Scalable Garbage Collection via Remembered Set Summarization and Refinement

A dissertation presented

by

Felix S Klock II

to the Faculty of the Graduate School

of the College of Computer and Information Science

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Northeastern University

Boston, Massachusetts

January, 2011

NORTHEASTERN UNIVERSITY
GRADUATE SCHOOL OF COMPUTER SCIENCE
Ph.D. THESIS COMPLETION APPROVAL FORM

THESIS TITLE:

Scalable Garbage Collection via Remembered Set Summarization and Refinement

AUTHOR:

Ph.D. Thesis Approved to complete all degree requirements for the Ph.D. Degree in Computer Science.

William D Clinger — *Thesis Advisor* — 20 Dec 2010 — *Date*

Olin Shivers — *Thesis Reader* — 2010/12/20 — *Date*

Guy L Steele Jr — *Thesis Reader* — 20 Dec 2010 — *Date*

Gene Cooperman — *Thesis Reader* — Dec. 20, 2010 — *Date*

GRADUATE SCHOOL APPROVAL:

Director, Graduate School — 1/21/2011 — *Date*

COPY RECEIVED IN GRADUATE SCHOOL OFFICE:

Recipient's Signature — 1/24/2011 — *Date*

Distribution: *One Copy to Thesis Advisor*
One Copy to Each Member of Thesis Committee
One Copy to Director of Graduate School
One Copy to Graduate School Office (with signature approval sheet, including all signatures)

Abstract

Regional garbage collection offers a useful compromise between real-time and generational collection. Regional collectors resemble generational collectors, but are scalable. A scalable collector guarantees a positive lower bound, independent of mutator and live storage, for the theoretical worst-case minimum mutator utilization (MMU).

Standard generational collectors are not scalable. Some real-time collectors are scalable, while others assume a well-behaved mutator or provide no worst-case guarantees at all.

This dissertation presents regional garbage collection, coupled with a theorem establishing that it is scalable in the sense above, as well as establishing upper bounds for its worst-case space usage and collection pauses.

Regional collectors separate *summarization* and *refinement* from the task of object reclamation. They resolve "popularity" problems via two novel technologies: *summarization wave-off*, and *region fame*.

Regional collectors cannot compete with hard real-time collectors at millisecond resolutions, but offer efficiency comparable to contemporary generational collectors combined with improved latency and MMU at resolutions on the order of hundreds of milliseconds to a few seconds.

A prototype regional collector performs acceptably on a wide range of benchmarks: It is comparable to a tuned generational collector on a set of fifty-eight non-collection-intensive benchmarks, and achieves acceptable throughput without violating its bounds on a set of thirteen collection-intensive benchmarks.

Acknowledgments

To my classmates at the Northeastern University Programming Research Lab, thank you for being a source of new ideas and inspiration, and for keeping me laughing during all of my stresses. Special thanks in particular to Carl Eastlund, Ryan Culpepper, and Sam Tobin-Hochstadt for being in my corner (literally and figuratively). Thanks as well to Christos Dimoulas, Ryan Culpepper, Stephen Chang, and Jesse Tov, for joining me as fellow teaching assistants (a.k.a. keeping me sane) during "Boot camp."

To Professors Mitch Wand and Matthias Felleisen, thank you for your assistance and encouragement in the development of my teaching skills.

To my thesis committee members, Gene Cooperman, Olin Shivers, and Guy Steele: thank you for your time spent poring over this document, your advice on how to improve it, and most of all, for the encouragement you provided to me.

To my advisor, Professor Will Clinger, thank you for being a source of support and advice in my research efforts, especially in the development of the Larceny code base. Perhaps not every exercise qualified as a step forward in the world of research; but they were all crucial in the development of my knowledge of programming language runtimes and just-in-time compilers, and that skill set has been a crucial asset in my industrial career.

To my parents, I cannot thank you enough for the support and guidance you have always provided. You have both challenged and inspired me.

To Stephanie, who changed my outlook on life. You have been my rock, my partner, and I cannot think of a better companion for the years ahead.

Contents

List of Figures

Chapter 1

Introduction

The ongoing shift from 32-bit to 64-bit processor environments forces garbage collectors to cope with the larger heaps made possible by the increased address space. On 32-bit machines, generational collectors that occasionally pause to collect the entire heap work well enough for many applications, but that paradigm does not scale up because collection pauses that take time proportional to the total heap size can cause alarming or annoying delays [24], even if they occur rarely.

Real-time, incremental, and concurrent collectors eliminate such delays but introduce complex invariants to the memory-management system. Maintenance of these invariants during execution reduces application throughput. Also, supporting these invariants increases the complexity of compilers, run-time infrastructure, and low-level libraries (e.g., client modules written in C and linked via a foreign function interface).

In non-real-time operating environments, real-time garbage collection is overkill. It would be better to preserve the throughput of generational collectors while eliminating the long delays associated with major collections. Implementors would also appreciate a system with hard bounds on pause times, but simpler than contemporary real-time memory managers.

Will Clinger (my thesis advisor) and I have designed, and I have implemented, a *regional garbage collector* that collects bounded subsets of the heap

during *every* collection, thus disentangling the worst-case mutator pause time from the total heap size. The design separates copying collections from auxiliary tasks that perform "summarization" and "refinement."

The regional collector incorporates a novel solution to the problem of "popular objects." The regional collector generalizes this problem from objects to regions. With that generalization, relatively few regions can be popular, so those popular regions can be temporarily "waved off" from collection *without* violating asymptotic bounds for space efficiency. As explained in Section 4.2.3 and proven in Section 5.4, this insight yields a solution to the popularity problem.

The regional collector also introduces a novel "fame" heuristic, an extension of popular region wave-off, to reduce the overhead of regional remembered set size and maintenance. This fame heuristic tends to improve throughput without sacrificing the worst case. The fame heuristic is described and evaluated in Section 7.3.

The three primary goals of the design are:

1. Constant worst-case bounds for the CPU time required by each collection, and constant worst-case lower-bounds for minimum mutator utilization (for granularities coarser than the worst-case CPU time bound for each collection).

2. Asymptotic worst-case bounds for memory usage, within a small constant factor of the total volume of live storage.

3. Typical throughput competitive with conventional generational garbage collection technology.

A prototype of this *regional collector* implemented atop the Larceny runtime shows that it performs acceptably on a wide range of benchmarks: It is comparable to an efficient generational collector on a set of fifty-eight small, non-collection-intensive benchmarks, and achieves acceptable throughput

without violating its bounds on a set of thirteen collection-intensive benchmarks.

My thesis is: **Regional garbage collection with summarization, wave-off, and snapshot refinement, provides mutator-independent worst-case bounds on pause times and minimum mutator utilization, and provides competitive throughput while maintaining a worst-case bound on overall memory usage.**

Chapter 2

Garbage collection: background and goals

This chapter outlines the goals of garbage-collection systems, defines standard terms used in the Computer Science community, and describes the particular problem solved in this work.

2.1 Garbage collection background

One way to document the design of an automatic memory manager is to describe how it supports and interacts with the main program, called the *mutator*. Most of the mutator's state is made up of object structures that are allocated in a portion of memory called the *heap*. Each object structure (or simply *object*) is made up of a sequence of words. Some objects may have header words that hold meta-data about the object, such as a type tag; others may have no header at all. The rest of the words in an object, depending on its type, are either raw binary data (uninterpreted by the garbage collector), or the object's *slots* (or *fields*); each slot potentially holds a reference to another object on the heap.

During the course of a computation, the mutator issues requests for new

objects to the run-time environment. The memory manager responds by identifying an unused area of the heap capable of holding an object of the requested size and returning its address. This action could be trivial if the heap has free space available, but if the free space is exhausted, the memory manager invokes the garbage collector to identify heap memory that is no longer usable by the mutator (or *dead*) and thus can be reclaimed. An automatic memory manager generally needs to perform dynamic analysis of the system state in order to identify such reclaimable storage, as opposed to memory-management systems that rely solely on static program analyses (which fundamentally cannot achieve the same level of precision as a dynamic analysis).

As objects are allocated and returned to the heap, the memory may become *fragmented*, depending on memory-management policies. If the system leaves gaps of free memory that are too small to accommodate future requests, then it exhibits *external fragmentation*. If the system allocates extra storage without intending to use it, for example by allocating an overly large structure to ensure that memory addresses will be properly aligned, then it exhibits *internal fragmentation*.

A *tracing garbage collector* (or often just *garbage collector*) identifies storage to reclaim by starting from a fixed set of the object references provided by the mutator, known as the *root set*, and transitively following the references to determine what objects the mutator could possibly reach. The objects reachable in this manner are the *live* objects; a sound mutator must access only objects that it can reach via some path of references from one of its roots. A simple tracing collector traverses all of the live objects starting from the root set; thus a simple tracing collector can pause the mutator for a duration proportional to the volume of live storage.

The memory store is often presented abstractly as a directed graph whose vertices are the elements of the root set and objects in the heap and whose edges are the object references in the objects' slots and in the root set. A

garbage collector finds some connected component of the abstract store that includes the root set; I sometimes refer to the smallest such connected component as the *object graph*, and to other such connected components as conservative approximations of the object graph. A conservative approximation of the object graph often results from treating unreachable object structures as if they are live (see "*float*" below).

A *copying collector* is a tracing collector that copies (or *forwards*) objects into a free memory area as it traces them, preserving the object graph structure by updating the references within all objects (both forwarded and non-forwarded) to refer to the new copies. (Having the freedom to manipulate the representation of the object graph in this manner can reduce fragmentation and improve locality.)

A *generational collector* is a tracing collector that partitions the heap in some manner so that younger objects are classified separately from older objects. (The notion of "object age" can differ between collector designs, but for now it suffices to think of it as some measure of the time since the object was initially allocated by the mutator.) The collector attempts to reduce tracing overhead by tracing only the young objects during most collections. In some generational collectors, the youngest generation is known as the *nursery*. Most objects are initially allocated in the nursery; when it fills up, *minor collection* evacuates all live objects out of the nursery alone into an older generation. (If the older generation runs out of room, a rare *major collection* traces through both the old and young objects.)

Any collector that reclaims dead storage from a part of the heap by tracing only objects within that part of the heap must ensure that there is no way to reach any reclaimed objects via some untraced path through the unprocessed portion of the heap. This assurance is typically provided by maintaining a *remembered set*: a set of objects that have references into the collected subset. Including the remembered set as part of the root set ensures that any reachable object in the collected subset will not be reclaimed.

Thus a generational collector must maintain a remembered set to track the older objects that have references to young objects.

When a collector maintains a *precise* remembered set, it is responsible for ensuring that any object in the remembered set actually does contain a reference that will need to be included during some future collection. Thus with maximally precise remembered sets there is a double implication, in that an object is in the remembered set if *and only if* it contains a reference that crosses the heap partitioning. Many generational collectors guarantee only that they maintain *imprecise* remembered sets, where any object with a reference that crosses the heap partitioning must be in the remembered set, but there is no constraint on how many extra objects with no such references can occur in the remembered sets.

Collectors that use a conservative approximation of the object graph may treat some unreachable object structures as if they were live. This *floating garbage*, or *float*, is not reclaimed until the collector refines its approximation of the object graph. Some amount of float is usually acceptable in an efficient collector, as the point of collecting only a part of the heap is to avoid the cost of analyzing the whole heap structure to determine the exact set of live objects. But if the amount of float grows unreasonably large, then performance can suffer: the memory usage becomes unacceptably high, and a copying collector wastes time copying and maintaining the useless data of the float objects.

In many collectors, particularly generational collectors, the mutator must notify the collector when it makes modifications to the memory store, so the collector can maintain internal meta-information about object referencing relationships in the store. Such notification is usually performed by a snippet of code that is automatically emitted by the compiler alongside every operation that modifies a memory cell in the store; this snippet is referred to as a *write barrier*. The main purpose of the write barrier in a generational collector is to ensure that references from old to young objects introduced

by mutation operations are saved in the appropriate remembered set; references from young to old objects need not be saved in a generational collector and are typically filtered out by the write barrier.

In an *incremental* collector, the work of collection is divided into small chunks, so that control passes to the collector for only a fixed amount of time before it returns to the mutator. A problem is that bounding an individual pause time is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high. The *mutator utilization* of a collector is the fraction of time in which the mutator does useful work in a given period; thus the *minimum mutator utilization* (MMU) is a lower bound on how much work the mutator is able to get done despite interruptions by the collector.

A *concurrent* collector runs some or all of the collection-related tasks in parallel with the mutator. The core difficulty of concurrent collection is that the mutator's and collector's views of the heap must be kept coherent as the concurrent tasks proceed. Supporting object forwarding concurrently with the mutator is possible but involves the maintenance of complex invariants.

The work performed by a computing system can be measured in a variety of ways, such as wall-clock elapsed time, or number of processor cycles. For most of our abstract discussions, we measure work performed by the mutator and garbage collector by the memory operations they perform: memory reads, memory writes, mutator allocation requests, and collector memory allocation and freeing. Therefore, we often present elapsed mutator time as a count of memory writes and allocation requests the mutator can make before control shifts to the collector, and collection pause times as the number of memory operations that the collector must perform before it can pass control back to the mutator. As long as the total memory usage remains reasonably bounded[1], this is not an absurd simplification, especially consid-

[1]The bound on total memory usage is relevant for simplifying our reasoning about elapsed time; it allows us to assume that memory operations do not cause significantly

ering the ever widening CPU/memory gap. We present wall-clock times in our performance results, however.

The regional collector presented here is implemented atop the Larceny runtime system. Its performance is compared against the stop-and-copy and generational collectors provided in Larceny. More information on Larceny and a download of the Larceny runtime is available at the following website.

`http://www.larcenists.org/`

2.2 The facets of scalability

Scalable systems must have reasonable interactive performance, without paying too much in terms of memory or throughput overhead.

Unlike standard generational collectors, the regional collector presented here is *scalable*: Theorem 1 below establishes that the regional collector's theoretical worst-case collection latency and MMU are bounded by nontrivial constants that are independent of the volume of reachable storage and are also independent of mutator behavior. The theorem also states that these fixed bounds are achieved in space bounded by a fixed multiple of the volume of reachable storage.

Although most real-time, incremental, or concurrent collectors appear to be designed for embedded systems in which they can be tuned for a particular mutator, some (though not all) hard real-time collectors are scalable in the same sense as the regional collector.

For programs that will run under non-real-time operating systems, hard real-time garbage collection is overkill. Many programs do not need guarantees for minimum mutator utilization at sub-millisecond resolutions, but would benefit from general-purpose scalable collectors that provide guaranteed lower bounds for worst-case MMU at resolutions of one second or less.

more OS-level page faults than would have occurred with some other collector.

If the relaxed resolution were accompanied by superior MMU and overall efficiency for the average case, all the better.

The following theorem characterizes the regional collector's worst-case performance.

Theorem 1. *There exist positive constants c_0, c_1, c_2, and c_3 such that, for every mutator, no matter what the mutator does:*

1. *GC pauses are independent of heap size: c_0 is larger than the worst-case time between mutator actions.*

2. *Minimum mutator utilization is bounded below by constants that are independent of heap size: within every interval of time longer than $3c_0$, the MMU is greater than c_1.*

3. *Memory usage is $O(P)$, where P is the peak volume of reachable objects: the total memory used by the mutator and collector is less than $c_2P + c_3$.*

The constants c_0, c_1, c_2, and c_3 are completely independent of the mutator. Their values do depend upon several parameters of the regional collector, upon details of how the collector is implemented in software, and upon the hardware used to execute the mutator and collector. Chapter 9 reports on the performance actually observed on a number of benchmarks.

To enforce scalability, the collector adheres to a set of policies that constrain the behavior of the mutator. Both the allocation rate and the amount of heap modifications must be bounded. To give a flavor for the mathematics involved, here are the formulas for the relevant bounds:

- Allocation during any one full cycle, A, is kept proportional to peak storage over the execution history. This is enforced via the policy

$$A = \min\left(\frac{1}{2}((1-k)L_{hard} - 1)P_{old}, (L_{soft} - 1)P_{old}\right).$$

- The mutator activity during a summarization cycle, C, is at most the product cN where

$$0 < c \leq \frac{F_2 F_3 - 1}{F_1 F_2} S - \frac{S}{\lceil \frac{N}{R} \rceil} - 1.$$

These formulas, their parameters, and relevant new terminology will be explained in the remainder of the document (largely in Chapter 5).

Chapter 3

Design space for heap-partitioning collection

This chapter provides a background for the regional collector via an exploration of the design space. I evaluate different points in the space by anticipating whether such a design would make it more difficult to guarantee bounds on pause times, mutator utilization, and memory usage. Not every point in the design space provides a suitable basis for scalable garbage collection.

3.1 Partitioning for independent collection

Ensuring scalability first requires that there be an upper bound on pause time: when the mutator shifts control to the collector's coroutines, they must finish their work within a fixed amount of time. Batching together related pieces of collection work is a second goal: forwarding several objects at once and reclaiming a large chunk of memory is preferable to working on only a single object at a time. Such batching of labor reduces overall system overhead, though it also entails that I cannot impose pause-time bounds as short as some real-time collectors [10, 15].

The design assumes that the collector will need to migrate objects; that is, it will use a copying collection scheme to some degree, rather than relying solely on a pure mark/sweep strategy that does not forward objects. This assumption is motivated by the difficulty of bounding the amount of memory fragmentation in a pure mark/sweep collector without significantly constraining the mutator *a priori*. If a memory-management strategy is to be space-efficient, it cannot allow fragmentation to grow without bound. Other arguments for favoring a design allowing object migration include enabling bump-pointer allocation rather than free-lists, and the potential for improving memory locality.

The first step I take towards bounding collection work in a copying collector appears simple: Partition the heap into disjoint regions, where each region is bounded in size. Then ensure that the collector only works toward reclaiming the memory associated with one region at a time. The region size bound is presumed large (with respect to the size of individual objects) to batch collection operations together and to bound the relative amount of internal fragmentation due to unused space in a region.

The fixed region size bounds the amount of copying performed, which may seem like it would immediately bound the maximum pause time. The mistake in such reasoning is that it ignores the effort required (1) to identify the region's live objects, which may be reachable only indirectly via objects in other regions, and (2) to update all references to the forwarded objects, *including* those outside the selected region.

3.2 Remembering region-crossing references

Generational collectors face a similar problem as the regional collector: they need to collect young generations without scanning the entirety of old generations, as the cost of such scanning would be self-defeating to their goal of avoiding work proportional to the size of the older generations. Most

modern generational collectors hosted on generic hardware employ some form of a *remembered set* [28] to track old-to-young pointers; the mutator is responsible for ensuring that modifications to old objects properly maintain the remembered-set invariant:

> Remembered-set invariant, Generational:
> If live object B is older than object A and B has a reference to A, then track B in the remembered set.

With this invariant in place, on each attempt to collect garbage a generational collector can choose a prefix of the generations (assuming a youngest-to-oldest ordering) and collect only the objects in that prefix, scanning the objects in the remembered set to find all pointers into the collected prefix from the older uncollected generations.

A generalization of this idea allows a region to be collected independently of other regions: use a remembered set that tracks references that cross regions, without regard to ordering (age-based or otherwise). In this scheme, the mutator must now ensure that modifications to objects maintain the following invariant:

> Remembered-set invariant, Regional:
> If live objects A and B belong to distinct regions and B has a reference to A, then track B in the remembered set.

3.2.1 The remembered set is a heuristic

With the addition of a regional remembered set, the collector can focus its attention on just the objects that contain region-crossing references, without scanning the entire remainder of the heap outside the collected region.

However, this elaboration of a generational collector's design does not provide a guaranteed bound on pause times. The remembered set might, as a direct consequence of the invariant, contain the address of every object in every region (especially if the object distribution across regions is particu-

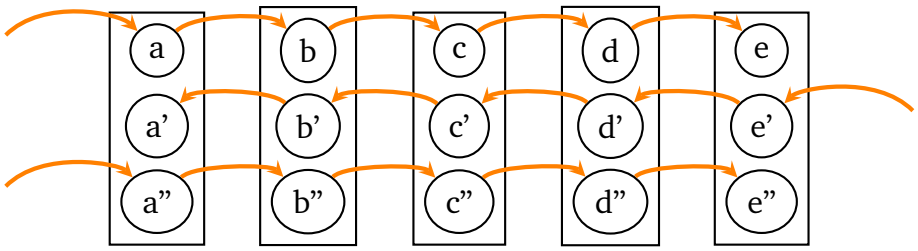


Figure 3.1: A pathological object distribution

larly bad, as illustrated in figure 3.1). Thus the remembered set can grow proportionally with the heap; some application benchmarks exhibit such growth. If collection pause time were proportional to remembered-set size, then the pause time would not be bounded by any application-independent constant.

3.3 Tracking region crossings: the design space

The regional remembered-set invariant (page 15) implicitly suggests one of many possible structurings of collector meta-data for tracking region-crossing references. At this point, it is useful to take a step back and consider alternative structures for narrowing the focus of the collector.

3.3.1 Points-out-of and points-into

One way of comparing such structures is to analyze how they distribute information across the set of regions.

The regional remembered-set structure was presented earlier as a single monolithic entity for the entire heap; that view is interchangeable with one that perceives the remembered set as an array of individual disjoint sets, one for each region. Each region's set then tracks objects within that region that may have references that point out to objects in other regions. I refer to such a structure as a "points-out-of" structure. Figure 3.2 illustrates such

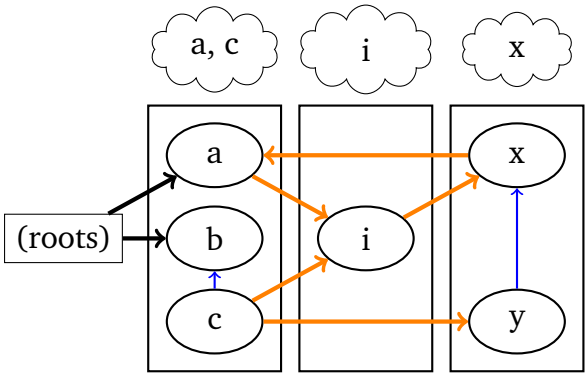


Figure 3.2: A "points-out-of" structure

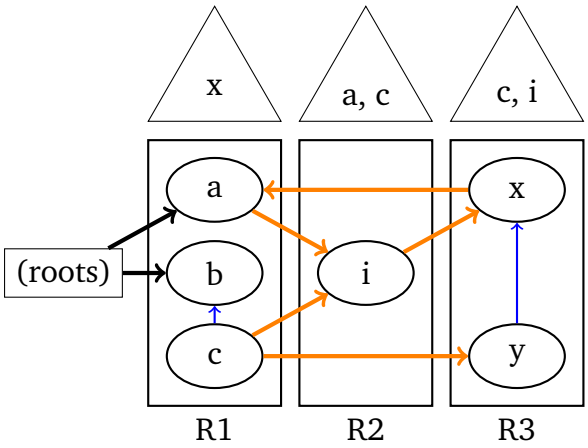


Figure 3.3: A "points-into" structure

a structure in a partitioned heap diagram; the clouds sitting above the regions collectively represent the remembered set (or, equivalently, an array of disjoint sets). Note that all objects holding a reference pointing out (of their respective region) appear in the remembered set, as required by the invariant.

It is not hard to manipulate this abstract picture to obtain alternative structures. One choice is to change what state is stored per-region so that instead of tracking references going *out of* a region, one instead tracks references coming *into* a region. Such a "points-into" structure is illustrated in figure 3.3.

The object graphs in figures 3.2 and 3.3 are identical; the only difference

is in how the meta-data structures of the collector abstractly describe the region-crossing relationships.

One apparent difference between these approaches is that the points-into structure provides the collector with an immediate focus on the objects relevant to collecting a region. For example, collecting region R3 by forwarding the objects x and y will require updating references in the objects i and c, respectively; both i and c appear directly in the points-into structure for R3, and the collector need not inspect the irrelevant objects held in the points-into structure for other regions.

In summary, the "points-out-of"/"points-into" distinction describes a variation in how reference-tracking meta-data could be distributed across (and abstractly charged to) different parts of the heap, using the direction of the tracked references (from the relative viewpoint of the region associated with the meta-data) as a convenient mnemonic. A "points-out-of" structure is the natural generalization of a standard generational remembered-set representation, but a "points-into" structure can represent a more focused view for the collector.

Lest the picture appear overly rosy for "points-into," section 3.3.3 discusses the main drawback to a points-into structure.

3.3.2 Imprecision

The previous section defined a directional mnemonic, "points-out-of" and "points-into," describing one manner in which reference-tracking meta-data could vary. Another important attribute of this meta-data is revealed by investigating an entirely different kind of direction: the direction of the implication in the regional remembered-set invariant. In particular, the invariant describes a unidirectional implication, not a bidirectional one. This detail is exactly what allows for the reference tracking performed by a remembered-set structure to be *imprecise*.

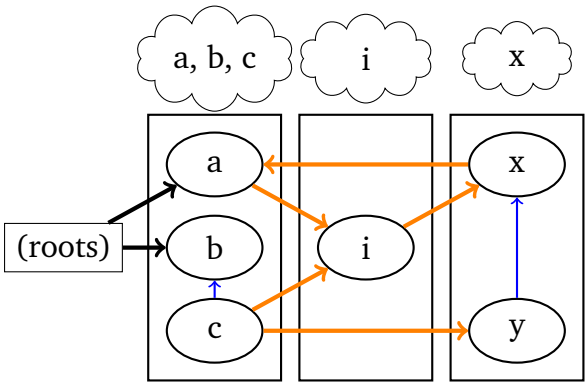


Figure 3.4: Less precise "points-out-of" structure

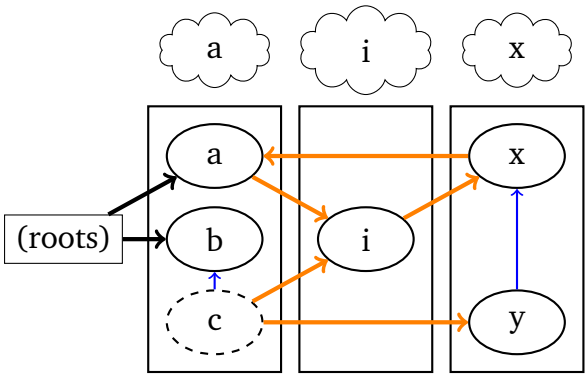


Figure 3.5: A maximally precise "points-out-of" structure

The converse of the invariant's implication is "if object B is tracked, then (1) there exist live objects A and B in distinct regions and (2) B has a reference to A." This statement can be violated in two interesting ways: a live object B can be tracked, but have no reference to an object in a distinct region, or B itself can be dead. Each of these two situations is a separate source of *imprecision* in a collector's reference tracking structure.

The first kind of imprecision is illustrated in figure 3.4; the object b has no references to any objects in other regions, yet is included in the remembered set. This sort of imprecision can arise when the mutator changes the object graph structure. The illustrated configuration could arise if b in the past had referred to some object in a distinct region (e.g. i), but the mutator replaced the reference to that object in b with another value.

The second kind of imprecision has already been implicitly illustrated: the remembered set in figure 3.2 is not minimal, because it contains the object c which is not reachable via any path from the roots. A minimal (and thus maximally precise) remembered set for the same heap is illustrated in figure 3.5.

Allowing imprecision in collector meta-data structures is important because it is too expensive to maintain maximal precision at all times. Consider the maximally precise remembered set in figure 3.5: A single modification by the mutator may require significant meta-data revision to recover maximal precision. For example, if the mutator were to change the object a to refer to b instead of i, then recovering maximal precision would obviously require removing a from the remembered set, since a no longer holds region-crossing references (though determining that might be expensive). It would also require removing the objects i and x, as the modification makes them unreachable. Correctly determining that all three must be removed and also performing the removal would be complicated and add too much overhead to the mutator's actions.

A "points-into" structure similarly requires some degree of imprecision. But this leads to a crucial problem with adopting a "points-into" structure, discussed in the next section.

3.3.3 Imprecision hinders bounding space for "points-into"

Since imprecision allows extra entries to appear in the reference tracking structure (be it "points-out-of" or "points-into") an obvious question arises: how much space could the extra entries occupy? Could the garbage collection meta-data violate the asymptotic space-efficiency bound?

For "points-out-of," there is a clear way to bound the space: since each object appears at most once in the remembered set, its structure cannot grow larger than the heap itself. Equivalently, each remembered-set cloud sitting

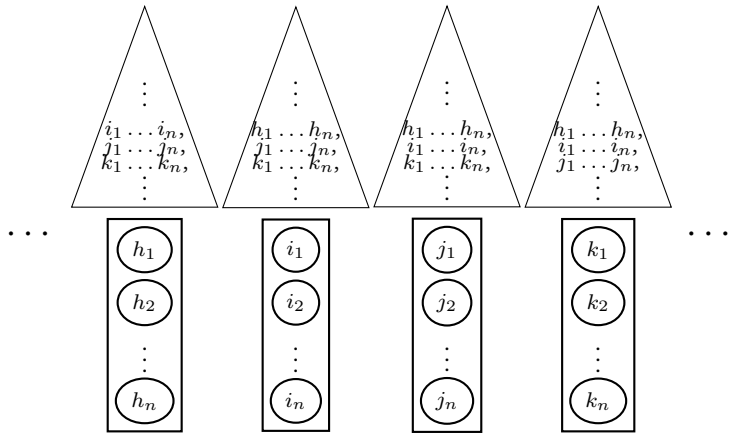


Figure 3.6: Quadratic space blowup of naïve "points-into"

above the regions in Figures 3.2, 3.4 and 3.5 can grow no larger than the region it is associated with.

Unfortunately, for "points-into," there is no similar linear structural bound on its size, because each object can appear multiple times in the entire structure, as illustrated by the two occurrences of c in figure 3.3. There is only a quadratic structural bound on the size of a "points-into" structure; if precision is not otherwise bounded, then a worst-case mutator will cause every region's associated set to contain every object in every *other* region, as illustrated in figure 3.6.

The two kinds of imprecision described in section 3.3.2 make the "points-into" structure less attractive. There is a third important kind of precision distinct from these two: the granularity of the reference tracking structure. This is the topic of the next section.

3.3.4 Granularity: objects versus locations

An implicit assumption in the presentation so far is that the collector meta-data accumulates objects, without tracking which slot within each object holds (or held) a region-crossing reference. When the collector attempts to utilize such a meta-data structure, it will need to scan each object to find all

region-crossing references it holds.

An alternative to working at the granularity of whole objects is to work at the granularity of individual locations of slots within the objects: a *location-tracking* rather than object-tracking structure. A location-tracking structure obeys the invariant that if slot i in live object B has a reference to object A in a different region, then the structure holds the location of $B[i]$ (in the notation of the C programming language, `&B[i]`, assuming the slot is located i words from the start of the object). Note that one object with multiple region-crossing fields will yield multiple entries in such a structure; thus there is a potential increase in meta-data space usage.

If a large object has few region-crossing references, focusing the collector's attention on particular slots within the object is cheaper than scanning the object in its entirety.

If most objects are small then tracking individual locations within the object may increase memory usage, as each object may contribute multiple entries to the meta-data structure, but saves little time. That is a bad tradeoff. One way to counter this problem is to track locations at a coarser grain than individual slots: when a location l needs to be tracked, a number of other locations near l in memory are also tracked in the meta-data structure. A standard way to achieve this is to store only the most significant bits of the word representing l; if several nearby locations need to be tracked, only one entry is added to the structure. Then, when the collector traverses the entries in the structure, it walks through all of the locations whose high-order bits match each entry. Such coarse-grained location-tracking structures are often called *card tables* [29] in the garbage collection literature. I add the qualification that such structures are *coarse-grained card tables*, to make it clear that the cards are introducing a kind of imprecision. One might imagine a similar bitmap structure that did not coalesce as many locations into one entry; such a structure could then be called a *fine-grained card table*.

Comparing coarse-grained and fine-grained card tables provides a clear

example of adjusting precision in order to trade time spent scanning for space (and, potentially, time) gained from a more compact imprecise representation. Card tables also illustrate that the choice of whether to track locations or objects is orthogonal to the level of precision sought.

3.3.5 Understanding the design space

I have presented three different design axes for a heap-partitioning garbage collector's meta-data structure: (1) "points-out-of" versus "points-into," (2) whether entries correspond to whole objects or to (sets of) locations, and (3) the degree of precision.

A typical remembered set that builds a hashtable of objects is a relatively precise points-out-of object-tracking structure. A card table is a relatively imprecise points-out-of location-tracking structure. The atypical remembered sets of Sun's garbage-first collector [18] are imprecise points-into location-tracking structures; points-into structures have other precedents as well, discussed in section 10.2.

I emphatically do not claim that these three options are the only axes on which a collector's meta-data structure may vary; instead, I present these axes because they represent three important technological differences between a typical generational collector and a regional collector.

To my knowledge, the structural distinction between "points-out-of" and "points-into" has not been previously explored in the manner above. This oversight can be explained by observing that the distinction is more significant for a system maintaining the regional remembered-set invariant than for a system maintaining the generational invariant. Figure 3.7 illustrates this with a generational heap partitioning, where the upper generations in the diagram are younger than the ones below. The diagram shows a "points-out-of" structure for the heap via the cloud shapes on the left, and a "points-into" structure via the triangles on the right.[1] A generational collector works

[1] This is not the only "points-into" structure imaginable; for example an alternative

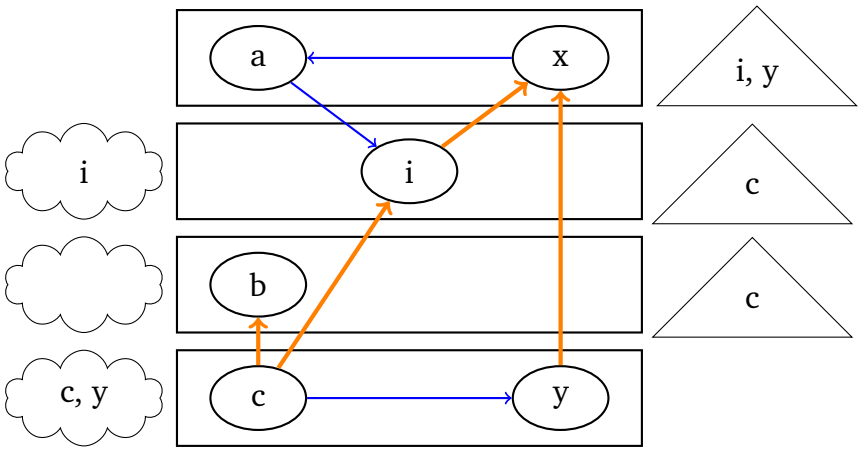


Figure 3.7: Generational "points-out-of" vs. "points-into"

by selecting a prefix of the generations and using the meta-data structure as a source for additional roots to scan. A collection of the top two generations in a points-out-of system would require scavenging the cloud structures of the bottom two generations; the same collection in a points-into system would require scavenging the triangle structures of the top two generations.

A naïve inspection of the situation might lead one to think that the "points-into" structure on the left is a mild reorganization of the "points-out-of" structure on the right. However, each setup will require different amounts of scavenging effort, and an imprecise "points-into" structure does not have a linear structural bound on its space-usage, as discussed in section 3.3.3.

