An Experimental Study of Renewal-Older-First Garbage Collection

Lars T Hansen
Opera Software
Oslo, Norway
lth@opera.com

William D Clinger
Northeastern University
Boston, MA 02115
will@ccs.neu.edu

Abstract

Generational collection has improved the efficiency of garbage collection in fast-allocating programs by focusing on collecting young garbage, but has done little to reduce the cost of collecting a heap containing large amounts of older data. A new generational technique, older-first collection, shows promise in its ability to manage older data.

This paper reports on an implementation study that compared two older-first collectors to traditional (younger-first) generational collectors. One of the older-first collectors performed well and was often effective at reducing the first-order cost of collection relative to younger-first collectors. Older-first collectors perform especially well when objects have queue-like or random lifetimes.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Algorithms, Measurement, Performance, Experimentation

Keywords

generational garbage collection, older-first

1 Introduction

Garbage collection is a technology that automatically reclaims unreachable heap storage [24]. (As is common in the literature on garbage collection, we use "live" as a synonym for reachable, and "dead" as a synonym for unreachable.) *Generational* garbage collectors divide the heap into two or more regions, known as generations because they often group objects of similar age, and collect these generations at different times [24, 27]. Most generational garbage collectors attempt to collect younger generations more frequently than older generations, so we call them *younger-first* collectors.

Younger-first generational garbage collectors usually outperform non-generational garbage collectors. Why?

One pop explanation is that "most objects die young", but Henry Baker revealed the inadequacy of this explanation by considering a radioactive decay model of object lifetimes, in which each live object has a 50% probability of dying during the next interval whose duration is the half-life that parameterizes the model [2]. If this half-life is short, then most objects die young, but a conventional generational collector will actually perform worse than a non-generational collector [9].

A more sophisticated explanation for the effectiveness of generational garbage collectors is that they try to predict which objects are likely to die soon, and how well they work depends upon the accuracy of these heuristic predictions. This explanation is also incorrect. The radioactive decay model makes it impossible to predict which objects will die soon. No heuristic predictor can do better or worse than chance, so if this explanation were correct we would expect generational collectors to perform the same as non-generational collectors for the radioactive decay model. In fact, conventional generational collectors perform worse.

How do they manage to do that? By collecting the wrong generations. Younger-first collectors collect young objects more often than old. These young objects haven't had much time to die, so collecting them doesn't recover much storage. An older-first generational collector, which collects old objects more often than young, would recover more storage for a similar amount of effort [9].

Why then do younger-first collectors usually perform well in practice? Because most programs satisfy the *weak generational hypothesis*, which asserts that young objects die at a considerably faster rate than older objects [19]. Hence the young generations often contain a higher fraction of unreachable objects than the older generations, despite the fact that young objects have not had much time to die.

The weak generational hypothesis is important. Its generalization, the *strong generational hypothesis*, postulates a negative correlation between age and mortality rate even for long-lived objects [19, 29, 30, 36]. There is little empirical support for the strong generational hypothesis. Although some of our own data on object lifetimes provide new support for the strong hypothesis, our data also show

ICFP'02, October 4-6, 2002, Pittsburgh, Pennsylvania, USA.

why the strong hypothesis does not matter very much: There cannot be a strong negative correlation between age and mortality rate that holds for objects of all ages, because the mortality rate cannot be less than zero. Even if the strong generational hypothesis holds, the difference between the mortality rate for a group of objects of age t and the mortality rate for older objects will tend to approach zero as t increases.

In other words, the strong generational hypothesis implies that the mortality among sufficiently long-lived objects must resemble that of a radioactive decay model. Since conventional younger-first collectors perform poorly for radioactive decay models, this would in turn imply that conventional generational collection is inappropriate for long-lived objects. This conclusion is remarkable because the strong generational hypothesis had been regarded as the primary justification for younger-first collection of long-lived objects.

In a previous paper, we described a novel algorithm for older-first generational garbage collection, and calculated that, for radioactive decay models of object lifetimes, our new collector should outperform non-generational and younger-first generational collectors. We also described our design and implementation of a hybrid collector that we hoped would combine the advantages of younger-first and older-first collection [9].

In this paper we report on the performance of that hybrid collector.

2 Renewal-Older-First (ROF) Collection

In this section we describe a pure *renewal-older-first* (ROF) generational collector. (This algorithm is exactly the same as the "non-predictive" algorithm that we described previously, but we have adopted the more descriptive name that Darko Stefanović gave to it [9, 31]. Our ROF algorithm described here should not be confused with Stefanović's *deferred-older-first* (DOF) algorithm [17, 31].)

A pure renewal-older-first collector divides the heap into two generations, and always collects the older generation. Instead of grouping objects according to their actual age, however, the ROF algorithm groups objects according to their *renewal age*, which is defined as the time that has passed since the object was last classified as reachable by a collection within its generation, or as its actual age if it has never been considered for collection.

After each collection, therefore, the objects in the older generation that survived the collection are considered to be the youngest objects. We implement this by dividing the heap into *steps* that contain objects of similar age, and relabel the steps following each collection.

A pure ROF collector never performs a full collection.

Figure 1 illustrates the ROF collector as implemented in Larceny, our implementation of Scheme [8, 16, 26]. The steps of the ROF heap are arranged by age, from youngest to oldest; step 1 holds the youngest objects. Additional steps are kept in reserve for use by a copying collector, and are not available for allocation. A policy parameter j determines the dividing line between the younger and older generations of the ROF heap. Steps $1 \ldots j$ are in the younger generation, and steps $j+1 \ldots k$ are in the older.

When the dynamic area is collected, the boldly outlined older generation (steps $j+1 \ldots k$) is collected by evacuating its live data into the reserve. Then the steps are rearranged: the younger genera-

