CHAPTER 4

EVALUATION: SETTING

The evaluation of garbage-collection algorithms requires us to pose a large number of questions in order to expose the strengths and weaknesses of algorithms in different circumstances. We developed a number of different experimental methods to answer them and we present these methods in detail here and in the following two chapters: the discussion of each method is immediately followed by a presentation of the experimental results.

The three main approaches to a comparative performance evaluation of garbage collector performance, offering different perspectives, are analytical modelling, trace-based simulation, and prototype implementation. *Analytical modelling* consists of constructing a mathematical description of the operation of the garbage collection algorithm, and applying it to a mathematical description of the load, i.e., the allocation and lifetime properties of objects and pointers. When successful, analytical modelling provides succinct models, and has useful predictive power with respect to the choice of collector configuration. Unfortunately, the premise of having reliable realistic mathematical descriptions of lifetime properties of objects, and especially properties of pointers, is not easily met with current knowledge.

Trace-based simulation consists of constructing a software simulation of the operation of the garbage collection algorithm, and applying it to an empirically obtained trace of the load, i.e., the sequence of allocations and other object-related operations of a real program. Simulation isolates the memory management tasks from other (computational) tasks that a real program performs, and allows a variety of configurations to be tried. It is even possible to try collection algorithms that would be excluded in practice, such as those using backtracking, or oracles, in their decisions. The analysis of simulation results provides explanatory and predictive power, but is not as succinct as analytical modelling. It is usually not difficult to extract traces when a working garbage collector is given, thus one can reasonably obtain traces from more than one object system and analyze them within the same simulation framework.

Prototype implementation consists of constructing complete garbage collection algorithms and implanting them in a working object-based system, in order to measure how each performs in operation. Implementation has the advantage that it accounts for the full cost of memory management; on the other hand, it is difficult to extract the contribution of each cost factor. Furthermore, that cost is necessarily a reflection of a number of implementation decisions. From a practical standpoint, it is a usually a difficult task to retrofit (a number of) garbage collectors onto a working system.

Our study primarily uses trace-based simulation using specially built object-level simulators. A prototype implementation was also completed according to the design of Section 3.2, and integrated with a trace simulator, which allows us to gather many of the performance statistics as in a working object system.

In the remainder of this chapter, we discuss our experimental setting: the design of the garbage collection simulators and the set of benchmark programs used. The following two chapters present the findings: Chapter 5 examines the copying cost of collection, and Chapter 6 examines non-copying costs of collection, viz., the cost of collector invocation, and the pointer management costs.

4.1 Evaluation using program traces

The primary method used in this study is evaluation by trace-based simulation. The experiment proceeds in two phases. First, by instrumenting an object-based system and executing a program, a trace is produced. Second, the trace is used to direct the execution of different garbage collection algorithms and to derive statistical measures of the program.

4.1.1 Trace content

The information contained in the trace must permit accurate simulation of each garbage collection algorithm. To the point, the trace must identify all objects that the program allocated in the heap, providing for each object its time of allocation, its time of demise, and its size.[1] Furthermore, all stores of pointer values into the objects in the heap must be present in the trace.

This information is sufficient for a garbage collector simulator to determine which objects are reachable in any chosen collection area at any chosen time. A real collector first identifies global roots by scanning the stack, and then computes the transitive closure of the points-to relation to find the reachable, hence live, objects. As it will be described below, the simulated collector relies instead on accurate allocation and demise records: all objects for which an allocation record has been seen but not a demise record, are live. In addition, the trace allows the collector to maintain an accurate image of the pointers among heap objects. In particular, the pointers from the uncollected region into the collected region are accurately known. Thus the closure of the points-to relation can be computed in the simulator to complete the set of collection survivors.

