Generational Garbage Collection and the Radioactive Decay Model

William D Clinger and Lars T Hansen
Northeastern University
{will,lth}@ccs.neu.edu

Abstract

If a fixed exponentially decreasing probability distribution function is used to model every object's lifetime, then the age of an object gives no information about its future life expectancy. This *radioactive decay model* implies there can be no rational basis for deciding which live objects should be promoted to another generation. Yet there remains a rational basis for deciding how many objects to promote, when to collect garbage, and which generations to collect.

Analysis of the model leads to a new kind of generational garbage collector whose effectiveness does not depend upon heuristics that predict which objects will live longer than others.

This result provides insight into the computational advantages of generational garbage collection, with implications for the management of objects whose life expectancies are difficult to predict.

1 Introduction

The computational advantages of generational garbage collection over non-generational collection are generally attributed to heuristic prediction of object lifetimes and to improved caching and paging behavior [5, 39]. This hypothesis can be tested by considering a uniform memory system and a model of object lifetimes for which no heuristic predictor can do better or worse than chance.

The radioactive decay model is a model of object lifetimes that defeats all heuristics that attempt to predict which objects will live longer than others, including heuristics that are tuned to a specific model and its parameters. This model has the remarkable property that the age of a live object, together with the time at which the object was allocated, gives absolutely no information about the probability that the object will survive to any future time. The model is also simple and easy to analyze.

This paper describes and analyzes a new kind of generational garbage collector, the *non-predictive* collectors. A calculation shows that non-predictive collectors can outperform non-generational collectors for the radioactive decay model. This result proves that generational garbage collection has an advantage in computational complexity that cannot be attributed to heuristic prediction of object lifetimes, to locality effects, or to the benefit of retaining a large part of its state from one collection to the next.

Although the radioactive decay model does not describe the overall behavior of objects in most real programs, it may be a useful model for long-lived objects. Experimental studies have thus far found few regularities in the behavior of long-lived objects that could be used to predict which of them are likely to live longer than others [24, 33, 32, 39]. A total absence of such information is the defining characteristic of the radioactive decay model.

Non-predictive collectors appear suitable for managing the oldest generations of an otherwise conventional multi-generation garbage collector. We have designed a prototype of such a collector for the Larceny implementation of Scheme [14, 21, 27]. This prototype began to work as this paper went to press. On most programs the new collector performs the same as the generational collector it replaces, but we expect the new collector to improve the performance of some programs that present a challenge to our conventional generational collector.

Our main conclusion is that research effort directed toward heuristic prediction of object lifetimes is slightly misplaced. What really matters is the heuristic selection of generations to collect. These two problems are not equivalent.

To appear in the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1997, in Las Vegas, Nevada.

2 The radioactive decay model

In the radioactive decay model, a single probability distribution function describes the life expectancy of every object. There is but one parameter of the model, the half-life h. For every object that is live at time t_0, the probability that the object will still be alive at time $t_0 + t$ is $2^{-t/h}$. The probability that the object will be dead at that time is $1 - 2^{-t/h}$. The derivative of this is the probability density function

$$P_h(t) = \frac{\log 2}{h} 2^{-t/h}$$

The radioactive decay model is characterized by two assumptions. The first assumption ensures that the age of a live object can never provide any clue to its future prospects. The second assumption ensures that no other clues are available either.

Assumption 1 *There exists $h > 0$ such that, for each live object o, the life expectancy of o is described by $P_h(t)$.*

Assumption 2 *Live objects have no other distinguishing characteristics that might be exploited by a generational garbage collector.*

With these assumptions, there is no rational basis for any policy that could be used by a generational garbage collector to decide which live objects should be kept in which generation. The collector might as well make all such decisions randomly.

If the time t is measured by the number of objects that have been allocated, then the radioactive decay model implies that an equilibrium will be approached after several half-lives of time have passed. At equilibrium one object can be expected to die per unit time. The expected number n of live objects at equilibrium is therefore related to the half-life h by $1 = n(1 - 2^{-1/h})$.

Let $r = 2^{-1/h}$. By L'Hospital's Rule,

$$r \approx 1 - \frac{\log 2}{h}$$

for large h [1, 34]. Small values of h imply a small number of live objects, which makes garbage collection too easy to be interesting, so this approximation can safely be used to calculate that the live storage at equilibrium is

$$n \;=\; 1/(1-r) \approx \frac{h}{\log 2} \doteq 1.4427h \qquad (1)$$

If most objects die young, then there must be few live objects. The radioactive decay model is useful only as a model for long-lived objects.

3 Generational garbage collection

Generational garbage collectors work by dividing heap storage into generations. They have policies that determine the generation in which a new object will be allocated, when and how to move objects from one generation to another, and when and how to collect garbage.

There are three distinct properties of generational collectors that allow them to reduce the overhead of garbage collection for some programs:

- By limiting the region in which new objects are allocated, and to a lesser extent by controlling the regions in which objects reside after allocation, generational collectors improve the caching and paging behavior of a program [37].
- By retaining a *remembered set* of cross-generational pointers from one collection to the next, and by maintaining that set in cooperation with the mutator, generational collectors do not have to start from scratch on every collection [35, 39].
- By collecting a cleverly selected subset of the generations, generational collectors can reduce the amortized overhead of garbage collection.

A first approximation to the amortized overhead of garbage collection can be estimated by dividing the number of objects that have been marked (or copied, or whatever) by the number of objects that have been allocated. This is the *mark/cons* ratio.

A generational collector attempts to lower the mark/cons ratio by collecting generations that contain a higher percentage of garbage than the average for all generations. Such generations require less marking and reclaim more storage than the average.

