Older-First Garbage Collection in Practice

A Thesis
Presented to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

by
Lars T Hansen

November 2000

Garbage collection is a technology that solves a practical programming problem, namely that of reclaiming unused heap storage. Manual management of heap storage is notoriously error prone and garbage collection can help to simplify programming significantly. Most new programming languages incorporate garbage collection, and despite advances in computer hardware, the performance of garbage collection continues to matter.

So-called generational collection has helped to improve 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 also older data.

This dissertation reports on an implementation study that compares two older-first collectors to traditional (younger-first) generational collectors. In general, the older-first collectors perform adequately and are often effective at reducing the first-order cost of collection relative to younger-first collectors. Older-first collectors are particularly effective at managing data with queue-like or random lifetimes and heaps containing large amounts of live data.

However, the older-first collectors are brittle in the sense that their performance may degrade quickly on some kinds of large linked data structures. My implementations incorporate techniques that restore robustness by effectively disabling older-first collection in cases where degradation occurs, and this dissertation discusses further measures the collectors may take to avoid performance degradation.

I thank my committee members, Mitchell Wand, Rajmohan Rajaraman and Eliot Moss, and particularly my advisor, Will Clinger, for their participation, comments and help.

Will Clinger also contributed in many ways over many years to Larceny, my experimental system, most obviously by writing the Twobit compiler for Scheme.

The benchmarks used in this dissertation were written or modified by or in other ways contributed to by Henry Baker, Hans Boehm, Bob Boyer, Will Clinger, David Detlefs, John Ellis, Marc Feeley, Richard Gabriel, Pete Kovac, Eugene Luks, and Richard O'Keefe.

The work was supported in part by NSF Grant CCR-9629801 and by the College of Computer Science at Northeastern University.

The most profound thanks still go to my wife Lynn, who provided boundless amounts of encouragement and support, and to my parents, who have always let me do what I want to and then helped me do it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 My thesis

A new class of generational garbage collectors, older-first collectors, are practical and have performance that is competitive with the performance of traditional younger-first generational collectors.

1.2 Motivation

Safe programming languages with heap-allocated storage provide *garbage collection*: program objects that are no longer in use by the program are automatically reclaimed by the system's garbage collector and returned to the pool of free memory. Garbage collection dates to the late 1950s but has been confined to sophisticated programming environments for much of time since then; only during the last decade has it seen use in mainstream programming languages. The slow acceptance of what is an essential technology for safe programs is in part due to doubts about whether garbage-collected programs can be as space-efficient, fast, and smoothly interactive as programs that are not garbage-collected. Some of those doubts are justified. With the plummeting prices of computer hardware, however, space and time issues have become irrelevant for many application areas, and incremental collectors alleviate annoying collection pauses. As a result, popular new languages like Java, Python, and Perl require garbage collection, and programmers are becoming accustomed to its advantages.

Two trends make it interesting to work on better garbage collection algorithms. First, the proliferation of small and inexpensive mobile or embedded devices will not afford us the luxury of throwing hardware at the garbage collector until it is fast enough for the application at hand. Second, programmers who have become accustomed to garbage collection will want to write ever more sophisticated programs in garbage collected languages, no doubt pushing the limits of current collectors' abilities.

My dissertation describes two different implementations of a new type of collector, the *older-first* generational collectors. My main result is that the implementations perform well in a reasonable amount of memory, that is, this type of collector is practical. Though the designs I have implemented are generally not faster than other good collectors on the benchmark programs I have run, I demonstrate that the design space for practical garbage collectors includes older-first collectors as well.

1.3 Collection costs and heuristics

1.3.1 Primary costs

Most garbage collectors operate by tracing out the graph of objects that are reachable from a distinguished set of locations (the *roots*), marking or moving all reachable objects and leaving garbage behind as unmarked or unmoved objects. Ideally the collector should only preserve the live objects—those that the program will be using in the future. Reachability is a conservative approximation to liveness, as liveness is not computable (it may depend on future inputs, for example). In the literature in general as well as in this dissertation, *live* is used to mean "reachable from the roots", and any object not live is *dead*.

The key to low garbage collection cost is to lower the mark/cons ratio—the ratio of the amount of live storage touched by the collector to the amount of garbage reclaimed—while keeping other costs under control. The mark/cons ratio r for a simple collector is given by

$$r = \frac{l}{M - l}$$

where l is the amount of live data and M is the amount of memory available to the program. Clearly, the mark/cons ratio can be reduced by increasing the amount of memory available to the program; as the amount of memory approaches the total amount allocated by the program, the mark/cons ratio goes to zero. Conversely, as the amount of live data approaches the amount of available memory, the mark/cons ratio goes to infinity.

The mark/cons ratio can also be reduced by using clever garbage collection algorithms. Often the amount of memory is fixed or limited, so algorithmic solutions are desirable.

It turns out that most objects in many programs die at a very young age, even in programming languages without garbage collection such as C++, and the rule of thumb that this empirical property holds generally has become known as the *weak generational hypothesis*. Though there will usually be little garbage among the very youngest objects, the weak generational hypothesis suggests that if we examine the relatively younger objects in the system as a group, we will find a substantial amount of garbage among them. Therefore, a garbage collector may be able to lower the mark/cons ratio by concentrating its efforts on collecting among young objects in the system. That observation led to the construction of the earliest *generational* garbage collectors.

Generational garbage collectors organize the heap in *generations*. Each generation contains objects of a certain contiguous age range, and the generations are arranged in order by the ages of objects within them. The collector can choose to collect any subset of its generations, but ought to collect the generations that give the best mark/cons ratio. The collector cannot know which generations that will be, so it relies on heuristics for its selection, and the weak generational hypothesis suggests the *younger-first* heuristic: collect the younger generations more often than older generations.

Recent theoretical and simulation work has suggested that there are advantages to generational collectors that use an *older-first* heuristic: though the mortality rate among young objects is high, objects in older generations will have had more time to die, so there might be more garbage in older generations than in younger generations. Not collecting among young data will then give a lower mark/cons ratio overall.

1.3.2 Secondary costs

In addition to the mark/cons ratio, several secondary factors influence the cost of garbage collection. In non-generational collectors and simple generational collectors, the secondary costs are low. It is important that collectors that use more sophisticated generational algorithms keep the secondary costs under control, or improvements in the mark/cons ratio may not translate to savings in running time.

The most important secondary costs are all incurred by the handling of intergenerational pointers. Liveness is a global property, and the collector must keep objects in the collected generation alive even if they are referenced only from outside that generation; the collector must also update all pointers to objects that it moves, but it should do so without scanning the uncollected generations. To accomplish this, a *write barrier* intercepts assignments that create pointers from one generation into another and records information about the intergenerational pointers in a *remembered set*; at collection time, the collector need only trace and update the pointers recorded in the remembered set.

The write barrier adds a cost to many assignments, and the total cost may be considerable if the volume of pointer assignments is high. The cost is increased further by collection algorithms that require stores performed by the garbage collector to be examined by the write barrier as well.

The remembered set is effective only if it is fairly small compared with the uncollected region. If a program creates a data structure that results in a large number of objects being inserted into the remembered set, then the cost of scanning the remembered set at collection time will be high. Additionally, a larger remembered set may require a large amount of memory, reducing the amount of memory available to the user program.

Finally, an object that is added to the remembered set but subsequently dies will remain in the remembered set until its generation is collected. Since the object is in the remembered set, the garbage collector must treat it as live, and therefore any objects it references will also be live. Objects that are kept alive by dead objects in the remembered set are said to be "floating garbage". If floating garbage objects are subjected to garbage collection they will be marked unnecessarily, and if there are many of them, the cost of doing so will increase the cost of garbage collection considerably.

1.4 Two novel collectors

1.4.1 The renewal-older-first collector

Clinger designed an older-first collector that assumes an object lifetime distribution modeled on radioactive decay, and characterized its behavior theoretically [6]. His central result is that a generational collector can have a theoretical advantage over a non-generational collector even in a system where the collector has no information about object lifetimes. That is, generational collection may work better than non-generational collection not just because it is a better fit with current technology (as some have assumed) but because it is inherently superior to non-generational collection.

Though few programs have data whose lifetimes follow the radioactive decay model, old objects in real programs show little or no correlation between age and future life expectancy; that is also a characteristic of the radioactive decay model. It is therefore possible that a collector based on the radioactive decay model may collect old data efficiently.

Clinger's collector divides the heap into two variable-size generations, for old objects and young objects. At each collection it collects among the old objects only and then incorporates the young objects among the old. Since old objects that survive a collection are treated as if they are younger than the most recently allocated objects—that is, their youth is "renewed"—the collector type is sometimes referred to as "renewal-age", and the particular collector I have implemented goes by the name renewal-older-first (ROF).

1.4.2 The deferred-older-first collector

Stefanović proposed an older-first collector, which he called deferred-older-first (DOF), that maintains objects in age order in the heap: objects distinguished by age before collection are distinguished also after collection [27]. He simulated the algorithm on a number of programs and found that it could significantly lower the mark/cons ratio relative to a standard generational collector.

Stefanović's collector moves a collection window across the heap from the older end toward the younger end. The collector collects among the objects within the window, compacting survivors to the older end of the window and making any free memory available for allocation at the young end of the heap. When the collection window reaches the young end of the heap, it is reset to the old end, and the collection cycle starts over.

The window-movement algorithm ensures that the collector spends little effort on collecting long-lived storage and instead tends to spend most of its effort on collecting areas where there is a high ratio of garbage to live data.

1.4.3 Problems with older-first collection

It is not obvious that older-first collection is practical, and there are two assumptions in particular that are tested by older-first collection.

First, it is a folk theorem in the garbage collection community that pointers from young objects to old objects dominate over pointers in the opposite direction. That situation is beneficial to younger-first collectors, but not to older-first collectors. In an older-first collector, a predominance of young-to-old pointers will result in more write barrier work, frequent insertions into the remembered sets, higher memory use by the remembered set, and more remembered set scanning by the collector.

Second, generational collection tends to work well because the amount of floating garbage is moderate; if it is not, then the collector may not be able to reduce the mark/cons ratio. The risk of floating garbage increases as the fraction of the heap that is collected decreases; since the older-first collectors I have examined all collect smaller fractions of the heap than traditional collectors, they are at relatively higher risk for floating garbage.

1.5 Scope, results, and contributions

I have implemented variants of Clinger's renewal-older-first collector and Stefanović's deferred-older-first collector in Larceny, a research implementation of Scheme, and compared them to Larceny's standard generational and non-generational collectors.

The standard generational collector is a typical multi-generation younger-first generational collector with fixed-size younger generations and a dynamically-sized oldest generation. It is described in detail in Chapter 2. The collector is mature after several iterations of design, implementation, and tuning.

I have obtained measurements from a range of benchmarks written in Scheme. Most of my benchmarks use a large amount of memory, and many of them are known to cause difficulties for garbage collection, for example because they exhibit worst-case behavior—as when they have little garbage or queue-like or random lifetimes—or because they have a large amount of permanent data.

My measurements show that the older-first collectors are at least competitive with the standard collector, and sometimes better than it. Thus I have an existence proof that it is possible to control the secondary costs of the older-first algorithms, even though those costs are substantially more difficult to control than in a traditional generational collector.

The contributions of my dissertation are several.

First, the results of the benchmark runs show that the new collectors are practical. This makes older-first collectors attractive targets of further research.

Second, I describe numerous techniques that are instrumental in making the new collectors practical, and I evaluate their efficacy.

Third, I have run the DOF and ROF collectors on a different set of benchmarks than those used by Stefanović in his simulations, and I have been able to provide further information on the strengths and weaknesses of these collectors.

Fourth, I contribute to the understanding of a previously known problem, namely that of highly referenced objects—so-called *popular objects*—by pointing out how they occur in programs that use symbolic data and objects with identity, and how they impact programs' performance by inflating the size of the remembered set. I propose an algorithm that detects popular objects while the garbage collector is running and reduces their negative impact by omitting their referrers from the remembered set.

1.6 Why an implementation study?

Many studies that compare garbage collectors empirically use simulators, either trace-driven simulators or algorithm simulators, and there are good reasons for doing so.

It is a daunting task to implement multiple garbage collectors, and harder still to implement all collectors equally well and not bias the comparison by incorporating crucial optimizations in some collectors but not in others. Measuring a program's performance on modern hardware and operating systems is itself difficult—operating systems are not particularly deterministic, and changing the cache line alignment of a loop may drastically alter its performance on a superscalar processor. Hardware changes quickly, and today's empirical results may be tomorrow's irrelevant data.

Simulators, on the other hand, allow events to be counted without worrying about the low-level efficiency of the implementation of the algorithm, and make measurements easy.

Why then an implementation study? The chief reason is that an implementation study focuses on and measures the costs that are ignored in simulations: what happens below or between the events, so to speak. Another reason is that in a real system, issues that are often glossed over in a simulation—block management, large objects, fragmentation, implementation effort, portability—all have to be addressed, one way or another, and their impact can be (to some extent at least) observed.

1.7 Related work

The history of garbage collection up to about 1980 is reported by Cohen in his survey [8].

Generational collection and the weak generational hypothesis were first described in 1983 by Lieberman and Hewitt [22], although earlier work foreshadowed the technique [9, 3, 16]. 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.

Important early contributions to generational collection were made by Ungar and Moon, both of whom described actual implementations [29, 23].

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 [23], 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 [19, 14].

Spurred by a comment by Baker that generational collection did not have an inherent advantage over non-generational collection, Clinger and Hansen presented an analysis of what they called the non-predictive generational collectors, showing that generational collection does in fact have an advantage even in theory [1, 6]. They also presented a prototype implementation of one such collector. Their paper outlined an implementation and discussed some of the problems it must contend with, but provided limited guidance on how an implementation can overcome these problems.

Stefanović generalized older-first collection in his deferred-older-first algorithm, and he presented extensive simulation results of that and other collection algorithms in his Ph.D. thesis [27]. His simulations showed that the deferred-older-first algorithm had lower mark/cons ratios than other algorithms on a suite of Smalltalk and Java programs, and by making an argument about the cost of collection primitives he suggested that an implementation of the deferred-older-first collector would perform well in practice. Stefanović also presented evidence that his programs had no bias in pointer direction in the heap, and that most pointers point to nearby objects, both factors that facilitate older-first collection. Unfortunately, most of the benchmark programs used in the study had fairly small heaps, and the simulation may hide some costs that are important in practice.

Chapter 2

The reference collectors: design

2.1 Garbage collector infrastructure

2.1.1 Overall structure

All the garbage collectors in Larceny—even the non-generational collectors—structure the heap as an ordered set of generations. Each generation contains objects of a similar age and is always collected as a unit. Generations are labeled by unique numbers, and higher-numbered generations are said to be older than lower-numbered ones.

Generations are grouped into three areas. The *ephemeral area* contains the youngest collected generations, the *dynamic area* contains the older 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. Figure 2.1 shows all the memory areas together, along with some pieces that will be discussed below.

Not all collectors use all areas: non-generational collectors do not have a dynamic area, and a collector need not use the static area, as permanent data can just as well be loaded into the collected generations.

The youngest ephemeral generation in Larceny's generational collectors is called the *nursery*, because 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 in practice. 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.

The nursery attempts to capitalize on the weak generational hypothesis: most objects die soon after they are allocated. Although there will be little garbage among the very youngest objects in the nursery, there is probably a

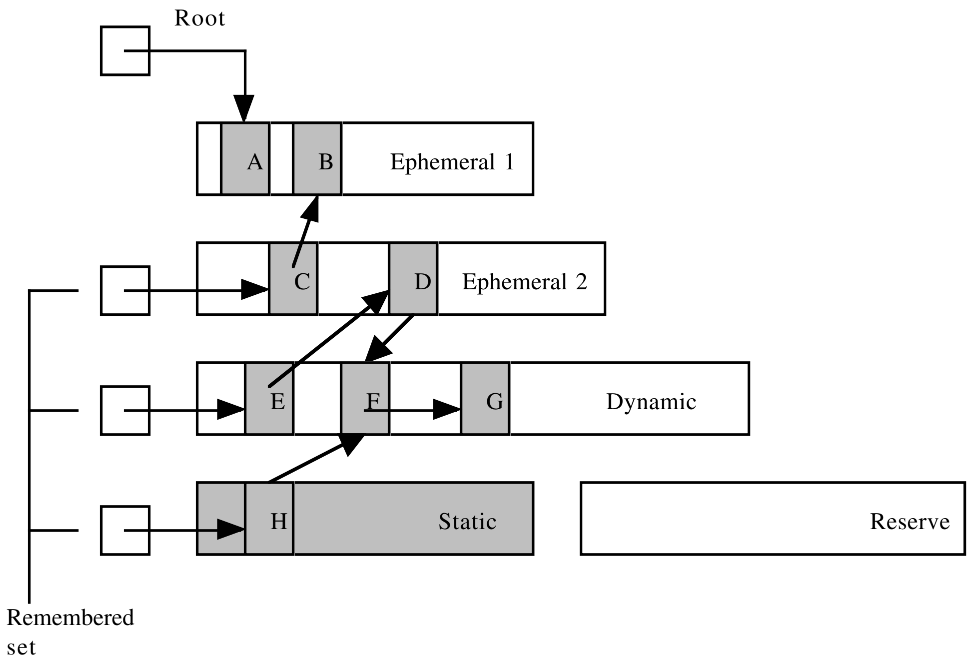

Figure 2.1: The structure of Larceny's garbage collectors.

large amount of garbage in the nursery as a whole, and evacuating objects from the nursery is an effective way to reclaim much space at little cost.

Because garbage collection is performed by evacuating live objects from the collected area and into some other area, the collector must keep a *reserve* of storage on hand for collection. In the worst case, the amount of memory required from the reserve is l, the amount of live data in the system at the time of collection. The collector cannot know l without performing a collection, so it must estimate it. Larceny's collectors usually estimate l to have the same value it had following the previous garbage collection.

After some types of collections the heap can be resized. All collectors use a user-provided policy parameter $1/L$, called the *load factor*, to compute the new heap size. The inverse load factor L expresses how much memory the user is willing to spend on the program as a multiple of the amount of live storage. If l is the amount of live data in the heap then the new heap size H' is computed as $H' = lL$. Since l memory is already live and l memory will be needed for the reserve for the next garbage collection, the amount of memory available for allocation is $H' - 2l$.

When the garbage collector runs in a heap with a fixed size, then a user-provided parameter similar to L, the *trigger* T, is used to determine how much of the heap to set aside for the reserve, independently of the amount of live data. The collector always reserves $1/T$ of the heap for collection, so the amount of memory available for allocation is $H - (H/T) - l$.

2.1.2 Large objects

The collectors that are described in this dissertation use Cheney's copying algorithm for all generations. The collector can save both time and space by managing *large objects* with a non-copying algorithm, and Larceny's collection infrastructure provides a generic large object space for that purpose [17]. In collectors that divide their generations into fixed-size blocks or use fixed-size generations, the large object space can also simplify the collector implementation because objects larger than a block or a generation are managed separately.

All of Larceny's collectors defer management of objects larger than 4 kilobytes to the large object space. The large object space is divided into multiple generations to mirror the generation structure of the rest of the system.

2.1.3 Write barrier and remembered set

Generational collectors use a write barrier and a remembered set to track those intergenerational pointers that must be traced when only part of the heap is collected.

The write barrier uses the generation labels to determine which assignments create intergenerational pointers that must be tracked by the remembered set. In the 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. A smaller generation number implies a younger generation, so if the generation number of `rhs` is less than that of `lhs`, then the store must be recorded, and `lhs` is inserted into the remembered set. For performance reasons, the insertion is done in two steps: information about the store is first inserted into a sequential store list, and the store list is later folded into the remembered set [18].

The remembered set is split into several subsets, one subset per generation except the youngest. Each subset remembers those objects in its associated generation that may reference objects in a younger generation. The remembered set for a collection consists of the union of the subsets of all generations older than the oldest generation being collected. Since the remembered set for a collection also tracks intergenerational pointers interior to the uncollected region, the remembered set may be somewhat larger than necessary.[1]

Figure 2.1 shows the remembered set on the left, tracking those objects in an older generation that reference objects in younger generations.

Larceny's remembered set uses hash tables to filter duplicates from the subsets. The hashing scheme incurs space and time costs but ensures that a remem-

[1]A remembered set structure that associates a subset with each generation except the oldest and tracks pointers from older generations into the subset's associated generation may result in less remembered set scanning work but will complicate garbage collection, as objects must be added to and removed from the subsets during a collection. Other structures are possible as well.

bered object is only scanned once during a garbage collection. The space cost of the hash table is affordable because the remembered set is small in practice; the time cost is assumed to be regained by not having to scan some objects repeatedly, especially large arrays that may have been inserted into the remembered set many times.

Since the remembered set contains all objects that may contain intergenerational pointers, its size is bounded only by the number of objects in the heap. The remembered set representation used in Larceny also requires storage that is proportional to the number of elements of the set.

There are viable alternative structures for the remembered set. A number of systems use remembered sets that remember the locations within objects that contain intergenerational pointers rather than the objects themselves. The chief advantage of remembering locations is that large objects that contain a few intergenerational pointers do not need to be scanned in their entirety by the garbage collector; only the locations that contain the intergenerational pointers do. However, if a large object contains many intergenerational pointers, many of its locations will be recorded in the remembered set, inflating the set's size relative to a system that records objects.

A popular alternative remembered set scheme is *card marking.* A map with one entry for each *card*—a memory region smaller than a page—records whether that card may contain an intergenerational pointer. The write barrier marks a card by storing a value in the map; later, the garbage collector scans the card map and then scans those cards that are marked. Card marking is attractive because duplicates are filtered automatically (marking an already marked card has no effect), because only the marked sections of large objects need be scanned by the collector, and because it reduces storage consumption (see below). The chief disadvantage of card marking is that the entire card map must be scanned on each garbage collection. If the map is large, collections that collect only the nursery may be unreasonably expensive. Large cards reduce the size of the card map, but then more work must be expended on scanning a marked card.

The storage required for the card map is proportional to the size of the address space used by the program. In principle the worst-case storage required is therefore roughly proportional to the number of objects in the system, which is no better than the space complexity of the remembered sets I currently use in Larceny. In practice, however, the constant of proportionality for card maps is significantly lower than for Larceny's object sets, being as low as a bit per card where Larceny's object sets use two to three words per object.

It is also possible to dispense with the hash table Larceny uses to filter duplicates by placing a mark on each object and setting the mark when the object is entered into the remembered set. The write barrier can recognize at low cost that a mark is set and avoid entering duplicates into the remembered set. The technique only works for objects that have space for a mark bit, so header-less objects must still be filtered by a hash table. Luckily, most garbage collected languages have implementations that do not use header-less objects, and the only header-less objects in a typical Scheme implementation are pairs, which are rarely updated.

Both of the older-first collectors I report on require objects to be recorded in several subsets at the same time. That requirement means that neither card maps nor object header marks are attractive for my older-first collectors: each card or object will need one mark bit for each subset of the remembered set that it can be recorded in, and the resulting space overhead may be prohibitive. The tricks one can use to work around that problem are too arcane to be discussed here.

2.1.4 Block management

Figure 2.1 may give the impression that the generations in Larceny are contiguous memory areas, but that is not the case. Memory is allocated and managed in large blocks (256 kilobytes in most collectors), and the generations and the reserve are composed of sets of these blocks. A global descriptor table maps an arbitrary address to a descriptor byte, which contains the generation number for the address (if it is allocated to a generation) and some additional information. Blocks can be moved from one generation to another simply by changing the values in the descriptor table.

The blocks are managed by an allocator that attempts to minimize the overall memory footprint of the garbage collector, if necessary by returning memory to the operating system or by moving free blocks to new addresses to create large blocks for large-object allocation.

2.2 Non-generational collector operation

The non-generational collector is a copying collector that uses Cheney's breadth-first copying algorithm to collect a single variable-size generation [4]. The collector scans the entire static area as part of its root set on each collection. Figure 2.2 shows the non-generational collector in operation.

Following each collection the amount of live data, l, is known precisely, and a new heap size can be computed easily.

2.3 Generational collector operation

The generational collector uses a fixed number of fixed-size ephemeral generations and a single variable-size dynamic generation. The number of ephemeral generations and their sizes are selectable at startup. Higher-numbered generations contain older objects than lower-numbered ones.

There are two types of garbage collections in the generational collector. In either case, when a particular generation is collected, all younger generations are collected as well—that way, the remembered set need track only old-to-young pointers, as all relevant young-to-old pointers will be seen by the collector. A *promotion into* a generation g promotes all live data from the younger generations $1 \ldots g-1$ into generation g, but does not reclaim garbage in generation g.

A *collection in* a generation g also promotes all live data from younger generations $1 \ldots g-1$ into generation g but additionally reclaims garbage in generation g. Figure 2.3 shows the generational collector in operation.

The fixed-size ephemeral areas require a conservative promotion policy, where the volume of survivor data must be estimated to be as large as the volume of allocated data for purposes of computing whether the survivors will fit—after all, all allocated data may be live. Thus, a generation can be the oldest collected generation only if the amount of storage allocated in that generation plus the amount allocated in younger generations does not exceed the size of the generation.

Once the collector has determined the oldest generation to collect, it must choose whether to collect in the generation or just promote younger data into it; the collector promotes into the generation only if it estimates that this will leave enough room for a promotion into the generation also the next time around. The purpose of that policy is to reduce the chance that an even older generation must be collected the next time around.

As a consequence of the conservative policy, the simple promotion algorithm used in Larceny's generational collector may sometimes promote very young data into fairly old generations. More adaptive policies are possible even in fixed-size areas, but have higher implementation complexity and as yet uncertain benefits.

Following a collection in the dynamic area, all live data are in that area and the static area, because all live data in younger generations were promoted into the dynamic area during the collection. Therefore the amount of live data is known exactly, and the collector can resize the heap based on precise knowledge.

The generational collector ignores the volume of large objects in making policy decisions for the ephemeral generations, and it simplifies the collector to do so. Larceny allows objects as large as 16 megabytes to be allocated, but it is a system invariant that all objects are allocated in the nursery, and the nursery is not usually that large. Thus, large object allocation may lead to a situation where the nursery is over-full. Clearly there are several ways to resolve this dilemma; ignoring the volume of large objects in the ephemeral areas is merely the easiest, and has the pleasant side effect that large objects are promoted at the same pace as smaller objects.

2.4 The compiler and the garbage collector

It is hard to interpret results from garbage collection research without having an understanding of the system being measured, as details of the compiler and data representations can have a significant impact on the performance of the garbage collector. In this section I describe some attributes of Larceny that I think will be helpful in understanding the results presented in later chapters.

A system that has a poor compiler or no compiler at all makes the garbage collector seem less important than it really is because the system spends much of its time executing user code. Similarly, a system that heap-allocates many

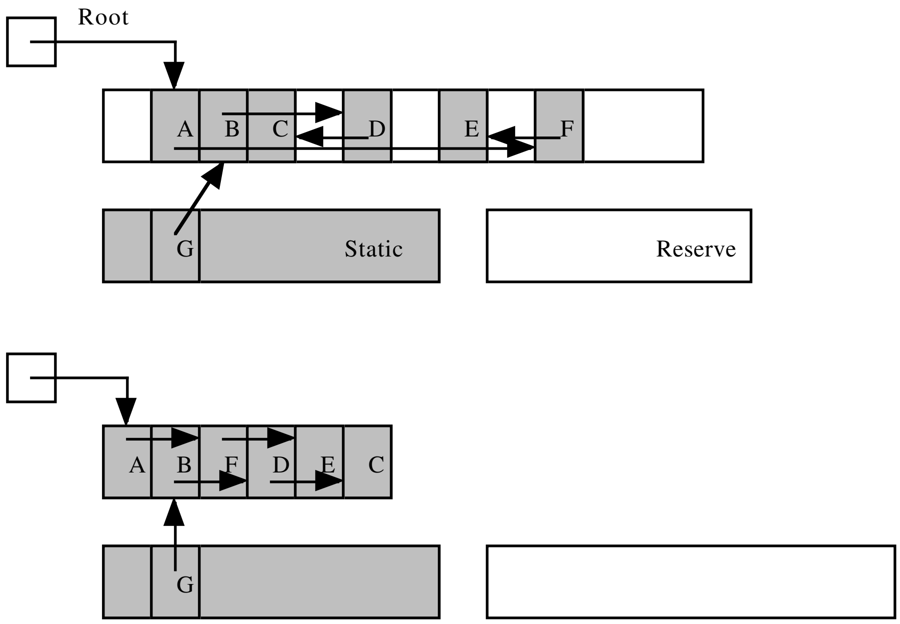

Figure 2.2: The non-generational collector in operation. Live objects in the collected area (A–F) are copied into memory taken off the reserve. Objects in the static area serve as roots but are not copied.

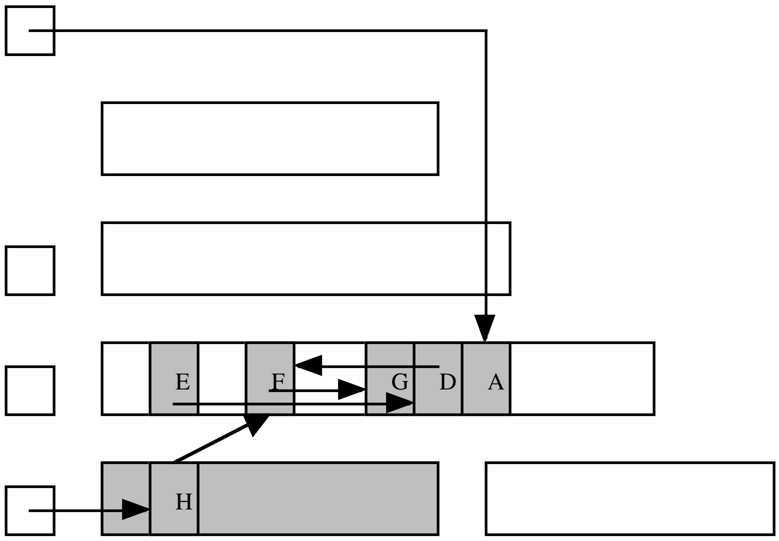

Figure 2.3: The generational collector in operation. This is the collector shown in Figure 2.1 following promotion into the dynamic area.

data structures that could have been allocated on a stack or not allocated at all makes the garbage collector seem more important than it really is because the system spends much of its time allocating and reclaiming storage.

Many implementations of Scheme are inadequate for garbage collection research on both counts, being straightforward interpreters that heap-allocate environments, continuation frames, and all values, even small integers that would fit in a register. (That these systems are as popular as they are only attests to the relative unimportance of performance for many day-to-day and exploratory computing tasks.)

Larceny uses the Twobit optimizing compiler for Scheme to compile Scheme to machine code, and Twobit is instrumental in producing good code for the user program as well as reducing the amount of heap allocation that is performed by the program.

As both Twobit and Larceny are research systems, there are rough spots that cause occasional problems, but on the whole the combination performs well. Previous measurements have established that Twobit and Larceny together have performance that is competitive with both Standard ML of New Jersey and commercial Common Lisp and Scheme systems [5]. I have also received anecdotal evidence that Larceny is competitive with a recent Java system from Sun Microsystems on at least one GC benchmark [7].

Twobit incorporates a number of optimizations that limit heap allocation largely to that allocation performed explicitly by the user program. In particular, neither continuation frames nor environment structures are allocated on the heap unless the program requests it by capturing the continuation or creating closures. Program variables that are updated by assignment are heap-allocated, but Scheme programs are largely functional in nature and heap-allocated variables are in practice few.

Larceny manages a run-time stack as part of the allocation area. The stack grows downward from the high end of the allocation area; the stack pointer serves as the allocation limit, and the allocation pointer as the stack limit. The stack cache is flushed into the heap as a chain of individual frames on each garbage collection, slightly increasing the amount of allocation and creating some opportunities for floating garbage in older-first collectors (see also Section 7.4.2).

Larceny represents all objects uniformly as one-word values. Objects that do not have identity and fit in a single word—integers up to 30 bits, boolean values, characters, and so on—are never allocated on the heap. All other objects are heap-allocated and represented by a pointer to the heap-allocated storage.

Every one-word object representation has a three-bit tag in the low bits that distinguishes classes of representations; further representation information is stored in the pointed-to object's header, if it has one. The least significant bit in every pointer is 1, and recognizing a pointer is a simple mask and branch operation.

Most heap-allocated objects have a header that contains the length of the object and information about its particular subtype—for example, the class of "vector-like" structures contains not only vectors but also symbols, records, I/O ports, and other data types that contain fields of tagged data. The header also stores an object's length, and can be used to store information (like mark bits) for the garbage collector. Only pairs are header-less, consisting of two consecutive words.

Larceny's garbage collectors never inspect the subtype field of object headers, as each class of objects is handled by the same code in the collectors—for example, all "vector-like" structures are handled together without any need to dispatch on the subtype. As there are only four classes of object layouts, dispatch in the collector is simple.

The most serious problem with Larceny and Twobit at present is that all floating point numbers, even intermediate results, are heap-allocated. IEEE double precision floating point numbers, which is the only kind currently available in Larceny, require two words and must therefore be allocated on the heap. The heap-allocated structure also needs a header word and must be aligned on a two-word boundary, so the total storage required is four words.

A more sophisticated compiler might be able to keep intermediate floating-point values in registers, and judicious use of homogeneous floating-point arrays in the user programs would remove the need for much of the remaining allocation. However, Twobit does not yet perform these optimizations,

Since all floating point numbers are heap allocated, floating point intensive programs allocate a great deal of short-lived storage, invoke the garbage collector frequently, and have poor performance generally. For example, an eight-by-eight matrix multiplication allocates 1024 floating point objects from the heap, and most of those objects will be dead after fewer than 50 instructions have been executed.

Chapter 3

The DOF collector: design

The deferred-older-first (DOF) collector is a design first proposed by Stefanović in his Ph.D. dissertation [27]. He simulated the operation of the DOF collector on a number of benchmark programs written in Java and Smalltalk, and found that it sometimes had markedly lower mark/cons ratios than many other collector types, including traditional generational collectors. He also reported event counts for the collector and argued that the secondary costs could be kept sufficiently under control for an implementation to capitalize on the lower mark/cons ratio.

In this Chapter I present the DOF collection algorithm and the design of my own DOF collector, which is modeled on Stefanović's design but is not equivalent to it. I discuss my design in some detail, pointing out the differences with the previous design, and justify my decisions.

3.1 Collector operation

The assumption behind the DOF collector is that the amount of garbage in an area increases with the time since the data in the area were allocated or collected, and therefore that garbage collection will have the greatest pay-off in the areas that have been collected least recently [28].

My DOF collector manages only the dynamic area, which it splits into relatively small blocks that it arranges by the age of the objects in them, from youngest to oldest. Objects are allocated at the young end of the heap, in the oldest block that is not yet full. The collector also keeps some blocks in a reserve, unavailable for allocation. (The reserve is used for copied objects during garbage collection.) When allocation has filled the available blocks, a *DOF collection* takes place. The collector places a *collection window* on the least recently collected blocks of the heap and evacuates all live objects from the window to blocks taken from the reserve. It then inserts the blocks containing the copied objects into the heap in the window position, and uses the now empty blocks in the collection window to extend the allocation area or to replenish the reserve.

Figure 3.1 illustrates the DOF collector in operation.

Eventually the collection window may cover the block that was last used for allocation. After that collection, the blocks least recently collected are at the old end of the heap, so the collector *resets* the window to the old end of the heap.

If the heap size is not fixed, then the collector will decide whether to grow the heap following a window reset.

The mutator and collector use write barriers to track stores that create interblock and intergenerational pointers. An object is added to the remembered set if it contains a pointer into a block that will be collected before the block containing the object is collected.

Occasionally the collector may decide—using some fixed policy or relying on heuristics—that it is necessary to perform a *full collection* of the heap to reclaim large circular structures or to remove dead objects in the remembered set. To do so it invokes the whole-heap garbage collector, which is a separate collector that uses a different collection algorithm but does not disturb the age ordering of the blocks or objects in the heap.

The DOF collector maintains a separate large object space to let the program use objects larger than a block and to avoid copying large objects. Variable-size blocks and collection windows look like they might be fairly difficult to implement, so in my estimation the large object space has helped to simplify the collector implementation.[1]

The generations of the collector are the positions in the dynamic area where the collector can place the collection window. A fixed-size collection window lets the collector determine the generation boundaries ahead of time, and the generations of the DOF collector truly are analogous to generations in a traditional generational collector.

The DOF collector maintains its generations in such an order that pointers from higher-numbered to lower-numbered generations are exactly those pointers that need to be tracked to collect generations in older-first order. Thus the collector can reuse the existing write barrier machinery and will run the same compiled code as the other generational collectors.

The generations are ordered so that the next generation to be collected is always the lowest-numbered generation in the dynamic area (logically the "youngest", though it does not contain the youngest data). Following a collection, the generations are reordered to maintain that invariant. Figure 3.2 shows the operation of the collector from Figure 3.1 with its five-generation structure revealed.

A word on nomenclature: I present four variations of the DOF collector in this dissertation. These collectors are called 1DOF, 2DOF, 3DOF, and 4DOF; the number in the name reveals the fraction of the dynamic area (not counting the reserve) that is collected by the DOF collector in a single collection. For

[1] Chiefly, fixed-size blocks simplify the algorithms for moving memory among the generations of the DOF collector. If blocks could vary in size, then generations might also have to vary in size, and that would make it hard to guarantee that a collection will not overflow the available memory without using overly conservative budgeting of memory.

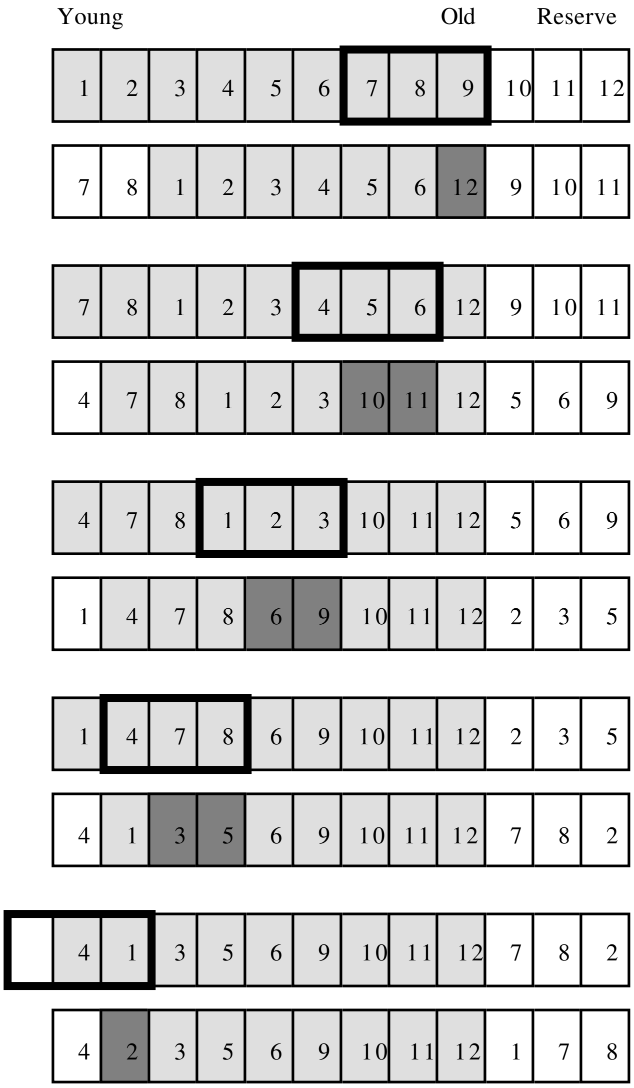

Figure 3.1: The DOF collector in operation. The figure shows pairs of collector configurations, right before and right after collection. Time travels down. The collection window is three blocks wide and is shown with a thick border. Shaded blocks are full, and unshaded blocks are empty; blocks with heavy shading hold survivors from the previous collection. A window reset occurs following the last collection in the figure. The labels on the blocks show how the blocks are moved by the collector. The unlabeled block appearing at the left of the heap in the last collection is an artifact of the implementation; see Figure 3.2.

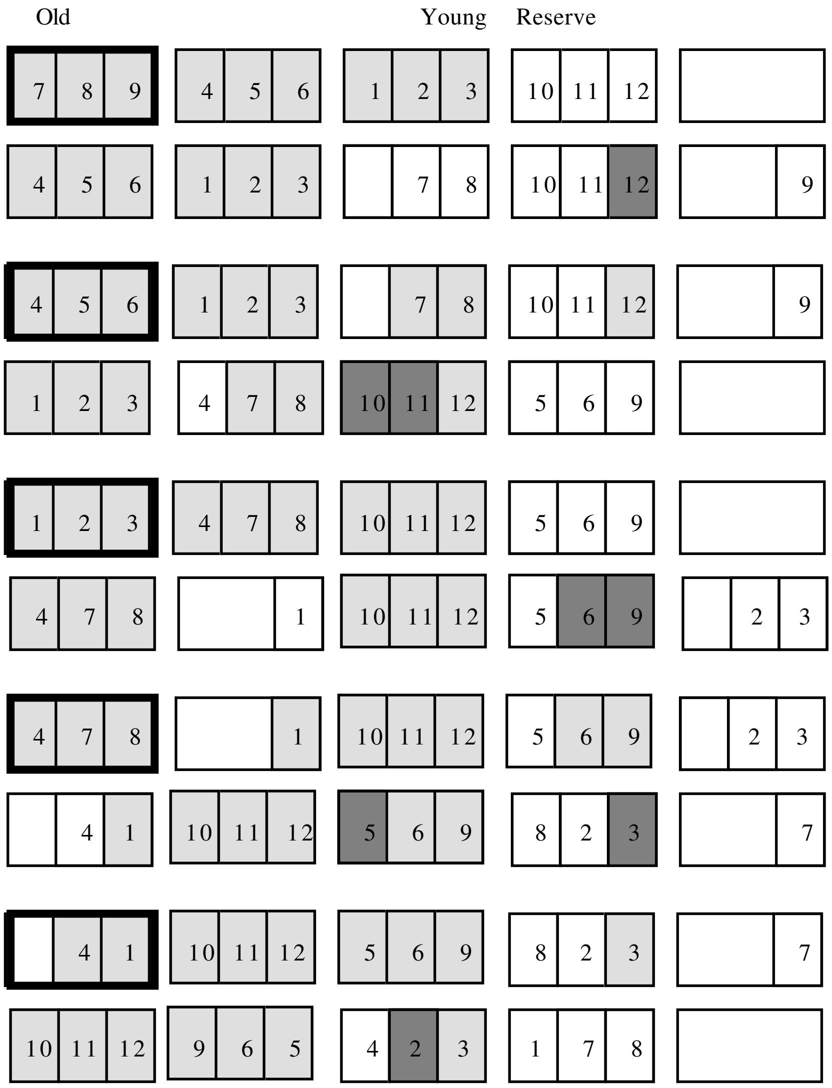

Figure 3.2: The 3DOF collector as implemented in Larceny, showing the movement of generations and blocks as they are rearranged to keep the collection window at the left end of the heap. Only labeled blocks have memory allocated to them. The blocks are labeled in correspondence with the blocks in Figure 3.1: the two figures represent two views of the same collector collecting the same program.

example, the collector shown in Figure 3.2 is 3DOF, because the the amount of memory collected at a time (3 blocks) is 1/3 of the non-reserve area of the heap (9 blocks). An nDOF collector always has $n+2$ generations, $S_D(n+1)$ memory allocated to it (where S_D is the size of one DOF generation), and a reserve size of S_D.

It is not practical for an implementation study like mine to examine all tradeoffs, as the design space is quite large. The next sections outline the important attributes of the the design space, as I see them, and explains how I have made my choices, and why.

3.2 The collector's scope

My DOF collector manages only the dynamic area, relying on one or more ephemeral generations for object allocation and to manage very young data.

A nursery appears to be contrary to the intuition behind the DOF collector, but as explained in Section 2.1, a nursery may improve the efficiency of collecting among very young objects. Scheme programs and implementations may have characteristics that favor a nursery. That point is discussed in some detail in this section. A more prosaic reason for using a nursery is that much existing code in Larceny assumes that any value can be stored into any newly allocated object without a write barrier check, and it would be labor-intensive to modify that.

The direct benefit of a nursery is that no write barrier is required on initializing stores, because the nursery is collected every time the garbage collector runs. If the write barrier is expensive, or if there would be a lot of write barrier activity due to initializing stores, then the nursery may reduce allocation costs. The expense of the nursery comes from promoting objects out of the nursery into the dynamic area: each surviving object must be copied. If the volume of survivors is large, then the copy cost may be considerable.

In contrast, allocating directly in the dynamic area dispenses with the cost of the copying but requires a write barrier check on every initializing pointer store.

We can break the primary costs into five variables: P is the number of initializing pointer stores in the system with the nursery, as a fraction of the allocation volume; S is the number of survivors out of the nursery, also as a fraction of the allocation volume; W_p is the average cost of the write barrier for an initializing pointer store; W_n is the average cost of the write barrier for an initializing non-pointer store; and C is the cost of copying and scanning the entire allocated volume. For the purposes of this discussion an initializing store is any store into an object before it is subjected to its first garbage collection.

The cost of promotion into the DOF area, D_{nurs}, and the overhead of allocation in the DOF collector without the nursery, D_{nonurs}, are given by

$$\begin{aligned} D_{\text{nurs}} &= S(PW_p + (1-P)W_n + C) \\ D_{\text{nonurs}} &= PW_p + (1-P)W_n. \end{aligned}$$

The equations assume that the mutator and GC write barriers have the same cost, which may not be true.

The value or cost of the nursery depends on the characteristics of the program's data structures. These are to some extent an artifact of the programming language used.

In general $W_p > W_n$, though the exact relationship depends on the details of the write barrier, which depends on the programming language.[2] A typical Scheme implementation will also have $W_n > 0$ because the language is dynamically typed and the write barrier must check every assignment. Some sophisticated Scheme implementations are able to use static analyses to determine the types of some program data, and can remove some write barrier checks.[3]

Furthermore, P is probably much closer to 1 in Scheme programs than in other languages because Scheme programs often perform symbolic computations, using objects with identity (represented as pointers) and lists extensively, and also because typical Scheme implementations represent even floating-point numbers as pointers to heap-allocated storage. A study of some Scheme programs gave a rough estimate for P of 0.7.

It is probably the case that D_{nurs} is dominated by $S(PW_p + C)$ and D_{nonurs} is dominated by PW_p, in typical Scheme implementations running typical Scheme programs.

Statically typed object-oriented languages and programs may have different characteristics. The mutator's write barrier in a statically typed language needs to check only a subset of the stores; the garbage collector, given an object's descriptor, can probably reduce the cost of the collector's write barrier too. Furthermore, object-oriented languages may have lower values of P than Scheme programs.[4]

The nursery pays off if

$$D_{\text{nurs}} < D_{\text{nonurs}}.$$

Given a standard generational collector and a DOF collector with a nursery it is possible to measure D_{nurs}, S, and SC, and those values are sufficient to compute C. Then D_{nurs} can be parameterized by S, and D_{nonurs} can be expressed in

[2]The possible write barrier mechanisms depend on the programming language because some languages allow the write barrier to be incorporated into existing mechanisms. For example, we have devised (though not yet implemented) a write barrier that has zero cost in those cases where it can piggyback on the type checking performed by dynamically typed languages like Scheme. Naturally, that mechanism is not necessarily useful in a statically typed language.

[3]Twobit performs the analyses but does not yet remove the checks. I have no data on the number of checks that might be removed in typical programs.

[4]Stefanović reports that the ratio of all mutator pointer stores to words allocated was on the average about 0.1 across his benchmark suite (see [27], Table 6.1). Few of the values stored (about 20% on the average) were null pointers, so most of the stores had to be processed by the write barrier. Stefanović's ratio is not the same number as my P, because it includes pointer stores even after the updated object had been subjected to garbage collection and does not include initializing pointer stores performed by the storage allocator, but it suggests a low P for that suite of imperative programs.

terms of the parameterized D_{nurs}:

$$D_{\mathrm{nonurs}}(S) = \frac{D_{\mathrm{nurs}}(S) - SC}{S}.$$

By plugging in values for S we get Table 3.1, which shows the measured values along with the highest values of S for which the cost with the nursery is estimated to be lower than the cost without the nursery. If the S for a program exceeds its cutoff value, then the nursery is probably not improving performance.

Some early estimates based on this method suggested that the nursery was clearly beneficial, but the DOF collector was since tuned substantially and Table 3.1 suggests that Larceny's nursery is not beneficial on any of the programs at the current nursery size, as the S cutoffs values are lower than the measured values of S on all programs. Thus I may have chosen poorly when I chose to use a nursery, and a study similar to mine but not using a nursery is an obvious candidate for future work. Indeed, the calculations of the S cutoff values are also subject to several sources of error and uncertainty, including different write barrier costs in the two collectors, noise in the measurements, and nonlinear changes in S with changes in nursery size.[5]

There are some indirect effects of using the nursery that are hard to quantify.

A nursery that fits within the processor's data cache should have good locality of reference, while a nursery-less DOF collector has a cyclic memory use pattern that will probably not have good locality unless the fraction of the heap collected also fits within the data cache. On the other hand, the nursery reduces the frequency of DOF collections, which may actually reduce locality of reference because live and dead objects will remain intermingled in the dynamic area longer. I have not studied the locality issue further.

The nursery may have a negative impact on DOF collection in that it disorders the objects that are promoted to the dynamic area. As long as the nursery is smaller than each generation in the dynamic area, this effect should be no worse than that caused by copying DOF collection (see the next section). In all my tests the nursery is 1 megabyte in size and each DOF generation is always larger than that, usually between 3 megabytes and 7 megabytes for 4DOF and larger for the other DOF collectors.

[5]The garbage collector's write barrier is probably substantially less expensive than the mutator's barrier in the current system—I would guess 30%–50%. (The savings are a result of the collector's barrier being able to keep more values in registers across invocations.) Since the nursery-less system will use the mutator's barrier for initializing stores, D_{nonurs} is larger than I have estimated it to be, and the nursery has more of an advantage than the computed values of S suggest.

The calculation of C is subject to noise because SC includes not only the cost to copy data but also the cost to scan the remembered set. The amount of remembered set scanning probably varies with S, so C varies with S also, whereas I have taken C to be constant for each benchmark.

Finally, it is well known that survival rates can change abruptly; in particular, delaying collection by providing a larger nursery increases the chance that data will not be copied at all, as later chapters will show.

	Measured values			
	S	D_{nurs} (seconds)	SC (seconds)	S cutoff
`5earley:12`	0.21	2.22	1.83	0.18
`gcbench:5`	0.27	13.32	11.04	0.18
`gcold:1,0`	0.44	4.68	4.05	0.004
`gcold:1,1000`	0.38	4.76	4.17	0.12
`gcold:10,1000`	0.06	0.51	0.67	-0.40
`nboyer:3`	0.45	1.36	1.10	0.20
`5nboyer:3`	0.44	7.21	5.13	0.29
`perm:200,1`	1.00	6.38	4.55	0.29
`perm:25,8`	1.00	6.34	4.44	0.29
`5sboyer:4`	0.26	2.37	1.96	0.18
`twobitlong`	0.08	4.01	3.94	0.02
`twobitshort`	0.18	1.10	0.97	0.12
`5twobitshort`	0.17	3.65	3.57	0.13

Table 3.1: Table of S cutoffs beyond which the nursery is probably not profitable, for a subset of the benchmark suite. The values for S and D_{nurs} are taken from a set of runs of 4DOF, and the values for SC are taken from a set of runs of 2GEN. Both sets were with variable heap size and $L = 3.0$. (The benchmarks are presented in Chapter 5.)
The negative result for `gcold:10,1000` indicates that 2GEN cannot win on that program unless the promotion rate can be brought below zero.

3.3 Collection method

The DOF collector must use some sort of compaction algorithm if it is to maintain strict age ordering of the objects in the heap. The alternative, Cheney's breadth-first copying algorithm as it is used in most copying collectors, will rearrange the objects [4].

My DOF collector nevertheless uses a copying algorithm to evacuate live objects from the collection window. Compaction normally requires several passes over the heap; it will be expensive compared with breadth-first copying and will reduce the benefit of DOF collection [20]. The rearrangement of the heap will be slight, as objects that are together in a collection window will be scattered over at most two generations after the collection. I have not attempted to quantify the impact of choosing copying over compaction.

3.4 Collection window size and positioning

A DOF collector manages memory in small blocks, and the collection window always includes an integral number of blocks.

My DOF collector uses a collection window that is a user-selectable fixed fraction of the heap size, as explained earlier, and it covers exactly one generation. The most important advantage of a fixed-size window is that the collector can maintain the remembered sets on a per-window basis: as in any generational collector, no objects that reference only objects within the same generation need be remembered. Filtering out generation-internal pointers reduces the remembered set size and processing time.

A further reason for choosing fixed-size windows in Larceny is that variable-size windows are only useful in conjunction with heuristics for selecting the window size, and it has not been my intention to examine such heuristics.

3.5 Collecting circular garbage

A garbage structure that is situated in the heap such that it is never wholly within the collection window will never be collected by the DOF collector, and can be collected only if the entire heap is collected occasionally.

My DOF collector uses a backup whole-heap collector to collect large circular garbage structures. The whole-heap collector is invoked at a fixed frequency and does not disturb the order of objects in the heap. I chose a fixed invocation frequency because that's a simple policy to implement; other invocation strategies (for example opportunistic strategies that exploit pauses initiated by the user [30]) could be used.

The user can specify the frequency of full collection in terms of DOF collections or window resets. The best frequency clearly depends on both the user program and on the fraction of the heap that is collected on each DOF collection.

A good choice for the whole-heap collector is a mark/sweep collector that marks the heap but sweeps only the remembered set: it removes all references to dead objects from the remembered set. Since those objects are the ones that cause circular garbage to be retained, later invocation of the DOF collector will take care of reclaiming the dead objects and compacting the heap. (A collector that sweeps only the remembered sets probably has no performance advantages over a standard mark/sweep collector, but it is simpler, as it does not have to worry about reclaiming and managing memory, or compacting the heap.)

3.6 Making room for large objects

It turns out that it is hard to accommodate truly large objects in a heap managed by the DOF collector. One benchmark program, `gcbench`, allocates an object that is 2 megabytes in size at the start of each of its five iterations, and the collector must find the space to allocate these objects and to reclaim them when they have died.

I have adopted several strategies to allow the collector both to allocate objects larger than a block and to reclaim them in a timely manner.

The DOF collector allocates all its blocks on demand so that it does not tie up unused memory in generations that are not active, like the reserve or the generations younger than the one in which allocation is taking place. This makes memory available for large object allocation.

Furthermore, Larceny's block manager reduces fragmentation on the block level by remapping parts of the address space to create large adjacent free blocks when necessary.

However, by allowing large object allocation to claim memory that may also be needed for garbage collection, there may not be enough memory available to perform a collection later on. If a low-memory condition is detected before a collection, the collector will run a full collection (using the whole-heap collector described in the previous section) to reclaim any dead large objects.

In practice, these techniques are sufficient to allow programs like `gcbench` to run to completion. However, full collections are expensive, driving up the total collection cost, and the solution is somewhat unsatisfactory.[6]

Several fudge factors are available to reduce the negative impact of the full collections. Two factors f_P ("free before promotion") and f_C ("free before collection") adjust the threshold for full collection. A full collection is only run before a promotion into the dynamic area if less than $S_E f_P$ memory is free, where S_E is the size of the ephemeral area. Likewise, a full collection is only run before a DOF collection if less than $S_D f_C$ memory is free, where S_D is the size of one DOF generation. Another factor f_A ("free after collection") determines whether a DOF collection has reclaimed enough memory. The amount of free memory following collection must be at least $S_E f_A$; if not, another DOF generation is collected to reclaim more. The effect of $f_A > 1$ is to reduce the chance that large object allocation before the next collection will reduce the memory level to the point where a full collection will be required.

I make no claims as to the generality of these fudge factors, though they are clearly not aimed specifically at helping `gcbench` run.

3.7 Area size

A DOF collector that does not use a fixed heap size must be able to grow, and possibly shrink, the heap dynamically in response to the program's memory demands.

My DOF collector allows the heap to grow, but not to shrink. Heap shrinking generally requires an expensive or complicated recomputation of the remembered set, and that added complication did not seem important for this study.[7]

[6] The other collectors have similar problems, but they also have large amounts of space that is kept in reserve for full garbage collections—the reserves for the reference collectors must be able to hold all live data, while the reserve for the DOF collector must be able only to hold the live data from a single generation. In the reference collectors, large objects can be allocated out of the reserve, and collection will often reclaim dead large objects before the system runs out of space.

[7] It might have been possible to implement a simple form of heap shrinking by dynamically reducing the number of DOF generations, but that has several drawbacks: it's a coarse-grained form of heap adjustment; it requires a strategy for adding DOF generations as well; and it introduces noise.

Larceny computes a new heap size H' based on the amount of live data in the heap l and a fixed load factor $1/L$: $H' = lL$. Since the DOF collector never collects the entire heap, it does not have an accurate value for l. The whole-heap collector can provide an accurate measurement of live data, but if it has not run recently, or at all, then the amount of live data can be estimated based on the running mark/cons ratio r, which the DOF collector can measure.

The mark/cons ratio provides an indication of how much memory survives an entire collection cycle. Observe that for whole-heap garbage collection, H, l, and r are related by $H = l(1 + E + 1/r)$ where E is a parameter that depends on the collector; it is the amount of extra memory needed during collection, as a fraction of l. For a stop-and-copy collector $E = 1$ (all live memory is copied), and for a mark-and-sweep collector $E = 0$ (live memory is marked in-place).

Assume the equation for H holds for the DOF collector also. An nDOF area is split into $n-1$ equal-size pieces, as one of the pieces has no memory allocated to it; therefore there are $n-2$ pieces that hold live data and one piece is for the reserve. The reserve therefore need be no larger than $H/(n-2)$, and we have $E = 1/(n-2)$. Thus we find that for the DOF collector, an estimate of live size $\hat{l}$ is

$$\begin{aligned} \hat{l} &= \frac{H}{1 + \frac{1}{n-2} + \frac{1}{r}} \\ &= \frac{H}{\frac{n-1}{n-2} + \frac{1}{r}}. \end{aligned}$$

Since the collector can compute $\hat{l}$, it can also compute the desired new heap size based on the load factor: $H' = \hat{l}L$.

The collector allows the user to specify a fudge factor to moderate the predictions of the heap size calculation. The heap size increase is divided by this factor, the *growth divisor*, to give the actual heap size increase. The default value is 1, but informal studies suggest a value of 2 works better in practice, either because $\hat{l}$ overestimates l or because the collector's inability to shrink the heap when appropriate makes it hard for it to temper growth.

3.8 Remembered set structure

The remembered set must track pointers from any generation i to any generation j where j can be collected before i.

My remembered set structure is somewhat complicated because it tries to reduce both the space used and scanning time. For each generation except the nursery there is an *assignment set* that holds the objects in the generation that contain pointers to younger generations; these pointers were all created by assignments. The assignment sets are scanned on every garbage collection. Larceny's other collectors use assignment sets in the same way.

The cost of collections in the ephemeral area is inflated by remembered set entries in the assignment sets that track intergenerational pointers that are interior to the dynamic area. Because the entries are in the remembered set they must be scanned, but as they reference no objects in the ephemeral area, their presence has no effect on ephemeral collections except to slow those collections down.

Since collections in the ephemeral area are frequent, the DOF collector maintains a *shadow set* for each assignment set to track pointers that are irrelevant to ephemeral collections. The shadow sets are initially empty, and are updated only by the DOF collector: Following each promotion into the dynamic area the contents of the assignment sets are merged into the shadow sets, as any objects still in the assignment sets must be there to track pointers interior to the dynamic area. The assignment sets can then be cleared. During later DOF collections, the shadow sets must be scanned, but ephemeral collections can ignore their contents. (Note that the vast majority of intergenerational pointers in the dynamic area are never in the assignment sets, and are handled differently. See below.)

The assignment sets and shadow sets use hash tables to filter duplicates, and a remembered set entry occupies about three words on the average.

Intergenerational pointers created by the garbage collector are tracked by *DOF sets*, which are designed for compact representation and low scanning cost. There is a matrix of DOF sets, one for each pair of DOF generations (i, j). The set $R_{i,j}$ contains all objects in generation i that contain pointers into generation j; naturally, $R_{i,i}$ is empty, as are all of the sets $R_{i,j}$ where generation i is collected before generation j. When the collector collects any generation j, the remembered set for the uncollected region is therefore just the union of all the $R_{i,j}$.

The DOF sets rely on the write barrier to filter duplicates, so the average space cost per entry is about one word. The DOF sets tend to grow much larger than the other sets, and the space optimization is important.

3.9 Write barrier

The mutator uses Larceny's standard write barrier; the ordering and reordering of generations ensures that that barrier traps exactly the necessary assignments.

The garbage collector's write barrier performs its own filtering. An object is a candidate for insertion into a DOF set only at the time it is scanned following a garbage collection. The collector's write barrier examines every pointer in the object and inserts the object into the appropriate DOF sets on the fly, marking the generations whose DOF sets already contain the object in a bit-vector, and testing the bit-vector before every insertion to filter out duplicate insertions.

The DOF collector's write barrier slows down collection relative to both the standard collector, which does not use a write barrier during collection, and the ROF collector, which uses a write barrier only during a minority of promotions into the dynamic area. Thus, the DOF collector will benefit more from a faster write barrier than the other collectors. While the barrier I'm using in Larceny is not the fastest design possible—Stefanović showed how a very fast barrier might be designed—it is clear from results presented in later chapters that the speed of the barrier is not going to make or break the algorithm's practicality, and that is all I'm investigating in this dissertation.

Chapter 4

The ROF collector: design

The renewal-older-first (ROF) collector was first proposed by Clinger, implemented in prototype form by myself, and reported on in a joint paper in 1997 [6].

Clinger observed that the literature suggested that there is little or no correlation between age and future life expectancy among old objects, and that the lack of correlation between age and future life expectancy is also one of the salient aspects of the radioactive decay model. He proposed a generational collection algorithm that assumes that objects die randomly, and showed that it had a theoretical advantage over non-generational collection with regard to its mark/cons ratio. He went on to propose that the algorithm might be effective at managing old data (which are usually managed by what is effectively non-generational collection in the oldest generation of a generational collector), and to make an argument that the secondary costs could be controlled.

The initial implementation of the ROF collector showed some promise but results were inconclusive. In addition, Stefanović simulated a nursery-less ROF collector and reported largely disappointing results in his Ph.D. dissertation [27]. In this chapter I describe the current incarnation of the ROF collector, now rather more mature than our initial implementation, and I describe both my design decisions and outline some new mechanisms that the collector uses to control secondary costs.

The ROF and DOF collectors differ in the way they treat objects that survive a collection: the DOF collector keeps objects in strict age order throughout their lifetimes, whereas the ROF collector does not distinguish between old and young objects following a garbage collection. The differences will become clear throughout this chapter.

4.1 Collector operation

The assumption behind the ROF collector is that on the average relatively younger data will contain less garbage than relatively older data, and therefore garbage collection will be more efficient if the collector only collects among older data [6].

To implement this idea, Larceny's ROF collector divides the dynamic area into two ROF generations, ROF-young and ROF-old; as the names suggest, ROF-young holds younger data than ROF-old. Data are promoted from the ephemeral area first into ROF-old and then, when ROF-old is full, into ROF-young. When ROF-young is full as well, a garbage collection takes place: ROF-old is collected by evacuating live data to space taken from the reserve, and the survivors are merged with ROF-young to become the new ROF-old (objects in ROF-young are not copied). The memory previously used by ROF-old becomes available to use for allocation and the reserve.

A lower level model is more useful to describe the policies and mechanisms of the ROF collector implementation, however. In this model, the dynamic area is divided into *steps* numbered $1 \dots k$. The steps are arranged by the age of the objects in them, from youngest to oldest; step 1 holds the youngest objects. Additional steps are kept in reserve for the copying collector and are not available for allocation. A policy parameter j determines the dividing line between ROF-young and ROF-old: steps $1 \dots j$ are in ROF-young, and steps $j+1 \dots k$ are in ROF-old.

When the dynamic area is collected, steps $j+1 \dots k$ are collected by evacuating live data into steps taken from the reserve. Then the steps are rearranged: Steps $1 \dots j$ become steps $k-j+1 \dots k$, and the s steps containing survivors of the collection become steps $k-j-s+1 \dots k-j$. (Since steps are arranged in age order, the survivors appear to be younger than the data moved from ROF-young, that is, they are "renewed"; it is this rearrangement that gives the collector its name.) The free steps that are not used to replenish the reserve become steps $1 \dots k-j-s$. Following collection, j is adjusted if necessary (see below). Figure 4.1 illustrates the ROF collector in operation.

In order to collect ROF-old efficiently, the mutator maintains a remembered set that tracks pointers from ROF-young into ROF-old. My ROF collector maps the ROF generations onto two of Larceny's generations: ROF-young is mapped to the older generation and ROF-old to the younger. Placing young data in the older generation and vice versa allows the mutator to continue to use the existing write barrier to track pointers from ROF-young into ROF-old. Thus code compiled for the standard generational collector runs unchanged with the ROF collector.

Following collection, j is set to $1/2$ the number of free steps. That particular value ensures that large circular garbage structures will be collected and that ROF-young will not be unreasonably large.

The following sections discuss the details of the design and justify some of the design decisions.

4.2 The collector's scope

The ROF collector manages only the dynamic area and relies on Larceny's standard nursery for allocation. Data are promoted into the dynamic area from

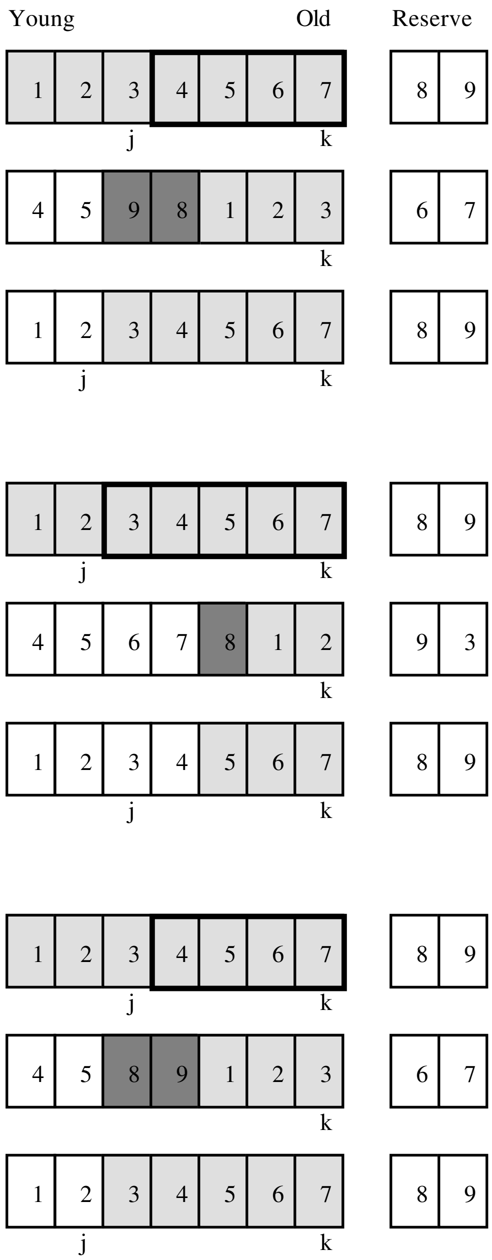

Figure 4.1: The ROF collector as implemented in Larceny. The figure shows triples of collector configurations: before a collection, after a collection but before renumbering of steps and selection of j, and after renumbering and selection of j. Shaded steps are full, and unshaded steps are empty. The 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.

the ephemeral area in the normal manner, and the ROF collector does not preclude multiple ephemeral generations.