4.1.2 Trace format

The common format for the traces is as follows. Each trace is a text file, in which each line records an event. Three kinds of events are represented: object allocation (A-records), object demise (D-records), and field update (U-records). An allocation record has the form:

`A` *allocation-address* *size*

where *allocation-address* is the hexadecimal byte-address of the starting word of the object, and size is the number of words in the object. A demise record has the form:

---

[1]These are the properties meaningful in *every* object-based system. In some systems it may be worthwhile to report more detailed information, when it is defined in the language at hand, such as the type (or class) of the object, or the allocation point. However, since age-based garbage collection algorithms do not make use of such information, we shall assume that only the general properties, applicable to all object-based systems, are recorded.

D *allocation-address*

where *allocation-address* is the hexadecimal byte-address of the starting word of the object. An update record has the form:

U *field-address* *value-address*

where *field-address* is the hexadecimal byte-address of the field which is being updated, and *value-address* is the pointer value stored. (The special value -1 represents the store of a *null pointer*.) The form of the traces reflects the assumption that an object is a contiguous sequence of fields, each field being an integer number of words. A pointer-valued field takes one word. A word is the smallest quantum of allocated space, and therefore all sizes and amounts are reported in words. In the systems studied (Smalltalk and Java 32-bit implementations), a word is four bytes long. Another assumption implicit in the trace format is that the location of an object is immutable. This assumption was made for simplicity: rather than have a separate notion of object identity, the address (whether allocation address or a field address within the object) uniquely identifies the object. Without loss of generality, it may also be specified that allocation addresses of objects in the trace follow consecutively, as if the objects were allocated from an infinite allocation area starting at zero.

4.1.3 Trace generation

Having described the content of the trace, let us consider how to generate a trace. For this study, the method is to augment an existing object-based system and its garbage collector with code to produce the required trace records. The existing system is assumed to have the functionality to identify each object allocation, object demise, and pointer store.

Every object-based system can identify object allocations.

Every system with a garbage collector can also identify object demise. However, if we rely on the garbage collector existing in the system (often a state-of-the-art generational collector), we must ensure that the collections consider the entire heap, as in a non-generational collector, to avoid the inaccuracies of excess retention (Section 5.3).[2] Following each such full collection, the reachability of objects is accurately known; any objects that are now not reachable, but were reachable at the previous full collection have died in the intervening period. If the time of demise must be known with accuracy, the period between full collections must be short. In fact, it is critical for the correctness of *simulated* collections to be able to determine reachability at every possible garbage collection instant (in simulation), i.e., *following every object allocation* (in simulation). Therefore, a full collection (in trace generation) must also follow every object allocation. As a result, the trace generation mechanism is very expensive: full collections are expensive, and many are needed. In fact, the cost is roughly proportional to the integral $\int_0^T v(t)dt$ of the live data amount over the duration of the program.[3]

Update records, the final component of the trace, are easily generated by instrumenting the write barrier in a system with a generational collector, since the collector must be informed of pointer stores. One must only ensure that in the existing system the write barrier is indeed invoked for each pointer store, that is, the system does not optimize away some pointer stores (e.g., initializing stores or null pointer stores), which it is allowed to do for operation of a generational collector, but not for trace generation.

Let us consider some questions about traces generated in this manner. Any measurement disturbs the system measured, and generating object traces is no exception. If the traced program relies on timing properties, such as by querying a real-time clock, its behavior will be severely disturbed by the heavy overhead of tracing. We do not have such programs among our benchmarks. If an application is multi-threaded, as many interactive Java programs are, and

[2]It is additionally assumed throughout that the collector is accurate (as opposed to conservative). It is certainly possible to obtain traces with a conservative collector, such as the one found in the Kaffe virtual machine for Java, but the utility of such imprecise traces is dubious: the garbage collection schemes in this study are meant to be accurate, not conservative.

[3]Note that the underlying assumption is that the existing object-based system and its garbage collector are to be exploited with minimally intrusive modifications—as dictated by technical considerations when dealing with large and complex systems, and also dictated by legal and commercial considerations when systems are not available for inspection and modification in their entirety. If, on the other hand, a system can be redesigned for the express purpose of generating our traces, then better solutions are possible than repeated full collections, in the form of incremental computation of object reachability.

if garbage collection requires synchronization of all threads, then the concurrency behavior of the application will be changed under tracing. The observed order of execution will be a valid order, but not a likely, or the "intended" order.

