1}^{\zeta-1} \varepsilon_0 V s_0(i\tau_0) + \varepsilon_0 V = \varepsilon_1 V s_0(\zeta\tau_0) + \sum_{i=0}^{\zeta-1} \varepsilon_0 V s_0(i\tau_0) = \varepsilon_1 V e^{-\lambda\zeta\tau_0} + \varepsilon_0 V \sum_{i=0}^{\zeta-1} e^{-\lambda i\tau_0}$$

or,

$$\varepsilon_1 = \varepsilon_0 \frac{\sum_{i=0}^{\zeta-1} e^{-\lambda igV}}{1-e^{-\lambda\zeta gV}} = \frac{\varepsilon_0}{1-e^{-\lambda gV}}.$$

In one generation 1 cycle, the amount allocated is ζgV, and the amount copied is $(\zeta-1)\varepsilon_0 V + \varepsilon_1 V$, so the mark/cons ratio is

$$\mu = \frac{(\zeta-1)\varepsilon_0 V + \varepsilon_1 V}{\zeta gV} = \frac{\varepsilon_0 V\left(\zeta - 1 + \frac{1}{1-e^{-\lambda gV}}\right)}{\zeta gV} = \frac{\left(1-e^{-\lambda gV}\right)\left(\zeta - 1 + \frac{1}{1-e^{-\lambda gV}}\right)}{\lambda\zeta gV} = \frac{\left(1-e^{-Lg}\right)(\zeta-1)+1}{\zeta Lg}$$

On the other hand,

$$\zeta - 1 = \frac{\alpha_1}{\varepsilon_0} = \frac{(1-g)}{\varepsilon_0} - \frac{\varepsilon_1}{\varepsilon_0} = \frac{(1-g)\frac{L}{\lambda}}{\frac{1}{\lambda}\left(1-e^{-\lambda gV}\right)} - \frac{1}{1-e^{-\lambda gV}} = \frac{L(1-g)-1}{1-e^{-Lg}}.$$

Thus,

$$\zeta = \frac{L(1-g)-e^{-Lg}}{1-e^{-Lg}}.$$

With a given factor L, for particular (integer) values of ζ, this equation can be solved numerically for g, and for such pairs L, g, the preceding analysis will be valid. Simplifying further,

$$\mu = \frac{1-g}{\zeta g}.$$

B Copying cost in the oldest-first collector

The analysis of the oldest-first collector is somewhat involved; we present it for the case of the exponential distribution, which allows us to simplify the presentation by appeal to the memoryless property of the distribution: each time an object is copied in collection, it is in effect, reborn. (For any other distribution, it is possible to formulate similar equations involving infinite sums of integrals of the survivor function.) Our analysis is exact, but the result is not in closed form, as it is a solution of a nonlinear equation.

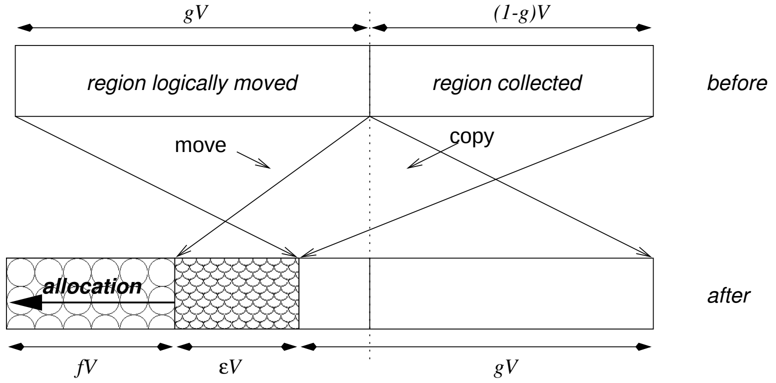


Figure 8: Operation of the oldest-first collector.

The analysis will be made easier if we view the collector as in Figure 8, and first ask what is in the region of size $(1-g)V$ that is collected. The young region is of size gV, and it therefore shifts by an amount $(1-g)V$ on each collection. On each collection, an amount εV of collected data is placed to the left of the shifted data; then an amount fV of new data is allocated in the leftmost, free, region. This pair of areas fV and εV are moved rightwards by $(1-g)V$ on each successive collection. It takes $N(g) = \left\lceil \frac{g}{1-g} \right\rceil$ collections for this pair to move entirely past gV, into the region collected. In other words, if N is an integer, then in N collections this pair becomes precisely the region collected. However, if N is not an integer, the analysis is more complicated. We first note that if $g \in \left(\frac{n-1}{n}, \frac{n}{n+1}\right], n \in \mathbb{N}$, then $N(g) = n$.