The nursery will only be effective if a substantial fraction of all objects die very quickly, but that is probably the case for most Scheme programs; thus, the ROF collector is most reasonably used to manage objects that have survived at least one garbage collection. Additionally, using a nursery for allocation speeds up allocation, as allocating directly in the ROF area would require a write barrier on every initializing store. These issues are discussed in greater detail in the context of the DOF collector (see section 3.2).

4.3 Collecting circular garbage

The ROF collector is unable to collect self-referential garbage structures that are not entirely within either of the dynamic generations, because those parts of the structure that are in the uncollected region will keep those parts that are in the collected region alive.

The simplest way to ensure that circular garbage is collected is to set j such that all live data are in ROF-old following a garbage collection, at least periodically. My ROF collector sets j in that manner following every collection, so circular garbage is not an issue.

4.4 Area size

An ROF collector that does not use a fixed heap size must be able to grow and shrink the heap in response to the program's memory demands.

In order to compute the desired heap size, the collector must know l, the amount of the live data in the heap; the heap size is then computed as $H' = lL$ where L is the inverse load factor. The ROF collector cannot compute l accurately unless it periodically sets j to zero, so it estimates that all of the data previously in steps $1 \ldots j$ are live. Given H' it is simple to compute k and j. Then k is adjusted by adding an amount to compensate for the fact that steps $1 \ldots j$ will not be collected. The effect of the adjustment is to make memory that was logically part of the reserve for the uncollected region available for allocation and collection in the collected region.

The ROF model assumes that all data in steps $1 \ldots j$ are live at the time of a collection in the dynamic area; thus j steps could be added to k. In practice it is unreasonable to expect that all data in steps $1 \ldots j$ are live at that point, so fewer than j steps should be added to k. Therefore $\lfloor fj \rfloor$ steps are added to k where f is a fudge factor that helps estimate how much data in steps $1 \ldots j$ are actually live.

The value of f is selectable at startup.[1] Its default value is 0, which is extremely conservative. A value around 0.5 seems to work well in practice.

[1] The appropriate question to ask when choosing f is "how lucky do I feel?", so the command line switch that selects f is called `-feeling-lucky`.

4.5 Remembered set structure

Functional programming languages do not allow the program to create pointers from relatively older objects to relatively younger objects, as assignments are not allowed. Though semi-functional languages like Scheme and Standard ML as well as some implementation techniques for non-strict functional languages do create old-to-young pointers, young-to-old pointers will usually predominate in functional programs. As a result, the standard remembered set that tracks pointers out of ROF-young is likely to become large and will be dominated by objects that contain pointers only into ROF-old.[2] Even so, that set will be scanned on every garbage collection, even collections that do not involve the dynamic area.

To avoid unnecessary scanning costs, an extra remembered set holds only those objects in ROF-young that reference objects in ROF-old. The extra set is updated following a promotion into ROF-young. At that time, the standard remembered set for ROF-young contains only objects that contain pointers into ROF-old, and it can be folded into the extra set and cleared.

The extra remembered set may grow large for some programs and incur high scanning costs, and it may be as efficient to collect the data in ROF-young as it is to scan and to maintain a large remembered set. ROF-young can be incorporated into ROF-old simply by reducing the value of j, clearing the extra remembered set, and merging ROF-young's standard remembered set into that of ROF-old. That adjustment can be made at any time, not just at the time of a collection in the dynamic area.

The ROF collector uses two strategies to decide when to reduce j. The strategies do not speed up the collector in normal cases, but are instead intended to reduce the susceptibility of the collector to poor performance in boundary cases.

The first strategy is to detect phase behavior in the program. A *phase detector* runs following each promotion into the dynamic area and tracks the size of the remembered set over time. If the remembered set grows rapidly for some small number of collections and then grows very little for the same number of collections then the collector decides that a growth phase is over and that some data structure is split across the two parts of the dynamic area. The ROF collector decreases j so that the entire data structure will be in ROF-old; if the structure is garbage, it will be reclaimed.

The second strategy is to track the absolute size of the extra remembered set, and if it exceeds a set limit then the collector assumes that the cost of maintaining it outweighs the potential benefits of ROF collection. Again the collector reduces j and clears the extra remembered set.

[2] Stefanović found that object-oriented programs have little bias in pointer direction, but my data suggest that for many Scheme programs, there is significant bias in favor of young-to-old pointers. Functional programs encourage young-to-old pointers and shared substructures. Structures with significant sharing may have many pointers between spatially distant objects, and those pointers will be have to be tracked by the remembered set.

4.6 Write barrier

The ROF collector incorporates an additional write barrier that captures those pointers from ROF-young into ROF-old created by promotions into ROF-young. These pointers must be traced when ROF-old is collected. The collector could have opted simply to scan all of ROF-young when ROF-old is collected, but since the additional write barrier is only required on promotions into ROF-young, and ROF-young usually occupies about 1/3 of the heap or less, the barrier is likely to be the cheaper alternative.

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 ROF collector.

Additional write barrier issues are discussed in Sections 2.1.3 and 3.9.

Chapter 5

Methods

5.1 Benchmark programs

5.1.1 Program admissibility

The measurements obtained by running benchmark programs are only as interesting as the benchmark programs themselves, and the criteria for selecting good programs are both numerous and sometimes conflicting. Clearly the programs should be compute-bound and the measurements from running them should be repeatable. It may also be desirable to include some "real" programs in a benchmark suite, as it is the collector's performance on real programs that matters in practice. For studies like mine, however, it is equally important that the benchmark programs provoke interesting behavior in the garbage collectors, allow that behavior to be measured reliably, and finally allow the behavior to be understood. As the behavior of the collector on worst-case programs is of some interest, it may even be attractive to synthesize worst-case benchmarks.

Many programs are easy to garbage collect, usually because they have little live data or a low rate of allocation. A collector should always handle easy programs well, but if we are to learn something about how a collector will behave when it is challenged, it will be necessary to select programs that are abnormal—programs that might have high allocation rates, large amounts of live data, and many kinds of object lifetimes, and often all those features in combination.

I designed my test system for programs that would have high allocation rates and use substantial space, so benchmark programs that invoke the collector only a few times or run in a very small heap are likely to produce measurements that are dominated by edge effects and other noise. The benchmark programs must be matched to the implementation, which in my case means that they must use time and space in amounts that greatly exceed the units of time measurements and memory allocation.

Garbage collection benchmarks are sometimes selected from among real applications on the assumption that it is only the performance of the collector on actual programs that matters. However, real programs are complex and do not lend themselves to detailed analyses, and any changes made to a real program to elicit a particular response from the garbage collector is likely to be partly or wholly masked by other effects in the system. Synthetic programs are a tractable means of studying the details of the collector's behavior and understanding how the collection algorithm can be improved, and should not be excluded from the benchmark suite because they are not "real".

Finally, the benchmark suite needs to have some variation so that the collectors' performance can be understood across a range of program behaviors. Some easy programs will allow me to verify that the collector handles them well, and some real programs will allow me to make predictions about the collector's behavior on programs that are more typical than are most of the benchmarks. Both kinds of programs belong in the benchmark suite.

5.1.2 The benchmark suite

The programs I have selected are programs that I know from past experience are challenging to garbage collection [6], and in addition one real application: an optimizing Scheme compiler.

The number of iterations given in the following program descriptions refer to how the programs are run; for reasons explained later, some of the programs only make sense as garbage collection benchmarks if they are run several times back-to-back in the same process and measurements are taken for all the iterations as a whole.

`earley` is an implementation of Earley's parsing algorithm, written by Marc Feeley [12]. The program first creates a parser from an ambiguous grammar; it then computes all the parses of an input that is constructed to have an exponential number of parses. Almost all data allocated by the program are pairs for the representation of the parses. `earley` constructs its output in a functional manner, and the resulting data structure necessarily contains all young-to-old pointers.

I run `earley` on an input of length 12, for five iterations.

`gcbench` is a synthetic garbage collection benchmark written by John Ellis and Pete Kovac, modified by Hans Boehm, and translated into Scheme by Will Clinger; it emulates the phase behavior of some programs. It starts by "stretching the memory" with a large tree that becomes garbage immediately. Then it allocates some ballast: a large array and a large tree that stay alive until the end of the iteration. Finally it enters a series of phases, where each phase allocates the same amount of memory in the form of trees constructed both top-down (imperatively) and bottom-up (functionally). The height of the trees increases in later phases, so the initial phases create many short trees and the final phases create a few tall trees. Each tree node contains integer data and pointers to other nodes, and `gcbench` uses vector structures for the tree nodes. The large long-lived array is a vector of floating-point numbers. In Larceny all of these floating point numbers are heap-allocated outside the vector. The total volume of ballast data in the program is 13 megabytes.

I have further modified `gcbench` in the following way: given a parameter n, the program performs the stretching phase once and then performs the rest of the program n times.

I run `gcbench` with $n = 5$.

`gcold` is a synthetic garbage collection benchmark that was 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.

Trees are created "top-down", that is, by creating a tree node and then populating it with subtrees. Thus all pointers in the trees initially point forward in time.

`gcold` attempts to abstract the behavior of a program running a long-running computation on a large data structure: much of the data structure is static, but it is updated over time both by adding data to it, removing old data, and moving data around.

The program accepts a number of parameters, the most important of which are the ratio of short-lived data to long-lived data and the mutation rate. The former controls how much allocation the program performs between each time it replaces and swaps subtrees; the latter controls how many subtrees are swapped.

I run `gcold` in three configurations. All configurations have 25 megabytes of large trees and run for 200 iterations. The three configurations differ in the ratio of short-lived to long-lived storage and the amount of mutation work: I use the settings $(1, 0)$, $(1, 1000)$, and $(10, 1000)$.

The three programs should in principle allocate the same amount of storage but several problems with the benchmark implementation that were discovered late prevent the allocation volumes from being equal. I have elected not to correct these problems, and comparisons across settings must therefore be made carefully.

`boyer` is a toy term-rewriting theorem prover derived from the Boyer benchmark in the Gabriel benchmark suite [13]. 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. `nboyer` and `sboyer` differ only in that `sboyer` uses a local tweak—shared consing—to reduce the amount of storage allocation; the tweak also results in a different live storage profile for `sboyer`.

I run `nboyer` with a scaling parameter of 3, as both a single iteration and iterated five times, and I run `sboyer` with a scaling parameter of 4, iterated five times.

`perm` is a synthetic garbage collection benchmark that was written by Will Clinger, Gene Luks, and myself. 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 that all represent all permutations of the the first eight integers. As new data structures are allocated, old data structures are removed from the queue and become garbage.

There are 40,320 such 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 I have chosen the survival rate out of the nursery is 100%.

I run `perm` in two configurations, called `perm200,1` and `perm25,8`. These two programs allocate the same amount of data—computing all permutations 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 [25]. `twobit` 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.

Program characteristics are shown in Table 5.1. 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.

In all cases, the heaps are large relative to the granularity of block allocation in the test system (64–256 kilobytes), and the garbage collectors are usually invoked several tens to hundreds of times during a run. In some programs the ephemeral-area collections are effective at filtering out short-lived storage, however, so the dynamic-area collector are not always run more than a few times (consult the tables in Appendix A for specific data). That particular issue is discussed in the next section.

Program	Lines of code	Allocation volume	Peak live (est.)	Promotion rate	Mutator time (stop+copy)
`5earley12`	658	299.0	13.5	0.21	11.04
`gcbench5`	226	1757.0	16.8	0.27	19.12
`gcold1,0`	381	347.0	26.5	0.44	5.37
`gcold1,1000`	381	402.0	26.5	0.38	40.56
`gcold10,1000`	381	264.0	26.5	0.06	36.89
`nboyer3`	767	99.0	13.0	0.45	4.30
`5nboyer3`	767	497.0	13.0	0.44	20.40
`5sboyer4`	781	259.0	13.4	0.26	61.41
`perm200,1`	324	229.0	11.5	1.00	4.44
`perm25,8`	324	229.0	11.5	1.00	4.55
`twobitlong`	23,789	665.0	7.9	0.08	138.13
`twobitshort`	23,789	119.5	7.5	0.18	22.17
`5twobitshort`	23,789	575.5	7.5	0.17	108.17

Table 5.1: Characteristics of the benchmark programs. Allocation volume and peak live 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; it may be subject to slight variations. The mutator time is given in seconds and averaged across runs at several settings; the mutator times of the stop-and-copy collector are reported because those are generally the lowest.

Figures in Appendix C show the storage profiles for all the benchmark programs.

5.1.3 A brief defense of the benchmark suite

The primary problem of the benchmark suite is that it is small and provides a test of a somewhat narrow range of program behaviors; it is clear that the dissertation would be stronger with a larger benchmark suite. Even so, later chapters show that the benchmarks have been effective in ferreting out a variety of problems with the older-first collectors, and I think that suggests that the suite is adequate.

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 and the mark/cons ratio are dominated by the collections that promote data into the dynamic area. Table 5.2 shows that the larger fraction of the time is spent on major collections in most programs; however, 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`.

	Major GC	Minor GC	Ratio
5earley:12	1392	1684	0.83
gcbench:5	18824	10534	1.79
gcold:1,0	6947	4072	1.71
gcold:1,1000	7025	4186	1.68
gcold:10,1000	919	538	1.71
nboyer:3	1498	1154	1.30
5nboyer:3	6784	5169	1.31
perm:200,1	11393	3955	2.88
perm:25,8	5578	4262	1.31
5sboyer:4	1831	1816	1.01
twobitlong	892	4009	0.22
twobitshort	652	907	0.72
5twobitshort	2804	3461	0.81

Table 5.2: Average time (in milliseconds) spent in collection and promotion in 2GEN, and the ratio of the two averages. The averages are computed across all runs for each benchmark.

5.2 Obtaining measurements

5.2.1 The benchmarking system

My test system is Larceny, a research implementation of Scheme [25]. Larceny accommodates several garbage collectors, including traditional generational and nongenerational collectors, conservative collectors, and the novel generational collectors discussed in this dissertation.

Larceny uses the Twobit optimizing compiler for Scheme [5]. Twobit performs Scheme-oriented optimizations like lambda lifting and representation recovery as well as traditional optimizations like common subexpression elimination and register allocation. It produces code for a portable register machine, and Larceny translates that code to machine code for the SPARC architecture [26]. Some aspects of Twobit and Larceny that are relevant to interpreting the measurements are described in Section 2.4.

Larceny itself consists only of the memory managers and other core functionality; the Scheme libraries and the benchmark programs are kept in heap images that are loaded from disk on startup. There is a separate heap image for each benchmark containing only that program's code, the standard libraries, and a simple interpreter. The heap images are about 1.5 megabytes in size, except for the images for `twobit`, which are about 3.5 megabytes.

The execution time and other reported events were measured by instrumentation in Larceny itself and does not include time to start up or terminate the Scheme process. Low-level event counters were only enabled to obtain event counts; most runs do not use them.

The benchmarking system is a Sun Ultra 5 workstation running Solaris 2.6. It has 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.

5.2.2 How the benchmarks were run

I obtained measurements while the benchmarking 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. It would have been preferable to take the machine down to single-user mode and disconnect it from the network to reduce sources of noise, but I judged that to be relatively unimportant; it was also impractical, as I was not co-located with the machine.

As mentioned earlier, some of the programs in the benchmark suite only make sense as garbage collection benchmarks if they are iterated. Neither `earley` or `sboyer` are suitable uniterated. Their storage use grows monotonically and little garbage is generated, so the programs reach their peak size with little opportunity for garbage collection. In fact, the dynamic-area collectors of the generational collectors will not run during the entire first iteration in some configurations.

Iterated processes are not uncommon, so I have chosen to run some of the other programs iterated as well, and of course programs like `gcbench` and `twobitlong` are naturally iterative.

5.2.3 Collector parameters

In Section 2.1 I described how the collectors are controlled by a user-selectable load factor $1/L$ for runs with variable heap size, and a similar user-selectable collection trigger $1/T$ for runs with fixed heap size. In addition, the collectors are controlled by the settings for the heap sizes, and by collector-specific parameters; the latter are described in Chapters 3 and 4.

In order to generate more data points and to examine the behavior of the collector under varied conditions, I ran the benchmarks with several groups of settings. These groups are called *upper*, *upper+lower*, and *variable*, and the names describe restrictions on the heap size during the run. The groups *upper* and *upper+lower* are jointly also called *fixed*.

Runs in the *upper* category place an upper limit on the heap size, though the collector is free to use less memory than the limit. The collection trigger $1/T$ controls how the collector divides the heap into a working area containing survivors of the previous collection and recent allocation, and a reserve for the next collection. I ran tests with $T = 3.0$ only.

Runs in the *upper+lower* category place both upper and lower limits on the heap size. In all runs the lower limit is set 2 megabytes below the upper limit; thus the collectors have a little room to maneuver, but not much.[1] The collectors cannot guarantee that they will use any particular amount of memory, but given

[1] The *upper+lower* category is effectively a hack that allows the collectors to operate in something resembling a fixed heap size.

an estimate of the amount of live data and a trigger, they size the allocation area in an attempt to use at least as much as the lower limit. The result of a significant underestimation of the amount of live data is most likely memory exhaustion and termination of the run. I ran tests with $T = 2.0$ and $T = 3.0$.

For both the *upper* and *upper+lower* runs, I chose five heap sizes, spaced at least two megabytes apart, in a range roughly between twice and three times the estimated peak size of the program.

Runs in the *variable* category place no upper or lower limits on the heap but control the sizing of the heap by the load factor alone. I used settings of L at 2.25, 2.5, 2.75, and 3.0.

In summary, the collectors were run at relatively high effective loads. I chose a narrow range of high loads because the differences between the collectors will be accentuated when they are running with constrained resources. In practice, as the amount of memory available increases, the differences between the collectors will diminish.

5.2.4 How the results are reported

I report CPU times and the best of three runs rather than elapsed time and the average of three runs (say) in order to reduce the amount of noise in the measurements. No benchmark program requires even nearly as much RAM as the test system has, so paging was never an issue.

The ROF and DOF collectors are controlled by several parameters in addition to the heap size settings, the load factor, and the trigger. When I compare the reference collectors to the older-first collectors, it is not reasonable to report results for all combinations of settings of the additional parameters. I have therefore chosen a set of *canonical settings* of these parameters for each collector, and for the most part I report only results for the canonical settings. The canonical settings for a collector are those that result in the smallest average total execution time for a subset of the benchmarks. (A subset was used because the canonical settings were determined before all programs had been added to the benchmark suite. `perm` and `gcold` were not used to determine the canonical settings.)

The measurement data from the runs are laid out in unprocessed form in tables in Appendix A, and in greatly reduced form in figures in Appendix B. In most cases, the reduced data incorporate all the data points from the canonical runs, from all three groups of settings (*upper*, *upper+lower*, and *variable*). In some cases I use only a subset of the data; for example, in my analyses of floating garbage I have used only data from the *upper+lower* runs, as the reduced variation in heap size makes the analysis easier.

I have purposely not measured or reported the length and clustering of pauses incurred by the different garbage collectors. Though these data are of interest, I have simply chosen not to focus on them.

5.2.5 Garbage collectors are brittle

The behavior of the garbage collector can be sensitive to small perturbations in the user program; by accident, I discovered exactly how sensitive.

My initial measurement infrastructure was written entirely in Scheme. Before an iteration, the infrastructure would obtain the current values of all the performance counters, and these values would be saved until after the iteration, when the values would be obtained again, and the differences would be computed and printed. Then the next iteration of the benchmark would be run, in the same process.

Since different collector configurations have different numbers of performance counters (for example, there are per-generation counters), different configurations will allocate slightly different amounts of storage and have slightly different live storage profiles due to the data retained during an iteration. In one case a DOF collector run of `5nboyer3` with the shadow remembered sets (see Section 3.8) allocated about 40 kilobytes more storage than the run without the shadow sets. This amount of allocation was entirely due to larger performance counter data structures in the run with the shadow sets (both their allocation and processing). Total allocation for the run was about 520 megabytes of storage.

That minor (0.008%) increase in allocation volume and change in storage profile resulted in an increase of 24 megabytes in the amount of storage copied by the collector, that is, an increase of 4.6%.

This was not an isolated incident: changes were noticeable across the entire benchmark suite, with increases in copying of up to 7.6%, and occasional decreases as well.

I changed my infrastructure so that it allocates very little, retains no measurement data during the iteration, and allocates what little it allocates in a way that does not depend on the garbage collector configuration. There is no longer any difference in the amount copied between runs that use the shadow remembered sets and those that do not, and the difference in allocation between runs that use the benchmarking infrastructure and those that just run the program without any measurements is very small.

Still, at least one run in a configuration that uses the benchmarking infrastructure copies fully 20 megabytes more than a comparable run that does not use the infrastructure. While that does not have implications for the accuracy of my data, it emphasizes how brittle the results can be.

5.3 Comparing collectors

5.3.1 Qualitative comparison

In addition to the quantitative measures I discuss in the next section, I also examine the collectors from a qualitative point of view. Qualitative measures include the frequency and nature of garbage collector breakdowns, a collector's adherence to given parameters like the inverse load L, its ability to control worst-case behavior like explosive growth of the remembered set and other auxiliary data structures, and its ability to handle programs with difficult data structures, like very large objects or structures with atypical pointer configurations. Sometimes the interpretation of these measures can be a little subtle; for example, garbage collection failures can be indicative both of a poor collector and of an aggressive collection policy.

Non-generational collectors are usually simple: they break down when the heap is close to full and collection cost dominates, or when an auxiliary data structure like a mark stack overflows. Simple generational collectors may perform better than non-generational collectors much of the time, but may perform badly if data have unfavorable lifetimes or the programs spend a lot of time in the write barrier. Older-first collectors may perform badly if the data structures in the heap increase the size of the remembered set or the amount of copying through popular objects, float, or circular garbage.

5.3.2 A measure of relative resource use

Throughout the subsequent chapters I use a measure I call the *advantage* of a system A over another system B to express how much better A is than B, and I compute the advantages for both time and space. Advantage is closely related to the conventional notion of speedup:

$$\text{speedup} = \frac{\text{time}_{\text{original}}}{\text{time}_{\text{enhanced}}}.$$

While speedup is measured on the interval $(0, \infty)$, advantage is measured on the interval $(-\infty, \infty)$. An advantage of 0 means the systems being compared perform equally well, and an advantage of 100 to one system over the other means the inferior system consumes 100% more of the resource in question than the superior one:

$$\text{advantage}_{A,B} = \begin{cases} 100(1 - B/A) & \text{if } A < B \\ 100(A/B - 1) & \text{otherwise.} \end{cases}$$

Using this formula, an advantage for A over B is expressed as a negative number, a trick that allows advantages for both systems to be plotted along a single number line. Figure 5.2 shows an example of how the advantages can be displayed graphically.

The sign is of course only a convention, and in those cases where I am scrutinizing one system by comparing it to others—as opposed to simply comparing two systems without showing more interest in one than the other—I usually elect to use a positive sign for the system under scrutiny even though that convention is the opposite of the one used by the formula above. Usually this causes no confusion at all, as it is generally clear from the context by which sign the advantage of a system is denoted.

5.3.3 Quantitative comparison

Garbage collectors attempt to find a reasonable trade-off of time and space, and should be judged on how successful they are at that. It is straightforward to compare measurements of time or space across several collectors but harder to incorporate both time and space in a comparison that aims to rank the collectors.

A time-honored approach is to fix space and measure time, and then compare collectors by time. However, a collector usually does not and often cannot guarantee that it will use the amount of space that it is given. To mention only two examples: a copying collector uses a policy parameter, the collection trigger, to guess how much memory it needs to set aside for garbage collection; and a reference counting collector will never use more memory than the peak amount of live storage. Fixing space and measuring time is not much different from fixing time and measuring space: unless the collector is willing to deliberately waste time, there is no guarantee that the given amount of time will be used exactly [24].

That might seem like a trifling objection, but the discrepancy between the amount of space provided and the amount used can be in the tens of per cent of the total heap size. The fix is also not as simple as saying that the implementation "should be better"; though all the collectors interpret the collection trigger the same way, they do not always compute the amount of live data the same way, and will therefore use different amounts of memory. (Recall from previous chapters that some garbage collectors never know exactly how much memory is live and must estimate it.) For these reasons it is preferable to take the measured space usage into account when assessing a garbage collector's performance.

In the simplest case a ranking function takes both time and space into account and returns a number; the values computed for different runs can be compared directly to determine a ranking.

One such ranking function is the unweighted space/time product. The space/time product is predictable in that if space grows by some percentage, the product grows by the same percentage. The product says *something* that may be meaningful since space and time can be traded; the product therefore at least approximates, in a predictable way, a measure of the collector's effectiveness at making the tradeoff on a given program.

However, the unweighted space/time product is a flawed measure. Consider something as simple as the model for the mark/cons ratio of a non-generational mark/sweep collector:

$$r = \frac{1}{L-1}.$$

If we take that equation as a measure of time, and L as a measure of space, then the space/time product is simply $L/(L-1)$, which decreases as L increases. If the space/time product is to be a useful measure of garbage collection resource use, then two runs that are on the curve of r ought to have the same space/time product, but they do not: the collector that uses more memory will have a lower space/time product.

A weighted or biased space/time product might work better, but even then it is not any clearer exactly what the space/time product means.

A good ranking scheme does not need to provide an absolute ranking of the collectors, only to provide a coherent view of the benchmark data. I have chosen a method that compares two runs by their space and time advantage values. The basis for the method is the *space/time diagram*, an example of which is shown in Figure 5.3. Each point on the space/time diagram shows the time and space advantages of two collectors running on the same program with comparable parameter settings. The plot provides an overall visual idea about how the two collectors compare.

The space/time diagram suggests the following method for comparison. Each program and parameter setting pair is a contest. If one collector beats another on both space and time—that is, if a point lies in quadrant I or quadrant III—then the collector with both advantages is a clear winner of that contest. The problem appears when one collector has an advantage on space but a disadvantage on time, or vice versa. In this case a collector is a winner if it has a relatively larger victory along one axis than the other has along the other axis; that is, if collector A has an advantage of X on space but collector B has an advantage of Y on time, then A is the winner of that contest if $X \gg Y$, and B is the winner if $Y \gg X$. The algorithm effectively enlarges quadrants I and III, as shown in Figure 5.1.

The algorithm will not classify all runs, but that does not matter, as it will in practice classify many of them and provide a summary of how collectors fare when they are compared simultaneously by space and time.

The exact comparison algorithm I have settled on is this: for a collector to win, its winning advantage must be more than 2.0 and must be more than 1.5 times its opponent's winning advantage. The cutoff at 2.0 is present to reduce the effects of noise in the measurements. Memory is allocated in blocks of 64–256 kilobytes, and the timer resolution is 10 milliseconds, so small differences are not meaningful. The factor 1.5 was chosen as a happy medium between including points close to the axes and excluding points in the middle. It is possible that that factor should not be constant, but vary over time, or that it should be an absolute difference rather than a relative one, but I have not followed it up further. The lines in Figure 5.1 reflect the cutoff and the advantage factor.

I have chosen to use total CPU time and heap space in the space/time comparisons throughout the dissertation. The obvious alternative would be to use total space—heap space plus remembered set space—rather than just heap space, and doing so would be more realistic in the sense that it measures the space the implementation actually uses. The DOF collectors use a particularly large amount of remembered set space, and leaving it out skews the results in their favor.

When I have chosen to use heap space, it is because the remembered set space consumption is amenable to tuning, both by choosing different representations and by improving the collection algorithms. Though the space/time diagrams now show an idealized situation rather than a real situation, they are indicative of what the situation may be like when more work has been performed on the DOF collector.

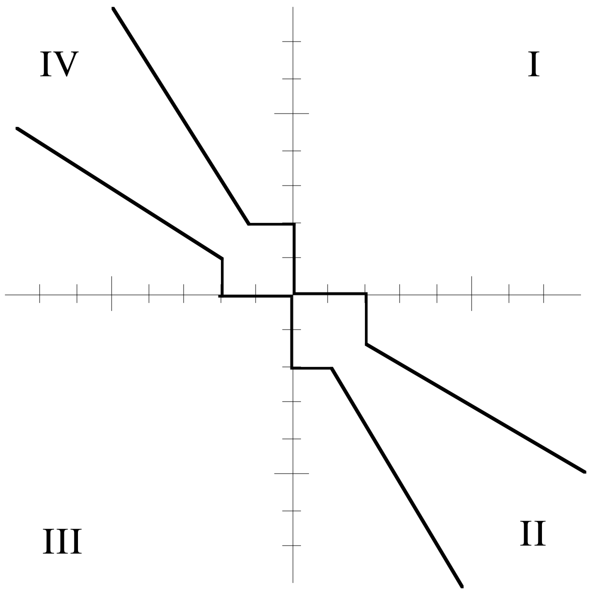

Figure 5.1: The quadrants of the space/time diagram as adjusted by the comparison algorithm: Quadrants I and III are enlarged, as shown by the thick lines, at the expense of Quadrants II and IV. Each tick represents one point of advantage.

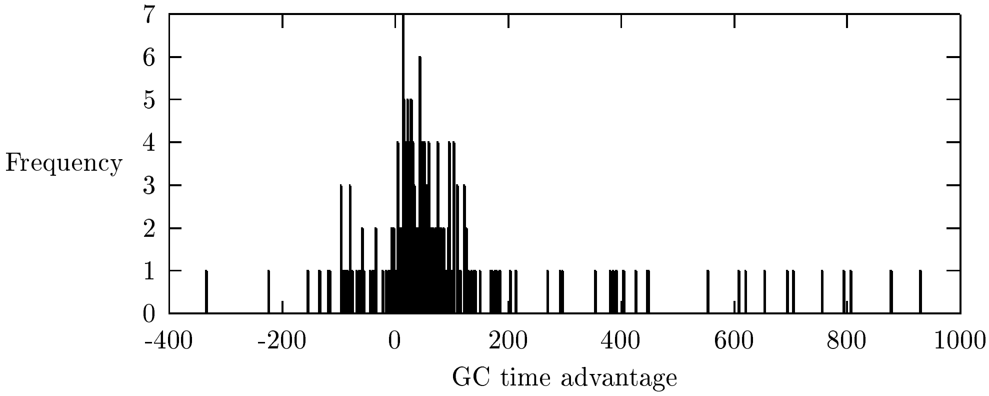

Figure 5.2: A diagram showing advantages in GC time for SC (negative x-axis) and 2GEN (positive x-axis). Each bar is two percentage points wide, and its height represents the number of comparable runs for which the better system had an advantage corresponding to the bar's distance from the origin (see the text). The bar of height 3 at 104 shows that on three runs, 2GEN had an advantage over SC of 103–105 points.

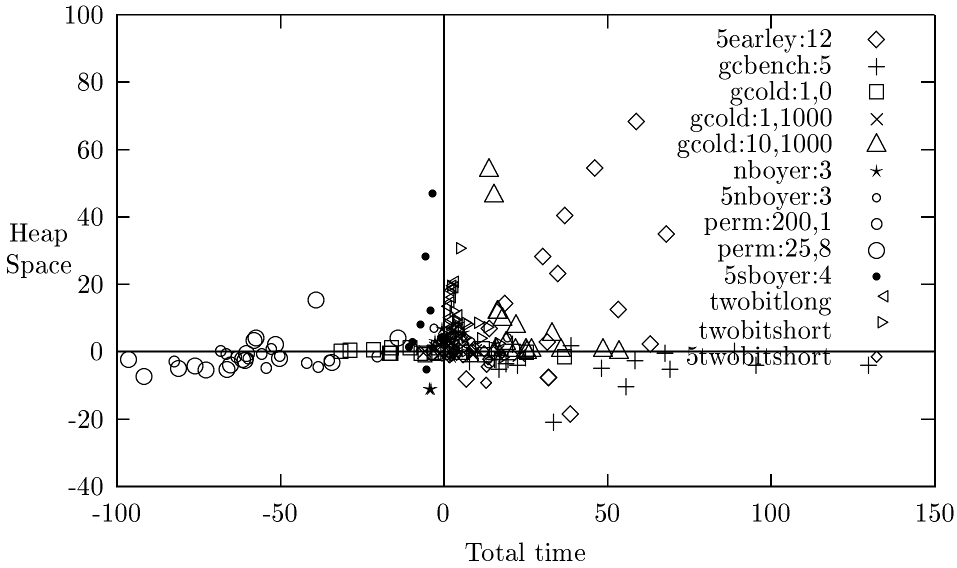

Figure 5.3: A space/time diagram showing advantages in time and space for SC (negative axes) and 2GEN (positive axes). The point at (68,35) shows that on this particular run of `earley`, 2GEN had a 68 point advantage in time and a 35 point advantage in space.

Chapter 6

The reference collectors: analysis

In this chapter I report on the performance and characteristics of the reference collectors: the stop-and-copy collector (SC) and the 2-generational and 3-generational collectors (2GEN and 3GEN).

This dissertation aims to demonstrate the practicality of the older-first collectors, and a thorough examination of the reference collectors is beyond its scope. However, as I will be using the standard generational collectors as reference points for the analyses of the older-first collectors, I want to establish that they are of sufficient quality. I do that by showing that they perform well compared to the stop-and-copy collector in a number of ways, and by appealing to the reader's good will when I now claim, without presenting any evidence, that the performance of my stop-and-copy collector is good.[1]

6.1 Performance of the reference collectors

The space/time diagram in Figure 6.1 shows that 2GEN usually has an advantage over SC both on total time and heap space, though there are exceptions—SC is clearly better on `perm` and `gcold`.[2] As discussed later, SC sometimes has advantages on mutator time and space, but those advantages are not usually large enough to make up for the significant disadvantage the collector has on GC time on many programs. (The Figures in Appendix B provide substantially more detail.)

The diagram in Figure 6.1 also shows that SC is more competitive against 3GEN than against 2GEN. 3GEN does not perform well for reasons that are discussed below, though its additional generation is beneficial on `gcbench` where

[1] It is hard to implement a stop-and-copy collector poorly—its performance is in some sense an absolute reference point. I invite the specially interested and the skeptical to examine the source code [21].

[2] See Section 5.3.3 for a brief justification for the use of heap space rather than total space.

	Mark/cons ratio			Major GCs		
	SC	2GEN	3GEN	SC	2GEN	3GEN
`5earley:12`	0.71	0.34	0.60	40.5	8.4	7.8
`gcbench:5`	1.44	0.69	0.43	278.0	84.3	31.1
`gcold:1,0`	1.28	1.02	1.61	23.4	10.8	11.1
`gcold:1,1000`	1.30	0.90	1.54	27.3	11.0	11.2
`gcold:10,1000`	1.29	0.14	0.29	17.9	1.2	0.9
`nboyer:3`	0.93	0.87	1.16	14.1	7.0	4.9
`5nboyer:3`	0.86	0.87	1.15	79.4	33.7	24.7
`perm:200,1`	1.93	2.76	3.26	40.9	40.5	36.7
`perm:25,8`	0.91	1.74	2.41	29.5	29.0	27.2
`5sboyer:4`	0.83	0.46	0.77	53.3	10.3	9.7
`twobitlong`	0.37	0.12	0.15	89.6	8.5	3.4
`twobitshort`	0.61	0.32	0.65	18.3	3.9	2.9
`5twobitshort`	0.59	0.31	0.68	81.6	18.8	12.9

Table 6.1: Average mark/cons ratios and number of major GCs for the reference collectors. A major GC is a collection that collects the dynamic area, or any collection in the SC collector. The averages hide certain facts: 2GEN and 3GEN sometimes run at lower heap sizes than SC, and runs at low heap sizes pull up the averages for the mark/cons ratio and number of major collections, making 2GEN and 3GEN look worse relative to SC than they actually are.

it reduces the number of times the program's large permanent data structures are copied.

The average mark/cons ratios for the collectors, shown in Table 6.1, echo the facts of the space/time diagrams. In most cases the mark/cons ratios of SC are at least twice as high as those of 2GEN, though often lower than those of 3GEN. Only on `perm` does SC have a significantly lower mark/cons ratio than the generational collectors. Since the generational collector cannot hope to have a mark/cons ratio below 1 on `perm`, as the survival rate out of the nursery is 100%, it is relatively easy to beat.

The intermediate generation of 3GEN may be able to keep data of some lifetimes from being promoted into the dynamic area, and in any case it reduces the frequency of major collections. If there is a large amount of live data in the dynamic area, then the reduction in major collections may reduce the overall mark/cons ratio. Table 6.1 shows that the reduction in major collections pays off handily on `gcbench`. That benchmark is a perfect fit for 3GEN, because it has a great deal of permanent data and a large volume of shorter-lived data structures that are promoted into the intermediate generation but for the most part never into the dynamic generation. As a result, 3GEN has just over a third as many major collections as 2GEN.

On other programs, 3GEN is usually substantially worse than 2GEN, despite reductions in the number of major collections. The likely reasons are partly endemic to multi-generational collectors, and partly implementation artifacts.

First, the size of the intermediate generation in 3GEN is fixed. That makes collection cost in 3GEN roughly independent of the heap size, and it effectively destroys the usual benefit of a larger heap on the mark/cons ratio.

Second, data are promoted into the dynamic area only when the intermediate generation is full, so it is possible for data to be copied repeatedly in the intermediate generation before being advanced. Even when a long-lived object is promoted directly from the intermediate generation into the dynamic generation, the stop-over in the intermediate generation will have doubled the promotion cost of the object relative to 2GEN.

Third, a conservative collection policy in the dynamic area must choose to collect the entire area if the survivors from the ephemeral area might not fit. Given a fixed amount of memory, 3GEN's ephemeral area will be larger and its dynamic area will be smaller than in 2GEN, both factors that will increase the frequency of major collections. (This does not appear to be a factor in my results, however, as 3GEN always has a smaller average number of major collectors than 2GEN.)

6.2 Secondary costs of generational collection

6.2.1 The write barrier

The stop-and-copy system will usually have better mutator times because it does not require a write barrier. Though a program may be written in a functional style, it may still call on library functions that are implemented using imperative techniques. The absence of the write barrier in the stop-and-copy system has a positive impact on mutator times on most of the benchmark programs, as shown in Figure B.18.

The advantage of the stop-and-copy system is especially large on `gcold`, `perm`, `gcbench` and `boyer`. These programs have high volumes of assignments—75 million in `gcbench` and 123 million in `5sboyer`, for example—so it's no wonder the write barrier makes a difference.[3]

Curiously, the generational system has significant advantages in mutator time on `earley` and `twobitshort`. The advantage on `earley`—about 8–10%—is possibly due to different details of stack cache management in the two systems. The stack cache is kept in the allocation area. The allocation area is larger in the generational system, and programs with a deep stack—like `earley`, whose stack grows to 1.4 megabytes—will incur fewer comparatively expensive stack cache underflows in that system. Measurements show that 2GEN in fact has 15%–20% fewer underflows than SC.

[3]As I point out elsewhere, Larceny's write barrier is not as finely tuned as it could be. As it is, the overhead from the write barrier in the generational collector is about 40 to 60 nanoseconds per store. A simple restructuring of the barrier would filter out stores of non-pointers earlier and would therefore reduce the cost of the barrier in `boyer` substantially—the program performs no pointer stores at all. Other improvements, like faster filtering using addressing tricks or marks stored in the object header, are possible as well.

I do not know why 2GEN has better mutator times on `twobitshort`; its advantage is in any case small.

6.2.2 Remembered sets

The generational collectors use space outside the heap for the remembered set. As Table 6.2 shows, the space required for the remembered set is moderate, and never exceeds 2% of the heap size in 2GEN and 3% of the heap size in 3GEN.

The space use reported in Table 6.2 is really a conservative estimate of remembered set size: it is the peak space allocated to the remembered set. As the remembered set is grown in fairly large increments, small remembered sets all show as having the same size, and the generally larger remembered sets in 3GEN than in 2GEN are entirely an artifact of 3GEN having a larger remembered set data structure to accommodate more generations. The remembered sets do grow beyond their minimum size on `twobit`, but otherwise not at all.[4]

	2GEN	3GEN
`5earley:12`	0.27	0.40
`gcbench:5`	0.27	0.40
`gcold:1,0`	0.27	0.40
`gcold:1,1000`	0.27	0.40
`gcold:10,1000`	0.27	0.40
`nboyer:3`	0.27	0.40
`5nboyer:3`	0.27	0.40
`perm:200,1`	0.27	0.40
`perm:25,8`	0.27	0.40
`5sboyer:4`	0.27	0.40
`twobitlong`	0.29	0.45
`twobitshort`	0.34	0.49
`5twobitshort`	0.34	0.53

Table 6.2: Average peak remembered set size, in megabytes, across all runs for a benchmark.

Table 6.3 shows the amount of remembered set work—objects inserted, objects scanned, and words scanned—across the benchmark suite. These data are mostly unexciting, though some features are notable. On `twobit`, the number of objects scanned is three to four times higher for 3GEN than for 2GEN; however, the number of words scanned is never more than 1.5 times higher. Furthermore, 3GEN has only a fifth the number of insertions of 2GEN on `gcbench`, yet the numbers of objects and words scanned are comparable.

[4] It would be better to show the actual remembered set peak sizes here as the number of elements recorded, but Larceny does not record those data. The discussions in later chapters illustrate that the data that are recorded, along with the remembered set traces in Appendix B, are sufficient to establish that some collectors have genuinely larger remembered sets than others.

In the case of `twobit`, the relatively larger number of objects scanned in 3GEN is explained by the remembered set structure: 3GEN has more and larger remembered sets, and they must be scanned more often than in 2GEN. The smaller difference in the number of words scanned is explained by the presence of very large objects in the dynamic area. Most pointer-containing objects manipulated by `twobit` are pairs. On `twobitlong` only 823,000 words would be scanned by 3GEN if all objects in the remembered set were pairs, but that is only 1% of the actual volume scanned. What is probably happening here is that the scanning of Larceny's symbol table accounts for a large fraction of all the scanning work. The table can grow quite large during the `twobit` benchmark.[5] Since the table is so large and will be scanned roughly the same number of times in both systems, it will dominate the scanning statistics.[6]

The reason for the smaller number of insertions for `gcbench` in 3GEN is quite obscure and is related to the distinction between the pointer stores recorded by the barrier and the insertion of data about those stores into the remembered set. The latter number is reported here, and it may be lower in 3GEN than in 2GEN in the following circumstance. In those situations where 3GEN chooses to collect in the intermediate generation rather than to promote into it, the remembered set tracking pointers from the intermediate generation into the nursery is ignored, and objects slated for the insertion into that remembered set are not actually inserted. As we have seen, `gcbench` encourages the collector to keep its operation local to the intermediate generation; this will require it to collect that generation quite frequently.

6.2.3 Float

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, the latter objects must be copied by the collector. They are called floating garbage, or just float.

I measured the amount of float in 2GEN and 3GEN by using the DOF collector's whole-heap collector to remove dead objects from the remembered

[5]Twobit, like many other Scheme programs, uses Scheme "symbol" data to represent the symbols in its internal representations. Symbols are system-wide and unique and are kept in a global data structure; Twobit tends to grow the symbol table quite a bit. Following `twobitshort` the table has 41,000 elements, and following `twobitlong` it has 68,000 elements. The symbol table in Larceny is represented as a single large vector, and the entire vector will be scanned on every collection if it is in the remembered set.

[6]There are actually two of these tables. When the heap image for the benchmark is created, the symbol table is placed with all other data in the static area; it contains about 10,000 elements. At some point during the run the compiler causes the table to overflow, and a larger table is allocated in the garbage collected heap. The old table remains in the static area, however. Since the old table contains pointers to new symbols in the garbage collected heap, it will remain in the remembered set for the static area, and will be scanned on every garbage collection! The excess work can be avoided if the symbol table manager clears the old table after reallocating a new one, since it will then be removed from the remembered set.

	Objects inserted		Objects scanned		Words scanned	
	2GEN	3GEN	2GEN	3GEN	2GEN	3GEN
`5earley:12`	7.7	9.0	3.2e3	3.3e3	6.5e3	6.6e3
`gcbench:5`	8.1e3	1.5e3	32.7e3	31.5e3	19.6e6	19.7e6
`gcold:1,0`	2.2e3	1.1e3	5.7e3	6.5e3	30.9e3	40.2e3
`gcold:1,1000`	5.8e3	10.6e3	9.8e3	26.6e3	42.0e3	103.7e3
`gcold:10,1000`	1.6e3	4.9e3	4.2e3	36.9e3	9.8e3	107.8e3
`nboyer:3`	64.0	64.0	1.9e3	1.9e3	3.8e3	3.8e3
`5nboyer:3`	325.0	325.0	9.5e3	9.5e3	18.9e3	19.0e3
`perm:200,1`	604.1	472.8	2.8e3	2.9e3	7.0e3	7.2e3
`perm:25,8`	621.6	473.0	2.9e3	2.9e3	7.1e3	7.3e3
`5sboyer:4`	91.8	92.0	4.9e3	5.0e3	9.9e3	10.0e3
`twobitlong`	176.8e3	149.4e3	190.1e3	421.6e3	61.9e6	84.7e6
`twobitshort`	73.2e3	68.1e3	75.0e3	326.0e3	5.3e6	7.7e6
`5twobitshort`	367.9e3	327.9e3	378.8e3	1.7e6	15.4e6	20.1e6

Table 6.3: Average amount of remembered set work for the younger-first collectors.

set before every garbage collection (see Section 3.5). This operation does not disturb the heap, so the differences in copy volumes between the normal runs and the float-reducing runs are reasonable measurements of float. Table 6.4 shows the results: float adds 38% to the volume copied by 2GEN on `gcbench`, and it also adds 10–30% to the the volume copied by both collectors on `twobit`.

The larger amount of float on 3GEN than on 2GEN is possibly a result of copying collections in the intermediate generation when parts of the data in that generation are floating garbage objects.

Table 6.4 shows that `gcbench` has a large amount of float in 2GEN, but not in 3GEN. `gcbench` relies in part on imperative update of data structures, which creates the kind of configurations that cause float. Though 3GEN has at least as many intergenerational pointers as 2GEN, it has little float because it collects the intermediate generation often: most intergenerational pointers point from the intermediate generation into the nursery, and the collections in the intermediate generation ignore all entries in the remembered set that track those pointers.

A more subtle and probably less important reason for float in `gcbench` is that it is written in such a fashion that a properly tail-recursive language implementation [6] will allow left subtrees constructed during the top-down phase to be reclaimed before right subtrees are constructed:

	Percent increase		Volume increase (megabytes)	
	2GEN	3GEN	2GEN	3GEN
`5earley:12`	0.0	0.0	0.0	0.0
`gcbench:5`	38.1	4.6	290.8	29.7
`gcold:1,0`	0.0	0.0	0.0	0.0
`gcold:1,1000`	0.0	0.6	0.0	3.4
`gcold:10,1000`				
`nboyer:3`	-0.3	0.1	-0.2	0.1
`5nboyer:3`	0.0	1.0	-0.1	4.8
`perm:200,1`	0.0	0.0	0.0	0.0
`perm:25,8`	0.0	1.6	0.0	7.9
`5sboyer:4`	0.0	0.1	0.0	0.2
`twobitlong`	18.8	29.6	11.6	23.0
`twobitshort`	10.4	27.9	3.4	16.8
`5twobitshort`	14.6	24.3	21.6	75.8

Table 6.4: Average float for 2GEN and 3GEN. Negative numbers (as well as nonzero numbers on `boyer`, which has no float) are artifacts of poorly controlled initial conditions or changes in collector invocation patterns due to reductions in copying during the measurement runs. Blank entries were not measured but will be zero.

```
(define (Populate iDepth thisNode)
  (if (<= iDepth 0)
      #f
      (let ((iDepth (- iDepth 1)))
        (node.left-set! thisNode (make-empty-node))
        (node.right-set! thisNode (make-empty-node))
        (Populate iDepth (node.left thisNode))
        (Populate iDepth (node.right thisNode)))))
```

The last recursive call to `Populate` leaves `thisNode` as garbage unless it's referenced from somewhere else. The initial call to `Populate` does not retain a pointer to the topmost node, so every left subtree can be reclaimed as the right subtree is constructed.[7] The reason this arcane detail affects float is this: if `thisNode` is promoted into an older generation then it and its left subtree are no longer reclaimable by collections in the nursery, and those parts of the left tree still in the nursery will instead be counted as float when the right subtree is being built. Thus the measured amount of float is higher than it would be in a language that is not properly tail-recursive, though the amount of copying may be the same.

[7]The Java program that `gcbench` is translated from did not have this property, but trivial dead-code analysis by the compiler is sufficient to transform the program into the form shown above. To my knowledge no current Java implementations are properly tail-recursive, however.

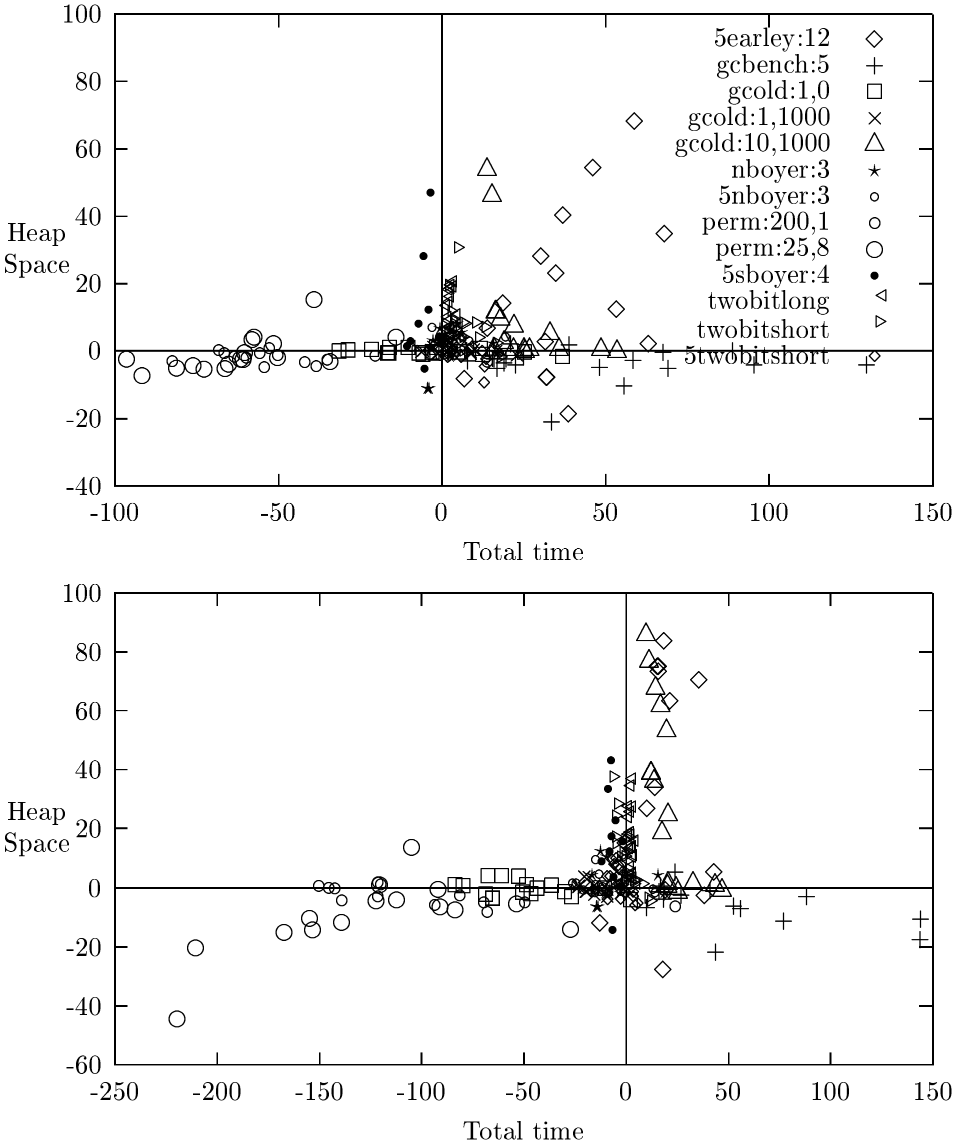

Figure 6.1: Space/time diagram comparing runs for SC and 2GEN (top) and SC and 3GEN (bottom). SC has an advantage along the negative axes.

Chapter 7

The DOF collector: analysis

In this chapter I report on the performance and characteristics of the deferred-older-first (DOF) collector, particularly as it compares to the 2-generational reference collector (2GEN). I also contrast my findings with those reported by Stefanović [27], and I propose a mechanism that I believe will allow the DOF collector to overcome some of the shortcomings reported in this chapter.

Chapter 3 describes the DOF collector implementation. Recall that the names 1DOF, 2DOF, 3DOF, and 4DOF refer to four different configurations of the same DOF collector, where an nDOF collector collects $1/n$ of the dynamic area every time it collects. The plural term "DOF collectors" refers to these four configurations collectively.

Throughout the chapter I make reference to the "canonical settings" of the DOF collectors. These are parameter settings of the collector-specific tuning parameters; the "canonical runs" are benchmark runs that were measured with the canonical settings enabled. The tuning parameters are:

Shadow remembered sets: enable additional remembered sets that reduce the number of pointers interior to the dynamic area that are traced when the ephemeral area is collected.

Whole-heap collector: enable a backup garbage collector that collects the entire heap at once by marking the live objects and sweeping the remembered sets, removing dead objects. Its primary function is to collect inter-generational cyclic garbage structures, but it can also help to reduce the amount of copying due to floating garbage. It runs whenever the collection window is reset to the old end of the heap.

All four DOF collectors enable the shadow remembered sets for their canonical runs, but only 4DOF enables the whole-heap collector for its canonical runs. Those settings were chosen because they result in the lowest average execution time for each collector across a subset of the benchmark suite; see Section 5.2.4.

It turns out that the DOF collectors are only somewhat competitive with 2GEN when space and time are taken into account: Though the heap space consumption of the DOF collectors is comparable to that of 2GEN, and the DOF collectors perform well in isolated cases, they often have higher mark/cons ratios, pay higher costs per unit copied, and have larger remembered sets than 2GEN.

The focus of this chapter will be on explaining why the DOF collectors are not doing well. In summary, floating garbage contributes substantially to the higher mark/cons ratio, especially on some iterated programs; the copy costs relative to 2GEN are inflated by the collector's write barrier and a large amount of remembered set work; and the remembered sets grow quite large both because the programs use large pointer-rich data structures and because extensive use of popular objects in some programs inflate the remembered sets significantly.

I also find that the shadow remembered sets are effective at reducing the amount of remembered set scanning, but that the whole-heap collector is generally expensive and is of use to only two of the programs in the benchmark suite.

7.1 Performance of the DOF collector

The space/time diagrams in Figures 7.1 through 7.4 provide a visual impression of how the DOF collectors compare against 2GEN. The general trend is that 1DOF uses less heap space than 2GEN but has inferior performance; 2DOF, 3DOF, and 4DOF use about as much space as 2GEN, and though they perform better than 1DOF, they do not often perform better than 2GEN.[1]

A quantitative view of the same data is provided by Table 7.1, which shows the results of comparing the DOF collector against 2GEN and 3GEN by both space and time using the method for comparison described in Section 5.3.3. A collector wins a point if it has an advantage on both space and time or if it has an advantage on one but not the other, and its winning advantage is at least two points and at least 1.5 times the winning advantage of the opponent; the total number of points available is the number of comparable runs. The numbers reported in the table are the percentages of wins for one collector over the other.

Table 7.1 shows that on these benchmarks, the DOF collectors are not quite as good as 2GEN but clearly competitive with 3GEN. Even compared against 2GEN, however, the DOF collectors are often competitive: 2DOF and 3DOF are clear winners against 2GEN on the `twobit` benchmarks, 2DOF performs well on `5sboyer`, 3DOF performs as well as 2GEN on `gcbench`, all DOF collectors do very well on `gcold:1,0`, and 3DOF and 4DOF perform well on `perm:200,1`.

Table 7.2 provides yet more views of the data: the percentages of runs, broken down individually by total time, GC time, total space, and heap space, on which the DOF collectors did no worse than 2GEN on comparable runs. The tables show that the DOF collectors perform better on total time than on GC

[1]These figures are based on data that use heap space rather than total space, and this makes a difference for the DOF collector as it usually has rather large remembered sets. See Section 5.3.3 for a brief justification of the use of heap space rather than total space.

	1DOF		2DOF		3DOF		4DOF	
	GEN	DOF	GEN	DOF	GEN	DOF	GEN	DOF
2GEN								
`5earley:12`	62.5	18.8	100.0	0.0	100.0	0.0	100.0	0.0
`gcbench:5`	80.0	20.0	37.5	37.5	57.1	28.6	28.6	57.1
`gcold:1,0`	89.5	0.0	42.1	31.6	0.0	100.0	0.0	100.0
`gcold:1,1000`	47.4	31.6	94.7	0.0	100.0	0.0	100.0	0.0
`gcold:10,1000`	10.5	57.9	10.5	73.7	10.5	78.9	5.3	84.2
`nboyer:3`	76.9	7.7	42.9	7.1	42.9	28.6	57.1	7.1
`5nboyer:3`	53.8	23.1	78.6	7.1	92.9	7.1	92.9	7.1
`perm:200,1`	100.0	0.0	85.7	7.1	7.1	71.4	0.0	71.4
`perm:25,8`	38.5	53.8	82.4	5.9	64.7	17.6	64.7	17.6
`5sboyer:4`	30.0	60.0	61.5	30.8	84.6	7.7	92.3	0.0
`twobitlong`	16.7	77.8	26.3	68.4	52.6	36.8	42.1	57.9
`twobitshort`	10.5	63.2	21.1	68.4	21.1	57.9	36.8	52.6
`5twobitshort`	5.3	78.9	26.3	57.9	73.7	5.3	31.6	52.6
3GEN								
`5earley:12`	50.0	43.8	84.2	5.3	100.0	0.0	89.5	10.5
`gcbench:5`	78.6	0.0	73.3	20.0	91.7	8.3	61.5	30.8
`gcold:1,0`	0.0	100.0	0.0	100.0	0.0	100.0	0.0	100.0
`gcold:1,1000`	0.0	100.0	36.8	42.1	47.4	31.6	31.6	42.1
`gcold:10,1000`	57.9	31.6	36.8	63.2	31.6	63.2	26.3	68.4
`nboyer:3`	53.3	20.0	31.2	43.8	12.5	56.2	31.2	18.8
`5nboyer:3`	14.3	64.3	20.0	66.7	73.3	6.7	80.0	6.7
`perm:200,1`	58.3	16.7	0.0	46.2	0.0	100.0	0.0	100.0
`perm:25,8`	0.0	100.0	26.7	53.3	0.0	93.3	20.0	66.7
`5sboyer:4`	30.0	50.0	53.8	30.8	92.3	0.0	84.6	0.0
`twobitlong`	11.1	55.6	31.6	47.4	68.4	15.8	42.1	47.4
`twobitshort`	0.0	89.5	5.3	84.2	10.5	73.7	21.1	73.7
`5twobitshort`	0.0	94.7	0.0	78.9	31.6	42.1	10.5	73.7

Table 7.1: Table of winning runs as compared by heap space and total CPU time. Each pair of columns shows the results of comparing the DOF collector listed at the very top with the GEN collector listed at the top of each section. An entry in the column "GEN" shows the percentage of runs won by the GEN collector; similarly an entry in the column "DOF" shows the percentage of runs won by the DOF collector. The percentages in the two columns do not add up to 100% because there are runs that are not classifiable.

time, but that is hardly surprising since GC time only accounts for a fraction of total time; the DOF collectors may also preserve preserve locality of reference better than 2GEN, however.

The poor showing for the DOF collectors on GC time is not surprising, given that the DOF collectors usually copy more and have higher cost per volume copied than 2GEN; Sections 7.2 and 7.3 examine those factors.

The DOF collectors compare well on heap space, and that has a positive effect on their overall rankings, but the DOF collectors usually use substantially more space overall than 2GEN, especially as the number of generations increase. The DOF collectors' remembered sets grow very large in some cases, and the back-up whole-heap collector uses additional memory for its mark bitmap when it runs.

The reasons for the large remembered sets are discussed in Section 7.5. It is possible to choose more space-efficient representations for the remembered sets, so the space penalty need not be as considerable as it is in my collector. (A better representation will not reduce the number of elements in the set, though, so scanning times will not necessarily improve.)

The mark bitmap for the whole-heap collector is allocated as a block from the heap every time that collector runs. In situations where memory is particularly tight, the memory allocated to the nursery could be used for the mark bitmap during full collections. If memory is not tight, then the overhead of the bitmap—one bit per two words of address space, or 1.5% on a 32-bit machine—is not important.

7.2 Mark/cons ratio

Table 7.3 shows the average mark/cons ratios for the DOF collectors and 2GEN across all the benchmarks. The general trend is that the DOF collectors have worse mark/cons ratios than 2GEN, though on several programs—`gcbench`, `gcold`, `perm`, iterated `nboyer`, and `twobit`—some DOF collectors are competitive with or better than 2GEN.

The main reason the DOF collectors have higher mark/cons ratios than 2GEN is *excess retention*, often called "floating garbage" or simply "float". Float is in turn caused by several factors, including a predominance of young-to-old pointers on several of the benchmarks. The phenomenon is examined in Section 7.4.

The graphs that plot the mark/cons ratio as a function of effective L show that the mark/cons ratio does decrease as expected as L increases (see Figures B.4 through B.7).

7.3 Copying cost

Every pointer copied by the DOF collectors must be examined by the write barrier, and some objects that are found to contain intergenerational pointers

	Total time				GC time			
	1DOF	2DOF	3DOF	4DOF	1DOF	2DOF	3DOF	4DOF
`5earley:12`	12	0	0	0	12	0	0	0
`gcbench:5`	0	69	79	93	0	69	79	93
`gcold:1,0`	0	47	100	100	0	47	84	100
`gcold:1,1000`	0	11	11	11	0	11	11	11
`gcold:10,1000`	26	47	53	53	0	37	42	42
`nboyer:3`	0	14	7	7	0	14	14	7
`5nboyer:3`	0	7	7	7	0	7	7	7
`perm:200,1`	0	14	86	50	0	21	79	57
`perm:25,8`	15	18	29	6	23	18	29	6
`5sboyer:4`	40	38	15	0	10	8	0	0
`twobitlong`	67	26	37	42	6	11	0	0
`twobitshort`	21	26	37	32	0	26	26	11
`5twobitshort`	21	47	26	47	0	0	0	0
	Total space				Heap space			
`5earley:12`	75	11	0	11	75	0	0	5
`gcbench:5`	87	38	0	36	67	31	0	14
`gcold:1,0`	100	58	79	74	100	58	79	74
`gcold:1,1000`	100	32	21	26	100	32	21	26
`gcold:10,1000`	89	84	89	95	89	84	89	95
`nboyer:3`	100	79	79	71	100	79	50	50
`5nboyer:3`	100	71	43	50	100	36	0	0
`perm:200,1`	100	29	57	79	100	29	57	79
`perm:25,8`	92	41	53	59	92	41	53	53
`5sboyer:4`	70	38	15	31	70	23	0	0
`twobitlong`	89	74	47	58	89	68	11	37
`twobitshort`	89	79	79	63	89	63	47	32
`5twobitshort`	89	79	32	68	95	47	0	26

Table 7.2: Percentage of DOF runs that were no worse than comparable 2GEN runs.

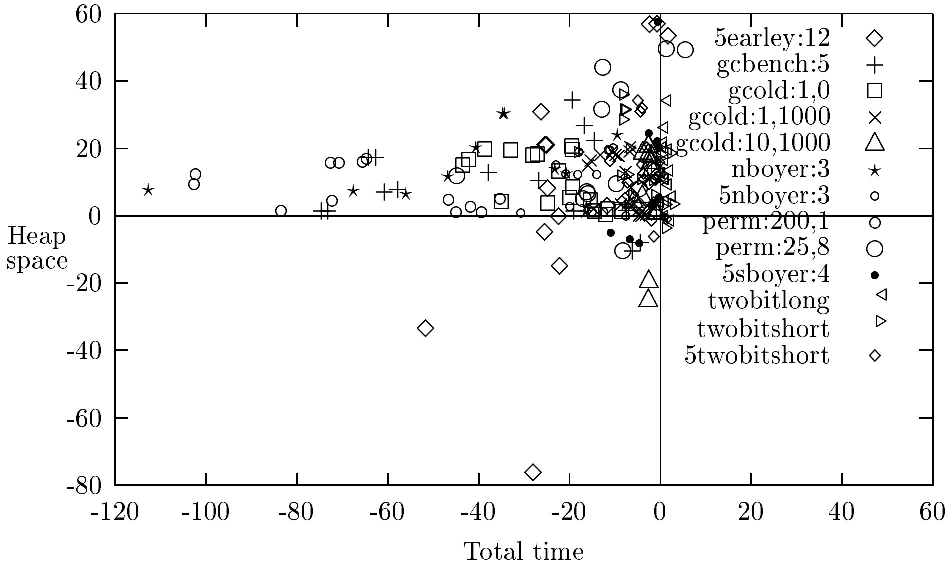

Figure 7.1: Space/time diagram comparing 2GEN (advantage along negative axes) and 1DOF (advantage along positive axes).

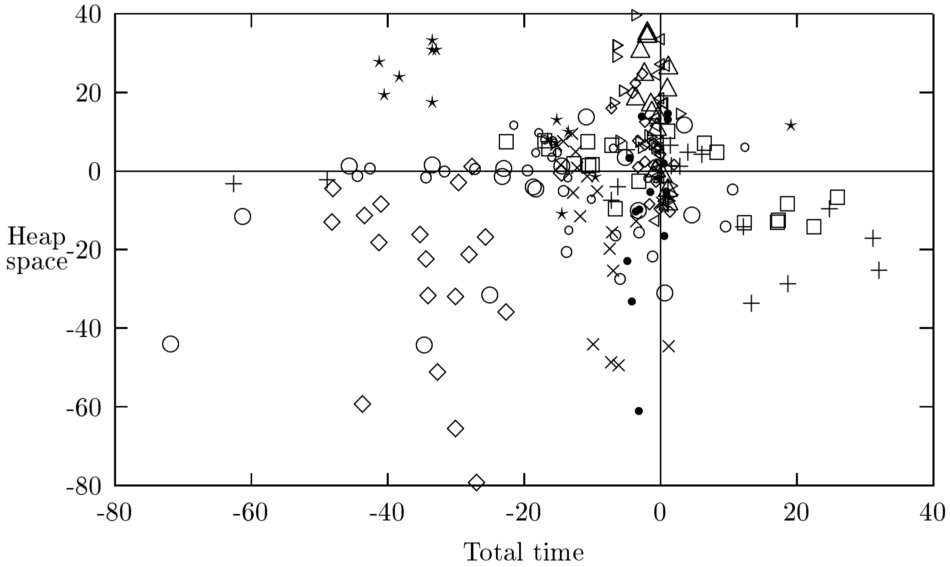

Figure 7.2: Space/time diagram comparing 2GEN (advantage along negative axes) and 2DOF (advantage along positive axes).