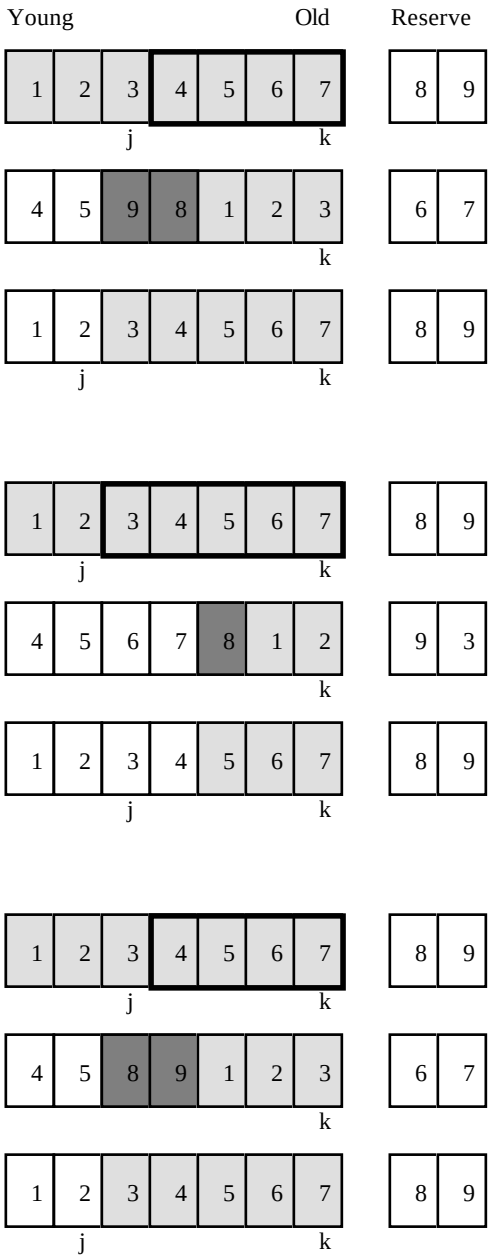

Figure 1. The ROF collector as implemented in Larceny. The figure shows triples of collector configurations: before a collection, after a collection but before renumbering of steps and selection of j, and after renumbering and selection of a new value for j. Shaded steps are full, and unshaded steps are empty. The thick frame surrounds the steps that are subjected to garbage collection, and the heavy shading shows the steps that contain the survivors of the collection. Steps that are not collected are simply moved.

tion (steps $1 \ldots j$) become the oldest steps (steps $k-j+1 \ldots k$) of the ROF heap, and the shaded survivors of the collected generation (the s steps of the reserve) become the youngest steps ($k-j-s+1 \ldots k-j$). (That is, the survivors are treated as if they were newly allocated objects; this is what gives the renewal-older-first collector its name.) Some of the free steps are used to replenish the reserve, and the remaining free steps become available for allocation (steps $1 \ldots k-j-s$).

Following collection, Larceny sets j to $1/2$ the number of free steps. This ensures that large circular garbage structures will be collected and that the younger generation of the ROF heap will not be unreasonably large.

3 Hybrid (3ROF) Collection

We implemented a hybrid ROF collector by allowing any number of younger-first generations to precede the ROF heap. The hybrid ROF collector usually performs best with only one younger-first generation, so that is the configuration we describe here.

Our 3ROF collector is a 3-generational collector that consists of two generations that are collected by the ROF algorithm, plus a youngest generation, the *nursery*, which is collected and evacuated as part of every collection. The hybrid algorithm is younger-first in the sense that it collects the nursery most often, but is older-first in the sense that the oldest generation is collected more often than the intermediate generation.

In fact, the intermediate (younger ROF) generation is never collected at all. Dead objects within that generation are collected only after they have been folded into the older ROF generation.

In addition to the usual write barrier for mutator code, the 3ROF collector itself incorporates an additional write barrier that records pointers from the intermediate (younger ROF) generation into the older ROF generation. These pointers are created when objects are promoted from the nursery into the intermediate generation, and must be traced during a major collection. This additional write barrier is used only during a minor collection that promotes into the intermediate generation; the mutator does not use it, nor does a major collection, nor does a minor collection that promotes directly into the older ROF generation.

Even so, the barrier adds a cost to promotions, and this cost is particularly noticeable in Scheme programs that use many pairs. The standard collector has been tuned to handle pairs particularly efficiently, and the additional cost of the write barrier, though low in absolute terms, makes up a substantial fraction of the cost of copying and scanning a pair in the hybrid ROF collector. This cost would be less significant in languages like Java.

4 Larceny and its Collectors

This section describes our Larceny implementation of Scheme and some characteristics of its garbage collectors [8, 16, 17, 23, 25].

4.1 Compiler

Larceny uses the Twobit optimizing compiler to compile Scheme to SPARC machine code. Previous measurements have established that Twobit and Larceny together have performance that is roughly competitive with Standard ML of New Jersey and with commercial Common Lisp and Scheme systems [8]. A few preliminary benchmarks also suggest that Larceny's default generational garbage collector is competitive with the default collector used in Sun's HotSpot Java system.

Twobit incorporates a number of optimizations that limit heap allocation largely to allocation performed explicitly by the source program. In particular, neither continuation frames nor environment structures are allocated on the heap unless the program explicitly captures the continuation or creates a closure using a lambda expression that escapes (as determined by a first-order closure analysis). Program variables that are updated by assignment are heap-allocated, but Scheme programs are largely functional in nature and heap-allocated variables are few in practice.

4.2 Collectors

Larceny currently supports five interchangeable garbage collectors, including four precise collectors that use Cheney's copying algorithm: a conventional younger-first generational collector, two different (ROF and DOF) hybrid older-first collectors, and a non-generational stop-and-copy collector [7]. The three generational collectors also use a non-copying algorithm to collect the large-object (< 4 Kby) spaces that are associated with each generation [20].

Generations are grouped into three areas. The *ephemeral area* contains the youngest collected generations, the *dynamic area* contains the oldest collected generations, and the *static area* contains objects that are permanent. The static area is never collected during normal operation. In addition, some memory is kept in a reserve for copying collection.

The youngest ephemeral generation is called the *nursery*; all object allocation takes place in that generation. Objects are allocated by incrementing an allocation pointer; the pointer is kept in a machine register, and allocation is very fast.

Garbage collection is driven by allocation: When the nursery is full, the collector evacuates all live objects from the nursery into some older generation. The collector may also choose to collect other generations at that time.

A collection that collects only the nursery is known as a *minor* collection. A collection that collects the entire heap is a *full* collection. A collection that collects at least one generation in addition to the nursery but does not collect the entire heap is classified as a *major* collection.

The heap may be resized following a major or full collection. All collectors use a command-line parameter $1/L$, called the *load factor*, to compute the new heap size. The inverse load factor L tells the collector how much memory it is permitted to use, as a multiple of the amount of reachable storage l. The new heap size H' is computed as $H' = lL$. Since l memory is already live, and l memory will be needed as reserve for the next garbage collection, the amount of memory available for allocation is $H' - 2l$.

4.3 Write Barrier and Remembered Set

Generational collectors use a write barrier and a remembered set to keep track of intergenerational pointers, which come into play when only part of the heap is collected.