Finally, recall that in the common trace format the object addresses are immutable. If the tracing object-based system uses moving collection, it is necessary, but easy, to translate object addresses into the absolute form.

4.2 Benchmarks and languages

We first consider the object-based systems ("languages") used in the study, and then the benchmark programs, detailing their general properties.

4.2.1 Languages

We used two object-based systems. The first, **Smalltalk**, is an older purely object-oriented language [Goldberg and Robson, 1983], which consists of three parts: the virtual machine, implemented in an earlier effort in the Object Systems Laboratory [Moss, 1987; Hudson *et al.*, 1991; Hosking *et al.*, 1992], the basic virtual image of Smalltalk-80 from PARC Place/Digitalk, and the additional classes and methods of the benchmark programs. The virtual machine includes the language-independent garbage collector toolkit, also developed in the Object Systems Laboratory [Hudson *et al.*, 1991]. The virtual machine provides the execution (by interpretation) of the bytecodes in the methods contained in the basic virtual image and the benchmark classes.

In the Smalltalk system, all of these classes and methods are present as objects in a benchmark virtual image, which is loaded at program start-up to build an initial heap state. When generating the trace, we distinguish between these objects and objects subsequently allocated as a result of program execution. The main results reported here reflect the program execution objects alone. This choice is justified by the observation that image objects could better be treated as static, instead of as part of the garbage-collected heap [Clinger and Hansen, 1997].

Another property of this implementation of Smalltalk is that method activation records (frames) are internally treated somewhat like objects [Moss, 1987]. However, frames are not treated as objects for the purposes of trace generation.

The second system used, **Java**,[4] is a contemporary object-oriented language that has found much use in Internet-based downloadable applications. It consists of a virtual machine, and a set of basic and benchmark classes and methods, loaded on demand. The Java virtual machine modified to generate traces for this study is an experimental machine, called the ExactVM, developed by Sun Microsystems Laboratories, which is equipped with a garbage collector that satisfies all the requirements outlined above. The Java virtual machine used has the option of running as interpreter, or as a just-in-time compiler. The same objects are allocated, and in the same order, in either case, and pointer updates, and, consequently, object demise points, also happen in the same order—as long as the compilation does not interleave the compiled code from successive bytecodes. Since any differences in behavior, if at all present, are incidental and not material to this study, they can be ignored. The interpreted mode of operation was then chosen for reasons of simplicity.[5]

4.2.2 Benchmarks

The selection of benchmark programs for evaluating garbage collector performance is not an easy task. There is no widely-accepted suite of benchmarks, unlike for general computer performance evaluation, where there are several (e.g., SPEC95). It was therefore necessary to collect and select useful benchmarks from various sources, and it was also necessary to develop the selection criteria.

---

[4]Java is a trademark of Sun Microsystems.

[5]To generate traces using the just-in-time compilation mode, it is necessary to modify the code-generation phase of the compiler so that it will produce appropriate code at object allocations and at pointer stores. Our experience with the Kaffe virtual machines shows this to be a relatively straightforward exercise if the compiler is cleanly written. It is, however, a futile exercise, because the performance advantage of compiled code is entirely annulled by the overhead of trace generation.

4.2.2.1 Collecting benchmarks

Collecting a large preliminary set of benchmarks posed distinct challenges in both languages under consideration. For Smalltalk, although the language is no longer in vogue, a number of public-domain programs (i.e., classes) implementing interesting applications exist and can be obtained on the Internet. Unfortunately, many of these programs depend on the presence of some vendor-specific classes, or are otherwise incompatible with the virtual image used in our Smalltalk-80 system. To the programs we could find, we added some that we have written ourselves and used previously in a study of Smalltalk garbage collection performance [Hosking *et al.*, 1992]. We also translated two SPECfp95 programs from FORTRAN into Smalltalk.