Clearly the collector's choice of generations to collect is critical. If the selected generations actually contain an unusually low percentage of garbage, then the mark/cons ratio will rise instead of fall.

That is what happens when a conventional generational garbage collector is used for a program whose object lifetimes resemble the radioactive decay model. The collector simply assumes that young objects have a shorter life expectancy than old objects, and concentrates its effort on collecting the generations that contain the most recently allocated objects. In the radioactive decay model, these are the objects that have had the least amount of time in which to decay, so the generations in which they reside contain an unusually *low* percentage of garbage. For the radioactive decay model, therefore, a conventional generational collector will perform *worse* than a similar non-generational collector.

This is remarkable because the radioactive decay model ensures that no heuristic predictor of object lifetimes can do worse than chance. Heuristic selection

	live storage in each step						
t	step 1	step 2	step 3	step 4	step 5	step 6	step 7
0	0	0	0	0	0	1024	1024
1024	0	0	0	0	1024	512	512
2048	0	0	0	1024	512	256	256
3072	0	0	1024	512	256	128	128
4096	0	1024	512	256	128	64	64
5120	*1024	512	256	128	64	32	32
gc	0	0	0	0	0	1024	*1024

*These two steps are exchanged, not collected.

Table 1: Live storage in a non-predictive generational collector.

of generations to collect must therefore be a different problem from heuristic prediction of object lifetimes.

Furthermore the fact that a poor choice of generations to collect can cause a generational collector to perform worse than a non-generational collector for the radioactive decay model suggests that a better choice of generations might allow it to perform better.

The main theoretical result of this paper is that not even the radioactive decay model can prevent a generational collector from organizing its generations to obtain a lower mark/cons ratio than a non-generational collector.

4 Non-predictive generational collectors

A non-predictive generational garbage collector does not attempt to predict the lifetimes of objects. Neither does it keep track of the ages of objects. It does, however, keep track of how much allocation has been performed since an object was allocated or last considered for collection.

Just as a conventional generational collector can use a mark/sweep, compacting mark/sweep, or a stop-and-copy collector to collect its generations, so can a non-predictive generational collector use any of those basic algorithms.

This section describes a specific 2-generation non-predictive collector. This collector divides heap storage into k *steps* of equal size. Step 1 is considered the youngest, and step k is the oldest. A dynamic tuning parameter $j \geq 0$ determines how many of these steps are allocated to the young generation, which consists of steps 1 through j. The old generation consists of steps $j+1$ through k.

All allocation occurs in the highest-numbered step that has free space. When it becomes full, the next highest-numbered step becomes the allocation area, and so on until step 1 fills. The tuning parameter j determines how many of the youngest steps will *not* be collected during the next collection. The collector essentially assumes that all objects in steps 1 through j are live.

When all steps become full:

- Steps $j+1$ through k are collected as a single generation, promoting objects to the highest-numbered step that contains free space.
- Following collection, steps $j+1$ through k become the new steps 1 through $k-j$; the original steps 1 through j become steps $k-j+1$ through k.
- Then a new value of the tuning parameter j can be chosen. For technical reasons described in Section 8, the implementation of this algorithm becomes simpler if j is always chosen so that steps 1 through j are empty.

An example of a non-predictive generational collector at work is shown in Table 1, which assumes j is fixed at 1. The numbers shown in this table are close to but nicer than the numbers that would be expected from the radioactive decay model with a half-life of 1024 and an inverse load factor of 3.5.

The mark/cons ratio for Table 1 is $1024/5120 = 0.2$. For a non-generational mark/sweep collector, the mark/cons ratio would be $2048/5120 = 0.4$. The non-predictive collector has almost as much overhead for sweeping as the non-generational collector, and has at least as much overhead for scanning roots, so the non-predictive collector's advantage is not quite so large as the mark/cons ratios would suggest.

5 Mathematical analysis

The garbage collection problem for the radioactive decay model has two degrees of freedom: the half-life h and the load factor, defined as the amount of live storage divided by the total size of the heap. Let L be the inverse of the load factor. The live storage

at equilibrium is $n \approx h/(\log 2)$, and the heap size is $N = nL \approx hL/(\log 2)$. Let

$$r = 2^{-1/h} = 1 - \frac{1}{n} \approx 1 - \frac{\log 2}{h}$$

Let $g = j/k$ be the fraction of storage devoted to the young generation. Let f be such that $0 \le f \le g$ and Nf is the space available in steps 1 through j following a garbage collection and renaming of steps.

Assume $g \le 1/2$, and assume that all unavailable storage in steps 1 through j is actually live. If j is chosen as we recommend in Section 8, then $f = g$ and this assumption will be vacuously true.

The next collection will occur after Nf new objects have been allocated. At that time, the expected number of live objects in steps 1 through j is

$$\begin{aligned}
\mathrm{live}_h(f,g) &= \sum_{t=1}^{Nf} 2^{-t/h} + N(g-f)2^{-Nf/h} \\
&= \frac{r(1-r^{Nf})}{1-r} + N(g-f)r^{Nf} \\
&= \frac{1-\frac{1}{n}}{1-(1-\frac{1}{n})}(1-r^{Nf}) + nL(g-f)r^{Nf} \\
&= n[(1-\frac{1}{n})(1-r^{Nf}) + L(g-f)r^{Nf}] \\
&\approx n[(1-r^{Nf}) + L(g-f)r^{Nf}] \\
&\approx n[(1-r^{hLf/(\log 2)}) \\
&\quad + L(g-f)r^{hLf/(\log 2)}] \\
&= n[(1-2^{-Lf/(\log 2)}) \\
&\quad + L(g-f)2^{-Lf/(\log 2)}] \\
&= n[1-2^{-Lf/(\log 2)}(1-L(g-f))]
\end{aligned}$$

Let

$$l(f,g) = 1 - 2^{-Lf/(\log 2)}(1-L(g-f))$$

Theorem 3

$$\lim_{h\to\infty} \frac{\mathrm{live}_h(f,g)}{n} = l(f,g)$$

$l(f,g)$ is the fraction of *live* storage that is expected to reside in steps 1 through j at the beginning of the next garbage collection.