Program	Lines of code	Allocation volume	Peak live (est.)	Promotion rate	Mutator time (stop+copy)	Major GC (msec)	Minor GC (msec)	Ratio	
5earley:12	658	299.0	13.5	0.21	11.04	1392	1684	0.83	
5earley:13	658		35.0		53.17	7455	4045	1.84	*
gcbench:5	226	1757.0	16.8	0.27	19.12	18824	10534	1.79	
gcold:25,1,0	381	347.0	26.5	0.44	5.37	6947	4072	1.71	
gcold:25,1,1000	381	402.0	26.5	0.38	40.56	7025	4186	1.68	
gcold:25,10,100	381	264.0	26.5	0.06	36.89	919	538	1.71	
gcold:100,1,0	381	1389.0	101.5		27.35	28840	11798	2.44	*
gcold:100,1,1000	381	1606.0	101.5		121.72	30282	11962	2.53	*
nboyer:3	767	99.0	13.0	0.45	4.30	1498	1154	1.30	
nboyer:4	767	266.5	35.0		7.98	4905	2708	1.81	*
nboyer:5	767	846.0	100.0		23.42	22660	8692	2.61	*
5nboyer:3	767	497.0	13.0	0.44	20.40	6784	5169	1.31	
5nboyer:4	767	1470.0	35.0		38.77	21535	12840	1.68	*
5sboyer:4	781	259.0	13.4	0.26	61.41	1831	1816	1.01	
5sboyer:5	781	687.0	30.0		121.77	7000	4218	1.66	*
perm:200,8,1	324	229.0	11.5	1.00	4.44	11393	3955	2.88	
perm:25,8,8	324	229.0	11.5	1.00	4.55	5578	4262	1.31	
perm:200,9,1	324	2059.0	100.0	1.00	30.43	82003	33550	2.44	*
perm:25,9,8	324	2059.0	100.0	1.00	30.53	81742	33125	2.47	*
twobitlong	23,789	665.0	7.9	0.08	138.13	892	4009	0.22	
twobitshort	23,789	119.5	7.5	0.18	22.17	652	907	0.72	
5twobitshort	23,789	575.5	7.5	0.17	108.17	2804	3461	0.81	

Table 1. Characteristics of the benchmark programs. Allocation and peak live volume are reported in megabytes. The promotion rate is the fraction of allocation that is promoted out of a 1 megabyte nursery in the generational collectors. The mutator time, in seconds, is the average across several runs of the stop-and-copy collector. The gc times, in milliseconds, are the average across several runs of the 2GEN collector. The gc times for major and minor collections are reported separately, and their ratio is shown. An asterisk (*) in the last column indicates that the benchmark was run by Clinger on a bigger and slightly faster machine.

In an assignment `*lhs=rhs` the write barrier first determines whether `rhs` is a pointer, and if it is, performs table lookups on `lhs` and `rhs` to determine their generation numbers. If the generation number of `rhs` is less than that of `lhs`, then `lhs` is inserted into a sequential store buffer, which will later be folded into the remembered set [21]. Larceny's remembered set currently uses hash tables to filter duplicates from its component subsets.

Larceny's older-first collectors require objects to be recorded in several subsets at the same time. That requirement makes card marking and header marking less attractive, since each card or object would need one mark bit for each subset of the remembered set in which it might be recorded.

For the benchmarks reported in this paper, the size of the extra remembered subset that is required by the 3ROF collector was limited to 32768 entries. When the size of that subset exceeded this limit, that remembered subset and the intermediate (younger ROF) generation were both cleared by reducing the value of the parameter j that determines the boundary between the younger and older ROF generations. This effectively protected the 3ROF collector from excessively large remembered sets by allowing it to degrade into a conventional 2-generational younger-first collector.

For more engineering details on the write barrier and remembered set, see our earlier paper and Hansen's PhD thesis [9, 17].

5 Benchmarks

Many programs pose little challenge to even a simple garbage collector, usually because they have little live data or a low rate of allocation. Furthermore all of the generational algorithms (2GEN, 3GEN, 3ROF) perform well even on most allocation-intensive programs, because those programs usually satisfy the weak generational hypothesis.

We are therefore interested primarily in how well a collector performs on programs that are abnormal in the sense that garbage collection accounts for a substantial fraction of their execution time. Most of these abnormally gc-intensive programs do not satisfy the weak generational hypothesis as well as more typical programs, which results in an unusually high rate of promotion out of the youngest generation—instead of the typical 1% promotion rate, gc-intensive programs may have promotion rates of 10–100%.

The benchmarks we selected are small Scheme programs that we knew to be gc-intensive, plus one larger benchmark (an optimizing Scheme compiler) that is not particularly gc-intensive but had been observed to perform poorly with the Boehm-Demers-Weiser conservative (imprecise) collector. Three of the programs are synthetic garbage collection benchmarks, which are especially useful for studying the best-case and worst-case behavior of garbage collectors.

Some of the other programs make sense as garbage collection benchmarks only if they are run several times back-to-back in the same process and measurements are taken for all the iterations as a whole. Neither `earley` nor `sboyer` are suitable uniterated, because their live storage grows monotonically and most garbage is short-lived, so these programs reach their peak size with little opportunity for garbage collection of older objects. Although the use of iterated benchmarks is a common practice, it should be noted that iteration skews the distribution of object lifetimes in a way that should favor older-first collection.

These benchmarks are available at our web site [10].

	Mark/cons ratio			Major GCs			Rem. Set (Mby)		
	2GEN	3GEN	3ROF	2GEN	3GEN	3ROF	2GEN	3GEN	3ROF
5earley:12	0.34	0.60	0.32	8.37	7.79	7.95	0.27	0.40	0.84
5earley:13				17.00	15.00	15.25			1.30
gcbench:5	0.69	0.43	0.89	84.26	31.07	122.44	0.27	0.40	0.53
gcold:25,1,0	1.02	1.61	0.77	10.84	11.05	8.95	0.27	0.40	0.53
gcold:25,1,1000	0.90	1.54	0.73	11.00	11.21	9.53	0.27	0.40	0.54
gcold:25,10,1000	0.14	0.29	0.14	1.16	0.95	1.21	0.27	0.40	0.55
gcold:100,1,0	1.22	1.84	0.91	14.25	14.50	11.50			0.54
gcold:100,1,1000	1.07	1.70	0.80	14.50	14.50	11.25			0.57
nboyer:3	0.87	1.16	0.90	7.00	4.88	7.27	0.27	0.40	0.99
nboyer:4	1.15	1.44	1.06	12.00	9.50	10.75			1.15
nboyer:5	1.44	1.80	1.32	21.00	17.00	18.50			1.60
5nboyer:3	0.87	1.15	0.90	33.71	24.67	35.40	0.27	0.40	1.05
5nboyer:4	1.04	1.36	0.97	52.00	41.75	46.25			1.26
perm:200,8,1	2.76	3.26	2.50	40.50	36.69	40.56	0.27	0.40	0.53
perm:25,8,8	1.74	2.41	1.36	29.00	27.20	25.89	0.27	0.40	0.53
perm:200,9,1	2.80	3.70	2.39	38.25	37.25	31.00			0.53
perm:25,9,8	2.65	3.56	2.25	59.00	57.5	47.75			0.53
5sboyer:4	0.46	0.77	0.45	10.31	9.69	10.50	0.27	0.40	0.91
5sboyer:5	0.66	1.03	0.61	17.25	15.25	15.00			1.18
twobitlong	0.12	0.15	0.11	8.53	3.37	7.89	0.29	0.45	0.84
twobitshort	0.32	0.65	0.30	3.89	2.95	3.58	0.34	0.49	0.90
5twobitshort	0.31	0.68	0.29	18.79	12.89	17.06	0.34	0.53	1.00

Table 2. Averages, across all runs, for mark/cons ratio, major garbage collections, and peak size of remembered set.