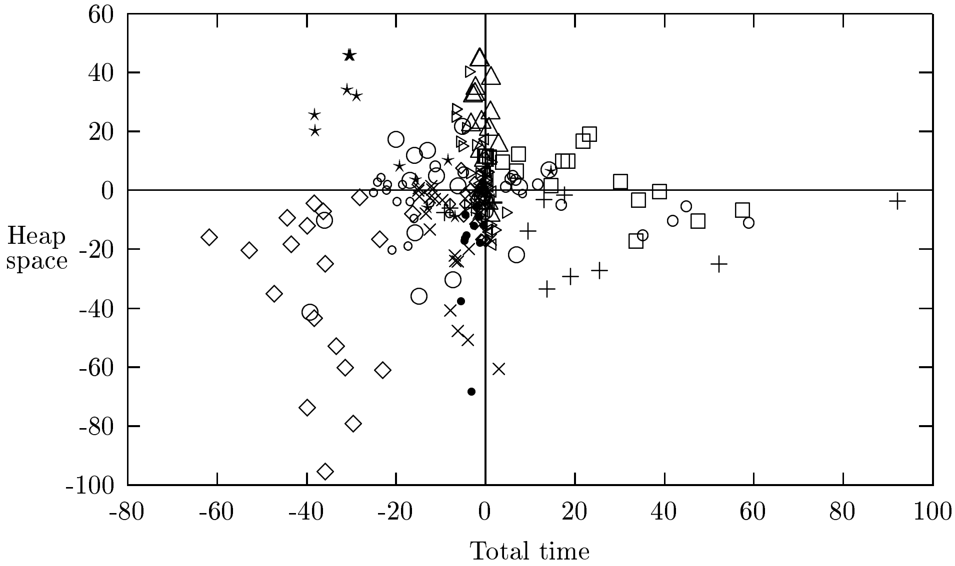

Figure 7.3: Space/time diagram comparing 2GEN (advantage along negative axes) and 3DOF (advantage along positive axes).

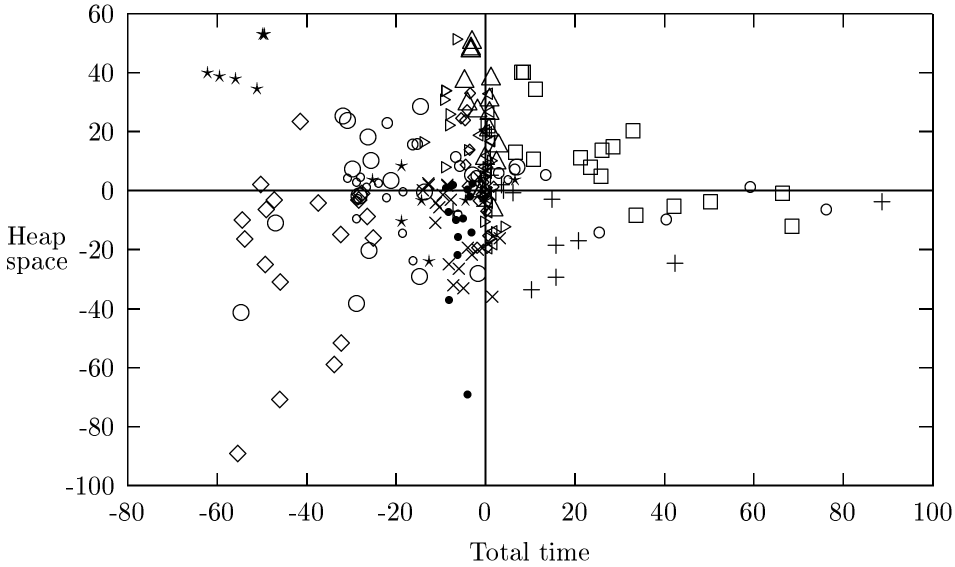

Figure 7.4: Space/time diagram comparing 2GEN (advantage along negative axes) and 4DOF (advantage along positive axes).

	2GEN	1DOF	2DOF	3DOF	4DOF
`5earley:12`	0.34	0.47	0.57	0.57	0.62
`gcbench:5`	0.69	0.94	0.54	0.44	0.44
`gcold:1,0`	1.02	1.32	0.89	0.68	0.66
`gcold:1,1000`	0.90	1.14	1.10	1.08	1.12
`gcold:10,1000`	0.14	0.21	0.15	0.13	0.16
`nboyer:3`	0.87	1.47	1.05	0.99	1.26
`5nboyer:3`	0.87	1.06	0.88	0.89	0.99
`perm:200,1`	2.76	4.74	3.03	2.08	2.11
`perm:25,8`	1.74	1.61	1.80	1.56	1.86
`5sboyer:4`	0.46	0.64	0.55	0.59	0.78
`twobitlong`	0.12	0.18	0.13	0.14	0.15
`twobitshort`	0.32	0.57	0.39	0.38	0.50
`5twobitshort`	0.31	0.41	0.32	0.34	0.37

Table 7.3: Average mark/cons ratio across all runs, broken down by benchmark and collector. Data for the DOF collectors include work performed by the whole-heap collector.

must be inserted into the remembered set so that the pointers can be tracked during later collections; the GEN collectors incur neither cost. Therefore, the DOF collectors ought to have higher copying and scanning costs than the GEN collectors, and Table 7.4 shows that that is the case, exemplified by the cost of performing copying promotions from the nursery into the dynamic area. The cost of promotion is also influenced by the amount of remembered set scanning performed during promotion. However, as the DOF collector's shadow remembered sets ensure that the amount of scanning during promotion is lower than in 2GEN, the figures in Table 7.4 at worst underestimate the cost of promotion in the DOF collector.

The added cost varies with the benchmark and is partly correlated with the kinds of data the collector copies. Pairs are common, so the collectors are tuned specially for pairs; however, that tuning makes the impact of the write barrier and remembered set insertions greater. On programs that manipulate only pairs—`earley`, `boyer`, and `perm`—the added cost in the DOF collector is predictably high.

On programs like `gcbench` and `gcold` that manipulate only vector data structures, the increase in copying cost is smaller, as vector copying is not as tightly optimized as pair copying. (In practice, vector copying is not as amenable to optimization.)

Pairs dominate in `twobit` as well, but the cost of promotion on the `twobit` benchmarks is strongly influenced by the cost of scanning the remembered set, which contains two very large objects, both representing the symbol table. Both objects will be scanned on most garbage collections and will drive up promotion cost. The phenomenon is discussed in Section 6.2.2.

	2GEN	4DOF	$\frac{\text{DOF}}{\text{GEN}}$
5earley:12	24.6	35.7	1.45
gcbench:5	23.8	28.4	1.19
gcold:1,0	25.8	30.0	1.16
gcold:1,1000	26.4	30.2	1.15
gcold:10,1000	35.3	39.6	1.12
nboyer:3	24.0	31.2	1.30
5nboyer:3	23.5	32.6	1.39
perm:200,1	19.7	27.7	1.41
perm:25,8	19.8	27.6	1.39
5sboyer:4	26.2	36.1	1.38
twobitlong	78.8	76.9	0.98
twobitshort	45.5	41.6	0.91
5twobitshort	36.3	39.3	1.08

Table 7.4: Promotion cost per volume broken down by collector and benchmark, in milliseconds of GC time per megabyte promoted.

The data in Table 7.4 show the presence of the large objects in two ways. First, the absolute promotion costs per volume are much higher for `twobit` than for the other benchmarks, and since there is nothing unusual about the data structures used by `twobit`, the only reasonable explanation is that the amount of remembered set scanning is much higher.

Second, the unit cost is very high on `twobitlong`, somewhat more moderate on `1twobitshort`, and more moderate still on `5twobitshort`. The decrease corresponds with how the programs use data: `twobitlong` creates new symbols all the time, so its symbol table grows throughout and will therefore be in the remembered set throughout. `1twobitshort` creates symbols in the early phase of the program when the source text is read and converted to its canonical form, but fewer thereafter, as the program is compiled in whole-program optimization mode. `5twobitshort` behaves the same way during the first iteration, but differently during subsequent iterations: symbols will be reused rather than created and the symbol table will be entered into the remembered set only rarely, if at all.

7.4 Float

Float is a direct consequence of the conservative approximation generational collectors take to liveness of data. 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, the latter objects must be copied by the collector. They are called floating garbage, or just float.

In addition to the increased amount of copying, float can also incur higher memory consumption as the volume of survivors of garbage collection increases. On fixed-heap runs, the higher memory consumption may result in more collection failures. On variable-heap runs, the heap will simply be larger than it otherwise would have been. This is not quite a storage leak, as the float will be reclaimed at some point, but since it is generally desirable to keep memory consumption low, the bloat caused by float is not welcome.[2]

I can measure the amount of float in my DOF collectors directly by performing a full collection before every garbage collection; this does not disturb or reclaim any heap data but removes all garbage objects from the remembered set (see Section 3.5). I can also choose to perform full collections only before DOF collections, thus distinguishing between float caused by DOF collection—"DOF float"—and float caused by generational collection in general—"promotion float". Table 7.5 shows that the DOF collectors can incur large amounts of DOF float, placing them at a disadvantage relative to the GEN collectors.

The column labeled "4DOF-NC" is just 4DOF with full collections disabled ("NC" for "non-canonical") and serves as an illustration of how bad float would have been had the 4DOF collector been run with the whole-heap collector disabled.

Although there is no theoretical reason why the promotion float in 3DOF should be different from the promotion float in 4DOF, there may be differences in practice because the different partitioning of the heap in the two collectors will lead to different collection patterns. Thus the volume promoted by 3DOF may be the same as the volume promoted by 4DOF, but some may be counted as float in one collector and not in the other.

In the following sections I examine each of the benchmarks and explain why they cause the amount of float they do. Table 7.7 will illustrate that discussion. I have not examined any variations in the heap size due to float reduction, as I do not believe that I have data at a sufficiently fine resolution to provide a meaningful analysis.

7.4.1 Float in `earley`

`earley` provides an example of the effects of a functionally-built data structure that spans DOF generations and that will cause float because none of the nodes can be collected until the youngest nodes are. The storage profile for `earley` is shown in Figure C.1.

A single iteration of `earley` does not run long enough to trigger even one DOF collection: the nursery filters out most short-lived allocation, and the survivors are promoted into the DOF area. When the iteration is over, the entire structure spans more than one DOF generation, every node is reachable from the youngest node, and the entire structure is dead. At this point, remembered

[2]Certain kinds of float are not reclaimable—circular objects that cross generation boundaries must be reclaimed by a special backup collector. See Section 3.5.

	Percent			
	2DOF	3DOF	4DOF	4DOF-NC
`5earley:12`	99.1	127.1	98.7	161.1
`gcbench:5`	1.2	3.5	6.0	6.0
`gcold:1,0`	12.8	9.4	7.1	7.1
`gcold:1,1000`	73.4	104.0	80.1	127.3
`gcold:10,1000`	9.2	13.3	1.9	1.9
`nboyer:3`	24.1	35.5	39.3	49.1
`5nboyer:3`	27.8	45.1	36.3	61.3
`perm:200,1`	21.2	56.2	62.9	73.7
`perm:25,8`	13.5	32.1	41.5	42.3
`5sboyer:4`	35.1	95.0	72.3	121.3
`twobitlong`	26.1	33.0	25.3	38.7
`twobitshort`	16.5	18.7	21.4	23.2
`5twobitshort`	30.2	54.0	34.1	74.9
	Volume (megabytes)			
`5earley:12`	75.1	80.3	62.4	101.8
`gcbench:5`	7.0	19.5	32.6	32.6
`gcold:1,0`	30.9	16.6	10.9	10.9
`gcold:1,1000`	182.7	219.7	154.2	245.2
`gcold:10,1000`	2.1	2.4	0.3	0.3
`nboyer:3`	15.1	19.9	21.0	26.2
`5nboyer:3`	89.8	132.3	102.7	173.3
`perm:200,1`	131.1	171.2	148.2	173.7
`perm:25,8`	46.5	79.2	95.1	96.9
`5sboyer:4`	32.1	66.8	57.6	96.7
`twobitlong`	15.0	18.1	13.8	21.1
`twobitshort`	4.6	4.9	5.5	6.0
`5twobitshort`	38.9	64.2	39.9	87.6

Table 7.5: Average DOF float, shown as a percentage of total copying and as the volume copied, for the canonical runs of three of the DOF collectors and 4DOF without full collections (4DOF-NC). 1DOF has no DOF float, by design.

	Percent			
	1DOF	2DOF	3DOF	4DOF
`5earley:12`	0.0	0.0	0.0	0.0
`gcbench:5`	56.0	40.5	39.7	39.1
`gcold:1,0`	0.0	0.0	0.0	0.0
`gcold:1,1000`	0.0	0.0	0.0	0.0
`gcold:10,1000`				
`nboyer:3`	0.0	0.0	0.0	0.0
`5nboyer:3`	0.0	0.0	0.0	0.0
`perm:200,1`	0.0	0.0	0.0	0.0
`perm:25,8`	0.0	0.0	0.0	0.0
`5sboyer:4`	0.0	0.0	0.0	0.0
`twobitlong`	17.2	18.3	18.7	18.7
`twobitshort`	10.8	11.0	12.0	12.3
`5twobitshort`	13.5	11.8	13.8	15.5
	Volume (megabytes)			
`5earley:12`	0.0	0.0	0.0	0.0
`gcbench:5`	380.7	167.9	159.3	155.3
`gcold:1,0`	0.0	0.0	0.0	0.0
`gcold:1,1000`	0.0	0.0	0.0	0.0
`gcold:10,1000`				
`nboyer:3`	0.0	0.0	0.0	0.0
`5nboyer:3`	0.0	0.0	0.0	0.0
`perm:200,1`	0.0	0.0	0.0	0.0
`perm:25,8`	0.0	0.0	0.0	0.0
`5sboyer:4`	0.0	0.0	0.0	0.0
`twobitlong`	12.5	8.9	8.6	8.6
`twobitshort`	4.2	2.8	2.8	2.8
`5twobitshort`	23.6	13.6	14.4	15.7

Table 7.6: Average promotion float, shown as a percentage of total copying and as the volume copied, for the canonical runs of the four DOF collectors. Blank entries were not measured but will be zero.

	Copying	Copying (reduced)	Promotion	DOF	DOF (reduced)	Actual float
`5earley:12`	150.7	63.2	63.2	87.5	≈ 0.0	$\approx \infty$
`gcbench:5`	588.4	524.6	469.6	118.8	54.9	116%
`gcold:1,0`	163.9	153.0	152.0	11.9	1.0	1099%
`gcold:1,1000`	393.9	192.7	152.8	241.1	39.9	505%
`gcold:10,1000`	16.7	16.4	15.3	1.4	1.1	28%
`nboyer:3`	77.3	53.5	44.3	33.0	9.2	259%
`5nboyer:3`	424.9	283.0	220.5	204.5	62.5	227%
`perm:200,1`	397.2	235.7	229.0	168.2	6.7	2422%
`perm:25,8`	327.0	229.2	229.0	98.0	0.2	58421%
`5sboyer:4`	157.1	79.7	66.7	90.4	13.1	592%
`twobitlong`	69.5	52.7	49.8	19.7	2.8	595%
`twobitshort`	29.1	23.6	20.6	8.6	3.0	184%
`5twobitshort`	176.3	112.4	94.2	82.1	18.2	351%

Table 7.7: The table shows various copy volumes for a particular set of runs of 4DOF. All numbers are in megabytes. "Copying" is the total amount of copying done by the canonical run. "Copying (reduced)" is the total amount of copying done by the run with a full collection before every DOF collection. "Promotion" is the amount of copying performed by the promotions from the ephemeral area into the dynamic area; this volume is the same for all DOF collectors, modulo differences in promotion float due to collection timing. "DOF" is the amount of copying performed by the DOF collector only—not including promotions into the dynamic area—on the canonical run. "DOF (reduced)" is the amount of copying performed by the DOF collector only on the run where there is a full collection before every DOF collection. "Actual float" is the amount of float incurred by the DOF collector, not diluted by the copying performed during promotions.

set entries in younger generations keep the dead objects in the oldest generations alive. As the second iteration fills up the DOF area it forces collections in the oldest generations, and all objects in those generation that are reachable from the remembered set are copied, unnecessarily.

7.4.2 Float in `gcbench`

The amount of DOF float is moderate in `gcbench`, because the DOF collector creates few opportunities for it to occur: most data structures that are promoted to the dynamic area fit entirely within a single DOF generation, even in 4DOF. Only at the beginning of an iteration, when a large tree is allocated, and at the end of an iteration, when other large trees are allocated, will some data structures be likely to be promoted into the dynamic area and span multiple generations. During most of the run, the garbage collector in the dynamic area is not even invoked. (The remembered set profile in Figure B.35 shows the incidence of garbage collection in 4DOF.)

The long-lived data may also contribute to DOF float. The long-lived data are referenced from local variables in the benchmark code. In Larceny, stack frames are copied into the heap at garbage collection time and promoted just like other data, so there is a chance that a stack frame that has been promoted into a younger DOF generation will be in the remembered set because it contains pointers to the long-lived data in an older generation. The long-lived data will not be collectible until the stack frame has been collected, so any collection in the older generation will cause the data to be copied unnecessarily.

7.4.3 Float in `gcold`

While the mark/cons ratio for `gcold` decreases in 2GEN as the mutation rate is changed from 0 to 1000, it increases significantly in 2DOF, 3DOF, and 4DOF, and as Table 7.5 shows, this increase in copying can be ascribed to a rapid increase in the amount of float. The most reasonable explanation for the increase in float is that the mutations create many cross-generation pointers among data structures that without the updates would have few such pointers.

7.4.4 Float in `nboyer` and `sboyer`

`nboyer` and `sboyer` have large amounts of DOF float. Both programs build large pointer-rich data structures in a functional manner, thus creating many young-to-old pointers that cross generation boundaries.

`sboyer` is similar to `earley` in that its data structure grows almost monotonically and that the first iteration completes before the DOF area is full (see the storage profile in Figure C.6). Since `sboyer` can complete an entire iteration before a collection in the dynamic area is required, virtually all copying done by the DOF collector is due to float. That most copying is float is apparent in the data from the float reduction runs: if a full collection is run before every DOF collection, the amount of copy work in the dynamic area drops by a factor of seven on the 4DOF run, and by higher factors on 3DOF and 4DOF-NC (see Table 7.5).

The behavior of `nboyer` is more complex: the allocation for an entire iteration will not fit in the DOF area, and the program has a much less regular storage profile than `sboyer`, so a DOF collection, even though it may copy a lot of garbage, also copies live data, and thus the fraction of float is lower than in `sboyer`.

7.4.5 Float in `perm`

The mark/cons results show that the `perm:200,1` benchmark has queue-like lifetimes that suit the DOF collector if the collection window is not too large, but that the `perm:25,8` benchmark benefits from a large collection window. That is in accordance with expectations, as the slow replacement rate of `perm:200,1` is a perfect fit for older-first collection while the high replacement rate of `perm:25,8` almost requires whole-heap collection to run efficiently.

The data in Tables 7.5 and 7.7 back this up further: though the volume of float for `perm:200,1` in the 4DOF collector is higher than the volume of float for `perm:25,8`, the amount of float in the DOF area is higher for the latter program: nearly 60,000%, as almost every object copied by the DOF collector is dead.

7.4.6 Float in `twobit`

`twobit`, like `gcbench`, is susceptible to promotion float because it relies in part on imperative update of data structures, and Table 7.6 shows that there is considerable promotion float in `twobit`.

There is up to twice as much DOF float as promotion float, however.

The relative volume of DOF float is also substantially larger for `5twobitshort` than for `1twobitshort`. Table 7.7 shows that—in this particular case at least—the amount of data copied for `5twobitshort` is about 6 times the amount copied for `1twobitshort`, but with float reduction the amount copied grows by less than a factor of 5. To cause that increase in copying, it suffices that about 5MB of the data from one iteration lingers as float until the next iteration is underway. That figure is high, but not unlikely, and I believe that it accounts for the relative increase in float in the iterated program.

7.4.7 Summary: reasons for float

In a simple younger-first collector like 2GEN, float is caused exclusively by intergenerational pointers from older generations to younger generations. In older-first collectors, however, float is caused by pointers in either direction among generations of the dynamic area as well.

In summary, the important contributors to DOF float are:

Young-to-old pointers in large data structures: these pointers are a feature of functionally constructed data structures and programs that compute with symbolic data or objects with identity, and may be common in programs written in many modern programming languages.[3] They cause float when the entire data structure crosses generation boundaries but those parts that are in the collected region are dead.

Old-to-young pointers in large data structures: these pointers are a feature of imperatively constructed data structures and heap-allocated global variables. They cause float when the data structure crosses generation boundaries, the older part is collected before the structure dies, and the younger part is collected after the structure dies: the younger part of the structure will be copied even though it is dead.

Segmented heaps: smaller segments (generations) increase the probability of any particular pointer being intergenerational. As I have shown, the larger the number of DOF generations, the larger the amount of float. 4DOF is shown in Table 7.5 as having less float than 3DOF, but 4DOF had the benefit of full garbage collections during the reported runs. The column for 4DOF-NC shows a truer picture, a significant increase over 3DOF.

Monotonically growing data structures and iterated computations: these exacerbate float because they create large garbage data structures that cross generation boundaries.

7.5 Remembered sets and popular objects

Remembered sets incur two costs: the space that is required to represent them, and the time it takes to maintain them, including time to insert objects, to scan elements, and to clear the sets [18].

Larceny's remembered set requires space that is proportional to the number of elements in the set. Remembered set processing costs are also proportional to the number of elements in the set. Keeping the set small thus reduces both space and time.

The remembered sets in the DOF collectors grow to be extremely large on some benchmarks (see Table 7.8). While the remembered set sizes for 2GEN are always well below 2% of the heap size, the remembered sets for some of the DOF runs grow to over 15% of the heap size on the average.[4]

Recall from Section 3.8 that the remembered sets in the DOF collectors are organized to minimize the amount of time spent scanning them at the cost of

[3]The traditional assumption is that young-to-old pointers are more common than old-to-young pointers, but that probably depends on the programming language and programming style.

[4]Again, the remembered set sizes in the table are the allocated size, not the peak size measured in the number of entries in the set. As the DOF collectors use smaller blocks for the remembered sets and allocate them less eagerly (both implementation artifacts), 1DOF is shown as having smaller sets than 2GEN, even though 1DOF actually has more generations than 2GEN. Even so, the imprecision of the recording mechanism does not preclude one from concluding that 2DOF, 3DOF, and 4DOF have very large remembered sets in some cases.

	2GEN	1DOF	2DOF	3DOF	4DOF	4DOF-NC
`5earley:12`	0.27	0.16	2.23	3.20	3.70	4.25
`gcbench:5`	0.27	0.16	0.34	0.43	0.47	1.18
`gcold:1,0`	0.27	0.16	0.52	0.46	0.51	0.51
`gcold:1,1000`	0.27	0.16	0.46	0.67	0.71	0.77
`gcold:10,1000`	0.27	0.16	0.44	0.64	0.70	0.70
`nboyer:3`	0.27	0.16	2.29	3.25	3.69	4.06
`5nboyer:3`	0.27	0.16	2.56	3.89	4.48	4.85
`perm:200,1`	0.27	0.16	0.28	0.37	0.45	0.45
`perm:25,8`	0.27	0.16	0.28	0.36	0.45	0.45
`5sboyer:4`	0.27	0.16	3.06	4.49	5.23	5.44
`twobitlong`	0.29	0.19	0.94	1.53	1.68	1.93
`twobitshort`	0.34	0.23	1.52	2.01	2.23	2.52
`5twobitshort`	0.34	0.23	1.86	2.71	2.68	3.47

Table 7.8: The average peak size of the remembered set, in megabytes, across collectors and benchmarks.

the space used to represent them. That is not the reason the sets are large, however. Instead, large remembered sets are caused by large pointer-rich data structures and by popular (frequently referenced) objects, as I show later in this section.

Table 7.9 shows more detailed data about the remembered set behavior across all the collectors. The number of objects inserted into a remembered set may be smaller than the number of stores intercepted by the write barrier, as was the case for the 3GEN collector (see Section 6.2.2); the amount of work performed by the barrier may therefore be higher than these data suggest. The number of words scanned are the best measure of remembered set work performed by the garbage collector.

Observe that:

- the number of objects scanned is often only slightly higher than the number of objects inserted, indicating that most objects are scanned only once for each insertion into the set;
- the number of words scanned is often only twice the number of objects scanned, indicating that most programs are rich in pairs;
- runs of 2GEN and 1DOF that have few remembered set insertions still have many scanned objects, namely objects in the static area that reference objects in the collected heap;
- 4DOF performs less remembered set work than 4DOF-NC, which did not have the benefit of full garbage collection to clear dead objects from the remembered set; and

- the amount of remembered set work is always larger for the DOF collectors than for 2GEN, since there are more intergenerational pointers that must be tracked.

The following sections discuss each program and pinpoint some reasons for the large remembered sets in each case. The remembered set traces in Figures B.34 through B.43 in the Appendix aid this discussion. They show the sizes of the remembered sets—the number of entries live—as a function of time of a set of runs of 4DOF.

7.5.1 Remembered sets in `earley`

Figure B.34 shows the remembered set profile for `earley`. For the first little while, the size is zero, then the sets grow rapidly before growth stops completely. Then a collection occurs, and the remembered set size drops by 90%, before rapid grows starts again and the behavior repeats.

What happens on `earley` is that the data structure constructed by the program, a list of trees, uses sharing extensively. The number of references to the shared substructures have an exponential distribution, and the most popular object has about as many references to it as there are trees in the list. `earley` constructs the list and the trees without side effect, so the most referenced objects are also among the oldest. Since the DOF collector's remembered set tracks young-to-old pointers, and the popular objects are in the oldest generation, most objects in younger generations will contain cross-generational pointers that must be recorded in the remembered set.

Initially, there is zero growth in the remembered set because data are promoted into the oldest DOF generation. When promotion continues into the next-oldest generation, the remembered set must grow rapidly as a large number of references to objects in the oldest generation are created. Then the iteration is over, and the next iteration starts (growth is now zero again). While the second iteration is in progress, the heap fills up, and a DOF collection collects the oldest generation. Because of float, all the objects are copied, but all objects that pointed into the collected area can be removed from the remembered set, and the size of the set drops.

7.5.2 Remembered sets in `gcbench`

Figure B.35 shows that the remembered set of `gcbench` is generally quite small; Table 7.9 shows that a large number of words are scanned.

There are only two classes of objects on `gcbench`, a single 500,000-word array and many 4-word arrays, and the remembered set scanning cost is completely dominated by the cost of scanning the large arrays. In a sample run of 2GEN the five large arrays were scanned a total of 38 times, a number that is very close to what one would expect based on the behavior of the collector and the allocation volume. In a comparable run in the DOF collector, the large arrays were scanned 61 times. Since the long-lived array contains pointers to heap-allocated floating-point values (all allocated after the array was allocated) and there is not enough

	2GEN	1DOF	2DOF	3DOF	4DOF	4DOF-NC
	Remembered set insertions					
`5earley:12`	7.7	9.0	3.2e6	5.1e6	5.1e6	7.1e6
`gcbench:5`	8.1e3	8.7e3	423.1e3	690.9e3	952.4e3	928.3e3
`gcold:1,0`	2.2e3	2.2e3	334.3e3	265.1e3	235.8e3	235.6e3
`gcold:1,1000`	5.8e3	5.9e3	598.5e3	1.0e6	1.1e6	1.5e6
`gcold:10,1000`	1.6e3	1.6e3	128.8e3	208.3e3	263.2e3	270.8e3
`nboyer:3`	64.0	64.0	2.5e6	3.0e6	3.3e6	3.8e6
`5nboyer:3`	325.0	325.0	9.1e6	12.7e6	13.8e6	16.4e6
`perm:200,1`	604.1	659.0	536.4e3	626.9e3	685.0e3	762.6e3
`perm:25,8`	621.6	659.0	345.8e3	538.0e3	773.6e3	749.3e3
`5sboyer:4`	91.8	92.0	4.6e6	7.9e6	9.9e6	11.1e6
`twobitlong`	176.8e3	181.1e3	1.0e6	1.8e6	2.0e6	2.5e6
`twobitshort`	73.2e3	75.6e3	859.1e3	1.2e6	1.5e6	1.6e6
`5twobitshort`	367.9e3	381.0e3	3.8e6	5.7e6	5.9e6	7.9e6
	Remembered set objects scanned					
`5earley:12`	3.2e3	3.5e3	3.0e6	4.4e6	6.0e6	6.2e6
`gcbench:5`	32.7e3	35.5e3	138.9e3	183.0e3	206.9e3	233.8e3
`gcold:1,0`	5.7e3	5.9e3	211.0e3	130.9e3	83.4e3	83.4e3
`gcold:1,1000`	9.8e3	10.1e3	364.7e3	601.1e3	535.7e3	747.9e3
`gcold:10,1000`	4.2e3	4.3e3	75.7e3	122.3e3	142.8e3	139.6e3
`nboyer:3`	1.9e3	2.3e3	2.2e6	2.5e6	4.1e6	3.2e6
`5nboyer:3`	9.5e3	10.4e3	8.6e6	11.8e6	16.7e6	15.0e6
`perm:200,1`	2.8e3	3.7e3	10.3e3	9.0e3	9.9e3	9.4e3
`perm:25,8`	2.9e3	3.3e3	6.3e3	7.5e3	10.2e3	8.3e3
`5sboyer:4`	4.9e3	5.4e3	4.3e6	6.9e6	11.9e6	9.8e6
`twobitlong`	190.1e3	194.9e3	829.0e3	1.4e6	1.7e6	1.7e6
`twobitshort`	75.0e3	78.1e3	678.4e3	878.2e3	1.4e6	1.2e6
`5twobitshort`	378.8e3	393.2e3	3.3e6	4.8e6	5.5e6	6.5e6
	Remembered set words scanned					
`5earley:12`	6.5e3	7.1e3	6.1e6	9.2e6	6.7e6	12.7e6
`gcbench:5`	19.6e6	21.1e6	26.0e6	28.5e6	30.4e6	30.4e6
`gcold:1,0`	30.9e3	31.8e3	654.7e3	416.4e3	275.3e3	275.4e3
`gcold:1,1000`	42.0e3	42.8e3	1.1e6	1.8e6	1.5e6	2.3e6
`gcold:10,1000`	9.8e3	9.9e3	248.1e3	385.8e3	415.8e3	433.1e3
`nboyer:3`	3.8e3	4.6e3	4.3e6	5.0e6	4.2e6	6.4e6
`5nboyer:3`	18.9e3	20.7e3	17.3e6	23.6e6	17.6e6	30.1e6
`perm:200,1`	7.0e3	9.1e3	23.3e3	20.7e3	18.2e3	21.7e3
`perm:25,8`	7.1e3	8.1e3	15.1e3	18.0e3	19.2e3	19.9e3
`5sboyer:4`	9.9e3	10.8e3	8.5e6	13.9e6	13.2e6	19.6e6
`twobitlong`	61.9e6	62.9e6	66.0e6	68.6e6	61.2e6	70.2e6
`twobitshort`	5.3e6	5.7e6	7.7e6	8.7e6	7.3e6	9.6e6
`5twobitshort`	15.4e6	16.2e6	23.3e6	27.2e6	13.0e6	31.5e6

Table 7.9: Average amount of 2GEN and DOF remembered set work

room for the long-lived tree, the array, and the values referenced from the array in a single DOF generation, the array must be scanned first when its generation is collected, and then again when the generation containing the floating-point values is collected.

7.5.3 Remembered sets in `gcold` and `perm`

The only remarkable feature of the remembered sets for these programs is that they are smaller than for most of the other benchmark programs; there are no popular objects to inflate the sets, and though there is sometimes substantial float in these programs, a small number of pointers cause that float.

7.5.4 Remembered sets in `boyer`

Figures B.38 and B.39 show the remembered set profiles for `nboyer` and `sboyer`. The profiles differ a great deal: the remembered set for `nboyer` often drops to near zero size, but the remembered set for `sboyer` stays large throughout. Part of the reason for the difference is that `sboyer` incurs few DOF collections, so when a collection does occur—and the remembered set is cleared—another iteration has started, and the remembered set has many legitimate entries already. `nboyer` incurs many more collections, and has more opportunities to clear the remembered set.

`boyer` has a problem with *popular objects*: objects that are referenced from many other objects, causing remembered set growth. Many of the term structures created by `boyer` contains references to shared symbol nodes, and those nodes are among the oldest objects in the system. Thus many terms will be added to the remembered set because they reference symbol nodes in the oldest generation.

I created two programs that I call `pnboyer` and `psboyer`: these are like `nboyer` and `sboyer` except that where objects in `nboyer` and `sboyer` contain pointers to symbol nodes, the objects in `pnboyer` and `psboyer` contain integer indices into a symbol node table. Thus no object is added to a remembered set simply because it references a symbol node. Table 7.10 shows the dramatic reduction in remembered set insertions for the `boyer` programs.

	2DOF	3DOF	4DOF
`nboyer:3`	92	89	85
`5nboyer:3`	95	93	92
`5sboyer:4`	30	31	31

Table 7.10: Reduction in percent in number of references added to the remembered set when symbol nodes are replaced by symbol references.

The reason the reduction is greater for `nboyer` than for `sboyer` is this: Most references in `nboyer` are between objects that have similar age, so the vast majority of remembered set entries result from references to popular symbol objects. `sboyer` has a more complex graph structure, with more cross-generation references to term nodes, resulting in more remembered-set entries that are not caused by references to symbol objects.

7.5.5 Remembered sets in `twobit`

`twobit` is slightly more interesting than the other programs, in that its remembered set sizes are influenced by popular objects, but its remembered set work is dominated by the cost of scanning a large table.

`twobitshort` creates many symbols to represent both the input and temporary names in the intermediate representation. These symbols will be of roughly the same age as the objects that reference them on the first iteration, but on subsequent iterations they will be among the oldest objects in the dynamic area because the symbol table is never garbage collected. During subsequent iterations many references to the symbol objects will cause the referrers to be added to the remembered set.

The remembered set trace in Figure B.43 exhibits two other phenomena. First, in some cases the remembered sets grow extremely rapidly at the time of a garbage collection. The explanation is that a generation with many pointers into younger generations was collected, and those objects must be added to the remembered sets to allow the younger generation to be collected later. Second, the remembered set grows rapidly also when the dynamic area is not being collected, indicating a large number of cross-generational young-to-old pointers.[5]

Nevertheless, the time spent scanning remembered sets is probably dominated by the time spent scanning the symbol table. `twobit` creates a large (40,000 to 70,000 elements) symbol table, and because that table is frequently updated, it will be scanned during most garbage collections and dominates all other objects in the scanning statistics. Its dominance is evident in that 2GEN and 4DOF scan about the same number of words from the remembered set, yet 2GEN can have no problem with popular objects and will have a much smaller remembered set. (See Section 6.2.2 for further details about the symbol table in `twobit`.)

7.5.6 Summary: reasons for large remembered sets

There are several reasons why remembered sets can grow very large in an older-first collector. As it happens, some of these are the same as the reasons for float. Float does not require large remembered sets, though, and a large remembered set does not mean there's float, so the general types of program behavior that cause both can have either effect, depending on the details.

In summary, the important contributors to large remembered sets are:

[5]The growth seen in the remembered set trace cannot be caused by pointers from the dynamic area into the ephemeral area, as those remembered set entries are not recorded in the data used to create the graph.

Young-to-old pointers: Functional programs create data structures that contain many young-to-old pointers; when these structures are large, they may reside in multiple generations and cause the remembered set to grow. Though pointers usually point to targets that are close to the source, and therefore do not cause remembered sets to grow, programs like `sboyer` and `earley` that reduce allocation by using shared structure create data structures where pointer distances are substantially larger, giving rise to more remembered set entries.

Old-to-young pointers: Pointers from old objects to young objects are added to the remembered set when the older generation is collected, causing the remembered set to grow until the young generation is collected also. If there are many "long" old-to-young pointers then growth may be substantial.

Segmented heaps: A larger number of generations makes it more likely that any given pointer is intergenerational, and more intergenerational pointers will result in a larger remembered set.

Popular objects: Some programming styles encourage the use of objects with identity. Some of those objects, often a small number, may be both heavily referenced and old, so many references to them will cause the referrers to be added to the remembered set.

7.6 Efficacy of the tuning parameters

7.6.1 Shadow remembered sets

The shadow remembered sets reduce the cost of promotions in the presence of many intergenerational pointers created by assignments and infrequent collections in the dynamic area.

The canonical runs for all four DOF collectors use the shadow remembered sets (see Section 3.8). The shadow sets make a real difference in some isolated cases and are rarely damaging when they do not help. Table 7.11 shows that the shadow sets sometimes cause great reductions in the number of words scanned on `gcbench`, `gcold`, `perm`, and `twobit`. A separate analysis of the impact of the shadow sets on GC time, not included here, shows that the negative impact never exceeds 3%, but that the positive impact can sometimes add up to an advantage of over 20%.

On `gcbench` the shadow sets reduce the number of times the large long-lived array is scanned. The array contains pointers to heap-allocated data in a younger DOF generation; though interior to the dynamic area, the pointers are tracked by the special DOF sets only following a DOF collection. DOF collections are infrequent in this system, particularly following the creation of the long-lived data, so the large array will be scanned on every promotion unless the shadow sets are used. (The remembered-set trace in Figure B.35 shows the incidence of DOF collections over time in this benchmark.)

	2DOF	3DOF	4DOF
`5earley:12`	0.00	0.00	0.00
`gcbench:5`	439.88	131.71	0.00
`gcold:1,0`	0.00	0.00	0.00
`gcold:1,1000`	0.00	0.00	48.63
`gcold:10,1000`	0.00	0.00	2.32
`nboyer:3`	0.00	0.00	0.00
`5nboyer:3`	0.00	0.00	0.00
`perm:200,1`	0.00	0.00	15.37
`perm:25,8`	0.00	0.00	3.89
`5sboyer:4`	0.00	0.00	0.00
`twobitlong`	31.37	31.05	40.32
`twobitshort`	6.84	6.84	9.37
`5twobitshort`	9.16	11.16	19.89

Table 7.11: Average advantage for shadow remembered sets as measured by the amount of scanning work, with full garbage collection enabled for 4DOF only.

On `gcold` and probably on `perm` the shadow sets reduce the impact from intergenerational pointers created by assignments, which are frequent in these programs. The remembered sets are large in these programs and most entries are caused by assignments.

On `twobit` the shadow sets reduce the number of times the symbol table is scanned—the benefit is particularly pronounced on `twobitlong`, which creates new symbols throughout the run of the program.

7.6.2 Whole-heap collector

The canonical runs for the DOF collectors enable the whole-heap garbage collector only for 4DOF. In earlier sections I have noted that full garbage collections tend to decrease float and remembered set scanning because they remove dead objects from the remembered set. They also reduce the size of the remembered set, as seen in Table 7.8. In this section I discuss the performance implications of whole-heap collection and whether more frequent full collections could be used to reduce float and remembered set sizes.

Table 7.12 shows that only rarely does the whole-heap collector, run once at every window reset, provide any performance benefits. It is never an advantage on 1DOF (nor should it be), and except for `earley` where full collection consistently improves GC times, and `5twobitshort` on 4DOF where it provides a 15% advantage, benefits are usually negative and only slight when they are positive. The whole-heap collector should probably only be used on a case-by-case basis, or it should be invoked in situations when there is substantial benefit to doing so.

`earley` is helped by full garbage collections because its enormous remembered sets, filled with largely dead objects, are effectively cleared by the full collector at low cost. The remembered set profile (Figure B.34) shows that each full collection effectively reduces the size of the remembered set.

`gcold` is helped by full collections because it reduces float, as Table 7.5 shows.

Though `sboyer` has a storage profile that is similar to that of `earley`, it is aided not at all by full collections. Its remembered set profile (Figure B.39) suggests that an occasional full collection is not effective at reducing the remembered set size.

`5twobitshort` is probably helped by full collections on 3DOF and 4DOF for the same reason that `earley` is helped: there is a great deal of float on `5twobitshort`, and occasional full collections will help reduce that.

Separate analyses of the amount of copying performed by the DOF collector and the marking performed by the whole-heap collector, not included here, show that full collections do reduce copying across the benchmark suite in 2DOF, 3DOF, and 4DOF, but that the sum of copying and marking rarely improves on the amount of copying performed by the DOF collector without help from the whole-heap collector. (In those cases where there is an improvement, those improvements are mirrored by positive entries in Table 7.12.)

Overall, whole-heap collection does not appear to be beneficial except in particular cases, and should probably be disabled by default. That it is enabled by default on the runs in 4DOF is related to the method for choosing the canonical settings: the benefits to `earley` and `5twobitshort` are so large that they outweigh the costs to the other programs.[6]

	1DOF	2DOF	3DOF	4DOF
`5earley:12`	-39.62	0.32	9.58	10.11
`gcbench:5`	-24.80	-5.00	-0.86	-1.50
`gcold:1,0`	-35.26	-13.68	-9.58	-6.11
`gcold:1,1000`	-35.26	-11.26	1.26	8.63
`gcold:10,1000`	-34.21	-13.68	-12.11	-11.79
`nboyer:3`	-58.27	-32.11	-23.79	-11.68
`5nboyer:3`	-53.87	-22.95	-11.26	1.68
`perm:200,1`	-73.33	-36.74	-25.89	-14.84
`perm:25,8`	-38.93	-19.47	-16.53	-17.26
`5sboyer:4`	-48.80	-24.75	-11.00	-2.53
`twobitlong`	-21.00	-4.21	3.16	4.11
`twobitshort`	-48.95	-20.53	-11.47	-10.21
`5twobitshort`	-39.89	-8.84	6.63	16.84

Table 7.12: Average advantage for full garbage collection as measured by GC time, with shadow remembered sets enabled on all runs. A positive entry denotes an advantage for full garbage collection on that collector and benchmark.

[6]`gcold` and `perm` were added to the benchmark suite after this evaluation was performed.

7.7 Comparison with Stefanović's work

It is instructive to compare my findings with those of Stefanović [27, 28], but I do not want to compare our collectors and results very closely. First, his DOF collector did not have a nursery, and the effect of the nursery is still an unresolved question. It would be wrong to draw any conclusions about his results based on mine. Second, though my benchmark programs are different from his and though it might be fruitful to contrast our results because of that, I have not actually tried to establish *how* they are different, save that where his are written in object-oriented languages, have fairly small amounts of live data, and are—at least in some cases—based on "real" programs, mine are written in a functional language, have rather larger data structures, and are in many cases difficult to garbage collect efficiently.

In fact, because our benchmarks are so different, I will not try to compare results from "similar" programs to see if our collectors behave similarly. I will instead generally contrast my results with those reported in Stefanović's Ph.D. thesis [27], without getting too specific.

Mark/cons ratio: The most encouraging result in Stefanović's thesis was that his DOF collector had a mark/cons ratio that was sometimes substantially lower than that of a 2-generational collector (what I am calling 2GEN). I have not seen a similar reduction: Table 7.3 shows a lower mark/cons ratio for DOF only on `gcbench` and `gcold:1,0`.

Stefanović's good results apply to only some of his benchmark programs, however. In many cases, the DOF collector did not distinguish itself, and it is probable that my benchmark suite is simply showing the DOF collector in its worst light. These benchmarks were used because they are, in some sense, anomalous.

Note also that `twobit`, which is not a synthetic program, has mark/cons ratios on 2DOF that are barely larger than those it has on 2GEN. As `twobit` has substantial promotion float, a nursery-less DOF collector might copy less than does mine.

Write barrier costs: Stefanović put substantial effort into developing a fast write barrier, and that barrier is used by both the user program (the mutator) and the garbage collector. My mutator write barrier is more expensive than his, but it is also invoked much less often: while Stefanović's design requires the use of a write barrier on every initializing store, mine intercepts only explicit pointer stores. (This issue is discussed in Section 3.9.)

In addition, I use a different and faster write barrier in the garbage collector.

Even so, my write barriers are more expensive than those designed by Stefanović, and my DOF collector pays a price for that. As an earlier section showed, promotion in the DOF collectors may be 20%–40% more expensive than promotion in the 2GEN collector. Some of Stefanović's ideas can be incorporated into Larceny's barrier, and in addition other optimizations of my write barrier are possible. I pursued neither option because they did not fit the current design of the system.

My less-than-optimal write barrier makes the DOF collectors look worse than they might be in the best case, but at present their main problems are clearly float and popular objects, not the micro-cost of the write barrier. In addition, even if the collector's barrier were to be improved, it would still have a cost and would still be at a disadvantage against 2GEN, which has no barrier.

Float: My measurements of float correspond well with Stefanović's. His figures show that float rises rapidly as the fraction of the heap collected decreases, in one case reaching 5000%. My float figures should be smaller because of the way I have reported them: I include the copying performed when data are promoted out of the nursery. The column labeled "Actual float" in Table 7.7 shows that DOF float might be several hundred per cent in my collector in common cases if just the amount of copying performed by the DOF collector were to be taken into account, and that DOF float can reach into thousands of percent in some cases.

Remembered set sizes: Stefanović reports that "the space needed for remembered sets is typically below 2%" in his DOF collectors. My results are a stark contrast to his: remembered sets grow larger as the fraction of the heap collected grows smaller and reach multi-megabyte sizes for programs that generally use not much more than 30 megabytes of heap. The case of `sboyer` is particularly grim; on 4DOF, the heap never exceeds 34 megabytes, yet the average remembered set peak size is 5.23 megabytes, over 15% of the heap size.

To explain that discrepancy it is useful to turn to a discussion of pointer distance and pointer direction. A pointer's *distance* in an age-ordered heap is the distance in bytes of data between its source and target. A pointer's *direction* is forward in time if it points from an older object to a younger object, and backward in time otherwise.

Stefanović found that most pointers were "short" and that there was no significant bias in their direction, thereby contradicting a folk theorem in the garbage collection community that says there is significant bias in favor of young-to-old pointers. I have measured neither pointer direction nor distance in my programs, but from inspection of the programs it is clear that there must be significant bias in favor of young-to-old pointers in many programs. In addition, the shared structure used by `earley` and `sboyer` create many "long" pointers. Both of these contribute to larger remembered sets, as demonstrated earlier in this chapter. In fact, it seems likely that only `gcbench` has both short pointers and no directional bias, and `gcbench` has small remembered sets. The other two programs with small remembered sets, `perm` and `gcold`, are also dominated by short pointers.

7.8 Coping with popular objects

This section describes a mechanism for detecting popular objects and reducing the remembered set costs they incur. Unlike the other mechanisms described in this dissertation, the algorithm that is described in this section has not been implemented and tested.

7.8.1 Introduction

One problem with the DOF algorithm is that popular objects—objects that are referenced from many other objects—inflate the remembered set and incur substantial remembered set management costs; similar problems may affect any collector that uses a highly segmented heap and a remembered set that records each intergenerational pointer.

It is possible for the collector to detect popular objects on the fly, and for it to avoid remembering intergenerational pointers to popular objects. The cost of this mechanism is low, and I believe the algorithm is practical.

The algorithm uses reference counting during garbage collection to detect popular objects. Popular objects become non-movable and non-collectible, so references to them need not be tracked by the remembered set. The collector reassesses the popularity of objects periodically, allowing once-popular objects to become movable and collectible again.

So-called conservative or ambiguous-roots collectors also flag objects as non-movable to avoid having to update references to them [2], and the use of the technique to control popular objects was suggested but not explored by Hudson and Moss [19].

7.8.2 The high-level algorithm

The algorithm uses reference counting on two levels to detect and manage the popular objects. When a generation is collected, reference counts that are managed as part of the copying collector's forwarding information track the number of references to copied objects. If a count exceeds some limit, the object in question is popular and is marked by setting the object's popular object mark. That mark is a one-bit reference count that abstracts the actual reference count for the object, and the collector keeps only that one bit past the end of the collection.

A second one-bit reference count tracks the number of intergenerational references to a popular object. The count is set to zero at the beginning of each collection cycle,[7] and is updated by the garbage collector when other generations are collected: any intergenerational reference to a popular object will

[7] A collection cycle is the period from one DOF window reset to the next.

cause the object's intergenerational reference count to be set to 1. If an object's intergenerational reference count is zero at the end of the collection cycle, then the popular object is no longer reachable from outside its own generation.

The popular object mark allows the collector to avoid moving or reclaiming popular objects, while the intergenerational reference count allows the collector to revoke a popular object's status as popular, marking the object as movable and reclaimable again.

On this high level, the algorithm operates at three points: following each garbage collection, and at the beginning and end of every collection cycle.

- At the beginning of the collection cycle, the intergenerational reference counts are all set to zero.

- During a garbage collection, the collector first copies all non-popular reachable object from the collected generation into the destination generation, but does not do anything with the popular objects. Following the collection, however, the collector "moves" popular objects from the collected generation to the destination generation without changing the objects' addresses (see below). The collector then scans the popular objects for pointers to uncopied objects in the collectible generation so that these can be copied as well.

- At the end of the collection cycle, the collector revokes the popular object status of all popular objects whose intergenerational reference counts are zero.

7.8.3 The low-level algorithm

The low-level garbage collection algorithm is responsible for making sure that the reference counts are all updated properly when a single generation is collected. The code that performs this work is known as the scanning and forwarding loop: it examines every reachable word in the collected generation, copies objects from the source generation to the destination generation, and updates all pointers to refer to the copies.

The standard object scanning and forwarding code, included here to allow comparison with the new algorithm, is shown in Figure 7.5 and works as follows. When the copying collector moves an object, it leaves a forwarding pointer in its place. In Larceny, a forwarding pointer is a two-word structure: a distinguished bit pattern in the first word, and a tagged pointer to the forwarded object in the second word. All objects are at least two words long, so there's always room for the forwarding pointer.

The new forwarding code must maintain the reference counts and is therefore more complicated; it is shown in Figure 7.6.

The distinguished bit pattern in the first word of a forwarding pointer can do double duty as a reference counter. Only the low 8 bits, say, need be distinguishable from other values in the same location, so 24 bits are available for a count. The reference count is set to 1 when the forwarding pointer is created,

```
w = *scanptr;
if (isptr(w))
  if (points_into_fromspace(w))
    x = *ptrof(w)
    if (x == FORWARD_PTR)
      n = *(ptrof(w)+1)
    else
      n = copy_obj( w )
      *ptrof(w) = FORWARD_PTR
      *(ptrof(w)+1) = n
    fi
    *scanptr = n
  else
    n = w
  fi
  if (must_be_remembered(obj_being_scanned,n))
    <add obj_being_scanned to the remembered set>
  fi
fi
```

Figure 7.5: Standard forwarding code in the DOF collector

and incremented each time the forwarding pointer is picked up by another reference. If the reference count reaches a set limit, then the object is marked as popular and as referenced from other generations (a conservative approximation that is necessary for correctness).

The forwarding code is careful not to copy popular objects in the collected generation, and it both sets the intergenerational reference count on objects in other generations and avoids inserting an object into the remembered set unless the object references non-popular objects.

7.8.4 Practical matters

The preceding discussion assumes that it is possible to "move" an object without changing its address, and that there is space available per object for the two one-bit reference counts. In practice, the reference counts are probably maintained on a per-page basis and kept with other information about the page in a descriptor table that is separate from the page itself. This organization also allows a page to be moved simply by changing data in the descriptor table. Larceny already uses such a table, as do other collectors [11].

However, moving pages rather than objects introduces another problem: all other objects on a page with a popular object—the "hangers-on"—will be marked as popular even if they are not (or even if they are garbage), and the space they occupy will not be reclaimed until the popular object no longer popular.

Popular objects are rare by nature, so the storage leaked by hangers-on is unlikely to be a significant problem in practice. Furthermore, the hangers-on are indistinguishable to the high-level algorithm from true popular objects, so the algorithm does not need to treat them specially.

The modified algorithm is slightly more expensive than the standard algorithm. The most important costs are those that are incurred for objects that are only referenced once—in practice, few objects are referenced multiple times.

The primary cost of the algorithm is a lookup in the descriptor table per pointer to determine if the object being pointed to is popular; this cost is incurred on both major branches in the forwarding code. There are other, minor costs as well.

The descriptor table lookup probably boils down to a load, a mask, and a conditional branch—not a great cost, but possibly non-negligible in code that is already tuned. For programs whose remembered sets grow substantially because of popular objects, the cost is almost certainly offset by the savings in remembered set work.

```
w = *scanptr;
ispop = FALSE
if (isptr(w))
  if (points_into_fromspace(w))
    x = *ptrof(w)
    if ((x & 255) == FORWARD_PTR)
      if (x < POPULAR_LIMIT)
        *ptrof(w) = x+256
      elif (x == POPULAR_LIMIT)
        *ptrof(w) = x+256
        mark_as_popular( *(ptrof(w)+1) )
        mark_as_referenced( *(ptrof(w)+1) )
        ispop = TRUE
      else
        ispop = TRUE
      fi
      n = *(ptrof(w)+1)
    elif (is_popular(w))
      n = w
      ispop = TRUE
    else
      n = copy_obj( w )
      *ptrof(w) = 256 + FORWARD_PTR
      *(ptrof(w)+1) = n
    fi
    *scanptr = n
  elif (is_popular(w))
    mark_as_referenced(w)
    n = w
    ispop = TRUE
  fi
  if (!ispop && must_be_remembered(obj_being_scanned,n))
    <add obj_being_scanned to the remembered set>
  fi
fi
```

Figure 7.6: Forwarding code to handle popular objects

Chapter 8

The ROF collector: analysis

In this chapter I report the measurements of the renewal-older-first (ROF) collector, which was described in Chapter 4. I compare the ROF collector to two of the reference collectors, the 2-generational and 3-generational collectors (2GEN and 3GEN).

Throughout the chapter I refer to the "canonical" settings for the ROF collector; these are the settings of the tuning parameters that have been chosen for all the runs that are compared against the reference collectors. The settings were chosen to minimize the average total time across a subset of the benchmarks, as explained in Section 5.2.4. The ROF collector has three tuning parameters. These are:

Phase-detection: triggers a reduction of the collector parameter j when growth of the extra remembered set—the set that tracks pointers from the ROF collector's young generation into its old generation—is found to have slowed from above a limit to below the same limit. The assumption is that reducing j at that point will allow garbage data structures that span both ROF generations to be collected at the next collection instead of causing float. I have used 0.1% as the limit that separates fast and slow growth.

Feeling-lucky: increases the size of the ROF-old generation in proportion to the expected amount of live data in ROF-young, thereby reducing the frequency of garbage collections. I use a parameter value of 0.5 to add half the size of ROF-young ($j/2$) to ROF-old following a collection.

Remembered-set-limit: triggers a reduction of j when the extra remembered set is found to have grown past a set limit on its size, as it will probably be more efficient to collect the data already in ROF-young than to scan a large remembered set. I use a fixed limit of 32768 entries in the remembered set.

The canonical runs for the ROF collector enable all three tuning parameters. The efficacy of the tuning parameters is reported in Section 8.6.

8.1 Summary of findings

The ROF collector's mark/cons ratios are frequently slightly better than those of the reference collectors. Only on one program, `gcbench`, does the ROF collector have worse mark/cons ratios, and that program has lifetimes that do not fit the assumptions the ROF collector makes. In general, programs like `gcbench`—programs with a large volume of long-lived storage and some intermediate-size but short-lived data structures—can be expected to cause the ROF collector to perform badly, as they tend to drive up the collection frequency and force the collector to copy the long-lived data more often.

The ROF collector also compares well with 3GEN overall, but since the ROF collector tends to use more memory and has higher cost per volume for promotion into the dynamic area, it does not always compare well against 2GEN. The ROF collector performs impressively on both `gcold` and `perm` however. `gcold` with its random mutations will have lifetimes that have a random component, and `perm` has queue-like lifetimes, both factors that favor older-first collection. Unlike the DOF collectors, the ROF collector performs well on `perm:25,8` because its collection window is large.

The ROF collector has a higher volume of float than the reference collectors, and it requires more space for its remembered sets, but in both cases the increases are moderate.

The tuning parameters are effective at reducing the amount of copying and remembered set scanning work, and should be enabled by default.

8.2 Performance of the ROF collector

The space/time diagrams in Figures 8.1 and 8.2 provide a general impression of how the ROF collector compares to the 2GEN and 3GEN reference collectors. The ROF collector tends to use more space than either, and while it generally has better performance than 3GEN, it performs better than 2GEN mostly on `gcold` and `perm`.

Though there are few circumstances where the ROF collector is spectacularly worse than the reference collectors, its overall performance on `gcbench` is not good, with a number of runs using more than 20% more time than the reference collectors.

The space/time data are summarized in Table 8.1, which shows the results of comparing the ROF collector against 2GEN and 3GEN by both space and time using the method for comparison described in Section 5.3.3. A collector wins a point if it has an advantage on both space and time or if it has an advantage on one but not the other, and its winning advantage is at least two points and at least 1.5 times the winning advantage of the opponent. The total number of points available is the number of comparable runs; the numbers reported in the table are the percentages of wins for one collector over the other.[1]

[1] These tables use heap space rather than total space in their comparison. See Section 5.3.3 for a brief justification of the use of heap space here.

	Winner		Winner	
	2GEN	ROF	3GEN	ROF
`5earley:12`	73.7	21.1	31.6	31.6
`gcbench:5`	93.8	0.0	100.0	0.0
`gcold:1,0`	0.0	100.0	0.0	100.0
`gcold:1,1000`	10.5	68.4	0.0	94.7
`gcold:10,1000`	0.0	73.7	26.3	68.4
`nboyer:3`	81.8	9.1	27.3	54.5
`5nboyer:3`	90.0	0.0	0.0	70.0
`perm:200,1`	0.0	92.9	7.7	92.3
`perm:25,8`	29.4	70.6	0.0	100.0
`5sboyer:4`	41.7	25.0	58.3	25.0
`twobitlong`	72.2	16.7	77.8	5.6
`twobitshort`	78.9	15.8	15.8	47.4
`5twobitshort`	66.7	16.7	27.8	44.4

Table 8.1: Percentage of winning runs by total CPU time and heap space.

Table 8.1 shows that on these benchmarks, the ROF collector does fairly well against 3GEN, but lags significantly behind 2GEN on all programs except `gcold` and `perm`.

Table 8.2 provides more detailed data, showing the number of runs on which the ROF collector is competitive with 2GEN and 3GEN, broken down by space and time. Again, the ROF collector is sometimes competitive against 2GEN on total time, but more rarely on GC time. In addition, ROF is competitive on space for less than half of the benchmarks. The reason for increased space use in the ROF collector relative to 2GEN is probably a combination of the use of the Feeling-lucky parameter, which requires more space for allocation and collection; a larger volume of float, which will cause the collector to overpredict the amount of live storage; and overprediction of the amount of live data in ROF-young.

Table 8.3 shows the average mark/cons ratios across the benchmarks for ROF, 2GEN, and 3GEN. The ROF collector is clearly able to reduce the mark/cons ratio relative to the reference collectors. On the average, it beats 2GEN on nine of thirteen benchmarks, and it is competitive on three more; it also beats 3GEN on twelve of thirteen benchmarks. In only one case—on `gcbench`—is the ROF collector significantly behind, with a mark/cons ratio that is 29% higher than that of 2GEN and 107% higher than that of 3GEN.

Finally, the ROF collector responds to increases in effective L by decreasing the mark/cons ratio, as it should; see Figure B.8.

The ROF collector shows a notable increase in collection frequency relative to 2GEN on `gcbench`. The problem is almost certainly that the lifetimes of data in this program are not at all like those assumed by the ROF collector. At the time of an ROF collection, most of the data in ROF-young will be dead, and a large fraction of the data in ROF-old—namely the long-lived data—will

	ROF vs 2GEN				ROF vs 3GEN			
	Total time	GC time	Heap space	Total space	Total time	GC time	Heap space	Total space
`5earley:12`	47	21	21	21	100	100	26	26
`gcbench:5`	19	19	44	38	7	7	50	36
`gcold:1,0`	100	100	100	100	100	100	100	100
`gcold:1,1000`	100	100	79	74	100	100	68	68
`gcold:10,1000`	89	63	100	95	100	95	37	37
`nboyer:3`	18	9	27	9	91	91	27	9
`5nboyer:3`	50	0	10	0	100	100	40	20
`perm:200,1`	100	100	100	93	92	92	100	100
`perm:25,8`	100	94	65	65	100	100	93	93
`5sboyer:4`	100	17	17	17	100	100	25	25
`twobitlong`	78	0	22	17	100	100	17	6
`twobitshort`	58	37	32	5	100	100	21	21
`5twobitshort`	33	17	17	17	100	100	39	28

Table 8.2: Percentage of ROF runs that were no worse than comparable 2GEN and 3GEN runs.

be live. Therefore the collection will reclaim a smaller amount of storage than a collection in 2GEN, and the frequency of collections must increase.

There are two main reasons why the ROF collector's mark/cons ratio on `gcbench` is poor. First, the ROF collection algorithm drives up the collection frequency on this program relative to a collector that collects the entire dynamic area. Since the large permanent data structures spend almost all of their life in ROF-old, they will be copied more often by the ROF collector than by the reference collectors.

The second reason why the ROF collector has a higher mark/cons ratio on `gcbench` is that the ROF collector has a substantial amount of float (excess retention) on this program.

Float is discussed in Section 8.4, but a simple calculation is enough to show that float and higher collection frequency together account for most of the increased copying in the ROF collector on `gcbench`. The ROF collector copies about 350 megabytes more data than 2GEN on `gcbench`. Of that volume, float accounts for about 150 megabytes (see Table 8.5). Counting only runs where both collectors succeeded, the average number of major collections in 2GEN is 64, but in ROF it is 78. Of the remaining 200 megabytes, the 14 additional garbage collections in the ROF area each copying 13 megabytes of permanent data accounts for 180 megabytes. That leaves only 20 megabytes unaccounted for, less than 6%.

	Mark/cons ratio			Major GCs		
	2GEN	3GEN	ROF	2GEN	3GEN	ROF
`5earley:12`	0.34	0.60	0.32	8.37	7.79	7.95
`gcbench:5`	0.69	0.43	0.89	84.26	31.07	122.44
`gcold:1,0`	1.02	1.61	0.77	10.84	11.05	8.95
`gcold:1,1000`	0.90	1.54	0.73	11.00	11.21	9.53
`gcold:10,1000`	0.14	0.29	0.14	1.16	0.95	1.21
`nboyer:3`	0.87	1.16	0.90	7.00	4.88	7.27
`5nboyer:3`	0.87	1.15	0.90	33.71	24.67	35.40
`perm:200,1`	2.76	3.26	2.50	40.50	36.69	40.56
`perm:25,8`	1.74	2.41	1.36	29.00	27.20	25.89
`5sboyer:4`	0.46	0.77	0.45	10.31	9.69	10.50
`twobitlong`	0.12	0.15	0.11	8.53	3.37	7.89
`twobitshort`	0.32	0.65	0.30	3.89	2.95	3.58
`5twobitshort`	0.31	0.68	0.29	18.79	12.89	17.06

Table 8.3: Average mark/cons ratio and major garbage collections across all runs. The averages hide certain facts: 2GEN and 3GEN sometimes run at lower heap sizes than ROF, and runs at low heap sizes pull up the averages for the mark/cons ratio and number of major collections, making 2GEN and 3GEN look worse relative to ROF than they actually are.

8.3 Copying cost

Though the ROF 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 8.4 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 the remembered set of 2GEN.

The additional cost of promoting into ROF-young but not into ROF-old shows up in Table 8.4 as higher unit cost as L increases. The amount of live data does not change with L, so when j is selected as the mid-point of the space that is free following garbage collection, the fraction of the dynamic area occupied by ROF-young increases with increasing L. Therefore, the fraction of storage promoted into ROF-young grows also, and the average cost per unit promoted increases.

	2GEN	ROF (for varying L)					ROF/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:1,0`	25.8	27.4	27.8	28.8	30.1	28.5	1.10
`gcold:1,1000`	26.4	27.5	28.2	29.5	31.0	29.0	1.10
`gcold: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:200,1`	19.7	22.6	23.8	24.4	24.9	23.9	1.21
`perm:25,8`	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 8.4: Average promotion cost per volume (ms/MB)

A later section will address the amount of work expended on inserting additional objects into the remembered set.

8.4 Float

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, the latter objects must be copied by the collector. They are called floating garbage, or just float.

I measured the amount of float in the ROF collector by using the DOF collector's whole-heap collector to remove dead objects from the remembered set both before every garbage collection and before collections in the dynamic area only (see Section 3.5). 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 8.5 shows *ROF float* (float due to collections in the dynamic area) and *promotion float* (float due to promotions into the dynamic area) across all benchmarks and all the settings for the ROF collector. Table 8.9 shows the data that the percentages in Table 8.5 are derived from: the actual volumes copied.

Overall, ROF float is moderate across the benchmarks—the high value shown for `gcold:10,1000` stems from a very small number of collections recorded for this benchmark—but promotion float is often high: nearly 50% on `gcbench`. That volume corresponds to the volume of promotion float on 2GEN, however.

	Percent		Volume	
	ROF float	Promotion float	ROF float	Promotion float
`5earley:12`	0.0	0.8	0.0	0.6
`gcbench:5`	17.5	48.5	149.5	278.7
`gcold:1,0`	0.0	0.0	0.0	0.0
`gcold:1,1000`	8.2	0.0	19.1	0.0
`gcold:10,1000`	35.7		8.6	
`nboyer:3`	0.0	0.0	0.0	0.0
`5nboyer:3`	0.0		0.0	
`perm:200,1`	0.0	0.0	0.0	0.0
`perm:25,8`	-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 8.5: ROF 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 (see Table 6.4). Blank entries were not measured but will be zero.

8.5 Remembered sets

The ROF collector uses more space for the remembered set than either of the reference collectors, and in most cases the remembered set grows to over 3% of the heap size (see Table 8.6).[2] The reason for the larger remembered set is that the ROF 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 [6].

Table 8.8 shows the amount of remembered set work in the 2GEN, 3GEN, and ROF collectors, as well as for a collector called ROF-NC: the ROF collector with none of the tuning parameters enabled ("NC" stands for "non-canonical").

The number of insertions into the remembered set is from three to 100,000 times higher in the ROF collector than in the reference collectors. Most of the additional insertions are the result of young-to-old pointers between the two generations in the dynamic area and are performed by the garbage collector's write barrier. The insertions contribute to the higher garbage collection cost.

[2]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 ROF has four—so the minimum remembered set size for the ROF 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 ROF collector grow beyond their minimum size more often than the remembered sets for 2GEN and 3GEN.

	2GEN	3GEN	ROF
`5earley:12`	0.27	0.40	0.84
`gcbench:5`	0.27	0.40	0.53
`gcold:1,0`	0.27	0.40	0.53
`gcold:1,1000`	0.27	0.40	0.54
`gcold:10,1000`	0.27	0.40	0.55
`nboyer:3`	0.27	0.40	0.99
`5nboyer:3`	0.27	0.40	1.05
`perm:200,1`	0.27	0.40	0.53
`perm:25,8`	0.27	0.40	0.53
`5sboyer:4`	0.27	0.40	0.91
`twobitlong`	0.29	0.45	0.84
`twobitshort`	0.34	0.49	0.90
`5twobitshort`	0.34	0.53	1.00

Table 8.6: Average peak remembered set size, in megabytes, across all runs for a benchmark.

Though the ROF collector has many more remembered set insertions than the reference collectors, the amount of remembered set scanning work is usually comparable in all collectors. The fact that the ROF collector does not seem to scan many of the objects that were inserted is due to the Remembered-set-limit tuning parameter: many objects are inserted into the extra remembered set during garbage collection, but that set is cleared and j is reduced immediately following the collection because the set has exceeded its limit.

(In fact, garbage collection cost could probably be reduced further if the collector is able to throttle the insertions by effectively reducing j during a garbage collection rather than following the collection. That change would also be effective in reducing the peak size of the remembered set, of course. The throttle would not have to be in the inner loop and should not cost much in practice.)

The columns for ROF-NC show the amount of work that the collector would have performed had the tuning parameters been disabled. Though the number of insertions into the remembered set are only slightly higher for ROF-NC than for ROF, the amount of scanning work is much higher: up to 500 times in some cases. The remembered set profiles in Figures B.44 through B.53 show remembered set growth over time, with and without the tuning parameters.