With respect to the contents of the region collected, there are four possibilities to consider, illustrated in Figure 9. Let $y = 1 - N(1-g)$. Case 3 obtains when $y = 1-g$, in other words when N is an integer. Otherwise, case 1 obtains when $f > y$, case 2 when $f < y$, and case 4 when $f = y$. In cases 1, 3, and 4, an εV region is wholly within the collected region, and thus older data from an entire sequence of ranges of allocation time will remain together. However, in case 2, two parts of two different εV regions lie in the collected region; thus it matters which data are in these parts, and which outside. In other words, it matters in which order the copied data are placed into the "to-space" on collection: in a sliding implementation, it is the original order of the data.

We proceed with the analysis for the exponential distribution of object lifetimes, which allows a simpler treatment of case 2, thanks to the memoryless property of the distribution.

Case 1. The collected region contains parts of two "new" areas, the younger of size $(f-y)V$, and the older of size yV; and an area with survivors of a previous collection that happened NfV ago. The live amount in the collected region is

$$\int_{(N-1)fV+yV}^{NfV} e^{-\lambda t}\,dt + \int_{NfV}^{NfV+yV} e^{-\lambda t}\,dt + \varepsilon V e^{-\lambda NfV}$$

$$= \frac{1}{\lambda} e^{-\lambda(Nf+1-N(1-g))V}\left(e^{\lambda fV}-1\right) + \varepsilon V e^{-\lambda NfV}.$$

This live amount can be equated with εV, thus:

$$\varepsilon V = \frac{1}{\lambda} e^{-\lambda(Nf+1-N(1-g))V}\left(e^{\lambda fV}-1\right) + \varepsilon V e^{-\lambda NfV}.$$

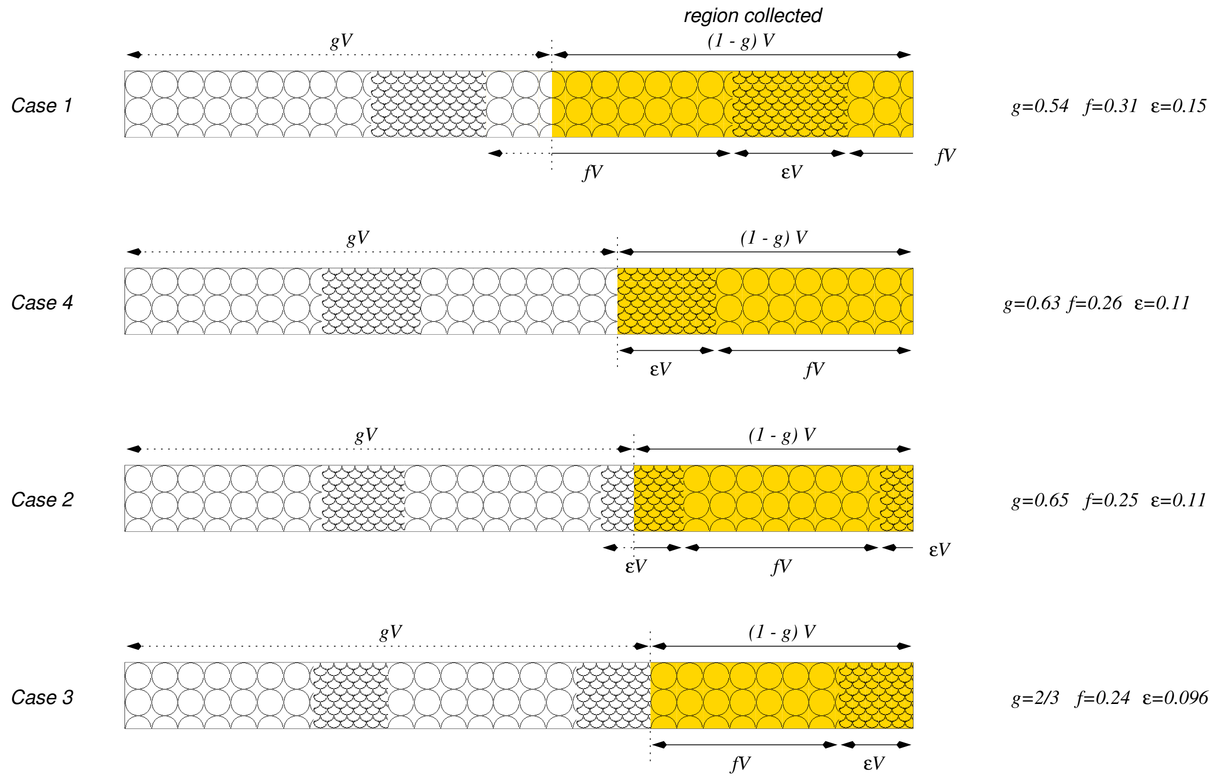


Figure 9: Operation of the oldest-first collector: contents of collected region (exponential distribution, $L = 2$).