The expected number of garbage objects in steps 1 through j is approximately

$$Ng - n\,l(f,g) \approx \frac{h}{\log 2}(Lg - l(f,g))$$

The expected number of live objects in steps $j+1$ through k is approximately

$$\frac{h}{\log 2}(1 - l(f,g)) \qquad (2)$$

The expected number of garbage objects in steps $j+1$ through k is approximately

$$\frac{h}{\log 2}(L(1-g) - 1 + l(f,g)) \qquad (3)$$

This is also the amount of space that will be reclaimed by the next collection, so it is the expected amount of free space after the next collection.

The expected amount of space available in steps 1 through j after the next collection and renaming of steps is therefore

$$\frac{h}{\log 2} L \min\left(g, 1-g+\frac{l(f,g)-1}{L}\right)$$

Theorem 4 *If $f = g$, $g \le 1/2$, and*

$$L(1-2g) \ge 1 - l(g,g)$$

then the expected mark/cons ratio is approximately

$$\frac{1-l(g,g)}{L(1-g)-(1-l(g,g))}$$

Proof:

$$l(g,g) = 1 - 2^{-Lf/(\log 2)}$$

$f = g$ means that all storage in steps 1 through j is free. Under the hypotheses of the theorem, it is expected that all storage in steps 1 through j will be free following the next collection also, which implies a stable equilibrium. All objects that are in steps 1 through j following a collection are live, because there aren't any. The mark/cons ratio thus becomes a trivial calculation.

2

If the hypotheses of Theorem 4 do not hold, then the mark/cons ratio can be estimated by computing a fixed point

$$f = \max\left(0, \min\left(1-g+\frac{l(f,g)-1}{L}, g\right)\right) \qquad (4)$$

and dividing (2) by (3). The result of that division is merely a lower bound, however, because it will not be the case that all storage in steps 1 through j is live following a collection and renaming.

If a non-generational mark/sweep collector were used instead, the mark/cons ratio would be $1/(L-1)$.

Corollary 5 *If $f = g$, $g \le 1/2$, and*

$$L(1-2g) \ge 1 - l(g,g)$$

then the expected mark/cons ratio relative to a non-generational mark/sweep collector is approximately

$$\frac{(L-1)(1-l(g,g))}{L(1-g)-(1-l(g,g))}$$

This measure of relative CPU overhead is graphed in Figure 1. The thin black lines in that figure are accurate. The thick lines are lower bounds computed using equation (4).

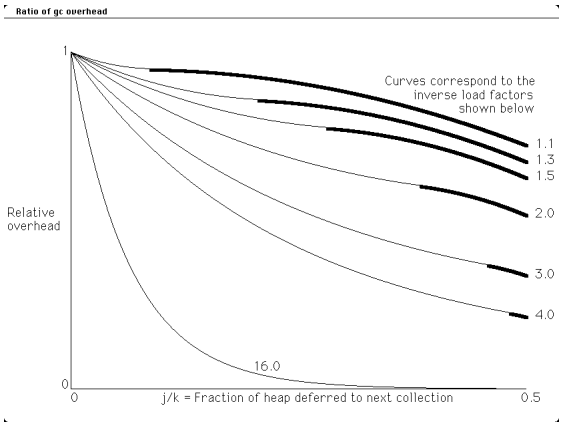

Figure 1: The mark/cons overhead for non-predictive generational gc divided by the overhead for non-generational gc, as a function of generation size and inverse load factor L, for the radioactive decay model. Thick lines are lower bounds.

6 Pragmatics

It remains to be seen whether non-predictive collectors are useful in practice. Several issues must be addressed:

- The radioactive decay model is a poor model of object lifetimes in most real programs.
- The dynamic tuning parameter j must be chosen to reduce garbage collection overhead.
- Cyclic structures will not be reclaimed unless the entire structure resides within a single generation.
- The size of the remembered set might increase because non-predictive collectors cannot rely on the heuristic that most inter-generational pointers point from younger to older generations.
- The remembered set must be maintained a little differently.
- The calculations in section 5 do not include the cost of a write barrier.
- The calculations in section 5 do not include the cost of tracing the root set (which includes part of the remembered set).

The first issue is the most important, and is almost independent of the garbage collector. We discuss it in Section 7.

In Section 8 we discuss the remaining issues in the context of Larceny, our implementation of Scheme for the SPARC architecture [14, 21]. Larceny allows the same binary code to be linked with any of several different garbage collectors. Two of these collectors are a non-generational stop-and-copy collector, and a conventional multi-generation collector that uses the stop-and-copy code for its basic algorithm.

We have modified the multi-generation collector to use a non-predictive collector for the oldest dynamic generation. This collector does not yet match the detailed description in Section 8. We will report on its performance at the conference and on the World-Wide Web [13].

7 Lifetimes of real objects

In most programs, young objects have a much shorter life expectancy than old objects. This *weak generational hypothesis* [22, 24, 39] is especially true of fast-allocating programs for which the performance of garbage collection is most critical, and appears to be true even of fast-allocating programs written in C [8].

