On the Effectiveness of GC in Java

Ran Shaham
Tel-Aviv University and
IBM Haifa Research
Laboratory
rans@math.tau.ac.il

Elliot K. Kolodner
IBM Haifa Research
Laboratory
kolodner@il.ibm.com

Mooly Sagiv
Tel-Aviv University
sagiv@math.tau.ac.il

ABSTRACT

We study the effectiveness of garbage collection (GC) algorithms by measuring the time difference between the actual collection time of an object and the potential earliest collection time for that object. Our ultimate goal is to use this study in order to develop static analysis techniques that can be used together with GC to allow earlier reclamation of objects. The results may also be used to pinpoint application source code that could be rewritten in a way that would allow more timely GC.

Specifically, we compare the objects reachable from the root set to the ones that are actually used again. The idea is that GC could reclaim unused objects even if they are reachable from the root set. Thus, our experiments indicate a kind of upper bound on storage savings that could be achieved. We also try to characterize these objects in order to understand the potential benefits of various static analysis algorithms.

The Java Virtual Machine (JVM) was instrumented to measure objects that are reachable, but not used again, and to characterize these objects. Experimental results are shown for the SPECjvm98 benchmark suite. The potential memory savings for these benchmarks range from 23% to 74%.

Keywords

Compilers, garbage collection, Java, memory management, program analysis

1. INTRODUCTION

GC does not (and in general cannot) collect all the garbage that a program produces. Typically, GC collects objects that are no longer reachable from a set of *root* references. However, there are some objects that the program never accesses again, even though they are reachable. This is illustrated pictorially in Figure 1.

Following [9], we refer to the time interval from the last use of an object until its collection as the object's *drag time* and to object itself as a *dragged object*.

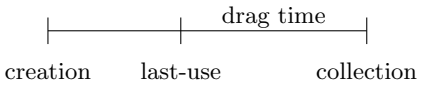

Figure 1: The lifetime of an object.

We study the effectiveness of GC algorithms by measuring the drag time for objects, expressed in bytes allocated from the time of last use until the time of collection. This provides an upper bound on the storage savings over current GC algorithms that could be achieved. The bound is not tight since it is not clear that all such dragged objects could be identified by automatic means. A small value for the upper bound would indicate that the GC reclaims unused memory in a timely fashion.

1.1 Measuring Dragged Objects

We instrumented Sun's JDK 1.2 [6] in order to measure dragged objects. Specifically, we record at regular intervals (1) the objects reachable at the end of the interval and (2) the objects that are also used subsequent to the interval. We call the former the *reachable* objects and the latter the *in-use* objects. Following [1] we measure the space-time product for each.

To calculate the product for the reachable objects, we plot the size of the reachable objects as a function of time (where time is expressed as bytes allocated since the start of the run) and compute the integral under the curve. We do the same for the in-use objects. We call these the *reachable integral* and the *in-use integral* respectively. The ratio between these integrals captures the average space savings.

The measurements were conducted on the SPECjvm98 benchmark suite [12]. The differences between the integrals for the reachable and the in-use objects range from 23% to 74%.

Interestingly, although the total size of dragged objects is sometimes very small, the potential benefit can still be significant. For example, the total size of dragged objects for `jess` is less than 1% of the total allocation size; however the potential benefit is more than 70%. This is due to two factors: (1) most of the dragged objects have a very large drag time and (2) the size of the reachable heap is small.

Röjemo and Runciman performed similar measurements for Haskell, a lazy functional language [9]. Unfortunately, our results are not directly comparable because they measure wall clock time where we measure time in bytes allocated.

Our instrumentation could also be used to understand memory allocation behavior (as done in [9]). We can track memory leaks and tune performance by inspecting the dragged objects. Other tools for memory profiling [11, 8] show the heap configuration and allocation frequency; they also help in tracking memory leaks by allowing the heap to be inspected at points during the execution of the program. As noted in [11], tracking memory leaks by inspecting dragged objects is orthogonal to tracking leaks by inspecting the heap during the course of execution.