	2GEN	3ROF (for varying L)					3ROF / GEN
		2.25	2.5	2.75	3.0	Mean	
5earley:12	24.6	27.6	29.8	31.7	33.3	30.6	1.24
gcbench:5	23.8	26.6	26.7	27.1	27.6	27.0	1.13
gcold:25,1,0	25.8	27.4	27.8	28.8	30.1	28.5	1.10
gcold:25,1,1000	26.4	27.5	28.2	29.5	31.0	29.0	1.10
gcold:25,10,1000	35.3	33.8	34.3	38.2	37.2	35.9	1.02
nboyer:3	24.0	28.8	33.6	34.5	38.0	33.7	1.40
5nboyer:3	23.5	29.3	30.5	32.4	33.4	31.4	1.34
perm:200,8,1	19.7	22.6	23.8	24.4	24.9	23.9	1.21
perm:25,8,8	19.8	22.3	23.2	24.7	25.3	23.9	1.21
5sboyer:4	26.2	30.3	33.7	36.3	38.6	34.7	1.32
twobitlong	78.8	89.4	89.4	91.6	94.7	91.3	1.16
twobitshort	45.5	49.7	49.4	49.5	53.4	50.5	1.11
5twobitshort	36.3	44.6	45.8	47.7	48.6	46.7	1.29

Table 3. Average promotion cost per volume (ms/MB)

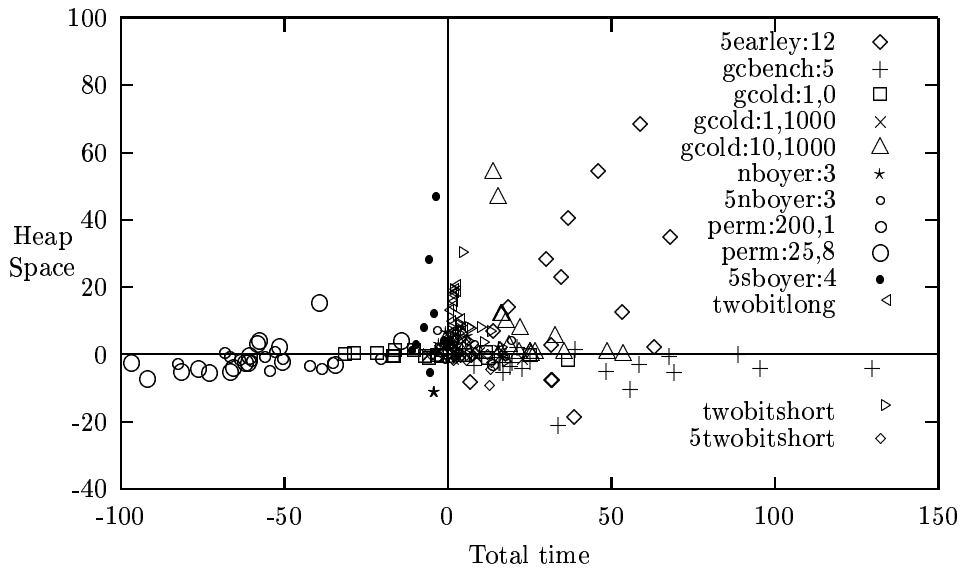

Figure 2. A space/time diagram comparing Larceny's stop-and-copy and 2-generational younger-first collectors. Negative coordinates indicate that the stop-and-copy collector is better; positive coordinates indicate that the 2-generational collector is better. For example, the point at $\langle 68, 35 \rangle$ shows that on this particular run of `earley`, the stop-and-copy collector requires 68% more CPU time and 35% more heap space than the 2-generational collector.

`earley` is an implementation of Earley's parsing algorithm, written by Marc Feeley [13]. In our benchmarks, this program first creates a parser from an ambiguous grammar, and then computes all parses of an input of length 12 or 13 that has an exponential number of parse trees; this is iterated 5 times. `earley` constructs its output in a functional manner, and the resulting data structure necessarily contains all young-to-old pointers.

`gcbench` is a synthetic garbage collection benchmark written by John Ellis and Pete Kovac, modified by Hans Boehm, and translated from Java into Scheme by Will Clinger. To reduce edge effects, we modified `gcbench` to perform its stretching phase once and then to iterate the rest of the program n times. `gcbench` was run with $n = 5$.

`gcold` is a synthetic garbage collection benchmark written by David Detlefs and translated from Java to Scheme by Will Clinger. The program creates a number of large trees and then repeatedly replaces random subtrees with newly allocated trees and moves subtrees around in the data structure by swapping them.

`gcold` was run in five configurations. Three configurations have 25 megabytes of large trees and run for 200 iterations, and the other two configurations have 100 megabytes of large trees and run for 800 iterations. At each of these two sizes, the configurations differ in the ratio of short-lived to long-lived storage and the amount of mutation work: we used the settings $(1, 0)$, $(1, 1000)$, and $(10, 1000)$, omitting the third setting for the larger size.

For each size, the different configurations were intended to allocate the same amount of storage, but several problems with the benchmark code that were discovered late prevent the allocation volumes from being equal. We elected not to correct these problems, which makes the results from this benchmark a little harder to interpret.

`boyer` is a toy term-rewriting theorem prover derived from the Boyer benchmark in the Gabriel benchmark suite [14]. The program uses pairs extensively and constructs its data structures functionally. The versions used here, `nboyer` and `sboyer`, fix some bugs, are written in portable Scheme, and incorporate a problem scaling parameter [1, 3, 4, 10]. `nboyer` and `sboyer` differ only in that `sboyer` uses a local tweak—shared consing—to reduce the amount of storage allocation; this tweak results in a radically different live storage profile for `sboyer` [3, 10, 17].

`nboyer` was run with scaling parameters 3, 4, and 5, as a single iteration or iterated five times. `sboyer` was run with scaling parameters 4 and 5, iterated five times.

`perm` is a synthetic garbage collection benchmark written by Will Clinger, Gene Luks, and Lars Hansen. It represents the worst case for a younger-first generational collector: no objects die young, and all object deaths are oldest-first.

The program maintains a queue of data structures, each representing all permutations of the first N integers. As new data structures are allocated, old data structures are removed from the queue and become garbage.