Most generational collectors take advantage of this by using a very light load factor for the youngest generations, and by using abandonment (as in the stop-and-copy algorithm [10, 17, 30]) instead of sweeping to reclaim storage. This makes the gc time for the young generations proportional to the amount of live storage, which is expected to be much smaller than the total size of the young generations. Objects that do not die young are copied to an older generation, where they can be managed using a heavier load factor and a different algorithm.

The result is that, compared to non-generational collectors, conventional generational collectors make short-lived objects much cheaper—a factor of 10 is typical—while making long-lived objects a little more expensive.

In effect, conventional generational collectors predict that every object will die young. Copying an object to an older generation is a fairly cheap way to recover from heuristic mis-prediction, however, so it is possible for such collectors to perform reasonably well even when object lifetimes do not match the heuristic predictor.

What should be done with objects that survive long enough to be promoted out of the young generations? We might hope to extract some computational advantage from the *strong generational hypothesis*, which postulates a positive correlation between age and life expectancy even for long-lived objects [24, 39]. Unfor-

name	lines of code	brief description
`nbody`	1428	inverse-square law simulation [20, 40]
`nucleic2`	3732	determination of nucleic acids' spatial structure [16, 23]
`lattice`	219	enumeration of maps between lattices
`10dynamic`	2342	Henglein's dynamic type inference [25]
`nboyer`	767	term rewriting and tautology checking (see text)
`sboyer`	781	tweaked version of `nboyer`

Table 2: Six allocation-intensive benchmarks.

tunately there is little evidence that long-lived objects show *any* strong correlation between age and life expectancy [24, 33, 32, 39].

7.1 Six fast-allocating benchmarks

Most programs spend little or no time in the garbage collector, so we have selected six exceptional benchmarks to illustrate the variety of storage behaviors with which a garbage collector must contend. Table 2 lists these programs, which are adapted from 5 of the 9 benchmarks considered by a recent paper on compiler optimization [28]. More benchmarks can be found at our web site [13].

The `10dynamic` benchmark consists of an interprocedural static analysis iterated 10 times on its own source code, to simulate its use on several files in succession. The source code is read only once, before the measured portion of the benchmark.

The `nboyer` benchmark is an updated version of a toy theorem prover written by Bob Boyer circa 1977. The original program was considered one of the more realistic of the Gabriel benchmarks, and its storage behavior has been examined by several authors [4, 6, 18, 31]. We have fixed one bug in addition to those noted by Baker [4, 6], replaced property lists by a faster and more portable data structure, and added a problem scaling parameter suggested by Boyer [7].

The `sboyer` benchmark is `nboyer` with a local tweak (shared consing) suggested by Henry Baker [6]. This benchmark illustrates the changes in object lifetimes that occur as a program is tuned to reduce excessive allocation.

As can be seen from Table 3, these programs allocate between one half and eight megabytes per second, which is enough to make garbage collection matter. The peak storage is estimated from the semiheap size chosen by Larceny's non-generational stop-and-copy collector. The times reported in Table 3 are elapsed time on an otherwise idle machine with enough RAM to avoid paging, taking the average of six runs (three in succession with each collector). Table 3 also shows the time spent in the garbage collector as a fraction of the time spent in the mutator, which is defined as the entire program excluding the garbage collector. The overhead reported for the generational collector was obtained using 1 megabyte for the youngest generation, an 800 kilobyte static area for the standard library, and an intermediate dynamic area consisting of a single generation whose size was adjusted to ensure that the generational collector would touch a little less storage than the stop-and-copy collector. Both garbage collectors would perform better if told to use a lighter load factor, especially for `nucleic2`.

Table 3 shows that the relationship between allocation rate and garbage collection overhead is not simple. To understand the gc overhead we must look more closely at several factors, including the lifetimes of objects.

7.2 Lifetimes of long-lived objects

The excessively rapid allocation by `nbody` and `nucleic2` is an artifact of Larceny's uniform 32-bit representation for objects and of the Twobit compiler's total ignorance of floating point arithmetic. Each of the 7 million floating point operations in `nucleic2` allocates 16 bytes of heap storage: a header word, a word of padding, and two data words for an IEEE double precision floating point number. Half of this storage allocation would be eliminated by using IEEE single precision, which Larceny does not currently support. A further 80 to 90 per cent would be eliminated by local or interprocedural optimization, respectively, of floating point values [23]. This would increase the speed of the mutator as well as reduce the overhead of garbage collection.

The `lattice` benchmark is typical of purely functional programs. Despite its fairly high allocation rate, it allocates almost no long-lived storage.

The `10dynamic` benchmark is more difficult than its allocation rate would suggest, because almost all of the storage it allocates during each iteration survives until nearly the end of the iteration. This benchmark satisfies neither the weak nor the strong generational hypothesis, which is why its performance becomes worse when Larceny's generational collector is used.

The storage allocated by one iteration of `10dynamic` is displayed graphically in Figure 2. The survival rates

name	storage allocated	peak storage (estimated)	semiheap size (stop-and-copy)	mutator time	(gc time) / (mutator time)	
					stop-and-copy	generational
`nbody2-128`	160 Mby	< 1 Mby	2 Mby	51 sec	85%	20%
`nucleic2`	130 Mby	< 1 Mby	2 Mby	16 sec	124%	96%
`lattice`	95 Mby	3 Mby	5 Mby	178 sec	5%	2%
`10dynamic`	18 Mby	2 Mby	3 Mby	13 sec	13%	28%
`nboyer2`	37 Mby	5 Mby	8 Mby	12 sec	52%	44%
`sboyer2`	10 Mby	1 Mby	3 Mby	14 sec	10%	4%
`sboyer3`	23 Mby	3 Mby	6 Mby	35 sec	9%	4%
`sboyer4`	54 Mby	10 Mby	15 Mby	100 sec	12%	6%

Table 3: Storage allocation and garbage collection overheads.