8.6 Efficacy of the tuning parameters

The tuning parameters are effective at reducing the amount of copying performed by the ROF collector, as shown by Table 8.9; Table 8.8 shows their effect at reducing the amount of remembered set work. Table 8.7 shows that the reduction in copying and remembered set scanning have real effects on both GC time and total time.

	GC time				Total time			
	Phase	Luck	Limit	All	Phase	Luck	Limit	All
`5earley:12`	15.68	26.21	28.53	42.00	4.95	6.21	5.89	11.16
`gcbench:5`	-1.11	13.12	-0.11	11.75	-0.78	8.12	0.00	6.50
`gcold:1,0`	0.00	9.58	0.63	9.47	0.00	4.42	0.00	4.53
`gcold:1,1000`	0.42	11.05	0.32	10.53	0.00	2.00	0.00	2.00
`gcold:10,1000`	3.89	9.05	1.68	15.47	0.11	0.42	0.00	0.63
`nboyer:3`	0.12	8.29	9.47	15.64	1.12	3.43	3.20	6.91
`5nboyer:3`	1.20	13.40	13.43	22.20	1.87	5.20	4.57	9.60
`perm:200,1`	0.00	11.38	0.00	11.75	0.00	8.25	0.00	8.12
`perm:25,8`	0.53	11.89	0.42	12.63	0.00	7.05	0.00	7.26
`5sboyer:4`	12.92	17.00	14.50	32.33	2.46	1.33	0.33	3.33
`twobitlong`	-0.32	3.44	2.11	4.22	0.42	0.33	-0.42	0.33
`twobitshort`	-1.26	7.68	2.42	13.37	0.00	1.05	0.42	1.26
`5twobitshort`	-0.95	10.84	5.05	16.44	-0.42	1.37	0.00	1.00

Table 8.7: Percent reduction in GC time and total time for the tuning parameters. Negative entries denote increases.

Phase-detection is only effective at reducing GC time on `earley` and `sboyer`. The remembered set profiles in Appendix B also show that there are not many opportunities for Phase-detection to work. It is also not clear whether Phase-detection is useful in combination with the other tuning parameters, since on the benchmarks where Phase-detection is effective the other tuning parameters are effective too.

Both Feeling-lucky and Remembered-set-limit are effective at reducing GC time for all the benchmarks, and the savings carry over to savings in total time. The savings are highest on those programs—`earley`, `boyer`—that are most adverse to older-first collection because of the dominance of young-to-old pointers. Only on `twobitlong` are savings in GC time less than 13%. That program is easy to garbage collect, though—its mark/cons ratio is only 0.11—so it's not surprising that the impact of tuning the collector is slight.

Feeling-lucky and Remembered-set-limit are effective at reducing copying (see Table 8.9), to different extents. Feeling-lucky has a positive effect on all the programs; the Remembered-set-limit is naturally less effective on `gcbench` and `twobit` with their smaller remembered sets.

The adverse effects of the tuning parameters on these benchmarks is limited to more frequent collector failure due to memory exhaustion on fixed-heap runs.

	2GEN	3GEN	ROF	ROF-NC
	Remembered set insertions			
5earley:12	7.7	9.0	662.9e3	938.1e3
gcbench:5	8.1e3	1.5e3	9.0e3	9.7e3
gcold:1,0	2.2e3	1.1e3	3.6e3	3.9e3
gcold:1,1000	5.8e3	10.6e3	21.9e3	25.6e3
gcold:10,1000	1.6e3	4.9e3	7.3e3	12.5e3
nboyer:3	64.0	64.0	634.6e3	875.3e3
5nboyer:3	325.0	325.0	3.3e6	4.8e6
perm:200,1	604.1	472.8	1.7e3	2.0e3
perm:25,8	621.6	473.0	1.6e3	1.7e3
5sboyer:4	91.8	92.0	1.1e6	1.8e6
twobitlong	176.8e3	149.4e3	495.0e3	519.9e3
twobitshort	73.2e3	68.1e3	269.4e3	314.1e3
5twobitshort	367.9e3	327.9e3	1.3e6	1.7e6
	Remembered set objects scanned			
5earley:12	3.2e3	3.3e3	3.3e3	872.9e3
gcbench:5	32.7e3	31.5e3	26.6e3	27.2e3
gcold:1,0	5.7e3	6.5e3	7.1e3	7.6e3
gcold:1,1000	9.8e3	26.6e3	25.8e3	29.7e3
gcold:10,1000	4.2e3	36.9e3	9.6e3	15.9e3
nboyer:3	1.9e3	1.9e3	1.8e3	855.7e3
5nboyer:3	9.5e3	9.5e3	8.1e3	4.8e6
perm:200,1	2.8e3	2.9e3	4.0e3	4.2e3
perm:25,8	2.9e3	2.9e3	3.9e3	4.0e3
5sboyer:4	4.9e3	5.0e3	4.7e3	1.7e6
twobitlong	190.1e3	421.6e3	218.7e3	526.4e3
twobitshort	75.0e3	326.0e3	77.6e3	313.7e3
5twobitshort	378.8e3	1.7e6	389.7e3	1.7e6
	Remembered set words scanned			
5earley:12	6.5e3	6.6e3	6.7e3	1.9e6
gcbench:5	19.6e6	19.7e6	19.3e6	19.8e6
gcold:1,0	30.9e3	40.2e3	41.1e3	44.7e3
gcold:1,1000	42.0e3	103.7e3	99.3e3	114.6e3
gcold:10,1000	9.8e3	107.8e3	32.5e3	60.6e3
nboyer:3	3.8e3	3.8e3	3.5e3	1.7e6
5nboyer:3	18.9e3	19.0e3	16.3e3	9.6e6
perm:200,1	7.0e3	7.2e3	9.6e3	10.2e3
perm:25,8	7.1e3	7.3e3	9.6e3	9.8e3
5sboyer:4	9.9e3	10.0e3	9.5e3	3.4e6
twobitlong	61.9e6	84.7e6	62.1e6	62.7e6
twobitshort	5.3e6	7.7e6	5.3e6	5.9e6
5twobitshort	15.4e6	20.1e6	15.4e6	18.4e6

Table 8.8: Average amount of 2GEN, 3GEN, and ROF remembered set work

	(none)	Phase	Luck	Limit	All
	No float reduction				
5earley:12	88.5	80.1	78.9	76.4	80.8
gcbench:5	1.4e3	1.4e3	1.0e3	1.4e3	1.0e3
gcold:1,0	224.8	224.8	211.3	224.8	211.3
gcold:1,1000	272.0	272.0	250.4	272.0	250.4
gcold:10,1000	36.2	36.5	32.5	36.2	32.8
nboyer:3	82.3	82.3	76.5	66.3	59.9
5nboyer:3	438.9	432.2	312.3	339.2	283.9
perm:200,1	555.4	555.4	506.1	555.4	506.1
perm:25,8	302.0	302.0	280.6	302.0	280.6
5sboyer:4	99.8	68.1	84.1	74.4	87.2
twobitlong	82.3	80.3	73.4	76.7	72.7
twobitshort	39.7	39.7	36.0	39.6	34.1
5twobitshort	187.2	187.2	170.0	192.8	161.4
	Float reduction before ROF collection				
5earley:12	71.6	68.7	70.4	76.4	80.8
gcbench:5	1.4e3	1.4e3	852.9	1.4e3	852.9
gcold:1,0	224.8	224.8	211.3	224.8	211.3
gcold:1,1000	248.6	248.6	231.3	248.6	231.3
gcold:10,1000	27.1	27.5	27.1	27.1	24.2
nboyer:3	73.9	73.9	62.0	66.3	59.9
5nboyer:3		347.3			283.9
perm:200,1	555.4	555.4	506.1	555.4	506.1
perm:25,8	291.6	291.6	284.1	291.6	284.1
5sboyer:4	72.1	67.2	74.8	74.4	87.2
twobitlong	78.4	75.9	71.6	76.8	69.8
twobitshort	36.7	36.7	34.3	38.9	34.0
5twobitshort	172.8	172.8	157.2	189.3	157.6
	Float reduction before every collection				
5earley:12	71.6	71.6	70.4	74.9	80.2
gcbench:5	996.7	997.6	574.2	996.7	574.2
gcold:1,0	224.8	224.8	211.3	224.8	211.3
gcold:1,1000	248.6	248.6	231.3	248.6	231.3
gcold:10,1000					
nboyer:3	73.9	73.9	62.0	66.3	59.9
5nboyer:3		351.4			
perm:200,1	555.4	555.4	506.1	555.4	506.1
perm:25,8	291.6	291.6	284.1	291.6	284.1
5sboyer:4	72.1	72.1	74.8	74.4	87.2
twobitlong	64.9	64.9	60.0	67.0	60.1
twobitshort	32.5	32.5	30.1	33.4	30.5
5twobitshort	152.2	152.2	137.6	162.7	120.0

Table 8.9: Copy volumes for ROF collector, in megabytes

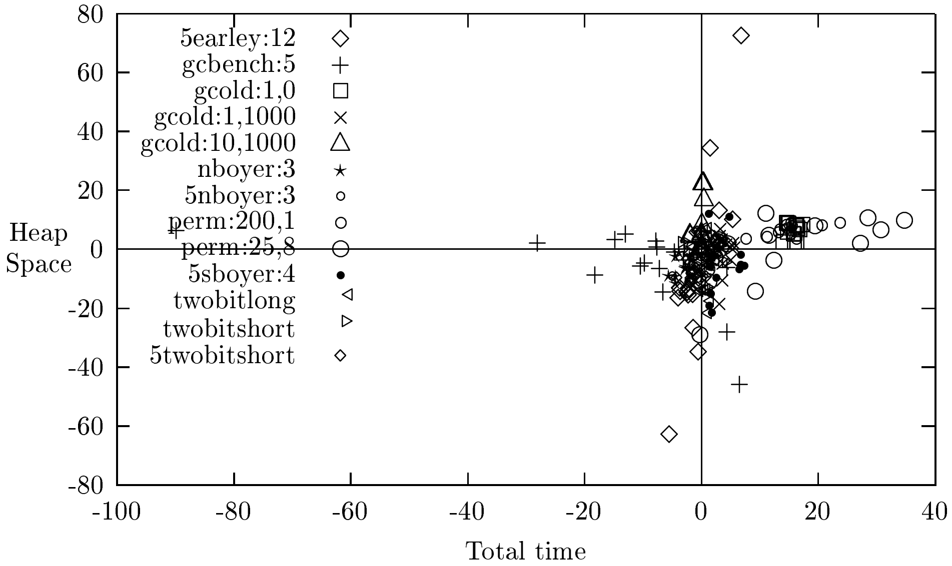

Figure 8.1: Space/time diagram comparing 2GEN (advantage along the negative axes) and ROF (advantage along the positive axes).

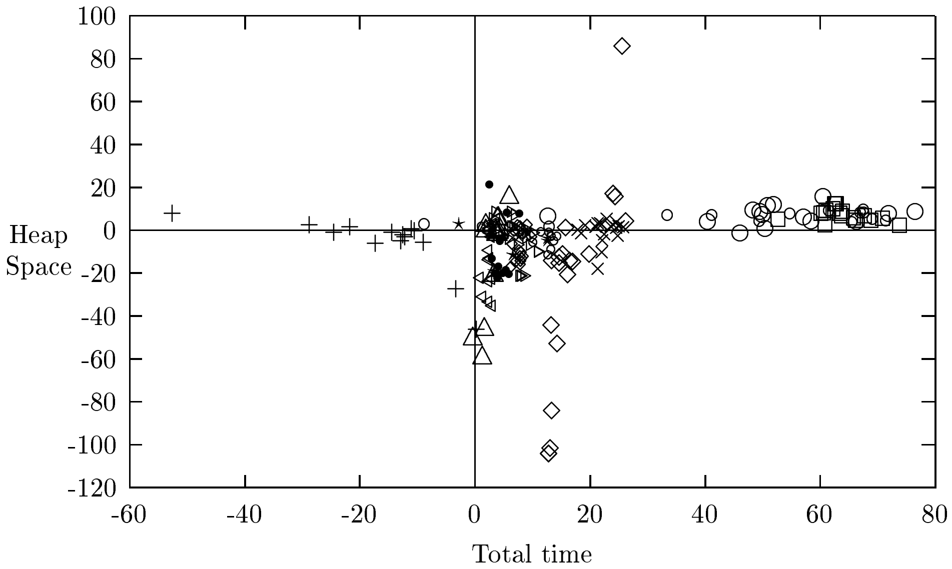

Figure 8.2: Space/time diagram comparing 3GEN (advantage along the negative axes) and ROF (advantage along the positive axes).

Chapter 9

Conclusion and future work

... in fact, it was a battle where courage and
sheer strength played a greater part than scientific methods.
— Thucydides

9.1 Conclusion

I have implemented the renewal-older-first (ROF) and deferred-older-first (DOF) collectors and I have shown that these implementations perform well in practice; in particular, their performance is close to that of a traditional two-generational (2GEN) collector on a number of benchmark programs written in Scheme.

My older-first collectors are not "pure" older-first: they use a nursery—a separate generation for object allocation—on the assumption that the savings from faster object initialization in the nursery will outweigh the cost of evacuating survivors from the nursery into the area that is collected older-first.

On these benchmarks, the ROF collector does tend to lower the mark/cons ratios relative to 2GEN, and on some programs, the DOF collector has mark/cons ratios that are better than or just barely worse than 2GEN. On benchmark programs whose object lifetimes correspond closely to the lifetime model assumed by the older-first collectors, the reduction in the mark/cons ratios can be substantial.

A primary reason the older-first collectors often do not show a significant reduction in the mark/cons ratio relative to 2GEN is that the segmentation of the older-first collected area results in a large amount of floating garbage—copying due to retention of garbage objects referenced from the remembered set.

Despite their competitive mark/cons ratios, the older-first collectors do not perform quite as well as a simple two-generational collector in practice. The latter collector's characteristics—robustness and low cost of object copying—give it an advantage that is matched by the more sophisticated collectors only in those cases where they have reduced the mark/cons ratios substantially.

In general, the older-first collectors are more sensitive to programs with extreme behavior than is 2GEN. For example, the remembered sets of the DOF collector grow much larger than the remembered set of 2GEN, occupying up to 10%–15% as much memory as the garbage-collected heap. In some cases the remembered set growth is due to popular objects, a problem that may be worse in Lisp than in other programming languages. In other cases, however, the remembered sets are large because the programs use shared data structures, large data structures built in a functional manner, or reuse objects throughout a lengthy computation.

The cost of evacuating objects from the nursery can be 30%–40% higher in the older-first collectors, significantly increasing the effective cost of object allocation in benchmarks with high survival rates. In the case of the DOF collector, a similar cost will be paid during other types of garbage collections as well. As the older-first collectors do not provide a substantially lower mark/cons ratio than 2GEN, their overall performance cannot be competitive.

In conclusion, the older-first collectors perform well enough to be considered viable alternatives to traditional younger-first collectors, and can have clear benefits on programs that are a good fit for the particular older-first collection model used by a collector, but there is as yet no evidence that they perform well enough in an actual implementation to be considered superior alternatives for general use.

9.2 Future work

There are several interesting problems that I have not attempted to address in this dissertation.

Foremost is the issue of whether the nursery increases or reduces the performance of the older-first collectors. It is fairly clear, I think, that the nursery limits the opportunities for the older-first collectors to reduce the mark/cons ratio, because a substantial fraction of the copying is performed by the collections that periodically evacuate objects from the nursery. On the other hand, the nursery certainly reduces the cost of initializing an object, and it is an open question whether the nursery is beneficial to performance.

Some of the problems with the older-first collectors—floating garbage and large remembered sets—may be controllable, as my work on tuning parameters in the ROF collector has shown. For example, it may be possible to move generation boundaries in the DOF collector in the same way that it is done in the ROF collector; if so, the problems that the DOF collector has with floating garbage may be reduced. However, the current design of my DOF collector does not allow that type of manipulation, and it is worth exploring alternative, more flexible, designs.

The popular object detection mechanism for the DOF collector outlined in Section 7.8 should be implemented, its performance should be measured, and its efficacy should be assessed.

I have been using a suite consisting of difficult benchmark programs, to show that the older-first collectors perform about as well as the reference collectors under adverse conditions. A different benchmark suite consisting of applications might be appropriate for discovering something about the typical behavior of the new collectors.

Finally, further reductions in the overheads of the older-first collectors may be possible, both reductions in copy cost and in the space required to represent the remembered set. However, since the mark/cons ratios of the older-first collectors are not substantially lower than that of 2GEN, it is unlikely that such tuning will do anything but close the existing gap between the older-first collectors and 2GEN; within the current collector structure, only work on reducing the mark/cons ratio will allow the older-first collectors to edge ahead.

Bibliography

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

[2] Joel F. Bartlett. *Compacting Garbage Collection with Ambiguous Roots.* WRL Research Report 88/2, Digital Equipment Corporation Western Research Laboratory, February 1988.

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

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

[5] 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.

[6] 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.

[7] William Clinger. Personal communication, August 2000.

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

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

[10] Damien Doligez and Xavier Leroy. *A concurrent generational garbage collector for a multi-threaded implementation of ML.* In Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, ACM SIGPLAN Notices, January 1993, pages 113–123.

[11] R. Kent Dybvig, David Eby, and Carl Bruggeman. *Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages.* Technical Report 400, Indiana University Computer Science Department, March 1994.

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

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

[14] 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.

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

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

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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.

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

[24] Andrei Broder and Jorge Stolfi. *Pessimal Algorithms and Simplexity Analysis.* ACM SIGACT News vol. 16 no. 7, Fall 1984, pages 49–53.

[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] SPARC International. *The SPARC Architecture Manual, version 9.* PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.

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

[28] 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.

[29] 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.

[30] 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.

Appendix A

Raw data

The following tables show raw data from the canonical runs of all the collectors on all the benchmarks, at all reported settings.

The benchmark, the collector, and the value of T (if applicable) are identified in a bold header above each section of data. The entries in the column for the amount of data marked always include data scanned in the static area (for the SC collector) and marked by the whole-heap collector (for the DOF collector).

CPU	CPU GC	Num GCs	Marked	Heap used	Heap limit
`gcbench:5` **SC upper+lower T=2.0**					
199.61s	180.28s	851	8290mb	24.0mb	26.0mb
121.37s	102.41s	485	4668mb	23.0mb	29.0mb
88.91s	69.79s	320	3126mb	26.0mb	32.0mb
73.29s	54.42s	260	2475mb	25.0mb	35.0mb
64.64s	45.71s	215	2068mb	28.0mb	38.0mb
`gcold:1,0` **SC upper+lower T=2.0**					
33.79s	28.46s	48	935.5mb	47.0mb	56.0mb
26.01s	20.66s	35	682.5mb	49.5mb	61.0mb
22.26s	16.94s	28	547.5mb	52.0mb	66.0mb
19.70s	14.39s	23	448.0mb	54.5mb	71.0mb
17.85s	12.51s	20	390.0mb	57.0mb	76.0mb
`gcold:1,1000` **SC upper+lower T=2.0**					
72.75s	32.36s	56	1094mb	47.0mb	56.0mb
64.34s	23.81s	41	800.0mb	49.5mb	61.0mb
60.83s	19.88s	33	649.0mb	52.0mb	66.0mb
56.77s	16.22s	27	526.0mb	54.5mb	71.0mb
55.00s	14.48s	24	468.0mb	57.0mb	76.0mb
`gcold:10,1000` **SC upper+lower T=2.0**					
56.82s	19.98s	36	700.5mb	47.0mb	56.0mb
51.79s	14.99s	27	525.5mb	49.5mb	61.0mb
48.53s	11.70s	21	408.5mb	52.0mb	66.0mb
46.97s	10.08s	18	350.5mb	54.5mb	71.0mb
45.76s	8.90s	16	311.5mb	57.0mb	76.0mb
`nboyer:3` **SC upper+lower T=2.0**					
					24.0mb
10.87s	6.55s	26	245.0mb	26.0mb	28.0mb
7.96s	3.52s	14	119.0mb	27.0mb	32.0mb
7.06s	2.69s	10	83.5mb	28.5mb	36.0mb
6.78s	2.37s	8	68.0mb	31.0mb	40.0mb
`perm:200,1` **SC upper+lower T=2.0**					
					25.0mb
44.01s	39.54s	149	1696mb	25.5mb	28.0mb
20.77s	16.40s	66	712.5mb	25.5mb	31.0mb
17.65s	13.34s	49	546.0mb	27.5mb	34.0mb
15.33s	10.99s	39	448.5mb	29.5mb	37.0mb
`perm:25,8` **SC upper+lower T=2.0**					
15.35s	10.86s	49	362.0mb	23.0mb	25.0mb
6.44s	1.97s	24	85.5mb	17.0mb	28.0mb
7.35s	2.87s	24	121.5mb	20.0mb	31.0mb
8.27s	3.77s	24	157.5mb	23.0mb	34.0mb
9.22s	4.68s	24	193.5mb	26.0mb	37.0mb
`twobitlong` **SC upper+lower T=2.0**					
145.19s	11.41s	147	569.5mb	19.5mb	18.0mb
146.74s	9.50s	122	474.0mb	19.0mb	20.0mb
145.27s	8.18s	104	403.5mb	19.0mb	22.0mb
143.58s	6.98s	89	341.5mb	19.5mb	24.0mb
143.34s	6.05s	78	296.5mb	20.5mb	26.0mb
`twobitshort` **SC upper+lower T=2.0**					
27.32s	3.82s	30	153.5mb	17.5mb	18.0mb
25.53s	3.59s	27	141.5mb	18.0mb	20.0mb
25.20s	3.06s	23	120.0mb	18.5mb	22.0mb
24.37s	2.53s	19	97.0mb	19.0mb	24.0mb
24.51s	2.23s	16	83.0mb	20.0mb	26.0mb

Table A.1: Raw data for SC: *upper+lower* runs, one iteration

CPU	CPU GC	Num GCs	Marked	Heap used	Heap limit
`gcbench:5` **SC upper+lower T=3.0**					
98.54s	79.41s	375	3618mb	23.5mb	26.0mb
65.37s	46.28s	221	2097mb	27.0mb	29.0mb
53.11s	33.85s	160	1514mb	30.0mb	32.0mb
50.13s	31.07s	145	1397mb	32.0mb	35.0mb
45.47s	26.28s	125	1196mb	33.5mb	38.0mb
`gcold:1,0` **SC upper+lower T=3.0**					
19.49s	14.10s	23	449.5mb	55.0mb	56.0mb
16.09s	10.72s	17	331.5mb	60.0mb	61.0mb
14.92s	9.55s	15	293.0mb	62.5mb	66.0mb
13.85s	8.48s	13	253.0mb	65.5mb	71.0mb
13.19s	7.84s	12	234.0mb	69.0mb	76.0mb
`gcold:1,1000` **SC upper+lower T=3.0**					
56.76s	16.30s	27	527.0mb	55.0mb	56.0mb
52.78s	12.25s	20	389.5mb	60.0mb	61.0mb
51.58s	11.09s	18	351.5mb	62.5mb	66.0mb
49.85s	9.28s	15	292.5mb	65.5mb	71.0mb
49.35s	8.81s	14	273.5mb	69.0mb	76.0mb
`gcold:10,1000` **SC upper+lower T=3.0**					
47.07s	10.28s	18	350.0mb	55.0mb	56.0mb
44.72s	7.89s	14	272.5mb	60.0mb	61.0mb
43.71s	6.85s	12	233.5mb	62.5mb	66.0mb
42.58s	5.71s	10	194.5mb	65.5mb	71.0mb
41.97s	5.13s	9	175.0mb	69.0mb	76.0mb
`nboyer:3` **SC upper+lower T=3.0**					
					24.0mb
					28.0mb
6.78s	2.35s	8	67.0mb	31.5mb	32.0mb
6.50s	1.98s	6	50.5mb	35.5mb	36.0mb
6.41s	1.85s	5	43.0mb	37.0mb	40.0mb
`perm:200,1` **SC upper+lower T=3.0**					
					25.0mb
					28.0mb
13.68s	9.25s	33	367.0mb	30.5mb	31.0mb
11.60s	7.11s	24	275.5mb	33.0mb	34.0mb
9.71s	5.20s	18	203.0mb	36.0mb	37.0mb
`perm:25,8` **SC upper+lower T=3.0**					
7.75s	3.26s	24	136.0mb	21.0mb	25.0mb
9.49s	4.90s	24	202.5mb	27.0mb	28.0mb
10.56s	6.00s	24	239.5mb	30.5mb	31.0mb
11.66s	7.10s	24	275.5mb	33.0mb	34.0mb
6.27s	1.47s	12	54.0mb	28.0mb	37.0mb
`twobitlong` **SC upper+lower T=3.0**					
145.52s	7.65s	95	369.5mb	19.0mb	18.0mb
143.54s	6.14s	76	292.5mb	21.0mb	20.0mb
142.19s	5.31s	67	254.0mb	22.5mb	22.0mb
143.49s	4.77s	61	231.5mb	23.0mb	24.0mb
142.96s	4.17s	54	201.5mb	24.0mb	26.0mb
`twobitshort` **SC upper+lower T=3.0**					
26.94s	3.35s	25	132.5mb	17.5mb	18.0mb
24.15s	2.35s	18	91.0mb	20.0mb	20.0mb
24.06s	2.07s	14	73.5mb	21.5mb	22.0mb
24.28s	1.61s	11	55.5mb	23.0mb	24.0mb
23.24s	1.48s	10	49.0mb	24.0mb	26.0mb

Table A.2: Raw data for SC: *upper+lower* runs, one iteration, continued

CPU	CPU GC	Num GCs	Marked	Heap used	Heap limit
`5earley:12` **SC upper+lower T=2.0**					
22.03s	11.24s	54	403.0mb	25.5mb	28.0mb
18.60s	7.85s	39	267.5mb	26.0mb	31.0mb
17.39s	6.56s	32	215.0mb	28.5mb	34.0mb
16.48s	5.59s	27	177.5mb	29.5mb	37.0mb
16.10s	5.18s	24	161.5mb	31.0mb	40.0mb
`5nboyer:3` **SC upper+lower T=2.0**					
					24.0mb
53.88s	33.25s	134	1230mb	26.0mb	28.0mb
38.45s	17.82s	74	603.5mb	27.0mb	32.0mb
33.82s	13.29s	54	420.0mb	28.5mb	36.0mb
31.96s	11.45s	44	343.0mb	31.0mb	40.0mb
`5sboyer:4` **SC upper+lower T=2.0**					
					20.0mb
					24.0mb
72.60s	11.06s	49	381.5mb	26.0mb	28.0mb
69.32s	8.13s	34	252.5mb	28.0mb	32.0mb
67.21s	5.59s	24	155.5mb	29.0mb	36.0mb
`5twobitshort` **SC upper+lower T=2.0**					
130.99s	19.44s	154	808.5mb	17.5mb	18.0mb
128.52s	18.10s	140	746.5mb	18.0mb	20.0mb
123.32s	15.01s	116	614.5mb	18.5mb	22.0mb
122.94s	12.61s	96	504.5mb	19.0mb	24.0mb
120.75s	11.05s	82	433.0mb	20.0mb	26.0mb
`5earley:12` **SC upper+lower T=3.0**					
					28.0mb
					31.0mb
14.87s	3.95s	19	115.0mb	31.5mb	34.0mb
14.62s	3.63s	17	102.0mb	34.0mb	37.0mb
13.73s	2.64s	14	69.5mb	34.0mb	40.0mb
`5nboyer:3` **SC upper+lower T=3.0**					
					24.0mb
					28.0mb
32.57s	12.08s	44	358.0mb	31.5mb	32.0mb
30.42s	9.83s	34	275.5mb	36.0mb	36.0mb
28.87s	8.26s	28	218.0mb	37.5mb	40.0mb
`5sboyer:4` **SC upper+lower T=3.0**					
					20.0mb
					24.0mb
					28.0mb
66.07s	4.67s	19	120.0mb	32.0mb	32.0mb
65.42s	3.16s	14	69.0mb	32.5mb	36.0mb
`5twobitshort` **SC upper+lower T=3.0**					
130.56s	17.31s	127	694.5mb	17.5mb	18.0mb
119.20s	12.06s	90	473.5mb	20.0mb	20.0mb
119.93s	10.12s	72	383.5mb	22.0mb	22.0mb
116.22s	8.20s	59	311.5mb	23.5mb	24.0mb
114.79s	6.70s	50	251.0mb	24.0mb	26.0mb

Table A.3: Raw data for SC: *upper+lower* runs, five iterations

CPU	CPU GC	Num GCs	Marked	Heap used	Heap limit
gcbench:5 **SC upper T=3.0**					
98.56s	79.49s	379	3612mb	25.0mb	26.0mb
67.35s	48.27s	230	2162mb	27.5mb	29.0mb
54.51s	35.50s	166	1576mb	30.5mb	32.0mb
51.44s	32.29s	151	1434mb	32.0mb	35.0mb
51.97s	32.85s	151	1424mb	34.5mb	38.0mb
gcold:1,0 **SC upper T=3.0**					
19.55s	14.22s	23	448.5mb	55.0mb	56.0mb
16.73s	11.40s	18	352.0mb	60.0mb	61.0mb
15.44s	10.06s	16	312.5mb	61.5mb	66.0mb
15.44s	10.08s	16	312.5mb	61.5mb	71.0mb
15.46s	10.08s	16	312.5mb	61.5mb	76.0mb
gcold:1,1000 **SC upper T=3.0**					
56.26s	15.57s	26	507.5mb	55.0mb	56.0mb
52.91s	12.34s	20	390.0mb	60.0mb	61.0mb
52.33s	11.74s	19	370.5mb	61.5mb	66.0mb
52.30s	11.70s	19	370.5mb	61.5mb	71.0mb
52.33s	11.74s	19	370.5mb	61.5mb	76.0mb
gcold:10,1000 **SC upper T=3.0**					
47.07s	10.26s	18	350.0mb	55.0mb	56.0mb
44.34s	7.42s	13	253.0mb	60.0mb	61.0mb
43.79s	6.81s	12	233.5mb	61.0mb	66.0mb
43.78s	6.80s	12	233.5mb	61.0mb	71.0mb
43.76s	6.83s	12	233.5mb	61.0mb	76.0mb
nboyer:3 **SC upper T=3.0**					
					24.0mb
					28.0mb
7.09s	2.93s	14	84.5mb	31.5mb	32.0mb
6.73s	2.58s	13	70.5mb	31.5mb	36.0mb
6.73s	2.56s	13	70.5mb	31.5mb	40.0mb
perm:200,1 **SC upper T=3.0**					
					25.0mb
					28.0mb
13.42s	9.09s	33	367.0mb	30.5mb	31.0mb
10.27s	5.89s	22	240.0mb	32.5mb	34.0mb
9.59s	5.12s	18	202.5mb	36.0mb	37.0mb
perm:25,8 **SC upper T=3.0**					
11.30s	6.82s	41	256.5mb	22.5mb	25.0mb
9.67s	5.29s	26	215.0mb	27.0mb	28.0mb
10.73s	6.23s	26	251.0mb	30.0mb	31.0mb
10.55s	6.03s	26	223.5mb	33.5mb	34.0mb
9.72s	5.09s	26	186.5mb	32.0mb	37.0mb
twobitlong **SC upper T=3.0**					
145.60s	7.80s	99	376.0mb	19.0mb	18.0mb
144.20s	6.47s	83	306.0mb	21.0mb	20.0mb
143.53s	5.57s	76	269.0mb	22.0mb	22.0mb
144.37s	5.43s	74	259.5mb	23.5mb	24.0mb
143.94s	5.48s	74	257.0mb	25.0mb	26.0mb
twobitshort **SC upper T=3.0**					
26.12s	3.48s	26	137.5mb	18.5mb	18.0mb
24.32s	2.65s	20	101.5mb	20.5mb	20.0mb
23.55s	1.96s	15	71.5mb	22.5mb	22.0mb
25.06s	1.72s	13	59.5mb	23.0mb	24.0mb
23.34s	1.81s	13	59.5mb	24.0mb	26.0mb

Table A.4: Raw data for SC: *upper* runs, one iteration

CPU	CPU GC	Num GCs	Marked	Heap used	Heap limit
`5earley:12` **SC upper T=3.0**					
					28.0mb
					31.0mb
17.44s	6.70s	43	202.5mb	30.0mb	34.0mb
17.41s	6.55s	41	201.0mb	30.5mb	37.0mb
17.40s	6.61s	41	201.0mb	30.5mb	40.0mb
`5nboyer:3` **SC upper T=3.0**					
					24.0mb
					28.0mb
34.63s	14.33s	71	426.5mb	32.0mb	32.0mb
33.18s	12.93s	65	360.0mb	35.0mb	36.0mb
32.70s	12.43s	61	353.0mb	36.5mb	40.0mb
`5sboyer:4` **SC upper T=3.0**					
					20.0mb
					24.0mb
					28.0mb
					32.0mb
67.67s	6.74s	38	191.5mb	33.5mb	36.0mb
`5twobitshort` **SC upper T=3.0**					
128.38s	17.21s	128	699.5mb	18.5mb	18.0mb
117.81s	12.22s	92	484.0mb	20.5mb	20.0mb
116.50s	10.20s	75	395.5mb	22.5mb	22.0mb
117.41s	8.46s	63	319.5mb	23.5mb	24.0mb
113.03s	7.68s	57	279.5mb	24.5mb	26.0mb

Table A.5: Raw data for SC: *upper* runs, five iterations

CPU	CPU GC	Num GCs	Marked	Heap used	Load
`5earley:12` **SC variable**					
25.65s	14.46s	101	512.0mb	27.5mb	2.25
21.60s	10.66s	66	349.0mb	29.5mb	2.5
19.36s	8.32s	50	263.5mb	30.0mb	2.75
17.91s	6.87s	40	209.0mb	32.5mb	3.0
`5nboyer:3` **SC variable**					
47.03s	26.40s	156	950.5mb	29.0mb	2.25
38.72s	18.38s	103	599.5mb	31.0mb	2.5
35.17s	14.82s	80	444.0mb	33.5mb	2.75
32.88s	12.56s	61	353.0mb	36.5mb	3.0
`5sboyer:4` **SC variable**					
72.70s	11.87s	93	402.5mb	26.5mb	2.25
69.63s	8.87s	64	275.5mb	29.0mb	2.5
67.89s	7.22s	50	211.0mb	31.0mb	2.75
67.08s	6.02s	37	169.0mb	33.5mb	3.0
`5twobitshort` **SC variable**					
120.38s	13.82s	111	553.5mb	19.5mb	2.25
117.76s	10.61s	82	406.5mb	22.5mb	2.5
115.00s	8.67s	65	322.0mb	24.5mb	2.75
113.62s	7.40s	54	265.5mb	27.0mb	3.0

Table A.6: Raw data for SC: *variable* runs, five iterations

CPU	CPU GC	Num GCs	Marked	Heap used	Load
`gcbench:5` **SC variable**					
110.11s	90.87s	439	4134mb	27.0mb	2.25
75.47s	56.24s	265	2520mb	29.5mb	2.5
59.41s	40.41s	196	1790mb	29.5mb	2.75
51.23s	32.11s	148	1418mb	37.5mb	3.0
`gcold:1,0` **SC variable**					
37.14s	31.73s	54	1054mb	46.5mb	2.25
23.44s	18.07s	30	585.0mb	51.5mb	2.5
18.41s	13.06s	21	410.0mb	56.5mb	2.75
15.45s	10.05s	16	312.5mb	61.5mb	3.0
`gcold:1,1000` **SC variable**					
76.10s	35.64s	63	1229mb	46.5mb	2.25
61.18s	20.63s	35	683.0mb	51.5mb	2.5
55.09s	14.56s	24	468.0mb	56.5mb	2.75
52.33s	11.75s	19	370.5mb	61.5mb	3.0
`gcold:10,1000` **SC variable**					
59.27s	22.55s	41	798.0mb	46.0mb	2.25
49.77s	12.81s	23	447.5mb	51.0mb	2.5
45.88s	8.93s	16	311.5mb	56.0mb	2.75
43.76s	6.77s	12	233.5mb	61.0mb	3.0
`nboyer:3` **SC variable**					
9.49s	5.39s	31	189.5mb	28.5mb	2.25
7.93s	3.85s	21	123.0mb	31.0mb	2.5
7.21s	3.09s	16	89.0mb	32.5mb	2.75
6.75s	2.62s	13	70.5mb	31.5mb	3.0
`perm:200,1` **SC variable**					
18.66s	14.26s	52	591.0mb	28.5mb	2.25
13.08s	8.77s	31	350.5mb	31.5mb	2.5
10.63s	6.27s	22	248.5mb	34.5mb	2.75
9.40s	4.94s	17	192.0mb	37.5mb	3.0
`perm:25,8` **SC variable**					
17.63s	13.22s	67	513.0mb	24.0mb	2.25
12.61s	8.18s	42	309.0mb	30.5mb	2.5
10.41s	5.97s	29	230.5mb	33.0mb	2.75
9.41s	4.87s	24	178.0mb	33.5mb	3.0
`twobitlong` **SC variable**					
146.14s	9.75s	138	486.5mb	21.0mb	2.25
144.37s	7.45s	106	367.0mb	23.0mb	2.5
144.98s	6.17s	86	297.0mb	24.5mb	2.75
144.15s	5.23s	73	251.0mb	26.5mb	3.0
`twobitshort` **SC variable**					
24.54s	2.86s	23	105.5mb	19.5mb	2.25
24.83s	2.24s	18	80.5mb	22.0mb	2.5
23.71s	1.92s	14	64.5mb	24.5mb	2.75
23.01s	1.67s	12	53.5mb	27.0mb	3.0

Table A.7: Raw data for SC: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **2GEN upper+lower T=2.0**						
86.76s	64.21s	8.16s	256	2755mb	25.0mb	26.0mb
64.25s	41.51s	9.44s	150	1778mb	23.0mb	29.0mb
56.05s	33.56s	10.33s	100	1392mb	27.0mb	32.0mb
49.35s	26.66s	10.41s	75	1114mb	26.5mb	35.0mb
46.47s	23.87s	10.83s	60	984.5mb	27.5mb	38.0mb
`gcold:1,0` **2GEN upper+lower T=2.0**						
26.96s	17.93s	3.77s	24	604.0mb	47.0mb	56.0mb
23.23s	14.25s	3.93s	17	470.0mb	49.0mb	61.0mb
21.13s	12.23s	4.01s	13	395.0mb	52.0mb	66.0mb
20.10s	11.14s	4.11s	11	359.5mb	55.0mb	71.0mb
19.07s	10.12s	4.12s	9	319.5mb	57.5mb	76.0mb
`gcold:1,1000` **2GEN upper+lower T=2.0**						
62.27s	18.47s	4.00s	25	626.0mb	47.0mb	56.0mb
58.55s	14.87s	4.07s	18	493.0mb	49.0mb	61.0mb
55.87s	12.10s	4.03s	13	397.0mb	52.0mb	66.0mb
55.00s	11.16s	4.12s	11	361.0mb	54.5mb	71.0mb
54.00s	10.23s	4.22s	9	323.0mb	57.0mb	76.0mb
`gcold:10,1000` **2GEN upper+lower T=2.0**						
38.19s	1.94s	0.37s	2	53.0mb	46.5mb	56.0mb
38.08s	2.12s	0.57s	2	53.0mb	49.0mb	61.0mb
38.24s	2.14s	0.53s	2	53.5mb	51.5mb	66.0mb
37.45s	1.27s	0.46s	1	34.5mb	54.0mb	71.0mb
37.51s	1.36s	0.53s	1	34.5mb	56.5mb	76.0mb
`nboyer:3` **2GEN upper+lower T=2.0**						
						24.0mb
9.29s	4.67s	1.03s	14	170.5mb	26.0mb	28.0mb
7.61s	3.08s	1.19s	7	103.0mb	27.0mb	32.0mb
6.88s	2.41s	1.31s	4	72.0mb	29.0mb	36.0mb
6.68s	2.18s	1.30s	3	66.5mb	30.0mb	40.0mb
`perm:200,1` **2GEN upper+lower T=2.0**						
						25.0mb
38.68s	32.50s	2.42s	114	1354mb	26.0mb	28.0mb
32.05s	25.99s	3.07s	84	1068mb	27.0mb	31.0mb
24.43s	18.36s	3.63s	53	757.0mb	28.5mb	34.0mb
20.71s	14.69s	3.99s	38	602.0mb	30.0mb	37.0mb
`perm:25,8` **2GEN upper+lower T=2.0**						
20.58s	14.60s	3.79s	49	536.0mb	24.0mb	25.0mb
12.34s	6.39s	4.39s	24	280.0mb	18.0mb	28.0mb
13.30s	7.28s	4.33s	24	316.0mb	21.0mb	31.0mb
14.27s	8.23s	4.32s	24	352.0mb	24.0mb	34.0mb
15.23s	9.21s	4.36s	24	388.0mb	27.0mb	37.0mb
`twobitlong` **2GEN upper+lower T=2.0**						
144.70s	5.43s	3.98s	16	104.0mb	17.0mb	18.0mb
143.78s	5.29s	3.85s	13	99.0mb	17.5mb	20.0mb
144.95s	5.22s	4.07s	10	87.0mb	18.5mb	22.0mb
145.12s	4.74s	3.81s	8	79.5mb	19.0mb	24.0mb
143.37s	4.99s	4.11s	7	75.5mb	20.5mb	26.0mb
`twobitshort` **2GEN upper+lower T=2.0**						
23.45s	1.87s	0.81s	6	50.0mb	18.0mb	18.0mb
23.44s	1.87s	0.88s	6	49.0mb	17.0mb	20.0mb
23.93s	1.78s	0.82s	5	46.0mb	18.0mb	22.0mb
23.36s	1.73s	0.90s	4	41.0mb	19.0mb	24.0mb
23.01s	1.50s	0.86s	3	37.0mb	20.5mb	26.0mb

Table A.8: Raw data for 2GEN: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **2GEN upper+lower T=3.0**						
58.17s	35.70s	9.99s	120	1495mb	25.0mb	26.0mb
53.18s	30.82s	10.69s	85	1256mb	28.0mb	29.0mb
45.33s	22.85s	11.15s	50	906.5mb	31.5mb	32.0mb
43.17s	20.94s	11.47s	40	831.5mb	32.5mb	35.0mb
42.02s	19.36s	11.30s	35	788.0mb	34.5mb	38.0mb
`gcold:1,0` **2GEN upper+lower T=3.0**						
20.07s	11.13s	4.12s	11	358.0mb	55.5mb	56.0mb
18.63s	9.71s	4.26s	8	302.5mb	59.5mb	61.0mb
18.10s	9.10s	4.19s	7	281.5mb	62.0mb	66.0mb
17.80s	8.84s	4.44s	6	265.0mb	65.5mb	71.0mb
17.32s	8.40s	4.58s	5	246.0mb	69.0mb	76.0mb
`gcold:1,1000` **2GEN upper+lower T=3.0**						
54.94s	11.35s	4.35s	11	360.5mb	55.5mb	56.0mb
54.01s	9.79s	4.29s	8	302.5mb	59.5mb	61.0mb
53.12s	9.38s	4.38s	7	285.5mb	62.0mb	66.0mb
52.71s	8.87s	4.39s	6	266.0mb	65.5mb	71.0mb
52.33s	8.49s	4.62s	5	247.5mb	69.0mb	76.0mb
`gcold:10,1000` **2GEN upper+lower T=3.0**						
37.48s	1.28s	0.45s	1	34.5mb	55.0mb	56.0mb
37.54s	1.44s	0.58s	1	34.5mb	58.5mb	61.0mb
37.60s	1.43s	0.58s	1	34.5mb	62.0mb	66.0mb
36.83s	0.63s	0.63s	0	15.5mb	45.0mb	71.0mb
36.81s	0.60s	0.60s	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **2GEN upper+lower T=3.0**						
						24.0mb
						28.0mb
6.70s	2.22s	1.30s	3	68.0mb	30.0mb	32.0mb
6.53s	2.00s	1.35s	2	60.5mb	33.5mb	36.0mb
6.58s	2.05s	1.36s	2	62.0mb	36.0mb	40.0mb
`perm:200,1` **2GEN upper+lower T=3.0**						
						25.0mb
						28.0mb
20.54s	14.52s	4.07s	37	592.0mb	31.0mb	31.0mb
18.53s	12.54s	4.30s	27	505.0mb	33.5mb	34.0mb
16.18s	10.19s	4.53s	19	416.5mb	36.0mb	37.0mb
`perm:25,8` **2GEN upper+lower T=3.0**						
13.63s	7.61s	4.34s	24	327.0mb	22.0mb	25.0mb
15.32s	9.40s	4.38s	24	391.0mb	27.5mb	28.0mb
						31.0mb
17.51s	11.53s	4.37s	24	464.0mb	33.5mb	34.0mb
12.31s	6.37s	4.95s	12	264.0mb	28.5mb	37.0mb
`twobitlong` **2GEN upper+lower T=3.0**						
143.77s	4.88s	3.77s	10	87.0mb	18.0mb	18.0mb
143.00s	4.68s	3.85s	7	75.0mb	20.5mb	20.0mb
143.62s	4.60s	4.06s	5	66.0mb	20.5mb	22.0mb
143.78s	4.86s	4.26s	5	67.5mb	21.5mb	24.0mb
143.23s	5.09s	4.35s	5	71.5mb	24.0mb	26.0mb
`twobitshort` **2GEN upper+lower T=3.0**						
23.33s	1.85s	0.91s	5	46.5mb	17.5mb	18.0mb
23.82s	1.69s	0.88s	4	42.0mb	20.5mb	20.0mb
22.88s	1.27s	0.88s	2	31.5mb	20.0mb	22.0mb
22.94s	1.40s	0.98s	2	31.5mb	21.5mb	24.0mb
22.73s	1.24s	1.01s	1	27.0mb	23.0mb	26.0mb

Table A.9: Raw data for 2GEN: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
5earley:12 2GEN upper+lower T=2.0						
13.48s	3.70s	1.70s	9	114.5mb	25.0mb	28.0mb
11.70s	2.00s	1.95s	4	63.5mb	15.5mb	31.0mb
11.89s	2.09s	1.86s	4	68.0mb	18.5mb	34.0mb
12.02s	2.24s	1.80s	4	74.0mb	21.0mb	37.0mb
12.34s	2.46s	1.82s	4	80.0mb	24.0mb	40.0mb
5nboyer:3 2GEN upper+lower T=2.0						
						24.0mb
45.75s	23.01s	4.83s	74	855.0mb	26.0mb	28.0mb
37.18s	14.31s	5.10s	39	521.5mb	27.0mb	32.0mb
33.98s	11.40s	5.50s	26	403.0mb	29.0mb	36.0mb
32.24s	9.41s	5.47s	19	339.5mb	30.0mb	40.0mb
5sboyer:4 2GEN upper+lower T=2.0						
						20.0mb
						24.0mb
72.47s	5.18s	1.84s	14	169.0mb	25.0mb	28.0mb
71.52s	4.17s	1.89s	9	128.0mb	27.5mb	32.0mb
69.51s	2.31s	2.02s	4	72.5mb	19.5mb	36.0mb
5twobitshort 2GEN upper+lower T=2.0						
115.35s	8.07s	3.26s	34	248.0mb	18.0mb	18.0mb
114.33s	7.68s	3.39s	30	229.5mb	18.0mb	20.0mb
113.44s	7.34s	3.37s	26	215.5mb	18.0mb	22.0mb
112.44s	6.43s	3.45s	20	183.5mb	19.0mb	24.0mb
113.95s	6.14s	3.59s	16	167.5mb	20.5mb	26.0mb
5earley:12 2GEN upper+lower T=3.0						
11.96s	2.17s	1.81s	4	72.0mb	20.0mb	28.0mb
12.27s	2.39s	1.75s	4	79.5mb	23.5mb	31.0mb
12.53s	2.68s	1.77s	4	87.5mb	27.5mb	34.0mb
12.81s	3.02s	1.83s	4	95.0mb	31.5mb	37.0mb
12.83s	2.97s	1.80s	3	93.5mb	37.0mb	40.0mb
5nboyer:3 2GEN upper+lower T=3.0						
						24.0mb
						28.0mb
32.32s	9.76s	5.55s	19	348.0mb	31.0mb	32.0mb
31.35s	8.57s	5.50s	14	307.0mb	33.5mb	36.0mb
31.56s	8.83s	5.63s	13	311.5mb	37.0mb	40.0mb
5sboyer:4 2GEN upper+lower T=3.0						
						20.0mb
						24.0mb
69.52s	2.32s	2.05s	4	72.5mb	19.5mb	28.0mb
69.69s	2.70s	2.07s	4	82.0mb	25.0mb	32.0mb
70.06s	2.88s	1.87s	4	92.5mb	30.0mb	36.0mb
5twobitshort 2GEN upper+lower T=3.0						
115.40s	7.61s	3.35s	28	224.0mb	19.0mb	18.0mb
116.86s	6.88s	3.54s	21	190.5mb	20.5mb	20.0mb
113.02s	5.77s	3.55s	14	155.5mb	21.0mb	22.0mb
112.84s	5.40s	3.60s	11	142.5mb	22.0mb	24.0mb
112.23s	5.03s	3.58s	9	134.0mb	24.0mb	26.0mb

Table A.10: Raw data for 2GEN: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **2GEN upper T=3.0**						
58.69s	36.66s	10.08s	126	1526mb	25.0mb	26.0mb
53.81s	31.73s	10.36s	91	1284mb	28.0mb	29.0mb
45.72s	23.58s	10.85s	56	947.5mb	31.5mb	32.0mb
44.43s	22.32s	11.14s	47	893.0mb	33.0mb	35.0mb
42.88s	20.93s	11.19s	42	831.5mb	33.5mb	38.0mb
`gcold:1,0` **2GEN upper T=3.0**						
19.88s	10.83s	3.96s	11	358.0mb	54.5mb	56.0mb
18.44s	9.52s	4.17s	8	303.0mb	59.5mb	61.0mb
17.97s	8.90s	4.01s	7	283.0mb	62.0mb	66.0mb
17.97s	9.05s	4.17s	7	283.0mb	62.0mb	71.0mb
17.97s	8.92s	4.05s	7	283.0mb	62.0mb	76.0mb
`gcold:1,1000` **2GEN upper T=3.0**						
54.75s	11.02s	4.13s	11	360.5mb	54.5mb	56.0mb
53.37s	9.61s	4.17s	8	304.0mb	59.0mb	61.0mb
52.90s	9.19s	4.25s	7	285.0mb	62.0mb	66.0mb
52.91s	9.19s	4.25s	7	285.0mb	62.0mb	71.0mb
52.93s	9.18s	4.25s	7	285.0mb	62.0mb	76.0mb
`gcold:10,1000` **2GEN upper T=3.0**						
37.53s	1.36s	0.54s	1	34.5mb	55.0mb	56.0mb
37.51s	1.41s	0.57s	1	34.5mb	55.0mb	61.0mb
37.51s	1.35s	0.52s	1	34.5mb	55.0mb	66.0mb
37.52s	1.40s	0.57s	1	34.5mb	55.0mb	71.0mb
37.53s	1.39s	0.56s	1	34.5mb	55.0mb	76.0mb
`nboyer:3` **2GEN upper T=3.0**						
						24.0mb
						28.0mb
6.93s	2.40s	1.12s	7	76.0mb	30.5mb	32.0mb
7.01s	2.52s	1.18s	7	77.5mb	35.0mb	36.0mb
7.01s	2.47s	1.13s	7	77.5mb	35.0mb	40.0mb
`perm:200,1` **2GEN upper T=3.0**						
						25.0mb
						28.0mb
20.48s	14.42s	3.98s	37	593.0mb	30.0mb	31.0mb
18.74s	12.70s	4.23s	28	513.5mb	33.5mb	34.0mb
16.13s	10.00s	4.38s	19	417.0mb	36.0mb	37.0mb
`perm:25,8` **2GEN upper T=3.0**						
18.78s	12.75s	3.88s	44	481.5mb	24.0mb	25.0mb
15.54s	9.54s	4.30s	25	398.0mb	27.5mb	28.0mb
						31.0mb
16.92s	10.91s	4.38s	25	434.5mb	33.5mb	34.0mb
15.35s	9.35s	4.45s	25	370.5mb	31.0mb	37.0mb
`twobitlong` **2GEN upper T=3.0**						
144.00s	4.95s	3.91s	11	86.0mb	18.0mb	18.0mb
143.68s	5.00s	4.07s	9	79.5mb	20.5mb	20.0mb
144.38s	4.81s	4.14s	7	69.5mb	20.5mb	22.0mb
144.68s	4.71s	4.03s	7	69.5mb	20.5mb	24.0mb
144.16s	4.71s	4.07s	6	68.0mb	22.0mb	26.0mb
`twobitshort` **2GEN upper T=3.0**						
23.36s	1.78s	0.86s	6	46.0mb	17.5mb	18.0mb
23.96s	1.57s	0.81s	5	40.5mb	19.0mb	20.0mb
23.02s	1.52s	0.91s	4	36.0mb	21.0mb	22.0mb
22.80s	1.32s	0.92s	3	30.5mb	22.5mb	24.0mb
22.83s	1.36s	0.95s	3	30.5mb	23.0mb	26.0mb

Table A.11: Raw data for 2GEN: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`5earley:12` **2GEN upper T=3.0**						
12.98s	3.21s	1.58s	11	112.0mb	22.0mb	28.0mb
13.45s	3.54s	1.54s	11	123.5mb	26.0mb	31.0mb
13.22s	3.29s	1.53s	9	116.0mb	29.5mb	34.0mb
13.17s	3.25s	1.55s	8	111.0mb	33.0mb	37.0mb
13.18s	3.36s	1.62s	8	111.0mb	33.0mb	40.0mb
`5nboyer:3` **2GEN upper T=3.0**						
						24.0mb
						28.0mb
33.41s	10.86s	5.14s	33	386.0mb	31.5mb	32.0mb
32.86s	10.30s	5.14s	27	371.5mb	35.5mb	36.0mb
32.77s	10.18s	5.21s	26	362.5mb	35.0mb	40.0mb
`5sboyer:4` **2GEN upper T=3.0**						
						20.0mb
						24.0mb
74.20s	3.26s	1.58s	12	116.0mb	22.5mb	28.0mb
73.95s	3.30s	1.73s	10	111.0mb	27.0mb	32.0mb
74.11s	3.61s	1.81s	9	117.0mb	32.5mb	36.0mb
`5twobitshort` **2GEN upper T=3.0**						
113.37s	7.50s	3.33s	29	223.5mb	19.0mb	18.0mb
114.30s	6.72s	3.55s	22	188.5mb	20.0mb	20.0mb
111.20s	5.72s	3.51s	15	155.5mb	21.0mb	22.0mb
110.91s	5.14s	3.42s	12	142.5mb	22.5mb	24.0mb
112.07s	5.33s	3.43s	11	143.5mb	23.5mb	26.0mb

Table A.12: Raw data for 2GEN: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
`5earley:12` **2GEN variable**						
15.24s	5.06s	1.43s	30	174.0mb	20.5mb	2.25
14.07s	3.95s	1.44s	15	139.5mb	26.5mb	2.5
13.94s	3.69s	1.38s	12	126.0mb	35.5mb	2.75
13.27s	3.37s	1.83s	11	107.5mb	26.5mb	3.0
`5nboyer:3` **2GEN variable**						
39.37s	16.50s	4.44s	73	597.0mb	27.5mb	2.25
35.71s	12.83s	4.83s	48	458.5mb	30.0mb	2.5
34.04s	11.13s	4.90s	35	398.5mb	33.5mb	2.75
33.04s	10.26s	5.13s	26	363.0mb	35.0mb	3.0
`5sboyer:4` **2GEN variable**						
73.33s	5.61s	1.48s	27	195.0mb	25.0mb	2.25
72.32s	4.48s	1.60s	17	153.5mb	26.0mb	2.5
71.40s	3.80s	1.71s	11	126.5mb	32.5mb	2.75
74.24s	3.79s	1.96s	9	117.0mb	33.0mb	3.0
`5twobitshort` **2GEN variable**						
111.61s	6.20s	3.34s	22	178.5mb	19.5mb	2.25
113.07s	5.68s	3.49s	15	156.5mb	22.5mb	2.5
112.23s	5.42s	3.43s	12	148.0mb	25.0mb	2.75
110.60s	4.96s	3.57s	10	133.0mb	25.5mb	3.0

Table A.13: Raw data for 2GEN: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
`gcbench:5` **2GEN variable**						
56.29s	34.13s	9.93s	109	1421mb	28.0mb	2.25
48.42s	26.33s	10.79s	68	1084mb	32.5mb	2.5
44.42s	22.28s	10.99s	52	902.5mb	35.5mb	2.75
42.69s	20.36s	11.04s	39	819.0mb	38.5mb	3.0
`gcold:1,0` **2GEN variable**						
27.13s	18.07s	3.65s	25	620.0mb	47.0mb	2.25
20.84s	11.82s	3.85s	13	395.0mb	52.0mb	2.5
19.47s	10.52s	3.92s	10	338.0mb	57.5mb	2.75
17.93s	8.87s	4.05s	7	283.0mb	62.0mb	3.0
`gcold:1,1000` **2GEN variable**						
61.83s	18.10s	3.94s	25	624.0mb	47.0mb	2.25
56.18s	12.36s	3.86s	14	417.0mb	52.0mb	2.5
54.40s	10.60s	4.04s	10	342.0mb	57.0mb	2.75
52.84s	9.05s	4.17s	7	285.0mb	62.0mb	3.0
`gcold:10,1000` **2GEN variable**						
38.56s	2.52s	0.52s	3	72.0mb	46.0mb	2.25
37.37s	1.31s	0.53s	1	34.5mb	48.5mb	2.5
37.50s	1.24s	0.44s	1	34.5mb	52.5mb	2.75
37.50s	1.49s	0.67s	1	34.5mb	55.0mb	3.0
`nboyer:3` **2GEN variable**						
8.19s	3.58s	0.88s	16	125.5mb	27.5mb	2.25
7.46s	2.91s	0.98s	11	96.0mb	29.0mb	2.5
6.88s	2.21s	0.92s	8	75.5mb	30.0mb	2.75
7.00s	2.42s	1.10s	7	77.5mb	35.0mb	3.0
`perm:200,1` **2GEN variable**						
22.48s	16.50s	3.75s	46	683.5mb	29.0mb	2.25
18.55s	12.55s	4.14s	28	513.5mb	32.5mb	2.5
16.54s	10.51s	4.33s	21	436.5mb	35.0mb	2.75
15.37s	9.40s	4.55s	16	386.0mb	38.0mb	3.0
`perm:25,8` **2GEN variable**						
20.09s	14.02s	3.57s	54	564.5mb	23.0mb	2.25
17.52s	11.52s	3.95s	39	458.5mb	26.5mb	2.5
15.74s	9.75s	4.25s	28	396.0mb	32.0mb	2.75
14.81s	8.81s	4.44s	24	356.5mb	32.5mb	3.0
`twobitlong` **2GEN variable**						
144.49s	5.01s	3.92s	13	88.0mb	17.5mb	2.25
143.10s	4.65s	3.83s	9	76.0mb	19.5mb	2.5
144.39s	4.92s	4.15s	8	72.5mb	21.0mb	2.75
143.82s	4.57s	3.94s	6	68.5mb	23.0mb	3.0
`twobitshort` **2GEN variable**						
23.81s	1.68s	0.96s	5	39.0mb	19.0mb	2.25
23.58s	1.49s	0.92s	4	34.5mb	22.5mb	2.5
22.93s	1.32s	1.01s	3	29.5mb	19.0mb	2.75
22.83s	1.39s	0.97s	3	30.5mb	23.0mb	3.0

Table A.14: Raw data for 2GEN: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **3GEN upper+lower T=2.0**						
						26.0mb
49.73s	27.66s	2.42s	54	966.0mb	27.0mb	29.0mb
47.09s	25.04s	2.79s	43	866.5mb	27.0mb	32.0mb
46.86s	24.48s	2.87s	42	860.5mb	27.0mb	35.0mb
42.23s	20.28s	4.62s	22	683.5mb	30.0mb	38.0mb
`gcold:1,0` **3GEN upper+lower T=2.0**						
37.24s	28.05s	5.53s	24	786.5mb	47.0mb	56.0mb
33.82s	24.47s	5.82s	19	699.5mb	50.0mb	61.0mb
30.31s	21.05s	6.13s	14	612.0mb	51.5mb	66.0mb
28.24s	19.14s	6.40s	11	560.0mb	54.5mb	71.0mb
26.84s	17.67s	6.38s	9	522.5mb	58.0mb	76.0mb
`gcold:1,1000` **3GEN upper+lower T=2.0**						
73.47s	29.19s	5.31s	24	844.5mb	47.0mb	56.0mb
70.15s	26.15s	5.77s	19	757.0mb	49.5mb	61.0mb
66.82s	22.66s	6.00s	14	669.0mb	51.5mb	66.0mb
64.77s	20.77s	6.23s	11	616.5mb	54.5mb	71.0mb
63.33s	19.33s	6.21s	9	581.5mb	56.5mb	76.0mb
`gcold:10,1000` **3GEN upper+lower T=2.0**						
39.52s	3.38s	0.85s	2	95.5mb	46.5mb	56.0mb
38.85s	2.55s	0.77s	1	77.5mb	48.5mb	61.0mb
38.87s	2.72s	0.97s	1	77.5mb	52.0mb	66.0mb
38.88s	2.67s	0.86s	1	77.5mb	54.0mb	71.0mb
38.11s	1.74s	0.77s	0	59.5mb	37.0mb	76.0mb
`nboyer:3` **3GEN upper+lower T=2.0**						
						24.0mb
9.12s	4.56s	0.96s	9	154.5mb	26.5mb	28.0mb
8.15s	3.59s	1.11s	5	123.5mb	27.5mb	32.0mb
7.74s	3.17s	1.24s	3	110.5mb	29.0mb	36.0mb
7.85s	3.40s	1.36s	3	114.0mb	31.0mb	40.0mb
`perm:200,1` **3GEN upper+lower T=2.0**						
						25.0mb
35.48s	29.08s	1.57s	77	1064mb	27.0mb	28.0mb
35.19s	28.95s	1.67s	76	1056mb	27.0mb	31.0mb
29.64s	23.48s	3.68s	47	822.0mb	29.5mb	34.0mb
27.78s	21.64s	4.35s	38	758.0mb	30.0mb	37.0mb
`perm:25,8` **3GEN upper+lower T=2.0**						
23.55s	17.36s	4.00s	44	629.0mb	24.5mb	25.0mb
20.57s	14.52s	5.42s	30	528.5mb	24.5mb	28.0mb
19.64s	13.67s	6.08s	24	495.0mb	23.0mb	31.0mb
20.91s	14.92s	6.00s	24	534.0mb	26.0mb	34.0mb
22.02s	15.91s	5.88s	24	567.0mb	29.0mb	37.0mb
`twobitlong` **3GEN upper+lower T=2.0**						
146.72s	6.44s	4.43s	6	113.5mb	17.5mb	18.0mb
146.94s	6.65s	4.76s	5	110.0mb	17.5mb	20.0mb
146.51s	6.42s	4.65s	4	106.5mb	18.5mb	22.0mb
146.19s	6.37s	4.72s	3	104.0mb	18.5mb	24.0mb
146.39s	6.16s	4.63s	2	99.0mb	18.5mb	26.0mb
`twobitshort` **3GEN upper+lower T=2.0**						
24.81s	2.86s	0.59s	4	82.5mb	18.5mb	18.0mb
24.78s	2.85s	0.59s	4	83.5mb	18.0mb	20.0mb
24.70s	2.84s	0.69s	3	80.5mb	18.5mb	22.0mb
24.70s	2.83s	0.63s	3	80.5mb	18.5mb	24.0mb
25.15s	2.62s	0.70s	2	76.0mb	20.0mb	26.0mb

Table A.15: Raw data for 3GEN: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **3GEN upper+lower T=3.0**						
						26.0mb
						29.0mb
45.49s	23.53s	3.29s	37	819.0mb	30.0mb	32.0mb
42.23s	20.03s	4.66s	21	687.0mb	33.0mb	35.0mb
41.24s	19.23s	5.14s	16	656.5mb	35.5mb	38.0mb
`gcold:1,0` **3GEN upper+lower T=3.0**						
28.96s	19.77s	6.36s	12	577.0mb	54.5mb	56.0mb
26.89s	17.47s	6.29s	9	524.0mb	58.0mb	61.0mb
25.12s	15.59s	6.55s	6	470.5mb	64.0mb	66.0mb
24.87s	15.65s	6.70s	6	471.5mb	65.5mb	71.0mb
24.21s	15.00s	6.68s	5	453.5mb	68.5mb	76.0mb
`gcold:1,1000` **3GEN upper+lower T=3.0**						
65.42s	21.35s	6.12s	12	634.0mb	54.5mb	56.0mb
63.35s	19.28s	6.18s	9	581.5mb	58.0mb	61.0mb
62.02s	17.94s	6.33s	7	546.5mb	62.5mb	66.0mb
61.31s	17.23s	6.28s	6	529.0mb	65.5mb	71.0mb
60.62s	16.49s	6.46s	5	511.5mb	68.5mb	76.0mb
`gcold:10,1000` **3GEN upper+lower T=3.0**						
38.91s	2.59s	0.78s	1	77.5mb	54.0mb	56.0mb
38.14s	1.89s	0.92s	0	59.5mb	37.0mb	61.0mb
38.11s	1.75s	0.76s	0	59.5mb	37.0mb	66.0mb
38.11s	1.81s	0.79s	0	59.5mb	37.0mb	71.0mb
38.12s	1.82s	0.83s	0	59.5mb	37.0mb	76.0mb
`nboyer:3` **3GEN upper+lower T=3.0**						
						24.0mb
8.31s	3.75s	1.13s	5	127.0mb	27.5mb	28.0mb
7.42s	2.90s	1.36s	2	102.0mb	28.5mb	32.0mb
7.55s	3.04s	1.34s	2	105.5mb	34.0mb	36.0mb
7.19s	2.70s	1.34s	1	95.0mb	33.0mb	40.0mb
`perm:200,1` **3GEN upper+lower T=3.0**						
						25.0mb
						28.0mb
30.06s	23.83s	3.69s	47	838.0mb	30.0mb	31.0mb
25.66s	19.47s	5.36s	28	676.0mb	32.5mb	34.0mb
23.52s	17.38s	6.32s	19	607.0mb	36.0mb	37.0mb
`perm:25,8` **3GEN upper+lower T=3.0**						
19.73s	13.73s	6.04s	24	498.0mb	23.0mb	25.0mb
						28.0mb
						31.0mb
22.33s	16.21s	6.24s	20	573.0mb	33.0mb	34.0mb
19.46s	13.37s	6.84s	13	490.0mb	33.5mb	37.0mb
`twobitlong` **3GEN upper+lower T=3.0**						
147.63s	6.63s	4.75s	4	108.0mb	18.5mb	18.0mb
147.44s	6.34s	4.68s	3	103.5mb	18.5mb	20.0mb
146.13s	6.16s	4.68s	2	99.0mb	18.5mb	22.0mb
146.04s	6.00s	4.65s	1	96.0mb	18.5mb	24.0mb
146.75s	6.14s	4.83s	1	96.0mb	19.0mb	26.0mb
`twobitshort` **3GEN upper+lower T=3.0**						
24.72s	2.84s	0.64s	3	80.5mb	18.5mb	18.0mb
24.71s	2.85s	0.64s	3	80.5mb	18.5mb	20.0mb
25.18s	2.64s	0.72s	3	78.5mb	20.0mb	22.0mb
25.06s	2.47s	0.68s	2	74.0mb	21.0mb	24.0mb
24.91s	2.46s	0.77s	1	72.0mb	21.0mb	26.0mb