For $N = 8$ there are 40,320 permutations in each list, occupying about 1.2 megabytes (sharing is extensive). The lists are generated without creating any garbage whatsoever, and with the settings we have chosen the survival rate out of the nursery is 100%.

`perm` was run in four configurations. `perm200,8,1` and `perm25,8,8` allocate the same amount of data—computing all permutations of 8 things 200 times—and have the same peak live size, but differ in the lifetime of the data: the former removes one datum from the queue on every iteration, the latter removes eight.

`twobit` is the Twobit optimizing compiler for Scheme [8, 25]. It is a typical old-style Lisp program: lists are used to represent

many data structures, and most of the objects allocated are pairs. `twobit` creates graph representations of the program being compiled and then annotates and updates those representations using side effects.

`twobit` is run in two configurations. The `twobitshort` benchmark compiles another program, Nucleic2, in whole-program optimization mode. Nucleic2 is about 3200 lines of source. The `twobitlong` benchmark compiles the source for `twobit` itself, about 23,800 lines.

The measurements for `twobit` include all allocation for I/O, but the time measurements do not include time spent waiting for I/O.

Some characteristics of these benchmark programs are shown in Table 1. The peak live sizes include the size of the static area because the collectors include it when they compute the heap size, and the peak live size for `earley` includes a worst-case stack of 1.4 megabytes.

An important question about the benchmarks is whether enough time is spent in garbage collection in the dynamic area. The suite will not be a good test of dynamic-area collection algorithms if garbage collection time is dominated by minor collections. Table 1 shows that the larger fraction of the time is spent on major collections in most of these programs, but 55%–60% of the GC time on `twobitshort` and fully 80% of the GC time on `twobitlong` are spent promoting data. Thus, the opportunities for the dynamic-area collector to improve the collection times are good in most cases, but notably restricted in the case of `twobitlong`.

6 Measurements

Hansen performed most of our measurements on a Sun Ultra 5 workstation running Solaris 2.6. It had an UltraSPARC-IIi processor running at 333 MHz, 2 megabytes of secondary cache, and 128 megabytes of RAM. It provides CPU time accounting with a resolution of 10 milliseconds. Measurements were obtained while this machine was connected to a network and operating in multi-user mode, but no other users were on the system. X Windows was running on the machine's console but was inactive.

Clinger supplemented these measurements by running larger versions of the more scalable benchmarks on a Sun Ultra 80 Model 4450 with 4 UltraSPARC-II processors running at 450 MHz, 4 megabytes of secondary cache, and 2 gigabytes of RAM. These benchmarks were run using a newer version of Larceny, without testing as many configurations or collecting as much data. Their timings are reported separately in Figures 5 and 6.

To reduce measurement noise, we report CPU times and the best of three runs (rather than elapsed time and the average of three runs, say). Paging was never an issue.

In reporting the space used by a collector, we count all heap space that is actually allocated by the collector, exclusive of storage for remembered sets. (Some representations for remembered sets are more compact than others, so our representation could be regarded as biased. We report the space required by our remembered sets separately in Table 2.)

Efficient garbage collection is a tradeoff between time and space, so when benchmarking it is important to control the amount of space that the different collectors use. Unfortunately, it is not much easier to keep space constant across collectors while measuring gc time than it is to hold the time constant while measuring space. Our compromise is to report both time and space as in Figure 2, while attempting to control space in three different ways:

- One set of runs placed an upper limit on the heap size, though each collector was free to use less memory than the limit. These runs were conducted with five heap sizes, spaced at least two megabytes apart, ranging between two and three times the peak storage required by the benchmark.
- Another set of runs placed both upper and lower limits on the heap size. In all runs the upper limit was set as above, and the lower limit was set 2 megabytes below the upper limit; thus the collectors had a little room to maneuver, but not much.
- The third set of runs instructed the garbage collector to resize the heap as necessary to use no more than L times the collector's estimate of live storage. These runs were conducted for $L = 2.25$, 2.5, 2.75, and 3.0.

In addition to measuring CPU time, we measured the number of words of storage that a collector copies during the execution of a benchmark. Dividing this by the number of words allocated by the benchmark yields the *mark/cons* ratio. The mark/cons ratio allows us to separate the abstract (theoretical) efficiency of a garbage collector from its concrete cost of copying a word, which has a lot to do with how tightly the collector's inner loops are coded. Both the mark/cons ratio and the cost of copying a word are of interest, and both together are more informative than CPU time alone would be.

	Percent		Volume (Mby)	
	3ROF float	Promo float	3ROF float	Promo float
`5earley:12`	0.0	0.8	0.0	0.6
`gcbench:5`	17.5	48.5	149.5	278.7
`gcold:25,1,0`	0.0	0.0	0.0	0.0
`gcold:25,1,1000`	8.2	0.0	19.1	0.0
`gcold:25,10,1000`	35.7		8.6	
`nboyer:3`	0.0	0.0	0.0	0.0
`5nboyer:3`	0.0		0.0	
`perm:200,8,1`	0.0	0.0	0.0	0.0
`perm:25,8,8`	-1.2	0.0	-3.4	0.0
`5sboyer:4`	0.0	0.0	0.0	0.0
`twobitlong`	4.1	16.1	2.9	9.7
`twobitshort`	0.2	11.4	0.1	3.5
`5twobitshort`	2.4	31.3	3.7	37.6

Table 4. 3ROF and promotion float, by average percentage of excess copying and average excess volume copied (in megabytes), for $L = 3.0$. The volumes of promotion float are very close to the volumes reported for 2GEN. Blank entries were not measured but will be zero.

7 Experimental Results

In this section we compare the performance of Larceny's hybrid 3ROF collector to Larceny's conventional 2-generational (2GEN) and 3-generational (3GEN) younger-first collectors.

The space/time diagrams in Figures 3, 4, and 5 provide a quick impression of how these collectors compare with respect to space and overall CPU time. Figure 6 shows how the 2GEN and 3ROF collectors compare with respect to garbage collection time for the larger benchmarks. Many more figures and tables can be found in Hansen's thesis, which is online [17].