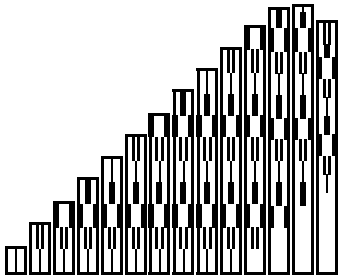

Figure 2: Live storage versus time for one iteration of the `dynamic` benchmark. The peak is 1.1 megabytes high. Each color represents the survivors from a 100,000-byte epoch of storage allocation. White represents storage that is more than 1,000,000 bytes old.

for one iteration are shown in Table 4.

100,000 to	200,000 bytes old:	91%
200,000 to	300,000 bytes old:	99%
300,000 to	400,000 bytes old:	99%
400,000 to	500,000 bytes old:	99%
500,000 to	600,000 bytes old:	99%
600,000 to	700,000 bytes old:	99%
700,000 to	800,000 bytes old:	99%
800,000 to	900,000 bytes old:	98%
900,000 to	1,000,000 bytes old:	98%
More than	1,000,000 bytes old:	95%

Table 4: Survival rates by age of object for one iteration of `dynamic`, shown as the percentage that survives the next 100,000 bytes of allocation.

When this pattern of storage allocation is iterated, the survival rates change remarkably. Table 5 shows the survival rates for the entire `10dynamic` benchmark. The oldest objects have the lowest survival rates because the end of each phase involves a mass extinction, killing off both young and old objects. Objects created near the beginning of a phase don't grow old until the phase is well under way. By then the end is coming. Young objects are populous throughout a phase, including the beginning of the phase when they can anticipate a long life.

500,000 to	1,000,000 bytes old:	59%
1,000,000 to	1,500,000 bytes old:	23%
1,500,000 to	2,000,000 bytes old:	1%

Table 5: Survival rates by age of object for `10dynamic`, shown as the percentage that survives the next 500,000 bytes of allocation.

Iterated processes, which are quite common, create survival rates that are the opposite of those predicted by the strong generational hypothesis. It may be possible to exploit these survival rates using a non-predictive collector, even though they arise from a distribution of object lifetimes that does not much resemble the radioactive decay model.

Of the six benchmarks considered here, `nboyer` is the only one that could be cited as evidence for the strong generational hypothesis. The storage pattern shown in Figure 3 arises from recursive duplication and rewriting of a tree that represents the theorem to be proved. Many short-lived objects are allocated during the rewriting of subtrees that are near the leaves. Once a large subtree has been rewritten into its canonical form, however, the storage used to represent that subtree becomes nearly permanent. The survival rates for long-lived objects are shown in Table 6. Although `nboyer` does satisfy the weak generational hypothesis, enough of its young objects survive to cause trouble for generational garbage collectors.

Henry Baker observed that most of the short-lived storage that is allocated by `nboyer` can be eliminated by making the term rewriter check to see whether the subterms it has rewritten are identical (in the sense of a pointer comparison) to the subterms of the term it is rewriting [6]. If they are, then the original term can be returned instead of a copy. This change makes the

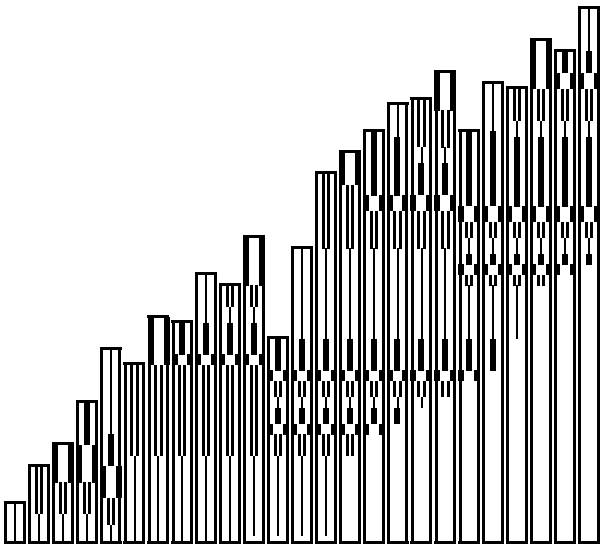

Figure 3: Live storage versus time for the `nboyer1` benchmark; for `nboyer2`, the graph is similar but larger and more complex. The peak is 2 megabytes high. Each color represents the survivors from a 500,000-byte epoch of storage allocation. White represents storage that is more than 5,000,000 bytes old.

mutator a trifle slower, but greatly decreases garbage collection time. The storage allocated by the modified program, `sboyer`, is shown in Figure 4, and its survival rates are shown in Table 7. The strong generational hypothesis is not satisfied, and the weak generational hypothesis is satisfied more weakly than before.

If this change in storage behavior is typical of the changes that occur as a program is tuned for performance, then the garbage collection overhead of production code may have more to do with the overhead of long-lived objects than with the short-lived objects that are the focus of conventional generational collectors.

Non-predictive collectors should perform well when the survival rate is independent of the age of an object, and should perform especially well when the survival rate decreases with age. In other words, non-predictive collectors have the potential to perform well on programs for which the strong generational hypothesis fails.