3.3.6 Popular objects: another blight for points-into

There is no bound on the number of incoming references that any particular object (or set of objects) may have. In particular, a single object may be referenced by a significant proportion of the heap. This is not a hypothetical problem; many applications have some central data structure or a collection

structure could, for each generation G, track the objects with references into G and also objects with references into any generation younger than G.

of interned objects that many other objects refer to. This is known as the "popular object problem" [18] in the GC community.

The presence of popular objects means that even with a 100% precise points-into meta-data structure, one would generally not be able to migrate all objects out of an arbitrary region within a bounded pause time. If the region contains a popular object, it will take time proportional to the size of the heap to process the region's points-into structure and update all of the references to the migrated popular object. The problem also generalizes to the case when a single region holds a set of semi-popular objects; to handle the worst case, one must address both situations.

Typical incremental collectors handle this by allowing both the popular object and a copy of it to persist simultaneously while the mutator runs. Then the work of updating all of the references to a popular object can be broken up into smaller units, at the cost of introducing overhead in space (the two versions of the object) and in time (the mutator must cooperate with the collector's concurrent copying). We suggest a significantly different approach, discussed in the next section.

3.4 Bounding collection pause times: insights

Section 3.1 established that when copying one region independently from the others, the interesting question is how to identify (and update all references to) the live objects in the region. Section 3.2 showed that a "points-out-of" remembered set will not provide guaranteed bounds on pause time. Sections 3.3.3 and 3.3.6 indicated that a "points-into" structure does not provide an immediate bound either, because of problems introduced by imprecision and popularity.

3.4.1 The popularity insight

Popularity comes from the structure of the heap itself: words within the heap are what contribute to the popularity of any particular region. Some number of regions may be so popular that they would have a points-into structure of size proportional to the heap, but most regions cannot be so popular at any particular instant. This is related to the observation that it is impossible for all regions to be more popular than average.

Section 5.4 presents a generalization of both of these observations formally. The upshot of this insight: The phenomenon of popular objects does not invalidate use of a points-into structure; it simply necessitates a bit more care for how such structure is used.

3.4.2 The imprecision insight

Imprecision can be tackled by taking a different view on maintenance of meta-data. Imprecision arises because the meta-data structure is not constructed solely from information garnered from the heap at one instant in time, but rather from smearing together a series of heaps, where mutator activity is introducing the gradual changes in the elements of the series. If one attempts to maintain an imprecise structure for too long, such smearing could make the structure take the useless form depicted in figure 3.6.

Bounding the degree of introduced imprecision is necessary for a points-into structure to work in general. The main approach I employ for bounding imprecision of a points-into structure is *on-demand construction*, discussed in the next chapter.

Chapter 4

Abstract structure of regional collection

The control structure of the regional memory management scheme is divided into four main components: the mutator coroutine, the forwarder, the marker, and the summarizer; the latter three components constitute the coroutines of the collector. The division into two mutator and collector coroutines is a standard design for garbage collected languages. The regional collector coroutine structure is a slight refinement.

The summarizer and marker coroutines both bound imprecision in the regional collector. The summarizer provides *on-demand construction* of points-into summary sets: rather than maintain meta-data for the whole heap during the entire application run, the regional collector incrementally constructs the points-into structure of a region scheduled for future collection. After the collector processes the region, region's points-into summary set is discarded. Section 4.2 further describes the summarizer component.

The marker coroutine provides *snapshot-based refinement* of the collector's meta-data. It incrementally traces a snapshot of the heap as the mutator and forwarder each progress on their own. After the marker completes its construction of the snapshot, the collector uses the snapshot to refine the

meta-data, regaining precision lost due to mutator actions. Refinement of the meta-data ensures that the presence of unprocessed regions and cyclic garbage do not lead to violation of the system's bounds on overall memory usage.

The intention of this component structure is to assign the bulk of the collection work (in the common case) to the forwarder, which copies objects and reclaims the newly unoccupied areas the objects came from. The addition of the marker ensures that the system satisfies its space bounds; the summarizer its pause-time bounds.

This chapter motivates the above decomposition by presenting a high-level overview of these components: what purpose they serve, why each is necessary, and how they interact. Descriptions of some system-wide invariants are included when they would be illuminating.

4.1 The summarization solution

As stated earlier, a "points-out-of" remembered set may grow proportionally with the heap. Therefore it is not generally acceptable to scan an entire "points-out-of" remembered set during a collection pause.

However, much of a "points-out-of" remembered set structure may not be relevant to the collection of a particular region.[1] Therefore, rather than waiting until a region is actually collected to scan the remembered set for incoming references, the regional collector periodically starts a *summarization* routine (or *summarizer*) for a subset of the regions. The summarizer incrementally scans the remembered set and builds up "points-into" *summaries* for each selected region, where a summary is the collection of locations relevant to collecting that region.

[1] Moreover, the collector ensures that there always exist regions for which not too much of the remembered set structure is relevant; see section 5.4.

Only regions with fully constructed summaries are eligible for collection. Furthermore, if a region's summary becomes too large, the region is removed from the set of candidates for collection. Thus, instead of requiring a scan of the entire remembered set during a collection pause, the collector need only focus its attention on the summary for the collected region. Since the summary for every collection candidate is bounded in size, the time spent scanning the summary is likewise bounded; we can find all of the pointers into a collected region within the pause time bounds.

Since the collector uses a region's summary to find references into the region, each summary must contain a superset of the locations pointing into its region; otherwise live objects within the region could be overlooked and erroneously reclaimed during collection. Therefore the regional collector must maintain a summarization invariant:

> Summarization Invariant:
> If live objects A and B belong to distinct regions, B has a slot f where $B[f]$ holds a reference to A, and A resides in a region considered eligible for collection, then the address of $B[f]$ is in the summary for A's region.

Summary construction and maintenance is complex. To my knowledge it is a novel aspect of this work (though others have made similar constructions).

4.2 Revising the reference-tracking structure

Chapter 3 presented both "points-out-of" and "points-into" reference-tracking meta-data structures. A typical remembered set is "points-out-of." If the volume of object locations held in a remembered set were small relative to the size of a collected region, then scanning the remembered set for additional references would not add significant overhead to the cost of collecting the region (assuming the remembered set does not use a coarse-

grained card table as described in section 3.3.4). But there is no guarantee that the "points-out-of" remembered set will be relatively small, as discussed in section 3.2.

4.2.1 Summaries are constructed on-demand

A "points-into" design may have more promise, as noted in section 3.4. A potential objection to it is: If one were to maintain such structures for all of the heap at once, even with 100% precision, then the "points-into" structure associated with each region could get quite large, and the sum of the space occupied by all such structures could be prohibitively large.

However, there is no requirement that such structures be maintained for all of the heap at once. That is the crucial counter to the objection: instead of maintaining "points-into" structures universally, one can instead construct such structures for a *proper subset* of the regions, and perform their construction on an *on-demand* basis. I refer to the "points-into" structures so built as *points-into summary sets* (or simply *summaries* when clear from context).

4.2.2 Summaries are imprecise, but not too imprecise

A second objection is that maintaining "points-into" summary sets at 100% precision, even for a subset of the regions, is too expensive. I deal with this problem by allowing the points-into summary sets to be imprecise, but bounding the amount of imprecision that can be introduced. (This sounds straight forward, but getting the details right requires careful analysis; see chapters 5 and 6.)

4.2.3 Summaries are waved off before getting too large

A third potential objection is that the number of references into any one region is limited only by the size of the heap; thus a complete points-into structure for such a popular region would still require time proportional to the heap to process.

Section 3.4.1 hinted at a simple solution to this third problem: this popularity scenario cannot be the case for *all* of the regions. So if a region's complete summary would be too large to process in our time bound, the runtime *waves off* (abandons) collection of the region this time around. Since the region is no longer scheduled for collection, one need not bother completing the construction of its summary either.

Waving off a popular region requires choosing another region to collect; therefore, one generally needs to construct multiple summary sets at once, because it will not suffice to build just one. This illustrates that it would be a misnomer to describe "points-into" summary-set construction as a "just-in-time" process; one will need to construct multiple summaries at once, but spread out the consumption of the summaries over several collections. We shall see that only a bounded number of regions can be waved off, and that there are always summarized regions eligible for collection.

4.2.4 Summary sets hold locations, not objects

Section 3.3.4 discussed how a reference tracking structure could track locations within objects, rather than whole objects. The regional collector's points-into summary sets track individual locations, not objects. If the summary-set structure were to track whole objects, then the number of entries in a particular summary set would not correspond to a precise measure of the number of words that point into the region, yielding two distinct problems described in the remainder of this section.

Suppose that the summary-set structure tracked whole objects. Thus the entries in the summary set for region r are (some superset of) the addresses of every object outside of r that has a reference to an object within r. Let s_r be the number of entries in such a summary set for region r; thus when a collection of r occurs, the collector will iterate over the s_r entries, and for each entry e, scan all of the slots in the object represented by e, searching for references pointing to objects in r that must be updated.

This means that s_r is not a terribly useful bound on pause time, because each entry has a non-constant amount of scanning time associated with it. That is the first problem.

The second problem with tracking objects is that the argument alluded to in section 3.4.1 (and to be formally shown in section 5.4) requires that locations in the summary sets be properly accounted for. If the summary sets tracked objects, then a single entry in a summary set could represent one slot from the corresponding object x, or all of the slots of x. The collector would be forced to make conservative estimates of how popular a region was becoming, and so the popularity lemmas 4 and 7 (introduced later in section 5.4) would not hold.[2]

When tracking locations instead of objects, the number of entries in a summary directly corresponds to the amount of work necessary during a collection, and each slot in an object contributes to at most one entry in a summary.[3] So by tracking locations instead of objects, both of the problems from above go away.

Proper accounting requires tracking locations, not objects.

[2]In a simplified domain where the number of slots per object is significantly limited (e.g., where all objects are pairs), the object versus location distinction is mostly a distraction. When one object can hold thousands of slots, the distinction is important.

[3]The "one slot : one entry" correspondence glosses over issues introduced by imprecision due to mutator activity; these issues are addressed by lemma 7 in section 5.4.

4.3 On-demand summary set construction

As control transfers between the mutator and the forwarder, the summarizer is responsible for preparing future regions for collection.

4.3.1 Incremental summary construction

In general, building the summary set for any one region r will require searching the whole heap for locations of references to objects in r; this means that constructing any complete summary set will generally require time proportional to the size of the heap.

In general, the time between collections will not be proportional to the size of the heap; thus the summarizer will not have time between two collections to build a complete summary set.

The solution to this problem is to design the summarizer as an *incremental* algorithm: it starts working alongside the mutator, but may be interrupted when the mutator transfers control to the forwarder. The summarizer and forwarder must cooperate to ensure that any intermediate state of the summarizer is properly maintained by the forwarder.

In addition, the summarizer and the mutator must cooperate to guarantee the end summarization state will reflects changes introduced by the mutator; such cooperation is implemented via the mutator's write barrier.

4.3.2 Multiple-summary construction

Each collection will consume the summary associated with the collected region; that is, it will discard the summary after all of the objects in the collected region have been forwarded to other regions. Therefore, at a minimum the regional collector will consume one summary for every collected region.

As mentioned in the previous section, constructing any one summary generally requires work proportional to the size of the heap. If the summarizer were to focus on building only one summary set at a time, the rate of production could not always keep up with this lower bound on the rate of summary consumption. Therefore, the summarizer must build multiple summaries at once. The effort of scanning the heap can then be amortized across all of the constructed summaries.

4.3.3 Searching for region crossings

The goal of the summarizer is to establish the summarization invariant (page 29) for a suitable set of regions. One can imagine many potential techniques for constructing summary sets, especially since imprecision is allowed.

For example, one could incrementally trace the object graph (starting from the roots) and record all of the locations with region-crossing references that point into the regions being summarized. Alternatively, one could maintain a "points-out-of" remembered set and use that to guide the summarizer's scanning of the heap, narrowing its focus on the objects that contain region-crossing references. A third alternative is to gather region-crossing references via a linear traversal of the heap's address space, given suitable assumptions. [4]

The third approach will generally produce less precise summaries and will often scan more locations than the first and second, which may seem like two strikes against it. However, for our proofs we are only concerned with simple models and worst-case scenarios. Summarization overhead is maximal when the whole heap is filled with live objects that have as many

[4] In particular, if (1) the object layout is formatted so that references to other objects can be differentiated from raw bytes during a such a traversal and (2) there are means of filtering out objects identified as unreachable in the past (e.g., a mark bit or type-tag tricks) then a direct scan of the heap can work.

region-crossings as possible; in this worst-case scenario the three approaches to summarization will not produce different results.

Therefore for now I describe summarization as an algorithm that works via an incremental linear traversal of the heap address space. Chapter 7 describes the second strategy as an important refinement of the linear scan algorithm. This refinement is crucial for the common case but has no effect on the theoretical worst case; thus I omit it from discussion of the policies and proofs.

4.4 The summarization algorithm

A summarization pass targeting a subset $\{r, \ldots\}$ of the regions is an incremental traversal of the heap that attempts to construct summary sets for $\{r, \ldots\}$. As summarization progresses, control shifts between the mutator, collector, and summarizer coroutines. Since the collector may be invoked in the middle of a summarization pass and a region must be summarized to be eligible for collection, at the start of a pass a collection of regions (distinct from $\{r, \ldots\}$) must already have summaries available for consumption. Thus, the on-going goal of summarization is to establish sufficiently many summarized, collectible regions to allow the next wave of summarization.

4.4.1 Region categorization

Since the whole point of targeting regions for summarization is to make them eligible for collection, it would not make sense to summarize empty regions that contain no objects. A partially-filled region is also unlikely to be worth the effort of summary construction if there exists a filled region to target instead. Thus one can see a preliminary dynamic categorization of regions into four groups:

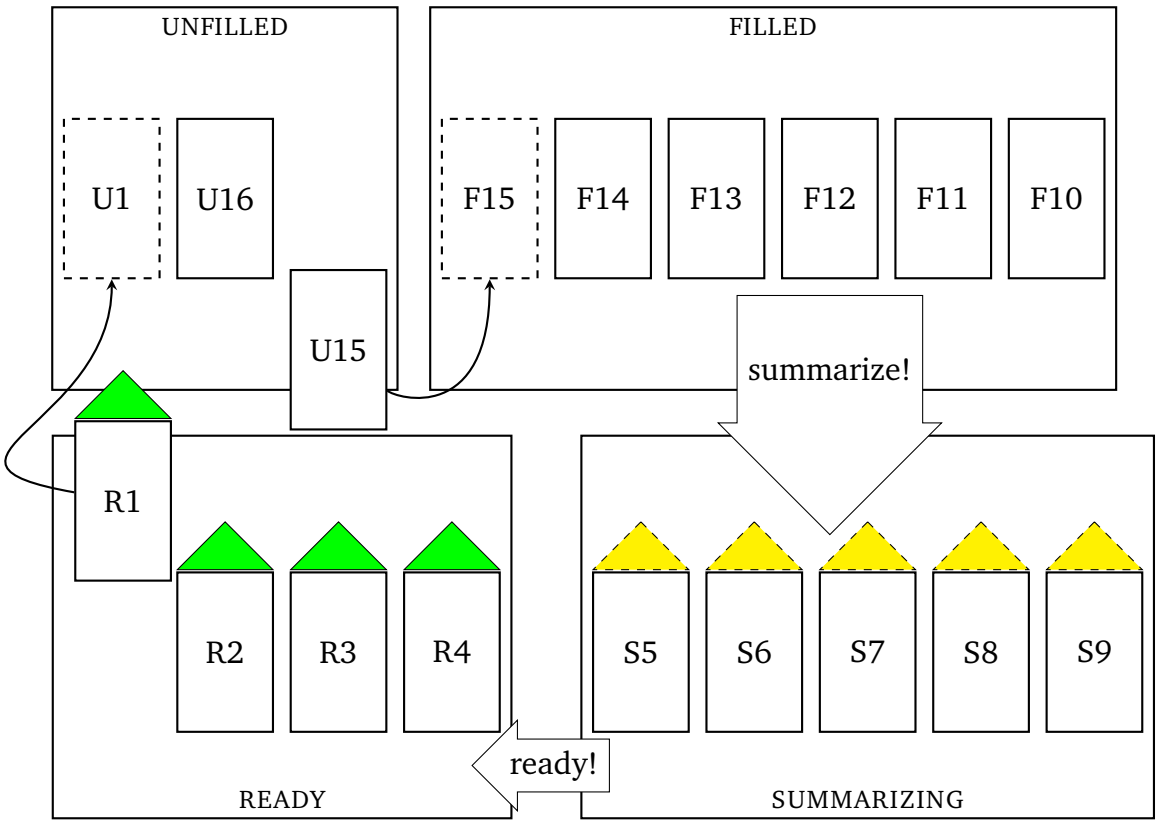


Figure 4.1: Preliminary region categorization

READY regions with complete constructed summaries and eligible for collection,

SUMMARIZING regions that are targets of the currently running summarization pass,

FILLED regions, recently filled with objects, that are eligible to be targets of a summarization pass, and

UNFILLED regions, not yet filled with objects, that are targets for the collector's object forwarding.

This preliminary categorization yields an immediate state transition diagram (figure 4.1) illustrating how the regions change roles over time. The thin arrow joining R1 to U1 represents the transition of a ready region when

the collector forwards all of the objects out of R1 and subsequently recategorizes the now empty region as UNFILLED. The thin arrow joining U15 to F15 represents the transition of an unfilled region when the collector fills it with newly-allocated and forwarded objects and subsequently recategorizes the now full region as FILLED. The thick arrow from SUMMARIZING to READY (but not connected to any region in particular) represents the recategorization after a summarization pass targeting many regions, now with completely constructed summary sets and eligible for collection. Likewise the thick arrow from FILLED to SUMMARIZING represents the recategorization of the newly targeted regions at the start of a summarization pass.

4.4.2 The POPULAR category

The previous preliminary categorization has omitted one crucial detail: a region may be waved off from collection if its summary becomes too large (section 4.2.3) to ensure that no one region requires an excessive amount of collection effort. Supporting wave-off requires the introduction of a new category:

POPULAR regions that, the last time they were selected for summarization, were waved off before they could be collected.

The addition of the POPULAR category requires an extension to our state transition diagram, shown in figure 4.2. Besides the addition of the new category for POPULAR, figure 4.2 has three crucial features distinguishing it from figure 4.1:

1. a thin arrow joins S8, a region in SUMMARIZING, to P8, a dotted space in POPULAR, representing the potential wave-off of a region while it is being summarized,

2. a thin arrow joins R2, a region in READY, to P2, a dotted space in POPULAR, representing the potential wave-off of a region *after* it has

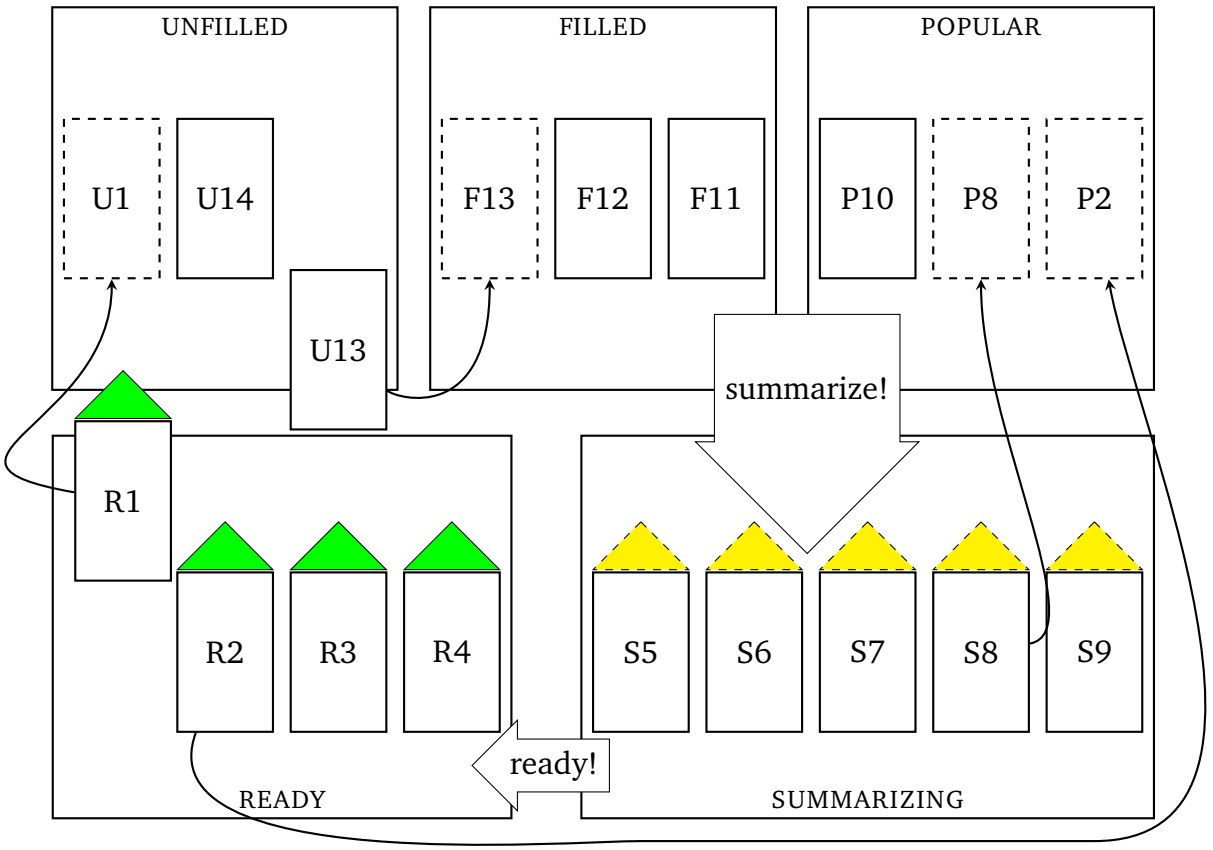


Figure 4.2: Region categorization with POPULAR

been completely summarized, and

3. the thick arrow joining FILLED to SUMMARIZING in figure 4.1 now has an origin that spans both FILLED and POPULAR, representing the potential reselection of a *popular* region to be summarized again.

The first difference mentioned above is a direct consequence of adopting the wave-off strategy. The second and third differences are more subtle; their necessity is not immediately obvious.

Wave-off of a region with a completely constructed summary set is necessary because mutator activity can make a region popular *after* it has been summarized. It is not possible to ensure in general that every READY region

is eventually collected without overly constraining the rate of mutator activity. Instead, I allow (a bounded percentage of) READY regions to become POPULAR; section 5.4.3 further discusses this issue.

Regions currently classified as POPULAR cannot generally remain uncollected; that is, we cannot assume that such regions will remain POPULAR for the remainder of the computation. Therefore POPULAR regions are generally candidates for summarization and subsequent collection. This detail is one of several characteristics distinguishing this collector's design from Sun's Garbage-First collector [18].[5]

4.4.3 High-level summarization algorithm

The preceding has provided a sketch of how on-demand summary set construction proceeds. Figures 4.3 and 4.4 show pseudo-code for a single summarization pass. The procedure SUMMARIZATIONPASS expects as parameters the current set of regions partitioning the heap and a number t counting how many regions to use as goal targets for this summarization pass. The value of t varies as a function of policy parameters and the number of regions; I defer discussion of t's definition to section 5.3.2.

SUMMARIZATIONPASS will spend most of its time in the nested loops in lines 7–12; these loops are mostly a simple traversal of the heap accumulating locations into summary sets as appropriate. The main points of interest are:

- Lines 1 and 8 of SUMMARIZATIONPASS keep track of which regions are scheduled for future scanning during this pass; this allows the write

[5] Section 7.3.1 introduces a stronger notion of popularity that would allow the system to avoid reselection of absurdly popular regions, at least until it has evidence that they are not likely to be waved off. In the general case, however, the collector cannot make such determinations sufficiently far ahead of time, and must instead optimistically pass them along to the summarizer.

```
SUMMARIZATIONPASS(regions, t)
    ▷ shared global state: will-be-summ-scanned, class, summaries,
    ▷                      last-completed-snapshot
 1  will-be-summ-scanned ← regions
 2  refine-basis ← last-completed-snapshot
 3  targets ← choose t regions from FILLED and POPULAR
 4  for r ∈ targets
 5      do class[r] ← SUMMARIZING
 6         assert summaries[r] = ∅
 7  for r ∈ regions
 8      do will-be-summ-scanned ← will-be-summ-scanned \ {r}
 9         for x ∈ objects-in(r)
10             do if not-long-dead? (x, refine-basis) and has-slots?(x)
11                then SUMMARIZEOBJECT(x, r)
12                     yield          ▷ allow control to shift to mutator
13  for r ∈ regions                   ▷ shift successful targets to READY
14      do if class[r] = SUMMARIZING
15         then class[r] ← READY
```

Figure 4.3: High-level code for summarization, part I

barrier to filter out cases that will be covered by the summarizer itself (line 6 of WRITEBARRIER-SUMM), and

- Line 10 of SUMMARIZATIONPASS filters out summarization of objects long known to be unreachable as well as objects known to contain no reference-holding slots (such as bytevectors).

The not-long-dead? function referenced in line 10 of SUMMARIZATION-PASS works by consulting a past snapshot of the state of the heap. Its speci-

```
SUMMARIZEOBJECT(x, r)
    ▷ shared global state: class, summaries
1   assert rgnof(x) = r
2   for l ∈ slots(x)
3       do v ← MEM[l]
4          if v tagged as reference
5             then r' ← rgnof(v)
6                  if r' ≠ r and class[r'] = SUMMARIZING
7                     then RECORDLOC(l, r')

RECORDLOC(l, r')
1   summaries[r'] ← summaries[r'] ∪ {l}
2   if |summaries[r']| exceeds its wave-off limit
3      then summaries[r'] ← ∅
4           class[r'] ← POPULAR
```

Figure 4.4: High-level code for summarization, part II

fication is simply

$$\text{not-long-dead?}(x, M) = x \in M.$$

Such snapshots are constructed by the marker, which is the topic of section 4.5. Before the initial marker run, *last-completed-snapshot* can be a trivial snapshot that classifies every object as live.

Figure 4.5 addresses the necessary cooperation between the summarizer and the mutator. The write-barrier for the regional collector is presented in increments; each piece of the write-barrier is presented with the component it cooperates with. The notation used in the definition,

$$\text{WRITEBARRIER-SUMM}(x\,[i]\ :=\ v),$$

```
WRITEBARRIER-SUMM( x[i] := v )
    ▷ shared global state: will-be-summ-scanned, class
1   if v tagged as reference and x is not in nursery
2      then r ← rgnof(x)
3           r' ← rgnof(v)
4           l ← location of x[i]
5           if r ≠ r' and class[r'] = SUMMARIZING
6              and r ∉ will-be-summ-scanned
7              then RECORDLOC(l, r')
8           elseif r ≠ r' and class[r'] = READY
9              then RECORDLOC(l, r')
```

Figure 4.5: High-level code for summarization portion of write-barrier

should be read as: "for every assignment statement of the form: $x[i] := v$, schedule[6] the following operations for eventual execution before the next collection."

4.5 Snapshot marking and refinement

The use of a reference-tracking structure such as a remembered set or points-into summary sets introduces a pitfall that most every incremental or generational collector suffers from: *floating garbage*, or *float*.

[6] This semantics of potentially delayed execution allows for the actual write-barrier of the mutator to be implemented by adding an entry to a log and batching together many invocations of RECORDLOC when the log is filled, rather than incurring the overhead of directly piggy-backing RECORDLOC onto potentially every assignment operation.

4.5.1 Imprecision implies floating garbage allowed

The implications in the remembered set and summarization invariants (pages 15 and 29) are unidirectional, capturing the meta-data's role as an *imprecise* record of objects with region-crossing references, as discussed in section 3.3.2.

In the particular case of floating garbage, the presence of b in the meta-data does *not* imply that b represents a live object B with a reference to an object A in another area of the heap, as the object B may no longer be reachable from the mutator. However, determining whether B is reachable requires processing the entire heap. Rather than incur that cost on every collection, generational collectors conservatively assume that any object reachable from the generational remembered set is live. Thus some objects that are not reachable from the mutator may still be copied during a collection; those objects are the float. The same problem arises in the regional collector: the summary sets may contain locations within objects that are no longer reachable by the mutator.

4.5.2 Local collections do not suffice

In general, the forwarder can remove *some* imprecision from the meta-data. In the regional collector, the summarizer process is imprecise and may add entries to summary sets that eventually will be determined to be unreachable by the forwarder. If the forwarder reclaims the heap storage of an object A in the region ρ, the forwarder can also remove all locations associated with A from *all* summary sets for both READY and SUMMARIZING regions. Removing locations associated with an unreachable object makes the summary sets more precise. However, such localized refinements of the summary sets generally will not suffice for two reasons: cycles and wave-off. The first is a problem common to many collectors, while the second is novel to the regional collector.

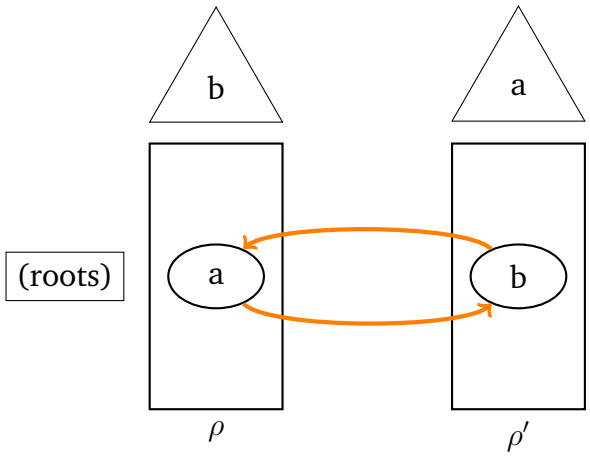


Figure 4.6: Floating garbage due to region-crossing cycles

A region-crossing cycle in the object graph, if left solely to the summarizer and forwarder to handle, could remain uncollected forever after becoming unreachable, due to the same problem suffered by memory management systems based on reference-counting.

As illustrated in figure 4.6, if objects A and B assigned to distinct regions ρ and ρ' mutually refer to each other, and if every time that ρ is collected its objects are forwarded to a target region different from the target when forwarding from ρ', then the locations in B that refer to A would keep A alive, and *vice versa*.

With luck, a single target region ρ'' might remain unfilled long enough that both objects A and B could end up co-located in ρ'', in which case the two objects would cease keeping each other afloat, but I do not assume such a fortunate event. Instead I ensure that region-crossing cycles of unreachable objects will eventually be identified by the separate marker coroutine. After the marker completes a snapshot of the heap in figure 4.6, B and A will be absent from the snapshot and thus excluded from the summary sets subsequently constructed. The objects A and B will be reclaimed when ρ and ρ' are next collected. Unreachable object cycles are the classic source of floating garbage in the system necessitating the use of the marker.

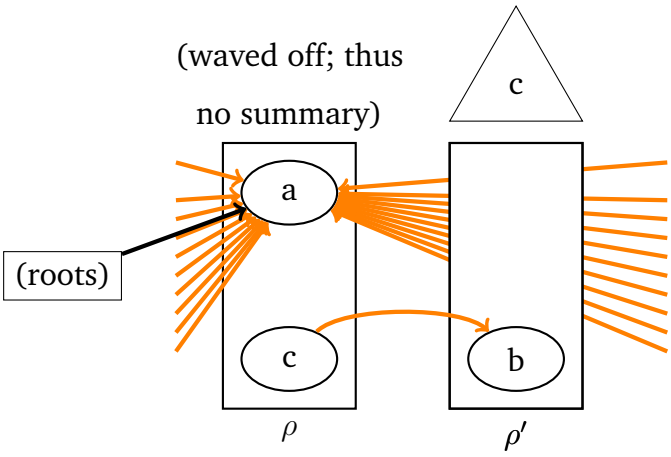


Figure 4.7: Floating garbage due to wave-off

The regional collector's novel source of float is due to wave-off of popular regions. Section 4.2.3 introduced the regional collector's solution to the problem of popularity: waving off collection of regions whose complete summary set would be too large to process during collection pause.

However, waving off whole regions (as opposed to isolating individual popular objects and treating them specially, as Sun's Garbage-First collector [18] does) has a subtle consequence. Unreachable objects within a waved-off region may have references to other unreachable objects residing in regions that are eligible for collection. A naïve summarization that scanned the heap without ever filtering out unreachable objects would continually misclassify all references coming out of waved-off regions as live, and build summaries that included such references.

Figure 4.7 illustrates this situation: The object a is live, resides in region ρ, and popular enough that ρ will be waved off from collection. Thus the unreachable object c that also resides in region ρ (but has a reference pointing into region ρ') will not be collected. If no action is taken to counter this, the summarization scan of ρ targeting ρ' will include c, and thus the storage of unreachable object b will not be reclaimed.

By a fortunate design coincidence, the same filtering of unreachable

objects used to identify cyclic garbage can be used to identify *waved-off* garbage. In this case, the point is not to reclaim the waved-off garbage itself directly, but rather to ensure such garbage structure will stop being *summarized*. Once waved-off garbage is filtered from summarization, unreachable structure it refers to in other regions can be reclaimed properly.

In the particular case illustrated in figure 4.7, the refinement introduced by the marker will ensure that c is marked as unreachable; thus c will be excluded from the summarization scan targeting ρ', and thus b will be reclaimed when the region ρ' is collected. Note that the storage of c itself may remain uncollected indefinitely – as long as the object a is popular, the region ρ will remain POPULAR, and the storage of c itself will not be reclaimed.[7]

4.5.3 Bounding floating garbage

If a memory management strategy is to be space-efficient, it cannot allow float to accumulate without bound.

A typical generational collector bounds the amount of float by performing periodic collections of the whole heap. During these whole-heap collections, the collector does not consult the remembered set, but instead determines each object's reachability in the object graph at the time of collection.

The regional collector never performs whole-heap collections. Instead it bounds the amount of float by bounding the imprecision of the remembered set, in the following manner. An incremental marking coroutine, initiated periodically, constructs a snapshot of the object graph at a particular point in time. Such construction is commonly known as a *snapshot-at-the-beginning* algorithm [30]. When the snapshot has been completely constructed, all objects are classified as either (1) unreachable or (2) reachable/unallocated at the time the snapshot construction was initiated.

[7] One could further extend the design of the regional collector to enable c to be collected and allow its storage within ρ to be reused. The point here is that it is possible to leave all of ρ (including c) uncollected and still have a scalable collector.

A snapshot, once completed, is used to *refine* the collector's meta-data, removing objects that have been proven to be unreachable at the time the snapshot was initiated. This refinement makes the meta-data more precise (though not maximally precise; just sufficiently precise to bound the amount of floating garbage). Refinement is the primary purpose of the marker component of the regional collector. (A secondary purpose of the marker is to measure the amount of reachable storage at the time of the snapshot, which affects heap-expansion policy; see section 5.3.3.) When refinement is complete, the snapshot is discarded.

Incremental marking is certainly not a novel technology [5, 12, 2, 30, 11, 7, 24, 20]. However, it is important to point out that using the snapshot as the basis for meta-data refinement differs from using the snapshot as the basis for *all* collector actions. Conclusions about snapshot-at-the-beginning from other contexts do not immediately apply to the regional collector's use of snapshots.