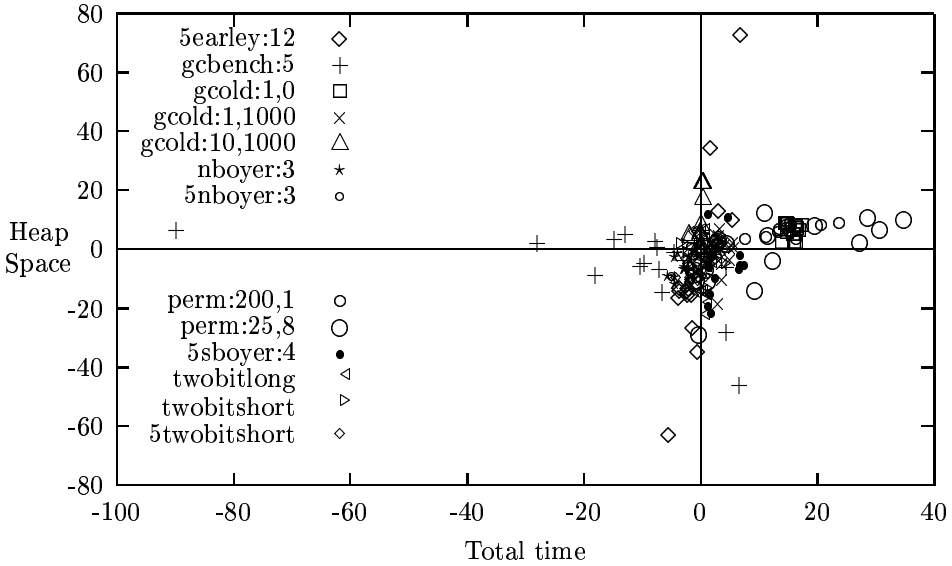

Figure 3. Space/time diagram comparing 2GEN (better along the negative axes) and 3ROF (better along the positive axes).

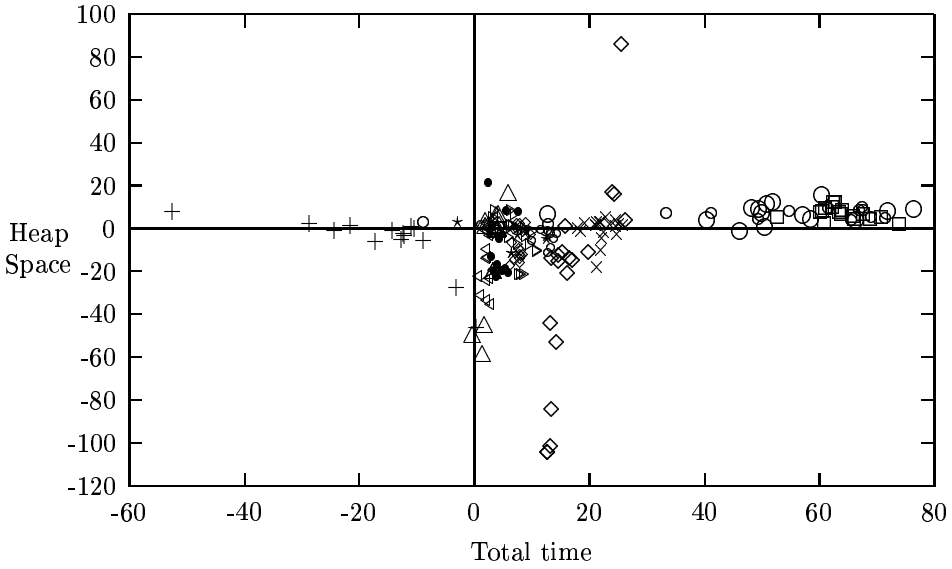

Figure 4. Space/time diagram comparing 3GEN (better along the negative axes) and 3ROF (better along the positive axes).

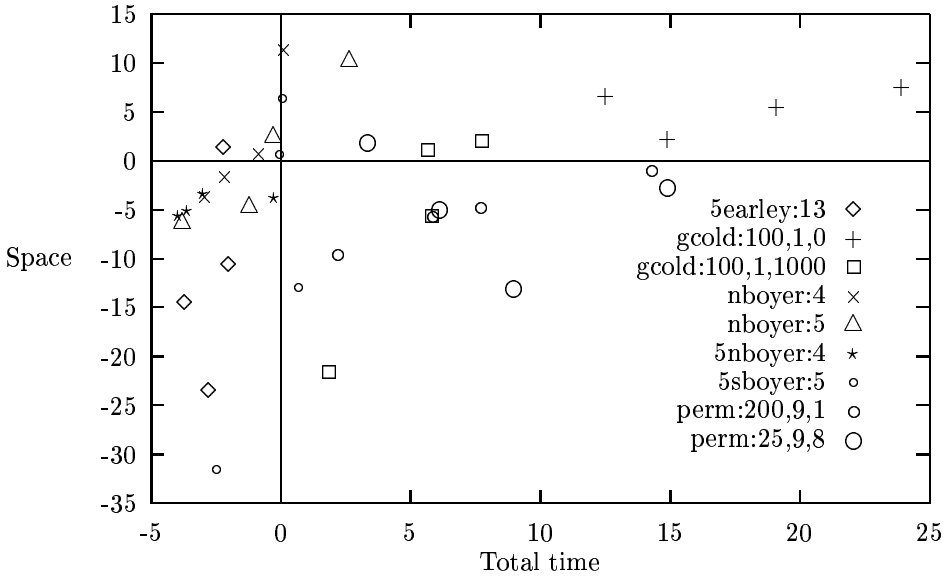

Figure 5. Space/time diagram comparing 2GEN (better along the negative axes) and 3ROF (better along the positive axes) for the larger benchmarks run on a bigger machine. The x-axis is total CPU time.

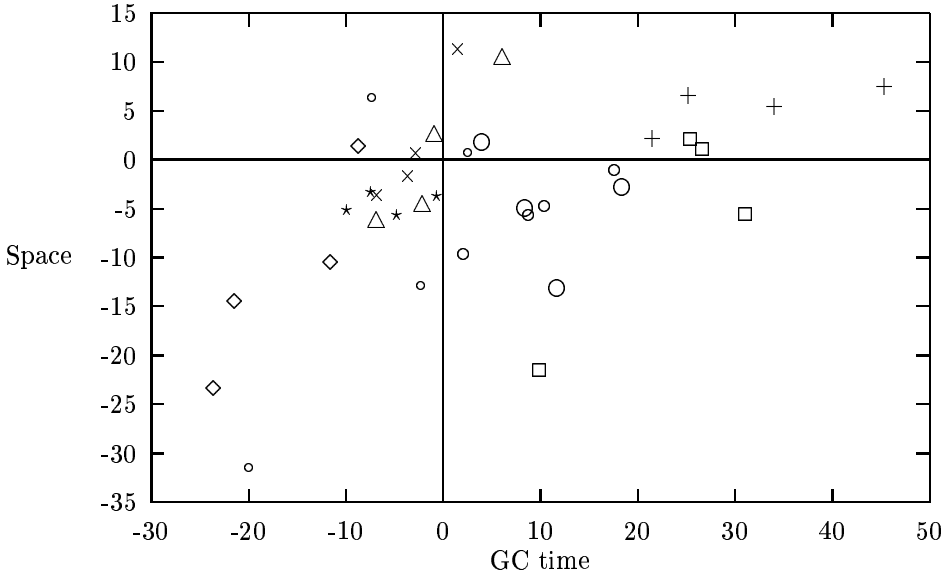

Figure 6. Space/gc-time diagram comparing 2GEN (better along the negative axes) and 3ROF (better along the positive axes) for the larger benchmarks run on a bigger machine. The x-axis is CPU time expended on garbage collection.