That version incompatibilities should be encountered in Smalltalk, a language which in its present form dates back eighteen years, is perhaps not surprising. However, we found that significant incompatibilities exist even in Java, a much younger language. They are primarily caused by changes in the class library between versions 1.0 and 1.1, assumed by most applications, and version 1.2, which the tracing Java virtual machine uses. A more significant difficulty in collecting Java benchmarks is the fact that the vast majority of publicly available Java code is in the form of Java *applets*, partial programs which execute under the control of a Web-browser and depend on it for the user interface. A browser must contain a Java virtual machine, but that machine is not accessible for instrumentation. We can only use full-fledged Java applications, which run directly on a virtual machine. It is perhaps not a great loss, since the majority of observed applets appear to be graphical widgets and similar toy programs, which neither run long nor allocate much data. On the other hand, the resulting benchmark set is skewed in favor of language-processing tools (compilers and optimizers). Additionally, we had access to the traces (although not the sources) of a preliminary set of Java programs considered for inclusion in the SPEC JVM98 suite.

4.2.2.2 Selecting benchmarks

Having collected a larger set of programs (circa 30 Smalltalk and 15 Java) we considered which objective criteria should decide their utility as benchmarks of garbage collector performance. The first criterion must be that the program exercises the memory management functions at all. For instance, some Java graphics applications are CPU-intensive, but allocate a negligible amount of data, and are of no use in this study.

Even programs that do allocate an appreciable amount of data do not necessarily exercise the garbage collector. Many programs, in particular most of the SPEC JVM98 programs, display a monotone-increasing heap profile: they slowly allocate pieces of some large data structure from start of execution to the end. Since the heap must be at least as large as the maximum amount of live data ever in it (in our cost model and simulation, the heap size is fixed), the program never fills the heap and the garbage collector never runs. Thus, loosely speaking, we need the criterion that the garbage collector runs at all.

In fact, we need a stronger criterion. Consider that, over the execution of the program, the garbage collector will run at different moments depending on the garbage collection algorithm. The amount actually live depends, somewhat unpredictably, upon the precise moment of collection. (Perhaps the most obvious effect is at the end of program execution—since no copying cost is charged past the last collection.) It is thus necessary to reduce the "random" variation in measured collection performance that the "random" collection moments introduce, so as to expose underlying trends. It is therefore desirable to have a large number of collections to even out the "randomness"; in other words, it is desirable to have benchmarks that allocate a very large amount of data relative to their average *live* amount.

We can formalize this requirement by observing that the number of collections performed by a non-generational collector is approximately proportional to the ratio of the amount allocated to the average live amount (see below). We require this number to be more than 10 to admit a measurement. Furthermore, we admit a benchmark only if admissible measurement of non-generational collection is possible for a heap size three times larger than minimum. Thus, if a benchmark is admitted, the range of admissible heap sizes is at least 1:2, non-generational collector requires more than 10 collections, and our investigated schemes typically require hundreds of collections, so the aforementioned "randomness" is largely avoided.

4.2.2.3 The final benchmark set

Our selection process resulted in a set of 11 Smalltalk benchmarks and 3 Java benchmarks. Their basic properties are listed in Table 4.1. Execution time was measured in normal operation, without any instrumentation and with default generational garbage collector configurations, on a 50 MHz SPARCstation 20 workstation. The amount of data allocated by the program is expressed in words (each word being 4 bytes). The remaining columns give the number of objects allocated, the maximum live amount in words (which is also the minimum required heap size to execute the program), and the number of pointer stores.

The results of non-generational collection for these programs are reported in Figures 4.1–4.10, pp. 71–80 (at the end of this chapter).