1.2 Characterizing Dragged Objects

This research is part of ongoing work to identify static analysis algorithms (both new and existing) that can be used to reduce drag time and to measure their effectiveness. Our work is at a preliminary stage. In the immediate future we intend to redo our experiments using precise GC [2, 5, 4] in order to check the savings due to precise stack scanning for Java.

Our experiments indicate that for some benchmarks, such as `jack`, most dragged objects are also reachable from roots, which are not local variables. For these benchmarks, techniques such as live-precise collection [1], which are based on liveness analysis of local variables, should have a small impact. Our experiments also indicate that most of these objects are reachable from non-private fields; dealing with this case would require analysis of the entire program.

Another interesting fact that we found is that dragged objects exist along very deep heap paths (e.g., a path length of 1231 for the `javac` benchmark). This fact suggests that statically analyzing heap paths, e.g., through shape analysis [3, 10], may result in reclaiming more memory.

Finally it should be noted that in some programs it might be hard if not impossible for any static analysis algorithm to identify all dragged objects. For example, in the `db` benchmark a random access database is maintained. Thus, static analysis of such a database is expected to conservatively determine that all parts of the database are in use.

1.3 Outline of the Rest of this Paper

The remainder of the paper is as follows. Section 2 describes the framework of our experiment. In Section 3 experimental results are given. We conclude in Section 4.

2. THE SCENE

2.1 The Concept

In order to identify the dragged objects, we associate a time field with every object. On every use of an object, we record the current time in its field. Thus, the field always holds the time at which the object was last used. When the object is collected, its drag time is calculated as the difference between the collection time and its last use (see Figure 1). We measure time in bytes allocated since the beginning of program execution. For simplicity, we assume that all uses of an object in the interval between consecutive garbage collection cycles are performed at the beginning of the interval.

For every dragged object we also report the kind of its roots. The major kinds of roots are shown in Table 1. This experiment may indicate the potential benefits of performing liveness analysis for some or all of these root kinds, specifically the benefit of performing local variable analysis [1].

In another experiment we measure the minimal distance of a dragged object from a root in the Java stack. This may indicate whether it is important to statically analyze deep heap paths.

2.2 Implementation

2.2.1 Recorded Information

We attach a trailer to every object to keep track of our profiling information. We do not count the space taken for this trailer in our data. The information in an object's trailer is written to a log file upon reclamation of the object or upon program termination. An object's trailer fields include its creation time, its last use time, its length in bytes, its allocation site, and mark bits to keep track of root reachability. The length includes the header and the alignment (i.e., the bytes that were skipped in order to allocate the object on an 8 byte boundary), but excludes the handle and the trailer. The allocation site holds the object's type if the object was allocated from native code. The mark bits for root reachability contain one bit for each kind of root.

2.2.2 Updating Information

Object information is updated upon the following events:

Object Creation The creation time, length and allocation site are set.

Object Use The last use time is set. The following events constitute an object *use*: (1) getting field information (e.g., via `getfield` bytecode), (2) setting field information (e.g., via `putfield` bytecode), (3) invoking a method on that object (e.g., via `invokevirtual` bytecode) (4) entering or exiting a monitor on that object (via `monitorenter`, `monitorexit` bytecodes) and (5) derefencing a handle to that object.

GC Before GC we clear the special markbits for all objects[1]. During the phase of GC that marks roots, we set the special markbits for the objects directly reachable from the roots according to the kind of root. At the completion of the trace phase, we perform a special trace phase per root kind in order to propagate the marks according to reachability. During the special trace phase for the Java stack root kind, we also measure the (minimal) distance from the Java stack for each object, which is reachable from the Java stack.

[1]Clearing the special markbits before every GC means that we report only the kind of roots that kept the object reachable in the interval just before it became unreachable. We have found that considering all possible root kinds from the last moment the object is used adds "white noise" to the results. For example, if an object dies at a young age, the native stack slot used to hold the address of the object during allocation may still hold the object's address.