7.1 Summary of Findings

The 2GEN collector had the best overall performance, the 3ROF collector was a close second, and the 3GEN collector was a distant third.

It was mildly surprising that 2GEN outperformed 3GEN on most of our benchmarks. We have identified several reasons for this, but the most important is that, by definition, gc-intensive benchmarks are atypical. In particular, they often have an unusually high fraction of fairly long-lived but not permanent objects.

The 2GEN collector promotes these objects by copying them just once, whereas the 3GEN collector must copy them at least twice before they enter the oldest dynamic generation. The 3ROF collector promotes these objects into the intermediate generation by copying them just once, and promotes them into the older ROF generation without copying them, where they often died without being copied a second time.

Table 2 shows that the 3ROF collector's mark/cons ratios are frequently better than those of the younger-first collectors. It should be noted that 2GEN and 3GEN sometimes use less heap memory than 3ROF, and lower heap sizes tend to pull up the averages for the mark/cons ratio and number of major collections, making 2GEN and 3GEN look worse relative to 3ROF than they actually are. It is clear, however, that the 3ROF collector often marks fewer words than the younger-first collectors, especially on the `gcold`, `perm`, and the larger `nboyer` and `sboyer` benchmarks.

Only on one program, `gcbench`, does the 3ROF collector have worse mark/cons ratios, and that program has object lifetimes that just do not fit the models for which the 3ROF collector was designed. With `gcbench`, most of the objects in the younger ROF generation will be dead when a major collection occurs, and the permanently live storage of that benchmark will account for a large fraction of the data in the older ROF generation. Therefore the major collection will reclaim a smaller amount of storage than would a full collection with 2GEN, and the frequency of collections must increase. 3GEN does even better than 2GEN on this one benchmark because, once it has promoted the permanent storage into the oldest generation, it doesn't have to look at it again except during full collections. Adding a fourth generation to the 3ROF collector would improve its performance on `gcbench`, but whether this can be done without compromising its performance on other programs is an open question.

Except for `gcbench`, the 3ROF collector was faster than the 3GEN collector, but sometimes used more space.

The 3ROF collector did not always compare well against 2GEN, as it tends to use more space and has a higher average cost per word promoted out of the nursery. The 3ROF collector did perform impressively on the `gcold` and `perm` benchmarks, however. `gcold`'s random mutations give it object lifetimes that resemble a radioactive decay model, and `perm` has queue-like lifetimes, both of which favor older-first collection.

The 3ROF collector usually spends about as much time in major collections as the 2GEN collector, but its minor collections tend to be more expensive, so its total gc time is often slightly higher. On the smaller benchmarks, this is often offset by a lower mutator time, despite the fact that the mutator code is identical for both [17]. Unlike 2GEN, the 3ROF collector never performs a full collection, which may have some advantages for cache performance. This effect did not show up in the larger benchmarks, possibly because their heaps are much larger than the hardware caches.

The 3ROF collector has more floating garbage than the younger-first collectors, and requires more space for its remembered sets, but in both cases the increases are moderate.

We have not calculated averages or geometric means across benchmarks because our benchmark programs are unlikely to be representative even of gc-intensive programs, and we have further distorted the averages by running more benchmarks with some programs (`gcold`) than others (`gcbench`). Also, averages would tend to direct a reader's attention toward the goal of optimizing for the (ill-defined) average gc-intensive program, when the more promising and important goal is to match each program to a collector that helps it to perform well.

7.2 Cost of Copying

Though the 3ROF collector usually has a lower mark/cons ratio than 2GEN, its GC times are often higher, and the reason is that it is more expensive to perform some collections. The collector must use a write barrier during promotions into the ROF-young generation (steps $1 \ldots j$) to allow the remembered set to track pointers into the ROF-old generation; it must also insert any intercepted pointers into the remembered set. The write barrier is not particularly expensive in itself, nor is remembered set insertion expensive, but the cost of copying and scanning a word is already small, so even the small absolute costs of the extra operations add up to a substantial relative cost. Table 3 shows the cost, expressed in milliseconds per megabyte, of promoting into the dynamic area (both generations). This is not pure copying cost: it also includes the cost of scanning the remembered set, which in some cases will be slightly larger than for the remembered set of 2GEN.

The high promotion costs for `twobit` are caused by a performance bug that affects all of Larceny's generational collectors [17].

The 3ROF collector's unit promotion cost rises with the inverse load factor L because a larger value of L increases the proportion of promotions into the younger ROF generation, where the extra write barrier comes into play.

7.3 Floating Garbage

The garbage collector must assume that all objects in the remembered set are live. That is not always true, as objects may die after they have been inserted into the remembered set but before their generation is collected. When the dead objects in the remembered set refer to other dead objects in the collected region, those dead objects must be copied by the collector. They are called floating garbage, or just float.

We measured the amount of float in the 3ROF collector by using a special mark/sweep collector that marks all live objects but sweeps only the remembered set. This operation does not disturb the heap, so the resulting reductions in copying are accurate representations of how the programs are affected by float.

Table 4 shows *3ROF float* (float due to major collections not being full collections) and *promotion float* (float due to minor collections that promote into the ROF heap), across all settings for the 3ROF collector. Overall, 3ROF float is moderate across the benchmarks—the high value shown for `gcold:10,1000` results from the very

small number of collections recorded for this benchmark. Promotion float is often high, but we measured a comparable amount of promotion float for the 2GEN collector, so the 3ROF collector does not appear to create much more float than conventional generational collectors.

7.4 Remembered Sets

The 3ROF collector uses more space for the remembered set than either of the younger-first collectors, and in many cases the remembered set grows to over 3% of the heap size (see Table 2).[1] The reason for the larger remembered set is that the 3ROF collector needs to track pointers from the ROF-young generation into the ROF-old generation. Many data structures in typical Scheme programs create young-to-old pointers, so the extra remembered set is expected to grow large in some of these programs [9].

More imperative languages, such as Java, have less bias toward young-to-old pointers.

8 Related Work

There are now several general surveys of garbage collection [11, 24, 36].

Generational collection and the weak generational hypothesis were first described in 1983 by Lieberman and Hewitt [27], although earlier work foreshadowed the technique [5, 12, 18]. Lieberman and Hewitt motivated the younger-first technique by appealing to the predominance of pointers from young data to old data, though they presented no evidence that such a predominance existed. Hayes stated and cast doubt on the strong generational hypothesis [19].