For example, consider the following statement from a standard garbage-collection textbook [22]:

> Snapshot-at-the-beginning algorithms are very conservative. No objects that become garbage in one garbage collection cycle can be reclaimed in that cycle: they must all wait until the next. Consequently, new objects acquired during a marking phase are effectively allocated black[8] even though the chance of a young object dying within a single cycle is high.

It is true that newly allocated objects are marked as live (or, more properly, "unallocated at time of snapshot") within a snapshot; however, in the regional collector this does *not* immediately imply that such objects are unreclaimable. Whether an object can be reclaimed in the regional collector is not solely tied to its state in the mark snapshot, but is rather dependent

[8] The term "allocated black" is jargon meaning that such objects are unreclaimable.

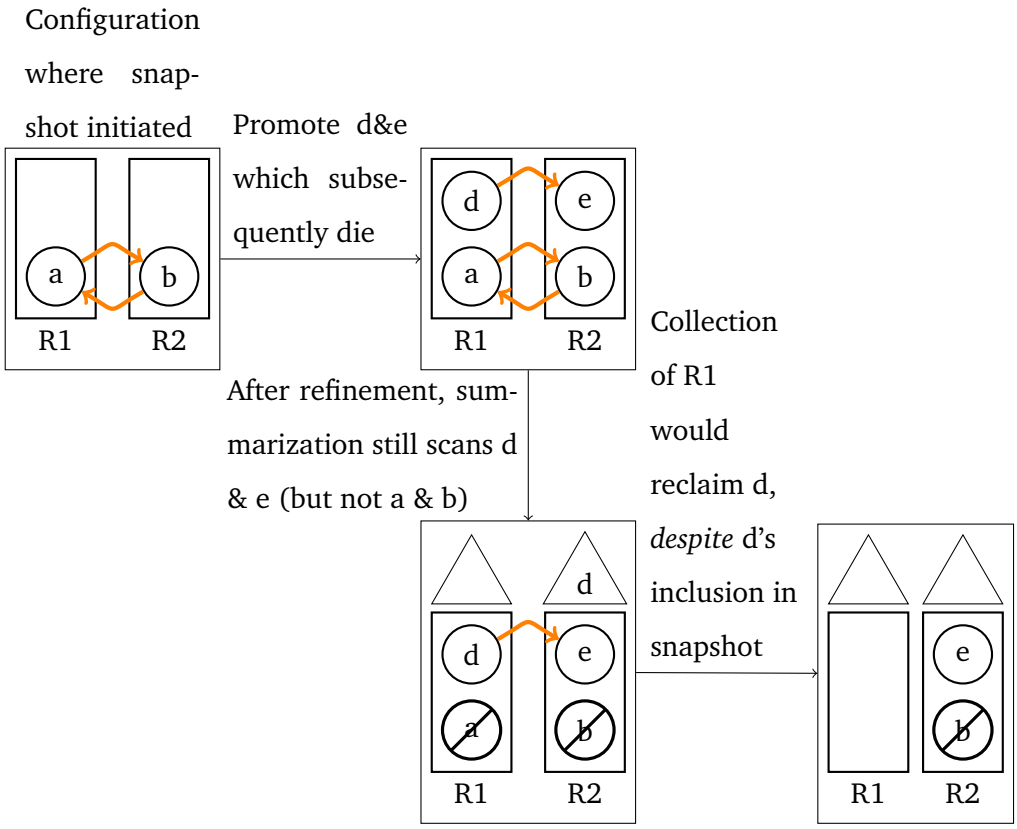


Figure 4.8: Snapshots dictate refinement, not reclamation

upon the snapshot, the partitioning of the object graph into regions, and the order that regions are considered for collection.

Figure 4.8 illustrates an instance of this phenomenon by showing a series of heap configurations as the mutator and collector each progress. In the figure, time progresses from left to right and then from top to bottom.

The initial configuration on the top left of the diagram has a garbage cycle formed by objects labeled a and b. For the example, assume that a snapshot is initiated at this initial configuration.

As the snapshot progresses, the mutator is allowed to make progress, so it might allocate the objects labeled d and e, as depicted in the top middle configuration. Since they were unallocated at the time of the snapshot, they are marked as live in the snapshot when they are promoted out of the mutator's nursery into the main heap area.

Assuming that d and e subsequently become unreachable, they will remain part of the snapshot. This demonstrates the above quoted objection to snapshot-at-the-beginning algorithms: these two unreachable objects are considered live in the snapshot. The lower-middle configuration depicts this: the objects a and b were dead at the time of the snapshot, and so are skipped during subsequent summarization scans; but d and e are live and thus included in the scan.

Finally, the transition between the lower-middle and lower-right configurations shows how the regional collector can be more aggressive than a usual snapshot-at-the-beginning algorithm: the evacuation of region R1 is able to reclaim the storage associated with d, because there are no references to d that were included in the summarization scan. The object e does not belong to R1 and so it would not be reclaimed during the collection of R1, but a subsequent collection of R2 would allow e to be reclaimed.

Note that the snapshot process did lead to the construction of an imprecise summary targeting region R2; while the object e is obviously reclaimable in the final lower-right configuration of figure 4.8, that is a consequence of the forwarder making the summary for R2 more precise as it collects R1, in the manner described in section 4.5.2. If the forwarder were to select R2 for collection before R1 (as it well might), then it would conclude from d's presence in R2's summary set that the object e must be copied; the object e would be floating garbage, and would remain ineligible for collection until after the next snapshot construction or after R1 is itself collected (whichever comes first).

4.5.4 High-level marking algorithm

Figure 4.9 describes how the incremental marking algorithm constructs a snapshot of the heap. This routine can only be initiated immediately after the end of a nursery evacuation. Nursery evacuations occur during both

```
MARKERBUILDSNAPSHOT(regions)
    ▷ shared global state: K_r (for r ∈ regions), K, T,
    ▷                      last-completed-snapshot
 1  for r ∈ regions
 2      do K_r ← [ ]              ▷ K_r is stack of references to objects in r
 3  K ← [ ]                       ▷ K is stack of references to stacks above
 4  T ← ∅                         ▷ T is snapshot state, a set of references
 5  for x ∈ ROOTREGISTERS
 6      do v ← REG[x]
 7          MARKERTRACEVALUE(v)
 8  while K ≠ [ ]
 9      do x ← pop(pop(K))
10          for l ∈ slots(x)
11              do v ← MEM[l]
12                  MARKERTRACEVALUE(v)
13          yield                 ▷ allow control to shift to mutator
    ▷ Trace completed; hand off snapshot as new basis for refinement.
14  last-completed-snapshot ← T

MARKERTRACEVALUE(v)
1  if isptr(v) and has-slots?(v) and v ∉ T
2    then r ← rgnof(v)
3         T ← T ∪ {v}
4         K_r ← push(v, K_r)
5         K ← push(K_r, K)
```

Figure 4.9: High-level code for snapshot marker

```
MARKERWRITEBARRIER( x[i] := v )
1  if x is not in nursery
2     then u ← x[i]            ▷ snapshot-at-beginning: save old value!
3          MARKERTRACEVALUE(u)
```

Figure 4.10: High-level code for snapshot marker portion of write-barrier

minor and major collections, so this is really just a matter of making the collector responsible for atomically initiating MARKERBUILDSNAPSHOT. This constraint is necessary to ensure the invariant that the snapshot state *never* includes objects that are currently in the nursery. This allows for an optimization of the mutator write-barrier, presented in figure 4.10.

The code for MARKERBUILDSNAPSHOT is mostly a standard marking algorithm to build up the globally shared snapshot state T. Tracing each object (MARKERTRACEVALUE) first checks if the object is already marked (in which case it is skipped), and otherwise marks and then pushes it onto the marking frontier of the graph traversal, scheduling the object for future processing by the marker.

One important point is that this frontier is represented as a stack-of-stacks K, rather than a single stack of object references. A single stack would not suffice, because the incremental marking algorithm must periodically yield to the mutator and collector, and the collector may forward objects before the snapshot has been completely constructed. Forwarding objects invalidates their old address on the mark frontier, so the mark frontier must be updated in response to object forwarding after each collection of a region. But in a pathological case, the mark frontier may contain every object on the heap; if the frontier were represented as a single stack of references, processing the entire stack would take too long to fit in a pause proportional to the fixed region size.

Instead of representing the mark frontier as a single stack of object references, the elements of K are references to stacks of object references K_r, where K_r is is a stack that solely holds references to objects in the region r. When the region r is collected, the frontier substructure K_r can be updated on its own, independently of the other stacks K_i on the mark frontier.

4.6 High-level forwarding algorithm

The remaining component of the regional collector is the copying collector itself, described in figures 4.11 and 4.12. The pseudo-code makes some assumptions about object representation; for example, all objects occupy at least two words, providing space to install forwarding pointers.

The most important modifications to the standard Cheney algorithm [14] are:

- only objects in the collected region rgn and the nursery are forwarded (lines 6, 10, and 19 of CHENEYEVACUATERGN); this is a standard modification for generational collectors,
- the summary for rgn is included in the root set (line 8 of CHENEYEVACUATERGN); this is analogous to including the remembered set in a generational collector, also a standard modification,
- the current mark frontier is included in the root set (line 3 of CHENEYEVACUATERGN),
- the current mark snapshot T is updated as objects are forwarded (line 8 of FORWARDRGN),
- RECORDLOC (figure 4.4) is invoked so that the summary sets reflect the changes to objects' addresses as they are forwarded (line 18 of CHENEYEVACUATERGN).

```
CHENEYEVACUATERGN(to, rgn, S)
    ▷ shared global state: K_r (for r ∈ regions), T
 1  scan ← to
 2  next ← to
 3  for l ∈ K_rgn
 4      do MEM[l] := FORWARDRGN(MEM[l], T)
 5  for r ∈ ROOTREGISTERS
 6      do if isptr(REG[r]) ∧ (rgnof(REG[r]) ∈ {rgn, nursery})
 7           then REG[r] := FORWARDRGN(REG[r], T)
 8  for l ∈ summaries[rgn]
 9      do                          ▷ l may no longer point into rgn
10         if isptr(MEM[l]) ∧ (rgnof(MEM[l]) ∈ {rgn, nursery})
11           then MEM[l] := FORWARDRGN(MEM[l], T)
12  while scan < next
13      do for l ∈ constituents(MEM[scan])
14             do v ← MEM[l]
15                if isptr(v)
16                  then rgn' ← rgnof(v)
17                       if class[rgn'] = SUMMARIZING
18                         then RECORDLOC(l, rgn')
19                       if rgn' ∈ {rgn, nursery}
20                         then MEM[l] := FORWARDRGN(v, T)
21         scan ← scan + objsize(MEM[scan])
22  for r ∈ regions        ▷ Real loop is more focused; see section 6.1.1.1.
23      do remove locations of rgn from summaries[r].
```

Figure 4.11: High-level code for region-modified Cheney copying collector

```
FORWARDRGN(l, T)
 1  if MEM[l] = FORWARDED
 2     then return MEM[l + 1]
 3     else  s ← objsize(MEM[l])
 4           t ← next
 5           for i ← 0 to s
 6                do MEM[t + i] := MEM[l + i]
 7           if l ∈ T
 8             then T ← T ∪ {t}
 9             else  assert t ∉ T
10           next ← next + s
11           MEM[l] := FORWARDED
12           MEM[l + 1] := t
13           return t
```

Figure 4.12: High-level code for regionally-modified object forwarding routine

In addition, one change that is *not* described in the above pseudo-code is that if an object at location l is forwarded to location t, and l is present in the remembered set, then t must replace l in the remembered set, as the old storage at l is about to be deallocated. This is a consequence of the regional remembered-set invariant and it is *not* standard in all generational collectors. Many generational collectors ensure that when an object is forwarded, all objects that it references end up in the same generation or an older one; in such systems, it suffices to just remove the old location l from the remembered set. The new location t will not be in the remembered set (unless of course the mutator introduces a new reference from the forwarded object to some newly allocated younger object). This change is not included in the above pseudo-code so that the reader will focus on the changes that are es-

sential to regional collection, as opposed to an artifact of a heuristic measure like the remembered set (further discussed in chapter 7).

The added invocations of RECORDLOC are an obviously necessary addition. It is also obvious that the snapshot state needs to be updated as objects are forwarded, so that partially explains the changes to update the snapshot state T and the inclusion of the mark frontier in the root set.

4.6.1 The mark stack must be a source of roots

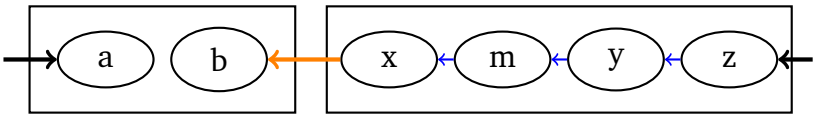
However, one question remains: why not simply update the mark frontier, and if an object on the frontier is not forwarded, then simply clear that entry in the frontier in some manner (in essence, treating the mark stack as a set of weak references). After all, the addition of the frontier to the root set may well introduce floating garbage, which is undesirable. So why do we need to adopt this harsh measure?

Figure 4.13 illustrates why the mark frontier must be included in the root set. It presents a heap structure and an event sequence where, if the mark frontier were a set of weak references, the system would become unsound.

The event sequence starts off with an chain of objects that crosses regions, and then proceeds as follows:

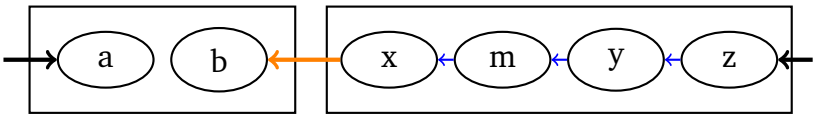
1. The marker initiates construction of a new snapshot. The marker gets as far as processing two of the involved objects (and scheduling a third (y) for future processing) before it yields to the mutator.

2. The mutator performs two actions: it adds a new reference from one of the marker-processed objects (a) to another object further along on the chain (x). Since the system is using a snapshot-at-the-beginning marker and the object a has already been processed, this reference from a to x will *not* be traced by the marker; instead it relies on the fact that x was reachable in the initial snapshot to ensure that x will eventually be marked.



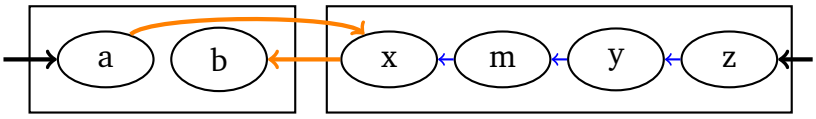
Rem.Set: { x }
Snapshot: { }
Frontier: { }

(marker initiates snapshot)



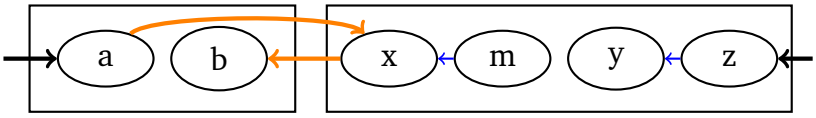
Rem.Set: { x }
Snapshot: { a, z }
Frontier: { y }

(mutator introduces a to x reference)



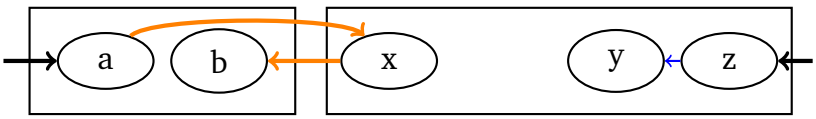
Rem.Set: { x }
Snapshot: { a, z }
Frontier: { y }

(mutator deletes y to m reference, making m unreachable)



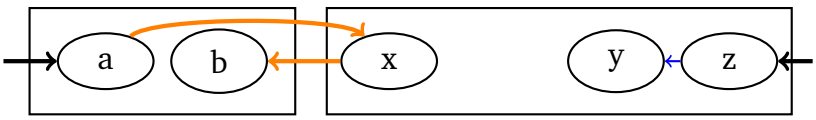
Rem.Set: { x }
Snapshot: { a, z }
Frontier: { y, m }

(collector collects region 2, reclaiming *weakly-held* m)



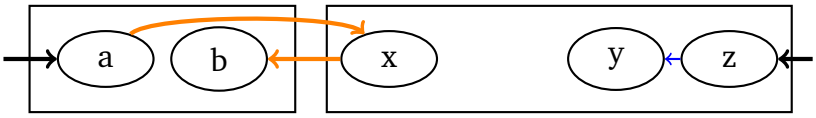
Rem.Set: { x }
Snapshot: { a, z }
Frontier: { y }

(marker finishes work; snapshot is “complete”)



Rem.Set: { x }
Snapshot: { a, z }
Frontier: { }

(refinement of the remembered set; reference to b is lost!)



Rem.Set: { }
Snapshot: { a, z }
Frontier: { }

Figure 4.13: Event sequence illustrating why mark stack must be in root set

3. Then the mutator breaks the link from y to m; note that the only reference to x present in the original snapshot was the link from m to x. Since this is a snapshot-at-the-beginning marker, the object m is added to the mark frontier, in order to ensure that the link from m to x is eventually processed.

4. Now the mutator yields to the collector, which happens to choose to evacuate the second region. This is where things go wrong. Since we are choosing to use the semantics where objects on the mark frontier are considered *weakly* referenced, the object m is reclaimed, and removed from the frontier structure. This removes the final arc to the object x, so it will never be marked.

5. The collector yields to the marker, which eventually completes its snapshot.

6. The summarizer sees that x is unmarked in the snapshot, and removes x from the remembered set. At this point, the heap structure violates the regional remembered set invariant; the object b may be incorrectly reclaimed as unreachable storage.

This counter-example illustrates why it is unsound to treat the mark frontier merely as a set of weak references to heap objects; it must be considered part of the root set. Note, however, that the mark *snapshot* is not part of the root set, as discussed in section 4.5.3.

Chapter 5

Regional collection policies

The three memory-management components in the regional collector are all coroutines that interoperate with the mutator. A crucial requirement of a scalable system is that if the operation of the whole system is observed for a reasonable[1] length of time, the mutator will make a reasonable amount of progress in that time. It would be unacceptable for a single component to hog the processor for an unbounded amount of time. It would also be unacceptable for the processor to spend its time solely shifting control between memory-management components and never give the mutator a reasonably sized timeslice.

This chapter describes how the memory-management coroutines are broken down into incremental operations; it also describes how their operations are scheduled. The scheduling is designed to ensure that, no matter what actions the mutator takes or how large the volume of live storage is,

- every memory-management component finishes its work by an appropriate deadline,

[1] Where "reasonable" here means both "reasonably short" (from the viewpoint of the client of the runtime), and "reasonably long" (from the viewpoint of the runtime implementor and provider). The client and provider must find a point where their different interpretations intersect.

- the space usage of the system never grows beyond a fixed multiple of the live storage plus a constant, and

- for some window size, the minimum mutator utilization has a non-zero lower bound.

5.1 Coroutine control shifts

The regional collector responds to two kinds of mutator actions: allocation of objects in the heap (adding new nodes to the object-reference graph), and imperative assignment to fields of live objects (changing the aforementioned graph structure). Object allocation, as observed by the regional collector, may take the form of promotions out of the nursery and into the larger heap; this can be readily modeled as a series of allocations all batched together, simplifying the abstract model.

Non-allocation and non-assignment actions of the mutator, such as arithmetic operations or reads of fields in objects, are not monitored by the regional collector.

Some of the collector's operations (namely summarization and snapshot construction) can be performed concurrently with the mutator (see Chapter 8). As an alternative to concurrent execution, these same operations can be serially interleaved with the actions of the mutator at a fine grain. I model such activity as overhead added to the summed cost of all of the actions performed by the mutator, avoiding the burden of specifying the relatively uninteresting control transfers between the mutator and those memory-management components.

It is relatively straight forward to break down these iterative tasks into interruptible sequences of short atomic actions; therefore the important thing is not how control transfer is initiated nor at what frequency, but rather what schedule the components must adhere to in order to ensure that the system

as a whole remains scalable.

5.1.1 Work-based scheduling and accounting

The collector's actions are interleaved with those of the mutator. Any action taken by the mutator that does not affect the collector, such as an arithmetic computation, could be viewed as an extra opportunity for the collector to focus on making forward progress (as opposed to merely catching up with the mutator's view of the heap structure). A truly optimal schedule would distribute collection work evenly with respect to mutator activity of any kind, *including* activity unrelated to the heap and object graph.

While the above observation matters when optimizing a collector for a particular application, it is a distraction when confronting worst-case scenarios with no *a priori* knowledge of the mutator. A hypothetical worst-case mutator spends all of its time attempting to foil the collector's attempts to maintain its space bounds and time schedule. Such a mutator will focus solely on unpredictable actions that the collector cannot ignore; in the case of a collector with a write-barrier but no read-barrier, this means allocations and assignments. Such allocations and assignments are the mutator *work* that the collector must respond to.

In the worst case, practically all mutator actions will be allocations and assignments requiring a response from the collector. Any scheduling method providing hard guarantees without *a priori* knowledge of the mutator will yield the same result as some work-based schedule, because wall-clock time, processor cycles, and other bases for scheduling all are converted into mutator work when a demon holding an oracle controls the mutator. Therefore we simplify our design effort and presentation by focusing on work-based scheduling policies that guarantee collector scalability.

5.1.2 Mutator and Collector Activity

The collector's responses to mutator work take the form of

allocation satisfying mutator requests for memory by promoting objects out of the mutator's nursery into some unfilled region,

evacuation reclaiming unreachable space by copying reachable objects from a filled READY region unto some unfilled region, updating fields so that the object graph is preserved,

summarization preparing for a future collection of a region r by scanning heap substructure, recording the fields of objects that hold references pointing into r in an (uncompleted) summary for r,

marking preparing for a future refinement of collector meta-data by progressively tracing an old snapshot of the heap state, and

write barrier updating collector meta-data to reflect that an assignment `a[i] := b` occurred.

The allocation and evacuation activities are collectively referred to as "object migration" or simply "migration." "Evacuation" is also referred to as "copying collection" or simply "collection" when unambiguous.

5.2 Policies

There are three main policies for scheduling collection activity, each relating to the three memory-management components:

- how often do regions need to be evacuated,
- how many summaries does a summarization pass attempt to construct and at what rate does it proceed, and

- how often must a new snapshot be constructed via tracing the heap (and how much latency can there be from initiation of the snapshot to the completion of its tracing).

All three policies refer to a notion of time ("how often," "at what rate") but as explained in section 5.1.1, I focus on work-based scheduling. "Time" is measured in units of mutator work: the number of allocations and the number of assignments. With this notion of time, these three policies imply a fourth policy: how many assignments can the mutator perform and how much storage can it allocate before the collector will forcibly interrupt in order to maintain its own schedule, and how long can such collector interruptions last. This fourth derivative policy is what determines the utilization of the mutator.

The three policies are related in other ways beyond their shared connection to the mutator's actions. For example, a region can be evacuated only if it has a usable summary. This means that summarization must proceed frequently enough to support the evacuation rate.

5.3 Policy parameters

The scheduling policies of the regional collector are defined in terms of abstract numeric parameters. These parameters are *not* intended to be tailored to particular mutators, but rather to particular runtime implementations of the regional collector. For example, one implementation might employ a heavily optimized summary set representation and could afford to construct a relatively large collection of summaries simultaneously. I refer to these as *runtime-selected parameters*, or just *parameters*. The parameters are mutually constrained; not all values for one parameter are necessarily compatible with all possible values for the others.[2]

[2]Note that a parameter's range may be constrained only by other parameters, not by dynamically derived variables; otherwise selecting an appropriate value would require *a priori*

There are also variables that the policies refer to that are not under the runtime's control, but rather are derived from the dynamic state of the system. I refer to these as *dynamically derived variables*, or just *variables*.

5.3.1 Global parameters and variables

The variable N denotes the number of word-sized *locations* allocated to hold the heap (but not the collector meta-data). N is trivially bounded from below by the current volume of live storage in the heap. The N words of the heap are partitioned into disjoint regions. The parameter R is the maximum size of any region, measured in words.

The particular number of bits per word is ignored by the policies; we assume that it is some constant large enough to represent a single tagged reference to some object on the heap.

I often refer to the current number of regions as N/R (which is a bit of a pun on N and R, as the number of regions is more accurately described by $\lceil \frac{N}{R} \rceil$, the form I shall use when the difference matters in algebraic manipulations).

5.3.2 Summarization parameters

There are four parameters governing the summarization routine: F_1, F_2, F_3, and S. The parameters S, F_1 and F_2 are reals greater than 1, and F_3 is a positive integer.

The three parameters F_1, F_2, and F_3 control the pace of summarization. The intuition behind the three parameters is roughly as follows: F_1 controls how many new summaries the system attempts to build on each scan of the heap. F_1 and F_2 together control how many summaries need to be completely constructed and kept up-to-date. Finally, F_3 controls how many

knowledge.

times the summarizer is allowed to start a fresh scan in order to reach the required number of summaries.

The *summarizing wave-off parameter*, S, bounds summary set size. If adding a new entry to a summary set for a SUMMARIZING region would cause the set to exceed SR entries, the collector discards the summary set and recategorizes its associated region as POPULAR.

A *summarization pass* of the regions $\{r_{i_1}, \ldots, r_{i_t}\}$ is a traversal of the heap structure to construct summary sets for t target regions. On each summarization pass, $1/F_1$ of the N/R regions are targeted for summarization; thus $t = \frac{\lceil N/R \rceil}{F_1}$. A *summarization cycle* is a sequence of summarization passes, where each pass targets a new set of regions. The cycle iteratively performs new passes until $1/(F_1F_2)$ of the N/R regions have usable summaries. The scheduling policy is responsible for ensuring that every summarization cycle requires *at most* F_3 summarization passes. (Note that the expectation is that most summarization cycles will require only *one* pass.)

Not every combination of F_i and S is valid; Sections 5.4.2 and 5.4.3 formally describes the constraints on their configuration.

The parameter configurations that perform best tend to be those that approach the limits of the constraints, coming as close as possible to violating the mathematical inequalities of Sections 5.4.2 and 5.4.3 without crossing over the line.

The proofs in this chapter involve formulas that can be used to derive the worst-case bounds on pause time, space usage, and minimum mutator activity, at least in the abstract: that is, up to constants of proportionality that are mainly determined by the hardware. For example, if one sets $1/F_1$ to a small fraction like $1/100$ (perhaps in an attempt to reduce memory usage by lowering the number of summaries that are constructed at a time), then intuitively each summarization will scan all of the regions in order to prepare at most one percent of the regions for collection, which is a poor space/time trade-off.

We used those formulas to find candidate configurations and found three that perform reasonably well. These three parameter configurations are described in Section 9.2.

5.3.3 Collection parameters

The parameters L_{hard} and L_{soft}, where $L_{hard} \geq L_{soft} > 1$, are the *absolute* and *desired inverse load factors*, respectively. The collection policies are obligated to ensure that N never grows beyond the product of L_{hard} and the volume of live storage. The second parameter, L_{soft}, describes a soft target that the runtime may violate (but attempts to satisfy).

The *ready wave-off parameter* $S' \geq S$ bounds how large the summary set for a READY region is allowed to grow. The parameter S' is similar to S: a READY region will be reclassified as POPULAR if mutator activity would cause its summary to exceed $S'R$ words. The (potentially) higher value of S' reflects that a fully constructed summary set is the result of significant summarization effort and it may warrant a higher threshold before abandoning collection of its corresponding region.

The runtime-selected parameter c controls the amount of mutator activity allowed by the collector relative to the current heap size; it is further discussed in sections 5.4.2.

5.3.4 Snapshot parameters

The main parameter for the marker is the rate at which it should trace the heap relative to the degree of mutator activity.

For these proofs, I assume that the rate of tracing matches the rate of allocation. A less aggressive rate will suffice in some cases, but in general a worst-case mutator will force the rates to match.

Now I derive properties of the regional collector, as functions of the runtime-selected parameters and dynamically derived variables defined above.

5.4 Popularity

The evacuation of any region r requires the construction of a summary set holding all locations that refer to objects in r. If such a set would be too large to process within the fixed pause time bound of the regional collector, then r is considered too popular to evacuate.

For such a strategy to make any progress at all, one must ensure a number of properties. Most obviously, it would be disastrous if the system could reach a state where every region were considered popular; choosing an appropriate value for S avoids this. Second, one must ensure that the summarizer will supply sufficiently many READY regions. Third, one must ensure that the mutator will not be able to invalidate the forwarder's schedule for reclaiming space from the READY regions!

Suitable policies described in this section ensure that all of the components can progress; these policies have associated formulae that capture the trade-offs involved between pause times, space usage, and mutator utilization.

5.4.1 Limiting popularity

I start off by deriving some simple results for the heap structure at any particular instant in time. I then generalize these derivations by accounting for modifications to the heap structure and associated meta-data over time.

Definition 2. *For any region r, the* true summary set *of r is the set of currently allocated locations in other regions $r' \neq r$ that hold references to objects in r.*

Definition 3. *A region is* truly popular *if the size of its true summary set meets or exceeds SR.*

Lemma 4. *The fraction of regions that are truly popular is $1/S$ or less.*

Proof. Let x be the fraction of the N/R regions that are truly popular. There can be at most N locations in a heap of N words. Every location holds at most one reference (to one object). Every truly popular region has at least SR distinct locations referring to it alone. Therefore,

$$N \geq x \left\lceil \frac{N}{R} \right\rceil SR \geq x \frac{N}{R} SR = xNS.$$

Thus $x \leq 1/S$. □

The regional collector does not guarantee that it constructs true summary sets. The summarizer builds imprecise approximations to the true summary sets by incrementally scanning the heap during a summarization cycle and incorporating changes as the mutator imperatively modifies the object graph.

Definition 5. *The* constructed summary set *of a region r is the meta-data structure built up by the summarization coroutine approximating the true summary set of r.*

The constructed summary set is an approximation of the true summary set, reflecting a series of closely related heaps rather than a heap at any particular instant. To account for this, I introduce a running total,

$$C = C_{al} + C_{as},$$

which counts the number of words allocated (C_{al}) and assigned (C_{as}) over the course of a summarization cycle. This allows the derivation of bounds with respect to C, accounting for imprecision that the mutator could have possibly introduced to the constructed summary sets.

Invariant 6. *If the summarization of a region r would require exceeding the limit of SR entries in the summary set of r, $class[r]$ is set to* POPULAR. *Every* POPULAR *region was at some point associated with a constructed summary set containing at least SR entries.*

Lemma 7. *If the number of words allocated and assigned performed during a summarization cycle is bounded by C, then the fraction of the N/R regions that could be classified as* POPULAR *during the summarization cycle is no greater than $(1 + C/N)/S$.*

Proof. Let x be the fraction of the N/R regions that would be classified as POPULAR if they were included in the summarization cycle. At most $N + C$ locations contribute to the group of constructed summary sets over the whole summarization cycle. Each of the POPULAR regions had at least SR distinct locations referring to it at some point during the summarization cycle.

$$N + C \geq x \left\lceil \frac{N}{R} \right\rceil SR \geq x \frac{N}{R} SR = xNS.$$

Therefore $x \leq \frac{1}{S}(1 + \frac{C}{N})$. □

Note that to approach the worst-case upper bounds implied by lemmas 4 and 7, the mutator would have to ensure that the references into the regions are precisely distributed so that they do not exceed SR by very much. That is, every reference into a region beyond the first SR such references is a reference that could have been put towards making another region popular; every such reference is wasted from the perspective of a demonic mutator.

I anticipate that real-world applications will not tend to distribute references in this manner; the size of the true summary set of any POPULAR region will tend to exceed SR (probably by a significant margin; see also section 7.3.2). If this prediction holds, the fraction of the N/R classified as POPULAR during a summarization cycle will not tend to approach these worst-case bounds.

It is clear from lemma 7 that the running total C must be bounded relative to N to ensure that the fraction of popular regions does not grow too large. In the next section, we will revisit the quantity C and introduce tighter constraints on the ratio C/N.

5.4.2 Ensuring summarization succeeds

Lemma 7's bound on the maximum number of regions classified as POPULAR is not enough on its own to ensure that the collector will always have regions with usable summary sets available to evacuate; just because non-popular regions exist does not imply that the summarization coroutine selected them as summarization targets.

In this section, we derive the constraints necessary to maintain the following invariant:

Invariant 8. *At the start of any one summarization cycle at least $1/(F_1F_2)$ of the regions will have usable summary sets.*

To maintain this invariant, the collector's parameters S, F_1, F_2 and F_3 must be appropriately selected and the running count C must be appropriately bounded.

Lemma 9. *Suppose S, F_1, and F_2 are greater than 1, F_3 is a positive integer, and C, the mutator activity during a summarization cycle of at most F_3 passes, is at most the product cN where*

$$0 < c \leq \frac{F_2F_3 - 1}{F_1F_2}S - \frac{S}{\lceil \frac{N}{R} \rceil} - 1.$$

Then F_3 passes suffice to ensure $1/(F_1F_2)$ of the summaries are usable.

Proof. Each summarization pass will attempt to build summaries for $1/F_1$ of the regions; therefore F_3 passes during the cycle will target a total of F_3/F_1 of the regions. By lemma 7, at most $1/S(1 + C/N)$ of the regions can be classified as POPULAR during the summarization cycle. Therefore the F_3 passes over the summarization cycle will yield at least u usable regions,

where

$$
\begin{aligned}
u &= \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left\lfloor\frac{1}{S}\left(1+\frac{C}{N}\right)\left\lceil\frac{N}{R}\right\rceil\right\rfloor \\
&\geq \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left\lfloor\frac{1}{S}\left(1+c\right)\left\lceil\frac{N}{R}\right\rceil\right\rfloor \\
&\geq \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left\lfloor\frac{1}{S}\left(1+\left(\frac{F_2F_3-1}{F_1F_2}S - \frac{S}{\lceil\frac{N}{R}\rceil}-1\right)\right)\left\lceil\frac{N}{R}\right\rceil\right\rfloor \\
&= \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left\lfloor\left(\frac{F_2F_3-1}{F_1F_2} - \frac{1}{\lceil\frac{N}{R}\rceil}\right)\left\lceil\frac{N}{R}\right\rceil\right\rfloor \\
&= \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left\lfloor\frac{F_2F_3-1}{F_1F_2}\left\lceil\frac{N}{R}\right\rceil - 1\right\rfloor \\
&\geq \frac{F_3}{F_1}\left\lceil\frac{N}{R}\right\rceil - \left(\frac{F_2F_3-1}{F_1F_2}\left\lceil\frac{N}{R}\right\rceil - 1\right) \\
&= \frac{1}{F_1F_2}\left\lceil\frac{N}{R}\right\rceil + 1 \\
&\geq \left\lceil\frac{1}{F_1F_2}\left\lceil\frac{N}{R}\right\rceil\right\rceil
\end{aligned}
$$

Therefore F_3 summarization passes will ensure that at least $1/(F_1F_2)$ of the targeted regions will have usable summary sets. □

Note that as the heap grows large relative to the region size, the $S/\lceil\frac{N}{R}\rceil$ term in c's upper bound becomes insignificant. Note also that this bound on c does not need to be enforced when the total number of regions $\lceil\frac{N}{R}\rceil$ is less than S, because when $\lceil\frac{N}{R}\rceil < S$, it is impossible for any regions to become popular, and therefore no summarization wave-off can occur.