Root Kind	Short Description
Java stack root	The Java stack is used for maintaining the execution frames of Java methods. A Java stack root is a memory slot of reference type.
native stack root	The native stack is used for maintaining the execution frames of native (usually C) code. A native stack root is a memory slot of reference type.
static variable root	A static reference variable.
other roots	For example, Java Native Interface (JNI) global references. (used by native code to access Java objects)

Table 1: The major kinds of roots recorded.

Benchmark	Short Description
javac	java compiler
db	benchmark simulating a database
jack	parser generator
raytrace	raytracer of a picture
jess	expert system for solving riddles

Table 2: The benchmark programs.

2.2.3 Reporting Information

After every 100 KB of allocation we trigger a *deep GC* (a larger interval yields less precise results). A deep GC consists of the following steps: (1) GC, (2) run finalizers for all objects waiting for finalization, (3) GC. Forcing finalization ensures instant reclamation of all unreachable objects and removes a source of non-determinism (since finalization would otherwise occur in a separate thread). When an object is freed, we log all of the information collected in its trailer. When the program terminates, we perform a last deep GC and then we log information for all objects that still remain in the heap.

The rules for the collection of `Class` objects are not the same as for regular objects. Thus, we exclude them and the special objects reachable from them (e.g., *constant pool* strings and per-class security-model objects) from our reports.

2.2.4 Process Reported Information

An analyzer processes the log file to produce the results reported in Section 3.

2.3 The Benchmark Programs

We ran our measurements for five of the benchmarks from the SPECjvm98 benchmark suite [12] and excluded three of the benchmarks. Two of the excluded benchmarks do not produce significant amounts of memory (`mpegaudio` benchmark) or significant amounts of objects (`compress` benchmark). The third excluded benchmark is a merely a multi-threaded version of another benchmark (`mtrt` benchmark is the multi-threaded version of `raytrace` benchmark). We show the benchmarks that are used in table 2.

3. EXPERIMENTAL RESULTS

3.1 Dragged Object Size

Figure 2 shows total number of bytes allocated and total number of bytes attributed to dragged objects for each benchmark. We define a *short-lived object* to be an object that is allocated and becomes unreachable in the same measurement interval. We also present the total size of the short-lived objects, since there cannot be any memory savings for these objects. Short-lived objects are excluded from the subsequent results, since these objects are not reachable at the sampling points.

Although in four benchmark less than 10% of the total object allocation are dragged objects, the total size of the dragged objects is large compared to the total size of the reachable objects, (as shown in Section 3.2), so there is a potential for a large savings in memory.

3.2 Reachable vs. In-Use Objects

Table 3 compares the reachable integral, with the in-use integral. We show the integrals when half of the total bytes have been allocated, labeled *halftime*, and when the program terminates. The potential for savings ranges from 23% to 74% of the reachable integral.

Figure 3 compares the reachable object size and the in-use object size over time. We see that the `javac` benchmark operates in several cycles. For every cycle there is an initial phase of allocating memory, in which almost every reachable object is in-use. As the program continues, the difference in sizes between the reachable and the in-use objects increases until the end of the cycle. At that time the memory consumption of the program drops at once, and another cycle begins.

For the `db` benchmark, after an initial phase of allocating memory, the difference between the reachable and the in-use objects is constant until the program allocates 40.59 MB. After that point the size of the reachable objects remains more or less constant, but fewer objects are used, so that the difference between the reachable and in-use object sizes increases. This motivates presenting the reachable and in-use allocation integrals half way through the program in Table 3.

`jack` behaves similarly to `javac`, and `raytrace` behaves similarly to `db`.

For `jess` the difference between the reachable and in-use object sizes is constant at 0.75 MB. This is further explained in Section 3.3.