Let us take as an example Figure 4.5. In the graph in Figure 4.5(a), we have plotted the absolute mark/cons ratio $\mu_{\text{non-generational}}$ against total heap size V. It is easy to check that the shape of the μ curve closely fits the theoretical prediction, $\mu_{\text{non-generational}} = \frac{1}{\frac{V}{v}-1}$ [Appel, 1987; Jones and Lins, 1996], where v is the effective live data amount. In the graph in Figure 4.5(b), we have plotted the number of garbage collections n against total heap size V. The horizontal dotted line is drawn at $n = 10$, thus the region below the horizontal dotted line (and to the right of the dotted vertical) is the region of heap sizes so large that they result in 10 or fewer non-generational collections. Because of inaccuracies that may be introduced under such conditions, we do not compare results for this region.

We now describe individual benchmarks, providing where possible details of their structure.

Table 4.1. Properties of benchmarks used

Benchmark	Ex. time (s)	Words alloc.	Objects alloc.	Max. live	Ptr. stores
Smalltalk					
Interactive-AllCallsOn	1.91	18 359	1 722	627	855
Interactive-TextEditing	2.30	22 747	3 556	1 098	3 509
StandardNonInteractive	6.11	204 954	46 157	1 160	7 251
HeapSim	67625	3 791 306	1 084 650	93 026	30 298
Lambda-Fact5	6.88	277 940	53 580	6 295	91 877
Lambda-Fact6	55.86	1 216 247	241 864	13 675	404 670
Swim	11.65	1 533 642	444 908	17 542	134 355
Tomcatv	3.92	2 117 849	624 612	30 593	286 032
Tree-Replace-Binary	3.32	209 600	35 785	9 784	39 642
Tree-Replace-Random	4.55	925 236	189 549	13 114	168 513
Richards	10819	4 400 543	652 954	1 498	763 626
Java					
JavaBYTEmark	353.76	1 161 949	109 896	59 728	49 061
Bloat-Bloat	387.54	37 364 458	3 429 007	202 435	4 927 497
Toba	178.05	38 897 724	4 168 057	290 276	3 027 982

- Interactive-AllCallsOn and Interactive-TextEditing. Two tests from the standard sequence specified in the Smalltalk-80 image [Goldberg and Robson, 1983], comprising only the macro-tests. Previously used in Ref. [Hosking *et al.*, 1992].

- StandardNonInteractive. A subset of the standard sequence of tests as specified in the Smalltalk-80 image [Goldberg and Robson, 1983], comprising the tests of basic functionality.

- HeapSim. Program to simulate the behavior of a garbage-collected heap, not unlike the simplest of the tools used in this study. It is however instructed to simulate a heap in which object lifetimes follow a synthetic (exponential) distribution, and consequently the objects of the simulator itself exhibit highly synthetic behavior.

- Lambda-Fact5 and Lambda-Fact6. An untyped lambda-calculus interpreter, evaluating the expressions 5! and 6! in the standard Church numerals encoding [Barendregt, 1984, p.140]. Previously used in Ref. [Hosking *et al.*, 1992]. Both input sizes are used in order to explore the effects of scale.

- Swim. The SPEC95 benchmark `102.swim`, translated into Smalltalk by the author: shallow water model with a square grid.

- Tomcatv. The SPEC95 benchmark `101.tomcatv`, translated into Smalltalk by the author: a mesh-generation program.

- Tree-Replace-Binary. A synthetic program that builds a large binary tree, then repeatedly replaces randomly chosen subtrees at fixed height with newly built subtrees. (This benchmark was named Destroy in Ref. [Hosking *et al.*, 1992; Hosking and Hudson, 1993].) Tree-Replace-Random is a variant which replaces subtrees at randomly chosen heights.

- Richards. The well-known operating-system event-driven simulation benchmark. Previously used in Ref. [Hosking *et al.*, 1992].

Our set of Java programs is as follows:

- JavaBYTEmark. A port of the BYTEmark benchmarks to Java, from the BYTE Magazine Web-site.

- Bloat-Bloat. The program Bloat, version 0.6, [Nystrom, 1998] manipulating the class files from its own distribution.

- Toba. The Java-bytecode-to-C translator Toba working on Pizza [Odersky and Wadler, 1997] class files [Proebsting *et al.*, 1998].