Table A.16: Raw data for 3GEN: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`5earley:12` **3GEN upper+lower T=2.0**						
15.91s	5.90s	2.07s	9	200.5mb	26.0mb	28.0mb
15.57s	5.48s	2.15s	8	191.0mb	26.0mb	31.0mb
14.30s	4.24s	2.56s	4	153.0mb	17.5mb	34.0mb
14.39s	4.48s	2.55s	4	159.5mb	22.0mb	37.0mb
14.60s	4.53s	2.48s	4	164.5mb	24.5mb	40.0mb
`5nboyer:3` **3GEN upper+lower T=2.0**						
						24.0mb
47.08s	24.16s	4.65s	52	801.0mb	26.5mb	28.0mb
41.65s	18.71s	5.22s	31	636.5mb	27.5mb	32.0mb
38.86s	16.01s	6.05s	20	555.5mb	29.0mb	36.0mb
38.51s	15.84s	6.26s	17	546.5mb	31.0mb	40.0mb
`5sboyer:4` **3GEN upper+lower T=2.0**						
						20.0mb
						24.0mb
74.53s	7.42s	2.11s	14	250.5mb	25.5mb	28.0mb
73.49s	6.01s	2.34s	9	208.0mb	27.0mb	32.0mb
71.93s	4.45s	2.57s	4	161.0mb	20.0mb	36.0mb
`5twobitshort` **3GEN upper+lower T=2.0**						
123.60s	13.45s	1.68s	19	412.5mb	18.5mb	18.0mb
123.84s	13.34s	1.62s	19	412.5mb	18.0mb	20.0mb
123.66s	13.35s	1.63s	17	407.5mb	18.5mb	22.0mb
123.18s	13.10s	1.80s	15	399.5mb	18.5mb	24.0mb
124.40s	12.61s	1.89s	12	388.0mb	20.0mb	26.0mb
`5earley:12` **3GEN upper+lower T=3.0**						
14.33s	4.27s	2.43s	4	158.0mb	20.0mb	28.0mb
14.58s	4.55s	2.51s	4	163.5mb	22.5mb	31.0mb
14.88s	4.88s	2.47s	4	173.0mb	27.5mb	34.0mb
15.05s	5.08s	2.53s	4	178.5mb	31.5mb	37.0mb
15.44s	5.46s	2.44s	4	187.5mb	38.0mb	40.0mb
`5nboyer:3` **3GEN upper+lower T=3.0**						
						24.0mb
						28.0mb
37.30s	14.51s	6.29s	14	512.5mb	32.0mb	32.0mb
37.87s	14.87s	6.51s	13	526.0mb	35.0mb	36.0mb
36.38s	13.65s	6.77s	9	483.0mb	36.5mb	40.0mb
`5sboyer:4` **3GEN upper+lower T=3.0**						
						20.0mb
						24.0mb
71.42s	4.44s	2.65s	4	158.0mb	18.0mb	28.0mb
71.89s	4.73s	2.64s	4	168.0mb	24.0mb	32.0mb
73.22s	5.06s	2.61s	4	178.5mb	29.5mb	36.0mb
`5twobitshort` **3GEN upper+lower T=3.0**						
124.79s	13.35s	1.70s	17	407.0mb	18.5mb	18.0mb
123.75s	13.06s	1.73s	15	399.0mb	18.5mb	20.0mb
123.61s	12.40s	2.02s	11	382.5mb	20.5mb	22.0mb
123.93s	12.48s	1.99s	9	377.5mb	21.0mb	24.0mb
123.45s	12.09s	2.07s	7	370.0mb	23.0mb	26.0mb

Table A.17: Raw data for 3GEN: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **3GEN upper T=3.0**						
						26.0mb
47.63s	25.71s	2.61s	48	898.0mb	27.5mb	29.0mb
46.48s	24.49s	2.85s	43	856.5mb	30.0mb	32.0mb
43.21s	21.22s	4.27s	27	724.0mb	32.5mb	35.0mb
41.70s	19.57s	4.81s	20	671.0mb	32.5mb	38.0mb
`gcold:1,0` **3GEN upper T=3.0**						
29.81s	20.57s	6.24s	13	594.5mb	53.0mb	56.0mb
26.85s	17.70s	6.43s	9	524.0mb	58.0mb	61.0mb
25.47s	16.27s	6.45s	7	487.5mb	64.0mb	66.0mb
25.50s	16.14s	6.40s	7	487.5mb	64.0mb	71.0mb
25.50s	16.28s	6.47s	7	487.5mb	64.0mb	76.0mb
`gcold:1,1000` **3GEN upper T=3.0**						
66.23s	22.00s	5.97s	13	651.5mb	53.0mb	56.0mb
63.53s	19.36s	6.26s	9	581.5mb	58.0mb	61.0mb
62.33s	17.98s	6.37s	7	546.5mb	62.5mb	66.0mb
62.33s	17.90s	6.37s	7	546.5mb	62.5mb	71.0mb
62.41s	17.93s	6.35s	7	546.5mb	62.5mb	76.0mb
`gcold:10,1000` **3GEN upper T=3.0**						
38.85s	2.44s	0.75s	1	77.5mb	44.0mb	56.0mb
38.84s	2.43s	0.71s	1	77.5mb	44.0mb	61.0mb
38.84s	2.64s	0.91s	1	77.5mb	44.0mb	66.0mb
38.83s	2.61s	0.84s	1	77.5mb	44.0mb	71.0mb
38.84s	2.49s	0.72s	1	77.5mb	44.0mb	76.0mb
`nboyer:3` **3GEN upper T=3.0**						
						24.0mb
7.89s	3.32s	1.10s	6	115.0mb	27.5mb	28.0mb
7.65s	3.08s	1.15s	5	108.5mb	31.5mb	32.0mb
7.68s	3.10s	1.16s	5	108.5mb	33.5mb	36.0mb
7.68s	3.12s	1.17s	5	108.5mb	33.5mb	40.0mb
`perm:200,1` **3GEN upper T=3.0**						
						25.0mb
						28.0mb
						31.0mb
25.69s	19.49s	5.41s	28	676.0mb	32.5mb	34.0mb
23.51s	17.40s	6.29s	19	607.0mb	36.0mb	37.0mb
`perm:25,8` **3GEN upper T=3.0**						
20.73s	14.62s	5.13s	33	534.5mb	24.5mb	25.0mb
						28.0mb
						31.0mb
23.24s	17.12s	5.74s	25	601.5mb	33.0mb	34.0mb
20.61s	14.51s	6.01s	24	523.5mb	33.0mb	37.0mb
`twobitlong` **3GEN upper T=3.0**						
146.32s	6.52s	4.72s	5	108.0mb	18.5mb	18.0mb
145.48s	6.38s	4.76s	4	103.0mb	18.5mb	20.0mb
146.04s	6.48s	4.87s	4	103.0mb	19.0mb	22.0mb
145.71s	6.29s	4.71s	4	103.0mb	19.0mb	24.0mb
146.14s	6.35s	4.77s	3	101.5mb	18.5mb	26.0mb
`twobitshort` **3GEN upper T=3.0**						
24.63s	2.86s	0.66s	4	80.5mb	18.5mb	18.0mb
24.54s	2.70s	0.57s	4	80.0mb	20.0mb	20.0mb
25.19s	2.74s	0.67s	4	78.5mb	20.0mb	22.0mb
25.10s	2.60s	0.58s	3	76.0mb	20.0mb	24.0mb
25.00s	2.50s	0.63s	3	74.5mb	20.0mb	26.0mb

Table A.18: Raw data for 3GEN: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
5earley:12 **3GEN upper T=3.0**						
15.01s	4.73s	2.25s	10	172.0mb	17.5mb	28.0mb
15.01s	4.75s	2.34s	10	172.0mb	17.5mb	31.0mb
15.01s	4.83s	2.37s	10	172.0mb	17.5mb	34.0mb
15.02s	4.69s	2.25s	10	172.0mb	17.5mb	37.0mb
15.02s	4.72s	2.25s	10	172.0mb	17.5mb	40.0mb
5nboyer:3 **3GEN upper T=3.0**						
						24.0mb
41.67s	18.94s	5.22s	38	647.0mb	27.5mb	28.0mb
38.49s	15.53s	5.72s	25	552.0mb	31.5mb	32.0mb
37.51s	14.68s	6.04s	20	520.0mb	33.5mb	36.0mb
37.53s	14.55s	5.98s	18	522.0mb	33.5mb	40.0mb
5sboyer:4 **3GEN upper T=3.0**						
						20.0mb
						24.0mb
72.94s	6.42s	2.24s	15	219.5mb	25.5mb	28.0mb
72.27s	5.87s	2.33s	10	202.5mb	26.0mb	32.0mb
72.29s	5.74s	2.38s	9	199.0mb	28.5mb	36.0mb
5twobitshort **3GEN upper T=3.0**						
123.71s	13.53s	1.68s	18	407.0mb	18.5mb	18.0mb
122.56s	12.83s	1.72s	16	399.0mb	20.0mb	20.0mb
122.79s	12.34s	1.95s	12	382.5mb	20.5mb	22.0mb
122.80s	12.40s	1.82s	11	383.5mb	21.0mb	24.0mb
123.17s	12.07s	2.04s	9	373.0mb	23.0mb	26.0mb

Table A.19: Raw data for 3GEN: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
5earley:12 **3GEN variable**						
17.89s	7.51s	1.72s	17	252.5mb	26.0mb	2.25
15.88s	5.68s	2.18s	15	197.5mb	17.5mb	2.5
16.39s	6.10s	2.13s	7	205.0mb	38.5mb	2.75
15.08s	4.78s	2.30s	10	173.5mb	17.5mb	3.0
5nboyer:3 **3GEN variable**						
41.47s	18.73s	4.66s	44	635.5mb	29.0mb	2.25
39.58s	16.64s	5.21s	30	580.0mb	31.0mb	2.5
38.40s	15.47s	5.79s	22	546.0mb	34.0mb	2.75
37.69s	15.02s	6.23s	17	524.5mb	36.0mb	3.0
5sboyer:4 **3GEN variable**						
73.88s	7.33s	2.03s	21	245.0mb	22.5mb	2.25
73.00s	6.15s	2.22s	14	212.5mb	23.5mb	2.5
72.25s	5.86s	2.45s	9	200.0mb	35.0mb	2.75
72.33s	5.77s	2.42s	9	196.0mb	29.5mb	3.0
5twobitshort **3GEN variable**						
123.49s	12.62s	1.77s	14	391.5mb	20.0mb	2.25
122.89s	12.17s	2.01s	10	378.5mb	22.0mb	2.5
122.81s	12.01s	2.05s	8	372.5mb	24.0mb	2.75
123.56s	11.84s	2.11s	6	369.0mb	27.5mb	3.0

Table A.20: Raw data for 3GEN: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
`gcbench:5` **3GEN variable**						
45.05s	22.91s	3.88s	35	787.5mb	29.5mb	2.25
42.57s	20.68s	4.65s	24	701.0mb	32.5mb	2.5
41.21s	19.20s	5.00s	18	653.5mb	36.0mb	2.75
40.21s	18.12s	5.18s	16	622.5mb	38.5mb	3.0
`gcold:1,0` **3GEN variable**						
36.33s	26.93s	5.29s	23	765.0mb	48.5mb	2.25
29.68s	20.38s	6.02s	13	592.0mb	53.0mb	2.5
26.94s	17.71s	6.42s	9	524.0mb	58.0mb	2.75
25.49s	16.13s	6.32s	7	487.5mb	64.0mb	3.0
`gcold:1,1000` **3GEN variable**						
74.21s	30.15s	5.46s	25	861.5mb	47.0mb	2.25
66.42s	22.24s	6.10s	13	651.5mb	53.0mb	2.5
63.47s	19.42s	6.30s	9	581.5mb	58.0mb	2.75
62.27s	17.76s	6.31s	7	546.5mb	62.5mb	3.0
`gcold:10,1000` **3GEN variable**						
40.22s	3.87s	0.62s	3	113.5mb	46.5mb	2.25
39.59s	3.35s	0.83s	2	95.5mb	52.0mb	2.5
38.87s	2.40s	0.68s	1	77.5mb	47.5mb	2.75
38.83s	2.58s	0.83s	1	77.5mb	44.0mb	3.0
`nboyer:3` **3GEN variable**						
8.18s	3.56s	0.88s	9	123.5mb	27.0mb	2.25
7.90s	3.30s	1.03s	7	115.5mb	29.0mb	2.5
7.84s	3.27s	1.11s	6	113.5mb	33.5mb	2.75
7.66s	3.10s	1.11s	5	108.5mb	33.5mb	3.0
`perm:200,1` **3GEN variable**						
27.88s	21.71s	4.28s	38	757.5mb	30.0mb	2.25
25.33s	19.23s	5.60s	26	663.0mb	33.0mb	2.5
23.54s	17.33s	6.27s	19	608.5mb	35.5mb	2.75
22.45s	16.44s	6.74s	15	577.5mb	39.0mb	3.0
`perm:25,8` **3GEN variable**						
22.36s	16.23s	3.61s	50	588.5mb	27.0mb	2.25
24.06s	17.80s	5.44s	27	627.0mb	32.5mb	2.5
21.33s	15.31s	5.84s	25	548.5mb	29.0mb	2.75
20.88s	14.74s	6.18s	21	532.0mb	35.0mb	3.0
`twobitlong` **3GEN variable**						
145.78s	6.57s	4.94s	4	102.0mb	18.5mb	2.25
145.45s	6.49s	4.96s	4	101.0mb	17.0mb	2.5
146.25s	6.30s	4.82s	3	99.5mb	23.0mb	2.75
146.41s	6.29s	4.88s	2	98.5mb	23.0mb	3.0
`twobitshort` **3GEN variable**						
25.06s	2.64s	0.69s	3	75.5mb	19.5mb	2.25
24.43s	2.58s	0.62s	3	75.0mb	22.0mb	2.5
25.15s	2.47s	0.73s	2	72.5mb	19.5mb	2.75
24.95s	2.53s	0.73s	2	72.5mb	20.0mb	3.0

Table A.21: Raw data for 3GEN: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **1DOF upper+lower T=2.0**								
								26.0mb
								29.0mb
60.14s	38.09s	13.38s	8.83s	70	69	1485mb	26.5mb	32.0mb
51.45s	29.34s	13.38s	0.46s	65	4	1121mb	28.5mb	35.0mb
49.22s	26.83s	13.11s	0.00s	55	0	1014mb	30.5mb	38.0mb
`gcold:1,0` **1DOF upper+lower T=2.0**								
30.84s	21.74s	4.56s	0.00s	26	0	647.5mb	46.5mb	56.0mb
26.01s	16.91s	4.72s	0.00s	18	0	496.5mb	49.0mb	61.0mb
23.56s	14.54s	4.74s	0.00s	14	0	418.0mb	51.0mb	66.0mb
21.79s	12.75s	4.79s	0.00s	11	0	362.0mb	54.5mb	71.0mb
20.69s	11.55s	4.82s	0.00s	9	0	323.5mb	56.5mb	76.0mb
`gcold:1,1000` **1DOF upper+lower T=2.0**								
65.10s	21.20s	4.61s	0.00s	25	0	629.0mb	47.0mb	56.0mb
60.94s	17.14s	4.69s	0.00s	18	0	496.5mb	49.0mb	61.0mb
58.51s	14.62s	4.73s	0.00s	14	0	419.0mb	51.0mb	66.0mb
56.87s	13.10s	4.95s	0.00s	11	0	362.5mb	54.5mb	71.0mb
55.58s	11.68s	4.86s	0.00s	9	0	324.0mb	56.5mb	76.0mb
`gcold:10,1000` **1DOF upper+lower T=2.0**								
38.43s	2.43s	0.70s	0.00s	2	0	53.5mb	46.0mb	56.0mb
38.46s	2.28s	0.55s	0.00s	2	0	53.5mb	48.5mb	61.0mb
38.51s	2.31s	0.53s	0.00s	2	0	53.5mb	51.0mb	66.0mb
37.70s	1.31s	0.40s	0.00s	1	0	34.5mb	53.5mb	71.0mb
37.71s	1.39s	0.48s	0.00s	1	0	34.5mb	56.0mb	76.0mb
`nboyer:3` **1DOF upper+lower T=2.0**								
								24.0mb
								28.0mb
8.85s	4.31s	1.59s	0.00s	8	0	115.0mb	26.5mb	32.0mb
7.99s	3.46s	1.63s	0.00s	5	0	87.0mb	28.5mb	36.0mb
7.19s	2.68s	1.69s	0.00s	3	0	66.5mb	29.0mb	40.0mb
`perm:200,1` **1DOF upper+lower T=2.0**								
								25.0mb
								28.0mb
58.76s	52.32s	6.19s	0.00s	137	0	1732mb	26.5mb	31.0mb
35.41s	29.00s	6.15s	0.00s	67	0	953.5mb	28.5mb	34.0mb
28.84s	22.71s	6.33s	0.00s	47	0	738.5mb	30.0mb	37.0mb
`perm:25,8` **1DOF upper+lower T=2.0**								
								25.0mb
								28.0mb
15.41s	9.26s	6.48s	0.00s	24	0	314.0mb	19.5mb	31.0mb
16.57s	10.47s	6.49s	0.00s	24	0	349.0mb	22.5mb	34.0mb
17.80s	11.63s	6.42s	0.00s	24	0	386.0mb	25.5mb	37.0mb
`twobitlong` **1DOF upper+lower T=2.0**								
								18.0mb
145.06s	6.20s	4.51s	0.00s	13	0	104.0mb	18.0mb	20.0mb
144.19s	5.70s	4.38s	0.00s	10	0	90.5mb	19.0mb	22.0mb
143.94s	5.19s	4.36s	0.00s	7	0	75.0mb	18.0mb	24.0mb
143.04s	5.00s	4.32s	0.00s	6	0	71.0mb	18.0mb	26.0mb
`twobitshort` **1DOF upper+lower T=2.0**								
25.37s	3.29s	1.00s	0.00s	13	0	86.5mb	16.5mb	18.0mb
23.71s	2.10s	1.01s	0.00s	6	0	48.5mb	17.0mb	20.0mb
23.46s	1.96s	1.09s	0.00s	4	0	42.0mb	18.0mb	22.0mb
23.32s	1.78s	1.06s	0.00s	3	0	38.0mb	20.0mb	24.0mb
22.99s	1.52s	1.10s	0.00s	2	0	31.5mb	20.0mb	26.0mb

Table A.22: Raw data for 1DOF: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **1DOF upper+lower T=3.0**								
								26.0mb
91.99s	69.82s	13.35s	29.83s	104	219	2796mb	27.5mb	29.0mb
73.64s	51.38s	13.28s	21.75s	70	166	2053mb	27.0mb	32.0mb
69.34s	47.10s	13.34s	18.37s	61	135	1862mb	30.5mb	35.0mb
57.80s	35.55s	13.19s	10.45s	50	79	1386mb	30.5mb	38.0mb
`gcold:1,0` **1DOF upper+lower T=3.0**								
28.53s	19.43s	4.63s	0.00s	22	0	570.5mb	47.5mb	56.0mb
24.75s	15.67s	4.61s	0.00s	16	0	456.5mb	50.0mb	61.0mb
22.99s	13.90s	4.71s	0.00s	13	0	399.5mb	52.5mb	66.0mb
21.24s	12.22s	4.85s	0.00s	10	0	342.5mb	55.0mb	71.0mb
20.68s	11.68s	4.96s	0.00s	9	0	324.0mb	57.5mb	76.0mb
`gcold:1,1000` **1DOF upper+lower T=3.0**								
63.30s	19.37s	4.53s	0.00s	22	0	572.0mb	47.5mb	56.0mb
59.63s	15.71s	4.59s	0.00s	16	0	457.0mb	50.0mb	61.0mb
57.89s	13.94s	4.65s	0.00s	13	0	400.5mb	53.0mb	66.0mb
56.22s	12.41s	4.89s	0.00s	10	0	343.5mb	55.0mb	71.0mb
55.71s	11.77s	4.90s	0.00s	9	0	324.5mb	57.5mb	76.0mb
`gcold:10,1000` **1DOF upper+lower T=3.0**								
38.44s	2.17s	0.39s	0.00s	2	0	53.5mb	46.5mb	56.0mb
38.47s	2.26s	0.54s	0.00s	2	0	53.5mb	48.5mb	61.0mb
38.52s	2.36s	0.58s	0.00s	2	0	53.5mb	51.0mb	66.0mb
37.72s	1.49s	0.59s	0.00s	1	0	34.5mb	53.5mb	71.0mb
37.73s	1.49s	0.57s	0.00s	1	0	34.5mb	56.0mb	76.0mb
`nboyer:3` **1DOF upper+lower T=3.0**								
								24.0mb
10.02s	5.43s	1.54s	0.00s	12	0	150.5mb	26.0mb	28.0mb
8.08s	3.42s	1.47s	0.00s	6	0	91.5mb	27.0mb	32.0mb
8.04s	3.42s	1.54s	0.00s	5	0	88.0mb	29.5mb	36.0mb
7.19s	2.56s	1.57s	0.00s	3	0	66.5mb	29.0mb	40.0mb
`perm:200,1` **1DOF upper+lower T=3.0**								
								25.0mb
77.11s	70.64s	6.31s	0.00s	194	0	2348mb	26.5mb	28.0mb
41.53s	35.18s	6.24s	0.00s	85	0	1158mb	27.5mb	31.0mb
31.96s	25.80s	6.33s	0.00s	56	0	842.0mb	29.0mb	34.0mb
26.65s	20.46s	6.36s	0.00s	40	0	666.0mb	31.0mb	37.0mb
`perm:25,8` **1DOF upper+lower T=3.0**								
								25.0mb
15.48s	9.36s	6.31s	0.00s	26	0	317.0mb	24.5mb	28.0mb
15.41s	9.33s	6.48s	0.00s	24	0	314.0mb	19.5mb	31.0mb
16.57s	10.45s	6.46s	0.00s	24	0	349.0mb	22.5mb	34.0mb
17.80s	11.71s	6.49s	0.00s	24	0	386.0mb	25.5mb	37.0mb
`twobitlong` **1DOF upper+lower T=3.0**								
144.88s	6.22s	4.15s	0.00s	16	0	117.0mb	17.5mb	18.0mb
144.17s	5.78s	4.34s	0.00s	11	0	94.5mb	17.5mb	20.0mb
143.56s	5.23s	4.14s	0.00s	9	0	83.5mb	19.0mb	22.0mb
143.83s	5.34s	4.53s	0.00s	7	0	75.0mb	18.0mb	24.0mb
143.53s	5.21s	4.51s	0.00s	6	0	71.0mb	18.0mb	26.0mb
`twobitshort` **1DOF upper+lower T=3.0**								
23.82s	2.33s	0.96s	0.00s	7	0	56.0mb	17.0mb	18.0mb
23.50s	2.03s	0.98s	0.00s	5	0	46.5mb	17.5mb	20.0mb
23.38s	1.92s	1.07s	0.00s	4	0	41.5mb	18.0mb	22.0mb
23.56s	1.81s	1.09s	0.00s	3	0	38.5mb	20.0mb	24.0mb
22.98s	1.60s	1.19s	0.00s	2	0	31.5mb	20.0mb	26.0mb

Table A.23: Raw data for 1DOF: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **1DOF upper+lower T=2.0**								
								28.0mb
14.96s	5.03s	2.25s	0.00s	9	0	124.0mb	27.5mb	31.0mb
12.80s	2.60s	2.39s	0.00s	4	0	68.0mb	17.5mb	34.0mb
12.59s	2.82s	2.36s	0.00s	4	0	74.0mb	20.5mb	37.0mb
12.88s	3.12s	2.39s	0.00s	4	0	79.5mb	23.5mb	40.0mb
`5nboyer:3` **1DOF upper+lower T=2.0**								
								24.0mb
								28.0mb
42.71s	19.86s	6.88s	0.00s	44	0	579.0mb	26.5mb	32.0mb
38.19s	15.39s	6.95s	0.00s	29	0	446.0mb	29.0mb	36.0mb
34.63s	11.59s	6.92s	0.00s	19	0	340.0mb	30.0mb	40.0mb
`5sboyer:4` **1DOF upper+lower T=2.0**								
								20.0mb
								24.0mb
								28.0mb
72.82s	5.20s	2.44s	0.00s	9	0	127.5mb	27.0mb	32.0mb
70.43s	2.77s	2.52s	0.00s	4	0	72.5mb	19.0mb	36.0mb
`5twobitshort` **1DOF upper+lower T=2.0**								
123.66s	14.21s	4.11s	0.00s	61	0	397.0mb	16.5mb	18.0mb
115.85s	8.75s	3.75s	0.00s	31	0	240.0mb	17.0mb	20.0mb
115.47s	8.29s	4.10s	0.00s	24	0	210.5mb	18.5mb	22.0mb
113.93s	7.19s	4.09s	0.00s	18	0	180.5mb	20.0mb	24.0mb
112.36s	6.36s	4.15s	0.00s	14	0	157.5mb	20.5mb	26.0mb
`5earley:12` **1DOF upper+lower T=3.0**								
								28.0mb
14.97s	5.04s	2.27s	0.00s	9	0	124.0mb	27.5mb	31.0mb
12.81s	2.61s	2.41s	0.00s	4	0	68.0mb	17.5mb	34.0mb
12.58s	2.83s	2.38s	0.00s	4	0	74.0mb	20.5mb	37.0mb
12.90s	3.04s	2.32s	0.00s	4	0	79.5mb	23.5mb	40.0mb
`5nboyer:3` **1DOF upper+lower T=3.0**								
								24.0mb
50.28s	27.22s	6.77s	0.00s	66	0	794.0mb	26.0mb	28.0mb
38.99s	15.99s	6.72s	0.00s	33	0	468.5mb	27.5mb	32.0mb
37.02s	14.00s	6.77s	0.00s	25	0	406.0mb	30.0mb	36.0mb
34.82s	11.96s	6.87s	0.00s	19	0	349.5mb	31.0mb	40.0mb
`5sboyer:4` **1DOF upper+lower T=3.0**								
								20.0mb
								24.0mb
								28.0mb
72.87s	5.25s	2.49s	0.00s	9	0	127.5mb	27.0mb	32.0mb
70.37s	2.78s	2.52s	0.00s	4	0	72.5mb	19.0mb	36.0mb
`5twobitshort` **1DOF upper+lower T=3.0**								
116.19s	9.59s	4.09s	0.00s	34	0	253.5mb	17.0mb	18.0mb
115.84s	8.41s	4.07s	0.00s	25	0	215.0mb	18.0mb	20.0mb
114.20s	7.47s	4.01s	0.00s	20	0	190.5mb	19.0mb	22.0mb
114.12s	7.24s	4.23s	0.00s	17	0	177.5mb	20.0mb	24.0mb
112.36s	6.25s	4.02s	0.00s	14	0	157.5mb	20.5mb	26.0mb

Table A.24: Raw data for 1DOF: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **1DOF upper T=3.0**								
								26.0mb
93.80s	71.09s	12.86s	29.91s	118	219	2849mb	27.5mb	29.0mb
74.99s	52.57s	13.15s	21.01s	85	159	2078mb	27.0mb	32.0mb
70.02s	47.82s	13.25s	17.05s	76	126	1866mb	30.5mb	35.0mb
54.23s	31.97s	13.12s	4.60s	67	35	1219mb	30.5mb	38.0mb
`gcold:1,0` **1DOF upper T=3.0**								
28.49s	19.34s	4.37s	0.00s	23	0	590.0mb	47.5mb	56.0mb
25.55s	16.46s	4.47s	0.00s	18	0	495.5mb	50.0mb	61.0mb
23.01s	13.97s	4.48s	0.00s	14	0	419.0mb	53.0mb	66.0mb
21.97s	12.94s	4.50s	0.00s	12	0	380.5mb	55.0mb	71.0mb
21.41s	12.37s	4.54s	0.00s	11	0	362.0mb	57.5mb	76.0mb
`gcold:1,1000` **1DOF upper T=3.0**								
63.42s	19.65s	4.62s	0.00s	23	0	591.0mb	47.5mb	56.0mb
60.36s	16.46s	4.48s	0.00s	18	0	495.5mb	50.0mb	61.0mb
58.02s	14.03s	4.36s	0.00s	14	0	419.0mb	52.5mb	66.0mb
56.83s	12.99s	4.58s	0.00s	12	0	381.0mb	55.0mb	71.0mb
56.32s	12.59s	4.65s	0.00s	11	0	362.0mb	57.5mb	76.0mb
`gcold:10,1000` **1DOF upper T=3.0**								
38.93s	2.81s	0.54s	0.00s	3	0	72.5mb	46.0mb	56.0mb
38.22s	2.00s	0.46s	0.00s	2	0	53.5mb	46.5mb	61.0mb
38.26s	2.16s	0.59s	0.00s	2	0	53.5mb	47.5mb	66.0mb
38.23s	2.03s	0.46s	0.00s	2	0	53.5mb	47.5mb	71.0mb
38.26s	2.22s	0.65s	0.00s	2	0	53.5mb	47.5mb	76.0mb
`nboyer:3` **1DOF upper T=3.0**								
								24.0mb
11.54s	6.90s	1.35s	0.00s	31	0	196.5mb	26.0mb	28.0mb
10.16s	5.59s	1.41s	0.00s	27	0	155.0mb	27.5mb	32.0mb
9.84s	5.24s	1.33s	0.00s	26	0	144.5mb	29.0mb	36.0mb
9.43s	4.72s	1.26s	0.00s	25	0	134.0mb	27.0mb	40.0mb
`perm:200,1` **1DOF upper T=3.0**								
								25.0mb
76.45s	69.94s	6.29s	0.00s	193	0	2336mb	26.5mb	28.0mb
41.47s	35.30s	6.22s	0.00s	86	0	1168mb	27.5mb	31.0mb
31.98s	25.82s	6.19s	0.00s	57	0	854.0mb	29.0mb	34.0mb
26.69s	20.60s	6.27s	0.00s	41	0	675.5mb	31.0mb	37.0mb
`perm:25,8` **1DOF upper T=3.0**								
								25.0mb
17.03s	11.02s	6.33s	0.00s	34	0	366.5mb	25.0mb	28.0mb
16.58s	10.44s	6.23s	0.00s	32	0	355.0mb	22.5mb	31.0mb
16.68s	10.51s	6.20s	0.00s	32	0	358.0mb	22.5mb	34.0mb
16.67s	10.62s	6.27s	0.00s	32	0	358.0mb	22.5mb	37.0mb
`twobitlong` **1DOF upper T=3.0**								
146.34s	7.14s	4.26s	0.00s	41	0	145.5mb	17.5mb	18.0mb
145.88s	6.75s	4.24s	0.00s	38	0	130.5mb	18.5mb	20.0mb
145.59s	6.68s	4.39s	0.00s	36	0	121.0mb	18.5mb	22.0mb
145.77s	6.54s	4.26s	0.00s	36	0	121.0mb	18.5mb	24.0mb
146.02s	6.61s	4.35s	0.00s	36	0	121.0mb	18.5mb	26.0mb
`twobitshort` **1DOF upper T=3.0**								
25.03s	3.23s	0.97s	0.00s	21	0	84.5mb	17.0mb	18.0mb
25.22s	2.96s	0.96s	0.00s	20	0	77.0mb	17.0mb	20.0mb
24.80s	2.86s	0.95s	0.00s	19	0	72.5mb	17.5mb	22.0mb
24.89s	2.76s	0.95s	0.00s	19	0	71.0mb	17.5mb	24.0mb
24.88s	2.73s	0.96s	0.00s	19	0	71.0mb	17.5mb	26.0mb

Table A.25: Raw data for 1DOF: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **1DOF upper T=3.0**								
								28.0mb
16.86s	6.89s	2.10s	0.00s	25	0	178.0mb	27.5mb	31.0mb
16.48s	6.56s	2.17s	0.00s	24	0	167.5mb	27.5mb	34.0mb
16.50s	6.69s	2.26s	0.00s	24	0	167.5mb	27.5mb	37.0mb
16.47s	6.52s	2.09s	0.00s	24	0	167.5mb	27.5mb	40.0mb
`5nboyer:3` **1DOF upper T=3.0**								
								24.0mb
51.65s	28.58s	6.59s	0.00s	85	0	840.5mb	26.0mb	28.0mb
41.08s	18.10s	6.54s	0.00s	54	0	532.5mb	27.5mb	32.0mb
38.73s	15.73s	6.56s	0.00s	46	0	461.0mb	30.0mb	36.0mb
36.53s	13.68s	6.75s	0.00s	41	0	399.0mb	29.5mb	40.0mb
`5sboyer:4` **1DOF upper T=3.0**								
								20.0mb
								24.0mb
								28.0mb
74.11s	6.57s	2.22s	0.00s	25	0	167.0mb	25.5mb	32.0mb
74.33s	6.71s	2.16s	0.00s	25	0	172.5mb	27.0mb	36.0mb
`5twobitshort` **1DOF upper T=3.0**								
117.56s	10.33s	3.93s	0.00s	48	0	281.5mb	17.0mb	18.0mb
117.09s	9.14s	3.94s	0.00s	39	0	241.0mb	18.0mb	20.0mb
115.13s	8.38s	4.07s	0.00s	35	0	217.5mb	19.0mb	22.0mb
115.26s	8.40s	4.23s	0.00s	34	0	212.0mb	19.5mb	24.0mb
115.29s	8.15s	4.00s	0.00s	34	0	212.0mb	19.5mb	26.0mb

Table A.26: Raw data for 1DOF: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`5earley:12` **1DOF variable**								
23.10s	12.73s	1.99s	0.00s	75	0	349.0mb	27.5mb	2.25
17.21s	7.13s	2.14s	0.00s	38	0	192.0mb	26.5mb	2.5
17.58s	7.39s	2.16s	0.00s	30	0	189.0mb	27.5mb	2.75
14.81s	4.58s	1.97s	0.00s	21	0	128.0mb	26.0mb	3.0
`5nboyer:3` **1DOF variable**								
51.39s	28.37s	6.73s	0.00s	120	0	822.0mb	27.5mb	2.25
42.79s	19.91s	6.81s	0.00s	71	0	575.0mb	29.0mb	2.5
38.78s	15.72s	6.54s	0.00s	51	0	461.0mb	30.0mb	2.75
36.65s	13.70s	6.72s	0.00s	41	0	399.0mb	29.5mb	3.0
`5sboyer:4` **1DOF variable**								
81.24s	13.27s	2.25s	0.00s	79	0	351.5mb	26.5mb	2.25
77.09s	8.94s	2.09s	0.00s	44	0	235.5mb	28.0mb	2.5
73.16s	5.62s	2.18s	0.00s	27	0	151.5mb	26.0mb	2.75
74.71s	6.73s	2.13s	0.00s	25	0	172.5mb	27.0mb	3.0
`5twobitshort` **1DOF variable**								
123.94s	14.99s	4.29s	0.00s	106	0	425.5mb	16.5mb	2.25
117.54s	10.80s	4.09s	0.00s	61	0	295.5mb	17.0mb	2.5
117.62s	9.15s	3.91s	0.00s	43	0	246.0mb	18.5mb	2.75
115.31s	8.07s	3.91s	0.00s	34	0	212.0mb	19.5mb	3.0

Table A.27: Raw data for 1DOF: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`gcbench:5` **1DOF variable**								
66.95s	44.58s	13.20s	0.00s	153	0	1682mb	27.5mb	2.25
55.37s	32.96s	13.10s	0.00s	99	0	1244mb	26.5mb	2.5
52.93s	30.79s	13.39s	0.00s	85	0	1146mb	26.5mb	2.75
49.72s	27.60s	13.05s	0.00s	67	0	1035mb	30.5mb	3.0
`gcold:1,0` **1DOF variable**								
36.61s	27.61s	4.50s	0.00s	36	0	837.0mb	45.5mb	2.25
25.97s	16.70s	4.27s	0.00s	19	0	514.0mb	50.5mb	2.5
22.45s	13.40s	4.53s	0.00s	13	0	400.0mb	55.0mb	2.75
21.49s	12.27s	4.40s	0.00s	11	0	361.5mb	59.0mb	3.0
`gcold:1,1000` **1DOF variable**								
70.81s	26.88s	4.42s	0.00s	35	0	820.5mb	46.0mb	2.25
60.95s	16.83s	4.38s	0.00s	19	0	515.0mb	50.5mb	2.5
57.44s	13.53s	4.50s	0.00s	13	0	400.0mb	55.0mb	2.75
56.48s	12.68s	4.65s	0.00s	11	0	362.5mb	59.5mb	3.0
`gcold:10,1000` **1DOF variable**								
40.75s	4.53s	0.48s	0.00s	6	0	129.5mb	44.0mb	2.25
38.79s	2.61s	0.46s	0.00s	3	0	72.5mb	46.0mb	2.5
38.15s	2.02s	0.55s	0.00s	2	0	53.5mb	46.0mb	2.75
38.26s	2.02s	0.46s	0.00s	2	0	53.5mb	47.5mb	3.0
`nboyer:3` **1DOF variable**								
17.41s	12.70s	1.31s	0.00s	85	0	367.0mb	26.0mb	2.25
12.49s	7.85s	1.32s	0.00s	47	0	223.0mb	27.0mb	2.5
10.73s	6.10s	1.34s	0.00s	33	0	170.0mb	28.5mb	2.75
9.40s	4.82s	1.38s	0.00s	25	0	134.0mb	27.0mb	3.0
`perm:200,1` **1DOF variable**								
38.71s	32.46s	6.18s	0.00s	78	0	1076mb	28.0mb	2.25
27.19s	20.96s	6.16s	0.00s	43	0	695.5mb	31.0mb	2.5
23.44s	17.27s	6.15s	0.00s	31	0	567.0mb	34.0mb	2.75
20.79s	14.60s	6.28s	0.00s	23	0	477.5mb	36.5mb	3.0
`perm:25,8` **1DOF variable**								
21.72s	15.65s	6.28s	0.00s	57	0	514.0mb	25.0mb	2.25
18.31s	12.18s	6.20s	0.00s	40	0	410.0mb	24.5mb	2.5
17.75s	11.61s	6.13s	0.00s	34	0	392.0mb	24.5mb	2.75
16.66s	10.62s	6.26s	0.00s	32	0	358.0mb	22.5mb	3.0
`twobitlong` **1DOF variable**								
149.41s	10.49s	4.19s	0.00s	114	0	274.0mb	17.5mb	2.25
146.73s	7.93s	4.03s	0.00s	65	0	180.5mb	18.0mb	2.5
145.91s	7.03s	4.30s	0.00s	46	0	141.5mb	17.5mb	2.75
144.97s	6.69s	4.44s	0.00s	36	0	121.0mb	18.5mb	3.0
`twobitshort` **1DOF variable**								
28.38s	6.17s	0.97s	0.00s	71	0	184.5mb	16.5mb	2.25
25.74s	3.89s	0.95s	0.00s	36	0	109.5mb	17.0mb	2.5
25.13s	3.04s	0.85s	0.00s	24	0	83.0mb	17.5mb	2.75
24.82s	2.80s	1.01s	0.00s	19	0	71.0mb	17.5mb	3.0

Table A.28: Raw data for 1DOF: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **2DOF upper+lower T=2.0**								
								26.0mb
48.63s	26.60s	13.85s	2.50s	92	18	903.0mb	28.5mb	29.0mb
44.88s	22.99s	13.85s	6.63s	56	51	830.0mb	29.5mb	32.0mb
41.54s	19.71s	14.01s	0.51s	60	4	662.5mb	34.0mb	35.0mb
40.96s	18.91s	13.87s	0.51s	52	4	638.5mb	37.0mb	38.0mb
`gcold:1,0` **2DOF upper+lower T=2.0**								
21.40s	12.46s	4.75s	0.00s	22	0	351.0mb	50.5mb	56.0mb
20.66s	11.47s	4.60s	0.00s	18	0	327.0mb	55.5mb	61.0mb
17.81s	8.97s	5.13s	0.00s	12	0	249.5mb	56.5mb	66.0mb
16.40s	7.46s	5.12s	0.00s	9	0	209.5mb	62.5mb	71.0mb
16.25s	7.33s	5.28s	0.00s	8	0	201.0mb	64.5mb	76.0mb
`gcold:1,1000` **2DOF upper+lower T=2.0**								
62.03s	18.12s	4.73s	0.00s	32	0	516.5mb	50.5mb	56.0mb
60.61s	16.68s	4.80s	0.00s	26	0	470.5mb	55.5mb	61.0mb
59.74s	15.85s	4.93s	0.00s	22	0	439.0mb	60.0mb	66.0mb
59.01s	15.10s	4.95s	0.00s	19	0	412.5mb	65.5mb	71.0mb
57.69s	13.67s	4.98s	0.00s	17	0	358.0mb	71.0mb	76.0mb
`gcold:10,1000` **2DOF upper+lower T=2.0**								
38.01s	1.61s	0.57s	0.00s	2	0	37.0mb	47.0mb	56.0mb
37.61s	1.23s	0.59s	0.00s	1	0	27.5mb	51.5mb	61.0mb
37.75s	1.42s	0.71s	0.00s	1	0	29.0mb	56.0mb	66.0mb
37.04s	0.71s	0.71s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.04s	0.74s	0.74s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **2DOF upper+lower T=2.0**								
9.42s	4.73s	1.55s	0.00s	18	0	116.5mb	21.5mb	24.0mb
7.79s	3.17s	1.61s	0.00s	9	0	79.0mb	23.5mb	28.0mb
7.61s	3.01s	1.64s	0.00s	7	0	73.0mb	27.5mb	32.0mb
7.53s	2.95s	1.70s	0.00s	5	0	68.5mb	29.5mb	36.0mb
7.64s	2.97s	1.71s	0.00s	4	0	66.5mb	33.0mb	40.0mb
`perm:200,1` **2DOF upper+lower T=2.0**								
43.96s	37.44s	6.17s	0.00s	200	0	1182mb	24.0mb	25.0mb
34.96s	28.53s	6.19s	0.00s	124	0	894.5mb	27.5mb	28.0mb
29.26s	22.96s	6.36s	0.00s	85	0	703.0mb	30.5mb	31.0mb
26.03s	19.73s	6.44s	0.00s	67	0	603.0mb	33.5mb	34.0mb
21.36s	15.14s	6.56s	0.00s	46	0	468.5mb	35.0mb	37.0mb
`perm:25,8` **2DOF upper+lower T=2.0**								
21.63s	15.42s	6.34s	0.00s	78	0	485.5mb	23.0mb	25.0mb
21.19s	15.10s	6.60s	0.00s	62	0	470.0mb	26.0mb	28.0mb
17.89s	11.66s	6.44s	0.00s	44	0	373.0mb	30.0mb	31.0mb
17.84s	11.76s	6.55s	0.00s	39	0	373.0mb	31.5mb	34.0mb
14.55s	8.57s	6.59s	0.00s	25	0	283.5mb	30.0mb	37.0mb
`twobitlong` **2DOF upper+lower T=2.0**								
145.56s	6.47s	4.62s	0.00s	19	0	95.5mb	18.5mb	18.0mb
145.46s	5.34s	4.36s	0.00s	12	0	71.0mb	18.0mb	20.0mb
147.01s	5.21s	4.51s	0.00s	9	0	63.5mb	18.5mb	22.0mb
145.79s	5.41s	4.78s	0.00s	8	0	62.0mb	21.0mb	24.0mb
146.01s	5.65s	4.78s	0.00s	8	0	67.5mb	23.0mb	26.0mb
`twobitshort` **2DOF upper+lower T=2.0**								
24.03s	1.73s	1.02s	0.00s	6	0	36.0mb	17.0mb	18.0mb
23.21s	1.85s	1.15s	0.00s	5	0	36.0mb	18.0mb	20.0mb
23.70s	1.88s	1.17s	0.00s	4	0	34.0mb	19.0mb	22.0mb
23.22s	1.61s	1.13s	0.00s	3	0	30.5mb	20.5mb	24.0mb
22.96s	1.48s	1.18s	0.00s	2	0	27.0mb	22.0mb	26.0mb

Table A.29: Raw data for 2DOF: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **2DOF upper+lower T=3.0**								
								26.0mb
79.09s	56.72s	13.66s	38.22s	68	300	2172mb	28.5mb	29.0mb
44.63s	22.83s	13.75s	6.59s	56	51	830.0mb	29.5mb	32.0mb
45.82s	23.71s	13.85s	6.63s	53	49	854.5mb	34.0mb	35.0mb
44.96s	22.88s	13.81s	5.84s	48	44	816.5mb	37.0mb	38.0mb
`gcold:1,0` **2DOF upper+lower T=3.0**								
21.49s	12.50s	4.70s	0.00s	22	0	354.5mb	52.0mb	56.0mb
21.68s	12.56s	4.75s	0.00s	19	0	348.0mb	56.5mb	61.0mb
17.89s	8.97s	5.09s	0.00s	12	0	249.5mb	56.5mb	66.0mb
16.42s	7.41s	5.05s	0.00s	9	0	209.5mb	62.5mb	71.0mb
16.27s	7.28s	5.23s	0.00s	8	0	201.0mb	64.5mb	76.0mb
`gcold:1,1000` **2DOF upper+lower T=3.0**								
61.99s	18.09s	4.80s	0.00s	32	0	515.5mb	50.5mb	56.0mb
60.67s	16.85s	4.88s	0.00s	26	0	472.5mb	56.5mb	61.0mb
59.54s	15.54s	4.75s	0.00s	21	0	432.5mb	61.5mb	66.0mb
58.28s	14.46s	5.01s	0.00s	17	0	390.0mb	66.5mb	71.0mb
57.13s	13.21s	5.08s	0.00s	16	0	346.5mb	72.5mb	76.0mb
`gcold:10,1000` **2DOF upper+lower T=3.0**								
37.98s	1.66s	0.62s	0.00s	2	0	37.0mb	47.0mb	56.0mb
37.60s	1.40s	0.76s	0.00s	1	0	27.5mb	51.5mb	61.0mb
37.73s	1.44s	0.74s	0.00s	1	0	29.0mb	56.0mb	66.0mb
37.04s	0.71s	0.71s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.00s	0.63s	0.63s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **2DOF upper+lower T=3.0**								
8.73s	4.09s	1.54s	0.00s	16	0	102.0mb	21.5mb	24.0mb
7.81s	3.19s	1.60s	0.00s	9	0	79.0mb	24.0mb	28.0mb
7.60s	2.97s	1.60s	0.00s	7	0	73.0mb	27.5mb	32.0mb
7.52s	2.96s	1.71s	0.00s	5	0	68.5mb	29.5mb	36.0mb
7.66s	3.08s	1.81s	0.00s	4	0	66.5mb	33.0mb	40.0mb
`perm:200,1` **2DOF upper+lower T=3.0**								
43.99s	37.49s	6.15s	0.00s	200	0	1182mb	24.0mb	25.0mb
34.97s	28.47s	6.14s	0.00s	124	0	895.5mb	27.5mb	28.0mb
29.27s	23.08s	6.41s	0.00s	85	0	703.0mb	30.5mb	31.0mb
24.40s	18.14s	6.41s	0.00s	58	0	557.0mb	33.5mb	34.0mb
19.32s	13.21s	6.75s	0.00s	39	0	409.5mb	36.0mb	37.0mb
`perm:25,8` **2DOF upper+lower T=3.0**								
21.98s	15.87s	6.46s	0.00s	74	0	495.5mb	24.5mb	25.0mb
20.44s	14.42s	6.60s	0.00s	59	0	448.0mb	27.0mb	28.0mb
17.95s	11.87s	6.59s	0.00s	44	0	374.5mb	30.0mb	31.0mb
16.91s	10.91s	6.54s	0.00s	35	0	349.5mb	30.0mb	34.0mb
14.55s	8.63s	6.68s	0.00s	25	0	283.5mb	30.0mb	37.0mb
`twobitlong` **2DOF upper+lower T=3.0**								
145.82s	6.37s	4.39s	0.00s	20	0	98.0mb	18.5mb	18.0mb
145.68s	5.56s	4.59s	0.00s	12	0	71.0mb	18.0mb	20.0mb
146.01s	5.49s	4.79s	0.00s	9	0	63.5mb	18.5mb	22.0mb
145.46s	5.09s	4.48s	0.00s	8	0	62.0mb	21.0mb	24.0mb
145.54s	5.65s	4.81s	0.00s	8	0	67.5mb	23.0mb	26.0mb
`twobitshort` **2DOF upper+lower T=3.0**								
24.12s	1.74s	1.02s	0.00s	6	0	36.0mb	17.0mb	18.0mb
23.31s	1.82s	1.10s	0.00s	5	0	36.0mb	18.0mb	20.0mb
23.75s	1.96s	1.26s	0.00s	4	0	34.0mb	19.0mb	22.0mb
23.35s	1.62s	1.14s	0.00s	3	0	30.5mb	20.5mb	24.0mb
23.05s	1.44s	1.14s	0.00s	2	0	27.0mb	22.0mb	26.0mb

Table A.30: Raw data for 2DOF: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **2DOF upper+lower T=2.0**								
15.42s	5.51s	2.26s	0.00s	17	0	130.5mb	25.0mb	28.0mb
14.84s	4.91s	2.19s	0.00s	12	0	115.0mb	27.5mb	31.0mb
15.45s	5.56s	2.26s	0.00s	12	0	127.0mb	30.5mb	34.0mb
17.26s	7.49s	2.41s	0.00s	15	0	167.0mb	34.0mb	37.0mb
16.36s	6.46s	2.20s	0.00s	12	0	148.0mb	36.5mb	40.0mb
`5nboyer:3` **2DOF upper+lower T=2.0**								
45.57s	22.26s	7.04s	0.00s	92	0	574.0mb	23.0mb	24.0mb
40.68s	17.48s	7.12s	0.00s	64	0	455.0mb	24.5mb	28.0mb
37.86s	14.92s	7.31s	0.00s	44	0	393.0mb	27.5mb	32.0mb
37.39s	14.43s	7.32s	0.00s	36	0	378.5mb	31.5mb	36.0mb
36.55s	13.53s	7.43s	0.00s	30	0	347.5mb	34.5mb	40.0mb
`5sboyer:4` **2DOF upper+lower T=2.0**								
								20.0mb
72.83s	6.09s	2.53s	0.00s	21	0	134.0mb	21.5mb	24.0mb
72.04s	5.11s	2.49s	0.00s	14	0	117.0mb	24.5mb	28.0mb
72.55s	5.64s	2.43s	0.00s	14	0	131.0mb	29.0mb	32.0mb
71.64s	4.93s	2.39s	0.00s	10	0	110.0mb	31.5mb	36.0mb
`5twobitshort` **2DOF upper+lower T=2.0**								
117.84s	9.22s	4.21s	0.00s	47	0	211.5mb	17.0mb	18.0mb
114.93s	8.45s	4.25s	0.00s	35	0	188.0mb	18.0mb	20.0mb
115.11s	7.96s	4.20s	0.00s	28	0	178.0mb	19.5mb	22.0mb
112.55s	6.71s	4.14s	0.00s	21	0	149.5mb	20.5mb	24.0mb
112.23s	6.97s	4.55s	0.00s	19	0	148.0mb	22.5mb	26.0mb
`5earley:12` **2DOF upper+lower T=3.0**								
16.03s	6.29s	2.41s	0.00s	17	0	144.0mb	26.5mb	28.0mb
15.70s	5.76s	2.27s	0.00s	15	0	139.0mb	29.0mb	31.0mb
15.73s	5.81s	2.21s	0.00s	12	0	133.0mb	32.5mb	34.0mb
18.36s	8.43s	2.22s	0.00s	18	0	190.5mb	35.5mb	37.0mb
16.36s	6.43s	2.18s	0.00s	12	0	148.0mb	36.5mb	40.0mb
`5nboyer:3` **2DOF upper+lower T=3.0**								
45.26s	22.23s	7.36s	0.00s	91	0	567.5mb	23.0mb	24.0mb
38.67s	15.68s	7.22s	0.00s	53	0	414.5mb	25.5mb	28.0mb
37.39s	14.28s	7.17s	0.00s	41	0	381.0mb	28.5mb	32.0mb
36.32s	13.28s	7.18s	0.00s	33	0	354.0mb	32.5mb	36.0mb
36.48s	13.36s	7.36s	0.00s	29	0	347.5mb	34.5mb	40.0mb
`5sboyer:4` **2DOF upper+lower T=3.0**								
								20.0mb
72.76s	5.68s	2.37s	0.00s	20	0	129.0mb	22.5mb	24.0mb
72.37s	5.16s	2.43s	0.00s	14	0	119.0mb	26.0mb	28.0mb
73.03s	5.88s	2.55s	0.00s	14	0	134.0mb	30.5mb	32.0mb
72.55s	5.16s	2.51s	0.00s	10	0	112.0mb	33.0mb	36.0mb
`5twobitshort` **2DOF upper+lower T=3.0**								
116.53s	8.35s	4.31s	0.00s	36	0	186.5mb	18.0mb	18.0mb
114.50s	8.58s	4.47s	0.00s	32	0	184.5mb	20.0mb	20.0mb
114.07s	7.63s	4.30s	0.00s	26	0	167.0mb	20.5mb	22.0mb
112.72s	7.09s	4.34s	0.00s	22	0	153.5mb	21.0mb	24.0mb
112.43s	6.83s	4.39s	0.00s	19	0	148.0mb	22.5mb	26.0mb

Table A.31: Raw data for 2DOF: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **2DOF upper T=3.0**								
								26.0mb
87.39s	64.90s	13.51s	44.25s	97	350	2484mb	29.0mb	29.0mb
47.92s	25.89s	13.46s	6.49s	87	50	926.5mb	30.0mb	32.0mb
41.83s	19.57s	13.31s	0.25s	85	2	672.5mb	31.5mb	35.0mb
41.63s	19.63s	13.51s	0.38s	79	3	665.5mb	33.0mb	38.0mb
`gcold:1,0` **2DOF upper T=3.0**								
21.99s	12.94s	4.47s	0.00s	25	0	371.5mb	51.0mb	56.0mb
22.59s	13.59s	4.69s	0.00s	25	0	384.0mb	55.5mb	61.0mb
21.00s	12.05s	4.76s	0.00s	22	0	343.0mb	58.0mb	66.0mb
20.24s	11.22s	4.68s	0.00s	19	0	319.0mb	61.0mb	71.0mb
19.76s	10.77s	4.68s	0.00s	19	0	310.0mb	61.0mb	76.0mb
`gcold:1,1000` **2DOF upper T=3.0**								
62.50s	18.55s	4.55s	0.00s	36	0	538.5mb	50.5mb	56.0mb
61.50s	17.57s	4.57s	0.00s	31	0	504.5mb	55.5mb	61.0mb
60.59s	16.68s	4.69s	0.00s	27	0	471.0mb	61.0mb	66.0mb
59.60s	15.51s	4.45s	0.00s	24	0	441.0mb	65.5mb	71.0mb
59.11s	15.07s	4.52s	0.00s	22	0	423.0mb	69.0mb	76.0mb
`gcold:10,1000` **2DOF upper T=3.0**								
38.60s	2.36s	0.49s	0.00s	5	0	60.0mb	42.0mb	56.0mb
38.19s	2.13s	0.69s	0.00s	4	0	49.5mb	40.5mb	61.0mb
38.23s	1.96s	0.51s	0.00s	4	0	49.5mb	41.0mb	66.0mb
38.19s	1.96s	0.51s	0.00s	4	0	49.5mb	41.0mb	71.0mb
38.21s	1.98s	0.54s	0.00s	4	0	49.5mb	41.0mb	76.0mb
`nboyer:3` **2DOF upper T=3.0**								
10.21s	5.55s	1.42s	0.00s	42	0	139.5mb	21.5mb	24.0mb
9.73s	5.24s	1.49s	0.00s	38	0	128.0mb	24.0mb	28.0mb
9.58s	4.91s	1.40s	0.00s	37	0	123.0mb	25.0mb	32.0mb
9.35s	4.73s	1.47s	0.00s	36	0	118.5mb	26.0mb	36.0mb
9.31s	4.65s	1.43s	0.00s	36	0	117.5mb	26.5mb	40.0mb
`perm:200,1` **2DOF upper T=3.0**								
44.15s	37.70s	6.17s	0.00s	202	0	1190mb	24.5mb	25.0mb
35.83s	29.36s	6.13s	0.00s	130	0	923.5mb	27.0mb	28.0mb
29.57s	23.16s	6.13s	0.00s	88	0	717.5mb	30.5mb	31.0mb
23.83s	17.59s	6.29s	0.00s	60	0	543.0mb	33.5mb	34.0mb
21.67s	15.35s	6.20s	0.00s	50	0	481.0mb	36.5mb	37.0mb
`perm:25,8` **2DOF upper T=3.0**								
23.11s	17.01s	6.26s	0.00s	87	0	538.0mb	24.0mb	25.0mb
22.61s	16.51s	6.33s	0.00s	78	0	516.0mb	27.0mb	28.0mb
19.35s	13.24s	6.25s	0.00s	59	0	422.0mb	29.0mb	31.0mb
18.74s	12.61s	6.24s	0.00s	55	0	406.5mb	29.5mb	34.0mb
18.86s	12.68s	6.19s	0.00s	53	0	409.0mb	30.5mb	37.0mb
`twobitlong` **2DOF upper T=3.0**								
145.78s	6.45s	4.56s	0.00s	43	0	99.0mb	17.0mb	18.0mb
145.28s	6.39s	4.44s	0.00s	43	0	100.5mb	17.5mb	20.0mb
145.16s	6.06s	4.48s	0.00s	41	0	92.0mb	17.5mb	22.0mb
145.68s	5.88s	4.27s	0.00s	41	0	92.0mb	17.5mb	24.0mb
145.67s	6.08s	4.48s	0.00s	41	0	92.0mb	17.5mb	26.0mb
`twobitshort` **2DOF upper T=3.0**								
24.92s	2.61s	1.05s	0.00s	24	0	59.0mb	16.5mb	18.0mb
24.46s	2.67s	1.04s	0.00s	23	0	58.5mb	17.5mb	20.0mb
24.44s	2.64s	0.99s	0.00s	23	0	58.5mb	17.5mb	22.0mb
24.43s	2.65s	1.02s	0.00s	23	0	58.5mb	17.5mb	24.0mb
24.44s	2.61s	1.01s	0.00s	23	0	58.5mb	17.5mb	26.0mb

Table A.32: Raw data for 2DOF: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **2DOF upper T=3.0**								
18.32s	8.35s	2.08s	0.00s	42	0	199.0mb	26.5mb	28.0mb
19.91s	9.88s	2.06s	0.00s	44	0	228.0mb	29.5mb	31.0mb
18.62s	8.50s	1.84s	0.00s	38	0	202.0mb	32.0mb	34.0mb
19.48s	9.46s	1.97s	0.00s	39	0	223.0mb	34.5mb	37.0mb
17.07s	7.20s	2.14s	0.00s	33	0	169.5mb	34.0mb	40.0mb
`5nboyer:3` **2DOF upper T=3.0**								
46.67s	23.38s	6.91s	0.00s	117	0	605.0mb	23.0mb	24.0mb
40.67s	17.62s	6.93s	0.00s	82	0	463.5mb	25.5mb	28.0mb
40.57s	17.21s	6.70s	0.00s	74	0	461.5mb	28.5mb	32.0mb
38.69s	15.41s	6.84s	0.00s	65	0	411.5mb	32.5mb	36.0mb
38.75s	15.76s	7.11s	0.00s	63	0	409.5mb	33.5mb	40.0mb
`5sboyer:4` **2DOF upper T=3.0**								
								20.0mb
76.35s	7.64s	2.30s	0.00s	39	0	173.0mb	23.0mb	24.0mb
73.72s	6.31s	2.34s	0.00s	32	0	147.0mb	26.0mb	28.0mb
73.29s	6.48s	2.30s	0.00s	32	0	151.0mb	28.5mb	32.0mb
73.28s	6.45s	2.28s	0.00s	32	0	152.0mb	28.5mb	36.0mb
`5twobitshort` **2DOF upper T=3.0**								
115.84s	8.91s	4.16s	0.00s	53	0	205.5mb	18.5mb	18.0mb
114.73s	8.59s	4.16s	0.00s	47	0	194.0mb	19.5mb	20.0mb
114.70s	8.62s	4.33s	0.00s	45	0	191.5mb	20.5mb	22.0mb
114.42s	8.29s	4.15s	0.00s	44	0	189.0mb	21.0mb	24.0mb
114.54s	8.40s	4.28s	0.00s	44	0	189.0mb	21.0mb	26.0mb

Table A.33: Raw data for 2DOF: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`5earley:12` **2DOF variable**								
18.67s	8.37s	2.05s	0.00s	48	0	197.0mb	28.0mb	2.25
18.89s	8.31s	1.77s	0.00s	43	0	204.0mb	32.5mb	2.5
18.83s	8.22s	1.82s	0.00s	36	0	197.0mb	41.5mb	2.75
17.25s	7.06s	2.24s	0.00s	31	0	161.5mb	35.0mb	3.0
`5nboyer:3` **2DOF variable**								
42.06s	18.85s	7.05s	0.00s	97	0	489.5mb	26.0mb	2.25
40.53s	17.19s	6.92s	0.00s	80	0	452.5mb	30.5mb	2.5
39.16s	15.93s	6.95s	0.00s	70	0	424.0mb	32.0mb	2.75
38.62s	15.51s	7.02s	0.00s	63	0	407.0mb	32.5mb	3.0
`5sboyer:4` **2DOF variable**								
75.55s	8.61s	2.18s	0.00s	51	0	196.0mb	27.5mb	2.25
74.19s	7.20s	2.39s	0.00s	42	0	160.5mb	23.0mb	2.5
74.59s	7.67s	2.26s	0.00s	38	0	173.5mb	31.5mb	2.75
73.39s	6.48s	2.26s	0.00s	32	0	152.0mb	28.5mb	3.0
`5twobitshort` **2DOF variable**								
119.56s	10.94s	4.24s	0.00s	77	0	258.0mb	17.0mb	2.25
115.94s	9.31s	4.41s	0.00s	56	0	209.5mb	18.0mb	2.5
116.72s	9.04s	4.19s	0.00s	50	0	205.5mb	20.5mb	2.75
114.50s	8.19s	4.03s	0.00s	44	0	189.0mb	21.0mb	3.0

Table A.34: Raw data for 2DOF: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`gcbench:5` **2DOF variable**								
42.88s	21.01s	13.82s	0.00s	100	0	704.0mb	32.5mb	2.25
43.13s	21.00s	13.55s	0.00s	88	0	722.0mb	37.0mb	2.5
42.64s	20.39s	13.55s	0.00s	85	0	682.5mb	34.0mb	2.75
41.94s	19.89s	13.52s	0.00s	75	0	680.0mb	38.0mb	3.0
`gcold:1,0` **2DOF variable**								
23.15s	14.16s	4.58s	0.00s	29	0	405.0mb	53.5mb	2.25
22.21s	13.22s	4.60s	0.00s	25	0	372.5mb	57.0mb	2.5
20.07s	11.04s	4.55s	0.00s	19	0	318.5mb	59.0mb	2.75
19.80s	10.84s	4.70s	0.00s	19	0	311.0mb	61.5mb	3.0
`gcold:1,1000` **2DOF variable**								
61.04s	17.04s	4.63s	0.00s	30	0	484.5mb	67.5mb	2.25
59.55s	15.55s	4.61s	0.00s	25	0	435.0mb	78.0mb	2.5
58.26s	14.51s	4.78s	0.00s	20	0	396.0mb	84.5mb	2.75
57.98s	14.07s	4.68s	0.00s	19	0	384.0mb	89.5mb	3.0
`gcold:10,1000` **2DOF variable**								
39.00s	2.72s	0.50s	0.00s	7	0	71.5mb	40.0mb	2.25
38.66s	2.49s	0.60s	0.00s	6	0	62.0mb	41.0mb	2.5
38.34s	2.10s	0.49s	0.00s	5	0	55.0mb	42.0mb	2.75
38.18s	1.95s	0.50s	0.00s	4	0	49.5mb	41.0mb	3.0
`nboyer:3` **2DOF variable**								
10.92s	6.29s	1.45s	0.00s	54	0	158.5mb	23.5mb	2.25
10.47s	5.81s	1.43s	0.00s	48	0	145.5mb	24.5mb	2.5
9.71s	5.07s	1.46s	0.00s	40	0	126.0mb	23.5mb	2.75
9.33s	4.72s	1.47s	0.00s	36	0	117.5mb	26.5mb	3.0
`perm:200,1` **2DOF variable**								
22.73s	16.43s	6.15s	0.00s	56	0	520.0mb	35.5mb	2.25
19.65s	13.66s	6.43s	0.00s	39	0	428.5mb	41.0mb	2.5
18.81s	12.75s	6.36s	0.00s	38	0	402.5mb	42.0mb	2.75
17.55s	11.55s	6.47s	0.00s	32	0	366.0mb	40.0mb	3.0
`perm:25,8` **2DOF variable**								
19.94s	13.75s	6.23s	0.00s	66	0	444.0mb	30.0mb	2.25
18.08s	12.04s	6.17s	0.00s	53	0	394.0mb	29.0mb	2.5
18.66s	12.48s	6.25s	0.00s	53	0	401.5mb	33.5mb	2.75
16.94s	10.88s	6.33s	0.00s	46	0	355.0mb	32.0mb	3.0
`twobitlong` **2DOF variable**								
146.19s	7.08s	4.16s	0.00s	62	0	128.5mb	17.0mb	2.25
145.39s	6.66s	4.27s	0.00s	55	0	114.5mb	16.0mb	2.5
144.83s	6.43s	4.43s	0.00s	43	0	102.0mb	17.0mb	2.75
145.30s	5.97s	4.36s	0.00s	41	0	92.0mb	17.5mb	3.0
`twobitshort` **2DOF variable**								
25.59s	3.33s	1.00s	0.00s	36	0	81.0mb	16.5mb	2.25
24.59s	2.97s	1.07s	0.00s	29	0	68.0mb	16.5mb	2.5
24.35s	2.64s	1.03s	0.00s	24	0	61.0mb	16.0mb	2.75
24.45s	2.71s	1.09s	0.00s	23	0	58.5mb	17.5mb	3.0