5.4.3 Ensuring ready regions suffice

Unfortunately, our work does not end with the proof of lemma 9. The assurance provided by invariant 8 (and proven by lemma 9) that each summarization cycle will end with a known fraction of READY regions ready to be used as bases for collection during the next summarization cycle is necessary, but will not suffice on its own.

The remaining difficulty is that regions can still be waved-off as too popular to collect after they have already been declared READY. If this is allowed to happen too often, the forwarder will run out of READY regions before the summarizer has finished even one summarization pass, and the system will fail to satisfy allocation requests!

To ensure that the work of the summarizer can be scheduled and distributed across the effort of the forwarder (and thus distributed across the work of the mutator), one must ensure that this scenario cannot arise.

Each summarization cycle ends with at least $1/(F_1F_2)$ READY regions with completely constructed summary sets; therefore it is reasonable to assume that the next summarization cycle starts with those same constraints in place. Furthermore, we can assume that the completely constructed summary sets have a total size of D, where D is known to the collector and $D \leq N + cN$ (and will often be significantly smaller).

I can guarantee the bound $D \leq N + cN$ because (1) at most N words contributed to the true summary sets at the start of the last summarization cycle, and (2) over the course of the cycle, simultaneous mutator activity contributed at most $C = cN$ words to the summary sets being built. Moreover, at the start of a summarization cycle, the collector knows the current value of D, as the summary sets for the ready regions are completely present. (Contrast this against the not-yet-built summary sets for the regions being summarized during the summarization cycle; the collector has no *a priori* knowledge of their size, and therefore our calculations in section 5.4.2 assume the worst.)

Finally, as mentioned in section 5.3.3, a completely constructed summary set represents a significant investment of effort by the summarizer; READY regions should not be waved-off without good reason; therefore the summary sets associated with READY regions use a different wave-off parameter, S', as the basis for determining if they should be recategorized as POPULAR.

Lemma 10. *Let D be the total size of the completely constructed summary sets of the* READY *regions at the start of a summarization cycle. If mutator activity is limited to $C = cN$ actions during the cycle then the collector can assume that at least b* READY *regions will be available for evacuation over the entirety of the summarization cycle, where*

$$b = \left(\frac{1}{F_1 F_2} - \frac{D}{NS'} - \frac{c}{S'}\right)\frac{N}{R}.$$

Proof. At the start of the summarization cycle the completely constructed summary sets of the READY regions have a total of D entries. Over the course of the summarization cycle at most cN new entries could be added to these summary sets, for a total of at most $D + cN$ entries in these summary sets.

Let v be the maximum number of READY regions waved off as popular during a summarization cycle. A READY region only becomes POPULAR when its summary set grows to contain $S'R$ entries. Therefore we have the constraint

$$vS'R \leq D + cN$$

Let b' be the minimum number of regions that can safely be assumed available during the entirety of the summarization cycle. The summarization cycle starts with at least $1/(F_1F_2)$ of regions classified as READY to be evacuated. At most v of those could be waved-off as popular during the cycle. Therefore,

$$\begin{aligned}
b' &= \frac{1}{F_1 F_2}\left\lceil\frac{N}{R}\right\rceil - v \\
&\geq \frac{1}{F_1 F_2}\left\lceil\frac{N}{R}\right\rceil - \frac{D + cN}{S'R} \\
&\geq \frac{1}{F_1 F_2}\frac{N}{R} - \frac{D + cN}{S'R} \\
&= \left(\frac{1}{F_1 F_2} - \frac{D}{NS'} - \frac{c}{S'}\right)\frac{N}{R} \\
&= b
\end{aligned}$$

Thus $b' \geq b$, as desired. □

Lemma 10 is useful only if $b > 0$; otherwise the number of available READY regions for evacuation during a summarization cycle has no lower bound of use. In other words, if the collector does nothing to confront a demonic mutator's increasing the size of the completely constructed summary sets of the READY regions, then the collector must ensure

$$\frac{1}{F_1 F_2} - \frac{D}{NS'} - \frac{c}{S'} > 0$$

or equivalently,

$$c < \frac{S'}{F_1 F_2} - \frac{D}{N}.$$

As mentioned above, the collector's policies can trivially ensure that $D \leq N + cN$; if D actually approaches that limit, the collector will encounter a worst case scenario of enforcing the constraint:

$$c < \frac{S'}{F_1 F_2} - \frac{N + cN}{N} = \frac{S'}{F_1 F_2} - 1 - c$$

or equivalently

$$c < \frac{1}{2}\left(\frac{S'}{F_1 F_2} - 1\right)$$

This is an additional constraint on the value taken by the runtime-selected parameter c; it is orthogonal to the constraint documented in lemma 9.

As mentioned above, the collector tracks the sizes of the summary sets of the READY regions. If

- the mutator activity during the cycle is not increasing their size, or
- if D is sufficiently small at the outset of the summarization cycle

then this second constraint on mutator activity is *irrelevant* to the collector and can be ignored.

In a nutshell, the bound of lemma 9 is a matter of *preparing* for the worst case ahead of time, while the mutator activity bound implied by lemma 10 is more a matter of *responding* to a series of worst-case events after they have already happened.

5.4.4 Bounding waved-off regions

Lemma 7 bounds the number of regions that may be waved off during summarization, but this does not immediately answer the question of whether that bound will scale as summarization cycles are iterated. To tie up the popularity story, I answer the question of how many regions could be considered popular during the entirety of a full collection cycle.

Lemma 11. *The fraction of regions left waved-off (and thus uncollected) during a full collection cycle is at most* $1 - 1/(F_2F_3)$.

Proof. In general, a single summarization cycle is divided into x summarization passes, where $1 \leq x \leq F_3$ (as F_3 is the upper bound on summarization restarts). A cycle of x passes attempts construction of summary sets targeting x/F_1 regions in total, because $1/F_1$ of the regions are targeted for summarization in a single pass.

Assuming the beginnings and ends of full collection cycles are aligned with those of summarization cycles, a full collection cycle of the whole heap will contain n such summarization cycles, where the ith cycle is divided into x_i passes. Since the ith summarization cycle attempts summary construction for x_i/F_1 of the regions, we have the constraint

$$\sum_{i=1}^{n} \frac{x_i}{F_1} = 1. \tag{5.1}$$

Furthermore, each summarization cycle constructs usable summaries for at least $1/(F_1F_2)$ of the regions. Therefore over the course of a full collection cycle, at most

$$\begin{aligned} \sum_{i=1}^{n} \left(\frac{x_i}{F_1} - \frac{1}{F_1F_2} \right) &= \sum_{i=1}^{n} \frac{x_i}{F_1} - \sum_{i=1}^{n} \frac{1}{F_1F_2} \\ &= 1 - \frac{n}{F_1F_2} \end{aligned} \tag{5.2}$$

of the regions are waved off from summarization and subsequently collection.[3]

The bound (5.2) is clearly maximized when the number of summarization cycles, n, is minimal. This occurs when the pass counts x_i in the constraint (5.1) are maximal: for all i, $x_i = F_3$. Therefore to minimize n we must have:

$$1 = \sum_{i=1}^{n} \frac{x_i}{F_1} = n\frac{F_3}{F_1}, \tag{5.3}$$

Thus the minimal $n = F_1/F_3$, allowing the revision of the upper bound (5.2) to:

$$1 - \frac{n}{F_1 F_2} = 1 - \frac{1}{F_2 F_3} \tag{5.4}$$

Thus $1 - 1/(F_2F_3)$ bounds the fraction of regions waved off during a full collection cycle. □

A natural question when first encountering lemma 11 is why it makes no reference to the wave-off factor S, as that is what determines the threshold between popular and non-popular regions. The answer is the system must ensure the preconditions of lemma 9, which will require that

$$\frac{F_2F_3 - 1}{F_1F_2}S - 1 > 0$$

and thus the value chosen for S constrains the range of valid choices for the parameters F_2 and F_3.

5.5 Setting the Allocation Rate

Section 5.4 addressed the rate of summarization relative to the rate of general mutator activity. It now remains to establish at what rate the forwarder

[3] This does not directly address wave-off of READY regions. Note that any effort the mutator spends waving off READY regions is effort that was *not* put towards waving off SUMMARIZING regions. Since the threshold for wave-off of READY regions, $S'R$, is at least as large as the threshold for wave-off of SUMMARIZING regions, the bound (5.2) is conserved.

should evacuate the pool of READY regions, relative to the rate of allocation activity by the mutator. From another perspective, this is the same as asking how much the collector needs to constrain the mutator's rate of allocation. I refer to this as the *allocation rate policy*.

There are two main properties that I will require of the allocation rate policy. Both are defined in terms of P_{old}, the peak live storage. The two properties are: (1) the total heap size N cannot exceed $L_{hard}P_{old}$, and (2) A, the allocation during any one full cycle, is proportional to the peak storage; that is, $A = \Theta(P_{old})$.

Isolating these two properties and proving the scalability theorem in terms of them modularizes the design, allowing for experimentation with new policies that need only preserve the above properties to remain scalable.

One detail that matters here is that we are deliberately requiring that $A = \Theta(P_{old})$, which is a stronger condition than $A = O(P_{old})$. That is, the promotion rate must not only be bounded from above, but also from below. The reason for this is that if we do not have $A = \Omega(P_{old})$, then we cannot subdivide the mutator actions into sufficiently large chunks in between collection pauses to get the required summarization and marking work done to support collection within a bounded pause time.

We express the policy in terms of P_{old} (which varies with the application behavior), and the inverse load factors L_{hard} and L_{soft} (which are fixed runtime choices). We also include the parameter k, a fixed upper bound on the worst-case fraction of non-empty regions that go uncollected during a full cycle (e.g. due to popularity). This is also fixed by other runtime parameters; lemma 11 proves that $k \leq 1 - 1/(F_2F_3)$.

The current prototype uses the policy

$$A = \min\left(\frac{1}{2}((1-k)L_{hard} - 1)P_{old}, (L_{soft} - 1)P_{old}\right)$$

which clearly is $\Theta(P_{old})$.

I have also occasionally experimented with a potential alternative policy:

$$A = \min\left(\frac{1}{2}((1-k)L_{hard}-1)P_{old}, (L_{soft}-1)N\right)$$

This causes the heap to grow more quickly in response to a sudden increase in the volume of storage, (assuming that L_{soft} is significantly less than L_{hard}), without the violating the hard space bound. It also means that the system will be slower to ratchet back the heap usage when the growth was premature. However, from a theoretical point of view, this policy is problematic because it does *not* ensure that $A = \Omega(P_{old})$, and so it breaks the rules we stated above. It is not hard to imagine fixes for this (e.g., incorporate a third component that calculates some other function in $\Theta(P_{old})$).

5.6 What was all this for again?

This chapter laid the ground-work for our end-goal: proving that the regional collector is actually scalable.

The regional collector uses the following properties to support its scalability:

- If the summarization of a region r would require exceeding the limit of SR entries in the summary set of r, $class[r]$ is set to POPULAR.

- At the start of any one summarization cycle at least $1/(F_1F_2)$ of the regions will have usable summary sets, and each summarization pass (of at most F_3 such passes) attempts to build summaries for $1/F_1$ of the regions.

- During a summarization cycle of at most F_3 passes, the mutator activity C (which includes both allocation and assignment) is at most the product cN where $0 < c \leq \frac{F_2F_3-1}{F_1F_2}S - \frac{S}{\lceil\frac{N}{R}\rceil} - 1$.

- Allocation during any one full cycle is proportional to peak storage: $A = \Theta(P_{old})$. I currently enforce this via the policy:

$$A = \min\left(\frac{1}{2}((1-k)L_{hard} - 1)P_{old}, (L_{soft} - 1)P_{old}\right).$$

The next chapter will be using the lemmas proven and properties described here in order to prove the scalability theorem.

Chapter 6

Proving scalability

Chapter 1 stated the first two of the goals for the design of the regional collector were to provide worst-case time and space bounds in order to ensure that the collector would remain scalable. These two goals correspond to the statement of theorem 1, which provides three proof obligations that must be met for some *mutator-independent* choice of positive constants c_0, c_1, c_2, and c_3:

1. c_0 is larger than the worst-case time between mutator actions.

2. Within every interval of time longer than $3c_0$, the minimum mutator utilization is greater than c_1.

3. The total memory used by the mutator and collector is less than $c_2P + c_3$, where P is the peak volume of reachable objects.

6.1 Bounding Memory Usage

I first address the last obligation: that the regional collector's memory usage is $O(P)$. This is easy to see. Section 5.5 specifies that the allocation rate maintains a bound on total heap size N of $L_{hard}P_{old} = O(P)$.

Assuming the addresses of heap objects are assigned reasonably close to one another,[1] the incremental snapshot structure is represented by a $O(N)$ bitmap (for the snapshot itself) and a stack of mark stacks (for the frontier of the snapshot under construction). The stack of mark stacks may have duplicate entries, since the snapshot traversal can push two references to the mark stack for one region. The total number of entries on the stack of mark stacks is bounded by the number of objects in the heap.[2] There are at most N/R mark stacks, one for each region (and each of size $O(R)$). So the total snapshot structure occupies

$$O(N) + O(N) + N/R \cdot O(R) = O(N)$$

space.

Likewise, the remembered set can be represented by any reasonable structure for holding a set of addresses; over the course of this project, I started with a fixed-size hashtable for each region, and then later migrated to a sparse representation: a shallow tree mapping a prefix of each address to a bitmap or a distinguished `null` value. These are both $O(N)$ representations; the migration was not motivated by theoretical concerns, but rather real-world performance problems with the large remembered sets that tended to arise with the regional collector.

The only component left that could lead to a violation of the space bounds is the summary set structure.

[1] This assumption requires a cooperative host environment; but there are reasonable work-arounds if the operating system insists on allocating addresses from disparate points in the address space, such as the sparse bitmap structure I used for the remembered set representation.

[2] This $O(N)$ size bound suffices because a $O(R)$ time collection pause does not process the stack of mark stacks.

6.1.1 Bounding time and space for summary set structure

The summary set structure is interesting because it has to satisfy the following constraints:

- Summary space: The space occupied by all of the summaries together must occupy at most $O(N)$ space,

- Summary traverse: When evacuating a region ρ, the collector must traverse the summary for ρ (finding the locations of references pointing *into* ρ) in $O(R)$ time,

- Summary update: When migrating objects from a region ρ to region ρ', the collector must update the summaries for the other regions in $O(R)$ time to reflect the change in locations for the migrated objects.

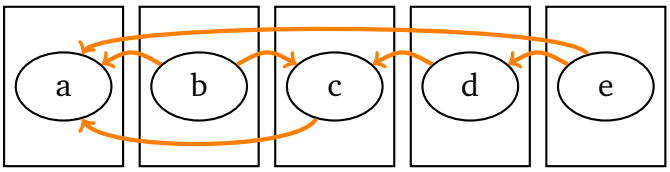
There are many obvious structures that can meet any two of the above goals, but satisfying all of the above conditions at once is non-trivial.

For example, consider a hypothetical representation that represented the summary for a region ρ by a table of all of the locations that could hold references into ρ. Since I wave off construction of summaries with more than SR elements, the total size of any one table is at most SR, and the design meets both the summary space and summary traverse requirements. However, after evacuating the region ρ, we would have a problem: there are $N/R - 1$ tables representing the summaries of other regions, each of size $O(R)$, and those tables might refer to locations in the region that the collector just finished evacuating. Even if the collector dynamically added new entries to the tables corresponding to the objects that had been migrated, the old summary entries still must be invalidated. Otherwise, the summaries with old entries will be unsound and the collector will eventually attempt to use the unsound summaries as the basis for collection.

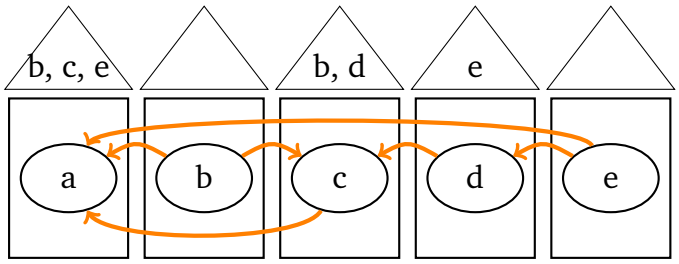
6.1.1.1 A sparse matrix structure

One suitable representation is a sparse matrix, where the rows and columns are both indexed by region. The column of the matrix associated with region ρ gathers the locations that may hold a reference pointing into ρ. The row of the matrix associated with region ρ' gathers the locations in that region that may hold a reference pointing into another region that has been summarized or is undergoing summarization. Thus a cell at (ρ, ρ') is all locations in ρ that may have a reference pointing into ρ'. As the heap grows large relative to the region size, most such cells will contain no locations at all, because it is impossible for a heap of N locations to densely populate a N^2 matrix. This motivates a sparse matrix structure, where the trivial cells containing no locations are not explicitly represented. Instead, the non-trivial cells can be linked together, enabling efficient traversal of either row or column.

Consider the following object graph, partitioned into five regions.

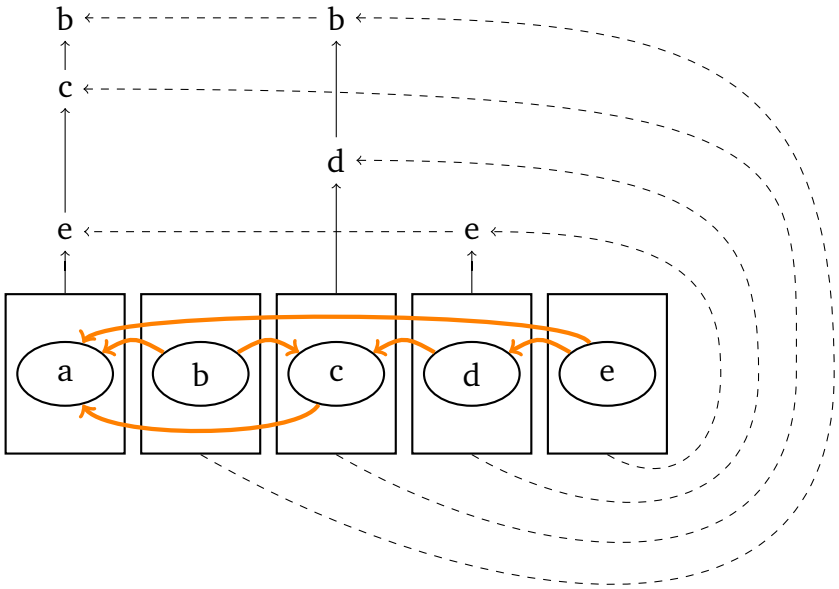


Abstractly, its summary set structure looks like this.[3]



The sparse matrix representation of the summary set structure might then look like this.

[3]The illustrations in this chapter all deliberately omit the distracting distinction between an object versus references to individual locations within the object.



In this illustration, a solid arc coming from the top of a rectangular region ρ represents the pointer to a linked-list of *column* entries representing the summary set holding references coming into ρ. A dashed arc coming from the bottom of a rectangular region ρ' represents the pointer to a linked list of *row* entries representing the contribution of ρ' to the entirety of the summary set structure. If a region does not have an arc originating from its top, then it has no column entries (a null list), and likewise for regions missing arcs at their bottoms.

Note that the cells of the matrix can and should batch together many locations from the heap; if each cell held only one address, the space occupied by the links of the sparse matrix would be intolerably large for reasonably sized heaps.

This structure enables the collector to traverse the summaries in a focused manner. For example, if the collector chose to evacuate the region holding the object c, it would traverse the column for that region, visiting the relevant locations within d and b to find all exterior references into that region. On the other hand, if the collector chose to evacuate the region holding b, it would not traverse any fields of objects in other regions because the

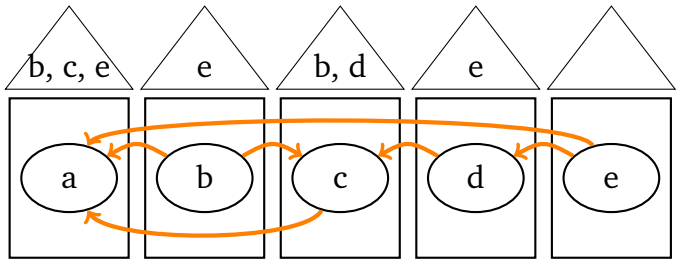
column for b's region is empty; but, it *would* traverse the row associated with the region and clear out the entries stored there. Thus the summary sets for the regions of a and c would both be modified to drop the location of b's field. Note that this clearing of the rows has nothing to do with whether b is still alive or is determined to be dead during the collection cycle. When the region is evacuated, b is either copied or deemed unreachable; either way, the storage associated with b from before the collection will be invalidated and reclaimed for future allocation.[4]

6.1.1.2 The rows are not redundant

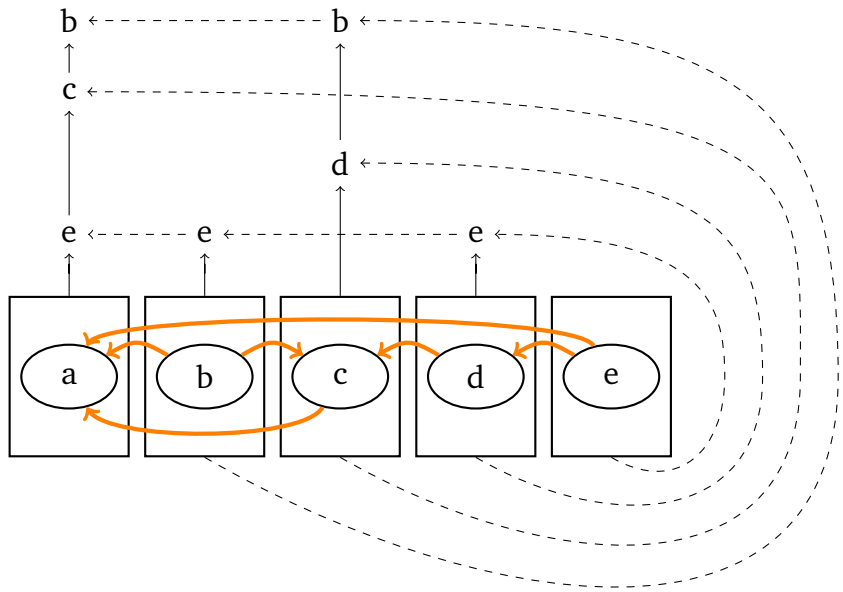
The illustration above hints at a correspondence between references going out of a region and the row for that region. This might lead the reader to wonder if this linked row structure is unnecessary, and if instead this cleaning of dead locations in the summary set structure could somehow be performed by traversing the region after it has been evacuated. There are a number of problems with this idea, however. Even ignoring details like the fact that the act of evacuating a region may overwrite the old state of its objects (e.g., by installing forwarding pointers), the simplest argument against this approach is that the summary set structure is *not* 100% precise!

As a concrete example, consider the same object graph, but now the summary set is slightly imprecise; perhaps at some point in the recent past e had a reference to b, now overwritten.

[4]Obviously this observation does not generalize to mark-sweep style non-moving collectors. It is not clear whether it would be possible to eliminate this row-traversal for live objects in a non-moving collector without introducing a prohibitive cost elsewhere in the system meta-data.



A corresponding sparse matrix representation for this abstract summary structure could be the following.

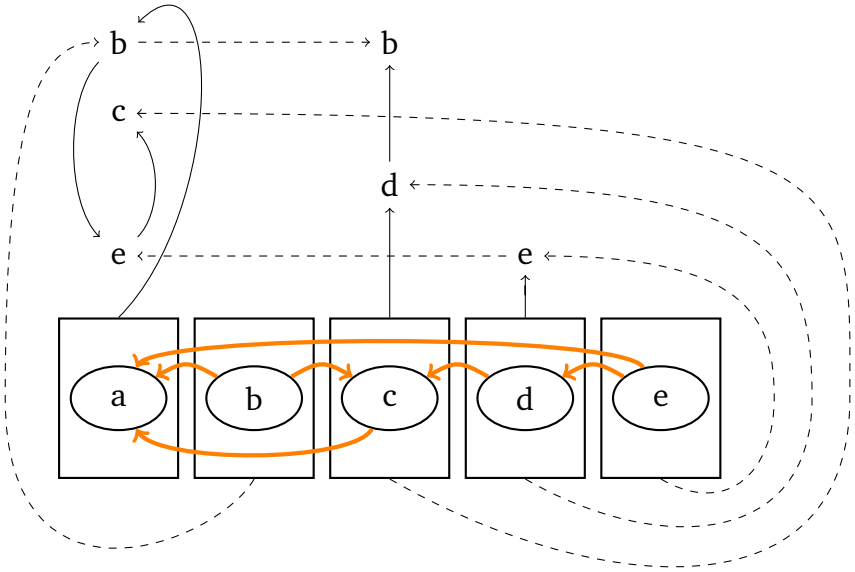


In this case, the row structure is crucial: if the collector evacuates the region holding e, it might be able to look at the fields of e to determine that the summaries for the regions holding a and d need to be updated to reflect the death or migration of e; there is nothing left in the state of e to tell it about the entry in the summary set of b's region. I *need* the rows represented by the dashed lines.

6.1.1.3 An anti-constraint: sortedness

A critical observation about the constraints on the summary set structure: the collector does *not* need to traverse the elements of the summary structure in any particular order. This means that the rows and columns of the

sparse matrix representation do not need to be linked in a sorted manner. The above illustrations present a linked structure that happened to be row- and column-ordered by the implicit region ordering in the presentation, but the abstract summary set structure is not uniquely represented. The original example could be represented by the following linked structure, among many others:



It is not as pretty a picture as the original illustration, but it provides a crucial insight: allowing the linked structure of the rows and columns to be arbitrarily ordered dramatically simplifies maintenance of the summary set structure.

If one had to keep the rows or columns linked in order, then the *first* introduction of a reference from region ρ to ρ' would require finding the correct spot in the row and column linked lists for the insertion. Such references could be introduced by the collector as it migrates objects or by the write-barrier as the mutator modifies fields. After finding the spot for the entry (ρ, ρ'), it is straight-forward to create a cell for that entry, and a mapping from the pair (ρ, ρ') to that cell can be kept in an auxiliary table (to handle subsequent introduction of other references between the same two regions; the matrix associates many locations within ρ with the cells of its row, otherwise the amount of space dedicated to the linkages in the summary set

structure would dwarf the number of actual locations stored). But if the links had to be kept ordered, then finding the correct spot for that first entry would generally require traversing an unaffordable amount of structure.

By allowing unordered linkage, the insertion process is simple: first check if (ρ, ρ') already has a cell by looking in the auxiliary table; if there is none, then take a free cell and add it to the head of the row list for ρ and the head of the column list for ρ'.

6.2 Bounding Time

The remaining two scalability properties relate to execution time:

1. c_0 is larger than the worst-case time between mutator actions, and
2. within every interval of time longer than $3c_0$, the minimum mutator utilization is greater than c_1,

where c_0 and c_1 are the existentially-bound constants from the main theorem.

6.2.1 GC Pauses

The constant c_0 is the constant bound on pause time. The most significant interruption to the mutator is due to a major collection evacuating a region ρ. Every region has size at most R, a summary of size at most SR, and an associated portion of the mark-stack bounded by $O(R)$.

Evacuating the region's objects and updating the mark stack can be performed in $O(R)$ time.

By using the sparse matrix structure described previously, scanning the summary column for ρ (to find locations of all incoming region-crossing references) takes time proportional to the summary's size.

There is also the cost of clearing the row for ρ. In one worst-case scenario, the mutator would guess a choice of small single-field objects in one

region ρ and iteratively update their fields to point to objects in every other region, thus introducing entries for all of those locations into the row for ρ. If nothing were done to eliminate such behavior, the row could grow to have $O(N/R)$ entries, which could not be cleared in $O(R)$ time.[5]

I already limit the amount of mutator work during a summarization cycle to cN; but this does not eliminate the hypothetical case above (since $cN = O(N/R)$).

However, not *every* row could grow to have $O(N/R)$ entries. The solution adopted here is to wave off collection of a region whose row is too large, the same way that the collector already waves off regions whose summary sets (the columns of the matrix) are too large. The bound of SR suffices for both purposes. Thus the time to clear a row for ρ is bounded by $O(R)$.

6.2.2 Worst-case MMU

My goal is to describe the worst-case minimum mutator utilization (MMU), independent of what program is executed or input data is fed to the mutator. The theoretical worst-case MMU at resolution Δt is the infimum of the mutator's CPU time divided by Δt, for all time slices of size Δt from any execution of any mutator.

A collector that occasionally interrupts the mutator to perform a full collection has a theoretical worst-case MMU of zero regardless of resolution Δt, because one can always select a program that will grow the heap to a point that requires more than Δt time to traverse its structure.

If there is a finite bound on the worst-case collection pause, as above, then the theoretical worst-case MMU may be positive for sufficiently large Δt.

[5] This scenario is perhaps absurd. Popular objects do arise in real programs and the threshold SR on summary set size is something the collector encounters in practice (albeit rarely). It is not clear what kinds of programming patterns would cause a corresponding blow-up in row-size.

Let c_0 be an $O(R)$ bound on worst-case duration of a collection pause, and let $\Delta t = 3c_0$. At time samples of that length or greater, the worst-case mutator utilization occurs when two maximal pauses surround a short interval of mutator work time. As already stated, $c_0 = O(R)$; we now show that the mutator will perform $\Omega(R)$ work between every two major collections.

Showing this tells us that the minimum mutator utilization, expressed as a percentage of the interval Δt, is independent of R. For example, if one were to choose to double R, the utilization would not suddenly be cut in half, as it might if the unit of uninterrupted mutator work were only bounded from below by some constant unrelated to R. Of course doubling R would increase $\Delta t = O(R)$; the MMU itself is a fraction of that interval.

The collector performs $\Theta(N/R)$ major collections per full cycle, and mutator work drives the scheduling of those collections. By the policies in section 5.5, the mutator performs A units of work (promotions or imperative assignments) per full cycle. Since $A = \Theta(P_{old}) = \Omega(N)$, the mutator work evenly distributes into $\Omega(AR/N) = \Omega(R)$ work between each major collection.

The above analysis accounts for time spent on collection effort and mutator work, but does not include the time spent on the marking and summarization processes. The effort expended by those two processes can be scheduled so the overhead per major cycle is $O(R)$, and therefore need not dominate the interval $\Delta t = 3c_0$ to the point where the mutator will not fulfill its obligatory c_1 units of work.

The marking process requires $O(N)$ time per full cycle, and it is straightforward to distribute its overhead to $O(R)$ per major cycle.

The summarization process performs at most F_3 passes over the remembered set per summarization cycle. Each pass takes $O(N)$ time scanning the remembered set and creates

$$O\left(\frac{1}{F_1}\frac{N}{R}SR\right)$$

entries in the summary set structure. The number of summarization cycles per full cycle is bounded by F_1, so the number of entries added by the summarization per full cycle is

$$O\left(F_3 \frac{N}{R} SR\right).$$

Distributing the work evenly across collection cycles in a full cycle implies that there is $O(F_3 SR) = O(R)$ overhead per major cycle.

Bounding the overhead to $O(R)$ does not imply that the overhead might not exceed the maximum time for a major collection, depending on how the summarization and marking processes are implemented. We in fact must choose c_0 to not only bound the maximum collection time, but also the time spent doing non-concurrent summarization and marking work during a major cycle. However, the reason to focus on bounding c_0 by the major collection time is that in the regional design, the summarization and marking processes can be run concurrently with the mutator, unlike the collector, whose execution is interleaved with the mutator. Therefore we assume that in the end, the summarization and marking work will not be a significant source of mutator interruption, and the important thing was more to show that the collector would not have to wait longer than $O(R)$ time for each to finish their obligations for a major cycle, because such waiting *would* be classified as added mutator interruption.

Thus completes the proof of Theorem 1, restated here: There exist positive constants c_0, c_1, c_2, and c_3 such that, for every mutator, no matter what the mutator does:

1. GC pauses are independent of heap size: c_0 is larger than the worst-case time between mutator actions.

2. Minimum mutator utilization is bounded below by constants that are independent of heap size: within every interval of time longer than $3c_0$, the MMU is greater than c_1.

3. Memory usage is $O(P)$, where P is the peak volume of reachable objects: the total memory used is less than $c_2P + c_3$.

Chapter 7

Bringing back the remembered set

Points-into summary sets are guaranteed to be bounded in size; but they must be periodically constructed. Section 4.3.3 outlines several summary set construction techniques; this chapter argues that using a remembered set to guide the summary set construction is an excellent heuristic technique.

7.1 Summary-set construction

Any summary-set construction technique must, given a subset $\{r_i, \ldots\}$ of the regions, incrementally visit every object in some sound approximation of the object graph and find all of the locations that hold references to objects in $\{r_i, \ldots\}$.

The following discussion of the construction techniques will ignore the requirement that the algorithm be incremental. All three techniques can be made incremental (mostly a matter of supporting interleaved object allocations and mutations), but describing those generalizations here would obscure the main points.

7.1.1 Construction via direct address scan

One approach to summary-set construction is to traverse the address space directly, dereferencing each location capable of holding an object reference. More specifically, for each mapped address l,

1. if l represents the start of an object o that was and is unreachable according to the last completed snapshot, then skip ahead to the next address following l; otherwise

2. if l is the address of a slot that may hold an object reference then dereference the slot for l. If l holds a reference to an object in r_i (where r_i is one of the regions $\{r_i, \ldots\}$ given for this summary-set construction cycle), and l itself is not part of the region r_i, then the summarizer adds the location l to the summary set for r_i.

Employing this technique does require the runtime system to track allocated memory blocks (as it must not attempt to dereference uninitialized memory). The runtime system must also be able to identify, from inspecting the object on its own, what slots within objects may hold references (as opposed to systems where objects do not have headers with dynamic type information and the runtime system relies solely on type tags encoded in the references to such objects or on static types encoded in the program text). Both requirements are easily satisfied in common garbage collection systems.

This technique is simple and conservative; it generally requires scanning all of the allocated memory, including some amount of floating garbage. It will traverse objects that have never held region-crossing references. Scanning all of allocated memory for each summarization pass intuitively sounds expensive; section 7.3 provides concrete performance comparisons.

7.1.2 Construction via reference graph tracing

The linear scan approach described above does filter out objects that are long dead (according to the snapshot) from its traversal, but each pass includes objects that have died since the last snapshot was taken. Therefore, one might well consider constructing the summary sets by tracing the object reference graph, starting from the roots, and record all encountered locations with region-crossing references.

This technique will yield more precise summaries, as it avoids traversing floating garbage objects, but it will still spend time traversing objects that have never held region-crossing references. Also, as an in-place traversal of the object graph, it will need to maintain meta-data to ensure that each object is traversed only once; this is additional overhead that the simpler scanning approaches do not incur.

7.1.3 Construction via remembered-set scan

Section 3.2.1 explains that a relatively small remembered set can focus the attention of the garbage collector, but that it is a heuristic technique and does not provide a guaranteed improvement. The remembered set is not guaranteed to be small relative to the total heap size.

7.1.3.1 The remembered-set observation

A sufficiently small remembered set is cheaper to scan in the common case than either of two options above (but not in the worst case).

Assume heap size N words, a regional remembered set of size M objects, live object graph of size O objects, and individual objects of size c words. Also assume that the remembered set is small relative to the live object graph.