3.3 Drag Time

Table 4 shows the distribution of dragged object size with respect to drag time. For `jess` 30% of the dragged object

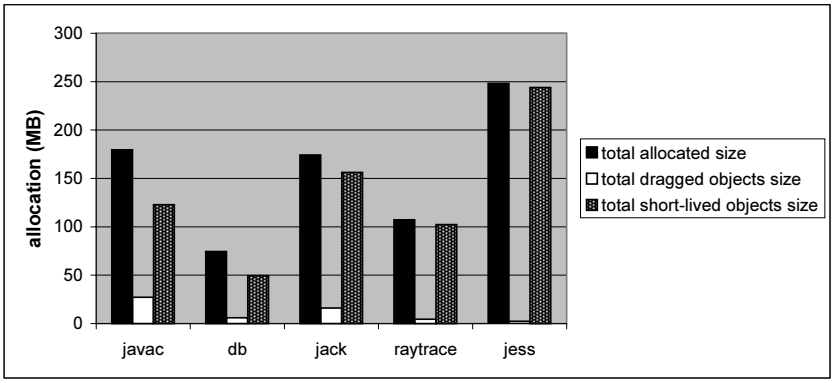

Figure 2: Total allocation size, total dragged object size and total short-lived object size.

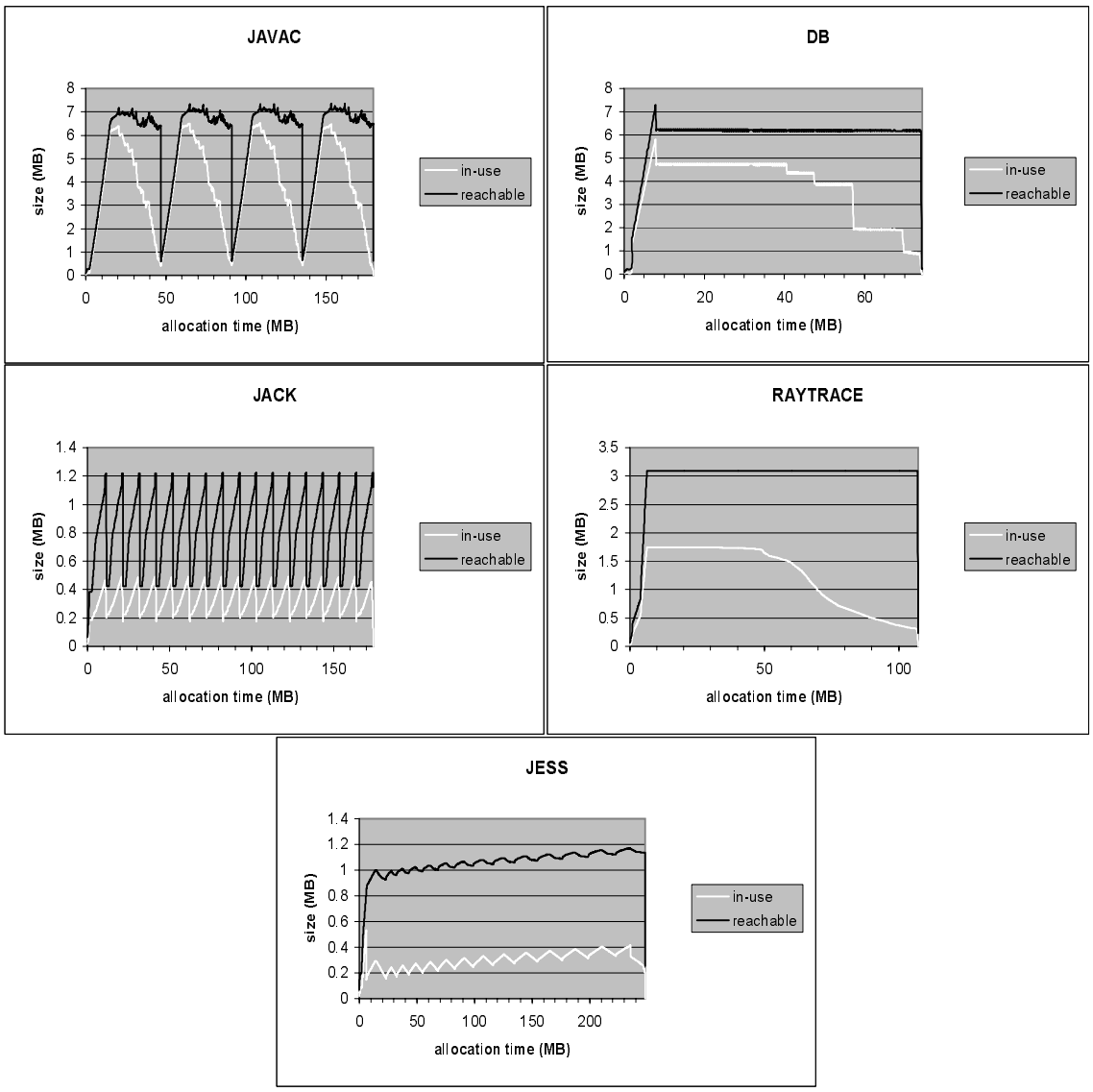

Figure 3: Reachable object size vs. in-use object size.

Benchmark Program	Halftime			Total		
	In-Use Integral (M $Byte^2$)	Reachable Integral (M $Byte^2$)	Saving Ratio (%)	In-Use Integral (M $Byte^2$)	Reachable Integral (M $Byte^2$)	Saving Ratio (%)
javac	345.06	504.48	31.60	692.34	1032.58	32.95
db	160.14	208.93	23.35	271.68	437.84	37.95
jack	28.04	69.40	59.60	57.18	142.12	59.76
raytrace	85.60	152.88	44.01	128.48	317.98	59.59
jess	31.73	122.72	74.15	73.84	261.15	71.72

Table 3: In-Use Integral vs. Reachable Integral.

Pct.	javac	db	jack	raytrace	jess
90%	0.2	0.3	0.2	0	0.2
80%	2.3	4.7	0.3	0	0.2
70%	4.3	17	0.3	0.1	0.2
60%	6.7	17	0.8	13.4	1.6
50%	9.6	17	1.1	34.8	6.6
40%	12.7	17.1	4.1	43.3	12.7
30%	15.7	33.6	6.7	100.9	241.9
20%	20.3	67.5	7.3	102	241.9
10%	26.2	70.4	8	103.1	245.2

Table 4: Distribution of total dragged object size with respect to drag time (expressed in MB).

size (which is 0.69 MB) has a drag time of at least 241.9 MB; this is nearly the lifetime of the program. Thus, these objects are used for their last time near the beginning of the program, yet they are not collected until the program terminates. As noted before, there is also a constant difference of 0.75 MB between the reachable and in-use objects. These two facts explain how a seemingly insignificant amount of dragged objects (less than 1% of the total allocation size) can result in a 72% savings in the reachable integral (i.e., 72% of average space savings).

3.4 Distribution of Dragged Objects by Root Kind

Figure 4 shows the distribution of the dragged objects by the types of the roots that keep them reachable. In most of the benchmarks (3 out of 5), most of the dragged objects have just one kind of root, the Java stack. For example, in `javac` close to 88% of the dragged objects are reachable solely through the Java stack.

For `db`, close to 97% of the dragged object are reachable from the Java stack, as well as from the native stack. `db` maintains a database; all objects in the database are reachable from the database root object of type `spec.benchmarks._209_db.Database`. This database root object is directly referenced from the native stack for the entire course of the program; thus, all database objects are considered as native stack reachable. In this case, live-precise collection should have little impact.

For the `jack` benchmark, close to 30% of the dragged objects are reachable only from static variables. Another 50% of the dragged objects are reachable from both the Java stack and from static variables. Investigating the bytecode we found that the relevant static fields have `public` access; thus, in order to perform liveness analysis for these variables the entire program would need to be analyzed. This could be very expensive and is difficult to apply to Java.