8 Technical details

In Larceny we will continue to use our conventional generational collector for the young generations, and will use a 2-generation non-predictive collector only for objects that survive long enough to be promoted into the oldest dynamic generation of our conventional collector.

500,000 to	1,000,000 bytes old:	79%
1,000,000 to	1,500,000 bytes old:	92%
1,500,000 to	2,000,000 bytes old:	94%
2,000,000 to	2,500,000 bytes old:	84%
2,500,000 to	3,000,000 bytes old:	97%
3,000,000 to	3,500,000 bytes old:	91%
3,500,000 to	4,000,000 bytes old:	98%
4,000,000 to	4,500,000 bytes old:	98%
4,500,000 to	5,000,000 bytes old:	95%
More than	5,000,000 bytes old:	98%

Table 6: Survival rates by age of object for `nboyer2`, shown as the percentage that survives the next 500,000 bytes of allocation.

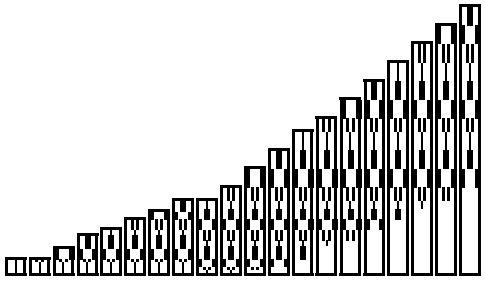

Figure 4: Live storage versus time for the `sboyer2` benchmark. The peak is 1.3 megabytes high. Each color represents the survivors from a 500,000-byte epoch of storage allocation. White represents storage that is more than 5,000,000 bytes old.

Our use of a non-predictive collector for long-lived objects has no direct impact on the performance of our garbage collector for short-lived objects.

Larceny version 0.27 allows for an ephemeral area consisting of one or more generations containing young objects that are managed by a stop-and-copy collector, a dynamic area for older objects that can be managed using a different garbage collection algorithm, and a static area for code, constants, and global data.

Our prototype implementation uses a stop-and-copy collector for all generations. We recognize that this is silly, but for experimental purposes it is easier to compare collectors that use the same basic algorithm throughout. If the prototype works well, we intend to add an alternative 2-generation non-predictive collector based on a mark/sweep algorithm with occasional compaction.

It is often simpler to describe Larceny as it should be rather than as it is, so the word "Larceny" in this section should be understood as an idealized implementation, not as any specific version.

500,000 to	1,000,000 bytes old:	99%
1,000,000 to	1,500,000 bytes old:	100%
1,500,000 to	2,000,000 bytes old:	100%
2,000,000 to	2,500,000 bytes old:	100%
2,500,000 to	3,000,000 bytes old:	100%
3,000,000 to	3,500,000 bytes old:	100%
3,500,000 to	4,000,000 bytes old:	98%
4,000,000 to	4,500,000 bytes old:	95%
4,500,000 to	5,000,000 bytes old:	100%
More than	5,000,000 bytes old:	100%

Table 7: Survival rates by age of object for `sboyer2`, shown as the percentage that survives the next 500,000 bytes of allocation.

8.1 The dynamic tuning parameter

The tuning parameter j can be changed immediately after every non-predictive collection, and its value can be decreased at any time.

Although its value can be viewed as a prediction about the mutator's future behavior, it is more useful to view j as a parameter that can be varied in response to what the mutator has done in the past. In particular, storage that is allocated by the mutator cannot reach the non-predictive portion of the Larceny heap except by surviving at least one collection in the ephemeral area. This gives the garbage collector a chance to see what is coming into the non-predictive heap, and to respond by adjusting the value of j. Some specific responses are mentioned below.

When a non-predictive collection is complete, and the steps have been renumbered, j should be chosen so that steps 1 through j are empty and $j \leq k/2$. Neither of these conditions is absolutely necessary, but they simplify the garbage collector. Subject to these constraints, the choice of j is not terribly critical because it can be reduced later.

If l is the greatest integer such that steps 1 through l are empty, then

$$j = \lfloor l/2 \rfloor$$

seems like a reasonable choice. It is convenient to reconsider this choice whenever a step becomes full.

8.2 Cyclic structures

If the tuning parameter j is chosen as we recommend, then steps 1 through j will be empty following a non-predictive collection and renumbering of steps. This means that any cyclic garbage within the non-predictive portion of the heap will be reclaimed by the next non-predictive collection.

8.3 Size of the remembered set

In a conventional generational collector, the remembered set consists of all objects (or object slots) that contain pointers into a younger generation.

With a non-predictive collector, the remembered set must also contain all objects (or slots) in steps 1 through j that point into steps $j+1$ through k. The remembered set does not have to contain objects in steps $j+1$ through k that point into steps 1 through j.

Objects (or slots) that are in the remembered set only because they point from steps 1 through j into steps $j+1$ through k are kept separate from the rest of the remembered set. See Section 8.6.

If j is chosen as we suggest, then $j \leq k/2$ so it might seem that the non-predictive collector is likely to decrease the size of the remembered set. This is possible, but an increase is more likely. For example, strict functional programs create structures whose pointers almost always point from younger to older objects. For a conventional generational collector, this implies that the remembered set is nearly empty. For a non-predictive collector, this implies that the remembered set may become very large unless the garbage collector acts first.

The garbage collector used for the ephemeral area can count the number of pointers within the ephemeral area that point outside the ephemeral area. This costs very little, because the ephemeral collector must recognize those pointers anyway. This count can be used to estimate the number of entries that will be added to the remembered set when objects within the ephemeral area are promoted into the non-predictive heap. If the current value of the tuning parameter j would cause the remembered set to grow unacceptably large, then its value can be reduced before those objects are promoted.