It costs roughly N to do the linear address scan, roughly cO for the graph traversal, and roughly cM to scan the objects in the remembered set. Obviously $cM < N$, and since the remembered set is small, we also have $cM \ll cO$.

7.2 The remembered-set hypothesis

The crucial assumption justifying the use of a remembered set is that it will tend to be small relative to the whole heap. This is, a natural, dynamic distribution of objects tends to yield a small remembered set in common case programs.

This, the "regional remembered-set hypothesis," is what justifies maintaining a remembered set and using it as the basis for the summary set structure.

The next section presents some data evaluating the remembered-set hypothesis.

7.3 Comparing summarization techniques

To evaluate the different techniques and to test the remembered-set hypothesis, I instrumented a prototype regional collector to analyze the heap periodically. On every nursery evacuation (roughly corresponding to every megabyte of objects allocated), the instrumentation code traverses:

1. the objects in the remembered set,
2. the live objects in the heap, and
3. all allocated objects, including float.

In each object, all slots that could hold references are read, to make the instrumentation code reflect the time it takes to load that data from memory.

The three traversals are timed, and each traversal counts the number of slots scanned.

This way, the traversals provide two foundations for comparing the three summarization techniques: the number of words each technique would read, and a rough estimate of the time each traversal takes. The area underneath each elapsed time curve gives a relative idea of how much total additional overhead a summarization technique adds; an occasional spike in the curve should not lead to the dismissal of a technique if the area underneath is small overall.

The instrumented collector was run with a nursery size of one megabyte and a region size of four megabytes. So if the garbage collector collects the entirety of a region without needing to allocate a fresh region, then the size of the whole heap drops by one million words; such behavior can be seen in the plotted curves below.

Figures 7.1 through 7.11 present the results of the instrumentation for a collection of benchmarks. The top chart for each figure is the traversal times and the bottom chart is the number of slots scanned; the two charts are presented so that the x-axes of the two charts are aligned. (Note that the shapes such as squares, triangles, and crosses that identify each set of data appear only periodically amongst the dots that make up full set of data.)

The data for figures 7.1 through 7.11 present the initial setup for each benchmark as well as the actual benchmark run itself. This is significant in cases like `gcold` where the majority of the object allocation is spent setting up the benchmark, while the actual benchmark behavior is a small slice at the right end of the presented data.

Figure 7.1 presents data from a benchmark (`earley:10`) that does not use much memory. One can clearly see a sawtooth pattern in the slots count for whole heap traversal, corresponding to the gradual allocation and sudden reclamation of memory. One can also see the sawtooth looking at the dots (as opposed to the data-tagging shapes) in figure 7.2, though the effect

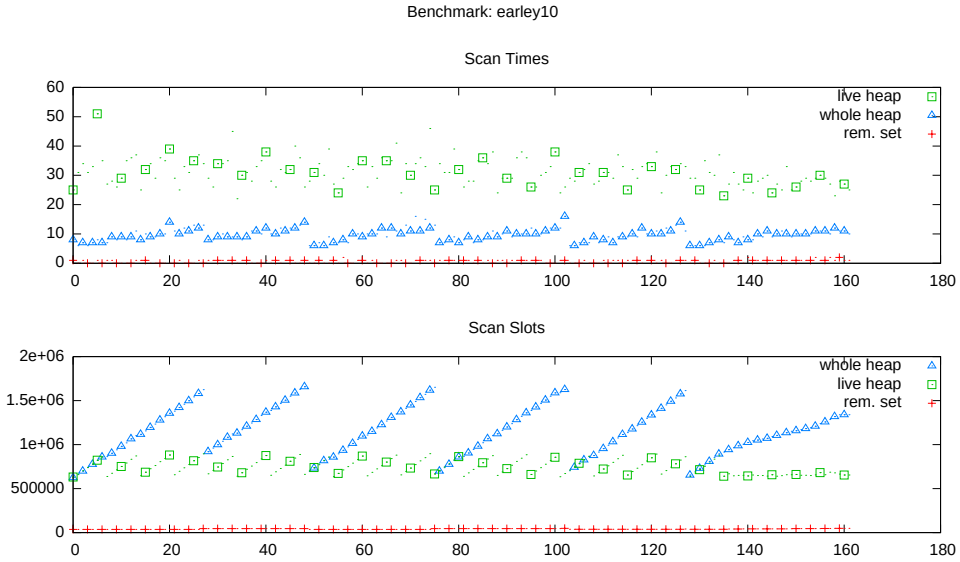


Figure 7.1: Summarization traversals, `earley:10`

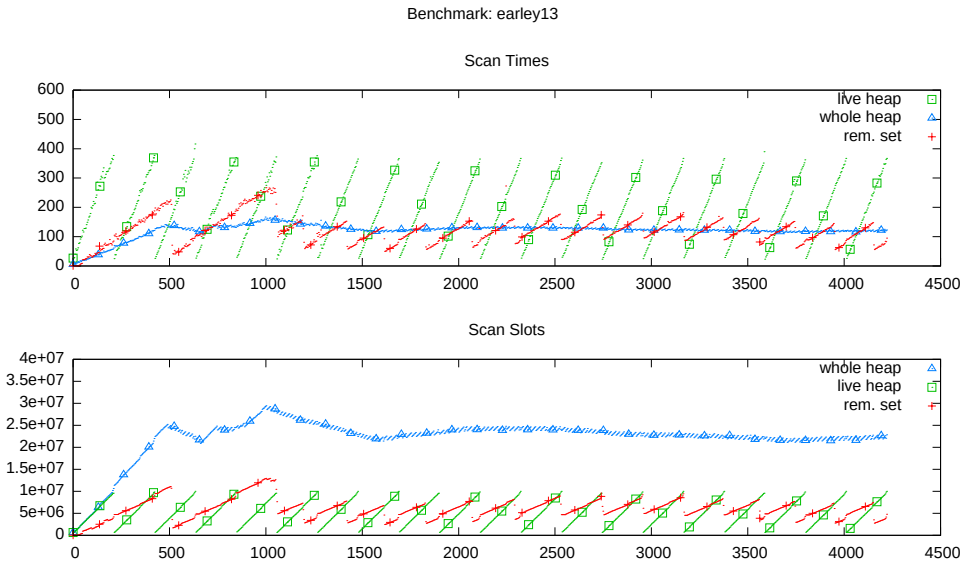


Figure 7.2: Summarization traversals, `earley:13`

is much more subtle here because this benchmark (`earley:13`) uses 20 times as much memory and so the drop with each collection is less significant.

In both figures 7.1 and 7.2, and also in all of the succeeding figures, the number of words scanned in the whole heap is always greater than or equal to the number of words scanned in the live heap, which is unsurprising since the whole heap is a superset of the live heap. More potentially surprising is that the time it takes to scan the whole heap is often significantly less than the time it takes to scan the live heap. I have two possible explanations for this phenomenon. One is that the overhead of tracing just the live heap and maintaining associated meta-data (such as a mark bitmap and mark stack to represent the seen nodes and frontier of the graph) far exceeds the extra time spent scanning floating garbage. The other attractive explanation is that the direct scan used to traverse the whole heap makes better use of the memory hierarchy, since it is essentially doing a streaming access of the address space, while tracing the live heap may appear like a series of random accesses from the viewpoint of the memory subsystem.

Figure 7.1's benchmark uses so little memory that the runtime can keep the live heap within a small number of regions (often just one) and it does not have many region-crossing references; thus its remembered set does not grow to a significant size, which is reflected in the series of red crosses that line the bottom of both charts.

In figure 7.2 (`earley:13`), we see that the remembered set does often grow large; the number of words scanned due to the remembered set sometimes exceeds the number of words scanned from the live heap. Likewise, the time to scan the objects of the remembered set sometimes exceeds the time it would take to scan the whole heap directly. Neither of the two techniques is an obvious winner.

In figure 7.3 (`gcbench`), the remembered set again occasionally grows large. But in this benchmark, it is always significantly cheaper with respect to time to scan the remembered set than to use the other two techniques.

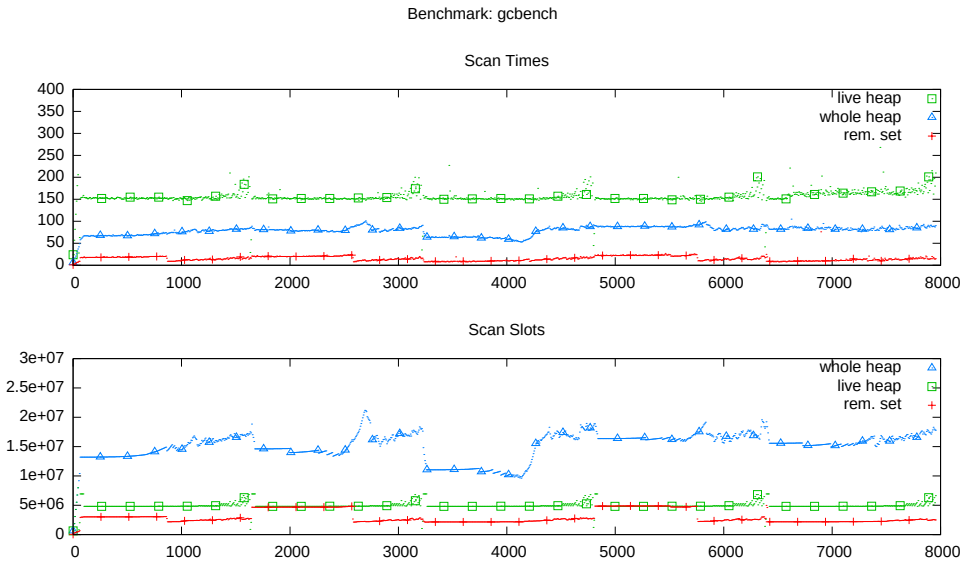


Figure 7.3: Summarization traversals, `gcbench`

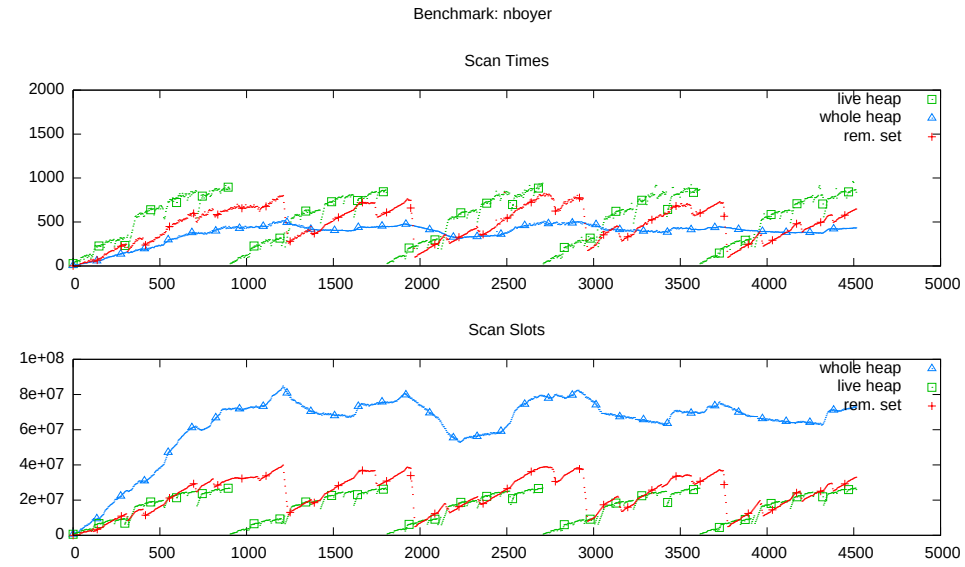


Figure 7.4: Summarization traversals, `nboyer`

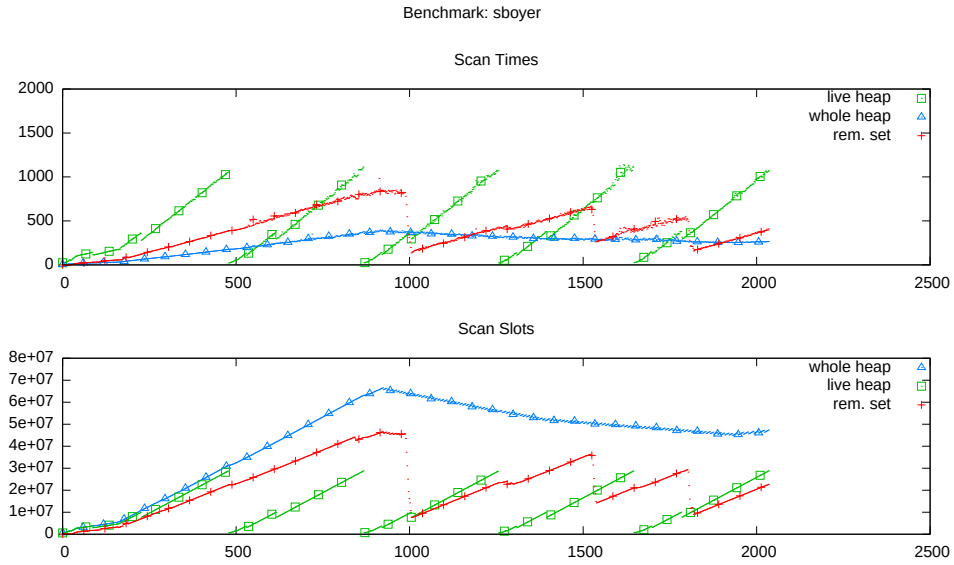


Figure 7.5: Summarization traversals, `sboyer`

In figure 7.4 (`nboyer`), the remembered set often takes more time to scan than just directly scanning the whole heap. Unlike `earley:13` (figure 7.2), this area underneath the scan-times curve for the remembered-set scanner seems to clearly exceed the area underneath the scan-times curve for the whole-heap scanner. This indicates that the remembered-set hypothesis is invalid: the size of the remembered set is growing quite large in the system's dynamic object distribution amongst the regions. This issue is addressed in section 7.3.1.

In figure 7.5 (`sboyer`), the remembered-set scanning times again often exceed the whole-heap scanning times.

In figure 7.6 (`perm9`), the remembered-set size and times are practically nil; using a remembered set to guide the summary-set scanner seems well motivated in this case.

In figure 7.7 (`twobit`), the size of the remembered set follows an interesting sawtooth pattern. Its size occasionally grows larger than the live heap, but the time to scan the remembered set is consistently far less than with either of the other two techniques. Again, using a remembered set seems

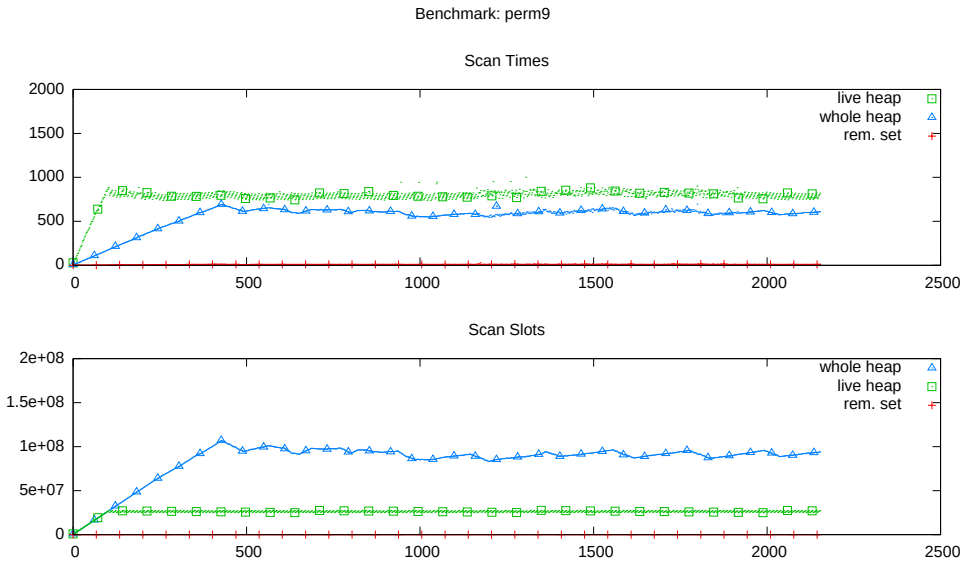


Figure 7.6: Summarization traversals, `perm9`

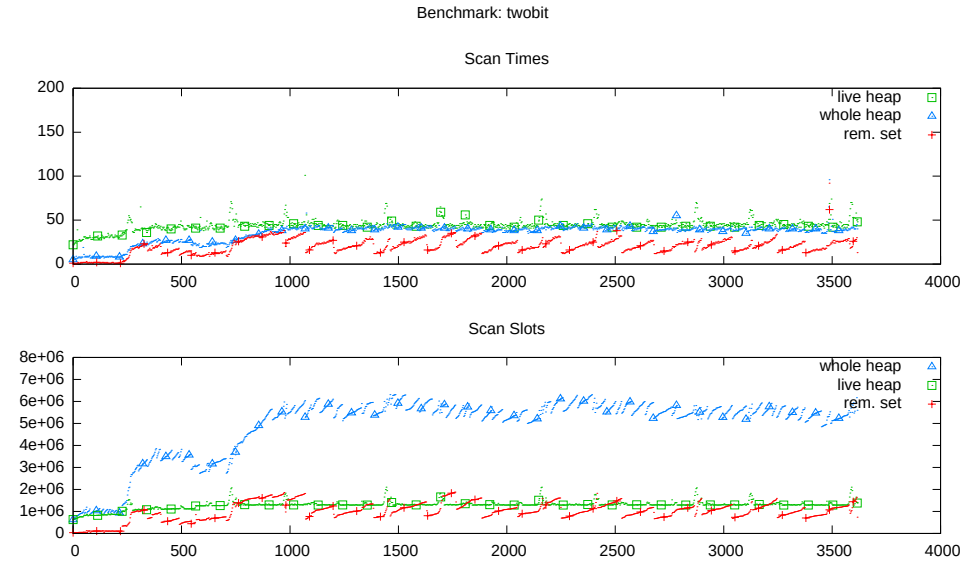


Figure 7.7: Summarization traversals, `twobit`

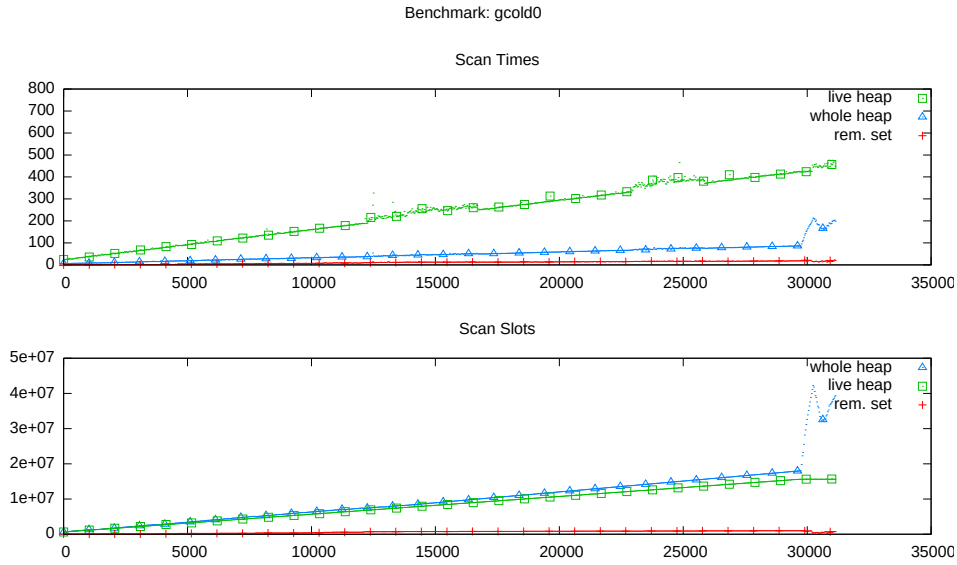


Figure 7.8: Summarization traversals, `gcold0`

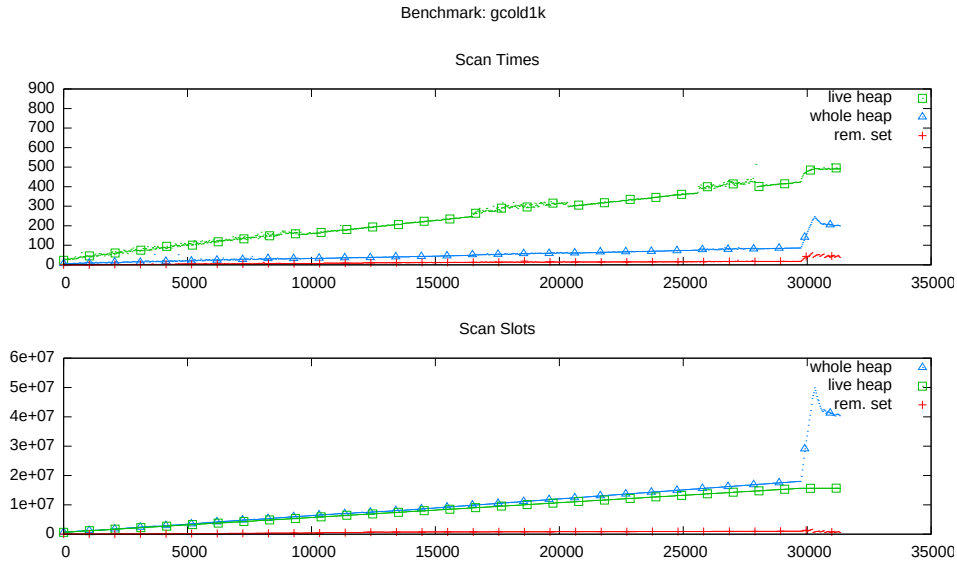


Figure 7.9: Summarization traversals, `gcold1k`

well motivated in this case.

In figures 7.8 and 7.9 (`gcold:0` and `gcold:1k`) the time scanning the remembered set is practically nil; another motivating example.

In figure 7.10 (`queue`), the time scanning the remembered set is practically nil.

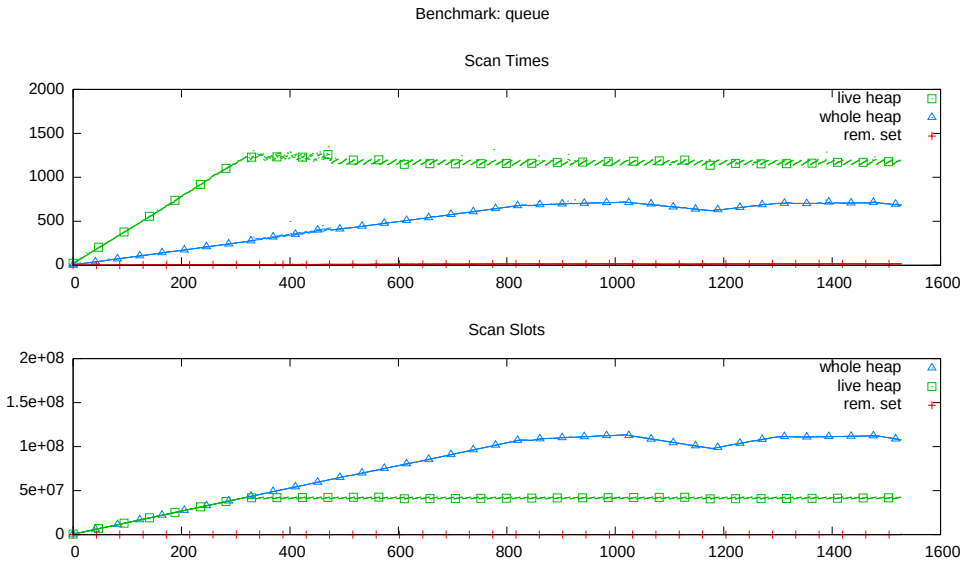


Figure 7.10: Summarization traversals, `queue`

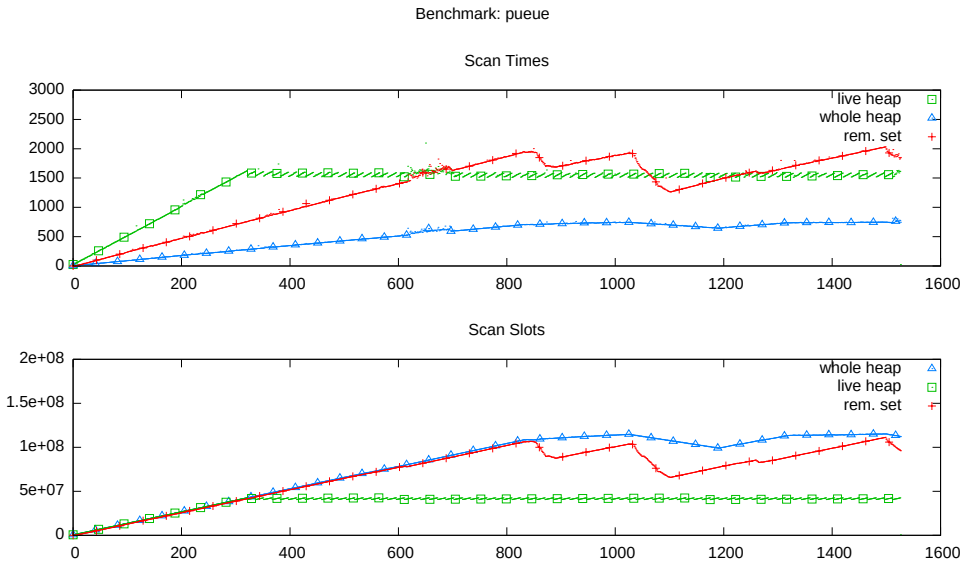


Figure 7.11: Summarization traversals, `pop-queue`

In figure 7.11 (pop-queue), which presents a variant of queue that incorporates popular objects, the remembered set grows to an enormous size (its size curve often approaches that of the whole heap), and the time spent scanning it is miserable.

This last example, as well as the earlier counter examples that seem to invalidate the remembered-set hypothesis, motivate a thought experiment presented in the next section.

7.3.1 Popularity and the remembered set hypothesis

Unfortunately, popular objects invalidate the regional remembered-set hypothesis. If every object in the heap has a reference to an object in the region ρ, then all of the objects in regions other than ρ have region-crossing references. In such a situation, a remembered set built according to the regional remembered-set invariant (page 15) would have size proportional to that of the heap.

This is troublesome for the regional collector. The region ρ will eventually be classified as popular: too expensive to be a candidate for collection. The objects within popular regions do not move. If most objects have no region-crossing references other than the reference(s) to the popular object, then the time spent scanning such objects for references into collectible regions is wasted.

In this scenario, the remembered set fails to focus the attention of the summarizer on the objects that actually matter for an upcoming collection.

7.3.2 Revising the hypothesis: region fame

The observation that popular objects invalidate the regional remembered-set hypothesis motivates a modification to the regional collector.

The regional collector already classifies some regions as too popular to collect, and abandons summary-set construction for such regions. Any effort

spent tracking references into popular regions in a remembered set (and potentially causing the remembered set to grow beyond a reasonable size) seems like it is wasted in the short term, since such regions may well remain popular long into the future.

In order to avoid wasting such effort, one must find a way to avoid tracking such references.

The modification I adopt is to introduce a subclass of the popular regions, named the *famous* regions, which have such an abundance of incoming references that the collector stops considering such references to be worth explicitly remembering. The modification is captured in the following revision to the remembered-set invariant.

Remembered-set invariant, Famous:
If live objects A and B belong to distinct regions, A's region is not famous, and B has a reference to A, then track B in the remembered set.

With famous regions, objects can have region-crossing references to popular objects without introducing the space and time overheads induced by a large remembered set. The mutator's write barrier can also be modified so that it does not record references pointing into famous regions.

Fame introduces two main complications: First, what policy determines when regions become famous? Second, how should the collector respond as objects in famous regions die?

7.3.3 Fame policy

Regions become famous when their incoming reference count exceeds a multiple of the region size. The summarization process can track these incoming reference counts independently of the summary set structure; that way, when the summarizer abandons construction of a summary set when a region becomes popular, it can continue to maintain the incoming reference count for the newly popular region.

The introduction of famous regions must not break the asymptotic space bounds of the collector. A simple way to ensure this is to restrict the maximum number of simultaneously famous regions to a fixed fraction of the heap. This restriction can be directly implemented: before reclassifying any region as famous, first ensure that the budget allows for the addition of another famous region. If the budget is exhausted, then the region is not reclassified as famous. (Note that fame is a *heuristic*, not a core part of the regional collector, and therefore it is always sound to abandon such reclassification, as opposed to popularity, where it is a necessity to reclassify regions as popular when appropriate in order to bound Cheney collection times.)

7.3.4 Recovery from fame

The objects in an famous region can die, of course. If the fame heuristic is going to be effective in a long-running program, then famous regions must be able to be eventually reclassified as non-famous and be subsequently collected. Otherwise a worst-case program would first fill up the budget of famous regions and then do the majority of its work with objects outside the famous regions, and the fame heuristic would be essentially useless.

The essential attribute of famous regions is that references into famous regions need not be tracked; such references can cross between regions and still belong to objects that are not in the remembered set. If an famous region is to be collected, then such objects need to be gathered and placed into the remembered set.

Some runtime process must accumulate these additions to the remembered set. The summarization process is not useful for this task, since it scans only the objects that were already in the remembered set in the first place. On the other hand, the marking process is a perfect match for this task: it incrementally traces a complete snapshot of the heap, so while it builds its snapshot, it can also reconstruct the remembered set as appropri-

ate.

Of course, one does not want to waste time reconstructing the remembered set just to discover that a region is *still* famous. Fortunately, the marking process again provides a suitable solution to this problem. Rather than guessing when to reconstruct the remembered set for an famous region, the marking process can maintain an incoming reference count for all famous regions. If the count stays high, then the region should remain famous. If the count drops suitably low, then the region is reclassified as *under-reconstruction*. The *next* marking trace of the heap is instructed to reconstruct the remembered set for references pointing into regions under-reconstruction as it builds its snapshot. Once this second trace has been completed, then all regions under-reconstruction can be reclassified as filled, making them available for summarization and subsequent collection.

Regions that are under-reconstruction still cannot be collected until the reconstruction has completed, but since the remembered set is being rebuilt for them, the mutator's write barrier must record any references pointing into regions under-reconstruction, because that is part of the remembered set recovery.

The big picture with respect to fame is that the summarization process is in charge of turning popular regions into famous ones (up to a bounded fraction of the regions), while the marking process is in charge of (1) determining when the number of references into famous regions has dropped sufficiently far to allow such regions to be garbage collected, and (2) rebuilding the remembered set in preparation for such collection.

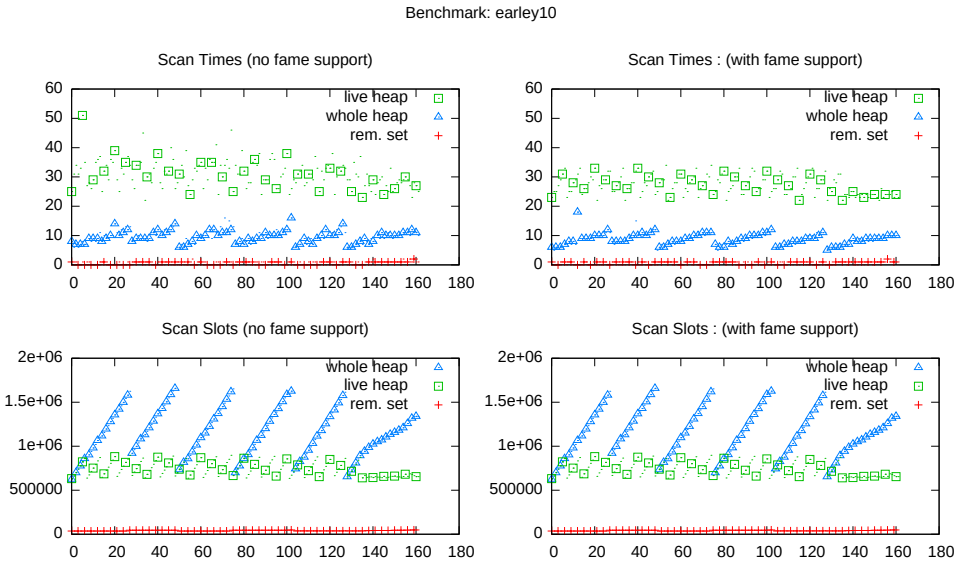


Figure 7.12: Traversals and fame comparison, `earley:10`

7.4 Fame and the regional remembered-set hypothesis

Figures 7.12 through 7.22 present the same comparison of the three traversal techniques, but with support added for famous regions. The left-hand side of each figure is the same data that was presented in figures 7.1 through 7.11; the right-hand side shows the results after adding fame support.

A comparison of the left and right hand sides of figures 7.12 and 7.13 shows that the addition of fame support makes little difference for the `earley` benchmark, regardless of whether the benchmark uses a small or large amount of memory. Likewise, figure 7.14 shows that fame support makes no significant difference for the `gcbench` benchmark.

Figure 7.15 shows the potential benefit of the fame heuristic. The remembered set size on the left side, before fame support, often rises above the size of the live heap, and costs a significant amount of time to traverse. After the addition of famous regions, the remembered set regularly drops to essentially zero size.