Important early contributions to generational collection were made by Ungar and Moon, both of whom described actual implementations [28, 34].

Not all generational collectors have been strictly younger-first. Moon reported that the garbage collector of the Symbolics Lisp machine would only collect those ephemeral generations that were full [28], and the mature object space collector (usually called the train algorithm), invented by Hudson and Moss, will collect data in an order that is similar to older-first but that is also influenced by the topology of the heap [15, 22].

Spurred by Baker's conjecture that generational collection had no theoretical advantage over non-generational collection for the radioactive decay model, we invented the ROF algorithm described here, and showed its theoretical advantage by calculation [2, 9]. We also outlined the implementation of our prototype 3ROF collector, but did not provide details of its performance [9].

Stefanović invented a better name for our ROF collector, which we had originally called "non-predictive," and invented several other older-first algoritms, notably his deferred-older-first (DOF) algorithm [31, 32]. His simulations of these algorithms showed that his DOF algorithm has lower mark/cons ratios than the other algorithms on a suite of Smalltalk and Java programs, and he argued that a DOF collector should also perform well in practice. Implementations of the DOF and the related Beltway collectors in Java have now confirmed this [6, 33].

Hansen implemented and benchmarked a hybrid version of the DOF algorithm as well as the hybrid 3ROF algorithm presented here. The hybrid DOF algorithm was usable but did not perform as well on our benchmarks as the 3ROF algorithm. Hansen's measurements revealed two main reasons for this: a pure DOF algorithm would have performed better than our hybrid version, and the DOF algorithm tends to create excessive amounts of floating garbage [17].

9 Conclusions and Future Work

Older-first generational garbage collection is a new technology that greatly expands the design space for generational garbage collectors. For example, the DOF and Beltway algorithms are inherently less disruptive than our ROF collector, which is in turn less disruptive than conventional younger-first collectors.

It would be desirable to replicate our experiments with more benchmarks, in other systems, and for other programming languages.

Clinger has been using linear combinations of radioactive decay models with different half-lives to model object lifetimes, and to analyze the theoretical performance of idealized algorithms for garbage collection. This work provides quantitative explanations for the success of younger-first generational collectors, and explains why they perform better than older-first or hybrid algorithms on some programs but worse on others.

Several runtime systems now offer a choice of multiple garbage collectors. Eventually we can expect runtime systems to select a garbage collector based on dynamic observation of the programs they are executing.

10 References

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

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

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

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

[5] A. Bawden, R. Greenblatt, J. Holloway, T. Knight, D. A. Moon, and D. Weinreb. Lisp machine progress report. AI Memo 444, MIT AI Lab, August 1977.

[6] Steve M. Blackburn, Richard Jones, K. S. McKinley, and J. Eliot B. Moss. Beltway: Getting Around Garbage Collection Gridlock. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 17-19, 2002.

[1] The remembered set sizes in the table are the allocated size, not the peak size measured in the number of entries in the set. 2GEN has two remembered set data structures, 3GEN has three, and 3ROF has four—so the minimum remembered set size for the 3ROF collector is twice that of 2GEN, as the entry for e.g. `gcbench` shows. The imprecision in the remembered set size measurements is regrettable but does not preclude one from concluding, as the table shows, that the remembered sets for the 3ROF collector grow beyond their minimum size more often than the remembered sets for 2GEN and 3GEN.

[7] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM* 13(11), November 1970, pages 677–678.

[8] William D Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*. *ACM LISP Pointers* VIII(3), July–September 1994, pages 128–139.

[9] William D Clinger and Lars T Hansen. Generational Garbage Collection and the Radioactive Decay Model. *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), *ACM SIGPLAN Notices* 32(5), May 1997, pages 97–108.

[10] William Clinger. Data provided via the World-Wide Web at `http://www.ccs.neu.edu/home/will/GC/index.html`.

[11] Jacques Cohen. Garbage collection of linked data structures. *ACM Computing Surveys* 13(3), September 1981, pages 341–367.

[12] L. Peter Deutsch and Daniel G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM* 19(7), July 1976, pages 522–526.

[13] J Earley. An efficient context-free parsing algorithm. *Communications of the ACM* 13(2), 1970, pages 94–102.

[14] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* The MIT Press, 1985.

[15] Jacob Seligmann and Steffen Grarup. Incremental mature garbage collection using the train algorithm. In *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, Springer-Verlag, August 1995.

[16] Lars Thomas Hansen. *The impact of programming style on the performance of Scheme programs.* M.S. Thesis, University of Oregon, 1992.

[17] Lars Thomas Hansen. *Older-first Garbage Collection in Practice.* Ph.D. Thesis, Northeastern University, November 2000. Available online—see [10].

[18] David R. Hanson. Storage Management for an Implementation of SNOBOL4. *Software—Practice and Experience* 7, 1977, pages 179–192.

[19] 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.

[20] Michael W. Hicks, Luke Hornof, Jonathan T. Moore and Scott M. Nettles. A Study of Large Object Spaces. *ISMM '98—International Symposium on Memory Management*, pages 138–147. ACM Press, 1998.

[21] Anthony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A Comparative Performance Evaluation of Write Barrier Implementations. *OOPSLA '92 Conference Proceedings, ACM SIGPLAN Notices* 27(10), October 1992, pages 92–109.

[22] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In *Proceedings of International Workshop on Memory Management*, (Yves Bekkers and Jacques Cohen, editors), Lecture Notes in Computer Science 637. Springer-Verlag, September 1992.

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

[24] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.

[25] Richard Kelsey, William Clinger, and Jonathan Rees (editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices* 33(9), September 1998, pages 26–76.

[26] The Larceny home page at `http://www.larceny.org/`.

[27] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM* 26(6), 419–429, June 1983.

[28] David A. Moon. Garbage Collection in a Large Lisp System. In *ACM Conference on Lisp and Functional Programming*, 235–246, 1984.

[29] 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.

[30] 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.

[31] Darko Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms.* Ph.D. thesis, University of Massachusetts, Amherst, Massachusetts, February 1999.

[32] Darko Stefanović, Kathryn S. McKinley, and J. Eliot B. Moss. Age-Based Garbage Collection. *OOPSLA '99 Conference Proceedings, ACM SIGPLAN Notices* 34(10), October 1999, pages 370–381.

[33] Darko Stefanović, Matthew Hertz, Steve M. Blackburn, K. S. McKinley, and J. Eliot B. Moss. Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine. *Workshop on Memory System Performance*, Berlin, Germany, June 2002.

[34] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT-SIGPLAN Practical Programming Environments Conference*, Pittsburgh PA, April 1984, pages 157–167.

[35] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. *OOPSLA '89 Conference Proceedings, ACM SIGPLAN Notices* 24(10), October 1989, pages 23–35.

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