Benchmark Program	Average Distance	Maximum Distance
javac	20.47	1231
db	6.24	20
jack	6.78	25
raytrace	22.19	1412
jess	6.72	28

Table 5: Distance of dragged objects from Java stack.

3.5 Distance from the Stack

Table 5 shows the average distance and maximum distance of dragged objects from the Java stack. For 3 out of 5 benchmarks the average distance for dragged objects is around 6.5. This seems like a reasonable number for object-oriented languages. `javac` and `raytrace` benchmarks have a much larger average distance from the Java stack. This is due to the presence of long linked lists (of length 1231 and length 1412, respectively) containing dragged objects. In `javac` this linked list represents the bytecodes of a method being compiled, while in `raytrace` this linked list represents the vertices of a complex polygon. Using shape analysis [3, 10] to analyze these paths may provide a memory savings.

4. CONCLUSION AND FUTURE WORK

We give an upper bound for the effectiveness of an existing garbage collector in Java. There is a potential benefit of 23% to 74% of space savings. We are continuing to work to identify the static analysis methods that can realize some of this potential. For example, by measuring the dragged objects of TVLA [7], a tool for three-value logic analysis we are already able to point out some code rewriting that can be done in order to save space.

5. ACKNOWLEDGEMENTS

We would like to thank Hans Boehm for giving us a reference to the work of Röjemo and Runciman.

The first author is supported in part by the Binational Sci-

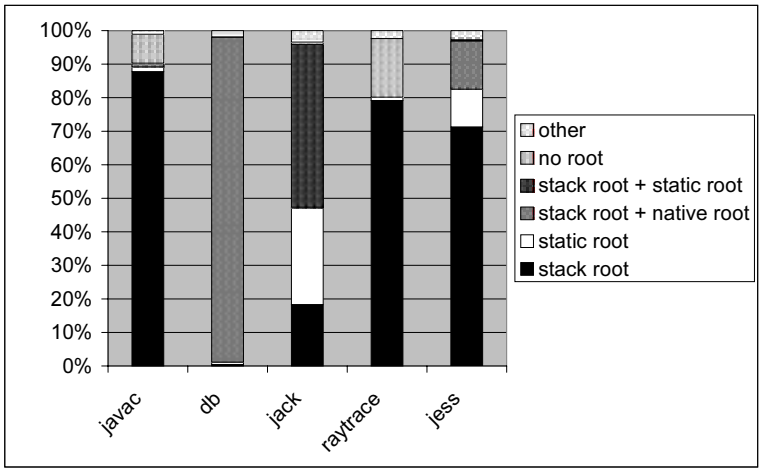

Figure 4: Distribution of dragged objects by root kind.

ence Foundation, grant number 96-00337.

6. REFERENCES

[1] O. Agesen, D. Detlefs, and E. Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 269–279, June 1998.

[2] A. W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[3] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[4] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 273–282, San Francisco, CA, June 1992.

[5] B. Goldberg. Tag-free garbage collection for strongly typed programming languages. *ACM SIGPLAN Notices*, 26(6):165–176, 1991.

[6] Sun JDK 1.2. Available at http://java.sun.com/j2se.

[7] T. Lev-Ami and M. Sagiv. TVLA: A framework for kleene based static analysis. In *SAS'00, Static Analysis Symposium*. Springer. Available at "http://www.math.tau.ac.il/∼ tla".

[8] W. D. Pauw and G. Sevitski. Visualizing reference patterns for solving memory leaks in java. In *ECOOP'99*, pages 116–134, Lisbon, Portugal, 1999.

[9] N. Röjemo and C. Runciman. Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 34–41, Philadelphia, Pennsylvania, 24–26 May 1996.

[10] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".

[11] M. Serrano and H.-J. Boehm. Understanding memory allocation of scheme programs. Submitted to ICFP'00, 2000.

[12] SPECjvm98. Standard Performance Evaluation Corporation (SPEC), Fairfax, VA, 1998. Available at http://www.spec.org/osg/jvm98/.