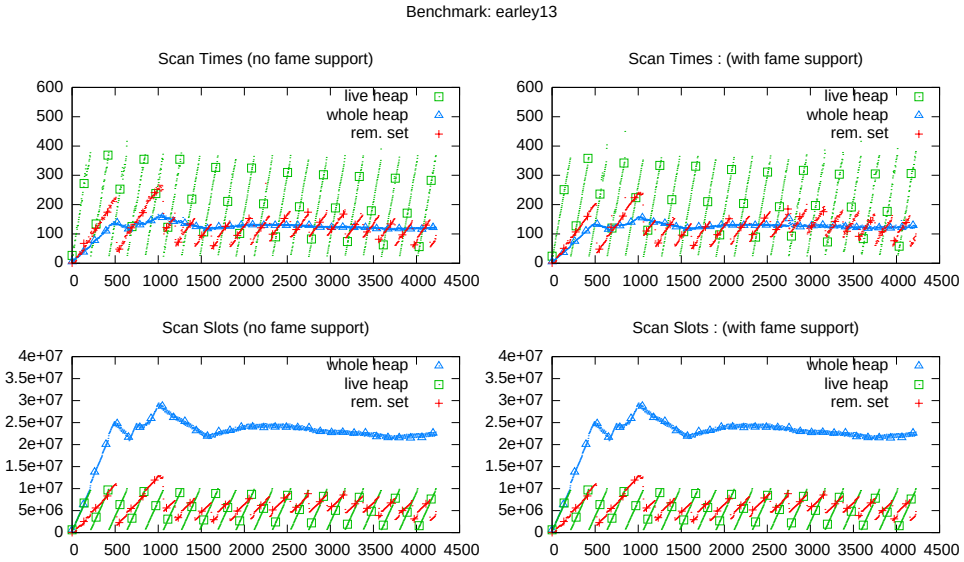


Figure 7.13: Traversals and fame comparison, `earley:13`

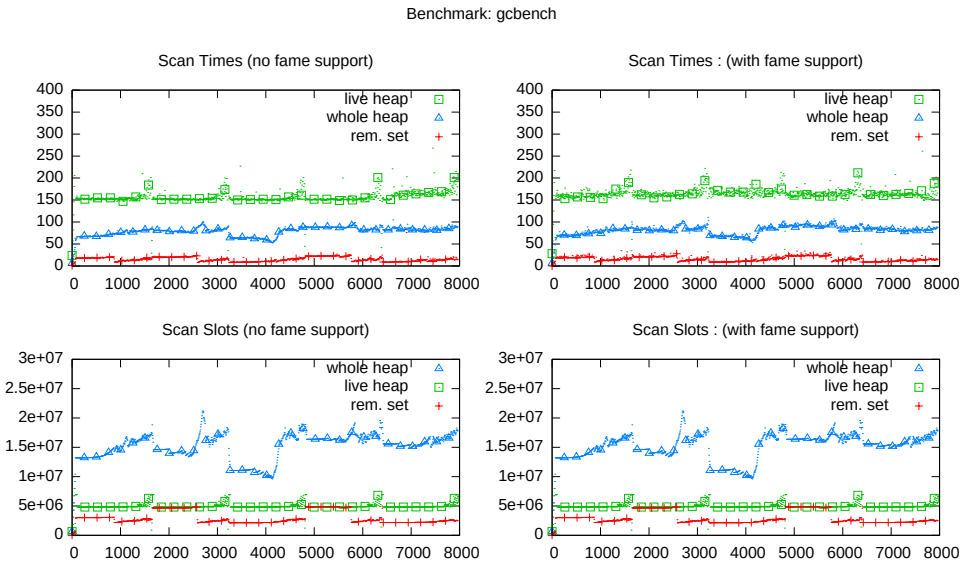


Figure 7.14: Traversals and fame comparison, `gcbench`

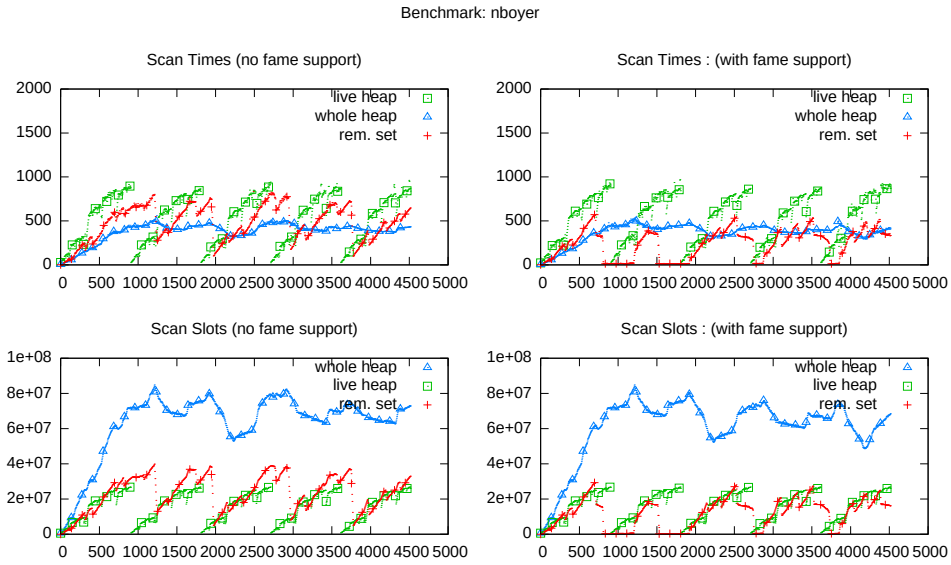


Figure 7.15: Traversals and fame comparison, `nboyer:5`

The sawtooth pattern of the remembered set size in the right hand side of figure 7.15 should not be surprising. The remembered set does not remain empty forever; the fame heuristic does not attempt to predict what objects will become popular ahead of time. Therefore one should expect to see the remembered set size rise and suddenly fall as the number of objects pointing into a region grows up to the fame threshold and then passes it.

Most importantly, the scan times in the right-hand side of figure 7.15 clearly show that the remembered set traversal is superior to both whole and live heap traversal.

Figure 7.16 shows that fame has some small effect on `sboyer`, but it is not a clear win in this case.

Figure 7.17 shows that fame has no effect on `perm9`, which is unsurprising; the point of fame is to reduce the remembered set size, and this benchmark already has a tiny remembered set.

Figures 7.18, 7.19 and 7.20 shows that fame has no effect on the `twobit` and `gcold` benchmarks.

Figures 7.21 and 7.22 shows that fame has no effect on the `queue` bench-

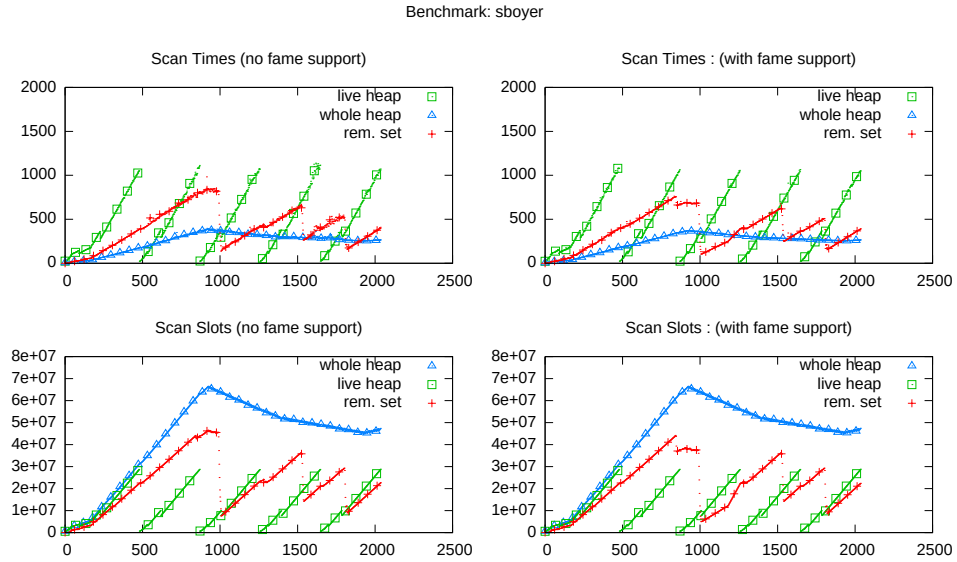


Figure 7.16: Traversals and fame comparison, `sboyer:6`

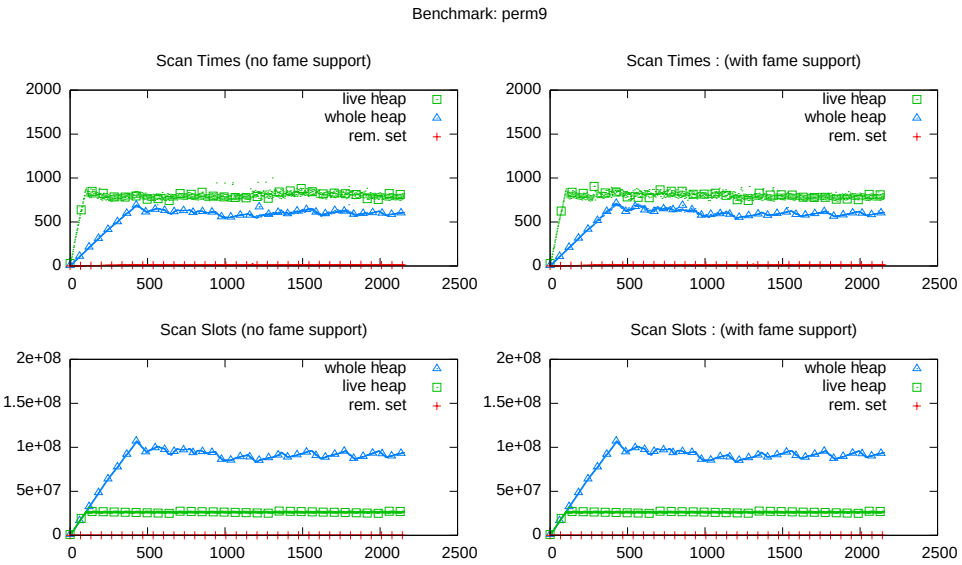


Figure 7.17: Traversals and fame comparison, `perm9`

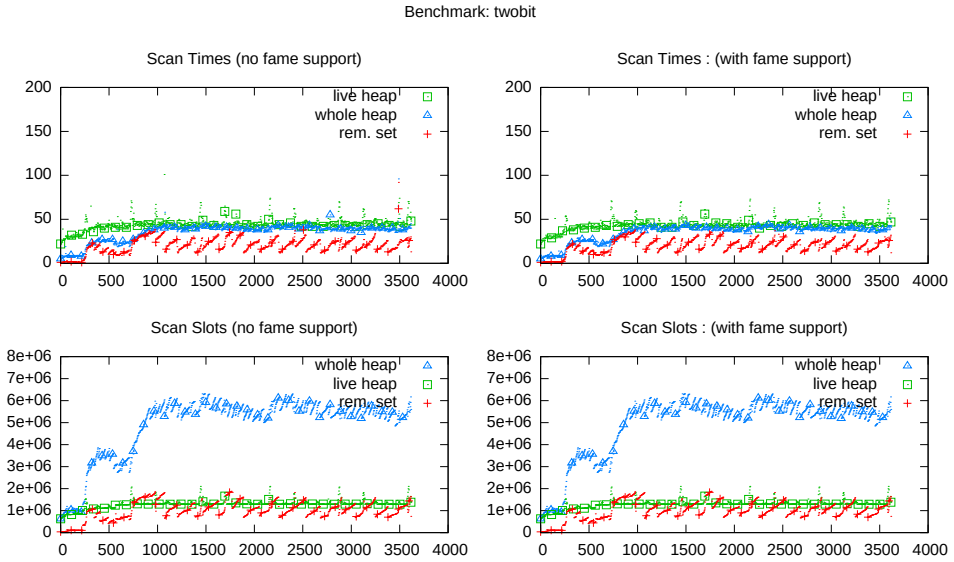


Figure 7.18: Traversals and fame comparison, `twobit`

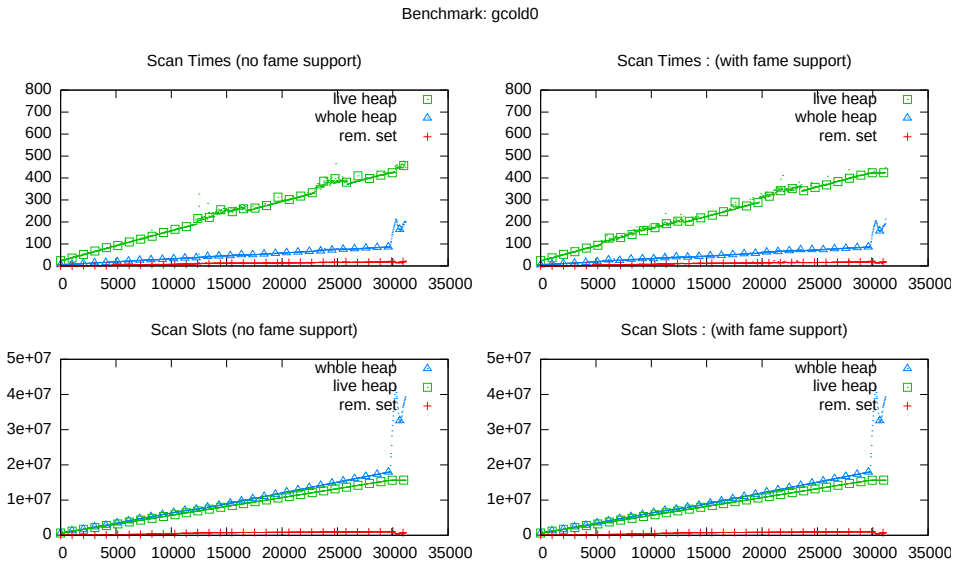


Figure 7.19: Traversals and fame comparison, `gcold:0`

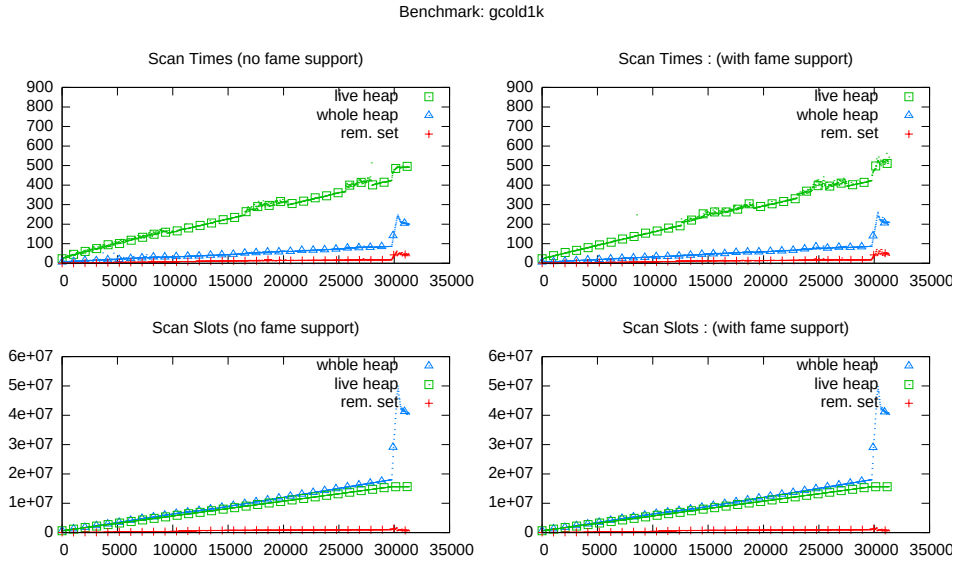


Figure 7.20: Traversals and fame comparison, `gcold:1k`

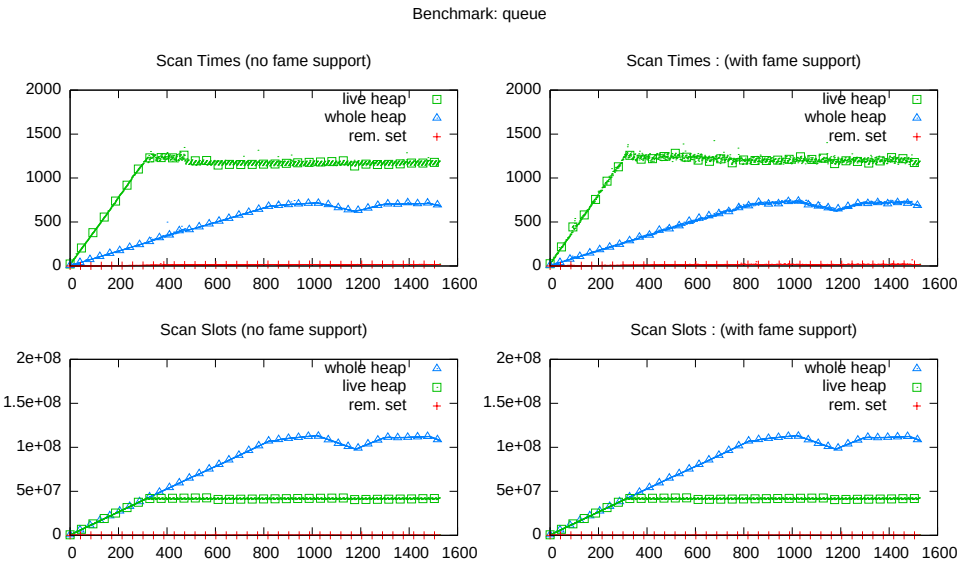


Figure 7.21: Traversals and fame comparison, `queue`

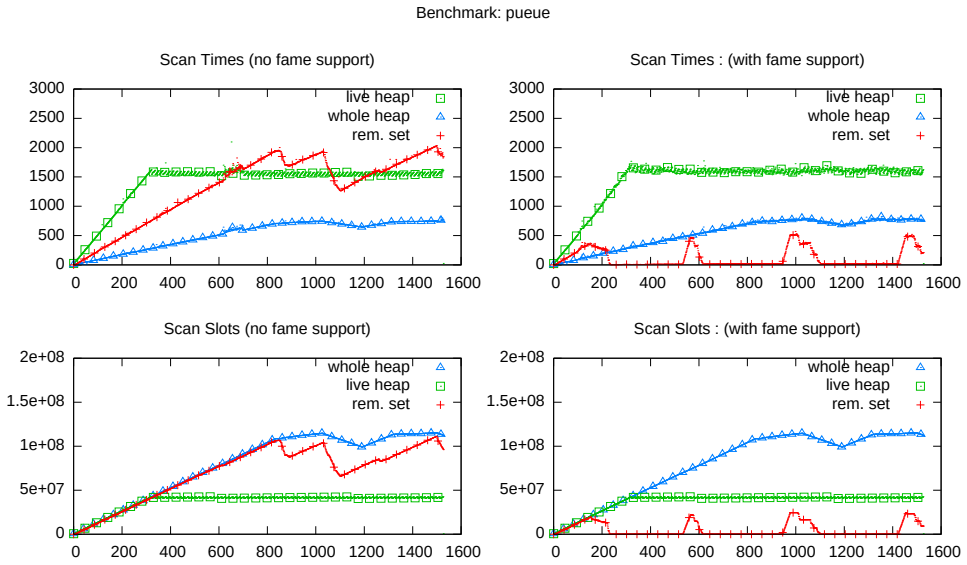


Figure 7.22: Traversals and fame comparison, `pop-queue`

mark, and that fame has a dramatic effect on the `pop-queue` benchmark. The `pop-queue` benchmark was specifically devised to illustrate the behavior of the regional collector on programs with popular objects. This confirms that fame has little effect on programs with no popular objects and that fame can greatly improve the effectiveness of a remembered set on programs with popular objects.

7.5 Effectiveness of the remembered set

The data presented in figures 7.1 through 7.22 is all based on preliminary instrumentation of a regional collector. Section 9.2 presents results including comparisons of overall elapsed times and memory usage before and after the addition of the fame heuristic.

Similar analyses measuring live storage versus time can be found in [16]. The main conclusions to draw from these figures are as follows.

- Using a remembered set is almost always more efficient than trying to traverse the live heap image.

- The focus provided by a remembered set can make traversing it more efficient in many cases than traversing the whole heap via a direct scan.

- A remembered set can be foiled by popular objects such as those found in the `nboyer` and `pop-queue` benchmarks; adding support for famous regions can make the remembered set more effective in such cases, as illustrated in figures 7.15 and 7.22.

- There are still cases (figures 7.13, 7.16) where the remembered set is not a clear win over a direct traversal of the whole heap.

Perhaps a hybrid approach to summary-set construction, that chooses a traversal technique dynamically, is worth investigating in future work. But absent further information, use of a remembered set in the regional collector appears justified in almost all cases.

Chapter 8

Opportunities for concurrency and parallelism

Four processes constitute the design of the regional collector: the mutator, the collector, the summarizer, and the marker.

I chose this decomposition because I believe it has much potential for making use of multi-core parallel processing hardware that has become ubiquitous.

This raises two questions: how much concurrent execution do we expect to exploit from this decomposition, and what complications arise when making a truly parallel version of the regional collector?

8.1 What can run in parallel

When running processes in parallel, one must consider what potential sources of interference could arise from their concurrent execution.

It is sound for many processes to read the same data structure in parallel, but if one process wants to modify the data structure, then either it must obtain exclusive access, or the whole system must be designed so that modification will not interfere with the actions of the parallel processes.

However, the meaning of the phrase "modify the data structure" is a little fuzzy in the context of a language runtime with a copying garbage collector. In particular, there are two kinds of modifications one must consider. The first class of modifications are high-level changes to the abstract object graph, which are introduced solely by the mutator (all other system processes should preserve the structure of the abstract object graph).[1] The second class of modifications consists of low-level changes to the memory representation of objects, which are introduced by a copying collector when it reclaims storage and reuses it as a destination for newly copied objects.

Running the mutator and collector processes in parallel is a heavily-researched area. We are deliberately *not* attempting to develop a runtime that does the same; we assume from the outset that concurrent collection introduces too many complications and overheads. We want a design that gives implementors more flexibility in how to implement their mutator and their collector.

However, the summarizer and marker processes are much simpler than the full-fledged collector process. In particular, they do not attempt to reclaim the memory associated with unreachable objects, and they do not modify the structure of the object graph. This makes them attractive candidates for parallel execution.

The read actions performed by the summarizer and marker could interfere with both classes of modifications (high- and low-level) listed above.

For this work, I assume that the complications introduced by designing the summarizer or marker to run in parallel with the collector are comparable to the difficulties of running the mutator in parallel with the collector. That assumption is largely based on my gut-intuition; it may well be the case that the collector could efficiently focus its attention on shuffling objects between regions that are unrelated to the regions being traversed by

[1] There is the potential exception of primitive data structures that must allow observation of the collection behavior, such as finalizers or weak references.

the summarizer or marker. But I doubt it, based on investigations while debugging prototype regional collectors. So the collector will not run in parallel with any of the other processes; when it runs, it will have exclusive access to the heap and all of the meta-data.

What about the mutator? The mutator's actions can also interfere with the summarizer and marker. So it would seem like the mutator would also require a lock on the heap when it runs. However, the mutator is already obligated to invoke a write-barrier when it modifies the object graph; this provides a hook where the system can avoid granting the mutator an exclusive lock for its write actions. Instead, the write-barrier acts as a channel that communicates the object graph changes to the summarizer and marker. Following a snapshot-at-the-beginning philosophy, mutator writes do not discard information about the past object graph state, but instead funnel the old and new arcs in the object graph through to the summarizer and marker. The summarizer and marker work with a nondecreasing collection of information about the object graph, regardless of what actions the mutator takes.

This effectively means that even though the mutator's writes present potential for a read-write conflict between it and the summarizer and marker processes, it is feasible to use the write-barrier to mediate the conflict. Thus we can treat the mutator, summarizer, and marker as parallel readers, while the collector is the sole writer.

8.2 How to make it work

A multiple-reader/single-writer locking mechanism is straight forward to construct atop more primitive mutex and condition variable abstractions; see for example section 7.1.2 of [13].

Many other complications already have to be handled in a merely incremental, not parallel, setting. For example, an incremental summarizer and marker must yield periodically to the collector; choosing the appropriate

grain of work unit and making their internal data-structures properly respond to the actions of the collector is a key task of making them incremental. See for example the event sequence presented in section 4.6.1, which describes a scenario that is fundamentally due to the processes operating incrementally and allowing control shifts between them.

In my prototypes so far, I have chosen one of the simplest units of work for the summarizer's scanning action: a whole region. If the mutator requests a collection, it must wait until the summarizer has finished scanning the region it is in the midst of, if any (or rather, the portion of the remembered set for that region). This approach was not applicable to the incremental marker, since it does not traverse the object graph region-by-region; there I kept a count of the number of arcs traced and used that to bound the amount of time the marker spent during any traversal step.

A bigger complication is the mutator's write-barrier. It may be possible to design the data structures of the summarizer and marker to allow the mutator to feed them information in a lock-free manner, but determining how to do this was out of the scope for this project. Instead, since the collector periodically interrupts the work of all the other processes to claim exclusive access to the heap, I chose to take advantage of that as a synchronization point. The mutator's write barriers, in addition to their work adding entries to the remembered set, also update logs for the summarizer and marker. The summarizer log tracks newly-introduced arcs from field `A.f` to object `B`; the marker log tracks the previous value held in `A.f`. Then when the collector runs, it folds those logs into the state of the summarizer and marker. In the case of the summarizer, it adds new entries to the summaries (both those under-construction and those that have been completed). In the case of the marker, it enqueues the log entries onto the mark stack, as appropriate for a snapshot-at-the-beginning system.

8.3 Caveats

The prototype described in this document does not yet support parallel execution of the summarizer or marker processes. It is beyond the scope of this project to add support for parallel tasks to the Larceny runtime, the test-bed for the research. I have outlined above the main issues I encountered while developing a smaller model of the system and then attempting to make it parallel. This should not be considered an exhaustive list of the problems one would encounter in a real-world implementation.

Chapter 9

Evaluation

This chapter presents data comparing the performance of the regional collector in various configurations with established generational and stop-and-copy collector technologies.

On programs that do not need much live memory at once, the regional collector is intended to behave more like a standard generational collector than a stop-and-copy collector. On programs that require large amounts of live storage, the regional collector will not behave like either of them, because both of the others will have occasional pauses of significant length, while the regional collector's pauses should never exceed a fixed bound.

9.1 Tiny Benchmarks

This section presents the performance of the regional collector on a set of "small programs": programs which do not need much memory because their peak volume of live objects is small.

Such programs are worth investigating because one does not want to pay much overhead for scalability guarantees when not intending to scale an application to large heaps.

By design, if the heap does not get large relative to the size of a region, then the regional collector should behave similarly to a generational collec-

tor. Both reap the benefits of a nursery, and since the heap is small, the regional collector's remembered set cannot grow terribly large.

I evaluate how well the regional collector satisfies this goal by running the regional collector on a large body of small benchmark programs. I compare the performance of the regional collector (in a few different configurations) to the performance of a stop-and-copy and a generational collector.

Each of the benchmarks in this section was run on top of six different runtime configurations. The reason for six configurations is to show how similar the regional collector is to the generational collector on most programs, as explained below.

The performance characteristics of the generational collector largely depend on the size of its nursery, which is a parameter of the runtime system for both the generational and the regional collectors. In this section, I have run the generational collector with a 1 megabyte nursery (its default setting on Larceny at this time my research was initiated) and a 4 megabyte nursery (its current default setting).

I have also run the regional collector with a 1 megabyte nursery and a 4 megabyte nursery (both with a region size of 8 megabytes) in order to answer the following question: does changing the nursery size largely affect the regional collector in the same manner that it does the generational collector? If so, then that provides evidence that the regional collector behaves much like the generational collector on the benchmarks in this section.

I have also run the regional collector with a 1 megabyte nursery and region size of 4 megabytes, because that is a configuration of the collector I will be discussing later when I talk about scalability on gc-intensive benchmarks.

On each benchmark, the following figures present a stack of three bar charts. Each chart presents a different aspect of a benchmark's behavior when run atop various runtime configurations.

- The top chart in each figure shows the maximum mutator pause time

observed over the program run.

- The middle chart shows the peak memory usage of each collector. The bar is broken down into the memory used by the mutator (the heap), which consumes the bulk of memory for most of these programs, and the meta-data for each of the collector's distinct subcomponents (the marker's bitmap and stacks, the summarizer's summary sets, and system's remembered set).

- The bottom chart shows the total elapsed time to run the benchmark. The bar is broken down into the time spent in the mutator, the Cheney collector, the incremental summarization coroutine, and the incremental marker coroutine.

In all three of the stacked charts, smaller bars are better.

In most of these non-gc-intensive programs, the elapsed time is dominated by mutator activity. Also, in many of these tiny programs, the elapsed mutator time is independent of the collector technology (but this is not the case in general).

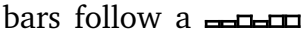
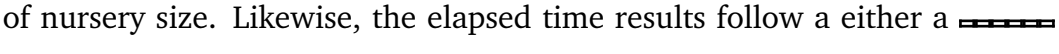
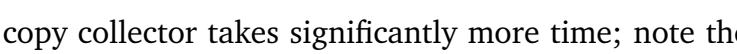
The chart presentation is organized first by benchmark program, then by collector technology and parameters. Each benchmark, which is labeled at the bottom of the figure, has six columns associated with it. The first three columns for each benchmark are the stop-and-copy and the two generational configurations (first the large nursery and then the small one). They are standard technology. The last three columns for each benchmark are three configurations of the regional collector: first the large nursery and large region size, then the small nursery with large region size, and finally the small nursery with small region size.

There are three main expectations for this data. First, on most small programs, the regional collector should act like the generational collector; in particular, its behavior should mostly depend on its nursery size, as that is what largely dictates the behavior of a generational collector. The two

generational columns are for a large and small nursery, and the three regional columns are for a large, small, and small nursery. So, for the most part, the second column (generational, 4 MB nursery) should look much like the fourth column (regional, 4 MB nursery). and the third column (generational, 1 MB nursery) should look much like the fifth and sixth columns (regional, 1 MB nursery).

Second, on most programs the regional collector should behave more like a generational collector than like a stop-and-copy collector. So the differences among the last five columns (generational and regional collector configurations) should tend to be small in comparison to how much they all differ from the first column (a stop-and-copy collector).

Third, the heaps never get terribly large, so I expect the pause times to be relatively small compared to when the heaps get large in section 9.2. The regional collector's maximum pauses for these small benchmarks may be worse than the generational collector's maximum pauses, but that is a result of the generational collector doing particularly well on those benchmarks. The regional collector design does not promise to always provide the best maximum pause times; it merely guarantees that its pauses will never exceed a program-independent fixed upper bound.

In figure 9.1, we see five benchmarks. For these five, the maximum pause time of the stop-and-copy collector dwarfs that of the generational and regional collectors. In four of the five benchmarks, the peak memory usage bars follow a  pattern, indicating that the choice between generational and regional technologies is insignificant in comparison to the choice of nursery size. Likewise, the elapsed time results follow a either a  (all elapsed times are about the same) or a  pattern (the stop-and-copy collector takes significantly more time; note the thick solid bar representing time dedicated to Cheney style copying garbage collection). This indicates that for these small programs, the choice between generational and regional technology is irrelevant compared to the benefit of either tech-

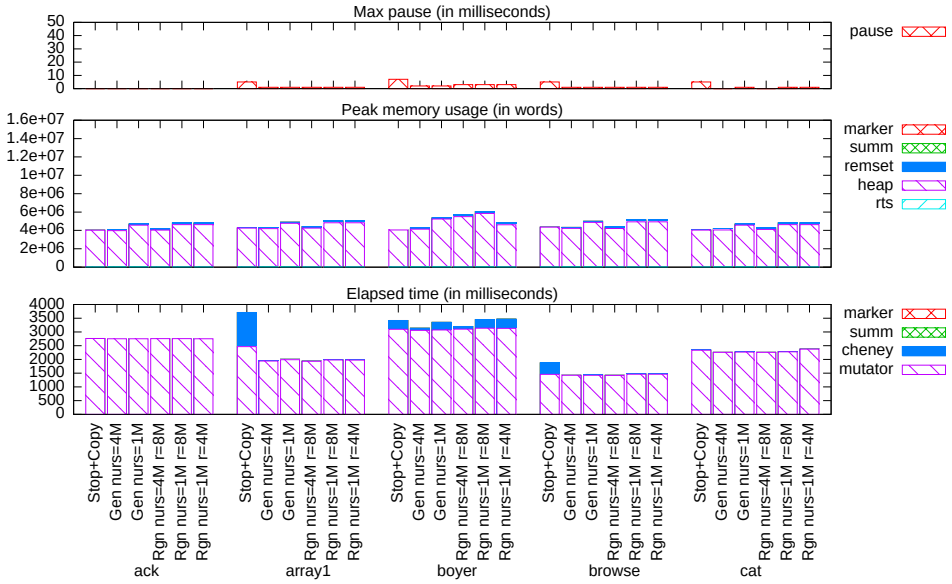


Figure 9.1: Collector comparisons, small programs 1

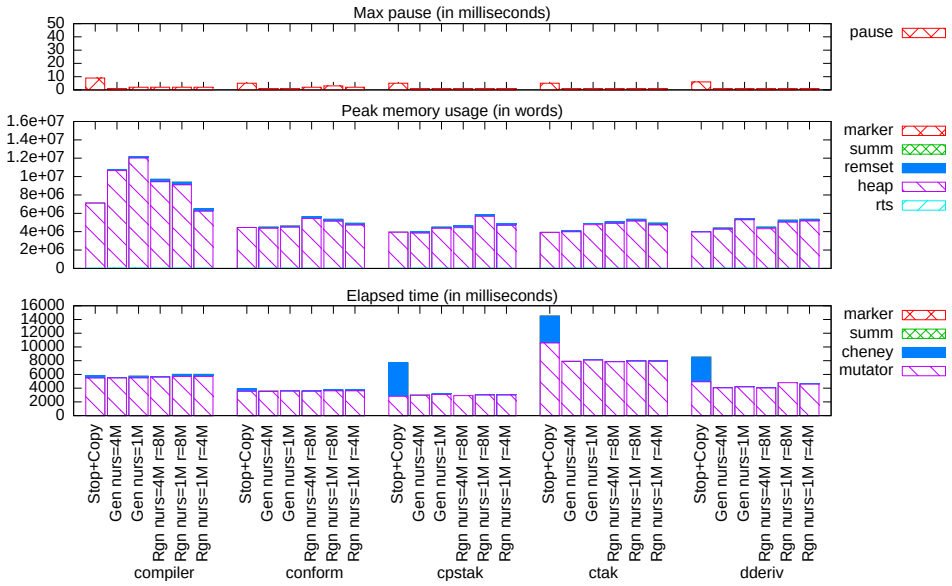


Figure 9.2: Collector comparisons, small programs 2

nology over a stop-and-copy collector.