4.2.3 Properties of benchmarks

We now examine some properties of the benchmark programs. We have already seen their behavior under non-generational collection. Before we proceed to their behavior under different age-based collectors, we look at those properties that are *independent* of any collection algorithm. We examine the distribution of object sizes, the distribution of object lifetimes, and the time-varying live amount profile. We shall later refer to these properties when we look for causes of particular performance behavior, but we shall find that their predictive value is not as great as expected.

We first look at the distribution of object sizes: the typical sizes of objects, and how much sizes deviate from the typical. The size of a *block* must be substantially larger than the typical object size in order to avoid fragmentation. More to the point, the fraction of objects larger than the block size must be small, to minimize the overhead of allocating in the Large Object Space, and the overhead of subsequent collections of objects in Large Object Space [Hicks *et al.*, 1998]. The distribution we measure is a time-averaged property: we do not consider how the numbers of small and large object in the heap change over time.

The size distributions for our set of benchmarks are presented in Figures 4.15– 4.28 (pp. 85–98). The distributions are computed according to four different weighting criteria: (a) by count of objects alone; (b) by volume, which is the object size itself (to reflect the fraction of the heap occupied by objects of a particular size); (c) by lifetime (to take into account that a longer-lived object is more likely to be encountered in collection); and (d) by both volume and lifetime. Each row of a figure corresponds to a weighting criterion. The left column gives the distribution in raw form, as a histogram, whereas the right column gives the cumulative distribution, normalized to 1, so that fractions can be read off easily.

Most programs have several dominant object sizes that constitute most of their allocated volume. Weighting is important: whereas, for instance, 99.5% of allocated objects in Swim are 10 words or smaller, these objects account for 95% weighted by volume, 92.5% weighted by lifetime, and just 50% weighted by both volume and lifetime (Figure 4.21, p. 91). The latter is the most realistic measure of the distribution as seen by a garbage collector,[6] and as we see, if small objects have short lifetimes, their participation in the load can be much less significant than their *numbers* may suggest. We note similar behavior—that smaller objects

[6]We note that age-based collectors that examine less than the whole heap are able to bias the distribution of the objects they see in particular ways with respect to the lifetime, though not with respect to size. Other (non-age-based) collectors are free to *use* size in their allocation decisions.

have shorter lifetimes than larger objects—in all Smalltalk and Java programs except three. The three exceptions are HeapSim, Lambda-Fact5, and Lambda-Fact6, in which a single size represents such an overwhelming fraction of allocation that no differentials can be observed, hence the cumulative distribution curves for different weightings are indistinguishable.[7]

As we have noted, many arguments about garbage collection performance invoke the properties of object lifetime distributions [Baker, 1993; Clinger and Hansen, 1997], sometimes assuming *a priori* distributions in order to derive tractable mathematical models. Even though it is not our goal to develop analytical models, it is proper to evaluate our benchmark set with respect to object lifetime distributions. We shall see that the relationship of the distribution to actual performance is tenuous.

We present the lifetime distributions by means of the survivor function and the mortality function (Section 2.2), shown for our benchmark set in Figures 4.29– 4.42 (pp. 99–103). For each benchmark, the lifetimes of all allocated objects were taken together, weighted by volume (i.e., by object size) and a cumulative distribution was found. Thus, the lifetime distribution is another time-averaged property. Each figure (a) shows the survivor function, plotting object age on a logarithmic scale to show the interesting behavior at young ages more clearly. Each figure (b) shows the mortality function, obtained by numerical differentiation and suitable smoothing from the survivor function. Because of the inherent error of this numerical process, applied to data that are not intrinsically smooth, the mortality curves should be taken *cum grano salis*, as merely indicative of the lifetimes.