Table A.35: Raw data for 2DOF: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **3DOF upper+lower T=2.0**								
45.13s	23.10s	13.80s	4.51s	109	33	815.5mb	26.0mb	26.0mb
42.16s	20.04s	13.68s	0.49s	101	4	678.5mb	28.5mb	29.0mb
								32.0mb
41.39s	19.26s	13.77s	0.54s	77	4	646.0mb	34.0mb	35.0mb
40.75s	18.53s	13.69s	0.48s	69	4	631.0mb	37.0mb	38.0mb
`gcold:1,0` **3DOF upper+lower T=2.0**								
17.11s	8.23s	4.87s	0.00s	18	0	239.5mb	50.5mb	56.0mb
15.74s	6.86s	4.95s	0.00s	13	0	200.5mb	54.5mb	61.0mb
15.80s	6.88s	4.95s	0.00s	13	0	202.0mb	61.0mb	66.0mb
14.45s	5.52s	5.05s	0.00s	9	0	164.5mb	55.0mb	71.0mb
14.20s	5.23s	5.02s	0.00s	7	0	156.5mb	59.5mb	76.0mb
`gcold:1,1000` **3DOF upper+lower T=2.0**								
61.38s	17.35s	4.79s	0.00s	41	0	485.5mb	54.5mb	56.0mb
60.68s	16.79s	4.96s	0.00s	36	0	465.0mb	59.0mb	61.0mb
59.60s	15.64s	5.01s	0.00s	30	0	434.5mb	63.5mb	66.0mb
58.64s	14.52s	4.92s	0.00s	25	0	404.5mb	67.5mb	71.0mb
57.26s	13.25s	4.98s	0.00s	20	0	368.5mb	70.5mb	76.0mb
`gcold:10,1000` **3DOF upper+lower T=2.0**								
37.85s	1.45s	0.62s	0.00s	2	0	32.5mb	48.5mb	56.0mb
37.58s	1.22s	0.70s	0.00s	1	0	25.0mb	53.0mb	61.0mb
37.08s	0.76s	0.76s	0.00s	0	0	15.5mb	45.0mb	66.0mb
37.09s	0.80s	0.80s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.06s	0.83s	0.83s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **3DOF upper+lower T=2.0**								
9.24s	4.55s	1.56s	0.00s	21	0	109.0mb	21.5mb	24.0mb
8.09s	3.45s	1.63s	0.00s	12	0	82.0mb	24.5mb	28.0mb
7.75s	3.02s	1.62s	0.00s	8	0	72.0mb	29.0mb	32.0mb
7.76s	3.16s	1.83s	0.00s	6	0	68.5mb	31.0mb	36.0mb
7.12s	2.51s	1.81s	0.00s	4	0	56.0mb	32.5mb	40.0mb
`perm:200,1` **3DOF upper+lower T=2.0**								
33.63s	27.13s	6.27s	0.00s	171	0	854.5mb	24.5mb	25.0mb
26.69s	20.42s	6.50s	0.00s	116	0	632.5mb	27.5mb	28.0mb
20.17s	13.99s	6.40s	0.00s	71	0	444.0mb	30.0mb	31.0mb
17.21s	11.18s	6.53s	0.00s	50	0	358.0mb	31.5mb	34.0mb
15.32s	9.23s	6.52s	0.00s	38	0	305.5mb	34.5mb	37.0mb
`perm:25,8` **3DOF upper+lower T=2.0**								
19.12s	13.13s	6.60s	0.00s	82	0	410.0mb	23.5mb	25.0mb
17.16s	10.98s	6.38s	0.00s	61	0	354.0mb	25.5mb	28.0mb
15.27s	9.11s	6.32s	0.00s	45	0	305.0mb	28.5mb	31.0mb
15.30s	9.25s	6.52s	0.00s	41	0	301.5mb	31.5mb	34.0mb
14.24s	8.25s	6.51s	0.00s	32	0	277.5mb	32.5mb	37.0mb
`twobitlong` **3DOF upper+lower T=2.0**								
144.96s	6.41s	4.55s	0.00s	24	0	90.5mb	17.0mb	18.0mb
146.13s	6.00s	4.71s	0.00s	17	0	76.0mb	19.0mb	20.0mb
146.24s	5.66s	4.45s	0.00s	14	0	75.0mb	21.5mb	22.0mb
145.26s	5.54s	4.63s	0.00s	11	0	67.5mb	22.5mb	24.0mb
144.65s	5.32s	4.72s	0.00s	8	0	59.0mb	23.5mb	26.0mb
`twobitshort` **3DOF upper+lower T=2.0**								
23.37s	1.86s	1.07s	0.00s	8	0	37.5mb	17.0mb	18.0mb
24.58s	1.71s	1.14s	0.00s	5	0	32.0mb	18.0mb	20.0mb
23.10s	1.62s	1.17s	0.00s	4	0	29.0mb	20.0mb	22.0mb
23.09s	1.64s	1.22s	0.00s	3	0	28.5mb	22.0mb	24.0mb
22.97s	1.52s	1.16s	0.00s	2	0	27.0mb	23.0mb	26.0mb

Table A.36: Raw data for 3DOF: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **3DOF upper+lower T=3.0**								
49.34s	27.39s	13.81s	9.70s	100	73	1000mb	25.5mb	26.0mb
								29.0mb
								32.0mb
46.52s	24.68s	14.12s	7.24s	68	53	872.5mb	34.5mb	35.0mb
45.74s	23.47s	13.50s	7.13s	61	52	851.0mb	37.0mb	38.0mb
`gcold:1,0` **3DOF upper+lower T=3.0**								
17.11s	8.10s	4.73s	0.00s	18	0	239.0mb	50.5mb	56.0mb
15.71s	7.00s	5.07s	0.00s	13	0	200.5mb	54.5mb	61.0mb
15.77s	6.83s	4.91s	0.00s	13	0	202.0mb	61.0mb	66.0mb
14.44s	5.49s	5.01s	0.00s	9	0	164.5mb	55.0mb	71.0mb
14.21s	5.38s	5.16s	0.00s	7	0	156.5mb	59.5mb	76.0mb
`gcold:1,1000` **3DOF upper+lower T=3.0**								
61.46s	17.46s	4.91s	0.00s	41	0	488.0mb	54.5mb	56.0mb
60.36s	16.49s	4.98s	0.00s	35	0	455.5mb	59.0mb	61.0mb
59.69s	15.64s	4.90s	0.00s	30	0	436.0mb	63.5mb	66.0mb
58.60s	14.61s	4.95s	0.00s	25	0	405.5mb	67.5mb	71.0mb
57.31s	13.28s	4.96s	0.00s	20	0	370.0mb	71.5mb	76.0mb
`gcold:10,1000` **3DOF upper+lower T=3.0**								
37.86s	1.63s	0.79s	0.00s	2	0	32.5mb	48.5mb	56.0mb
37.60s	1.26s	0.74s	0.00s	1	0	25.0mb	53.0mb	61.0mb
37.06s	0.69s	0.69s	0.00s	0	0	15.5mb	45.0mb	66.0mb
36.97s	0.91s	0.91s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.06s	0.82s	0.82s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **3DOF upper+lower T=3.0**								
9.07s	4.37s	1.56s	0.00s	20	0	105.5mb	22.5mb	24.0mb
7.99s	3.31s	1.58s	0.00s	11	0	78.5mb	24.5mb	28.0mb
7.74s	3.01s	1.62s	0.00s	8	0	72.0mb	29.0mb	32.0mb
7.77s	3.05s	1.71s	0.00s	6	0	68.5mb	31.0mb	36.0mb
7.12s	2.44s	1.73s	0.00s	4	0	56.0mb	32.5mb	40.0mb
`perm:200,1` **3DOF upper+lower T=3.0**								
33.02s	26.48s	6.12s	0.00s	164	0	833.5mb	24.5mb	25.0mb
21.61s	15.46s	6.43s	0.00s	83	0	487.5mb	27.5mb	28.0mb
18.38s	12.22s	6.37s	0.00s	59	0	393.0mb	30.0mb	31.0mb
17.35s	11.32s	6.51s	0.00s	50	0	360.5mb	32.5mb	34.0mb
15.32s	9.30s	6.53s	0.00s	38	0	305.5mb	34.5mb	37.0mb
`perm:25,8` **3DOF upper+lower T=3.0**								
18.53s	12.43s	6.46s	0.00s	75	0	394.0mb	24.0mb	25.0mb
16.99s	11.03s	6.52s	0.00s	60	0	352.0mb	26.5mb	28.0mb
15.28s	9.29s	6.46s	0.00s	45	0	305.0mb	28.5mb	31.0mb
15.32s	9.35s	6.66s	0.00s	41	0	301.5mb	31.5mb	34.0mb
14.23s	8.19s	6.41s	0.00s	32	0	277.5mb	32.5mb	37.0mb
`twobitlong` **3DOF upper+lower T=3.0**								
144.89s	6.10s	4.37s	0.00s	22	0	87.0mb	18.0mb	18.0mb
146.53s	5.90s	4.57s	0.00s	17	0	76.5mb	19.5mb	20.0mb
145.84s	5.91s	4.70s	0.00s	14	0	75.0mb	21.5mb	22.0mb
144.97s	5.61s	4.71s	0.00s	11	0	67.5mb	22.5mb	24.0mb
144.20s	5.18s	4.60s	0.00s	8	0	59.0mb	23.5mb	26.0mb
`twobitshort` **3DOF upper+lower T=3.0**								
23.42s	1.94s	1.16s	0.00s	8	0	37.5mb	17.0mb	18.0mb
24.56s	1.70s	1.13s	0.00s	5	0	32.0mb	18.0mb	20.0mb
23.07s	1.64s	1.20s	0.00s	4	0	29.0mb	20.0mb	22.0mb
23.15s	1.66s	1.24s	0.00s	3	0	28.5mb	22.0mb	24.0mb
23.01s	1.60s	1.24s	0.00s	2	0	27.0mb	23.0mb	26.0mb

Table A.37: Raw data for 3DOF: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
5earley:12 **3DOF upper+lower T=2.0**								
18.35s	8.41s	2.20s	0.00s	31	0	190.0mb	27.0mb	28.0mb
15.14s	5.15s	2.20s	0.00s	16	0	120.0mb	27.5mb	31.0mb
16.61s	6.70s	2.34s	0.00s	19	0	147.5mb	32.0mb	34.0mb
14.76s	4.78s	2.26s	0.00s	11	0	107.0mb	34.0mb	37.0mb
16.44s	6.36s	2.28s	0.00s	15	0	140.0mb	37.0mb	40.0mb
5nboyer:3 **3DOF upper+lower T=2.0**								
45.03s	21.43s	7.18s	0.00s	105	0	525.5mb	23.0mb	24.0mb
42.21s	18.74s	7.09s	0.00s	77	0	470.0mb	26.5mb	28.0mb
40.10s	16.89s	7.47s	0.00s	60	0	416.5mb	29.0mb	32.0mb
39.36s	15.52s	7.25s	0.00s	48	0	390.5mb	32.0mb	36.0mb
37.79s	14.32s	7.31s	0.00s	40	0	361.5mb	35.5mb	40.0mb
5sboyer:4 **3DOF upper+lower T=2.0**								
								20.0mb
74.53s	7.51s	2.55s	0.00s	32	0	160.5mb	22.5mb	24.0mb
73.22s	6.22s	2.47s	0.00s	22	0	133.5mb	25.0mb	28.0mb
72.76s	5.94s	2.53s	0.00s	18	0	125.5mb	29.0mb	32.0mb
71.66s	4.87s	2.47s	0.00s	11	0	105.5mb	33.0mb	36.0mb
5twobitshort **3DOF upper+lower T=2.0**								
116.03s	10.36s	4.51s	0.00s	59	0	219.0mb	18.5mb	18.0mb
116.76s	9.53s	4.42s	0.00s	47	0	199.5mb	19.5mb	20.0mb
114.47s	8.80s	4.38s	0.00s	40	0	186.0mb	21.0mb	22.0mb
113.12s	7.50s	4.34s	0.00s	29	0	155.5mb	22.0mb	24.0mb
113.86s	8.17s	4.46s	0.00s	29	0	169.0mb	24.0mb	26.0mb
5earley:12 **3DOF upper+lower T=3.0**								
17.59s	7.75s	2.28s	0.00s	29	0	172.0mb	27.5mb	28.0mb
15.15s	5.38s	2.38s	0.00s	16	0	121.5mb	27.5mb	31.0mb
17.94s	7.97s	2.31s	0.00s	23	0	174.0mb	33.0mb	34.0mb
14.86s	4.89s	2.27s	0.00s	11	0	109.0mb	34.5mb	37.0mb
16.42s	6.42s	2.28s	0.00s	15	0	141.0mb	38.0mb	40.0mb
5nboyer:3 **3DOF upper+lower T=3.0**								
44.83s	21.26s	7.14s	0.00s	104	0	524.0mb	23.0mb	24.0mb
41.58s	18.07s	7.04s	0.00s	73	0	454.0mb	27.0mb	28.0mb
39.34s	16.01s	7.30s	0.00s	56	0	402.0mb	30.5mb	32.0mb
39.14s	15.67s	7.53s	0.00s	47	0	387.0mb	33.5mb	36.0mb
37.74s	14.45s	7.46s	0.00s	39	0	363.5mb	38.5mb	40.0mb
5sboyer:4 **3DOF upper+lower T=3.0**								
								20.0mb
75.45s	8.54s	2.52s	0.00s	35	0	179.5mb	23.0mb	24.0mb
73.21s	6.38s	2.52s	0.00s	21	0	136.5mb	27.0mb	28.0mb
72.90s	6.04s	2.53s	0.00s	17	0	126.0mb	29.0mb	32.0mb
71.80s	5.16s	2.72s	0.00s	11	0	106.5mb	33.5mb	36.0mb
5twobitshort **3DOF upper+lower T=3.0**								
116.89s	10.61s	4.34s	0.00s	58	0	225.5mb	19.5mb	18.0mb
116.18s	9.13s	4.42s	0.00s	43	0	190.5mb	20.5mb	20.0mb
114.63s	8.34s	4.27s	0.00s	35	0	174.5mb	22.0mb	22.0mb
114.85s	7.91s	4.29s	0.00s	30	0	164.0mb	23.5mb	24.0mb
113.41s	8.03s	4.68s	0.00s	26	0	161.0mb	25.0mb	26.0mb

Table A.38: Raw data for 3DOF: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **3DOF upper T=3.0**								
								26.0mb
47.51s	25.54s	13.57s	6.15s	124	46	916.5mb	29.0mb	29.0mb
								32.0mb
43.52s	21.63s	13.65s	0.53s	115	4	731.5mb	34.0mb	35.0mb
44.10s	22.07s	13.58s	0.00s	117	0	741.0mb	35.5mb	38.0mb
`gcold:1,0` **3DOF upper T=3.0**								
18.50s	9.55s	4.52s	0.00s	26	0	280.0mb	49.0mb	56.0mb
18.27s	9.25s	4.47s	0.00s	25	0	275.0mb	53.5mb	61.0mb
17.96s	8.83s	4.33s	0.00s	23	0	266.5mb	55.5mb	66.0mb
17.97s	9.18s	4.67s	0.00s	23	0	266.5mb	55.5mb	71.0mb
17.91s	9.04s	4.61s	0.00s	23	0	266.5mb	55.5mb	76.0mb
`gcold:1,1000` **3DOF upper T=3.0**								
62.91s	18.87s	4.66s	0.00s	49	0	537.0mb	54.5mb	56.0mb
61.25s	17.30s	4.67s	0.00s	41	0	489.5mb	58.5mb	61.0mb
60.66s	16.60s	4.55s	0.00s	38	0	468.0mb	63.5mb	66.0mb
59.85s	15.82s	4.53s	0.00s	34	0	447.0mb	67.0mb	71.0mb
59.47s	15.54s	4.70s	0.00s	32	0	435.0mb	70.5mb	76.0mb
`gcold:10,1000` **3DOF upper T=3.0**								
38.29s	1.94s	0.46s	0.00s	5	0	50.5mb	40.5mb	56.0mb
38.32s	2.00s	0.48s	0.00s	5	0	50.5mb	41.5mb	61.0mb
37.95s	1.72s	0.62s	0.00s	4	0	42.0mb	38.0mb	66.0mb
37.92s	1.53s	0.41s	0.00s	4	0	42.0mb	38.0mb	71.0mb
37.94s	1.73s	0.62s	0.00s	4	0	42.0mb	38.0mb	76.0mb
`nboyer:3` **3DOF upper T=3.0**								
10.10s	5.40s	1.41s	0.00s	50	0	132.5mb	22.0mb	24.0mb
9.36s	4.64s	1.37s	0.00s	46	0	115.5mb	22.0mb	28.0mb
9.06s	4.33s	1.41s	0.00s	44	0	109.5mb	23.0mb	32.0mb
9.13s	4.40s	1.37s	0.00s	44	0	110.5mb	24.0mb	36.0mb
9.12s	4.42s	1.42s	0.00s	44	0	110.5mb	24.0mb	40.0mb
`perm:200,1` **3DOF upper T=3.0**								
34.28s	27.93s	6.31s	0.00s	172	0	878.5mb	24.5mb	25.0mb
22.28s	16.16s	6.27s	0.00s	90	0	509.0mb	27.5mb	28.0mb
20.63s	14.54s	6.33s	0.00s	74	0	457.5mb	30.0mb	31.0mb
17.65s	11.47s	6.22s	0.00s	55	0	372.5mb	32.0mb	34.0mb
16.91s	10.89s	6.35s	0.00s	49	0	354.0mb	34.0mb	37.0mb
`perm:25,8` **3DOF upper T=3.0**								
18.89s	12.83s	6.30s	0.00s	90	0	408.0mb	24.5mb	25.0mb
18.15s	12.19s	6.35s	0.00s	78	0	388.0mb	26.5mb	28.0mb
17.44s	11.41s	6.27s	0.00s	74	0	370.0mb	26.5mb	31.0mb
17.77s	11.63s	6.27s	0.00s	74	0	374.5mb	27.5mb	34.0mb
17.76s	11.69s	6.30s	0.00s	74	0	374.5mb	27.5mb	37.0mb
`twobitlong` **3DOF upper T=3.0**								
146.76s	7.11s	4.41s	0.00s	54	0	114.5mb	18.5mb	18.0mb
146.64s	7.21s	4.47s	0.00s	53	0	114.5mb	19.5mb	20.0mb
146.19s	6.65s	4.32s	0.00s	50	0	105.5mb	19.5mb	22.0mb
146.28s	6.77s	4.37s	0.00s	50	0	106.0mb	20.0mb	24.0mb
146.17s	6.61s	4.21s	0.00s	50	0	106.0mb	20.0mb	26.0mb
`twobitshort` **3DOF upper T=3.0**								
24.41s	2.73s	1.06s	0.00s	29	0	58.5mb	17.0mb	18.0mb
24.50s	2.83s	1.10s	0.00s	29	0	59.5mb	18.0mb	20.0mb
24.57s	2.73s	1.01s	0.00s	29	0	60.0mb	18.0mb	22.0mb
24.55s	2.91s	1.12s	0.00s	29	0	60.0mb	18.0mb	24.0mb
24.56s	2.82s	1.06s	0.00s	29	0	60.0mb	18.0mb	26.0mb

Table A.39: Raw data for 3DOF: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **3DOF upper T=3.0**								
19.81s	9.82s	2.09s	0.00s	57	0	221.0mb	27.0mb	28.0mb
21.73s	11.67s	2.01s	0.00s	61	0	255.0mb	30.0mb	31.0mb
19.04s	8.97s	1.95s	0.00s	49	0	205.5mb	32.5mb	34.0mb
18.18s	8.34s	2.12s	0.00s	45	0	183.5mb	34.5mb	37.0mb
18.41s	8.47s	2.00s	0.00s	45	0	189.5mb	37.0mb	40.0mb
`5nboyer:3` **3DOF upper T=3.0**								
45.95s	22.38s	7.06s	0.00s	135	0	550.0mb	23.5mb	24.0mb
42.91s	19.77s	7.22s	0.00s	109	0	484.5mb	27.0mb	28.0mb
41.17s	17.66s	7.07s	0.00s	92	0	441.5mb	30.5mb	32.0mb
40.75s	17.18s	7.11s	0.00s	85	0	429.0mb	34.5mb	36.0mb
39.98s	16.62s	7.23s	0.00s	81	0	415.5mb	35.0mb	40.0mb
`5sboyer:4` **3DOF upper T=3.0**								
								20.0mb
76.02s	8.93s	2.31s	0.00s	56	0	190.0mb	23.0mb	24.0mb
75.03s	8.13s	2.45s	0.00s	47	0	171.0mb	26.5mb	28.0mb
74.95s	7.68s	2.30s	0.00s	46	0	163.0mb	29.5mb	32.0mb
74.50s	7.45s	2.40s	0.00s	43	0	156.0mb	31.5mb	36.0mb
`5twobitshort` **3DOF upper T=3.0**								
116.93s	10.93s	4.26s	0.00s	77	0	237.0mb	19.0mb	18.0mb
116.77s	9.87s	4.13s	0.00s	65	0	214.0mb	20.5mb	20.0mb
116.53s	10.12s	4.22s	0.00s	62	0	217.5mb	23.0mb	22.0mb
119.49s	9.56s	4.41s	0.00s	56	0	198.0mb	23.5mb	24.0mb
117.09s	9.04s	4.19s	0.00s	54	0	192.0mb	23.5mb	26.0mb

Table A.40: Raw data for 3DOF: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`5earley:12` **3DOF variable**								
20.68s	10.13s	1.83s	0.00s	62	0	231.0mb	40.0mb	2.25
19.43s	8.95s	2.04s	0.00s	55	0	201.5mb	38.0mb	2.5
18.90s	8.62s	2.07s	0.00s	47	0	194.0mb	44.5mb	2.75
17.42s	7.13s	2.08s	0.00s	41	0	163.0mb	42.5mb	3.0
`5nboyer:3` **3DOF variable**								
44.22s	20.60s	6.80s	0.00s	129	0	512.0mb	29.0mb	2.25
41.67s	18.27s	7.08s	0.00s	103	0	456.0mb	31.0mb	2.5
40.31s	16.91s	7.06s	0.00s	89	0	424.5mb	33.0mb	2.75
39.92s	16.34s	7.07s	0.00s	78	0	406.5mb	42.5mb	3.0
`5sboyer:4` **3DOF variable**								
76.58s	9.50s	2.36s	0.00s	64	0	204.0mb	29.5mb	2.25
75.37s	8.14s	2.26s	0.00s	52	0	176.0mb	30.0mb	2.5
74.44s	7.73s	2.34s	0.00s	44	0	167.0mb	35.0mb	2.75
74.45s	7.38s	2.40s	0.00s	42	0	154.0mb	37.0mb	3.0
`5twobitshort` **3DOF variable**								
116.53s	10.07s	4.10s	0.00s	80	0	226.5mb	20.5mb	2.25
115.68s	9.75s	4.28s	0.00s	65	0	210.0mb	23.0mb	2.5
114.86s	9.05s	4.23s	0.00s	58	0	196.0mb	24.0mb	2.75
116.40s	8.82s	4.19s	0.00s	53	0	188.5mb	23.5mb	3.0

Table A.41: Raw data for 3DOF: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`gcbench:5` **3DOF variable**								
44.79s	22.62s	13.55s	0.00s	140	0	765.5mb	35.5mb	2.25
44.15s	22.00s	13.64s	0.00s	126	0	744.0mb	37.0mb	2.5
43.52s	21.34s	13.67s	0.00s	115	0	720.0mb	37.0mb	2.75
42.41s	20.42s	13.73s	0.00s	100	0	684.5mb	41.0mb	3.0
`gcold:1,0` **3DOF variable**								
20.81s	11.75s	4.54s	0.00s	35	0	344.0mb	46.0mb	2.25
19.49s	10.42s	4.43s	0.00s	29	0	308.5mb	49.0mb	2.5
18.75s	9.65s	4.43s	0.00s	26	0	286.5mb	52.5mb	2.75
17.95s	9.15s	4.66s	0.00s	23	0	266.5mb	55.5mb	3.0
`gcold:1,1000` **3DOF variable**								
59.95s	16.11s	4.70s	0.00s	34	0	451.0mb	75.5mb	2.25
58.28s	14.38s	4.63s	0.00s	28	0	402.0mb	78.5mb	2.5
57.70s	13.82s	4.71s	0.00s	25	0	382.5mb	84.0mb	2.75
56.90s	13.00s	4.65s	0.00s	23	0	358.5mb	87.5mb	3.0
`gcold:10,1000` **3DOF variable**								
38.83s	2.52s	0.53s	0.00s	8	0	65.5mb	37.5mb	2.25
38.52s	2.34s	0.62s	0.00s	7	0	57.5mb	40.0mb	2.5
38.47s	2.14s	0.48s	0.00s	6	0	54.5mb	39.5mb	2.75
37.93s	1.78s	0.66s	0.00s	4	0	42.0mb	38.0mb	3.0
`nboyer:3` **3DOF variable**								
10.53s	5.77s	1.38s	0.00s	66	0	142.0mb	21.0mb	2.25
10.29s	5.62s	1.48s	0.00s	59	0	137.5mb	24.5mb	2.5
9.50s	4.75s	1.38s	0.00s	48	0	119.5mb	24.0mb	2.75
9.13s	4.48s	1.45s	0.00s	44	0	110.5mb	24.0mb	3.0
`perm:200,1` **3DOF variable**								
19.20s	13.11s	6.28s	0.00s	66	0	417.5mb	30.5mb	2.25
17.73s	11.60s	6.23s	0.00s	56	0	379.0mb	32.0mb	2.5
16.59s	10.63s	6.42s	0.00s	45	0	347.0mb	36.0mb	2.75
17.09s	11.03s	6.37s	0.00s	49	0	355.5mb	35.0mb	3.0
`perm:25,8` **3DOF variable**								
20.09s	14.06s	6.33s	0.00s	101	0	444.5mb	24.5mb	2.25
18.57s	12.54s	6.33s	0.00s	87	0	400.5mb	26.0mb	2.5
17.77s	11.77s	6.35s	0.00s	75	0	377.5mb	28.5mb	2.75
17.75s	11.73s	6.34s	0.00s	74	0	374.5mb	27.5mb	3.0
`twobitlong` **3DOF variable**								
147.32s	7.45s	4.14s	0.00s	72	0	131.5mb	18.0mb	2.25
147.13s	7.41s	4.35s	0.00s	62	0	124.0mb	19.5mb	2.5
146.69s	7.11s	4.30s	0.00s	57	0	116.0mb	20.0mb	2.75
146.27s	6.67s	4.31s	0.00s	50	0	106.0mb	20.0mb	3.0
`twobitshort` **3DOF variable**								
25.06s	3.23s	0.92s	0.00s	43	0	75.5mb	16.0mb	2.25
24.69s	2.96s	1.04s	0.00s	35	0	66.5mb	16.5mb	2.5
24.33s	2.68s	1.06s	0.00s	30	0	58.5mb	17.0mb	2.75
24.55s	2.84s	1.04s	0.00s	29	0	60.0mb	18.0mb	3.0

Table A.42: Raw data for 3DOF: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **4DOF upper+lower T=2.0**								
45.96s	23.47s	13.66s	1.90s	158	15	799.0mb	26.0mb	26.0mb
45.05s	22.74s	13.82s	1.85s	133	14	770.0mb	28.5mb	29.0mb
48.31s	25.99s	13.81s	9.25s	93	70	950.5mb	32.0mb	32.0mb
42.56s	20.24s	13.88s	1.28s	97	10	682.5mb	34.0mb	35.0mb
42.00s	19.55s	13.83s	1.10s	85	9	662.5mb	37.0mb	38.0mb
`gcold:1,0` **4DOF upper+lower T=2.0**								
15.99s	6.97s	4.86s	0.36s	19	1	218.0mb	53.0mb	56.0mb
13.95s	4.95s	4.81s	0.00s	12	0	155.0mb	49.5mb	61.0mb
14.06s	5.32s	5.19s	0.00s	11	0	155.0mb	54.0mb	66.0mb
14.13s	5.22s	5.05s	0.00s	10	0	155.5mb	58.0mb	71.0mb
14.25s	5.34s	5.18s	0.00s	9	0	155.5mb	62.5mb	76.0mb
`gcold:1,1000` **4DOF upper+lower T=2.0**								
60.21s	16.09s	4.76s	1.98s	39	5	493.5mb	54.5mb	56.0mb
59.50s	15.53s	5.01s	1.96s	33	5	473.5mb	59.0mb	61.0mb
57.95s	13.88s	4.86s	1.56s	27	4	425.5mb	62.0mb	66.0mb
56.55s	12.56s	4.99s	1.19s	22	3	376.0mb	66.5mb	71.0mb
57.08s	13.05s	5.11s	1.21s	22	3	381.5mb	72.0mb	76.0mb
`gcold:10,1000` **4DOF upper+lower T=2.0**								
37.45s	1.21s	0.81s	0.00s	1	0	22.0mb	49.5mb	56.0mb
37.08s	0.87s	0.87s	0.00s	0	0	15.5mb	45.0mb	61.0mb
37.08s	0.75s	0.75s	0.00s	0	0	15.5mb	45.0mb	66.0mb
37.06s	0.79s	0.79s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.09s	0.80s	0.80s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **4DOF upper+lower T=2.0**								
9.79s	5.16s	1.59s	1.04s	23	3	128.0mb	21.5mb	24.0mb
8.71s	4.13s	1.70s	0.78s	14	2	100.5mb	25.5mb	28.0mb
7.94s	3.27s	1.68s	0.39s	10	1	79.0mb	28.0mb	32.0mb
8.16s	3.53s	1.77s	0.43s	8	1	79.5mb	32.0mb	36.0mb
7.51s	2.89s	1.87s	0.00s	5	0	62.5mb	37.0mb	40.0mb
`perm:200,1` **4DOF upper+lower T=2.0**								
29.96s	23.72s	6.43s	6.12s	134	19	765.5mb	24.5mb	25.0mb
24.30s	17.94s	6.33s	3.81s	100	12	585.5mb	25.5mb	28.0mb
18.19s	12.01s	6.54s	1.63s	65	5	394.5mb	28.5mb	31.0mb
17.39s	11.28s	6.53s	1.30s	55	4	369.5mb	31.5mb	34.0mb
16.50s	10.29s	6.54s	0.95s	48	3	339.5mb	34.5mb	37.0mb
`perm:25,8` **4DOF upper+lower T=2.0**								
21.14s	14.85s	6.51s	2.76s	100	10	485.0mb	22.5mb	25.0mb
19.06s	12.79s	6.46s	1.90s	76	7	422.5mb	25.5mb	28.0mb
17.12s	10.93s	6.63s	1.15s	59	4	358.0mb	29.0mb	31.0mb
16.35s	10.18s	6.56s	0.88s	51	3	334.0mb	31.0mb	34.0mb
15.47s	9.33s	6.52s	0.63s	42	2	308.5mb	34.5mb	37.0mb
`twobitlong` **4DOF upper+lower T=2.0**								
144.94s	5.87s	4.57s	0.21s	23	2	79.0mb	17.5mb	18.0mb
145.26s	5.84s	4.59s	0.19s	20	1	77.5mb	19.0mb	20.0mb
144.83s	5.71s	4.51s	0.20s	16	1	74.0mb	21.0mb	22.0mb
144.98s	5.89s	4.88s	0.13s	13	1	68.0mb	22.5mb	24.0mb
144.86s	5.69s	4.73s	0.00s	11	0	65.0mb	24.5mb	26.0mb
`twobitshort` **4DOF upper+lower T=2.0**								
23.62s	2.10s	1.13s	0.21s	9	1	40.5mb	17.0mb	18.0mb
23.76s	2.13s	1.24s	0.13s	8	1	38.0mb	19.0mb	20.0mb
23.17s	1.76s	1.22s	0.00s	5	0	31.0mb	20.5mb	22.0mb
23.10s	1.60s	1.19s	0.00s	3	0	28.0mb	22.0mb	24.0mb
23.03s	1.72s	1.37s	0.00s	2	0	26.5mb	24.0mb	26.0mb

Table A.43: Raw data for 4DOF: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **4DOF upper+lower T=3.0**								
								26.0mb
								29.0mb
								32.0mb
46.22s	24.02s	13.96s	6.59s	85	50	851.0mb	35.0mb	35.0mb
								38.0mb
`gcold:1,0` **4DOF upper+lower T=3.0**								
15.95s	6.93s	4.84s	0.36s	19	1	218.0mb	53.0mb	56.0mb
14.00s	5.07s	4.93s	0.00s	12	0	155.0mb	49.5mb	61.0mb
14.07s	5.24s	5.09s	0.00s	11	0	155.0mb	54.0mb	66.0mb
14.10s	5.17s	5.01s	0.00s	10	0	155.5mb	58.0mb	71.0mb
14.28s	5.32s	5.15s	0.00s	9	0	155.5mb	62.5mb	76.0mb
`gcold:1,1000` **4DOF upper+lower T=3.0**								
59.53s	15.59s	4.93s	1.97s	37	5	475.5mb	54.0mb	56.0mb
58.49s	14.40s	4.79s	1.57s	31	4	439.0mb	58.5mb	61.0mb
58.10s	14.15s	5.03s	1.58s	27	4	428.0mb	63.0mb	66.0mb
56.69s	12.53s	4.88s	1.18s	22	3	377.5mb	67.5mb	71.0mb
57.67s	13.10s	5.08s	1.23s	22	3	383.0mb	73.0mb	76.0mb
`gcold:10,1000` **4DOF upper+lower T=3.0**								
37.45s	1.10s	0.70s	0.00s	1	0	22.0mb	49.5mb	56.0mb
37.11s	0.81s	0.81s	0.00s	0	0	15.5mb	45.0mb	61.0mb
37.08s	0.92s	0.92s	0.00s	0	0	15.5mb	45.0mb	66.0mb
37.07s	0.83s	0.83s	0.00s	0	0	15.5mb	45.0mb	71.0mb
37.08s	0.94s	0.94s	0.00s	0	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **4DOF upper+lower T=3.0**								
9.65s	5.05s	1.63s	1.04s	22	3	124.5mb	22.0mb	24.0mb
8.71s	4.05s	1.66s	0.78s	14	2	100.5mb	25.5mb	28.0mb
7.95s	3.29s	1.69s	0.39s	10	1	79.0mb	28.0mb	32.0mb
8.17s	3.61s	1.84s	0.44s	8	1	79.5mb	32.0mb	36.0mb
7.51s	2.86s	1.85s	0.00s	5	0	62.5mb	37.0mb	40.0mb
`perm:200,1` **4DOF upper+lower T=3.0**								
29.96s	23.63s	6.39s	6.08s	134	19	765.5mb	24.5mb	25.0mb
21.64s	15.41s	6.38s	2.84s	86	9	500.0mb	27.0mb	28.0mb
18.09s	11.91s	6.47s	1.61s	62	5	393.0mb	29.0mb	31.0mb
17.39s	11.22s	6.48s	1.30s	55	4	369.5mb	31.5mb	34.0mb
16.50s	10.34s	6.58s	0.94s	48	3	339.5mb	34.5mb	37.0mb
`perm:25,8` **4DOF upper+lower T=3.0**								
20.00s	13.71s	6.48s	2.20s	87	8	449.0mb	24.0mb	25.0mb
18.55s	12.45s	6.63s	1.69s	72	6	403.5mb	26.5mb	28.0mb
17.11s	10.84s	6.45s	1.13s	59	4	358.0mb	29.0mb	31.0mb
16.35s	10.21s	6.63s	0.88s	51	3	334.0mb	31.0mb	34.0mb
15.50s	9.40s	6.60s	0.62s	42	2	308.5mb	34.5mb	37.0mb
`twobitlong` **4DOF upper+lower T=3.0**								
145.38s	5.82s	4.46s	0.22s	23	2	79.0mb	17.5mb	18.0mb
145.69s	5.98s	4.65s	0.21s	20	1	77.5mb	19.0mb	20.0mb
145.25s	5.84s	4.62s	0.19s	16	1	74.0mb	21.0mb	22.0mb
144.99s	5.67s	4.65s	0.14s	13	1	68.0mb	22.5mb	24.0mb
144.84s	5.57s	4.66s	0.00s	11	0	65.0mb	24.5mb	26.0mb
`twobitshort` **4DOF upper+lower T=3.0**								
23.65s	2.11s	1.14s	0.22s	9	1	40.5mb	17.0mb	18.0mb
23.74s	2.07s	1.20s	0.13s	8	1	38.0mb	19.0mb	20.0mb
23.19s	1.75s	1.22s	0.00s	5	0	31.0mb	20.5mb	22.0mb
23.13s	1.71s	1.29s	0.00s	3	0	28.0mb	22.0mb	24.0mb
23.05s	1.70s	1.35s	0.00s	2	0	26.5mb	24.0mb	26.0mb

Table A.44: Raw data for 4DOF: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **4DOF upper+lower T=2.0**								
17.22s	7.43s	2.37s	1.29s	27	4	169.0mb	25.5mb	28.0mb
18.16s	8.17s	2.18s	1.56s	27	4	187.5mb	29.5mb	31.0mb
17.33s	7.40s	2.33s	1.29s	20	3	166.0mb	31.5mb	34.0mb
16.06s	6.21s	2.49s	0.86s	15	2	137.0mb	34.0mb	37.0mb
16.29s	6.47s	2.53s	0.88s	15	2	140.0mb	37.0mb	40.0mb
`5nboyer:3` **4DOF upper+lower T=2.0**								
46.67s	23.29s	7.26s	4.72s	113	16	593.5mb	23.0mb	24.0mb
43.51s	20.08s	7.16s	3.60s	91	12	517.5mb	25.5mb	28.0mb
39.43s	16.11s	7.36s	1.83s	65	7	406.0mb	29.0mb	32.0mb
40.25s	16.97s	7.69s	1.81s	57	6	414.5mb	33.5mb	36.0mb
37.39s	14.14s	7.82s	0.60s	44	4	341.5mb	37.0mb	40.0mb
`5sboyer:4` **4DOF upper+lower T=2.0**								
79.39s	12.23s	2.63s	3.13s	56	10	264.0mb	19.0mb	20.0mb
72.98s	5.88s	2.59s	0.47s	28	4	125.0mb	21.0mb	24.0mb
74.84s	8.12s	2.59s	1.48s	29	4	172.0mb	25.5mb	28.0mb
74.96s	8.23s	2.54s	1.38s	24	3	176.0mb	30.5mb	32.0mb
72.16s	5.44s	2.73s	0.31s	15	2	113.0mb	33.5mb	36.0mb
`5twobitshort` **4DOF upper+lower T=2.0**								
115.92s	9.73s	3.98s	1.57s	64	8	222.5mb	17.0mb	18.0mb
114.04s	8.16s	3.88s	0.92s	49	5	185.0mb	19.0mb	20.0mb
112.72s	7.69s	4.05s	0.70s	39	4	171.5mb	21.0mb	22.0mb
114.46s	7.62s	4.00s	0.68s	35	3	167.0mb	22.5mb	24.0mb
113.38s	7.33s	4.24s	0.50s	30	3	157.0mb	24.0mb	26.0mb
`5earley:12` **4DOF upper+lower T=3.0**								
17.42s	7.56s	2.33s	1.35s	27	4	174.0mb	26.5mb	28.0mb
18.29s	8.56s	2.40s	1.66s	27	4	191.5mb	29.5mb	31.0mb
15.66s	5.84s	2.47s	0.53s	16	2	123.0mb	32.0mb	34.0mb
16.17s	6.33s	2.53s	0.91s	15	2	139.0mb	34.5mb	37.0mb
16.35s	6.38s	2.38s	0.89s	15	2	141.5mb	37.5mb	40.0mb
`5nboyer:3` **4DOF upper+lower T=3.0**								
45.84s	22.65s	7.39s	4.51s	110	15	575.0mb	23.0mb	24.0mb
43.08s	19.19s	7.31s	3.15s	83	10	491.5mb	26.5mb	28.0mb
40.01s	16.70s	7.49s	2.02s	64	7	418.5mb	30.0mb	32.0mb
40.25s	16.92s	7.58s	2.01s	54	6	417.5mb	34.5mb	36.0mb
37.36s	13.93s	7.66s	0.59s	44	4	341.5mb	37.0mb	40.0mb
`5sboyer:4` **4DOF upper+lower T=3.0**								
78.29s	11.53s	2.63s	2.81s	53	9	247.0mb	19.0mb	20.0mb
73.54s	6.82s	2.66s	0.88s	29	4	144.5mb	23.0mb	24.0mb
75.09s	8.13s	2.52s	1.48s	28	4	175.5mb	26.5mb	28.0mb
74.04s	6.98s	2.49s	1.11s	21	3	154.5mb	30.5mb	32.0mb
72.13s	5.41s	2.62s	0.34s	15	2	115.0mb	34.0mb	36.0mb
`5twobitshort` **4DOF upper+lower T=3.0**								
115.79s	9.42s	4.06s	1.35s	57	7	210.5mb	18.5mb	18.0mb
114.52s	8.07s	3.71s	0.96s	47	5	186.5mb	20.0mb	20.0mb
113.18s	7.82s	4.18s	0.71s	39	4	171.5mb	21.0mb	22.0mb
114.42s	7.78s	4.23s	0.70s	34	3	165.0mb	23.0mb	24.0mb
112.74s	7.46s	4.31s	0.52s	30	3	157.0mb	24.0mb	26.0mb

Table A.45: Raw data for 4DOF: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`gcbench:5` **4DOF upper T=3.0**								
								26.0mb
46.70s	24.50s	13.51s	5.31s	156	43	877.0mb	28.5mb	29.0mb
43.85s	21.36s	13.42s	1.47s	153	16	748.0mb	31.5mb	32.0mb
44.44s	22.05s	13.48s	1.64s	147	16	769.5mb	34.5mb	35.0mb
42.28s	20.35s	13.91s	0.95s	138	13	697.0mb	32.0mb	38.0mb
`gcold:1,0` **4DOF upper T=3.0**								
18.60s	9.49s	4.51s	1.08s	30	3	311.5mb	48.5mb	56.0mb
16.56s	7.35s	4.40s	0.72s	24	2	250.0mb	44.5mb	61.0mb
16.55s	7.44s	4.55s	0.72s	24	2	250.0mb	44.5mb	66.0mb
16.55s	7.36s	4.43s	0.73s	24	2	250.0mb	44.5mb	71.0mb
16.55s	7.61s	4.62s	0.72s	24	2	250.0mb	44.5mb	76.0mb
`gcold:1,1000` **4DOF upper T=3.0**								
61.55s	17.33s	4.56s	2.72s	47	7	552.0mb	53.5mb	56.0mb
60.13s	16.28s	4.74s	2.35s	41	6	507.5mb	57.5mb	61.0mb
60.26s	16.30s	4.62s	2.35s	39	6	510.0mb	62.0mb	66.0mb
58.70s	14.74s	4.60s	1.99s	34	5	458.5mb	64.5mb	71.0mb
58.80s	14.84s	4.62s	1.98s	33	5	460.0mb	69.0mb	76.0mb
`gcold:10,1000` **4DOF upper T=3.0**								
38.63s	2.38s	0.57s	0.44s	7	1	67.0mb	36.5mb	56.0mb
38.67s	2.36s	0.57s	0.44s	7	1	67.0mb	37.0mb	61.0mb
38.65s	2.29s	0.47s	0.44s	7	1	67.0mb	37.0mb	66.0mb
38.65s	2.42s	0.61s	0.44s	7	1	67.0mb	37.0mb	71.0mb
38.65s	2.59s	0.75s	0.45s	7	1	67.0mb	37.0mb	76.0mb
`nboyer:3` **4DOF upper T=3.0**								
10.45s	5.83s	1.45s	1.43s	59	11	151.5mb	21.0mb	24.0mb
10.48s	5.81s	1.36s	1.55s	58	11	152.5mb	21.5mb	28.0mb
10.46s	5.82s	1.45s	1.54s	57	11	151.0mb	23.0mb	32.0mb
10.46s	5.76s	1.39s	1.50s	57	11	151.0mb	23.0mb	36.0mb
10.47s	5.75s	1.37s	1.52s	57	11	151.0mb	23.0mb	40.0mb
`perm:200,1` **4DOF upper T=3.0**								
27.96s	21.53s	6.17s	5.49s	125	17	707.5mb	24.5mb	25.0mb
22.14s	15.93s	6.36s	3.23s	89	10	523.5mb	27.0mb	28.0mb
20.70s	14.30s	6.27s	2.62s	77	8	472.5mb	29.0mb	31.0mb
19.81s	13.68s	6.40s	2.27s	71	7	447.0mb	31.0mb	34.0mb
18.74s	12.60s	6.39s	1.92s	66	6	415.5mb	31.0mb	37.0mb
`perm:25,8` **4DOF upper T=3.0**								
21.30s	15.12s	6.31s	2.92s	107	13	499.0mb	24.0mb	25.0mb
20.15s	13.89s	6.31s	2.33s	98	11	456.5mb	25.5mb	28.0mb
19.83s	13.60s	6.31s	2.28s	95	11	453.0mb	25.0mb	31.0mb
19.36s	13.16s	6.36s	2.02s	91	10	435.0mb	26.0mb	34.0mb
19.37s	13.15s	6.29s	2.02s	91	10	435.0mb	26.0mb	37.0mb
`twobitlong` **4DOF upper T=3.0**								
145.69s	7.34s	4.06s	1.09s	65	12	136.0mb	17.0mb	18.0mb
145.26s	6.65s	3.87s	0.89s	60	11	122.0mb	17.0mb	20.0mb
145.07s	6.66s	3.94s	0.87s	59	11	120.5mb	17.5mb	22.0mb
145.07s	6.50s	3.75s	0.88s	59	11	120.5mb	17.5mb	24.0mb
144.93s	6.98s	4.29s	0.86s	59	11	120.5mb	17.5mb	26.0mb
`twobitshort` **4DOF upper T=3.0**								
25.69s	3.70s	0.99s	0.97s	40	9	88.5mb	16.5mb	18.0mb
25.14s	3.59s	0.92s	0.96s	39	9	87.5mb	17.0mb	20.0mb
25.12s	3.40s	0.92s	0.86s	38	9	83.5mb	17.5mb	22.0mb
25.14s	3.36s	0.89s	0.88s	38	9	83.5mb	17.5mb	24.0mb
25.10s	3.39s	0.89s	0.91s	38	9	83.5mb	17.5mb	26.0mb

Table A.46: Raw data for 4DOF: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Heap limit
`5earley:12` **4DOF upper T=3.0**								
19.95s	9.67s	2.12s	2.44s	57	12	232.5mb	26.0mb	28.0mb
20.75s	10.34s	2.04s	2.70s	56	12	246.0mb	28.5mb	31.0mb
19.44s	9.15s	2.23s	2.13s	51	11	214.0mb	30.5mb	34.0mb
19.76s	9.46s	2.16s	2.31s	51	11	222.5mb	32.5mb	37.0mb
18.09s	7.93s	2.34s	1.63s	46	10	182.0mb	34.5mb	40.0mb
`5nboyer:3` **4DOF upper T=3.0**								
45.94s	22.38s	6.90s	4.54s	144	23	582.5mb	23.0mb	24.0mb
45.49s	22.14s	7.04s	4.51s	129	21	572.0mb	27.0mb	28.0mb
43.68s	20.12s	7.05s	3.55s	114	18	515.0mb	30.5mb	32.0mb
41.99s	18.72s	7.40s	2.77s	106	16	472.0mb	34.0mb	36.0mb
42.14s	18.72s	7.22s	2.84s	105	16	474.0mb	36.5mb	40.0mb
`5sboyer:4` **4DOF upper T=3.0**								
80.71s	14.08s	2.44s	4.08s	83	17	309.5mb	19.0mb	20.0mb
76.00s	9.22s	2.44s	2.10s	61	12	207.0mb	22.5mb	24.0mb
78.67s	10.03s	2.47s	2.41s	60	12	224.0mb	26.0mb	28.0mb
78.73s	10.75s	2.38s	2.79s	59	12	240.0mb	29.5mb	32.0mb
76.97s	10.51s	2.50s	2.46s	55	11	227.0mb	32.5mb	36.0mb
`5twobitshort` **4DOF upper T=3.0**								
116.30s	10.47s	3.88s	1.91s	84	14	246.0mb	18.5mb	18.0mb
115.09s	9.89s	3.88s	1.75s	77	13	236.0mb	20.0mb	20.0mb
115.57s	9.73s	3.87s	1.66s	74	13	228.0mb	20.5mb	22.0mb
115.65s	9.67s	3.86s	1.66s	74	13	228.0mb	20.5mb	24.0mb
115.90s	9.76s	3.90s	1.67s	74	13	228.0mb	20.5mb	26.0mb

Table A.47: Raw data for 4DOF: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`5earley:12` **4DOF variable**								
19.54s	9.07s	2.05s	2.26s	72	16	217.5mb	21.0mb	2.25
20.93s	10.84s	2.43s	2.81s	65	14	248.0mb	28.0mb	2.5
19.68s	9.42s	2.24s	2.46s	56	12	220.5mb	29.0mb	2.75
17.54s	7.58s	2.52s	1.59s	46	10	172.5mb	30.5mb	3.0
`5nboyer:3` **4DOF variable**								
48.05s	24.95s	7.19s	5.64s	157	28	640.0mb	28.5mb	2.25
45.18s	21.84s	7.14s	4.30s	129	22	557.0mb	29.5mb	2.5
43.83s	20.48s	7.21s	3.65s	120	19	520.5mb	32.5mb	2.75
42.55s	19.15s	7.30s	3.00s	105	16	481.5mb	38.5mb	3.0
`5sboyer:4` **4DOF variable**								
78.57s	11.78s	2.48s	3.25s	79	17	262.0mb	24.5mb	2.25
78.24s	11.19s	2.32s	2.99s	67	14	249.5mb	28.0mb	2.5
77.64s	10.29s	2.35s	2.51s	59	12	228.0mb	32.0mb	2.75
76.29s	9.24s	2.42s	2.08s	53	11	204.0mb	32.0mb	3.0
`5twobitshort` **4DOF variable**								
117.58s	11.83s	3.85s	2.60s	115	23	290.5mb	15.5mb	2.25
117.60s	11.77s	3.75s	2.71s	105	19	285.5mb	17.5mb	2.5
115.88s	9.93s	3.88s	1.73s	83	14	233.5mb	18.5mb	2.75
115.47s	9.67s	3.80s	1.66s	74	13	228.0mb	20.5mb	3.0

Table A.48: Raw data for 4DOF: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	CPU Full	Major GCs	Full GCs	Marking	Heap used	Load
`gcbench:5` **4DOF variable**								
46.54s	24.39s	13.85s	2.03s	174	21	837.5mb	32.5mb	2.25
45.51s	22.98s	13.30s	1.90s	171	19	806.5mb	32.5mb	2.5
42.80s	20.36s	13.53s	1.05s	143	14	708.0mb	35.0mb	2.75
42.44s	20.16s	13.71s	0.96s	138	13	697.0mb	32.0mb	3.0
`gcold:1,0` **4DOF variable**								
21.96s	12.90s	4.53s	2.16s	43	6	436.5mb	43.5mb	2.25
18.79s	9.77s	4.52s	1.10s	31	3	322.0mb	47.5mb	2.5
19.33s	10.29s	4.51s	1.46s	33	4	345.5mb	47.0mb	2.75
16.59s	7.65s	4.68s	0.73s	24	2	250.0mb	44.5mb	3.0
`gcold:1,1000` **4DOF variable**								
60.83s	16.85s	4.59s	2.71s	43	7	537.5mb	63.5mb	2.25
58.84s	14.81s	4.52s	1.95s	35	5	461.0mb	69.0mb	2.5
58.24s	14.24s	4.61s	1.94s	31	5	445.0mb	75.0mb	2.75
57.06s	13.17s	4.76s	1.56s	27	4	400.0mb	77.5mb	3.0
`gcold:10,1000` **4DOF variable**								
39.17s	2.99s	0.62s	0.81s	8	2	91.5mb	36.0mb	2.25
38.82s	2.58s	0.60s	0.41s	8	1	73.0mb	37.5mb	2.5
39.18s	3.01s	0.70s	0.82s	7	2	89.0mb	38.0mb	2.75
38.69s	2.32s	0.51s	0.44s	7	1	67.0mb	37.0mb	3.0
`nboyer:3` **4DOF variable**								
12.75s	8.13s	1.42s	2.56s	84	18	212.5mb	20.0mb	2.25
11.88s	7.13s	1.32s	2.30s	68	15	189.5mb	21.0mb	2.5
11.14s	6.48s	1.36s	1.90s	65	13	169.5mb	21.5mb	2.75
10.48s	5.85s	1.44s	1.54s	57	11	151.0mb	23.0mb	3.0
`perm:200,1` **4DOF variable**								
21.83s	15.58s	6.26s	3.19s	87	10	513.5mb	27.5mb	2.25
19.77s	13.49s	6.27s	2.27s	74	7	447.5mb	29.0mb	2.5
19.06s	12.85s	6.44s	1.95s	69	6	421.5mb	30.0mb	2.75
18.74s	12.58s	6.38s	1.92s	66	6	415.5mb	31.0mb	3.0
`perm:25,8` **4DOF variable**								
25.75s	19.40s	6.22s	4.71s	141	19	635.5mb	23.0mb	2.25
21.98s	15.83s	6.39s	3.17s	112	14	519.5mb	24.0mb	2.5
20.75s	14.55s	6.30s	2.74s	98	11	480.0mb	25.5mb	2.75
19.36s	13.18s	6.34s	2.03s	91	10	435.0mb	26.0mb	3.0
`twobitlong` **4DOF variable**								
146.47s	7.89s	3.92s	1.46s	84	18	158.0mb	16.0mb	2.25
147.43s	7.19s	3.81s	1.18s	74	15	140.5mb	17.0mb	2.5
146.62s	7.11s	4.06s	1.09s	64	14	131.0mb	18.0mb	2.75
145.14s	6.72s	4.01s	0.86s	59	11	120.5mb	17.5mb	3.0
`twobitshort` **4DOF variable**								
25.97s	4.32s	0.93s	1.31s	53	14	109.5mb	15.5mb	2.25
25.35s	3.70s	0.93s	1.07s	45	11	92.5mb	15.0mb	2.5
26.36s	3.75s	0.94s	1.03s	42	10	91.5mb	17.0mb	2.75
25.12s	3.38s	0.89s	0.90s	38	9	83.5mb	17.5mb	3.0

Table A.49: Raw data for 4DOF: *variable* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **ROF upper+lower T=2.0**						
164.55s	141.72s	3.09s	641	6182mb	23.5mb	26.0mb
75.83s	53.75s	9.60s	196	2228mb	25.0mb	29.0mb
58.59s	36.52s	10.62s	111	1475mb	27.0mb	32.0mb
52.79s	30.63s	11.04s	87	1238mb	28.0mb	35.0mb
49.46s	27.74s	11.74s	71	1098mb	31.5mb	38.0mb
`gcold:1,0` **ROF upper+lower T=2.0**						
23.15s	14.20s	3.93s	20	476.5mb	46.0mb	56.0mb
20.03s	11.24s	4.20s	14	365.0mb	48.0mb	61.0mb
18.24s	9.44s	4.27s	11	302.0mb	49.5mb	66.0mb
17.25s	8.38s	4.35s	9	264.0mb	51.5mb	71.0mb
16.37s	7.59s	4.56s	7	229.5mb	53.5mb	76.0mb
`gcold:1,1000` **ROF upper+lower T=2.0**						
58.90s	15.47s	4.28s	21	503.0mb	46.5mb	56.0mb
55.72s	12.09s	4.20s	15	391.5mb	49.0mb	61.0mb
54.02s	10.47s	4.35s	12	331.0mb	50.5mb	66.0mb
53.14s	9.57s	4.40s	10	296.0mb	53.0mb	71.0mb
52.27s	8.89s	4.76s	8	263.0mb	55.5mb	76.0mb
`gcold:10,1000` **ROF upper+lower T=2.0**						
38.08s	1.89s	0.40s	2	50.5mb	46.5mb	56.0mb
37.26s	1.28s	0.54s	1	32.5mb	49.0mb	61.0mb
38.13s	2.14s	0.67s	2	49.5mb	52.0mb	66.0mb
37.39s	1.32s	0.61s	1	32.0mb	51.5mb	71.0mb
37.47s	1.26s	0.52s	1	32.5mb	54.0mb	76.0mb
`nboyer:3` **ROF upper+lower T=2.0**						
						24.0mb
9.37s	4.79s	1.21s	14	169.5mb	25.5mb	28.0mb
7.89s	3.45s	1.52s	7	106.5mb	27.0mb	32.0mb
7.18s	2.63s	1.47s	4	74.0mb	30.0mb	36.0mb
6.97s	2.48s	1.60s	3	67.5mb	32.5mb	40.0mb
`perm:200,1` **ROF upper+lower T=2.0**						
						25.0mb
38.59s	32.50s	2.50s	114	1345mb	26.0mb	28.0mb
31.16s	25.19s	3.26s	82	1027mb	26.5mb	31.0mb
20.99s	15.09s	4.39s	43	591.5mb	27.5mb	34.0mb
18.60s	12.71s	4.69s	33	492.5mb	29.0mb	37.0mb
`perm:25,8` **ROF upper+lower T=2.0**						
15.74s	9.79s	4.65s	37	362.0mb	22.5mb	25.0mb
12.36s	6.38s	4.88s	25	257.0mb	23.5mb	28.0mb
12.92s	7.05s	5.05s	24	280.5mb	20.5mb	31.0mb
13.85s	7.94s	5.01s	24	316.0mb	23.5mb	34.0mb
14.84s	8.92s	5.01s	24	352.0mb	26.5mb	37.0mb
`twobitlong` **ROF upper+lower T=2.0**						
144.12s	5.68s	4.19s	15	102.0mb	17.5mb	18.0mb
145.44s	5.66s	4.42s	12	92.0mb	17.5mb	20.0mb
145.37s	5.65s	4.49s	10	87.5mb	19.5mb	22.0mb
144.65s	5.16s	4.44s	7	71.0mb	19.0mb	24.0mb
144.60s	5.32s	4.66s	6	68.5mb	19.5mb	26.0mb
`twobitshort` **ROF upper+lower T=2.0**						
23.53s	1.90s	0.94s	6	46.0mb	17.5mb	18.0mb
23.48s	1.97s	1.05s	5	44.5mb	18.0mb	20.0mb
23.29s	1.68s	0.96s	4	39.5mb	18.0mb	22.0mb
23.74s	1.59s	0.94s	3	36.5mb	19.5mb	24.0mb
23.97s	1.68s	1.06s	3	37.5mb	20.5mb	26.0mb

Table A.50: Raw data for ROF: *upper+lower* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **ROF upper+lower T=3.0**						
						26.0mb
						29.0mb
51.96s	30.05s	11.49s	75	1138mb	30.5mb	32.0mb
47.33s	25.25s	11.97s	55	958.0mb	34.0mb	35.0mb
46.35s	24.23s	11.94s	48	910.5mb	36.5mb	38.0mb
`gcold:1,0` **ROF upper+lower T=3.0**						
17.26s	8.51s	4.44s	9	265.5mb	51.5mb	56.0mb
16.21s	7.34s	4.58s	7	223.5mb	55.0mb	61.0mb
15.48s	6.64s	4.74s	5	195.5mb	58.5mb	66.0mb
15.48s	6.67s	4.89s	5	191.5mb	60.5mb	71.0mb
15.11s	6.23s	4.91s	4	180.0mb	63.5mb	76.0mb
`gcold:1,1000` **ROF upper+lower T=3.0**						
53.20s	9.81s	4.60s	10	299.5mb	52.0mb	56.0mb
52.32s	8.51s	4.34s	8	262.5mb	57.0mb	61.0mb
52.00s	8.31s	4.56s	7	245.5mb	61.0mb	66.0mb
51.71s	8.19s	4.94s	6	230.5mb	66.0mb	71.0mb
51.36s	7.68s	4.93s	5	214.0mb	68.0mb	76.0mb
`gcold:10,1000` **ROF upper+lower T=3.0**						
38.14s	2.10s	0.64s	2	50.5mb	53.0mb	56.0mb
38.22s	2.11s	0.65s	2	51.0mb	56.0mb	61.0mb
37.58s	1.50s	0.75s	1	32.5mb	59.0mb	66.0mb
36.86s	0.76s	0.76s	0	15.5mb	45.0mb	71.0mb
36.86s	0.67s	0.67s	0	15.5mb	45.0mb	76.0mb
`nboyer:3` **ROF upper+lower T=3.0**						
						24.0mb
						28.0mb
						32.0mb
6.70s	2.27s	1.66s	2	58.5mb	35.5mb	36.0mb
6.95s	2.28s	1.61s	2	61.0mb	39.0mb	40.0mb
`perm:200,1` **ROF upper+lower T=3.0**						
						25.0mb
28.05s	21.94s	3.39s	71	899.0mb	26.5mb	28.0mb
17.78s	11.79s	4.64s	31	465.5mb	28.5mb	31.0mb
14.97s	9.05s	5.19s	20	350.5mb	31.0mb	34.0mb
14.03s	7.95s	5.24s	16	309.5mb	33.0mb	37.0mb
`perm:25,8` **ROF upper+lower T=3.0**						
13.50s	7.59s	5.03s	24	296.5mb	23.5mb	25.0mb
14.65s	8.65s	4.99s	24	332.0mb	27.0mb	28.0mb
12.85s	6.97s	5.38s	17	264.5mb	29.0mb	31.0mb
12.98s	7.06s	5.35s	15	267.0mb	30.5mb	34.0mb
12.38s	6.54s	5.67s	13	243.5mb	31.5mb	37.0mb
`twobitlong` **ROF upper+lower T=3.0**						
145.56s	5.51s	4.45s	10	84.5mb	20.0mb	18.0mb
143.33s	5.35s	4.61s	7	73.5mb	19.5mb	20.0mb
144.65s	5.30s	4.56s	6	71.5mb	23.5mb	22.0mb
143.61s	5.34s	4.66s	5	70.5mb	25.5mb	24.0mb
143.56s	5.18s	4.72s	4	63.0mb	23.5mb	26.0mb
`twobitshort` **ROF upper+lower T=3.0**						
24.06s	1.93s	1.00s	5	44.0mb	17.5mb	18.0mb
23.16s	1.66s	1.04s	3	36.5mb	20.5mb	20.0mb
23.03s	1.52s	1.14s	2	31.5mb	20.5mb	22.0mb
23.18s	1.45s	1.06s	2	31.0mb	22.0mb	24.0mb
22.89s	1.29s	1.08s	1	27.0mb	23.0mb	26.0mb

Table A.51: Raw data for ROF: *upper+lower* runs, one iteration, continued

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
5earley:12 **ROF upper+lower T=2.0**						
12.78s	2.61s	2.23s	5	72.5mb	22.5mb	28.0mb
12.32s	2.51s	2.03s	5	74.5mb	25.0mb	31.0mb
11.91s	2.14s	2.08s	4	63.0mb	19.5mb	34.0mb
12.48s	2.66s	2.16s	4	75.0mb	24.5mb	37.0mb
12.52s	2.85s	2.19s	4	79.0mb	28.0mb	40.0mb
5nboyer:3 **ROF upper+lower T=2.0**						
						24.0mb
46.54s	24.04s	5.84s	77	890.5mb	26.0mb	28.0mb
36.88s	14.74s	6.59s	35	492.5mb	27.5mb	32.0mb
33.98s	11.69s	6.87s	22	371.0mb	30.0mb	36.0mb
33.83s	11.27s	6.86s	19	361.0mb	32.5mb	40.0mb
5sboyer:4 **ROF upper+lower T=2.0**						
						20.0mb
						24.0mb
71.38s	5.99s	2.27s	15	183.0mb	26.5mb	28.0mb
68.19s	2.95s	2.52s	5	77.0mb	25.0mb	32.0mb
68.22s	2.68s	2.50s	4	69.5mb	24.0mb	36.0mb
5twobitshort **ROF upper+lower T=2.0**						
114.30s	7.84s	3.61s	30	224.0mb	18.0mb	18.0mb
115.37s	7.96s	3.94s	27	217.0mb	18.0mb	20.0mb
115.70s	7.40s	3.82s	24	201.5mb	18.5mb	22.0mb
114.27s	6.93s	3.82s	19	183.5mb	20.0mb	24.0mb
116.70s	6.70s	4.21s	15	164.5mb	21.0mb	26.0mb
5earley:12 **ROF upper+lower T=3.0**						
12.23s	2.46s	2.17s	4	69.5mb	23.0mb	28.0mb
12.54s	2.90s	2.28s	4	78.5mb	27.5mb	31.0mb
12.97s	3.08s	2.10s	4	89.0mb	31.5mb	34.0mb
13.27s	3.41s	2.16s	4	96.5mb	36.0mb	37.0mb
12.44s	2.65s	2.33s	2	71.0mb	32.5mb	40.0mb
5nboyer:3 **ROF upper+lower T=3.0**						
						24.0mb
						28.0mb
						32.0mb
						36.0mb
32.14s	9.81s	7.44s	11	284.0mb	40.0mb	40.0mb
5sboyer:4 **ROF upper+lower T=3.0**						
						20.0mb
						24.0mb
						28.0mb
68.55s	2.96s	2.36s	4	80.5mb	28.5mb	32.0mb
69.09s	3.56s	2.50s	4	93.5mb	35.5mb	36.0mb
5twobitshort **ROF upper+lower T=3.0**						
116.09s	8.10s	3.92s	27	219.0mb	18.0mb	18.0mb
						20.0mb
114.56s	6.25s	4.18s	13	150.5mb	23.0mb	22.0mb
114.82s	6.13s	4.25s	10	143.0mb	24.5mb	24.0mb
114.35s	5.85s	4.42s	8	133.0mb	26.5mb	26.0mb

Table A.52: Raw data for ROF: *upper+lower* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`gcbench:5` **ROF upper T=3.0**						
75.10s	53.22s	8.64s	196	2215mb	24.5mb	26.0mb
57.93s	35.73s	10.37s	112	1454mb	27.0mb	29.0mb
51.57s	29.75s	11.40s	77	1139mb	30.0mb	32.0mb
47.71s	25.63s	11.41s	60	999.0mb	32.5mb	35.0mb
						38.0mb
`gcold:1,0` **ROF upper T=3.0**						
17.14s	8.43s	4.37s	9	268.0mb	52.0mb	56.0mb
15.71s	6.95s	4.48s	6	214.0mb	55.0mb	61.0mb
15.67s	6.72s	4.40s	6	208.5mb	57.0mb	66.0mb
15.67s	7.09s	4.78s	6	208.5mb	57.0mb	71.0mb
15.67s	6.73s	4.42s	6	208.5mb	57.0mb	76.0mb
`gcold:1,1000` **ROF upper T=3.0**						
53.01s	9.42s	4.35s	10	298.5mb	54.5mb	56.0mb
52.04s	8.34s	4.35s	8	261.5mb	56.0mb	61.0mb
51.32s	7.55s	4.42s	6	231.5mb	60.5mb	66.0mb
50.99s	7.31s	4.66s	5	215.5mb	65.0mb	71.0mb
51.10s	7.65s	4.93s	5	215.0mb	68.5mb	76.0mb
`gcold:10,1000` **ROF upper T=3.0**						
37.41s	1.32s	0.60s	1	33.0mb	45.0mb	56.0mb
37.33s	1.29s	0.57s	1	33.0mb	45.0mb	61.0mb
37.38s	1.31s	0.58s	1	33.0mb	45.0mb	66.0mb
37.34s	1.28s	0.55s	1	33.0mb	45.0mb	71.0mb
37.37s	1.31s	0.58s	1	33.0mb	45.0mb	76.0mb
`nboyer:3` **ROF upper T=3.0**						
						24.0mb
						28.0mb
						32.0mb
						36.0mb
7.20s	2.74s	1.48s	7	77.0mb	37.0mb	40.0mb
`perm:200,1` **ROF upper T=3.0**						
						25.0mb
28.05s	22.03s	3.49s	71	899.5mb	26.5mb	28.0mb
17.76s	11.77s	4.64s	31	465.5mb	28.5mb	31.0mb
15.52s	9.52s	4.93s	22	374.0mb	31.0mb	34.0mb
14.08s	8.09s	5.25s	16	314.5mb	33.0mb	37.0mb
`perm:25,8` **ROF upper T=3.0**						
14.76s	8.79s	4.70s	33	331.5mb	23.5mb	25.0mb
13.92s	7.98s	4.89s	27	308.0mb	26.5mb	28.0mb
13.13s	7.14s	4.96s	24	279.5mb	26.5mb	31.0mb
13.16s	7.19s	4.95s	25	278.0mb	30.5mb	34.0mb
12.84s	6.97s	5.17s	22	267.0mb	28.5mb	37.0mb
`twobitlong` **ROF upper T=3.0**						
						18.0mb
143.98s	5.56s	4.61s	9	80.0mb	22.0mb	20.0mb
144.26s	5.48s	4.73s	7	71.5mb	22.0mb	22.0mb
146.37s	5.39s	4.56s	7	73.0mb	23.5mb	24.0mb
144.74s	5.30s	4.50s	6	70.5mb	25.5mb	26.0mb
`twobitshort` **ROF upper T=3.0**						
24.07s	1.98s	1.05s	6	44.0mb	18.0mb	18.0mb
23.17s	1.58s	1.02s	4	35.0mb	20.0mb	20.0mb
22.87s	1.38s	1.02s	3	29.5mb	22.5mb	22.0mb
23.30s	1.53s	1.14s	3	30.0mb	24.5mb	24.0mb
23.32s	1.38s	0.98s	3	30.0mb	24.5mb	26.0mb

Table A.53: Raw data for ROF: *upper* runs, one iteration

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Heap limit
`5earley:12` **ROF upper T=3.0**						
13.24s	3.54s	1.99s	11	107.5mb	25.0mb	28.0mb
13.11s	3.39s	2.00s	9	102.0mb	26.5mb	31.0mb
13.23s	3.44s	1.92s	9	105.0mb	32.0mb	34.0mb
13.31s	3.48s	1.94s	9	106.5mb	35.5mb	37.0mb
13.31s	3.54s	2.00s	9	106.5mb	35.5mb	40.0mb
`5nboyer:3` **ROF upper T=3.0**						
						24.0mb
						28.0mb
						32.0mb
						36.0mb
33.29s	10.99s	6.70s	24	342.5mb	37.0mb	40.0mb
`5sboyer:4` **ROF upper T=3.0**						
						20.0mb
						24.0mb
69.02s	3.55s	2.09s	11	109.0mb	23.5mb	28.0mb
69.20s	3.45s	2.04s	10	105.0mb	27.5mb	32.0mb
69.53s	3.93s	2.29s	9	110.5mb	34.5mb	36.0mb
`5twobitshort` **ROF upper T=3.0**						
114.64s	7.70s	3.61s	28	219.0mb	18.0mb	18.0mb
113.34s	6.78s	4.07s	19	173.0mb	21.5mb	20.0mb
113.18s	5.78s	3.93s	13	144.0mb	23.0mb	22.0mb
115.52s	6.34s	4.54s	11	141.0mb	24.5mb	24.0mb
115.03s	6.29s	4.44s	10	141.5mb	26.5mb	26.0mb

Table A.54: Raw data for ROF: *upper* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
`5earley:12` **ROF variable**						
15.43s	5.68s	1.55s	26	189.5mb	26.0mb	2.25
13.84s	4.03s	1.74s	16	135.5mb	19.5mb	2.5
13.04s	3.13s	1.78s	13	104.5mb	20.5mb	2.75
13.32s	3.63s	2.05s	9	106.5mb	35.5mb	3.0
`5nboyer:3` **ROF variable**						
39.13s	16.70s	5.24s	67	577.5mb	28.5mb	2.25
35.95s	13.52s	5.85s	43	450.5mb	33.0mb	2.5
34.43s	12.06s	6.36s	32	383.0mb	34.0mb	2.75
33.26s	10.97s	6.68s	24	342.5mb	37.0mb	3.0
`5sboyer:4` **ROF variable**						
71.73s	6.18s	1.85s	27	195.5mb	25.5mb	2.25
70.48s	4.88s	2.05s	16	151.5mb	28.5mb	2.5
70.44s	4.10s	2.17s	12	122.0mb	29.0mb	2.75
69.51s	3.80s	2.17s	9	110.5mb	34.5mb	3.0
`5twobitshort` **ROF variable**						
113.91s	6.82s	4.05s	19	173.5mb	20.5mb	2.25
113.56s	6.22s	4.05s	14	153.0mb	25.0mb	2.5
112.59s	5.51s	4.09s	11	135.5mb	23.5mb	2.75
112.48s	5.71s	4.33s	9	130.5mb	28.5mb	3.0

Table A.55: Raw data for ROF: *variable* runs, five iterations

CPU	CPU GC	CPU Promote	Major GCs	Marked	Heap used	Load
`gcbench:5` **ROF variable**						
57.92s	35.86s	10.22s	116	1480mb	29.0mb	2.25
46.34s	24.20s	11.42s	56	960.0mb	34.5mb	2.5
42.52s	20.53s	12.20s	35	769.5mb	45.5mb	2.75
40.05s	17.95s	12.55s	23	661.5mb	56.5mb	3.0
`gcold:1,0` **ROF variable**						
23.79s	15.00s	3.97s	21	503.5mb	46.0mb	2.25
18.15s	9.33s	4.10s	11	308.5mb	49.5mb	2.5
16.72s	7.98s	4.40s	8	248.5mb	53.0mb	2.75
15.65s	6.87s	4.55s	6	208.5mb	57.0mb	3.0
`gcold:1,1000` **ROF variable**						
59.31s	15.81s	4.10s	22	528.0mb	46.0mb	2.25
53.27s	9.79s	4.30s	11	315.0mb	51.0mb	2.5
51.80s	8.15s	4.48s	7	246.5mb	59.0mb	2.75
51.27s	7.63s	4.85s	5	215.5mb	73.5mb	3.0
`gcold:10,1000` **ROF variable**						
38.51s	2.43s	0.49s	3	68.0mb	46.0mb	2.25
37.30s	1.13s	0.44s	1	32.5mb	45.0mb	2.5
37.31s	1.31s	0.62s	1	32.0mb	45.0mb	2.75
37.35s	1.29s	0.56s	1	33.0mb	45.0mb	3.0
`nboyer:3` **ROF variable**						
7.95s	3.39s	1.01s	15	114.5mb	27.5mb	2.25
7.61s	3.24s	1.33s	11	96.5mb	30.0mb	2.5
7.18s	2.77s	1.40s	8	79.0mb	33.5mb	2.75
7.18s	2.70s	1.47s	7	77.0mb	37.0mb	3.0
`perm:200,1` **ROF variable**						
20.88s	14.90s	4.23s	42	594.5mb	28.0mb	2.25
16.37s	10.43s	4.83s	25	411.0mb	30.5mb	2.5
14.57s	8.62s	5.15s	18	336.0mb	32.5mb	2.75
13.40s	7.49s	5.38s	14	286.5mb	35.5mb	3.0
`perm:25,8` **ROF variable**						
19.83s	13.86s	4.01s	53	532.5mb	25.5mb	2.25
16.03s	10.05s	4.62s	33	385.0mb	30.0mb	2.5
14.17s	8.30s	5.08s	26	310.5mb	28.5mb	2.75
13.17s	7.22s	5.14s	22	273.0mb	33.5mb	3.0
`twobitlong` **ROF variable**						
144.39s	5.48s	4.44s	11	82.5mb	20.5mb	2.25
145.46s	5.23s	4.46s	8	73.0mb	23.0mb	2.5
142.82s	5.04s	4.55s	6	63.5mb	22.5mb	2.75
144.13s	5.39s	4.73s	6	69.0mb	28.5mb	3.0
`twobitshort` **ROF variable**						
23.21s	1.68s	1.01s	5	37.0mb	20.0mb	2.25
23.24s	1.58s	1.05s	4	34.5mb	23.5mb	2.5
22.83s	1.29s	0.95s	3	30.0mb	22.0mb	2.75
23.35s	1.54s	1.15s	3	30.0mb	24.5mb	3.0

Table A.56: Raw data for ROF: *variable* runs, one iteration

Appendix B

Processed data

This appendix contains charts and diagrams that provide a more concise basis for understanding how the collectors compare than do the tables in Appendix A.