8.4 Computation of the remembered set

A full collection empties the remembered set and promotes all live storage to the static area. Full collections occur only when requested explicitly by the mutator.

In Larceny, the remembered set contributes to the root set that must be traced during every garbage collection. When an object (or slot) in the remembered set is traced, the garbage collector can determine whether the object still contains any cross-generational pointers that require it to remain in the remembered set. If not, the object can be removed from the remembered set.

This test adds a little to the cost of tracing the remembered set, so it is omitted from the code that is normally used during a minor collection within the ephemeral area.

There are six ways in which an object that resides

within the non-predictive heap could become part of the remembered set. The first three ways can arise with conventional generational collection, and the other three are analogous.

1. The object survives a garbage collection, and contains a pointer into the ephemeral area.
2. The object is promoted from the ephemeral area into the non-predictive heap, and contains a pointer into the ephemeral area.
3. The object is the target of an assignment that stores into it a pointer into the ephemeral area.
4. The object survives a non-predictive garbage collection, resides somewhere in steps 1 through j after the steps are renumbered and the new j is chosen, and contains a pointer into steps $j+1$ through k.
5. The object is promoted from the ephemeral area into steps 1 through j, and the object contains a pointer into steps $j+1$ through k.
6. The object resides somewhere in steps 1 through j, and is the target of an assignment that stores into it a pointer into steps $j+1$ through k.

Situations 1, 2, and 4 do not occur in Larceny. Situation 1 does not arise because a non-predictive collection always promotes all live objects out of the ephemeral area into the non-predictive heap. Likewise situation 2 does not arise because a promoting collection from the ephemeral area into the non-predictive heap promotes all live objects. Situation 4 does not arise because j is chosen so that steps 1 through j are empty following a non-predictive collection.

Situation 5 can only occur during a conventional stop-and-copy collection that promotes live storage from the ephemeral area into the non-predictive heap; it cannot occur during a non-predictive collection. Before the collection begins, Larceny decides whether all objects will be promoted into the generation comprising steps 1 through j or into the generation comprising steps $j+1$ through k; Larceny never allows a promoting collection to promote some objects into steps 1 through j and others into steps $j+1$ through k. Larceny can decide to promote all objects into steps $j+1$ through k without knowing how much storage will be occupied by the promoted objects because it has the flexibility to decrease j following the collection.

Situation 5 is detected during the promoting collection when the object is traced, after it has been copied into the non-predictive heap. When it is traced, each pointer that it contains must be tested anyway to determine whether it points

- to storage that has already been copied out of the ephemeral area,
- to storage within the ephemeral area that has not yet been copied, or
- to storage elsewhere that should not be copied.

Situation 5 cannot arise during the first two cases, because all objects are promoted into the same generation. If objects are being promoted into steps 1 through j, then the last case involves an additional test to determine whether the pointer points into steps $j+1$ through k. The marginal cost of this test appears to be small.

Objects that enter the remembered set via promotion (situation 5) are kept separate from objects that enter the remembered set via side effect (situations 3 and 6). See Section 8.6.

Situations 3 and 6 are detected by the write barrier, which does not distinguish between them.

8.5 The write barrier

Larceny's conventional collector already uses a write barrier, so the marginal cost is very small.

8.6 Cost of tracing roots

The non-predictive collector has no effect on the cost of tracing roots for a collection in the ephemeral area, or for a collection that promotes from the ephemeral area to the dynamic area, because the root set for such collections does not include the portion of the remembered set that records pointers from steps 1 through j into steps $j+1$ through k.

These pointers do form part of the root set for a non-predictive collection, but it appears to be much cheaper to trace only these pointers than it would be to trace every live pointer in steps 1 through j. Even so, this is probably one of the more significant costs that reduce the advantage of non-predictive generational collection.

9 Related work

The original paper on generational garbage collection used decay as a metaphor, and the radioactive decay model is often used in studies of storage management, so it is surprising that there has been no mathematical analysis of generational garbage collection using the model [29, 38].

Henry Baker observed that the radioactive decay model is an important test case for generational garbage collection, and conjectured that generational collection had no advantage over non-generational collection for

the radioactive decay model and a uniform memory system [5, 39].

There is a great deal of empirical evidence to show that most heap-allocated objects die young, even in languages like C [8, 9, 11, 12, 24, 31, 36, 35, 39]. There is no empirical support for the idea that long-lived objects show a strong correlation between age and life expectancy [24, 33, 32, 39].

Equation (1) quantifies a relationship noted by Barry Hayes, who pointed out that if all objects decay at the same rate, as in the radioactive decay model, then the low survival rates that are observed for young objects would imply a negligible number of old objects, contrary to experience. Nevertheless Hayes concluded that "the spectacular differential in reclamation rates between very new objects and slightly older objects is absent at later times" [24].

We observe that it is impossible for survival rates to increase monotonically with age while sustaining a spectacular differential in survival rates at all ages, because the survival rate cannot exceed 100%. If survival rates increase monotonically, then the survival rates of sufficiently long-lived objects must be fairly uniform, as in the radioactive decay model. We also observe that uniform survival rates, or rates that decrease with age, are favorable to non-predictive generational collection.

Larceny's non-predictive collector is a generational collector that uses an unconventional promotion policy for long-lived objects. Promotion policies are usually described in the context of young generations, which are typically managed as a pipeline between the youngest and oldest generations [2, 9, 19, 26, 35, 36]. Policies for long-lived objects have been described by both Moon and Zorn [31, 42].