Using $L = V\lambda$, we have:

$$\varepsilon L\left(1 - e^{-NfL}\right) = e^{-(Nf+1-N(1-g))L}\left(e^{fL} - 1\right).$$

Finally, $\varepsilon = 1 - g - f$, and the equation, to be solved numerically for f, given L and g, is:

$$(1 - g - f)L\left(1 - e^{-NfL}\right) = e^{-(Nf+1-N(1-g))L}\left(e^{fL} - 1\right).$$

Case 2. The collected region contains one "new" area of size fV, and parts of two areas with survivors of previous collections; one, of size $(N(1-g) - g)V$, from a collection NfV ago, and the other, of size $\varepsilon V - (N(1-g) - g)V$, from a collection $(N+1)fV$ ago. The live amount in the collected region is

$$\int_{NfV}^{(N+1)fV} e^{-\lambda t}\,dt + (N(1-g)V - gV)e^{-\lambda NfV} + (\varepsilon V - N(1-g)V + gV)e^{-\lambda(N+1)fV}$$

$$= \frac{1}{\lambda}\left(e^{-\lambda NfV} - e^{-\lambda(N+1)fV}\right) + (N(1-g)V - gV)e^{-\lambda NfV} + (\varepsilon V - N(1-g)V + gV)e^{-\lambda(N+1)fV}.$$

Again, this live amount can be equated with εV:

$$\varepsilon V = \frac{1}{\lambda}\left(e^{-\lambda NfV} - e^{-\lambda(N+1)fV}\right) + (N(1-g)V - gV)e^{-\lambda NfV} + (\varepsilon V - N(1-g)V + gV)e^{-\lambda(N+1)fV}.$$

Using $L = V\lambda$, we have:

$$\varepsilon L = e^{-NfL}(1 + N(1-g)L - gL) + e^{-(N+1)fL}(-1 + \varepsilon L - N(1-g)L + gL),$$

or:

$$\varepsilon L\left(1 - e^{-(N+1)fL}\right) = (1 + N(1-g)L - gL)e^{-NfL}\left(1 - e^{-fL}\right).$$

With $\varepsilon = 1 - g - f$:

$$(1-g-f)L\left(1-e^{-(N+1)fL}\right) = (1+N(1-g)L-gL)e^{-NfL}\left(1-e^{-fL}\right).$$

Again, this equation must be solved numerically for any given L and g.

Case 3. The fV region contains data with ages between NfV and $(N+1)fV$, and the εV region contains data that were last collected $(N+1)fV$ ago. The equation is then:

$$\varepsilon V = \int_{NfV}^{(N+1)fV} e^{-\lambda t}dt + \varepsilon V e^{-\lambda(N+1)fV} = \frac{1}{\lambda}\left(e^{-\lambda NfV} - e^{-\lambda(N+1)fV}\right) + \varepsilon V e^{-\lambda(N+1)fV},$$

which, with $L = V\lambda$, $K = N+1 = \frac{1}{1-g}$ simplifies to:

$$(1-Kf)L\left(1-e^{-KfL}\right) = Ke^{-KfL}\left(e^{fL}-1\right).$$

This models a collector with K areas of equal size, which functions as a queue with K stations, similar to a *train* of Ref. [Hudson and Moss, 1992].

Case 4. This is the case where the solutions for cases 1 and 2 coincide, at such points $g_N^\star$ that $f = y$. The equation governing these points is:

$$(N-(N+1)g_N^\star)L\left(1-e^{-N(1-N(1-g_N^\star))L}\right) = e^{-(N+1)(1-N(1-g_N^\star))L}\left(e^{(1-N(1-g_N^\star))L}-1\right).$$

Once f is found, we compute $\varepsilon = 1-g-f, \mu = \frac{\varepsilon}{f}$, $\rho = \mu(L-1)$. Efficient computation first determines the values of N of interest, corresponding to the interval $\left(\frac{n-1}{n}, \frac{n}{n+1}\right]$, and for each N and L finds the point $g_N^\star$, within this interval. To the left of $g_N^\star$, case 1 applies, and to the right of $g_N^\star$, case 2 applies.

As $g \to 1_-$ and $L \to \infty$, the performance approaches that of an ideal object queue, $\mu = e^{-L}$.