The charts in Figures B.1 through B.7 show the measured mark/cons ratio of the collectors as a function of the effective inverse load L. The results are averaged over all (canonical) runs for a collector and benchmark. The thick line shows the expected mark/cons ratio for a non-generational copying collector.

The bar charts in Figures B.9 through B.14 show the results of comparing the collectors pairwise by total time, GC time, or space. Comparable runs from the two collectors are paired and the advantage number for the pair is computed; these numbers are then accumulated in the bar charts. The charts contain data from all benchmarks and give a rough idea of how the collector behaves on the benchmark suite as a whole.

In these charts, each bar is two points wide, and the height of a bar represents the number of comparable runs for which the better system had an advantage corresponding to the bar's distance from the origin.

The box-and-line diagrams in Figures B.15 through B.33 show the same data as the bar charts, but broken down by benchmark. The thin line shows the range of the results; the vertical line shows the mean, and the box shows the extent of the middle two quartiles of the set of results. A mutator time diagram is shown only for SC-vs-2GEN; the other generational collectors have mutator times that vary little from those of 2GEN.

Finally, the remembered-set profiles in Figures B.34 through B.53 show how the size of the remembered set varies over time in the DOF and ROF collectors, for some of the benchmarks. Time is measured in megabytes of allocation, and the size of the remembered set is measured in the number of objects in the set following GC. The graph shows both the size of the remembered set as well as the incidence of major and full garbage collections. The figures for the DOF and ROF collectors have slightly different interpretations. In the DOF profiles, the line that shows the size of the remembered set represents data from canonical runs. In the ROF profiles, there are two lines: the lower line was gathered from canonical runs, and the upper line from runs with none of the tuning parameters enabled. In all cases, the profile for entire runs—including initialization, which is substantial in some programs—are shown.

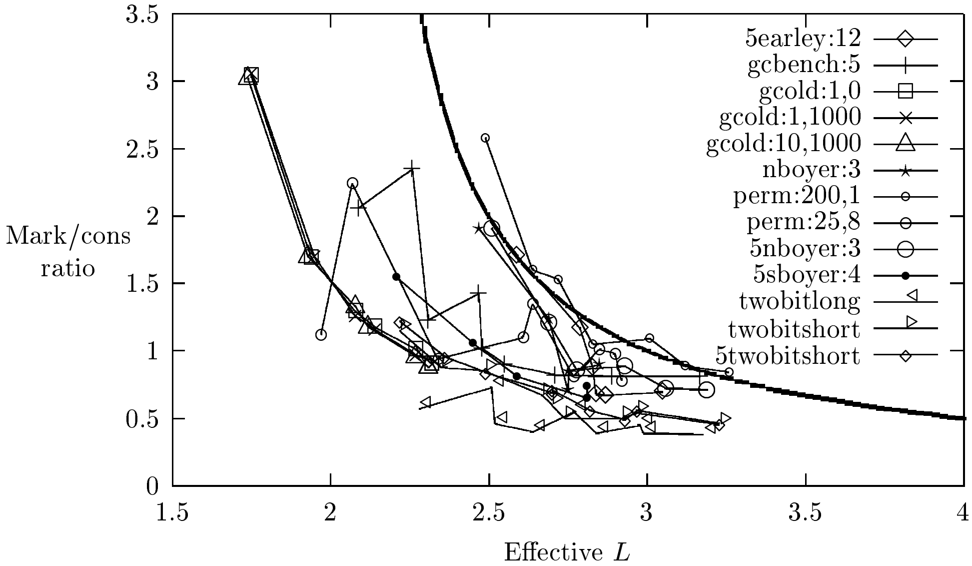

Figure B.1: SC mark/cons ratio as a function of L

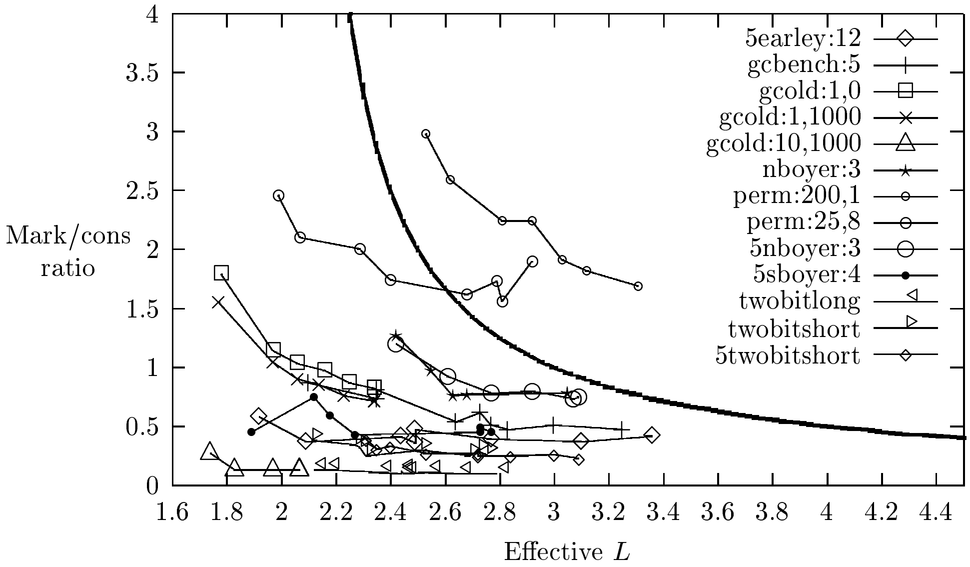

Figure B.2: 2GEN mark/cons ratio as a function of L

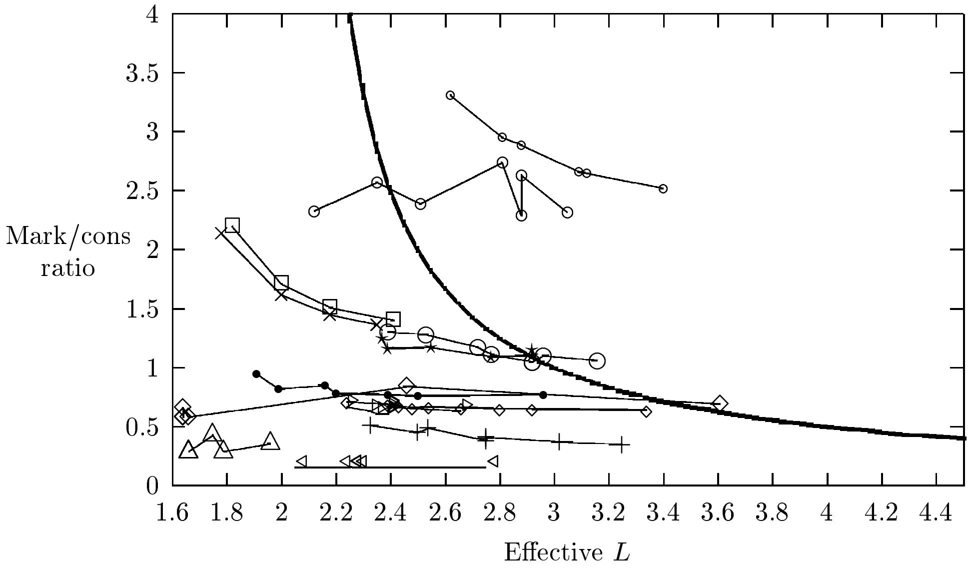

Figure B.3: 3GEN mark/cons ratio as a function of L

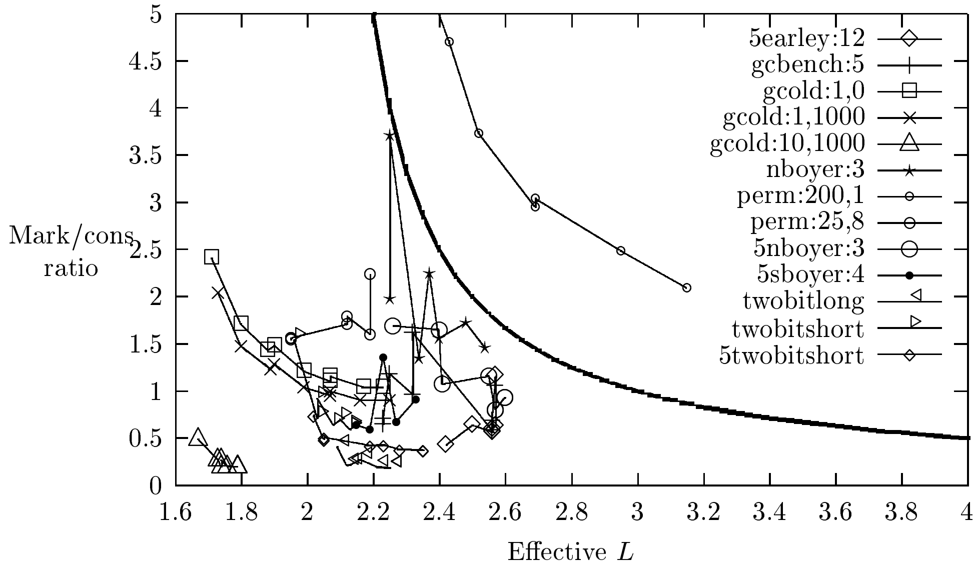

Figure B.4: 1DOF mark/cons ratio as a function of L

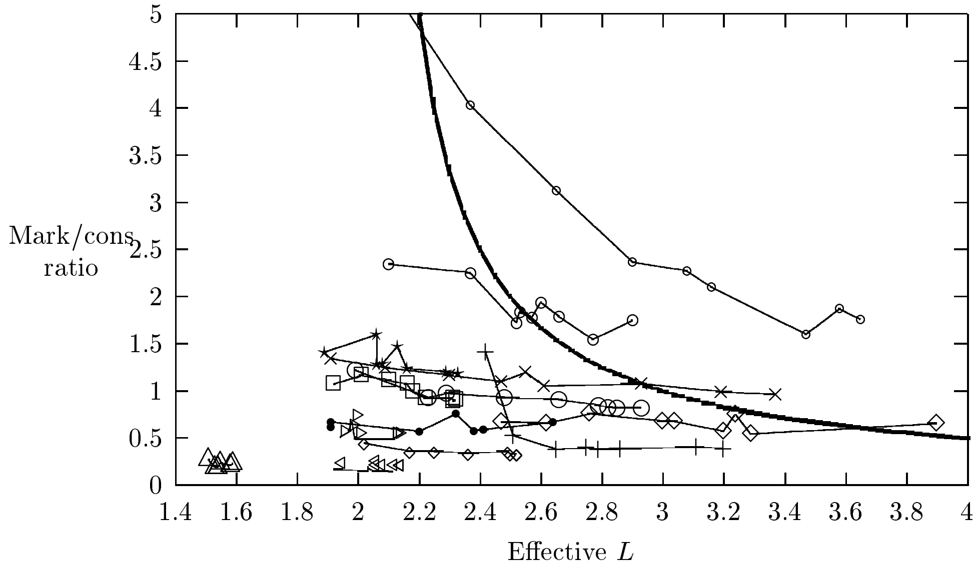

Figure B.5: 2DOF mark/cons ratio as a function of L

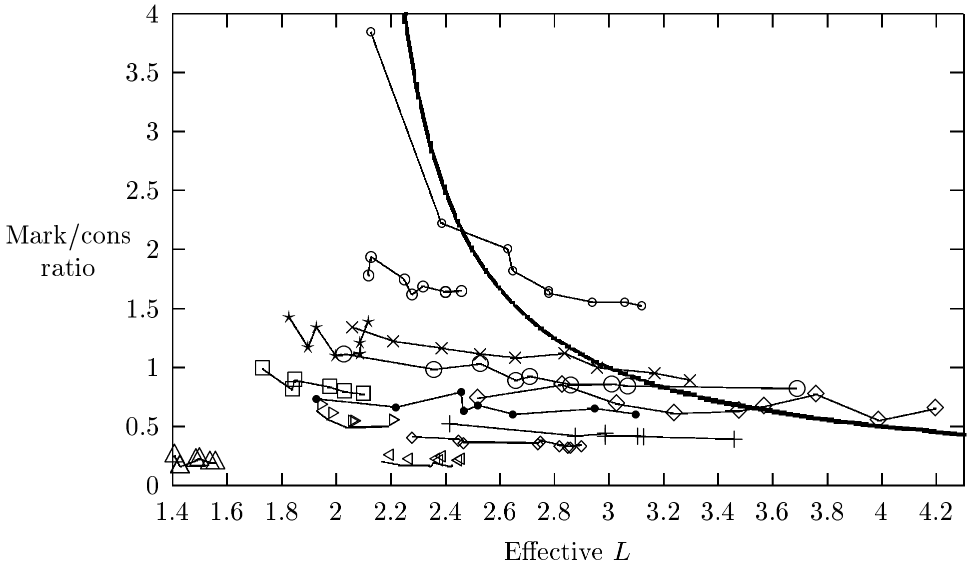

Figure B.6: 3DOF mark/cons ratio as a function of L

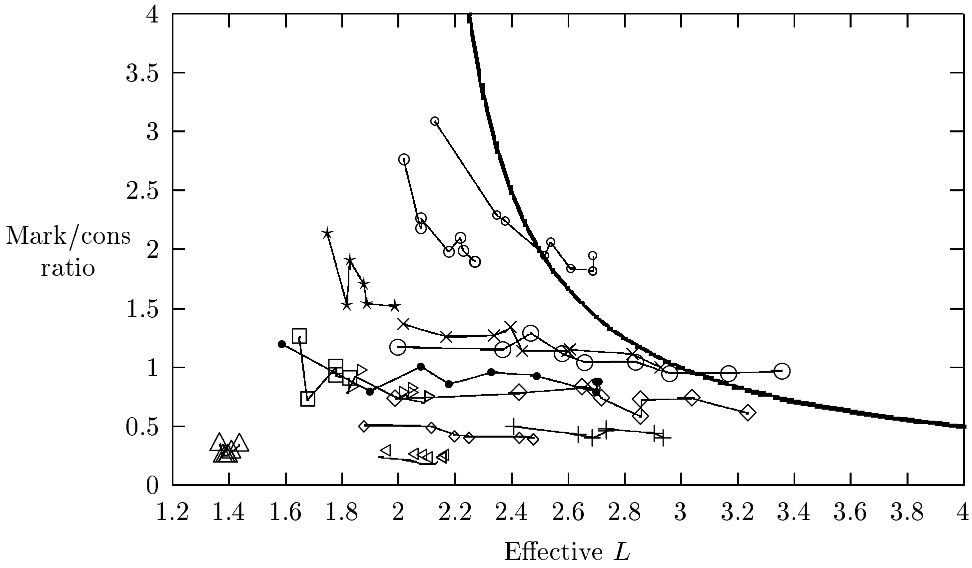

Figure B.7: 4DOF mark/cons ratio as a function of L

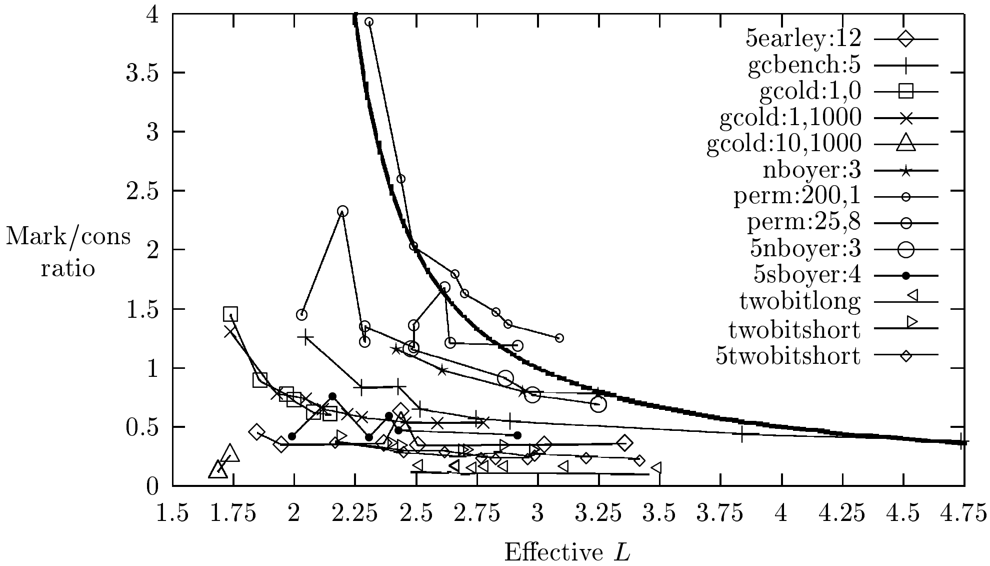

Figure B.8: ROF mark/cons ratio as a function of L

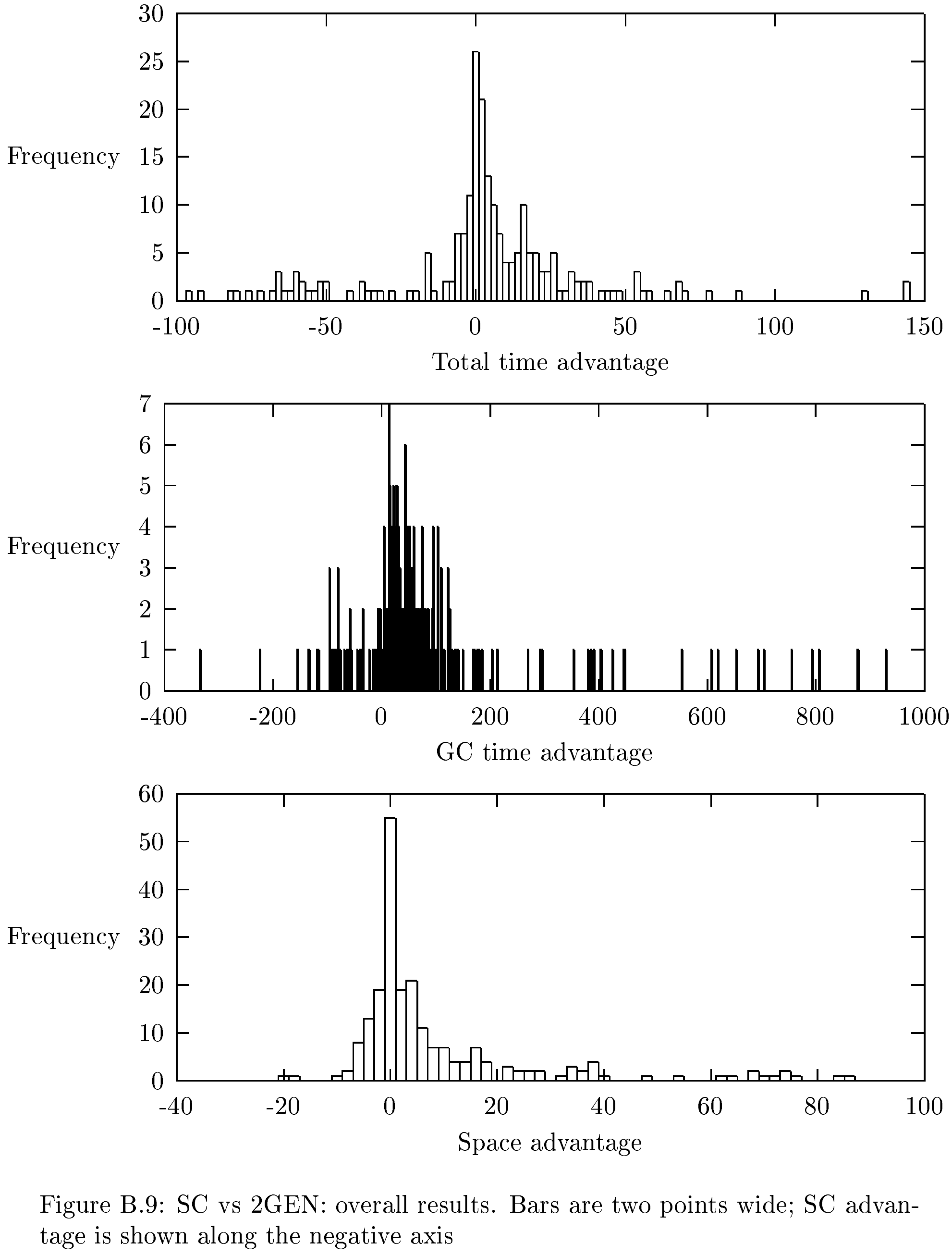

Figure B.9: SC vs 2GEN: overall results. Bars are two points wide; SC advantage is shown along the negative axis

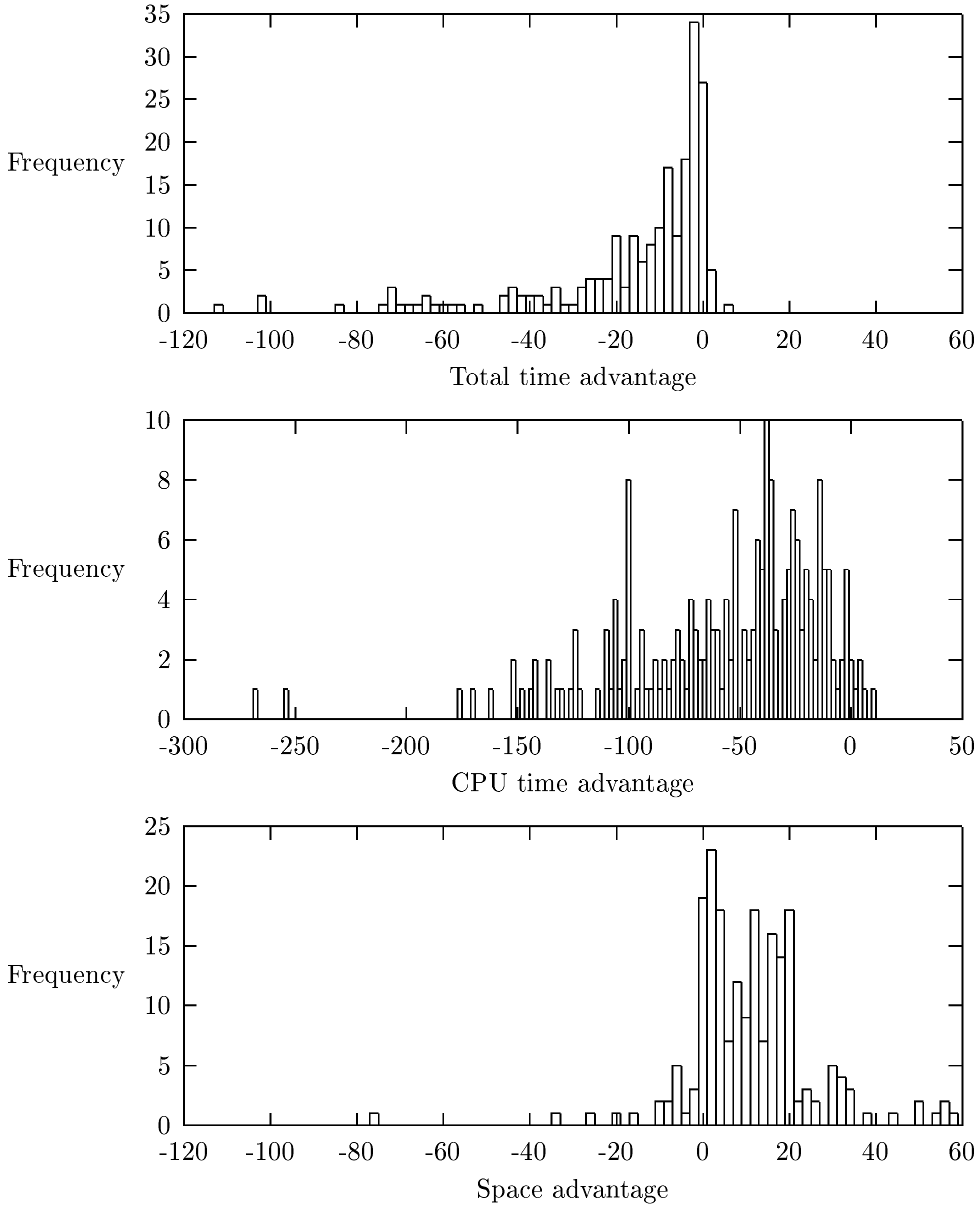

Figure B.10: 2GEN vs 1DOF: overall results. Bars are two points wide; 2GEN advantage is shown along the negative axis

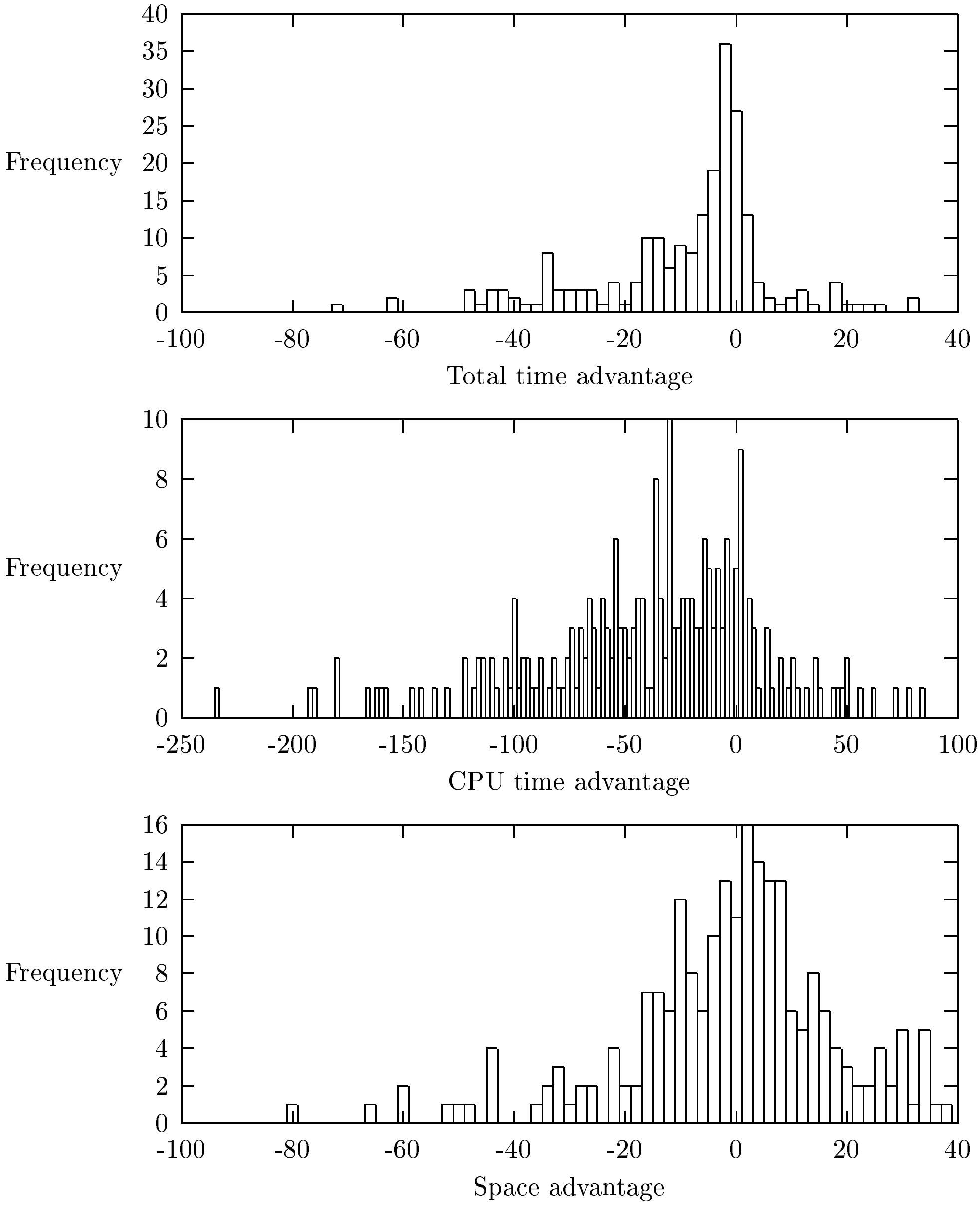

Figure B.11: 2GEN vs 2DOF: overall results. Bars are two points wide; 2GEN advantage is shown along the negative axis

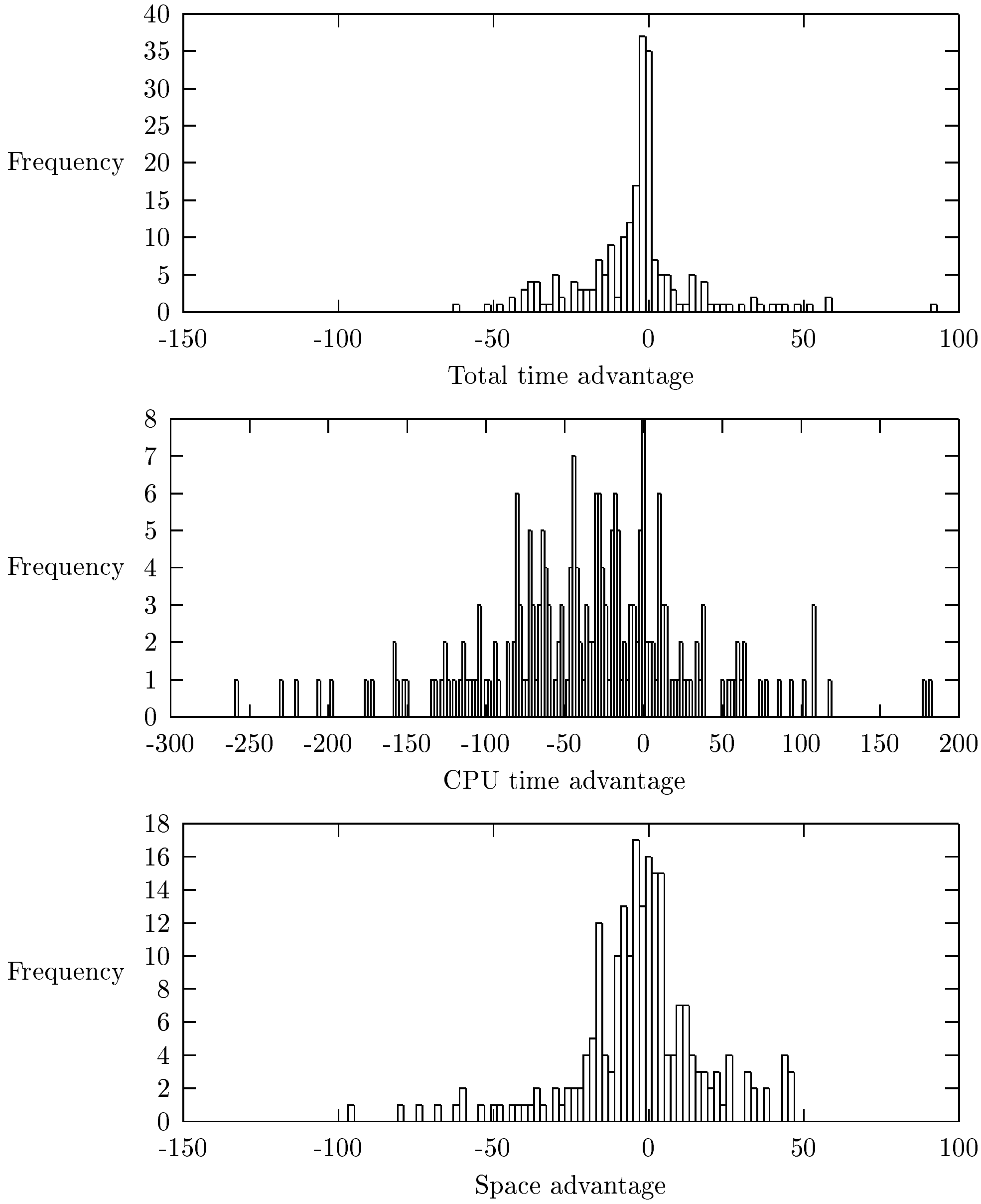

Figure B.12: 2GEN vs 3DOF: overall results. Bars are two points wide; 2GEN advantage is shown along the negative axis

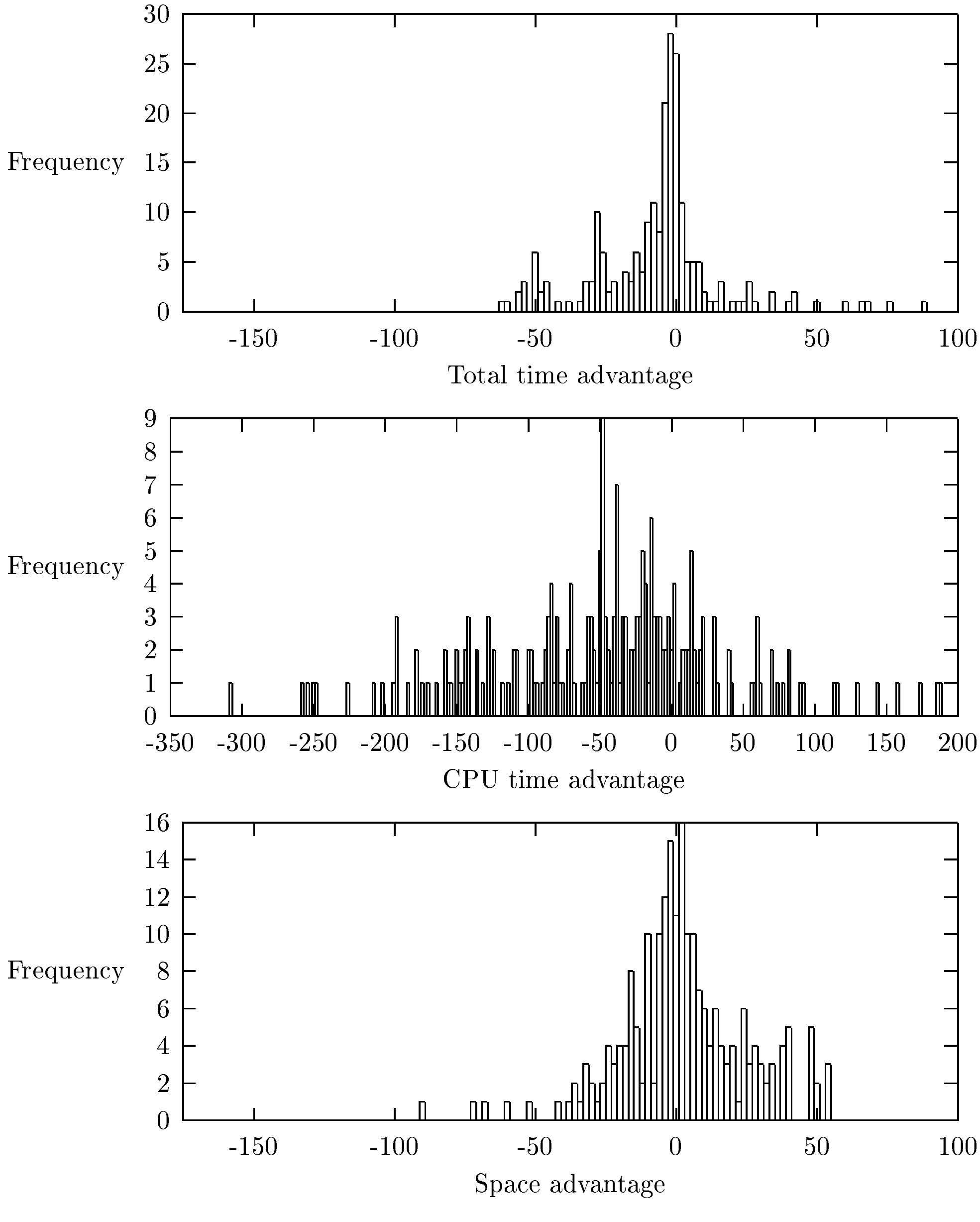

Figure B.13: 2GEN vs 4DOF: overall results. Bars are two points wide; 2GEN advantage is shown along the negative axis

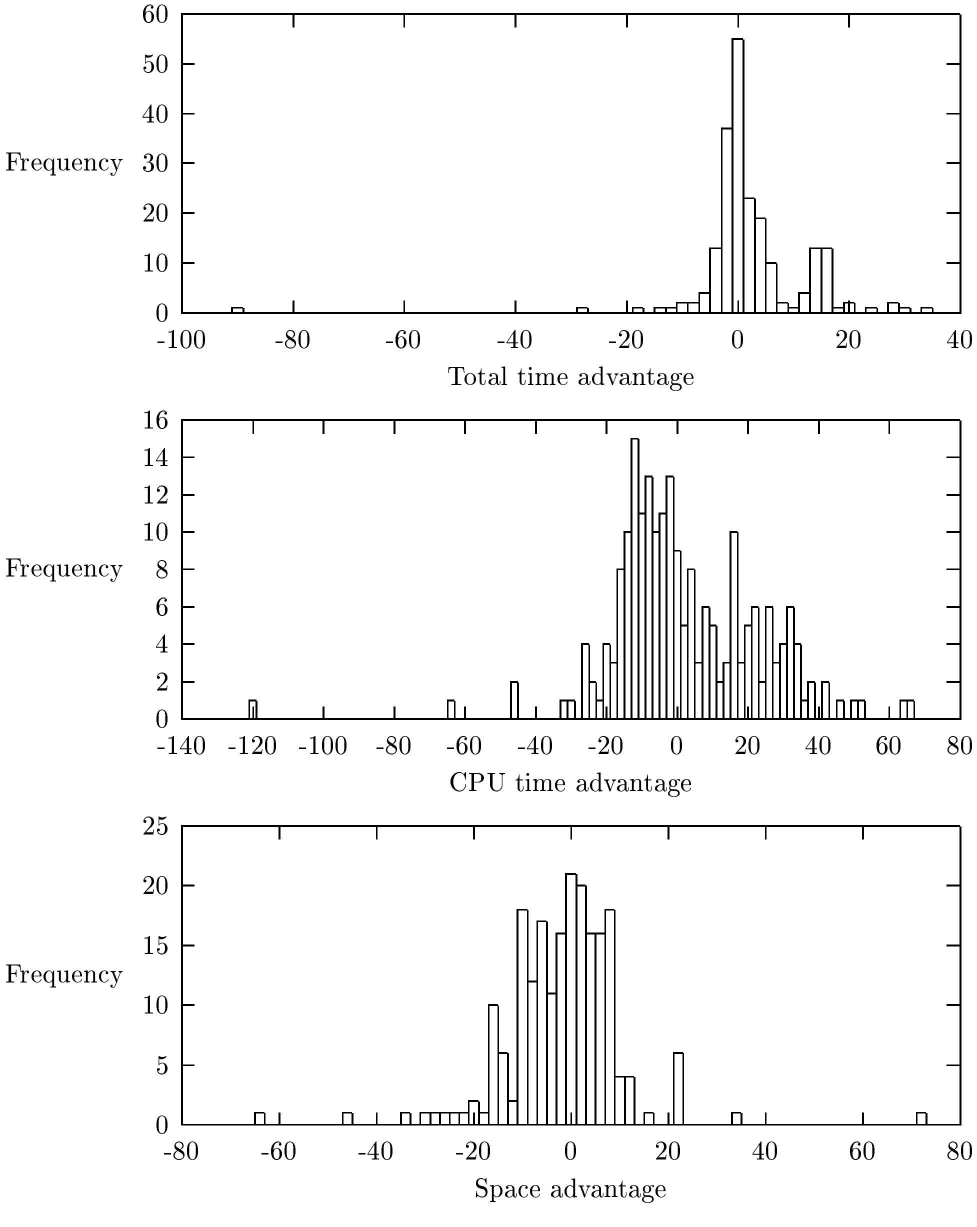

Figure B.14: 2GEN vs ROF: overall results. Bars are two points wide; 2GEN advantage is shown along the negative axis

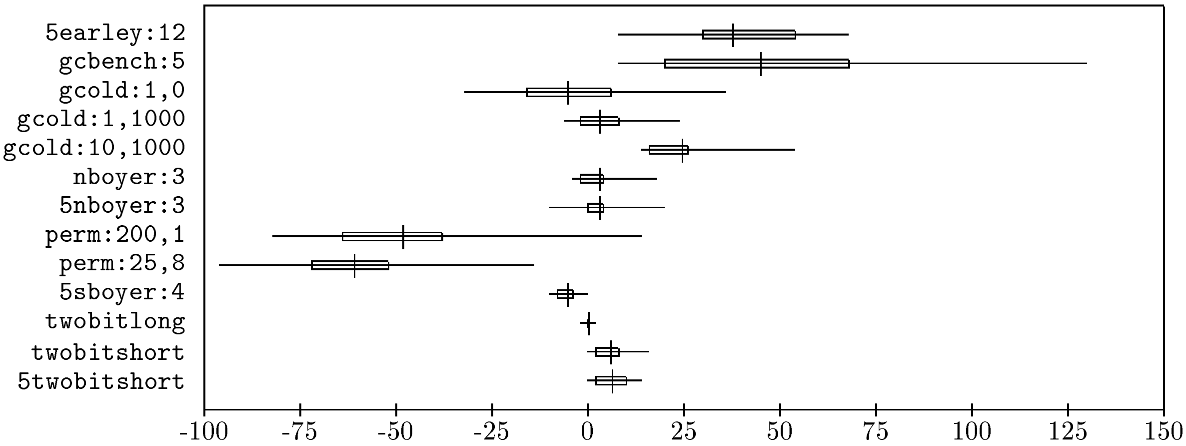

Figure B.15: SC vs 2GEN: total time advantage broken down by benchmark. SC advantage is shown along the negative axis.

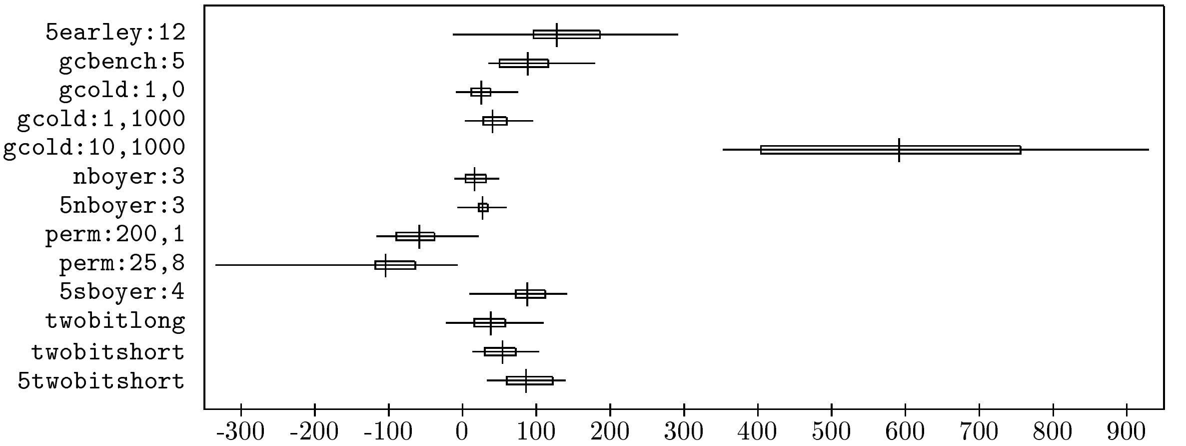

Figure B.16: SC vs 2GEN: GC time advantage broken down by benchmark. SC advantage is shown along the negative axis.

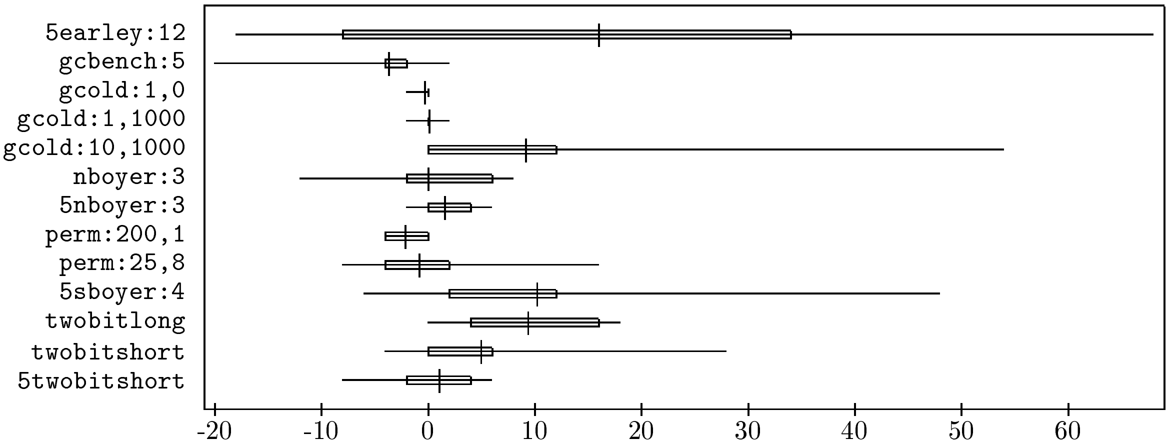

Figure B.17: SC vs 2GEN: space advantage broken down by benchmark. SC advantage is shown along the negative axis.

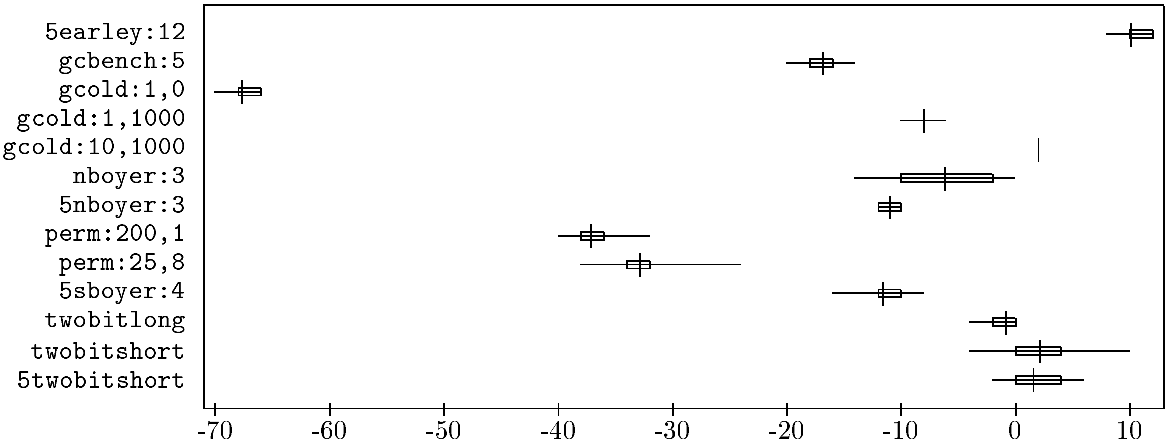

Figure B.18: SC vs 2GEN: mutator time advantage broken down by benchmark. SC advantage is shown along the negative axis.

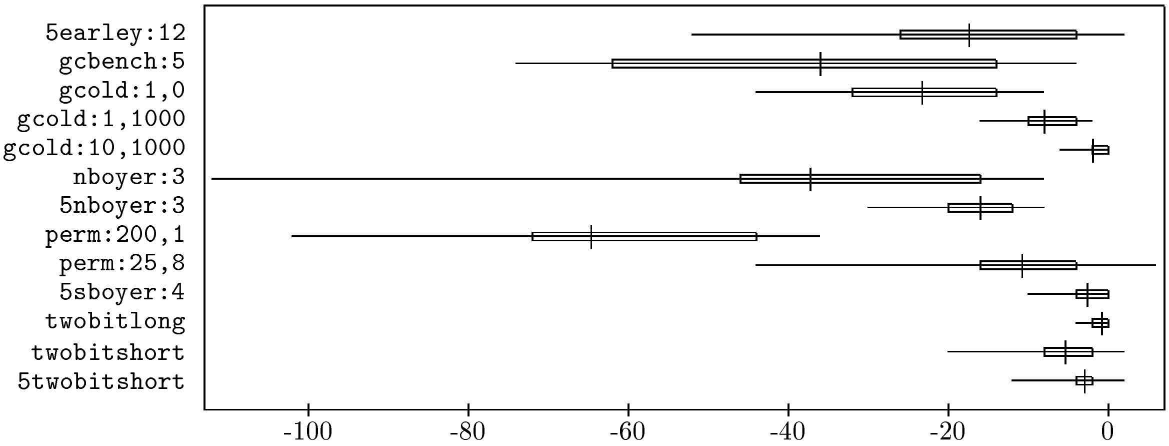

Figure B.19: 2GEN vs 1DOF: total time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

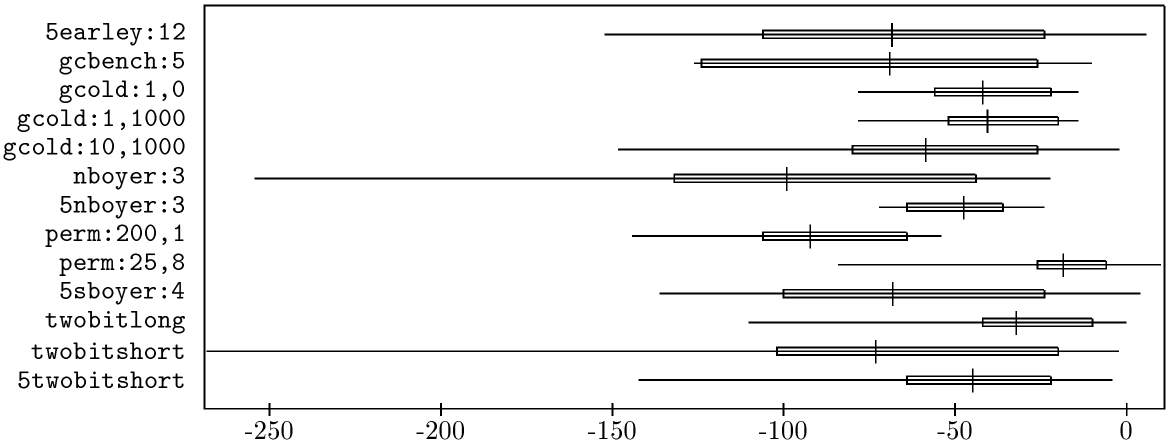

Figure B.20: 2GEN vs 1DOF: GC time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

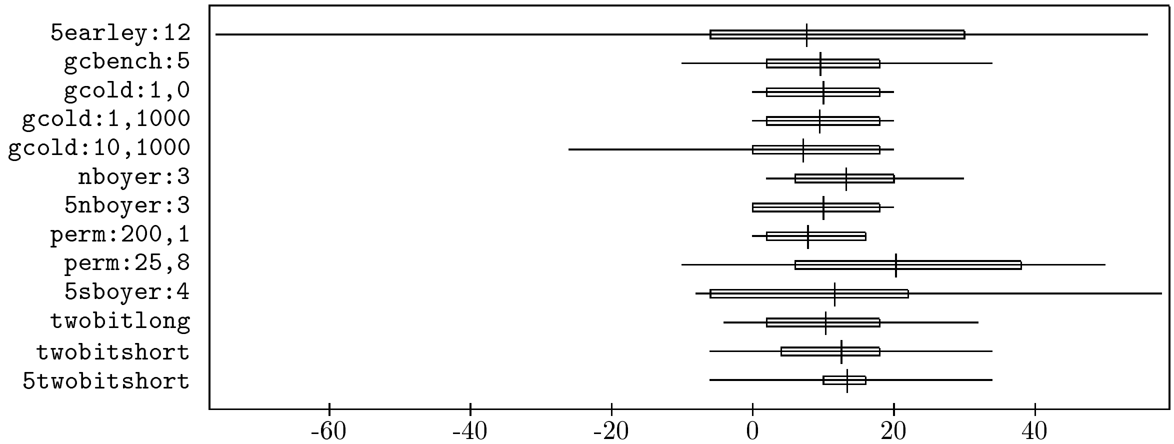

Figure B.21: 2GEN vs 1DOF: space advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

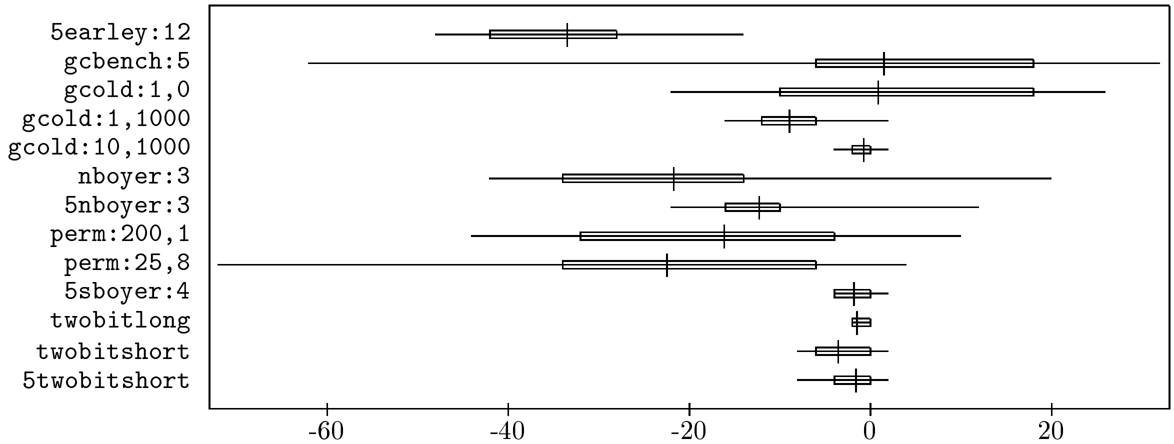

Figure B.22: 2GEN vs 2DOF: total time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

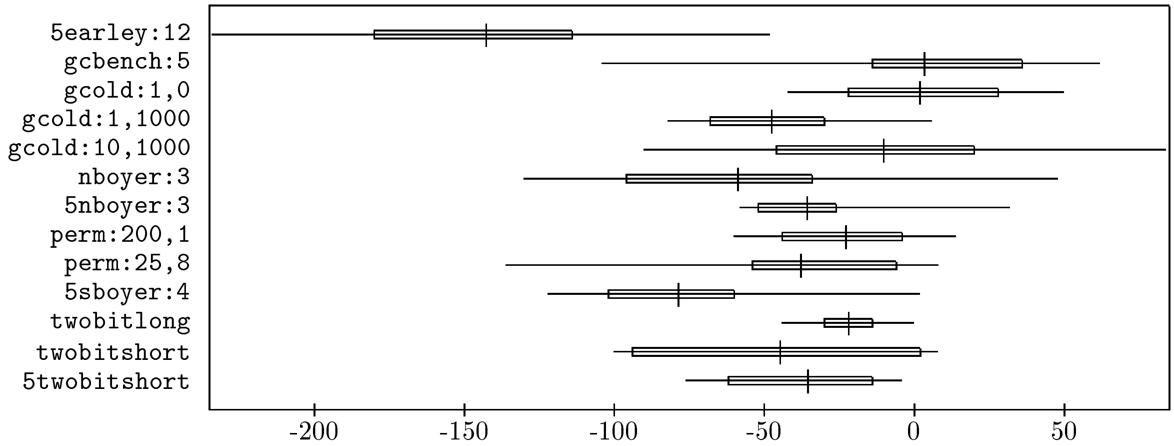

Figure B.23: 2GEN vs 2DOF: GC time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

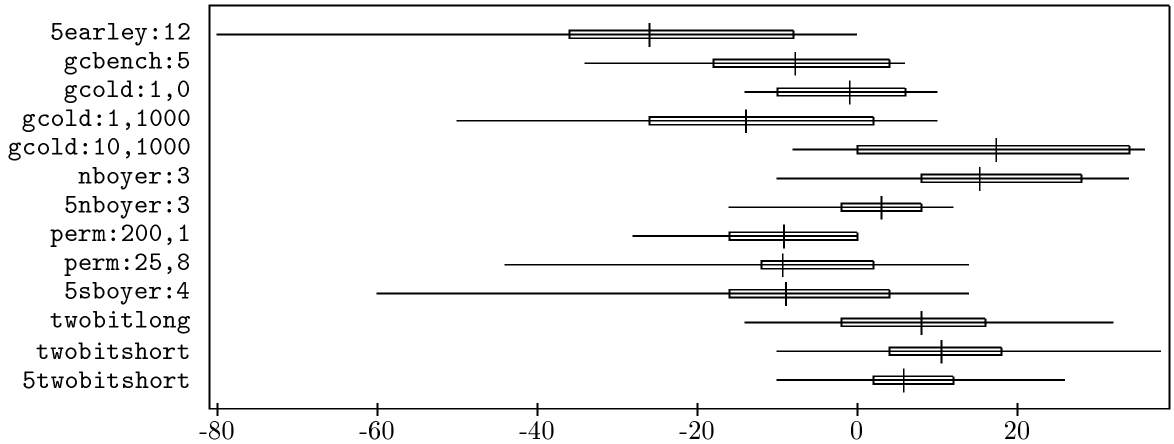

Figure B.24: 2GEN vs 2DOF: space advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

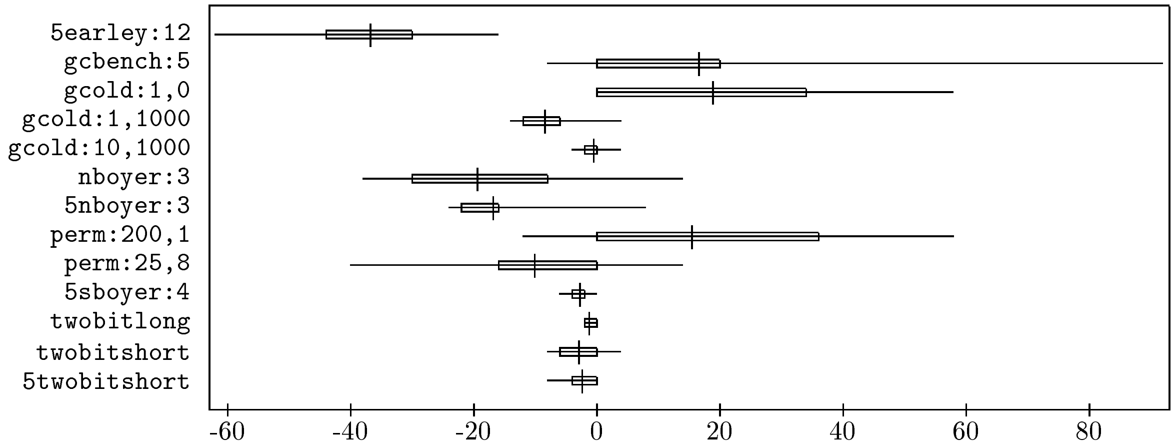

Figure B.25: 2GEN vs 3DOF: total time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

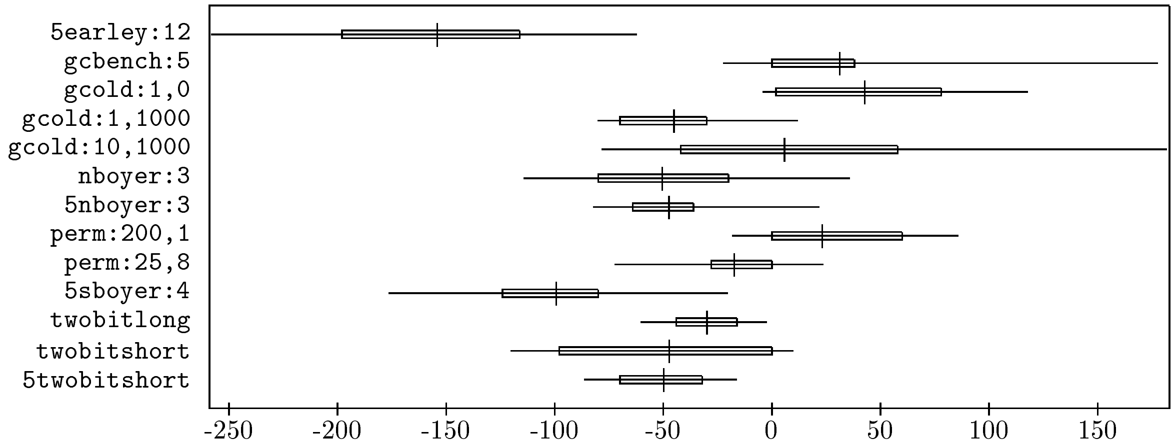

Figure B.26: 2GEN vs 3DOF: GC time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

