

Profile-Driven Pretenuring for Java

Stephen M. Blackburn John Cavazos Sharad Singhai Asjad Khan
Kathryn S. McKinley J. Eliot B. Moss Sara Smolensky

Architecture and Language Implementation Laboratory, Department of Computer Science
University of Massachusetts, Amherst, MA 01003-4610

Overview

Pretenuring is a technique for reducing copying costs in garbage collectors. When pretenuring, the allocator places long-lived objects into regions that the garbage collector will rarely, if ever, collect. We extend previous work on profiling-driven pretenuring as follows. (1) We develop a collector-neutral approach to obtaining object lifetime profile information. We show that our collection of Java programs exhibits a very high degree of homogeneity of object lifetimes at each allocation site. This result is robust with respect to different inputs, and is similar to previous work on ML, but is in contrast to C programs, which require dynamic call chain context information to extract homogeneous lifetimes. Call-site homogeneity considerably simplifies the implementation of pretenuring and makes it more efficient. (2) Our pretenuring advice is neutral with respect to the collector algorithm, and we use it to improve two quite different garbage collectors: a traditional generational collector and an older-first collector. The system is also novel because it classifies and allocates objects into 3 categories: we allocate *immortal* objects into a permanent region that the collector will never consider, *long-lived* objects into a region in which the collector placed survivors of the most recent collection, and *short-lived* objects into the *nursery*, i.e., the default region. (3) We evaluate pretenuring on Java programs. Our simulation results show that pretenuring significantly reduces collector copying for generational and older-first collectors.

1. MOTIVATION AND APPROACH

Generational copying garbage collection partitions the heap into age-based generations of objects, where age is measured in the amount of allocation (the accepted practice in the garbage collection literature). Collection consists of three phases: (1) identifying roots for collection; (2) identifying and copying into a new space any objects identified as ‘live’ through transitive reachability from those roots; and (3) reclaiming the space vacated by the live objects. Rather than collecting the entire heap and incurring the cost of copying all live objects upon each invocation, generational collectors collect the youngest objects in the heap, the *nursery*, and include successively older generations only if necessary.

This basic approach has two problems. First, the collector may

copy long lived objects multiple times (into higher generations). Second, the most recently allocated objects in the nursery have little time to die, and the collector may copy many of them, even if they have short lifetimes. The main focus of this work is using pretenuring to solve the first problem, but we also combine pretenuring with a solution to the latter problem that collects older objects before younger ones [Stefanović et al. 1999].

An ideal pretenuring algorithm would inform the allocator of the exact lifespan of a new object, and then the allocator would select the ideal generation in which to place the object. The collector would thus consider an object only after it has sufficient time to die, avoiding copying it from the nursery and perhaps other generations. If an object will die before the next nursery collection, then the allocator places it in the nursery (the default), whereas if the object lives until the termination of the program, then the allocator places the object into a permanent region. The use of pretenuring is not limited to generational collectors; any collector that divides the heap into regions and delays or prevents collection of some regions will benefit from such information. We present simulation results for both generational and older-first [Stefanović et al. 1999] collectors with pretenuring.

Without an oracle, pretenuring advice can be gleaned from application profiling on a per allocation-site [Cheng et al. 1998] or call-chain [Barrett and Zorn 1993; Seidl and Zorn 1998] basis. For our collection of Java programs, we found that allocation-site advice results in accurate predictions. These results are robust over different input data. The results are similar to those for ML programs [Cheng et al. 1998], whereas C programs need the additional context of a call-chain [Barrett and Zorn 1993; Seidl and Zorn 1998].

Our pretenuring advice is unique in that it is neutral with respect to the collector algorithm, and we use the same advice to improve two collectors. Instead of pretenuring based on whether the objects at a call site survive a nursery collection, as in previous work [Cheng et al. 1998], we use two object lifetime statistics measured in bytes allocated: *age* and *time of death*. Object age refers to how long an object lives in bytes of allocation, while time of death refers to the point in the allocation history of the program at which the object becomes unreachable. We then classify each object as either *immortal*—the time of death was at the end of the program, *short lived*—it lives less than some fraction of the maximum live size of the heap, or *long lived*—everything else. Each call site is given a classification that is representative of the objects allocated at that site. We then modify the allocator statically as follows. For an immortal site, the allocator places objects into a permanent region that the collector will never consider (previous work scanned but did not copy these objects [Cheng et al. 1998]). For a long-lived site, objects go into a region into which the collector normally copies

survivors of collections. For short-lived sites, allocation remains the same.

2. ALLOCATION SITE HOMOGENEITY

We analyze age and lifetime statistics using an execution profile for each application. The profile takes the form of an object graph mutation trace, which records all object allocations, pointer mutations, and object deaths in our JVM. We instrument all allocations, write barriers to track pointer mutations, and when the collector frees an object. To obtain the accurate object death information, we trigger a non-generational, full heap collection frequently.