In figures 9.2, 9.3, 9.4, 9.5 and 9.6, the patterns in the maximum pause times mostly match those of figure 9.1. The peak memory usage results are inconclusive for these benchmarks. Sometimes the regional collectors use a bit more memory than the other two technologies, but there are cases

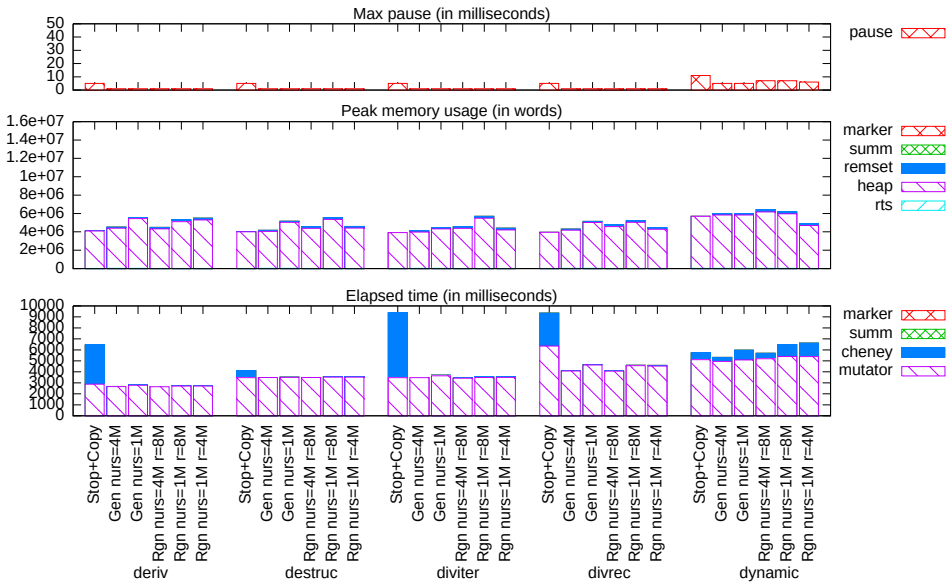


Figure 9.3: Collector comparisons, small programs 3

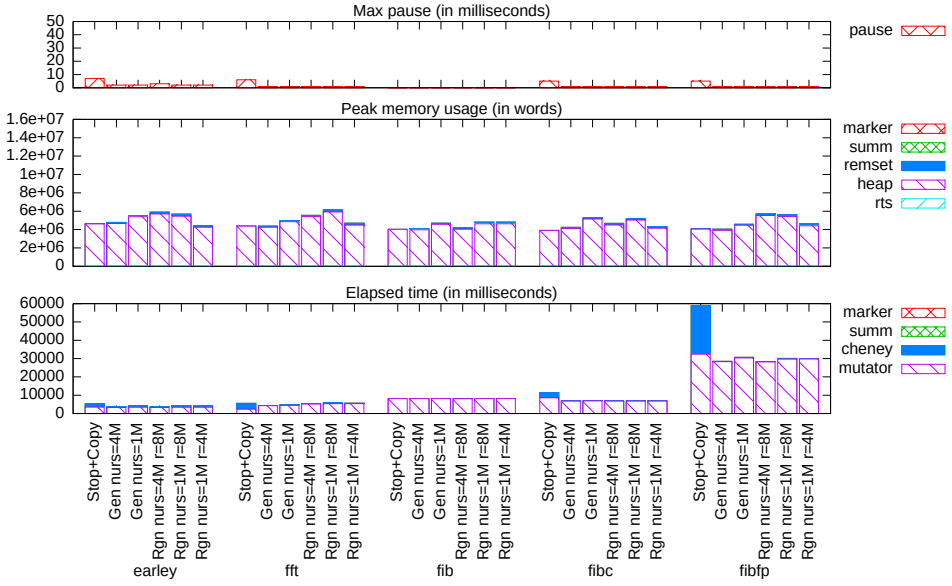


Figure 9.4: Collector comparisons, small programs 4

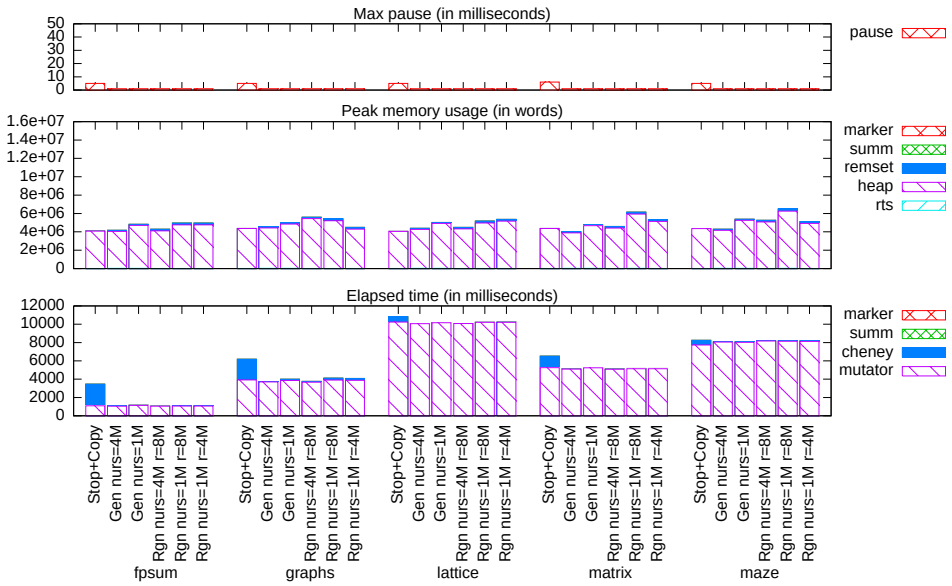


Figure 9.5: Collector comparisons, small programs 5

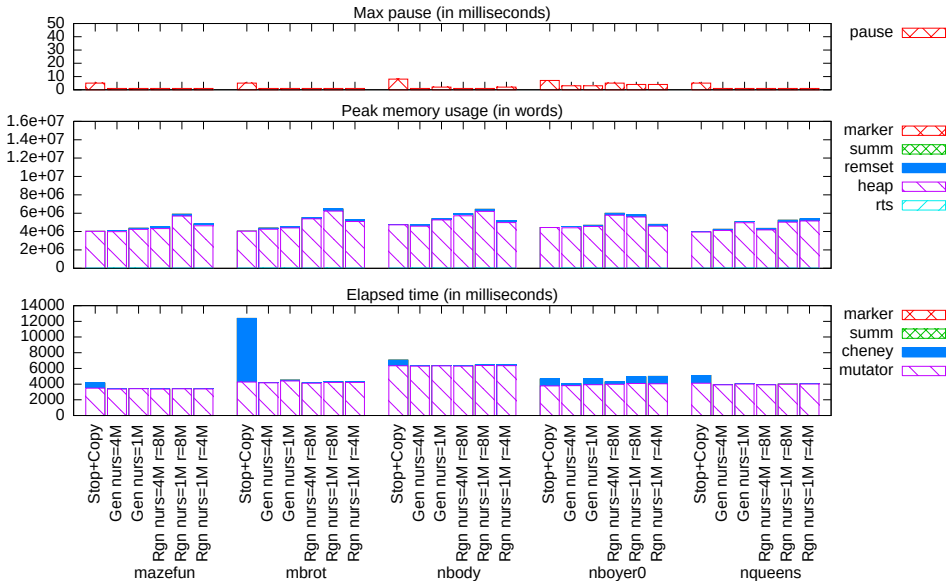


Figure 9.6: Collector comparisons, small programs 6

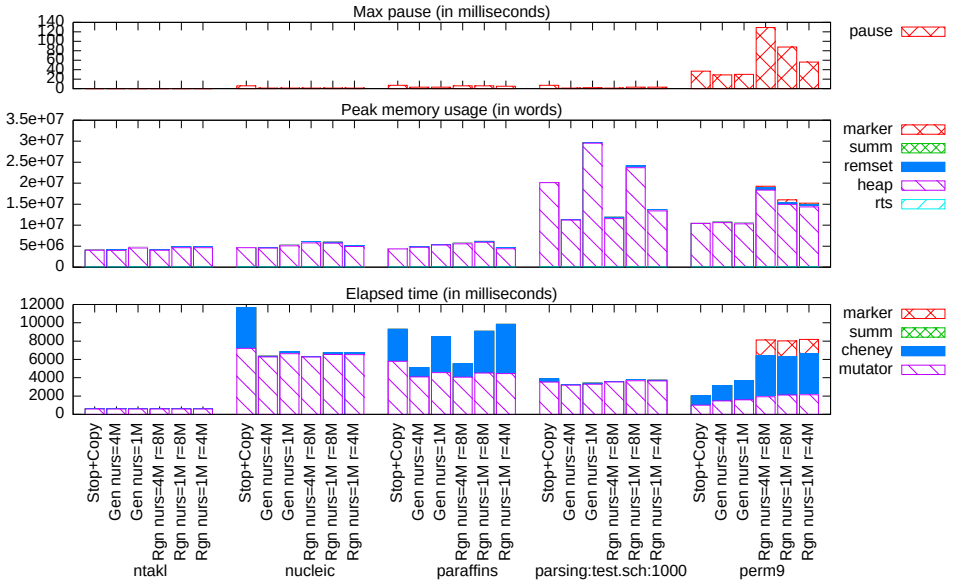


Figure 9.7: Collector comparisons, small programs 7

like the `compiler` benchmark (figure 9.2) where the generational collector has the largest peak memory usage. The elapsed time results again indicate that for these small programs, the choice between generational and regional technology is irrelevant compared to the difference between such technologies and a stop-and-copy collector.[1]

Figure 9.7 presents some interesting behaviors worth pointing out. The elapsed time results for the `paraffins` benchmark follow a ⊓⊔⊓⊓ pattern, again indicating that the difference between generational and regional is insignificant compared to the choice of nursery size (the small bars correspond to a 4 megabyte nursery, while the last three large bars correspond to a 1 megabyte nursery). The data for `parsing:test.sch` indicates a non-trivial relationship between collector technology and peak memory usage. The ⊓⊔⊓ pattern indicates that generational/regional sometimes performs

[1]There are cases like the `dynamic` benchmark (figure 9.3) where the stop-and-copy collector performs a bit better than generational/regional with a 1 megabyte nursery; the point is not that generational/regional technologies provide a guaranteed benefit, but rather that on most small benchmarks, the regional collector behaves like a generational one, for better or for worse.

better than stop-and-copy, sometimes worse, and it is not clear what role the nursery is playing in these results, as a 4 megabyte nursery consistently wins, but a 1 megabyte nursery loses in two out of three cases.

Finally in figure 9.7, the data for the `perm9` benchmark, the first case among the small benchmarks where the max pause duration is significant for the regional collector, shows two interesting pause time characteristics in its ▭▭▭▯▭ pattern. First, the staircase effect for the final three bars shows that the max pause time for the regional collector clearly depends on both nursery size and the regional size; as either decreases, the maximum pause time decreases. Second, on this benchmark the regional collector's maximum pause time is worse than that for the stop-and-copy and generational collectors; moreover, its maximum pause time approaches 130 milliseconds, significantly larger than any of the other maximum pauses in the remaining data for the small benchmarks. The pause time for the regional collector on this benchmark, and thus the worst-case bound on the regional collector's pause time, is significantly larger than what one typically witnesses for established garbage collector technology on most small programs. In general, the worst-case bound of the regional collector has no relationship to the pauses observed on typical *small* programs. The goal of the regional collector is to bound the pauses on programs using *large* amounts of memory, as discussed in section 9.2.

The peak memory usage and elapsed time for the `perm9` benchmark is also interesting. It is one of the few cases among the small benchmarks where the regional collector requires significantly more memory than both the stop-and-copy and generational collectors. Likewise the regional collector requires significantly more elapsed time; some of that can be charged to the incremental marker, but clearly a significant portion is due to time spent doing extra Cheney collection. The Cheney collection time and extra memory usage is an indication that the regional collector is suffering more from floating garbage than the other collector technologies. This benchmark

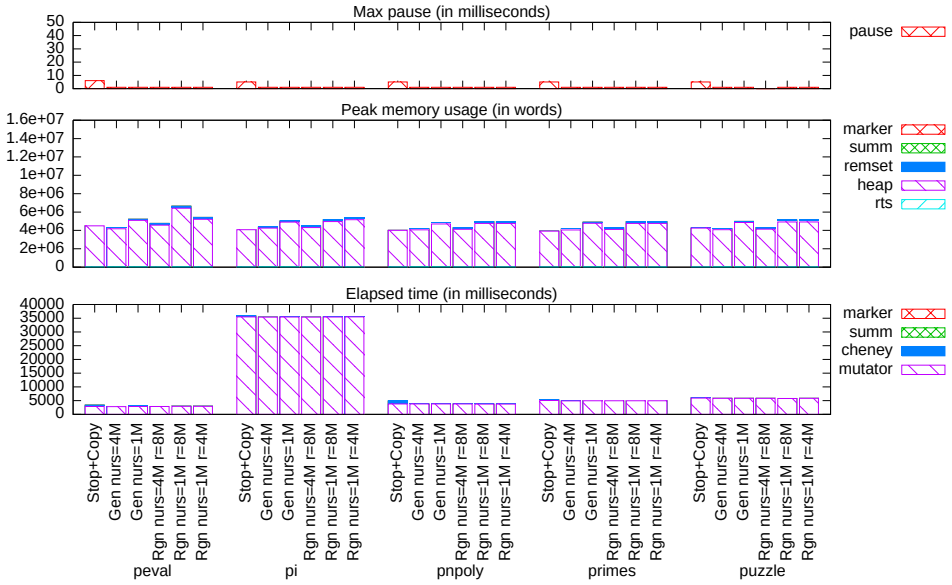


Figure 9.8: Collector comparisons, small programs 8

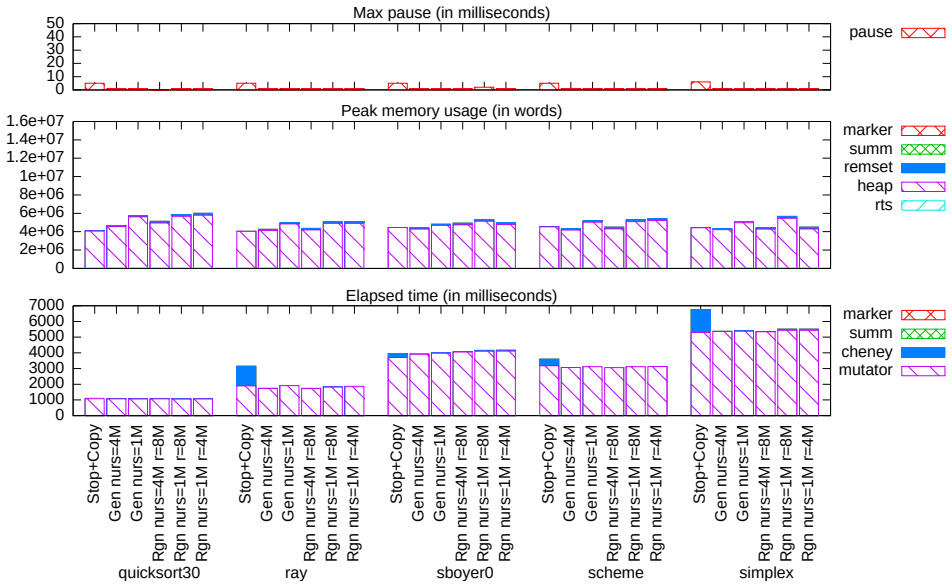


Figure 9.9: Collector comparisons, small programs 9

clearly illustrates an interesting atypical case.

The data presented in figures 9.8 and 9.9 mostly match the patterns presented in figure 9.1. The ▬▫▬▫▫ pattern resurfaces in several of the peak memory usage charts, and the maximum pause of the stop-and-copy collector again dwarfs the generational and regional collectors.

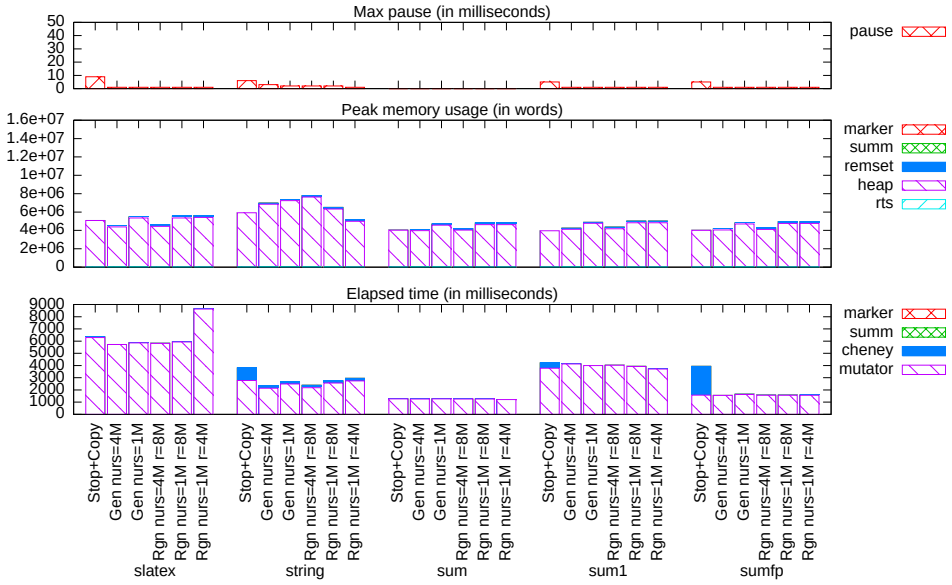


Figure 9.10: Collector comparisons, small programs 10

In figure 9.10, the elapsed times for the `slatex` benchmark follow a ▭▭▭▭▯ pattern, indicating that using the regional collector with a small nursery and small region size, which was previously shown to have the best pause time bound (observed), does have a potential cost, as that configuration adds additional overhead to the mutator's running time on this benchmark.

In figure 9.11, the peak memory usage for the `tail` benchmark follows a ▭▭▭▯▭ pattern, indicating that the regional collector can be better or worse than the other two technologies depending on nursery and region size.

Figure 9.12 shows three final small benchmarks where all of the collectors have the same performance characteristics.

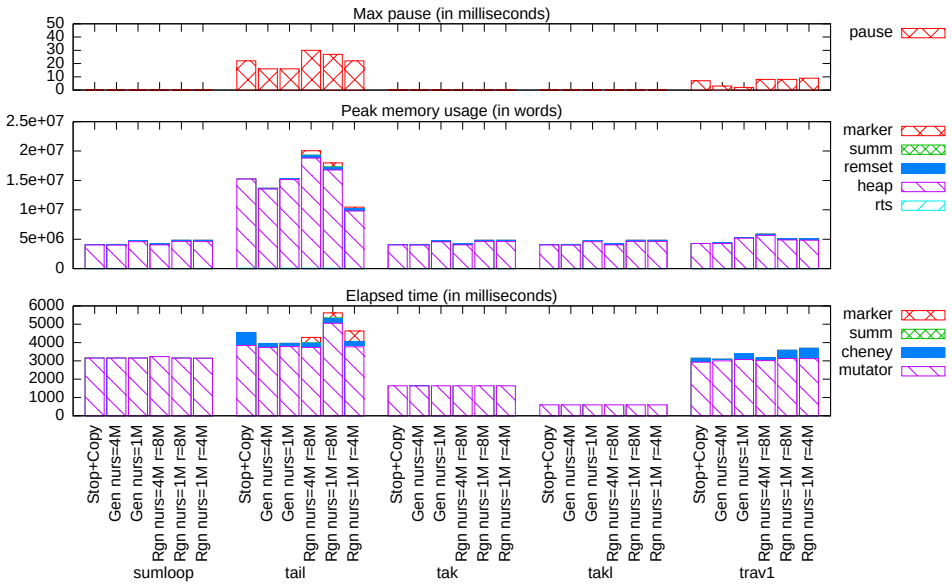


Figure 9.11: Collector comparisons, small programs 11

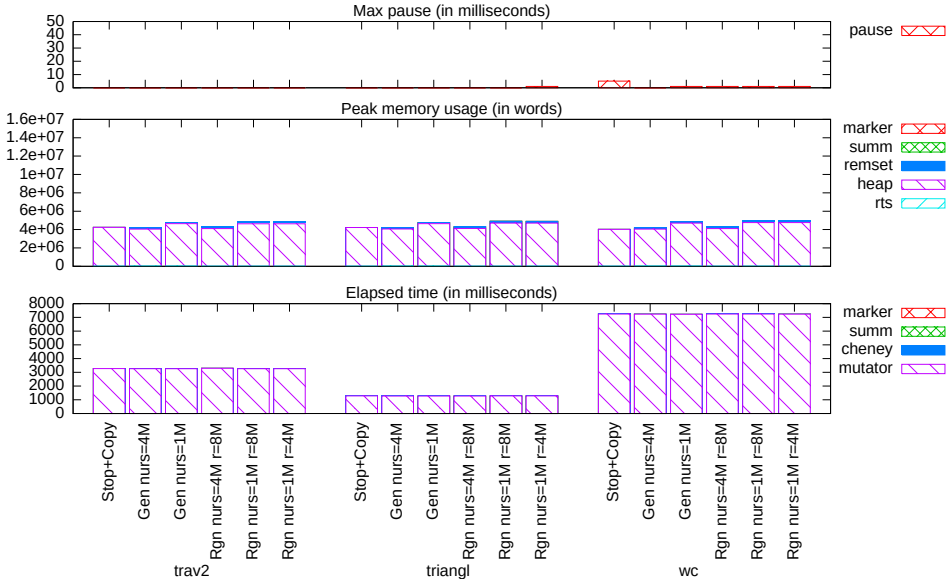


Figure 9.12: Collector comparisons, small programs 12

9.2 Big benchmarks, and the big picture

The regional collector does not behave like a generational one on programs that have large amounts of live storage. On such programs, generational collectors often have fewer significant pauses than a simple non-incremental stop-and-copy collector, but the pauses still occur occasionally.

A regional collector's pauses should never exceed a fixed bound. This does not mean that a regional collector's pauses will always be shorter than those of a stop-and-copy or generational collector. This is not just a hypothetical statement; several of my benchmark programs are cases where the longest pause of a generational collector is shorter than the longest pause of the regional collector, and this is reflected in the data described below.

Note also that the set of runtime configurations (the subdivisions within a group) is totally different for this set of data than in the previous section. In particular, I am no longer trying to illustrate similarities between generational and regional collection, but rather their differences, as well as illustrating the trade-offs when varying the parameters of the regional collector.

The set of runtimes benchmarked here are:

- The stop-and-copy collector.
- The generational collector (with a 4 MB nursery).
- Three configurations of the regional collector (see below), with fame support (see section 7.3.2).
- Finally, the same three regional configurations were run again, but with region fame disabled.

My main goal in analyzing three configurations of the regional collector was to explore the effect of varying the popularity factor, S. It is not

generally possible to vary S without having also to vary one of the other parameters of the collector. (One could choose extremely conservative values for all the regional collector's parameters, but such conservative configurations would not yield useful comparisons against the stop-and-copy and generational collector configurations.)

Therefore, I selected three sets of configurations that occasionally yield interesting differences in behavior.

The first configuration is one that Will Clinger discovered during early experiments with the system; it uses a relatively large value of 8 for the popularity factor S, and a minimal number of summarization scan attempts ($F_3 = 1$).

The second and third are configurations I found more recently, in an effort to find reasonable configurations with smaller values for S.

The first regional configuration is: $F_1 = 2$, $F_2 = 2$, $F_3 = 1$, $S = 8$. The second regional configuration is: $F_1 = 1.75$, $F_2 = 2.75$, $F_3 = 2$, $S = 6$. The third regional configuration is: $F_1 = 2$, $F_2 = 3$, $F_3 = 2$, $S = 4$.

9.3 The big benchmark suite

This section presents the same pause/memory/time charts in the same style shown earlier, but now for a set of benchmarks where the collector must manage a large amount of memory. In particular, the benchmarks evaluated are those presented in chapter 7 (figures 7.12 through 7.22).

Figure 9.13 shows the max pause, peak memory, and overall elapsed time for `earley:10`, `earley:13`, and `gcbench`. The `earley:10` benchmark is short running and does not require much heap memory; the pause times for all runtime configurations for `earley:10` are also relatively short. The `earley:13` benchmark data illustrates an effect of scaling up the problem size to a larger heap (reflected in the larger bars of the middle chart). Now the collector's summarization and mark state occupy a non-trivial amount

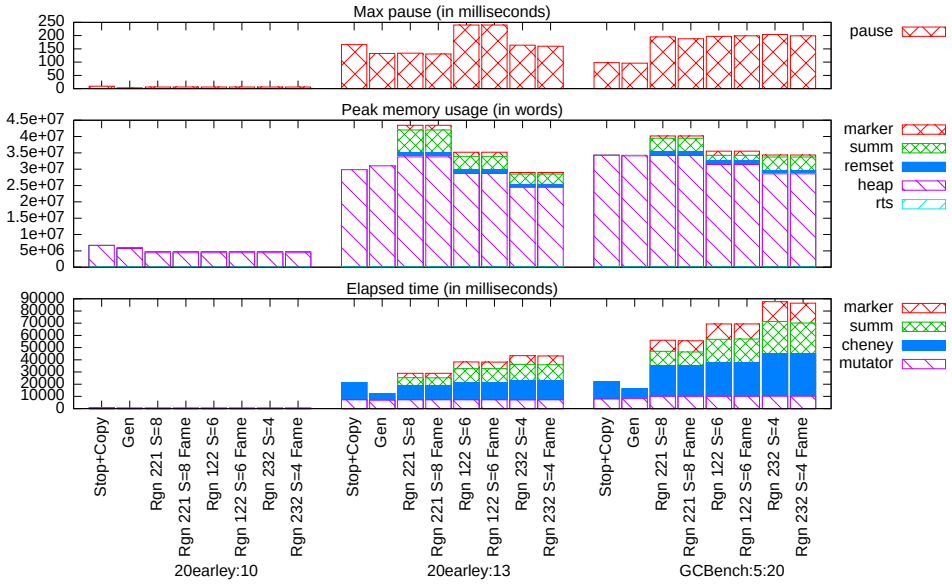


Figure 9.13: Collector comparisons, big programs 1

of memory, though the size of this meta-data still pales in comparison with the overall heap. The summarization state is largest with configuration $F_i = [2, 2, 1]$; this makes sense, because the fraction of regions with summaries is bounded from above by $1/(F_1F_2)$, which is maximal when the product F_1F_2 is minimal. With $F_i = [2, 2, 1]$, the product $F_1F_2 = 4$, while for the other two regional runtime configurations the product F_1F_2 is 4.8125 and 6, respectively. The memory saved with larger values for F_1F_2 does not come without cost; when a smaller fraction of the regions have summaries, more summarization cycles are necessary relative to the number of collection cycles, which costs more in overall elapsed time. This fact is reflected in the elapsed time chart at the bottom, where the summarize portion of the bars grows significantly for the configurations with $F_i = [1.75, 2.75, 2]$ and $F_i = [2, 3, 2]$ in the `earley:13` benchmark.

The maximum pauses for all runtimes in `earley:13` are significantly longer than those when running `earley:10`. Two configurations of the regional collector have max pause times competitive with the stop-and-copy and generational variants, while one configuration ($F_i = [1.75, 2.75, 2]$) has

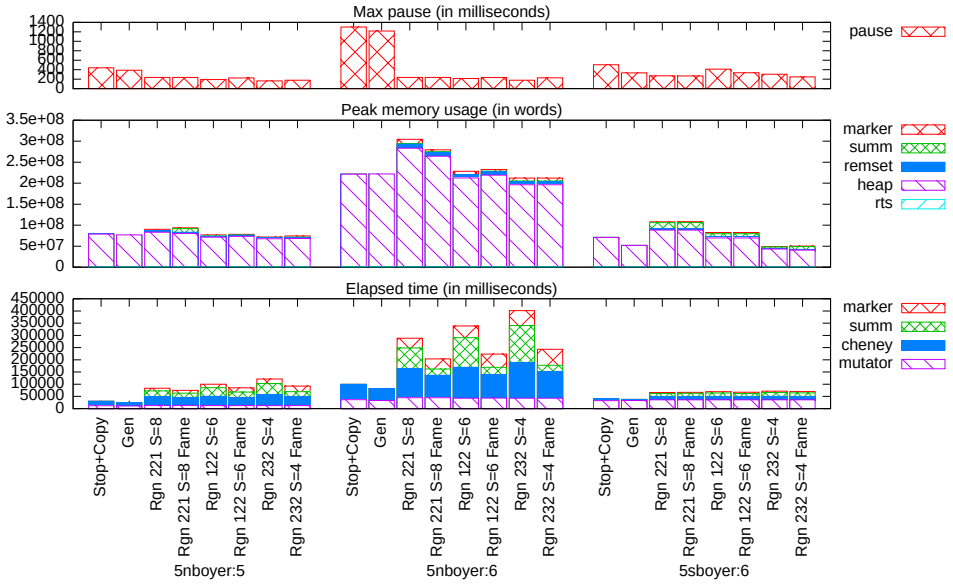


Figure 9.14: Collector comparisons, big programs 2 (`boyer` family)

a longer max pause time.

The `gcbench` benchmark data do not vary quite so much in response to changes in the runtime configuration. The maximum pause times for the regional collector are essentially the same regardless of the F_i setting. This `gcbench` data also repeats the illustration that higher values for the product F_1F_2, which reduces the fraction of summarized regions, yields an increase in the overall time spent performing summarization; however, in this case summarizing a larger fraction of the regions does not uniformly cost more as it did for `earley:13`.

The data in figure 9.13 makes it clear that effect of the fame heuristic is insignificant for these benchmarks; each column of data with fame turned on looks essentially the same as the column with fame turned off. This matches what one would predict from the instrumentation data presented in figures 7.12, 7.13, and 7.14.

Note that all of the pauses in figure 9.13 are relatively short; the y-axis of figure 9.13 only goes up to 250 milliseconds, while the other figures go up to 1400, 900, 300, and 4000 milliseconds respectively.

Figure 9.14 shows the max pause, peak memory, and overall elapsed time for `5nboyer:5`, `5nboyer:6` (the same benchmark but scaled to a larger problem size, denoted by the `:6` suffix), and `5sboyer:6` (the same benchmark, but now modified to use "sharing cons" [6]). The two `nboyer` runs heavily stress the summarization routine. As noted in section 7.4's analysis of figure 7.15, `nboyer` is a benchmark where the benefit of the fame heuristic is quite significant. These results are repeated here, as one can see by noting how the size of the green summarize elapsed-time bar drops significantly in both the `5nboyer:5` and `5nboyer:6` benchmarks.

In the `5sboyer:6` benchmark, fame does not make an impact, probably because the use of "sharing cons" implies that fewer distinct objects are generated, and so while there may be more sharing of values overall in this benchmark, no individuals become as "popular" as in `nboyer`. The maximum pause times for the regional collector remain bounded but are larger on `sboyer` than for any of the other benchmarks evaluated; I have not isolated what characteristic of this benchmark yields the increase in maximum pause time compared to `nboyer`. However, I repeat here that the goal of this design is not to provide pause times that are always shorter than what one might achieve via a stop-and-copy or generational collector, but rather to provide a bound on the pause times.

The data from the `200perm9:10:1` and `400perm9:20:1` benchmarks, presented in figure 9.15, is interesting in two ways. First, it illustrates that the maximum pause time for the regional collector remains stable even as the problem size is scaled up; note that both the stop-and-copy and generational collector's max pause time more than doubles as the `perm` benchmark is scaled up. Second, as the problem size increases, the amount of time spent doing Cheney copying collections goes up for *all* of the runtimes. This added cost is exacerbated in the regional collector, so that the blue Cheney elapsed-time bar is quite large in `400perm9:20:1`. Note that the regional collector's Cheney elapsed-time bars were already large compared to the other two col-

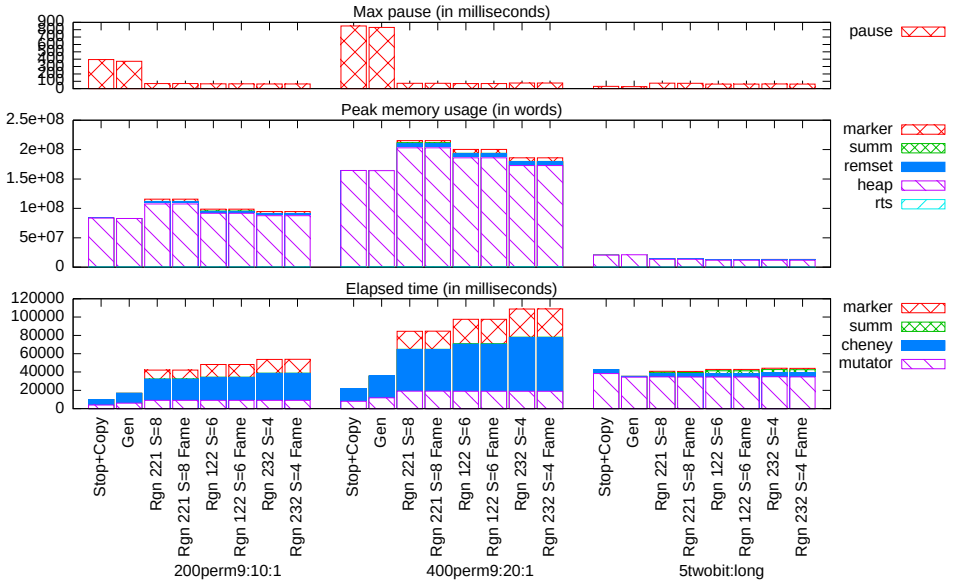


Figure 9.15: Collector comparisons, big programs 3

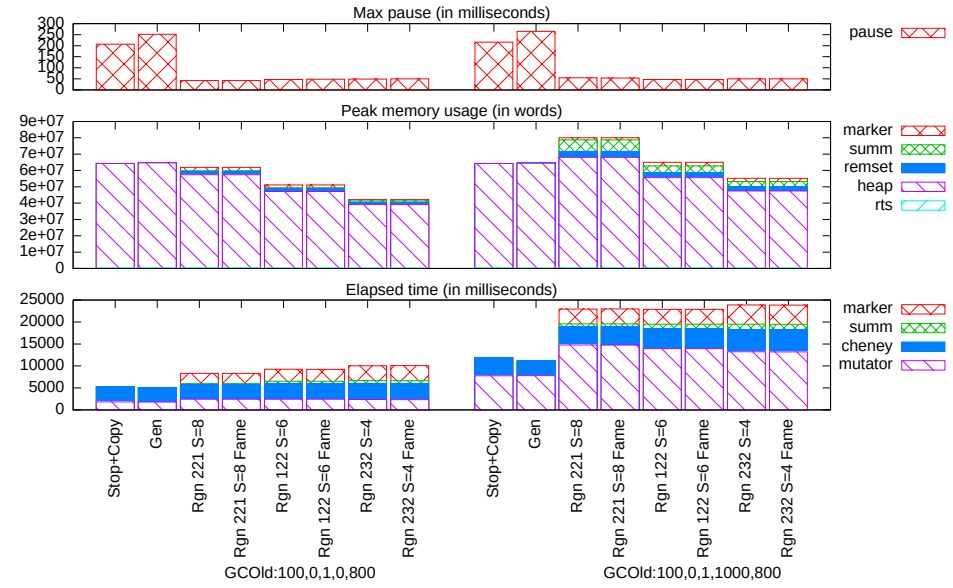


Figure 9.16: Collector comparisons, big programs 4 (`GCOld` family)

lectors for `200perm9:10:1`. This illustrates a clear cost in overall elapsed time for employing the regional collector on the `perm` benchmark.

Figure 9.16 shows the `GCOld` benchmark. On the left, the parameters are set so that the benchmark performs no mutations; here the regional collector competes well on all fronts. On the right, the benchmark performs a signif-

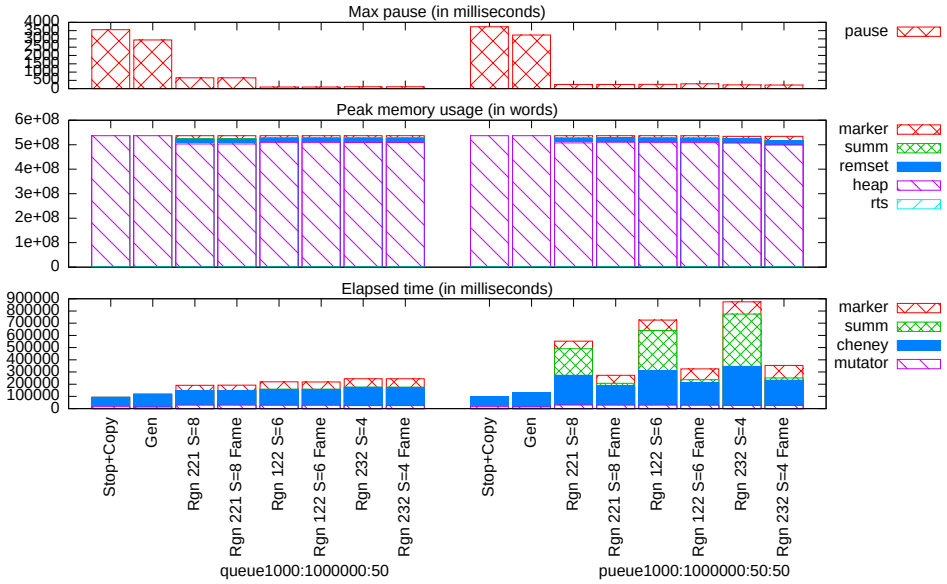


Figure 9.17: Collector comparisons, big programs 5 (`queue` family)

icant number of mutations, and this incurs significant relative overhead in memory and time for the regional collector; however, its pause times remain stable.