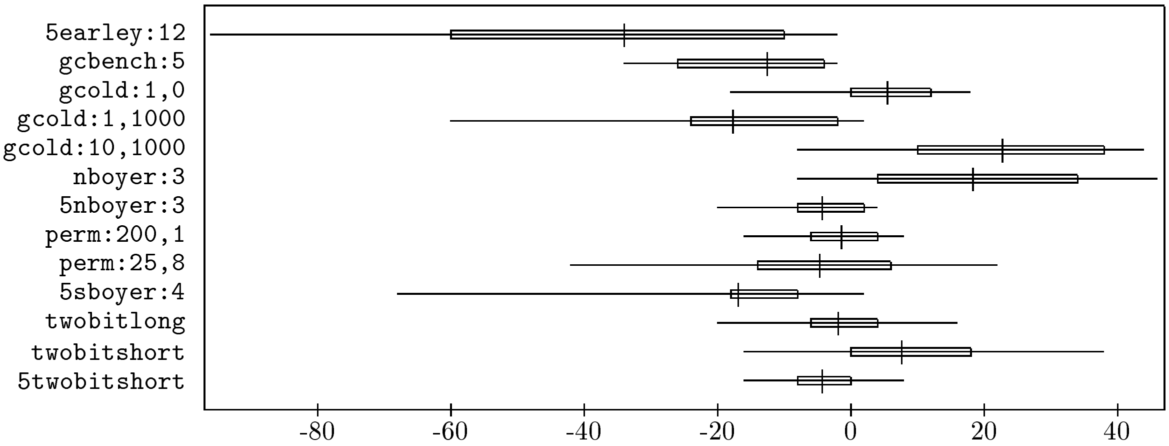

Figure B.27: 2GEN vs 3DOF: space advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

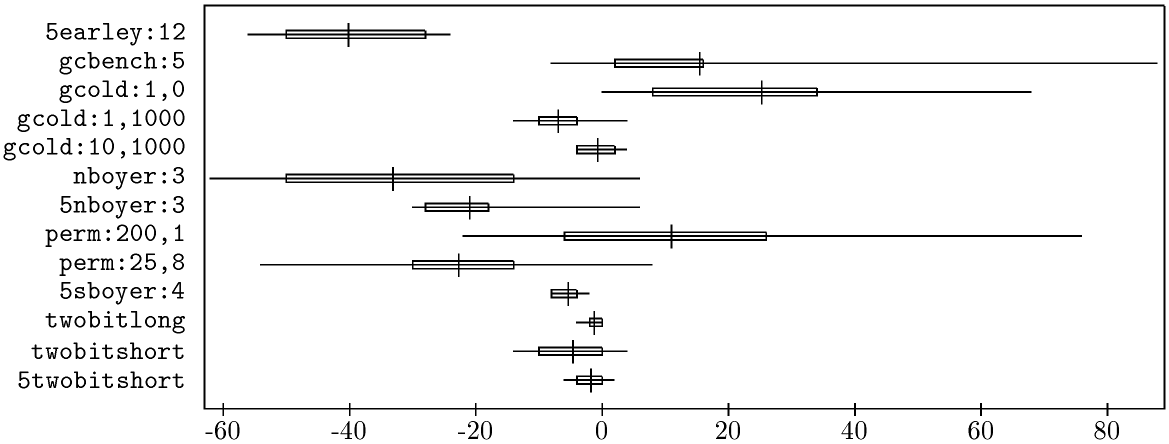

Figure B.28: 2GEN vs 4DOF: total time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

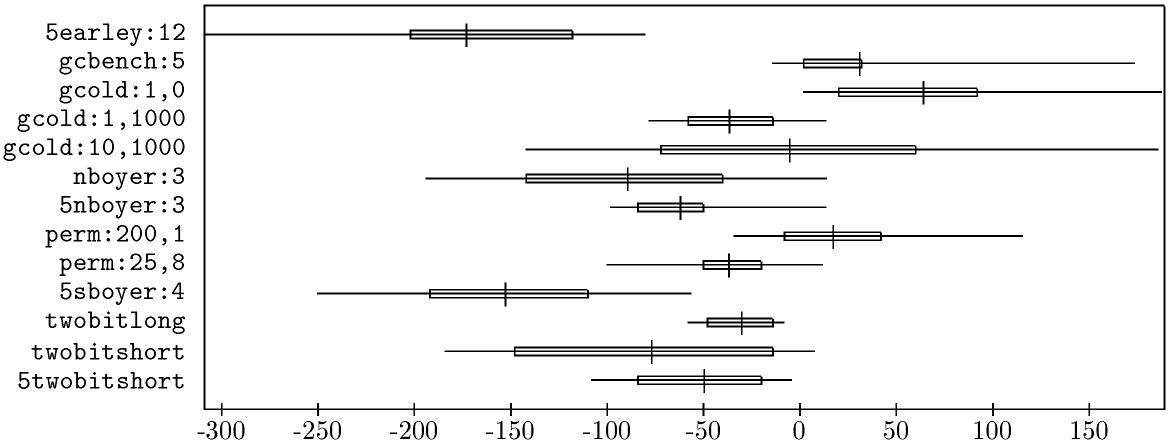

Figure B.29: 2GEN vs 4DOF: GC time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

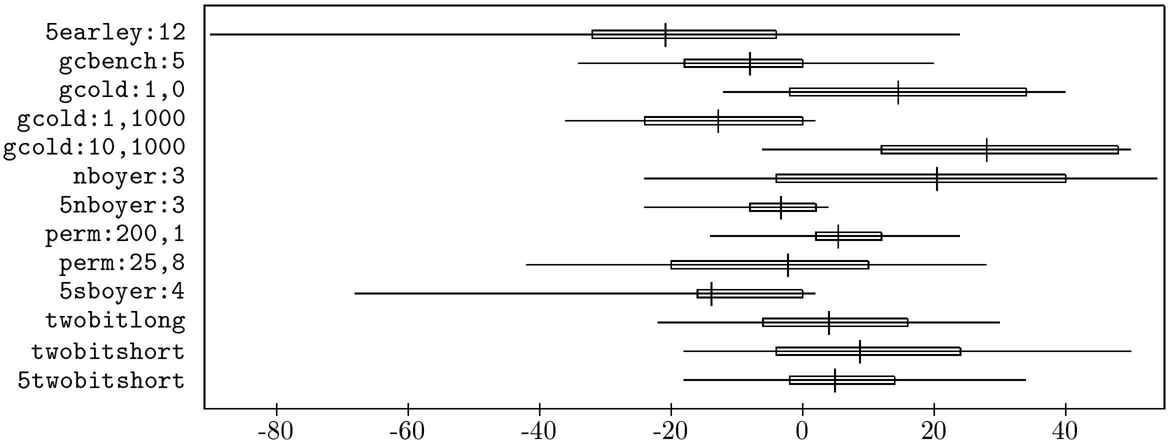

Figure B.30: 2GEN vs 4DOF: space advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

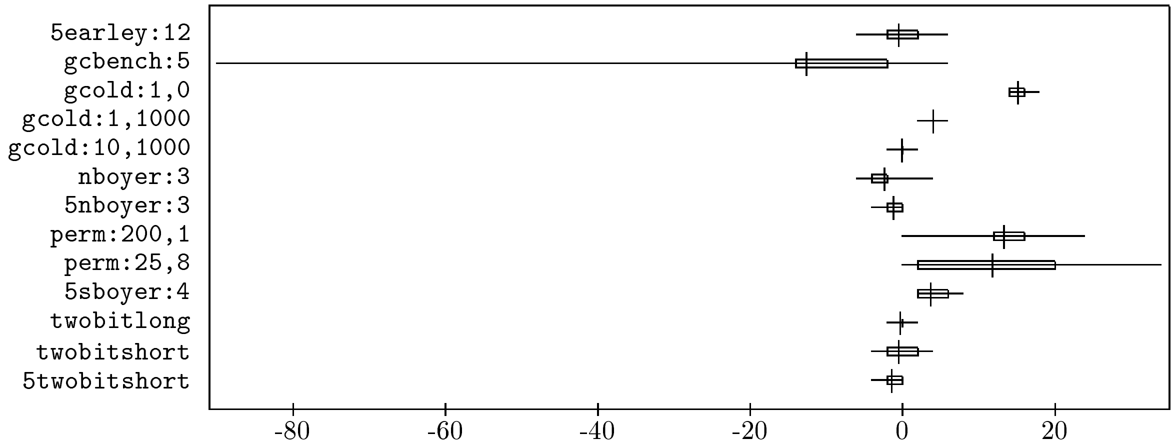

Figure B.31: 2GEN vs ROF: total time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

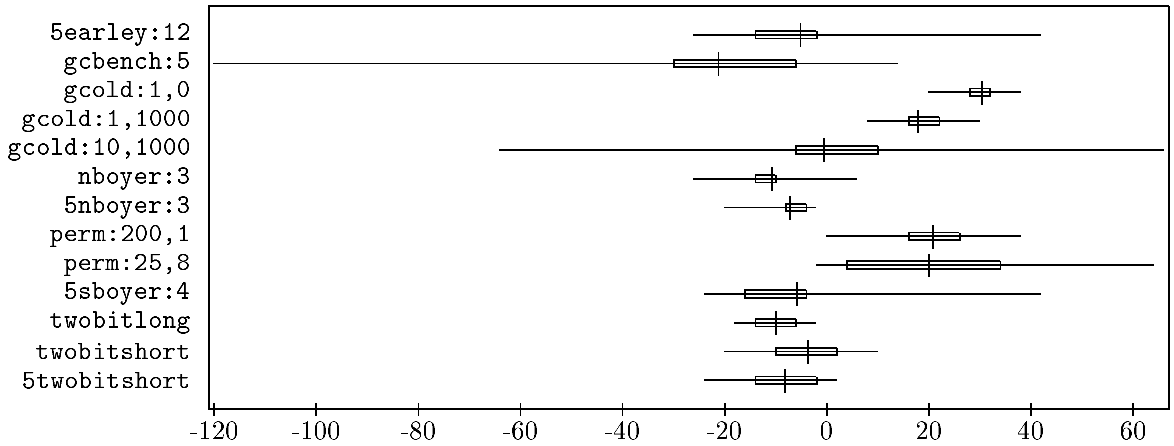

Figure B.32: 2GEN vs ROF: GC time advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

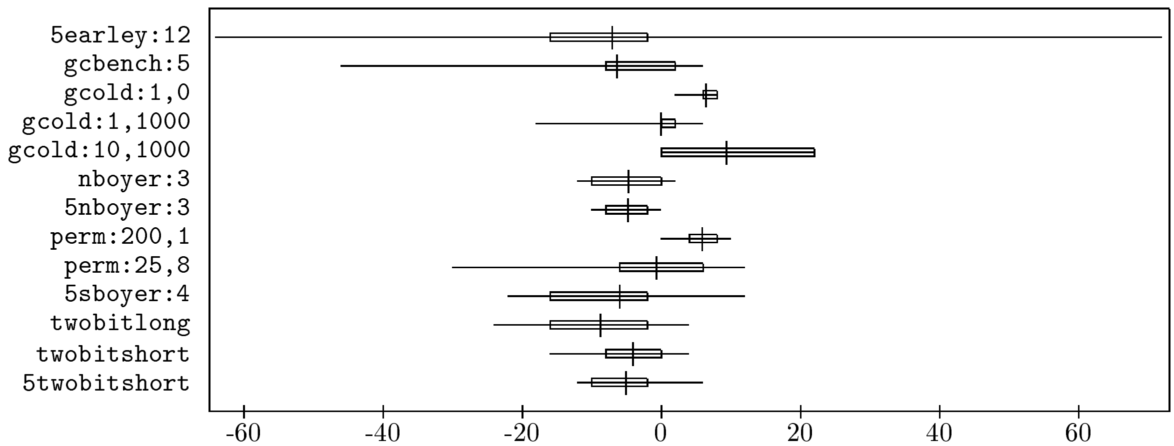

Figure B.33: 2GEN vs ROF: space advantage broken down by benchmark. 2GEN advantage is shown along the negative axis.

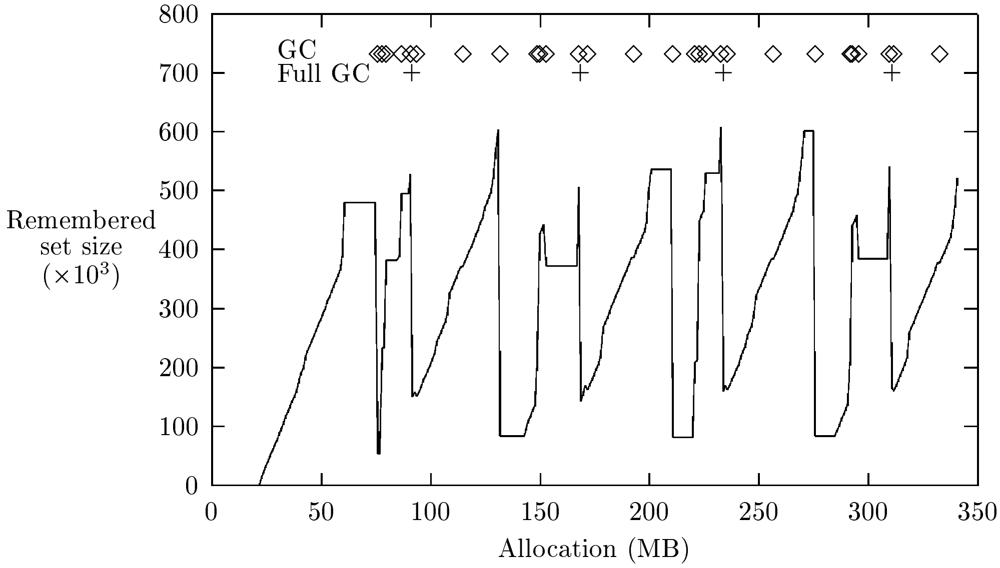

Figure B.34: 4DOF remembered set profile: `earley`

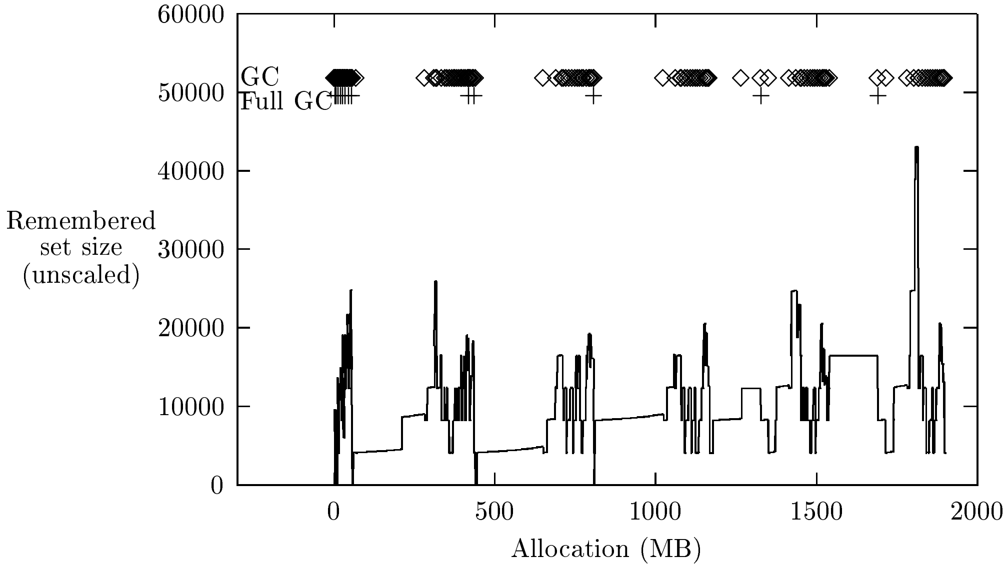

Figure B.35: 4DOF remembered set profile: `gcbench`

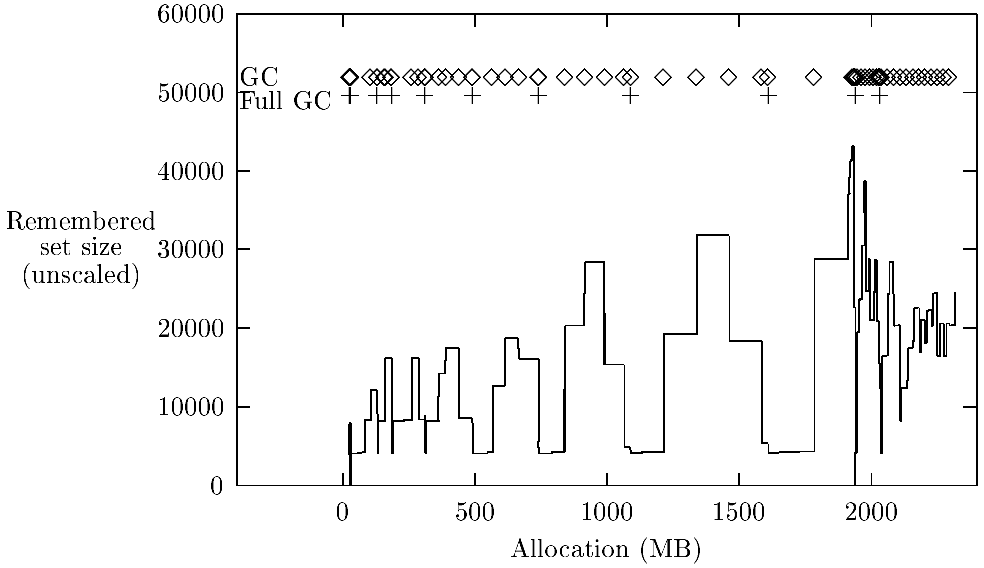

Figure B.36: 4DOF remembered set profile: `gcold 1,0`

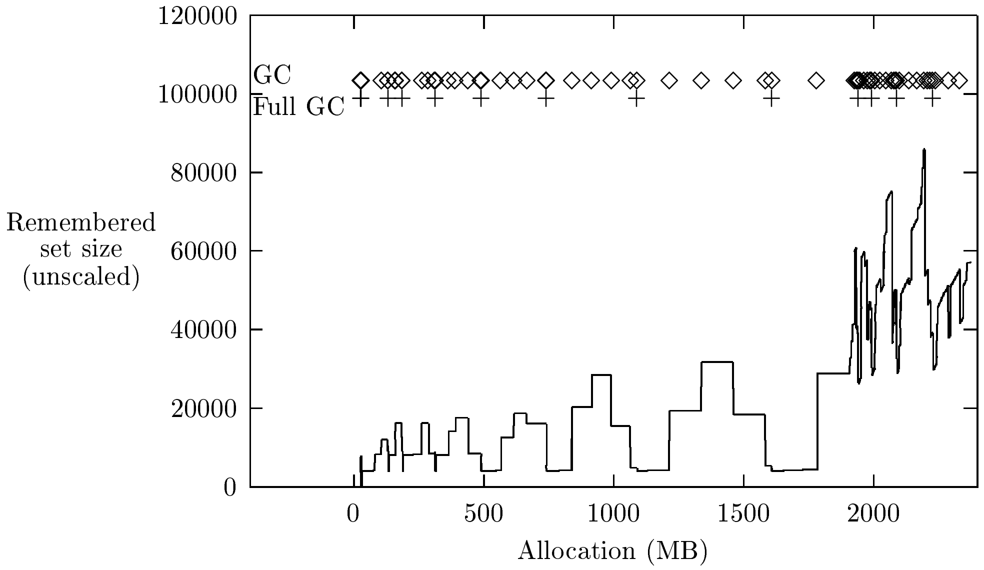

Figure B.37: 4DOF remembered set profile: `gcold 1,1000`

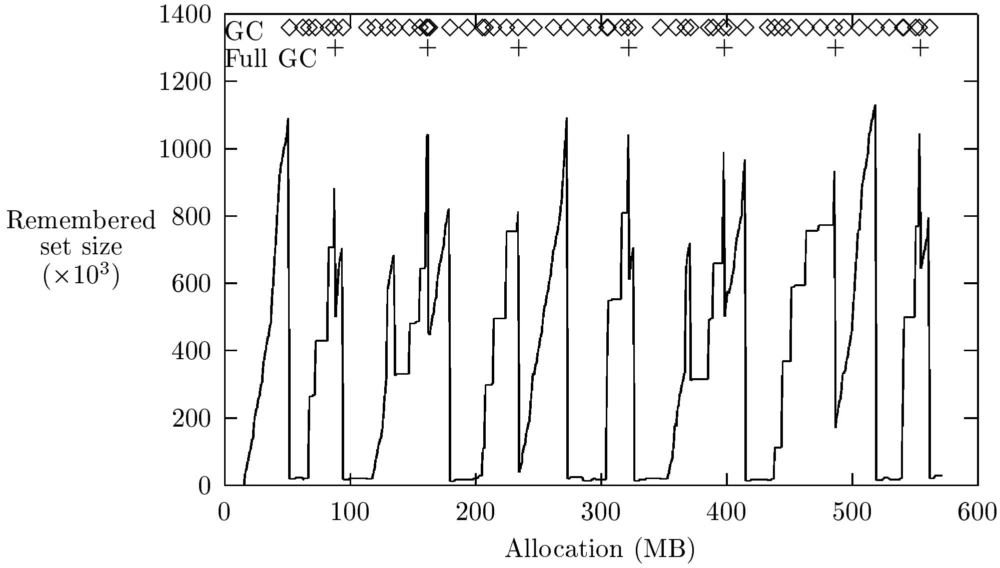

Figure B.38: 4DOF remembered set profile: `5nboyer`

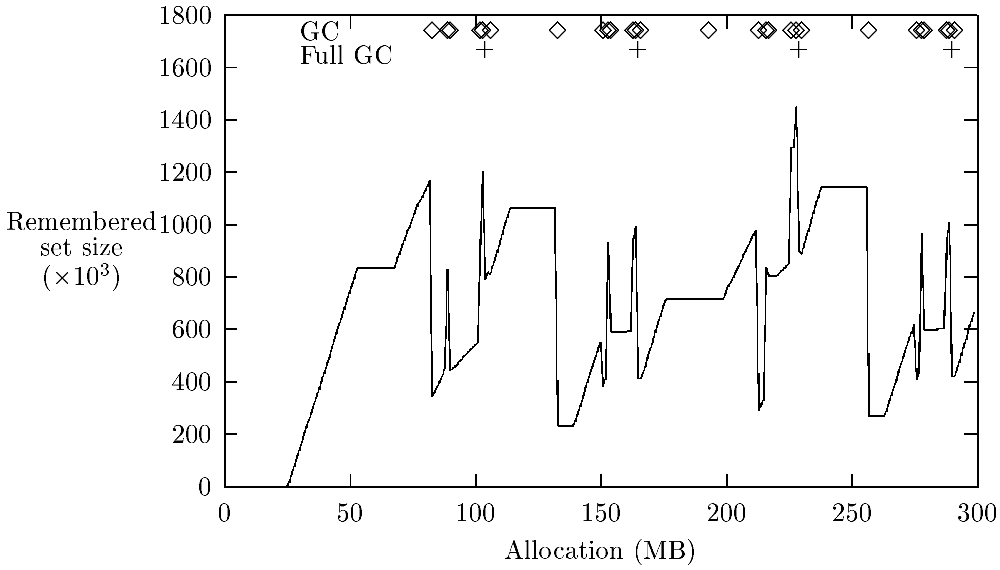

Figure B.39: 4DOF remembered set profile: `5sboyer`

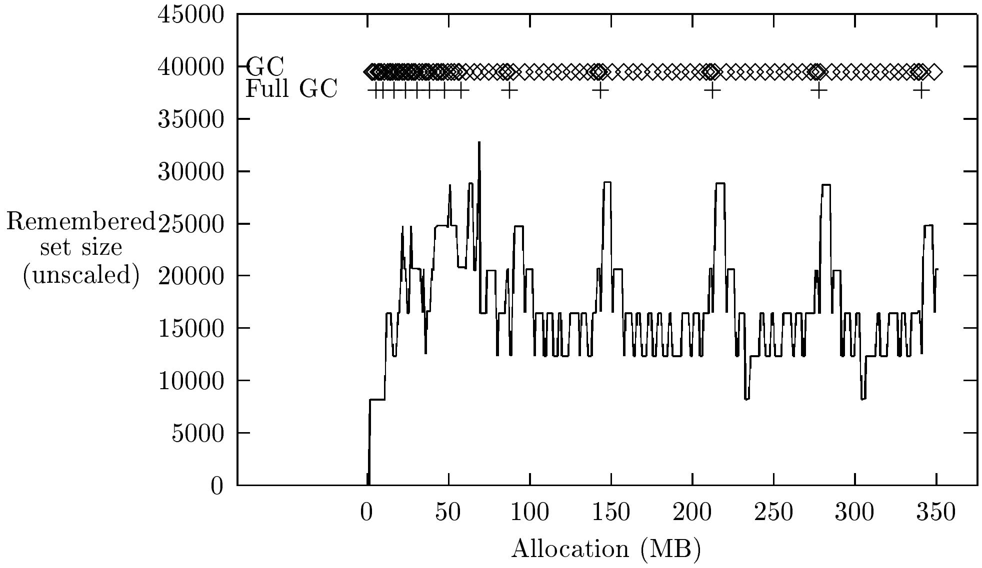

Figure B.40: 4DOF remembered set profile: `perm 200,1`

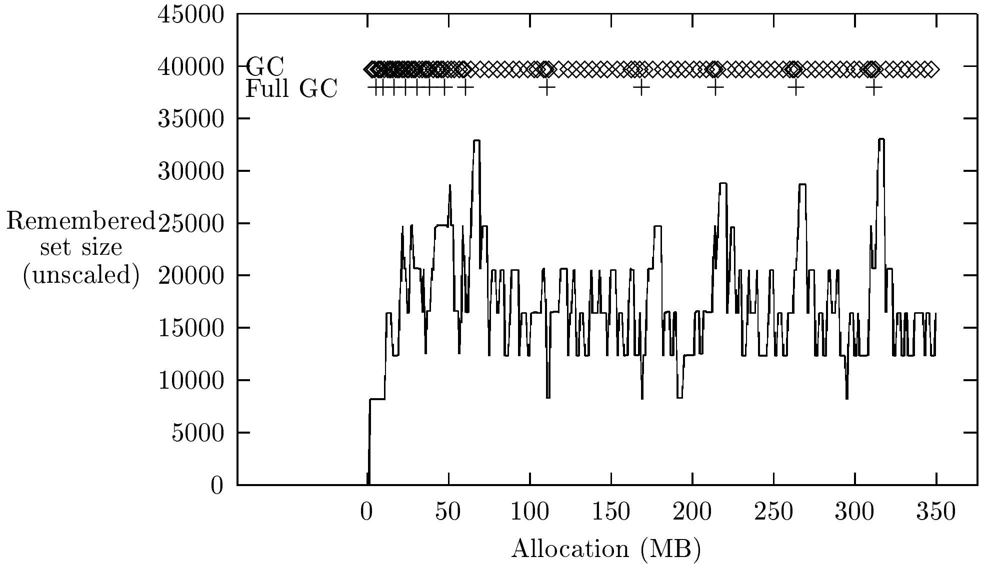

Figure B.41: 4DOF remembered set profile: `perm 25,8`

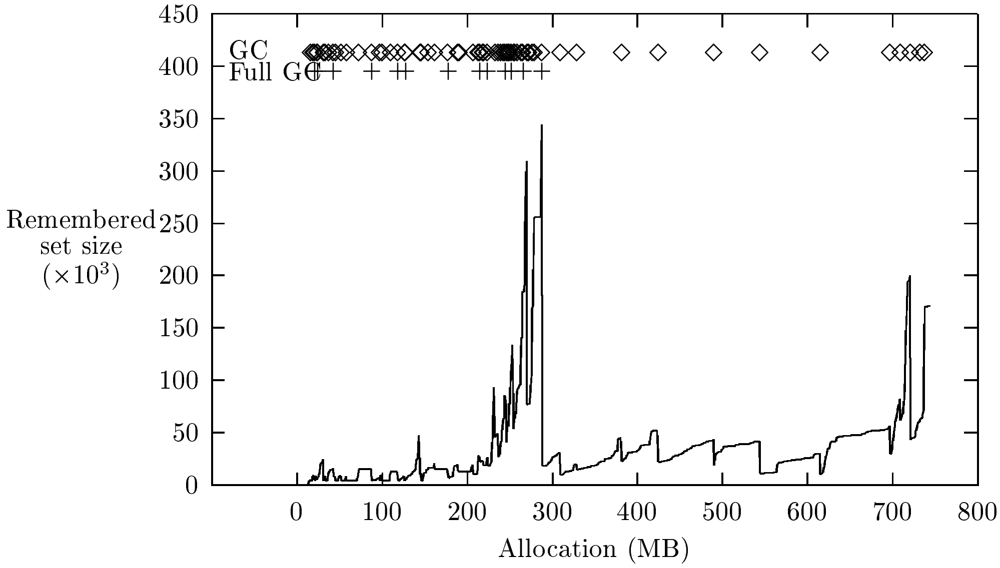

Figure B.42: 4DOF remembered set profile: `twobitlong`

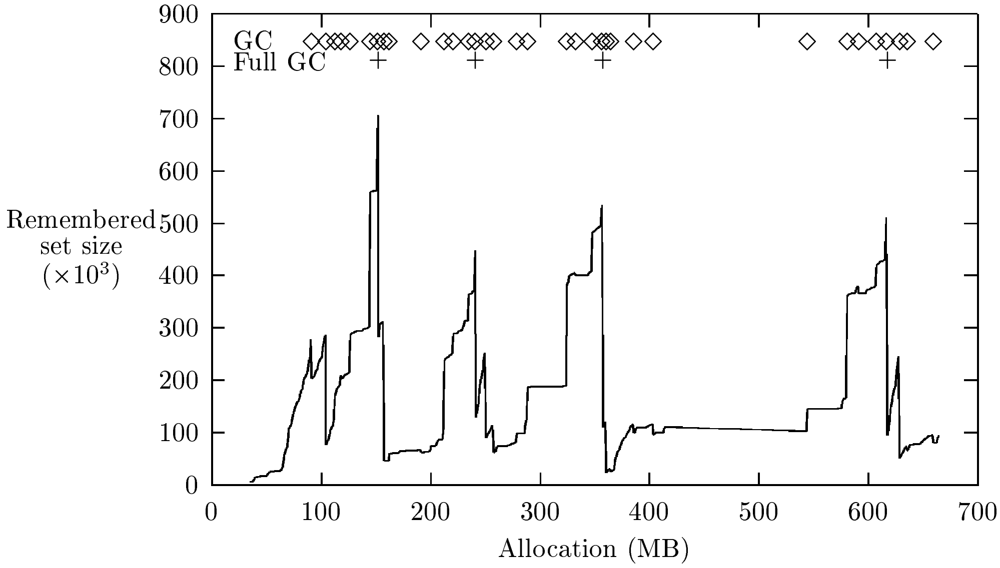

Figure B.43: 4DOF remembered set profile: `5twobitshort`

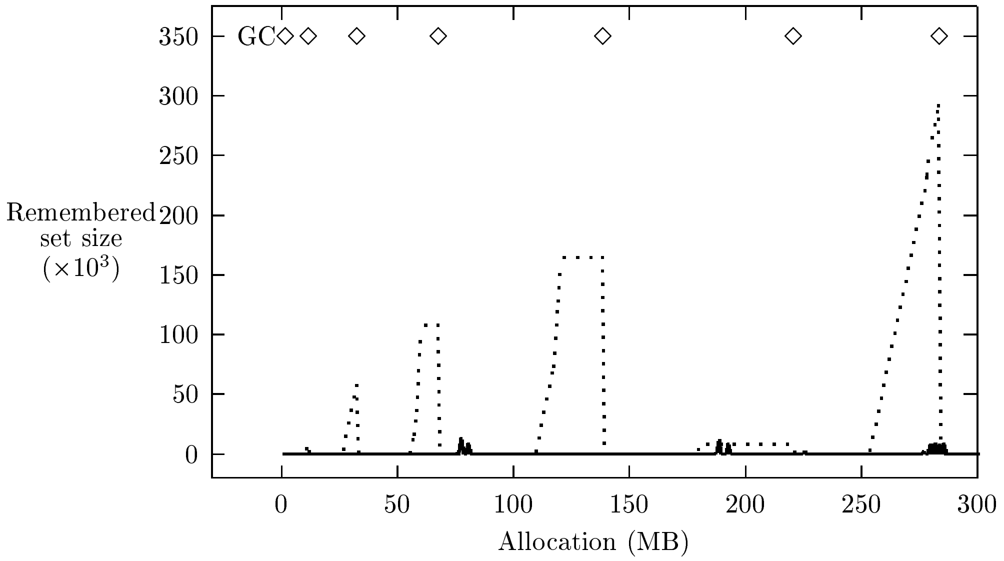

Figure B.44: ROF remembered set profile: `earley`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

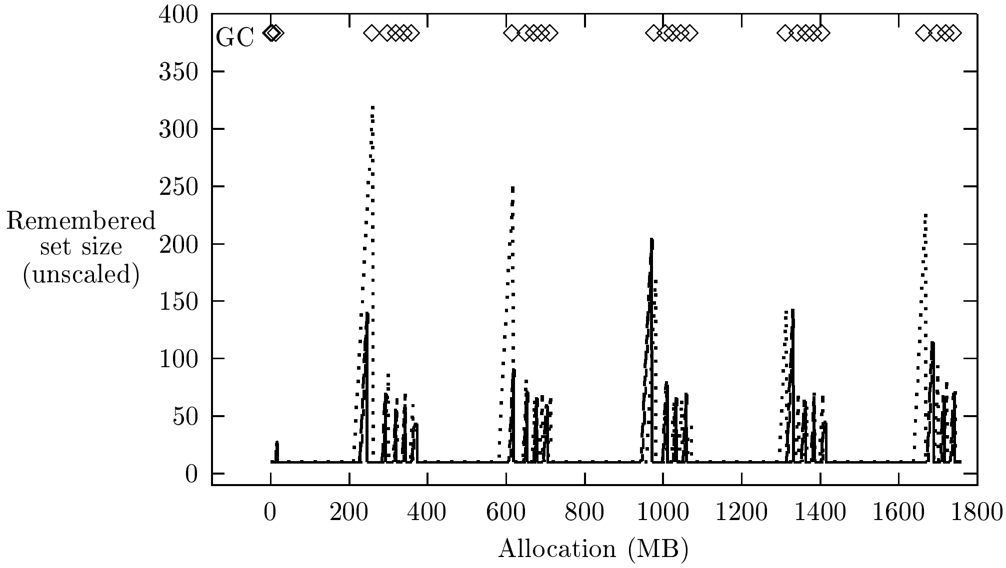

Figure B.45: ROF remembered set profile: `gcbench`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

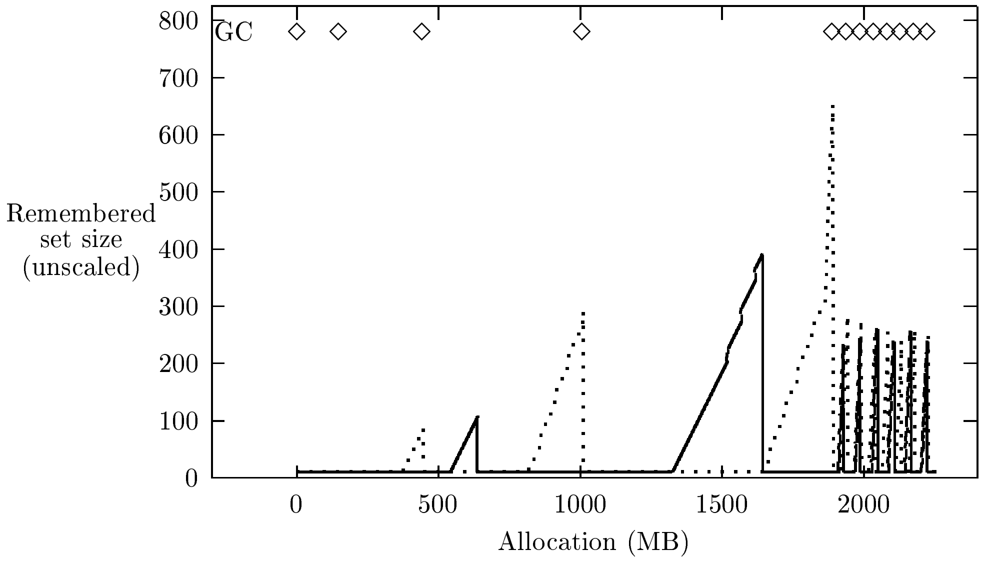

Figure B.46: ROF remembered set profile: `gcold 1,0`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

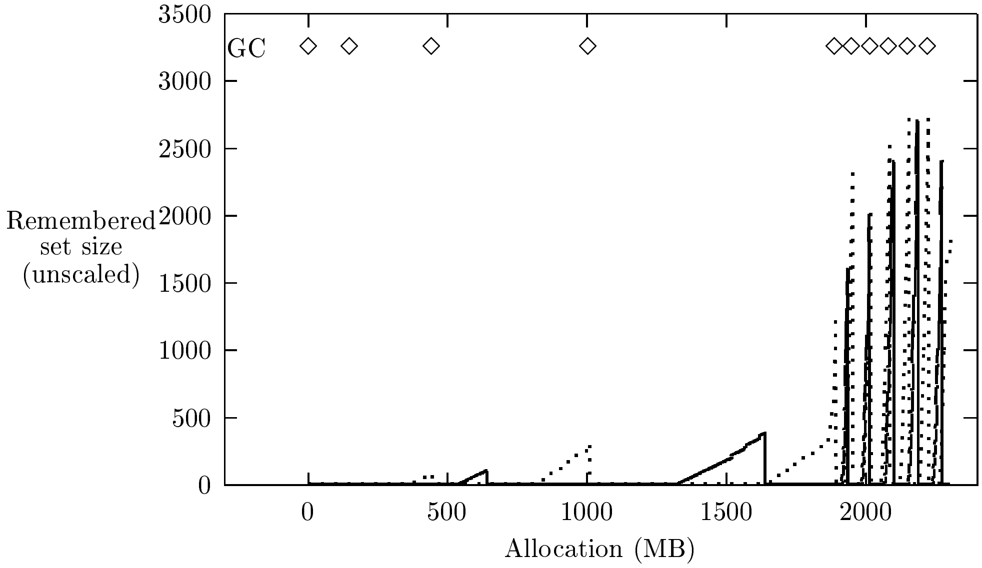

Figure B.47: ROF remembered set profile: `gcold 1,1000`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

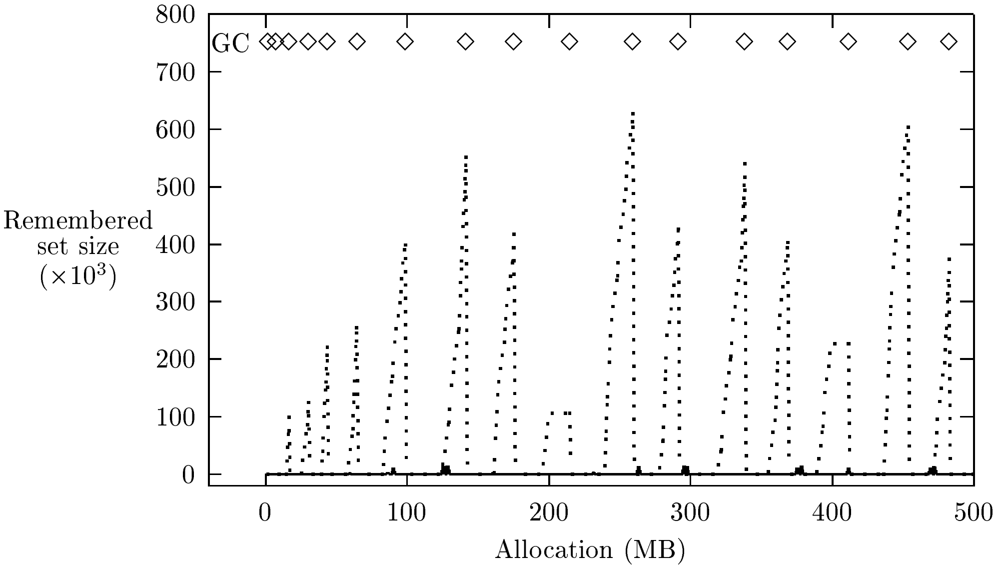

Figure B.48: ROF remembered set profile: `5nboyer`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

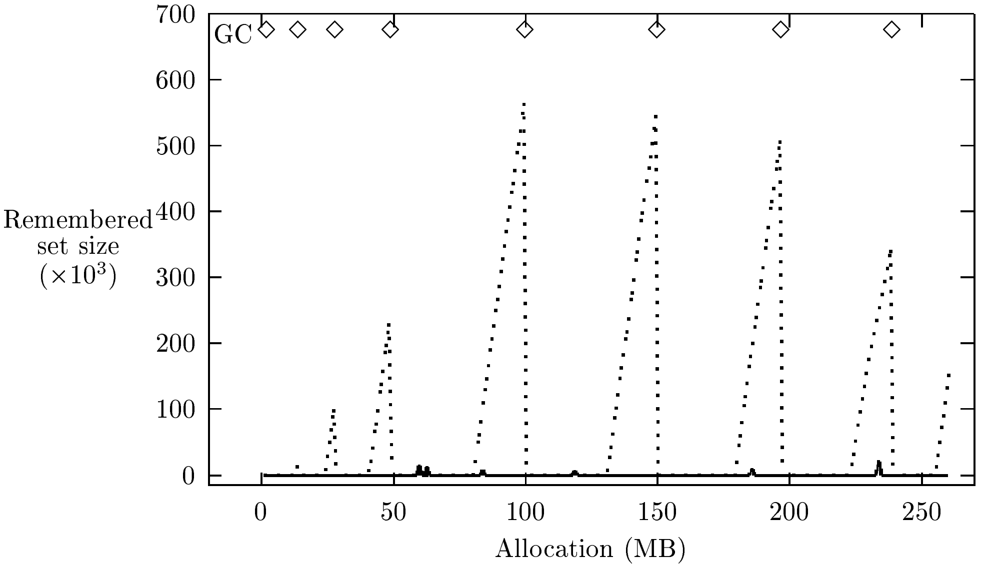

Figure B.49: ROF remembered set profile: `5sboyer`. The dotted line shows the size without tuning parameters and the solid line shows size with tuning parameters.

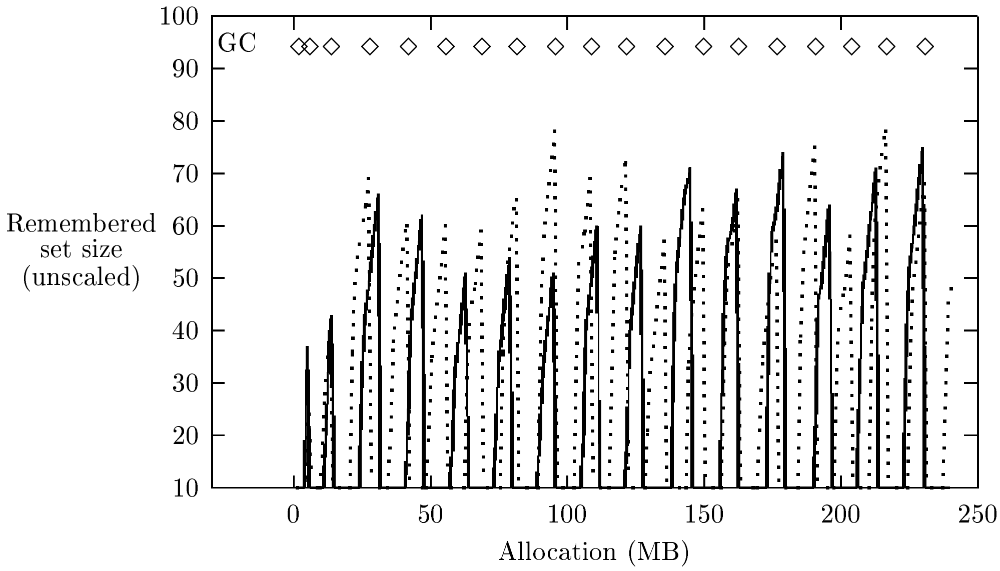

Figure B.50: ROF remembered set profile: `perm 200,1`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

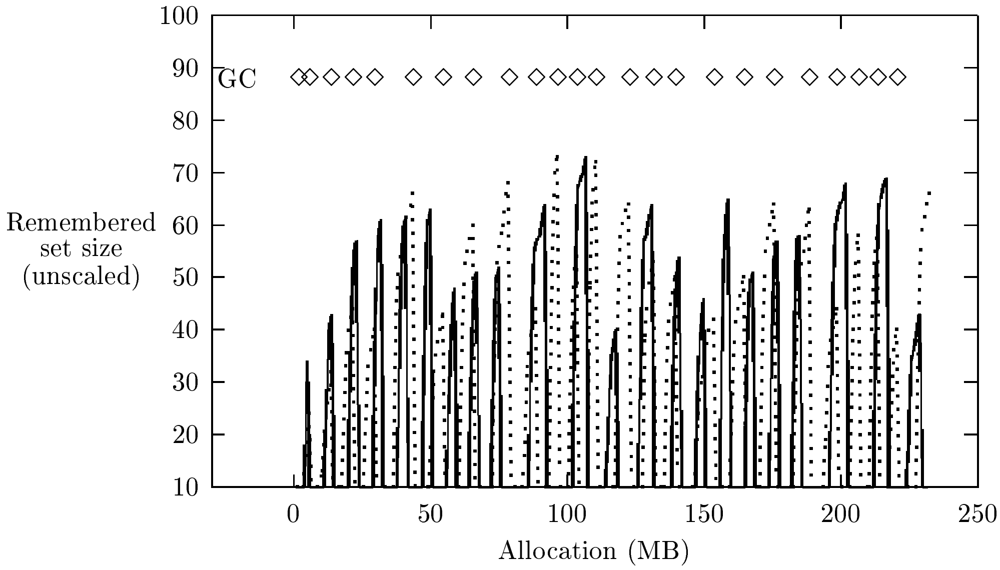

Figure B.51: ROF remembered set profile: `perm 25,8`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

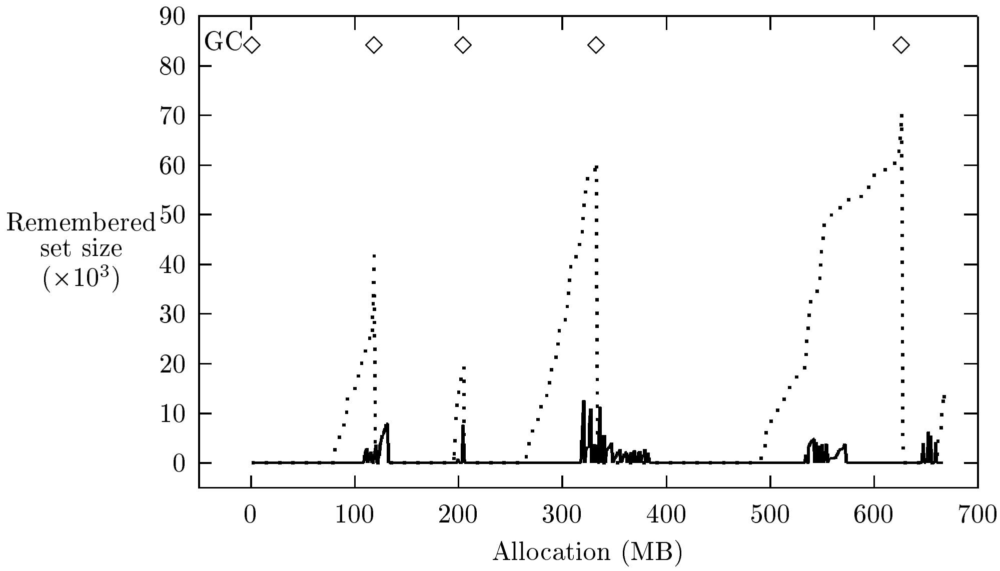

Figure B.52: ROF remembered set profile: `twobitlong`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

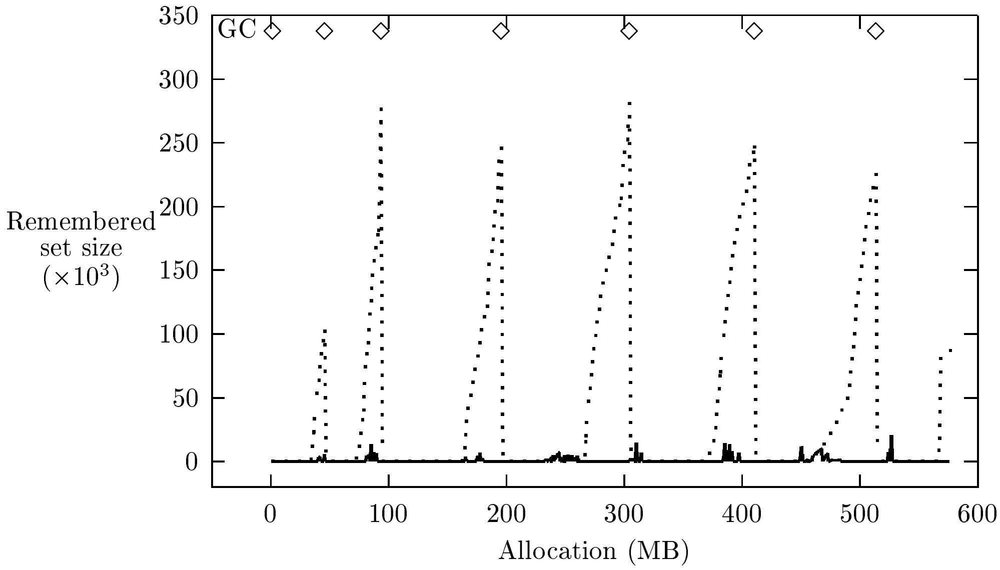

Figure B.53: ROF remembered set profile: `5twobitshort`. The dotted line shows the size without tuning parameters and the solid line shows the size with tuning parameters.

Appendix C

Storage profiles

This appendix contains storage profiles for some of the benchmark programs. Time is measured by the number of bytes allocated and is shown along the x-axis; the amount of live data is shown as a function of time along the y-axis.

The amount of live data is sampled every megabyte of allocation. The samples are reasonably accurate but measure heap allocation performed by the program and instrumented libraries only; notably, the samples do not measure space allocated to closures, nor space allocated by the (uninstrumented) I/O system. The benchmarks I use do not allocate many closures so that inaccuracy is not important. The amount of allocation performed by I/O is important only for `twobit`; in practice, I/O in that program stretches the profile along the x-axis but has little impact on the profile's height.

The profiles do not include the contents of the static area: the libraries, the interpreter, and the program code.

The profiles show strata of live data from one-megabyte cohorts, with the data from the youngest cohort at the top.

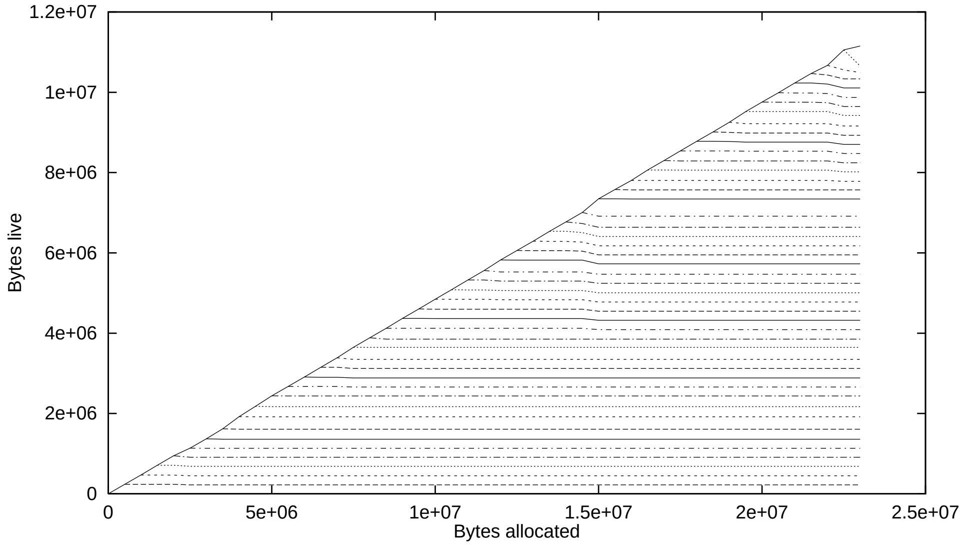

Figure C.1: Storage profile for `1earley12`

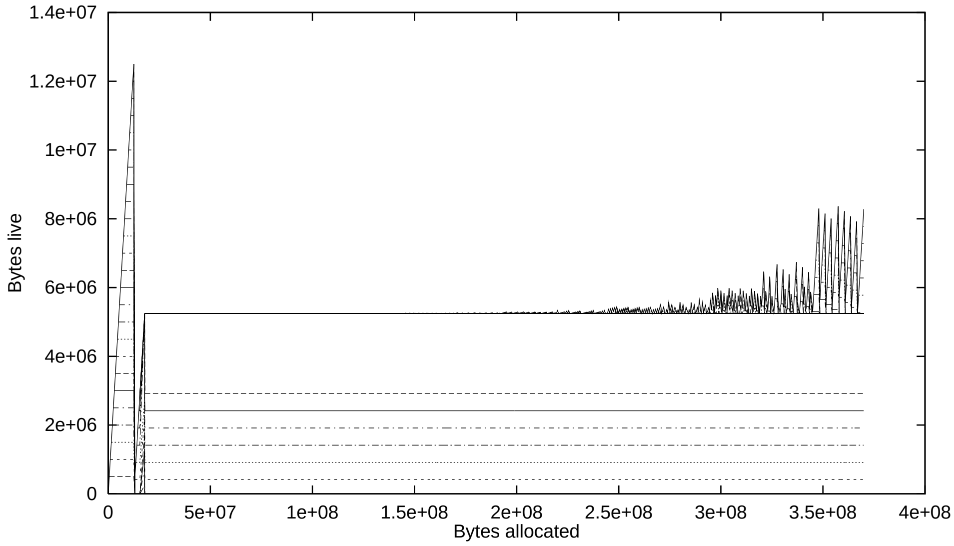

Figure C.2: Storage profile for `1gcbench18`, not showing 8 megabytes of permanent floating-point data

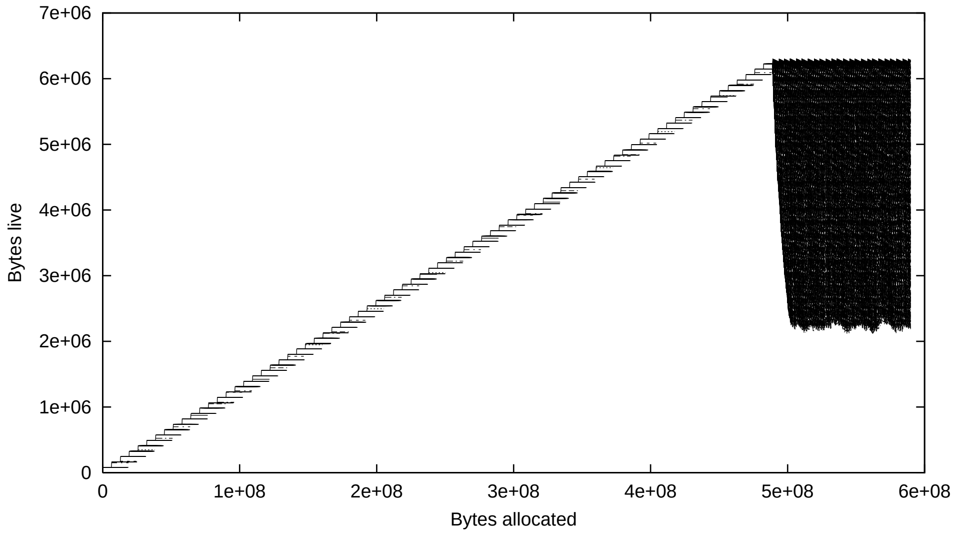

Figure C.3: Storage profile for `gcold:1,1000` (courtesy William Clinger)

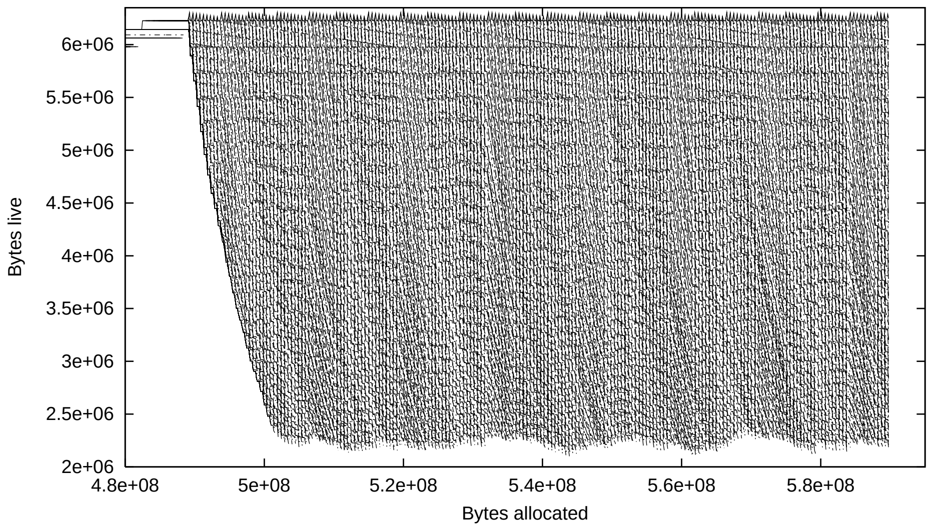

Figure C.4: Storage profile for `gcold:1,1000` (detail; courtesy William Clinger)

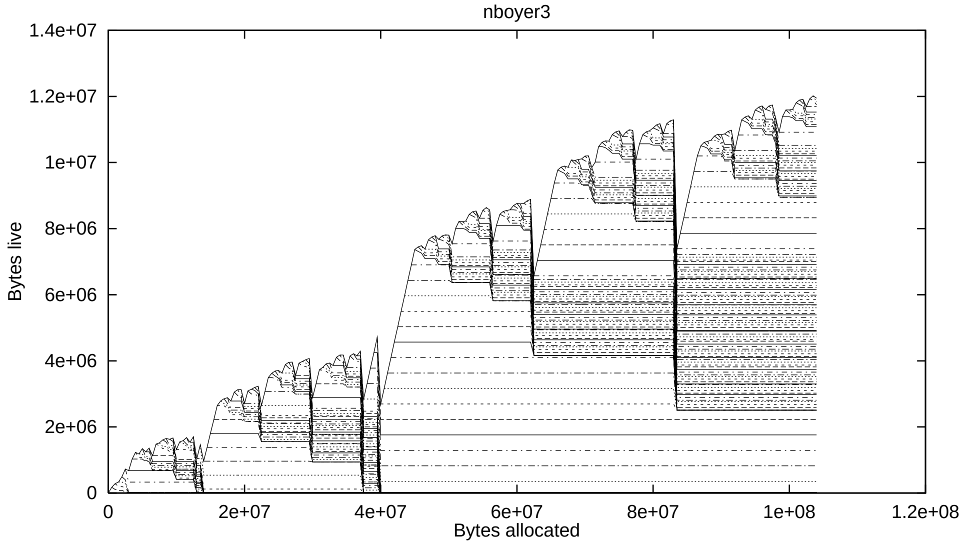

Figure C.5: Storage profile for `1nboyer3`

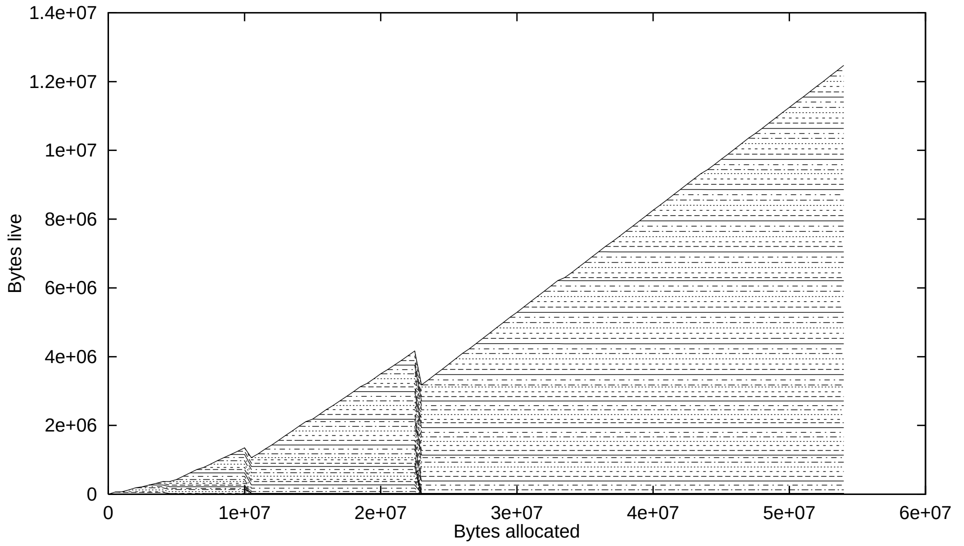

Figure C.6: Storage profile for `1sboyer4`

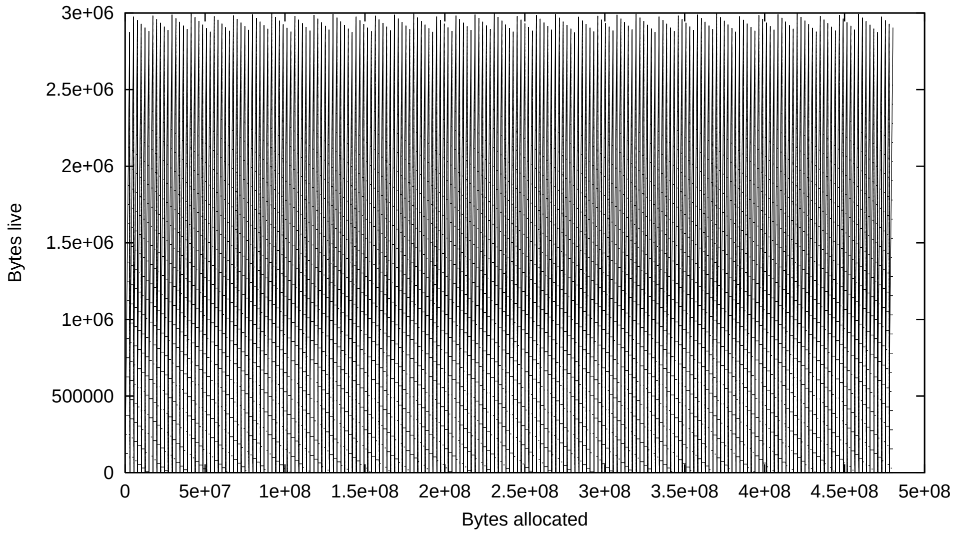

Figure C.7: Storage profile for `perm:25,8` (courtesy William Clinger)

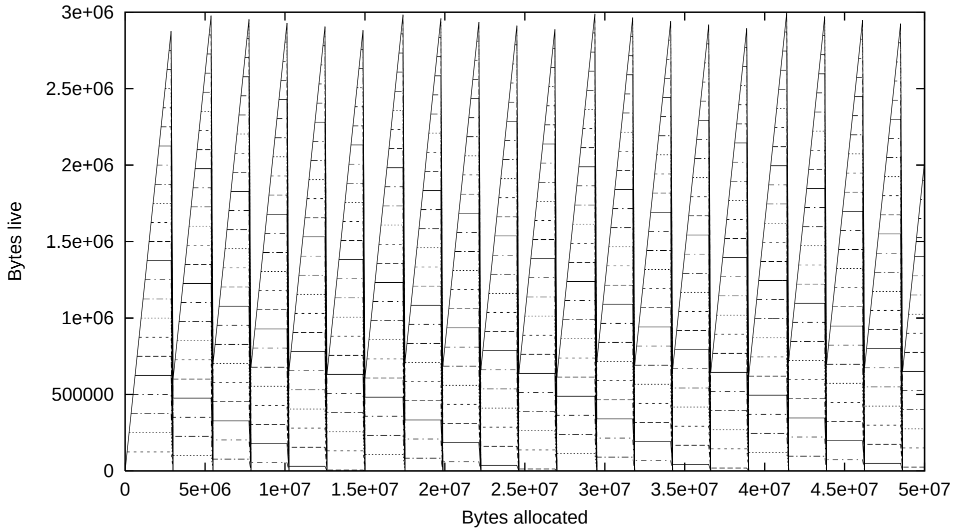

Figure C.8: Storage profile for `perm:25,8` (detail; courtesy William Clinger)

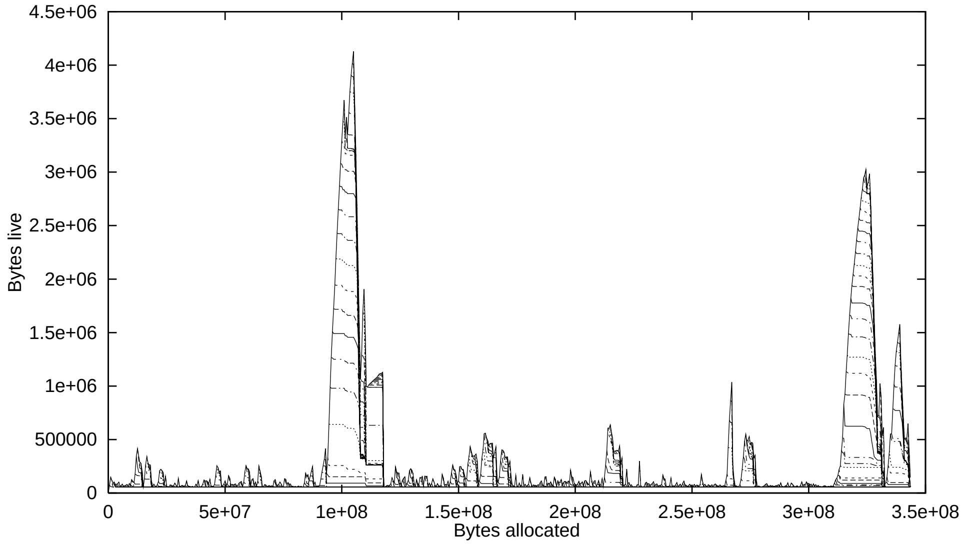

Figure C.9: Storage profile for `twobitlong`

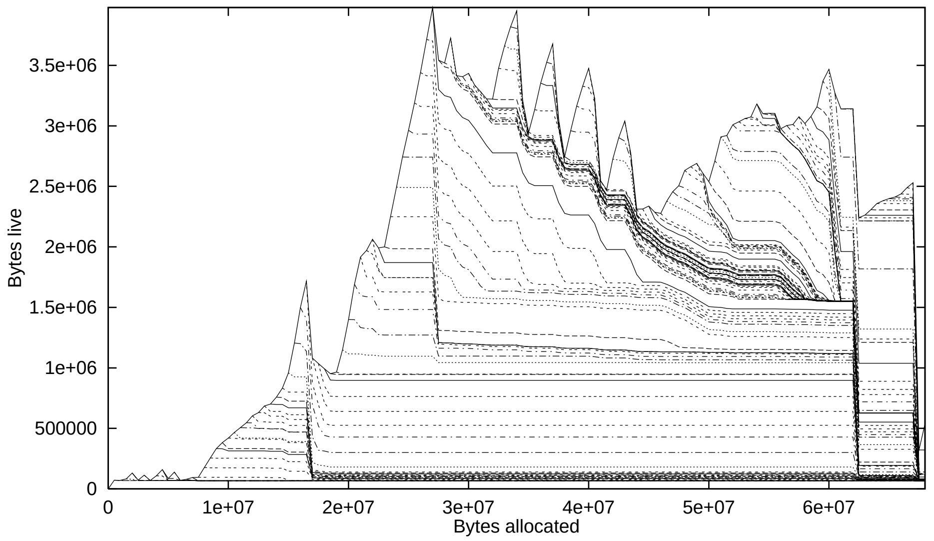

Figure C.10: Storage profile for `1twobitshort` (courtesy William Clinger)