The fact that the survivor function is rapidly decreasing, even with a logarithmic scale for the age axis, confirms the observation made many times in the literature, that object lifetimes

[7] The most striking observation is that Java programs have large maximum object sizes, whereas only one Smalltalk program, HeapSim, has objects larger than 140 words. We note that block size in implementation is restricted to be a power-of-two multiple of the virtual memory page size, and that page sizes are of the order 1000–2000 words (8 KB, or 2048 words for Sun Solaris 2.5.1 running on an UltraSparc). Therefore, there is no additional constraint upon block size to make a large object space *unnecessary for these Smalltalk program executions*. The same program may allocate larger objects on a different input. Whether the potential reduction in memory management overhead, by not having a large object space, is significant, and worth the expense of compile-time analysis needed to prove that no large objects are allocated, is beyond the scope of this work.

tend to be short. A few programs exhibit significant fractions of allocated data living to ages close to the duration of the program (Swim, Tomcatv, HeapSim, Tree-Replace-*, Bloat-Bloat, Toba). In others, the survivor function approaches negligible values at an age two orders of magnitude below the duration of the program.

It is tempting to make performance predictions by extrapolating from lifetime distributions. We can speculate that the presence of sharp peaks in the mortality curve (as in StandardNon-Interactive, Figure 4.31, and Richards, Figure 4.39) indicates an opportunity for good performance of the FC-DOF collector, provided that the *age* of high mortality translates into a *heap region*, compatible with the collection window, with low copying cost. On the other hand, there is little to distinguish the survivor functions of Lambda-Fact5 (Figure 4.33) and Richards (Figure 4.39), yet we shall see that their behavior under age-based collection is radically different.

In addition to the preceding time-averaged properties, we look at the live profile: the amount of live data in the heap as a function of time. Our traces already express exact liveness information, thus it is easy to produce the live profiles, given in Figures 4.43– 4.56. The live profiles reveal periodic patterns, as in Figures 4.48 and 4.49, which result from iterative or recursive control structure of the program.

4.3 Simulating garbage collection

The primary goal of garbage collection algorithm simulation in this study is to estimate the copying cost of collection. The simulators accurately compute the amount of data copied in collection; this amount is a good estimate of copying cost. In addition, the simulators compute the number of objects copied, the number of garbage collections invoked, and other secondary statistics.

We first describe object-level simulators. They do not model the costs realistically incurred in remembered set maintenance, although they themselves do maintain per-object remembered sets. These costs will be addressed in Section 4.3.3 using block-level simulation based on the prototype implementation. The simulators do not model the locality effects, namely the locality of access within the collection process, the locality from one collection to another, the locality interference between the mutator and the collector, and lastly, instruction locality. The organization of the simulator is entirely different from that of a realistic garbage collector, thus any timings or locality measurements of the simulated collector are meaningless; moreover, the computational actions of the mutator are not available at all in the trace. These effects can be of interest on modern hardware, and they should be considered in future implementation experiments. However, the primary cost, which is that of copying, is accurately measured by simulation. Simulation has additional benefits, such as the freedom to explore configuration spaces thoroughly and at low cost since computation is eliminated and only memory management is simulated, the freedom to explore new collection algorithms *in abstracto*, and the ability to operate with a common trace format uniformly for various source languages.

The number of garbage collections invoked is easily and accurately obtained by simulation. It is of some importance in evaluating the performance of an algorithm. Each invocation of the collector entails a considerable overhead: the collector must find the global roots of the collection, typically by scanning the execution stack (or multiple stacks in multiple threads of computation). For this reason, the scheme with fewer collector invocations would be preferred, *ceteris paribus*. However, the number of invocations can be viewed from a different angle: the fewer collections there are, the more work is done by each, and the longer the pauses are. In an interactive or real-time setting especially, it is desirable to limit pause times. More precisely, a garbage collection algorithm that imposes an upper bound on the amount of work at each collection may be preferable, in spite of a larger number of collector invocations, to a collector that performs a smaller number of collections, each of longer duration.[8] In summary, it is not appropriate to use the number of garbage collections invoked as a performance measure without reference to the intended application.

[8]Traditional generational collection performs a larger number of collections than non-generational collection, and these collections are on average shorter. However, it *occasionally* collects the entire heap, and thus its upper bound on the amount copied in a collection is not better than with non-generational collection.