Finally, figure 9.17 shows the `queue` and `pop-queue` benchmarks. Once again, the regional collector's pause time remains stably bounded, at some cost in elapsed time. In the `pop-queue` benchmark, the fame heuristic makes a significant difference in overall elapsed time. This is unsurprising, since the `pop-queue` benchmark was specifically constructed to illustrate how each system deals with popular objects, and the fame heuristic is specifically devised to address that problem in the regional collector.

9.4 Big benchmarks, Observed utilization

In previous sections, I have argued that regional collectors provide worst-case bounds on MMU, and provided a high-level proof of why this should be true.

Unfortunately, my prototype, like many real-world systems, is a large and complicated code base. I cannot hope to formally prove that *it* has worst-case bounds on MMU. (Furthermore, such a proof may well fail due to bugs in the prototype.)

However, I can provide evidence for the claim that regional collectors provide worst-case bounds on MMU, by presenting the observed MMU.

In these graphs, the x-axis is the window size (on a *log scale*) and the y-axis is the minimum fraction of time that the mutator received for its own utilization. Note that the range of the y-axis does not always go to 1.

Many of these charts are not flattering towards the regional collector. The percent utilization is often quite low (MMU is a harsh metric). But it does seem that there *is* a window size where, for every benchmark, the regional collector's MMU is non-zero. Most importantly, this remains true even when the heaps get large.

We saw previously (pages 140, 142, 143) that the benchmarks with the largest memory usage are nboyer:6, perm, queue, and pop-queue. These are the important cases when asking questions about scalability; and these are the cases where the MMU for the regional collector shines; see below on pages 147, 149, and 151.

Figures 9.18 and 9.19 plot the minimum mutator utilization for the `earley` benchmark running on the different runtime configurations analyzed in section 9.2. In both cases the stop-and-copy and generational collectors exhibit better minimum mutator utilization than any configuration of the regional collector.

It is to be expected that when the stop-and-copy and generational collec-

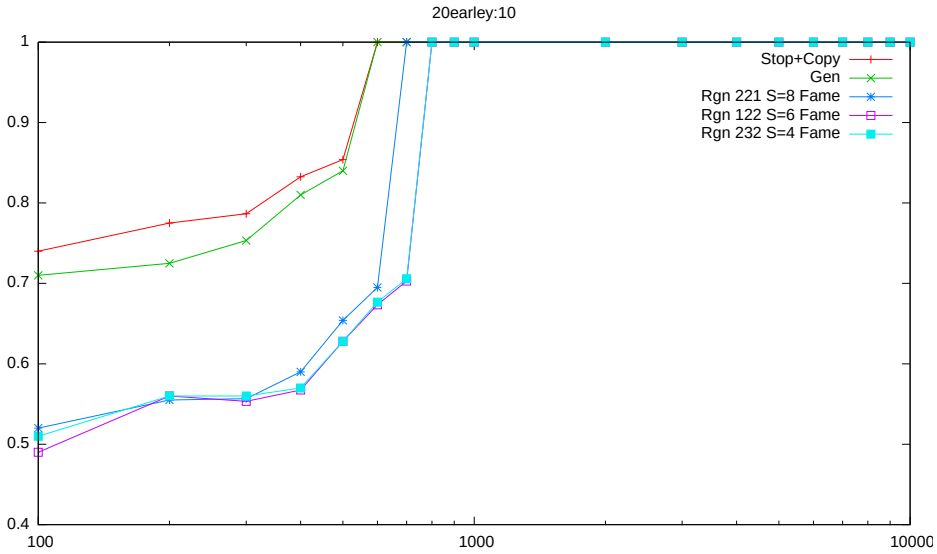


Figure 9.18: 20earley:10 MMU

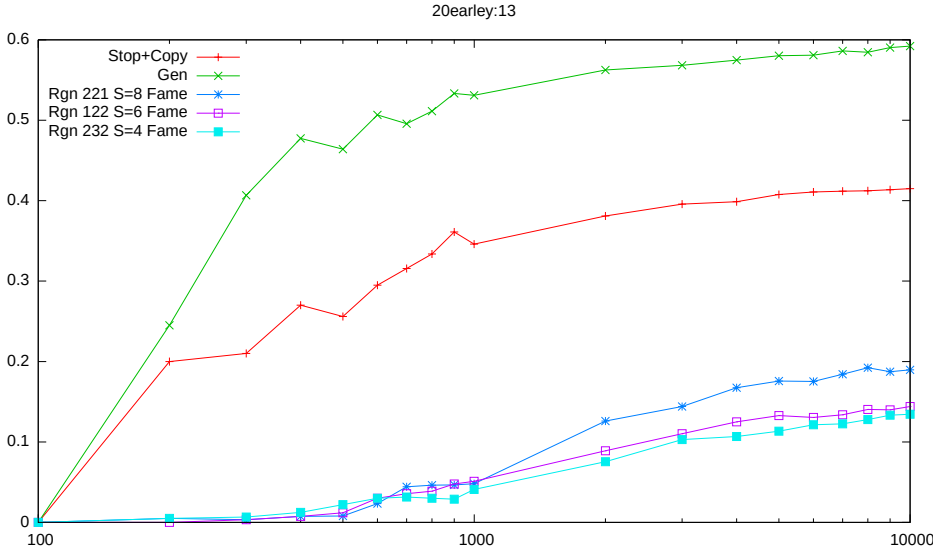


Figure 9.19: 20earley:13 MMU

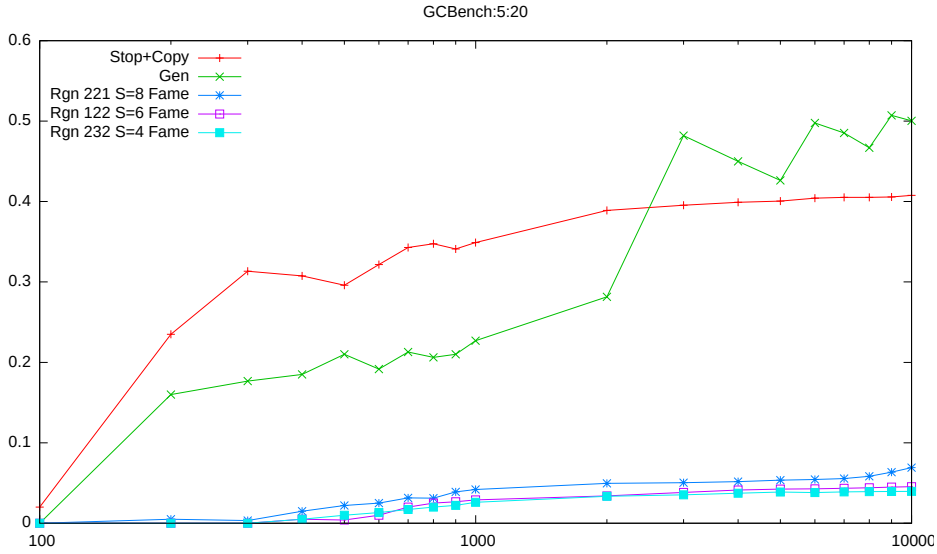


Figure 9.20: GCBench:5:20 MMU

tors have relatively small maximum pause times that they would also have quite impressive minimum mutator utilizations, because in both cases once a major collection has been completed, the heap is adjusted relative to the amount of live storage; with Larceny's inverse load factor of 3, this guarantees that there is a lot of space available for new allocation/promotion before the next major collector. Thus the pauses in both technologies are widely spread apart. If the maximum pause for a benchmark is short, then the minimum mutator utilization (for window sizes significantly larger than that maximum pause time) will be large for these two garbage collection technologies.

The regional collector also increases its maximum allowed heap size, but policies that drive the collection of regions force major collections more frequently for the regional collector than for the stop-and-copy and generational collectors.

Figure 9.20 plots the minimum mutator utilization for the `GCBench` benchmark. Again, the other two runtimes exhibit better minimum mutator utilization than any configuration of the regional collector.

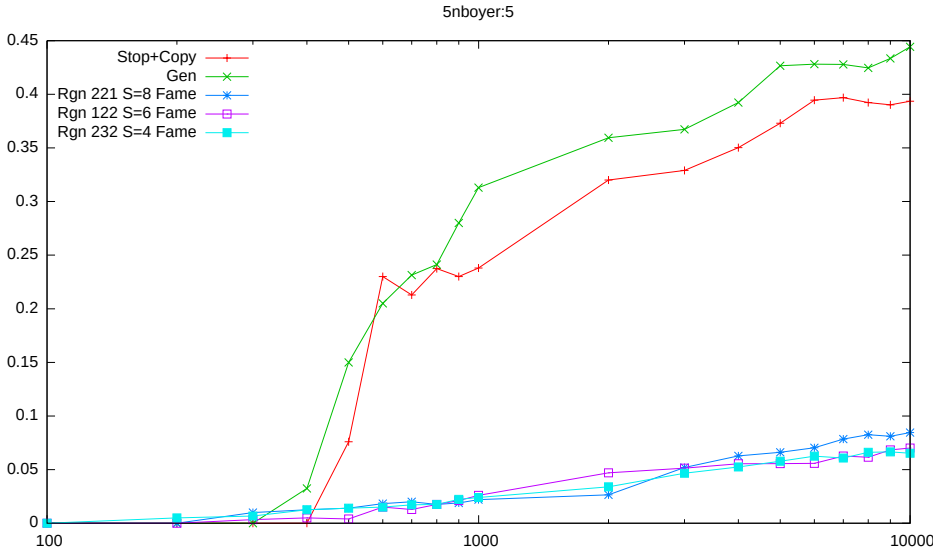


Figure 9.21: 5nboyer:5 MMU

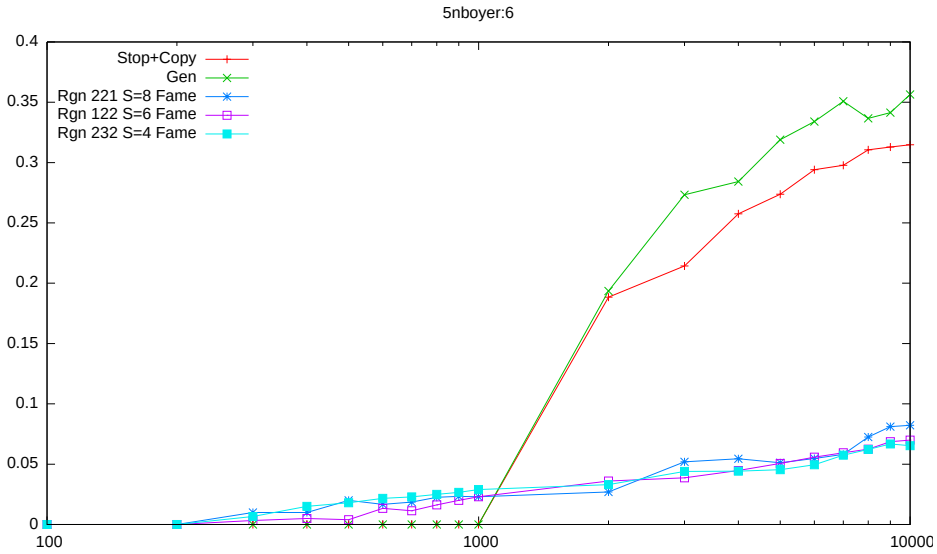


Figure 9.22: 5nboyer:6 MMU

While the regional collector has a guaranteed minimum mutator utilization, that is not a guarantee that it will outperform the other runtimes in all cases.

Figures 9.21 and 9.22 illustrate this point. They present the MMU data for the `nboyer` benchmark. In this cases, both the stop-and-copy and gen-

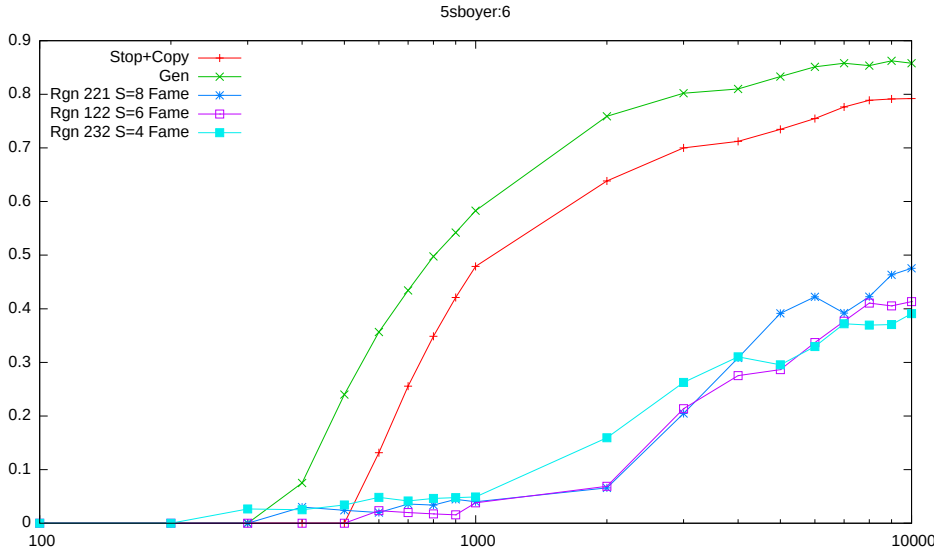


Figure 9.23: 5sboyer:6 MMU

erational collectors have *zero* utilization at small window sizes. This underscores the point that these technologies have high minimum mutator utilization only at window sizes that are significantly longer than their maximum pause times. As the maximum pause time grows, the number of window sizes that are large enough dwindles; in `5nboyer:6`, one must employ a window longer than a second before seeing non-zero minimum utilization for the two non-regional collectors. The regional collector's minimum mutator utilization curve remains consistent.

Figure 9.23 presents the minimum mutator utilization for the `5sboyer:6` benchmark. As mentioned in section 9.2, the regional collector's maximum pause times are larger for `sboyer` than for any other benchmark I evaluated. Thus its minimum mutator utilization suffers, yielding zero minimum utilization in more cases. however, the regional collector configuration with $F_i = [232]$ and $S = 4$ does achieve non-zero utilization at window size 300.

Figures 9.24 and 9.25 quite dramatically illustrate that the regional collector behaves in a consistent manner with respect to minimum mutator utilization. As the `perm` benchmark's problem size is scaled up, the two

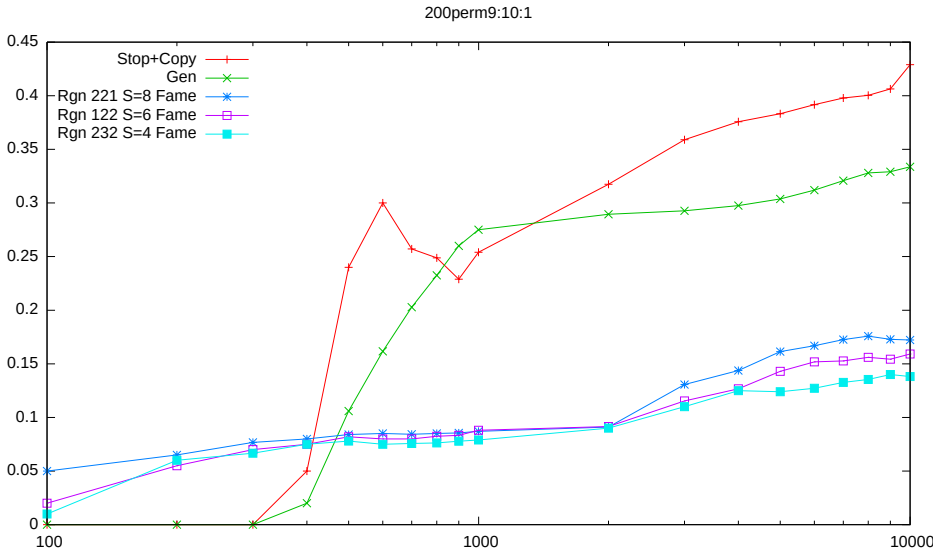


Figure 9.24: 200perm9:10:1 MMU

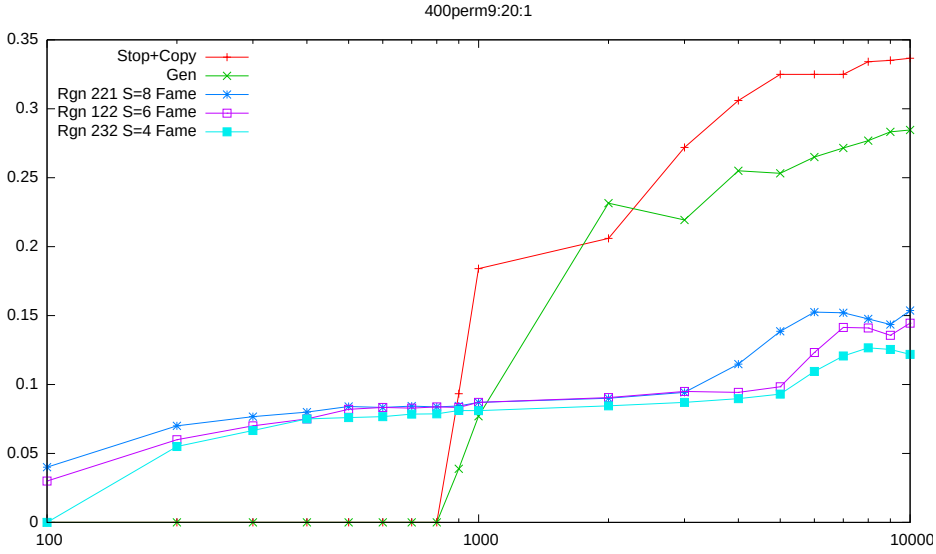


Figure 9.25: 400perm9:20:1 MMU

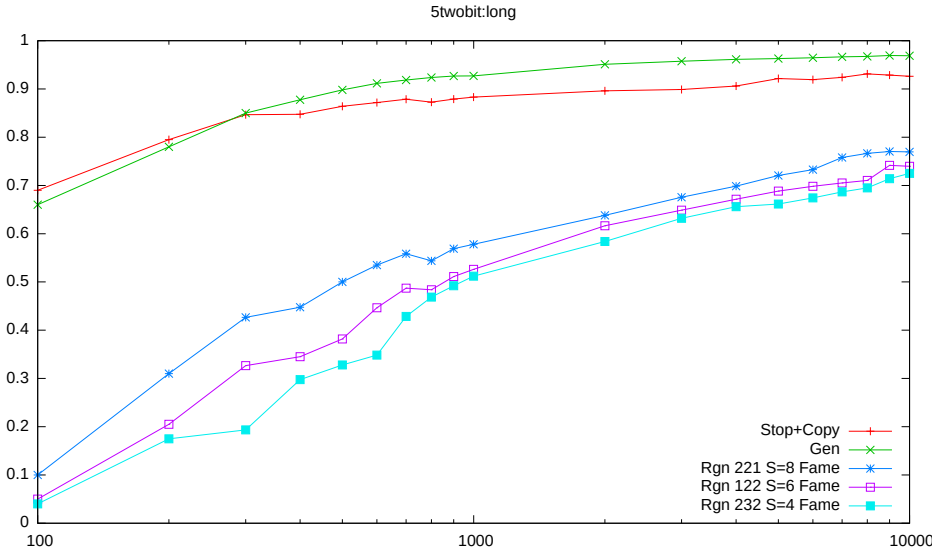


Figure 9.26: 5twobit:long MMU

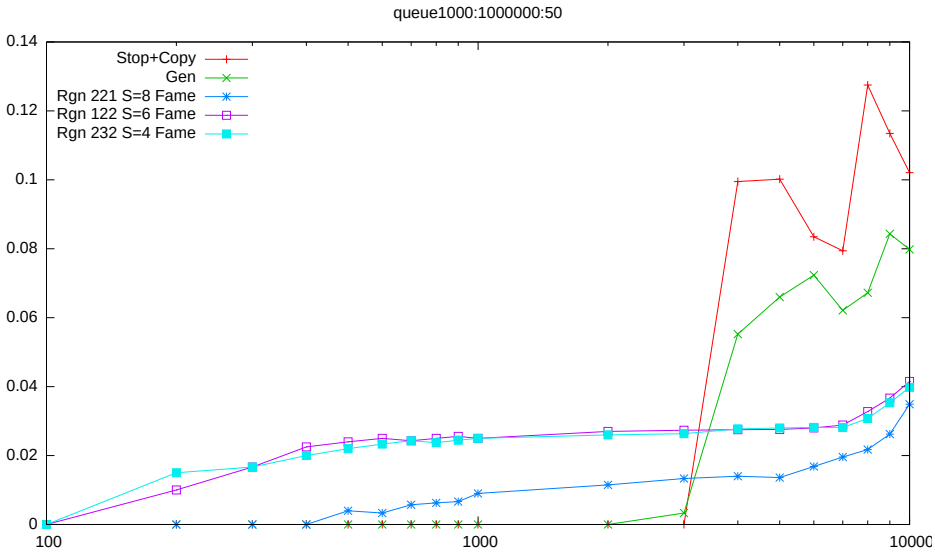


Figure 9.27: queue:1000:1000000:50 MMU

non-regional collector's both have zero minimum utilization at steadily increasing window sizes, while the curves for the regional collector remain relatively stable.

Figure 9.26 illustrates another case where the non-regional collectors exhibit high minimum mutator utilization at all window sizes.

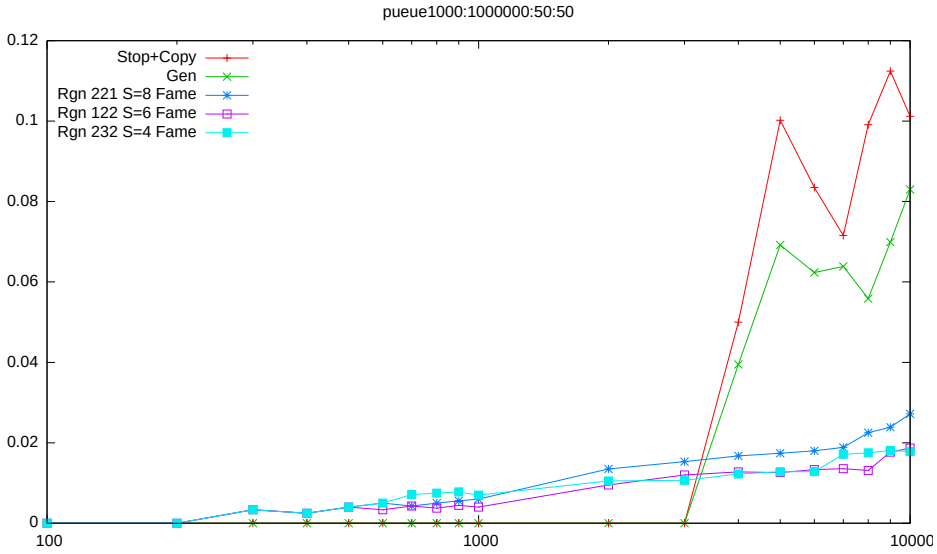


Figure 9.28: pueue:1000:1000000:50:50 MMU

Finally, figures 9.27 and 9.28 illustrate a near worst-case benchmark for the non-regional collectors; their maximum pause time is so long that they exhibit zero minimum utilization at window sizes of 2 and 3 seconds. The regional collector, on the other hand, exhibits a small but non-zero minimum utilization at several window sizes in the sub-second range.

Chapter 10

Related Work

This chapter discusses related research efforts.

10.1 Generational garbage collection

Historically the idea of generational collection was introduced by Lieberman and Hewitt [23]. A simplification of that design was first implemented by Ungar [28]. Most generational collectors implemented today are modeled after Ungar's, but our regional collector's design is more similar to that of Lieberman and Hewitt.

10.2 Heap partitioning

Our regional collector is centered around the idea of partitioning the heap and collecting the parts independently, which dates at least back to Bishop [8]; his work targets Lisp machines and requires hardware support.

The *Garbage-First* collector of [18] inspired many aspects of our regional collector. The garbage-first collector does not have worst-case bounds on space usage or pause times.

The *Mature Object Space* (a.k.a. *Train*) algorithm of [21] uses a fixed policy for choosing which regions to collect. To ensure completeness, their

policy migrates objects across regions until a complete cycle is isolated to its own train and then collected. This gradual migration can lead to significant problems with floating garbage. We use a concurrent marker to provide collection completeness and non-directional remembered sets to allow more flexible policies.

The *Beltway* collector of [9] uses a heap partitioning infrastructure to enable flexible selection of policies expressive enough to emulate the behavior of semi-space, generational, renewal-older-first, and deferred-older-first collectors. They demonstrate more flexible policy parameterization can improve significantly upon a fixed generational collection policy. Unfortunately, in the Beltway system one must choose between incremental or complete collection. Our design achieves both.

The *MarkCopy* collector of [26] partitions the heap into fixed sized *windows*. During a collection pause, it constructs precise points-into remembered sets via a whole-heap marking pass. The authors claim the system could support real-time constraints via extensions that perform the copying and the marking incrementally, but only implemented and benchmarked incremental copying.

10.3 Bounding collection pauses

There is a broad body of research on bounding the pause times introduced by garbage collection, including [2, 5, 7, 11, 12, 20, 24, 30]. Several attempts to reduce pause times run afoul of the problem that bounding an individual pause is not enough; one must also ensure that the mutator can accomplish an appropriate amount of work in between the pauses, keeping the processor utilization high.

Blelloch and Cheng [10, 15] describe a real-time concurrent copying collector with proven bounds on pause times and space usage, and also introduce the notion of minimum mutator utilization as a metric for evaluat-

ing how much progress the mutator can make concurrently with collection. They report that supporting parallelism adds 39% overhead to the collection time and supporting real-time constraints adds an additional 12% overhead.

Metronome [4] is a hard real-time collector. It is mostly non-moving, but will copy objects to reduce fragmentation. Metronome requires a read barrier, but its developers managed to reduce the read-barrier overhead to an impressive average of 4%. Metronome must also be provided with application's maximum memory consumption and maximum allocation rate in order to meet its bounds; if these *mutator-specific* parameters are underestimated, Metronome may exceed its predicted time and space bounds.

In contrast to Metronome, our regional collector is mostly copying and has no read barrier. It does not make any hard real-time guarantees, but provides scalability guarantees independent of the mutator. Our collector is a different point in the design space. Our collector's worst-case guarantees are independent of the mutator; Metronome, however, must be tuned to a particular mutator, and its worst-case guarantees hold only for the mutator to which it is tuned. For any fixed mutator, I would expect Metronome's worst-case guarantees to be better than our mutator-independent guarantees, but deriving Metronome's mutator-dependent parameters with sufficient precision represents a significant burden to a developer using Metronome.

A fairer comparison would be to employ a hypothetical set of parameter values that are sufficiently conservative to allow Metronome to support an arbitrary application, thus providing a mutator-independent configuration. Metronome's worst-case performance under such a configuration has not been described in the literature, and it is not even clear whether a sufficiently conservative set of parameters can be found.

On typical programs, for which our regional collector's performance is nearly indistinguishable from that of a conventional generational collector, it should deliver better throughput than Metronome. But the most important point is that our regional collector imposes no requirement on the client

to estimate memory consumption nor allocation rate, since its scalability guarantees are mutator-independent.

10.4 Concurrent collection

There are many treatments of concurrent collectors dating back to [19], which specifically points out how difficult they are to implement correctly. In our collector, reclamation of dead object state is not performed concurrently. We believe this makes our design significantly easier to understand and implement, *because* it is not fully concurrent.

Enabling concurrent summarization of the portion of the remembered set relevant to scheduled collections was inspired by the performance of Detlefs' concurrent refinement of the remembered set to reduce time spent scanning objects during collection pause [17].

Extending our design to use concurrent processes to remove some scanning and tracing work from the critical path of the collector requires a write barrier. Larceny already inserts a write barrier to support its generational collector; we piggy-back the new regional write barrier atop the generational barrier already present. This is similar to how [25], building on the work of [11], merges the overhead of maintaining concurrency-related invariants with the overhead of maintaining generational invariants.

Another mark-sweep collector that uses a variant of snapshot-at-the-beginning is [3]; in that context, snapshot-at-the-beginning removes the need for a global stop-the-world pause, which is useful in the context of systems with many mutator threads. We use a snapshot-at-the-beginning marker to ensure collection completeness; Sun's garbage-first collector shows that collectors similar to ours can still support a reasonable number of mutator threads.

Chapter 11

Future Work

This chapter discusses ideas for future research.

11.1 Distributed collection

This document presented a prototype regional collector that performs summarization and marking in an incremental fashion; it did not make use of parallel processors. Chapter 8 discussed some ideas for how that extension would work.

A step beyond the presentation of Chapter 8 would be to see if the ideas here are applicable in a distributed garbage collection setting. The collection of any particular region is a local process, which is a good sign. Building a complete summary is global, but the scan of each region when doing such construction appears to be local. Two interesting questions are: (1) how to efficiently communicate the results of summarization between the distributed nodes, and (2) how to perform the snapshot-at-the-beginning marking in a distributed fashion. In fact, the latter problem of the marker seems like it is another instance of the original problem: How to perform distributed garbage collection. Perhaps the real question is whether there would be a benefit to layering a regional collection scheme on top of a pre-existing distributed garbage collector.

11.2 Operating system integration

In the prototype presented here, the region meta-data was maintained entirely by the runtime system, without any support from the virtual memory subsystem of the operating system. This was a design decision: the Larceny runtime is portable across multiple operating systems, and we wanted to maintain that when adding the regional collector.

Would there be any benefit to abandoning that design constraint, and tying the regions of this collector into the virtual memory subsystem? For example, would write-protecting the pages associated with a region undergoing a summarization scan allow a more efficient write-barrier? Appel reports retrospectively [1] that repurposing virtual memory functions for this may not work as well as one would hope, but perhaps just isolating the repurposing to the summarization or snapshot marking would be reasonable.

11.3 Language integration

This research focused on changing the garbage collector in a virtual machine. Our prototype test-bed was a Scheme runtime, but the ideas apply more broadly.

One interesting research avenue would be to extend the programming language by exposing the notion of regions at a level visible to the application programmer.

For example, the programmer may have knowledge about what classes of objects are likely to be popular. A programming language feature that allowed such objects to be statically annotated as “popular,” and subsequently kept isolated to a distinct region that the collector classifies as popular at the outset, may allow for more efficient overall execution of the regional collector, since it would not waste time building summaries for regions that

are likely to be waved off. This sounds similar to generational systems that allow pre-tenuring of objects.

Another example: The reference structure of the heap may follow patterns that can be statically described. For example, objects of class X may have references to those of class Y, but never in the other direction from Y to X. If that information were fed into the regional collector, it could potentially allocate each class to its own region, and subsequently choose an ordering for region summarization and/or collection that would avoid processing the references into objects of class Y until after the objects of class X have been processed. Such effort may benefit from integration with region-based[1] memory-management schemes [27].

11.4 Tool integration

If we keep the programming language unchanged, the regional collector could be integrated into the development tools for that language. For example, the summarization process is building up a structure of incoming references to a region; knowledge of that structure, such as the origins of the references (or even just a count of incoming references), could be useful information to provide to a developer profiling an application's behavior.

[1] The similarity in terminology is a historical accident; Tofte and Talpin's technology is unrelated to that presented here.

Chapter 12

Conclusion

I have described and prototyped a regional collector, which is a new kind of generational garbage collector.

I have proved that the regional collector is scalable: It guarantees worst-case bounds for gc latency, minimum mutator utilization, and space usage, independent of the peak live storage and mutator behavior.

The regional collector incorporates novel and elegant solutions to the problems presented by popular objects and floating garbage. The fame heuristic introduced in chapter 7 handles one common source of remembered-set size blowup.

The overheads introduced by the incremental summarization and marking coroutines are not insignificant. The regional collector was explicitly designed to allow these processes to be offloaded onto additional processor cores. Future research efforts should investigate what effect that change would make.

I have prototyped the regional collector and analyzed its behavior on a large set of benchmarks, both gc-intensive and not, to illustrate its performance. The gathered data show that on non gc-intensive benchmarks, the regional collector performs as well as a generational collector, and on gc-intensive benchmarks, the regional collector meets its scalability goals while remaining competitive with a stop-and-copy collector.

The regional collector maintains a bounded pause time and bounds the amount of memory overhead beyond the heap structure. Since the regional collector also ensures that the mutator time between pauses is bounded from below, this ensures that its minimum mutator utilization is also bounded from below.

Regional garbage collection with summarization, wave-off, and snapshot refinement, provides mutator-independent worst-case bounds on pause times and minimum mutator utilization, and provides competitive throughput while maintaining a worst-case bound on overall memory usage.

Bibliography

[1] Andrew W. Appel. Retrospective on real-time concurrent collection on stock multiprocessors. *SIGPLAN Not.*, 39:205–216, April 2004.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *ACM SIGPLAN Notices*, 23(7):11–20, 1988.

[3] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[5] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[6] Henry G. Baker. The Boyer benchmark at warp speed. *SIGPLAN Lisp Pointers*, V(3):13–14, August 1992.

[7] Henry G. Baker. The Treadmill, real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[8] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, MIT Laboratory for Computer Science, May 1977. Technical report MIT/LCS/TR–178.

[9] Stephen M. Blackburn, Richard Jones, Kathryn S. McKinley, and J. Eliot B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 153–164, Berlin, June 2002. ACM Press.

[10] Guy E. Blelloch and Perry Cheng. On bounding time and space for multiprocessor garbage collection. In *Proceedings of SIGPLAN'99 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 104–117, Atlanta, May 1999. ACM Press.

[11] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *ACM SIGPLAN Notices*, 26(6):157–164, 1991.

[12] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[13] David R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[14] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[15] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *Proceedings of SIGPLAN 2001 Conference on Programming Languages*

Design and Implementation, ACM SIGPLAN Notices, pages 125–136, Snowbird, Utah, June 2001. ACM Press.

[16] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. In *Proceedings of SIGPLAN'97 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 97–108, Las Vegas, Nevada, June 1997. ACM Press.

[17] David Detlefs, William D. Clinger, Matthias Jacob, and Ross Knippel. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '02)*, San Francisco, CA, August 2002.

[18] David Detlefs, Christine Flood, Steven Heller, and Tony Printezis. Garbage-first garbage collection. In Amer Diwan, editor, *ISMM'04 Proceedings of the Fourth International Symposium on Memory Management*, Vancouver, October 2004. ACM Press.

[19] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[20] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[21] Richard L. Hudson and J. Eliot B. Moss. Incremental garbage collection for mature objects. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, University of Massachusetts, USA, 16–18 September 1992. Springer-Verlag.

[22] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[23] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[24] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[25] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In Tony Hosking, editor, *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, volume 36(1) of *ACM SIGPLAN Notices*, Minneapolis, MN, October 2000. ACM Press.

[26] Narendran Sachindran and Eliot Moss. MarkCopy: Fast copying GC with less space overhead. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[27] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 188–201, New York, NY, USA, 1994. ACM.

[28] David M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the

ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 157–167, April 1984.

[29] Paul R. Wilson and Thomas G. Moher. A card-marking scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *ACM SIGPLAN Notices*, 24(5):87–92, 1989.

[30] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.