We associate pretenuing advice in a tag for each allocation site using three discrete classifications: *short*, *long*, and *immortal*. We normalize our measures of time in terms of the ‘max live size’, i.e., maximum amount of live objects, which is equivalent to the minimum heap size in which the application can execute. The following algorithm is used: 2. If an object’s age is less than $T_d \times \text{max live size bytes}$, then it is classified *short*. 2. Otherwise, if an object’s time of death is within $T_d \times \text{max live size bytes}$ of the end of allocation, then it is classified *immortal*. 3. In all other cases, an object is classified *long*.

To assess the level of per site homogeneity, we classify objects as (short (*s*), long (*l*), and immortal (*i*)) on both a per object and per site basis. We examine the per object (exact) and per site (representative) decisions for each object to establish the level of error in the per site decisions. We classify each decision in terms of the nine decision pairs that arise from the cross product of the two three-way choices ($\langle s_o, s_s \rangle$, $\langle s_o, l_s \rangle$, $\langle s_o, i_s \rangle$, $\langle l_o, s_s \rangle$, ...), where a_o and a_s refer to object and site advice respectively.

The nine decision pairs fall into three categories: neutral, bad, and good with respect to the non-pretenued status quo. Neutral pretenuing advice ($\langle s_o, s_s \rangle$, $\langle l_o, s_s \rangle$, and $\langle i_o, s_s \rangle$) allocates all the objects into the nursery. Bad pretenuing advice ($\langle s_o, l_s \rangle$, $\langle s_o, i_s \rangle$, and $\langle l_o, i_s \rangle$) leads to space wastage through deferred collection of objects that die. Good pretenuing advice ($\langle l_o, l_s \rangle$, $\langle i_o, l_s \rangle$, and $\langle i_o, i_s \rangle$) delays collection of long lived objects and completely avoids collection of immortal objects.

On average 98.2% to 99.3% of pretenuing decisions are good or neutral. The majority of pretenuing decisions are neutral, and a significant fraction are good. In particular, about 9% of allocations of immortal objects are homogeneous enough to pretenu into the permanent space, and we allocate an additional 2% into the long lived space. We identify about 4% as long lived objects.

3. SIMULATION RESULTS

We implemented this technique in a simulator using *self prediction* (profile information from the same input), and *true prediction* (profile information from a different program input).¹ The simulation results show our pretenuing algorithm significantly reduces collector copying for generational and older-first collectors, and is robust with respect to different inputs. We used traces of four applications in our simulation, jBYTEmark, javac, the Olden ‘health’ benchmark, and a bytecode optimization and analysis tool, ‘bloat’.

When using self predicting pretenuing advice, we found that for the four applications all showed some reduction in copying cost when using a conventional generational collector. For javac and health, the improvement was generally a reduction of 45% to 85%.

¹The terms *self prediction* and *true prediction* are coined by Barrett and Zorn [1993].

For bloat the reduction was from 5% to 15%, and for jBYTEmark the improvement was negligible, with some results marginally better and some marginally worse. Our results for the older first collector [Stefanović et al. 1999] were very similar, although overall copying costs (with and without pretenuing) were lower and for jBYTEmark pretenuing resulted in a slightly higher copying cost.

We ran javac and health with varying inputs and were thus able to assess the performance of our pretenuing advice using true prediction. Advice was generated on the basis of one execution of each of the benchmarks and then applied to executions with different inputs. For health this resulted in major changes to the scale of the program execution (number of bytes allocated varied by a factor of about 16), whereas for javac we compiled different programs and so exercised different areas of the program. Using the generational collector, we saw perfect true prediction for health—the pretenuing advice was robust to such large changes in the heap size and volume of allocation. The effectiveness of true prediction was more variable for javac, generally there were improvements from about 5% to 50%, but very occasionally true prediction lead to an increase in copying. The results for the older first collector were very similar, although the improvements for javac were more modest.

4. CONCLUSIONS

Our principal conclusions are as follows. Java programs exhibit remarkable lifetime homogeneity at each allocation site which makes effective pretenuing easy and cheap to implement. We characterize lifetime in a simple way, largely insensitive to parameters or to varying program inputs. This characterization is independent of collection algorithm and configuration, as witnessed by our results for two different collection algorithms. Pretenuing is often effective in reducing object copying for Java programs for both generational and older-first collectors. The result is that pretenuing for Java is quite promising for reducing memory management costs.

Bibliography

- BARRETT, D. A. AND ZORN, B. 1993. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the ACM SIGPLAN’93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, June 23-25, 1993*, Volume 28 of *SIGPLAN Notices* (June 1993), pp. 187–196. ACM.
- CHENG, P., HARPER, R., AND LEE, P. 1998. Generational stack collection and profile-driven pretenuing. In *Proceedings of the ACM SIGPLAN ’98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, 17-19 June 1998*, Volume 33 of *SIGPLAN Notices* (May 1998), pp. 162–173. ACM.
- SEIDL, M. L. AND ZORN, B. G. 1998. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, California, October 3-7, 1998*, Volume 33 of *SIGPLAN Notices* (Nov. 1998), pp. 12–23. ACM.
- STEFANOVIĆ, D., MCKINLEY, K. S., AND MOSS, J. E. B. 1999. Age-based garbage collection. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA ’99), Denver, Colorado, November 1-5, 1999*, Volume 33 of *SIGPLAN Notices* (Nov. 1999), pp. 379–381. ACM.