4.3.1 General simulation algorithm

The simulation algorithm resembles in its outline the operation of an actual garbage collector (Section 3.1), but differs in important details. We begin with an informal account.

Let us first consider the actions of the simulator while it is simulating the actions of the mutator given in the trace. The simulator sets up a certain, immutable, heap size and an initial allocation area size, and then begins to read the trace. For each `A` record, the allocation area decreases by the size of the allocated object, a running total of volume allocated increases by the same amount, and a data structure is created to represent the object. The simulator maintains a global set of data structures representing all objects present in the heap. A splay-tree data structure [Sleator and Tarjan, 1983] is used for the purpose, because of its good temporal-locality properties. For each `D` record, the simulator marks the corresponding data structure as *known-dead*, but does not discard it immediately. For each `U` record, the simulator first identifies the object in which the updated field is and the object to which the pointer points, and then it records the newly stored pointer in the data structures of the two objects. (The data structure for an object contains both the outgoing pointers and the incoming pointers.)

Eventually, following the allocation of an object, the allocation area will be exhausted, and at that point the simulator begins simulating a collection. The first task is to choose the subset of objects that will be subjected to collection (the collected region C). How this choice is made differs between collection algorithms, as described in Chapter 3. The remaining objects will be the uncollected set U.

The simulator finds the roots of the collection as follows. First, all objects in C that are not *known-dead* are roots. These objects would be reachable from the stack and globals at this point in actual execution. Second, all objects in C which have an incoming pointer from U are roots. These objects must be assumed live because *all* of U is assumed live in region-based collection.

The simulator then computes the transitive closure of the points-to relation over the root set within C, using a simple worklist mechanism. The result is the survivor set S. Data structures corresponding to the garbage set $G = C \setminus S$ can now safely be discarded from the global set. Finally, the sum of the sizes of objects in S is added to the running sum of volume copied, and a collection counter is incremented.

The available allocation area increases by the amount freed, that is, the sum of the sizes of objects in G. If this amount is still not greater than zero, the simulator announces that the collector has failed.

As we have seen, the collection simulator keeps track of the total volume of allocated objects, as well as the total volume of survivor objects over all performed collections. By definition, the total volume of survivor objects is the measure of the copying cost of collection. The mark/cons ratio μ is also directly available as the ratio of the total volume of survivor objects to the total volume of allocated objects.

4.3.2 Age-based simulation

In age-based simulation, the relative order of object allocation remains the (logical) order of objects in the heap, and determines collection decisions. Therefore, the global set is implemented as a (doubly-linked) list. New allocation adds objects to the young (right) end of the list. The choice of the collected region is restricted to selecting a contiguous sublist C of the global list. The collection then finds the sublist of survivors S, which is spliced back into the global list in the place which C occupied.

When simulating age-ordered schemes, the time of allocation serves as a unique timestamp, and membership in a sublist can be determined by timestamp comparison instead of a full-fledged set-membership test; other manipulations are simplified as well.

For renewal-age schemes, S is attached past the young end instead. The objects in S receive *renewed* timestamps.

4.3.3 Block-based simulation

We implemented a prototype version of the block heap organization for age-based garbage collection, as outlined in Section 3.2. This prototype runs on 32-bit Sun SPARC machines and does not use the design for large address spaces of Section 3.2.4. It is heavily instrumented in order to record all collector actions, as well as all the write barrier actions.

In order to examine how the block implementation works for the same traces that were examined using object-level simulation, we construct a *pseudomutator*: a program that accepts a trace input in our standard format, and performs the same heap-related actions that the original traced program did. We then integrate the pseudomutator with the (various) garbage collectors in the block implementation prototype. Because the computational aspects of the program are replaced with table lookup in the pseudomutator, the combined system remains a simulator, rather than an accurate replica of the original program. Therefore we are content with obtaining thorough statistics of copying costs (as counts of words copied) and pointer-management costs (as counts of different write-barrier and remembered-set processing operations).