10 Conclusion

Generational collectors derive part of their efficiency from heuristic selection of the generations to collect. This is an easier problem than heuristic prediction of object lifetimes.

Not even the radioactive decay model can prevent a generational collector from organizing its generations to obtain a computational advantage over non-generational collectors.

In theory, a non-predictive generational collector can exploit survival rates that are independent of age or decrease with age. Some real programs exhibit such rates for long-lived objects, and there are theoretical reasons to believe such rates may be common.

The practical significance of this remains to be seen.

11 Acknowledgements

We are grateful to Henry Baker, Marc Feeley, and Andrew Wright for providing us with source code or ideas for these benchmarks.

References

[1] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions.* National Bureau of Standards Applied Mathematics Series, 55, June 1964.

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

[3] Henry G. Baker. List processing in real time on a serial computer. *CACM* 21(4), April 1978, pages 280–294.

[4] Henry G. Baker. The Boyer benchmark at warp speed. *ACM Lisp Pointers* 5(3), July–September 1992, pages 13–14.

[5] Henry G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Notices* 28(4), April 1993, pages 55–57.

[6] Henry G. Baker. The Boyer benchmark meets linear logic. *ACM Lisp Pointers* 6(4), October–December 1993, pages 3–10.

[7] Henry G. Baker. Personal communication via electronic mail, 6 November 1995, quoting a personal communication via fax from Bob Boyer dated 3 December 1993.

[8] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993, pages 187–196.

[9] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, October 1986, pages 119–130.

[10] C. J. Cheney. A nonrecursive list compacting algorithm. *CACM* 13(11), November 1970, pages 677–678.

[11] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *CACM* 20(2), February 1977, pages 78–87.

[12] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1), January 1979, 51–59.

[13] William Clinger. Further measurements relevant to this paper will be made available through the World-Wide Web. See `http://www.ccs.neu.edu/home/will/GC/index.html`.

[14] William Clinger and Lars Thomas Hansen. Lambda, the ultimate label; or a simple optimizing compiler for Scheme. In *ACM Conference on Lisp and Functional Programming*, June 1994, pages 128–139.

[15] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: framework and implementations. In *ACM Symposium on Principles of Programming Languages*, January 1990, pages 261–269.

[16] Marc Feeley, Marcel Turcotte, and Guy Lapalme. Using Multilisp for solving constraint satisfaction problems: an application to nucleic acid 3D structure determination. In *Journal of Lisp and Symbolic Computation* 7(2/3), 1994, pages 231–246.

[17] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *CACM* 12(11), November 1969, pages 611–612.

[18] Richard P. Gabriel. *Performance and Evaluation of Lisp Programs.* MIT Press, 1985.

[19] Marcelo J. R. Gonçalves and Andrew Appel. Cache performance of fast-allocating programs. In *ACM SIGPLAN-SIGARCH-WG2.8 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, June 1995, pages 293–305.

[20] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[21] Lars Thomas Hansen. *The Impact of Programming Style on the Performance of Scheme Programs.* M.S. thesis, University of Oregon, 1992.

[22] David R. Hanson. Storage management for an implementation of SNOBOL4. In *Software: Practice and Experience* 7(2), March 1977, pages 179–192.

[23] Pieter H. Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. In *Journal of Functional Programming* 6(4), July 1996, pages 621–655.

[24] Barry Hayes. Using key object opportunism to collect old objects. In *Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, October 1991, pages 33–46.

[25] Fritz Henglein. Global tagging optimization by type inference. In *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1992, pages 205–215.

[26] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA '92)*, October 1992, pages 92–109.

[27] IEEE Standard for the Scheme Programming Language. IEEE Std 1178-1990.

[28] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In *Proceedings of the 2nd International Static Analysis Symposium*, Springer-Verlag LNCS 983, September 1995, pages 207–224.

[29] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM* 26(6), June 1983, pages 419–429.

[30] Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. MIT AI Memo 58, 1963.

[31] David Moon. Garbage collection in a large Lisp system. In *ACM Conference on Lisp and Functional Programming*, 1984, pages 235–246.

[32] Patrick M. Sansom and Simon L. Peyton Jones. Generational garbage collection for Haskell. In *Conference on Functional Programming Languages and Computer Architecture*, 1993, pages 106–116.

[33] Darko Stefanović and J. Eliot B. Moss. Characterisation of object behaviour in Standard ML of New Jersey. In *ACM Conference on Lisp and Functional Programming*, 1994, pages 43–54.

[34] George B. Thomas, Jr. *Calculus and Analytic Geometry.* Addison-Wesley, 1968.

[35] David M. Ungar. Generation scavenging: a non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, *ACM SIGPLAN Notices* 19(5), May 1987, pages 157-167.

[36] David Ungar and Frank Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1), January 1992, pages 1–27.

[37] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *ACM Conference on Lisp and Functional Programming*, June 1992, pages 32–42.

[38] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: a survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management*, September 1995, Springer-Verlag LNCS. Available via anonymous ftp from `cs.utexas.edu`, in `pub/garbage`.

[39] Paul R. Wilson. Uniprocessor garbage collection techniques. *ACM Computing Surveys*, to appear. Available via anonymous ftp from `cs.utexas.edu`, in `pub/garbage`.

[40] Feng Zhao. An $O(N)$ algorithm for three-dimensional n-body simulations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[41] Benjamin Zorn. The measured cost of conservative garbage collection. *Software Practice and Experience* 23(7), 1993, pages 733–756.

[42] Benjamin Zorn. Garbage collection using a dynamic threatening boundary. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1995, pages 301–314.