Providing hints for garbage collection

D. Buytaert, K. Venstermans, L. Eeckhout

Promoter(s): Koen De Bosschere

Abstract—**This paper presents a mechanism that uses off-line profile information to examine when garbage is best collected. This information is then used to guide the garbage collection frequency in order to reduce the garbage collection time and total execution time.**

Keywords—**Java, garbage collection, scheduling**

I. Introduction

Object-oriented languages with automatic memory management (i.e. garbage collection), such as Java, have become increasingly popular. While garbage collection offers many benefits, the time spent reclaiming memory can account for a significant portion of the total execution time [1]. Although garbage collection research has been a hot research topic for many years, little research is done to help garbage collectors (GCs) decide *when* to collect.

Typically, garbage is collected when either the heap, or a part thereof (i.e. a *region* or a *space*), is full. Clearly, postponing garbage collection until the heap is full minimizes the number of required collections and maximizes the amount of garbage that can be collected each run. However, this does not necessarily minimize the application's execution time.

Therefore, we manipulate the garbage collection frequency by inserting *garbage collection hints* at favorable points. At run-time, the GC will use these hints to determine whether to collect or not. We refer to this mechanism as *garbage collection scheduling*.

Preliminary results show that for a number of benchmark configurations, garbage collection scheduling reduces the GC's execution time by a factor 10 and improves the program's total execution time by more than 20%.

II. Experimental setup

For our study, we use Jikes RVM [2] on an AMD Athlon XP with different garbage collection algorithms. To evaluate our mechanism, we use the SPECjvm98[1] benchmark suite. Due to space constraints, we only present some preliminary results obtained with the `_213_javac` benchmark. Figure 1 depicts `_213_javac`'s live bytes over time. We used the `-s100` input set, a fixed size heap of 106MiB[2] and ran the benchmark four times using the `-m4 -M4` arguments. The data is obtained using Jikes RVM's `BaseBaseGenMS` configuration which uses a generational collector with two spaces.

D. Buytaert, K. Venstermans and L. Eeckhout are with the Department of Electronics and Information Systems (ELIS), Ghent University, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium. E-mail: {dbuytaer, kvenster, leeckhou}@elis.UGent.be

[1] `http://www.spec.org/jvm98/`

[2] The notations KiB, MiB, and GiB used in this paper are SI standard notations for binary multiples, and denote 2^{10}, 2^{20}, and 2^{30} respectively. See also `http://physics.nist.gov/cuu/Units/binary.html`.

III. Methodology

Our mechanism to guide garbage collection frequency works in three steps. As a first step, we use an off-line profile run to collect and correlate information about object lifetime and method invocations. Second, we use this information to identify favorable collection points. In the third and last step, conducted at run-time, we instrument the methods that have been identified as favorable collection points.

A. Collecting profile information

To collect information about method invocations, we modified Jikes RVM to timestamp and report all thread switch events, method entries and exits. Upon method entry and method exit, a timestamp counter is incremented and written to a trace file along with the current method and thread IDs.

To obtain accurate information about object death times, we used the Merlin trace generator [3] to perform a lifetime analysis at every point an object could have died. Merlin has been modified to use our alternative timestamping method.

B. Identifying favorable collection points

After correlating all profile information, we can identify methods that represent favorable collection points. For a method to be selected, it needs to satisfy two criteria:

1. The method's invocations should represent local minima in terms of the number of live bytes, so garbage can be collected with minimal effort.
2. The method's invocations should have good coverage. That is, there should be a minimum and a maximum distance between subsequent calls (uniform spread) where distance is expressed in terms of allocated memory.

C. Guiding collection through instrumentation

Once the favorable collection points have been identified, they are instrumented. The instrumentation code guides the GC to decide whether garbage collection should be triggered or not.

Garbage collection is best done when the following criteria are met:

1. The *live ratio* is low. The fewer objects/bytes are live, the fewer objects will need to be scanned and the fewer objects will need to be copied.
2. The *death ratio* is high. The more objects/bytes are dead, the more memory can be reclaimed, and the longer we can postpone the next garbage collection run.
3. A collection reduces the time spent in subsequent garbage collection runs.

Because traditionally, garbage is collected when the heap is full, we can only increase the frequency of garbage collection. As such, there is a trade-off between reducing the amount of garbage collection work and having to collect more frequently.

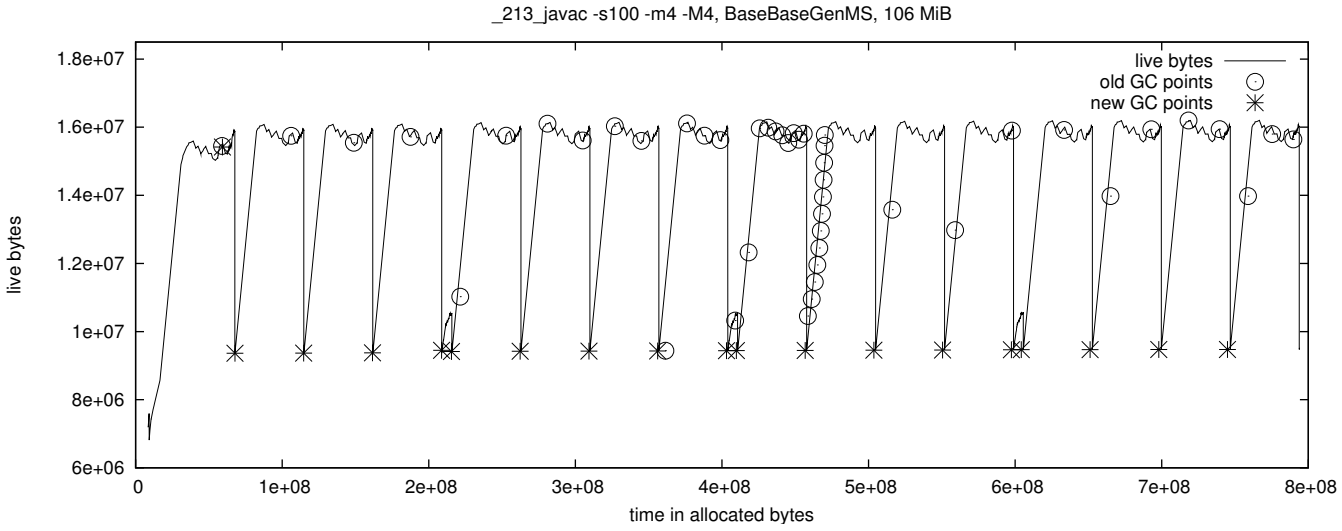

Fig. 1

GARBAGE COLLECTION POINTS BEFORE AND AFTER GARBAGE COLLECTION SCHEDULING

It is important to note that in case of generational collectors, we can decrease the frequency of garbage collection by manipulating what region is collected.

IV. RESULTS

Figure 1 shows the points at which Jikes RVM would normally collect, labeled *old GC points*. Two observations can be made: (i) garbage collection is often done at points where a lot of bytes are live and (ii) Jikes RVM uses smaller nursery collections while collecting the entire heap would have been more beneficial.

Next, we analyzed the application with our tool and found that there was a method that (i) represents the local minima (ii) with good coverage, namely `spec.benchmarks._213_javac.ClassPath.<init>()`. Adding garbage collection hints to this method reduced the number of garbage collections (multiple nursery collections are replaced by full-heap collections), and caused garbage collection to be triggered in local minima. The new garbage collection points are shown on Figure 1 with the label *new GC points*. Because of the improved garbage collection frequency, the GC's execution time has been reduced by a factor 10 and the program's total execution time improved by more than 20%. Note that it is not always possible to improve performance, and that mispredictions will degrade performance.

V. RELATED WORK

Although a number of studies have been done on the interaction of heap size, garbage collection strategy and overall performance, very few of them investigate scheduling garbage collections.

Recent work of Blackburn *et al.* [4] shows that small changes in heap size can produce significant differences in execution and garbage collection time. In other work, Brecht *et al.* [5] found that garbage collection frequency can have substantial impact on the execution time and that a sweet spot exists in terms of minimizing execution time. Both observations suggest that it is meaningful to schedule garbage collection.

VI. ACKNOWLEDGMENTS

Dries Buytaert is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Kris Venstermans is supported by a BOF grant from Ghent University. Lieven Eeckhout is a Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (FWO—Vlaanderen).

REFERENCES

[1] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere, "Method-level phase behavior in Java workloads," in *Proceedings of the ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA'04)*. Oct. 2004, ACM Press.

[2] B. Alpern, B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley, "The Jalapeño Virtual Machine," *IBM Systems Journal*, vol. 39, no. 1, pp. 211–238, Feb. 2000.

[3] Matthew Hertz, Stephen M Blackburn, J Eliot B Moss, Kathryn S. McKinley, and Darko Stefanovic, "Error free garbage collection traces: how to cheat and not get caught," in *Proceedings of the 2002 ACM SIGMETRICS international conference on measurement and modeling of computer systems (SIGMETRICS'02)*. 2002, pp. 140–151, ACM Press.

[4] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinely, "Myths and realities: the performance impact of garbage collection," in *Proceedings of the 2004 ACM SIGMETRICS international conference on measurement and modeling of computer systems (SIGMETRICS'04)*. 2004, ACM Press.

[5] Tim Brecht, Eshrat Arjomandi, Chang Li, and Hang Pham, "Controlling garbage collection and heap growth to reduce the execution time of Java applications," in *Proceedings of the 16th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA'01)*. 2001, pp. 353–366, ACM Press.