Efficient Object Sampling Via Weak References

Ole Agesen*
VMware
3145 Porter Drive
Palo Alto, CA 94304
agesen@vmware.com

Alex Garthwaite
Sun Microsystems Laboratories
1 Network Drive
Burlington, MA 01803-0902
alex.garthwaite@sun.com

ABSTRACT

The performance of automatic memory management may be improved if the policies used in allocating and collecting objects had knowledge of the lifetimes of objects. To date, approaches to the pretenuring of objects in older generations have relied on profile-driven feedback gathered from trace runs. This feedback has been used to specialize allocation sites in a program. These approaches suffer from a number of limitations. We propose an alternative that through efficient sampling of objects allows for on-line adaption of allocation sites to improve the efficiency of the memory system. In doing so, we make use of a facility already present in many collectors such as those found in Java™ virtual machines: weak references. By judiciously tracking a subset of allocated objects with weak references, we are able to gather the necessary statistics to make better object-placement decisions.

1. INTRODUCTION

With better knowledge of the behavior of objects in a running application, we can make better decisions about how to manage those objects. These decisions may affect, for example, how and where objects are placed. Typically, instrumenting all objects to gather this knowledge would slow the application too much to be practical. We present a technique for dynamic sampling of a subset of the allocated objects that incurs low runtime overheads. Coupled with automatic memory management or collection, this technique allows us to improve the efficiency of the collector by segregating objects, sampled and non-sampled alike, based on observed characteristics such as object lifetime. The sampling technique can track many kinds of information but for purposes of this paper we concentrate on the improvement for generational garbage collectors.

1.1 Improving Generational Collectors

Strongly typed languages like the Java™ programming language rely on automatic memory management, also known as garbage collection. Memory management services free the programmer from the burden of explicitly reasoning about the use of memory and eliminate two classes of errors:

- *memory leaks* errors where the application loses track of allocated memory without freeing it.
- *dangling references* where the application frees memory while retaining references to it and subsequently accesses this memory through these references.

Garbage collectors work by handling all requests to allocate memory, by ensuring that this allocated memory has an appropriately typed format, and by guaranteeing that no memory is freed until it is proven that the application holds no references to that memory.

Garbage collection services typically allocate objects in a heap. Periodically, the collector locates the set of objects in the heap still reachable from the running program and frees the rest of the memory in the heap so that it may be used for new allocation requests. Often, the collection technique involves stopping the application while this process of finding the reachable objects is performed. For large heaps, this may lead to long pauses during which the application is unable to proceed.

Generational collectors are designed to address part of this pause-time problem. The observation is that recently allocated objects tend to die (that is, become unreachable) quickly. The approach in generational collectors is to divide the heap into an ordered sequence of two or more subheaps or generations. Objects are allocated primarily in the first generation; as objects survive in a particular generation across collections, that generation will have a policy for promoting these longer-lived objects to the next generation. Most collection work is performed in the youngest generation which is sized to cause most collection-time pauses to be of acceptably short duration.

One inefficiency of a generational heap, however, is that long-lived objects may be copied many times before reaching the appropriate generation. Our technique improves generational collectors by identifying objects that will most likely survive to be tenured to a particular generation and by allocating such objects directly in that generation. By sampling a subset of the objects and studying their lifetimes, we are able to better place objects to reduce the number of times such objects are copied within a generation or promoted between generations. A side-benefit is that by not allocating long-lived objects in younger generations, we reduce the number of collections in those generations. Finally, our approach uses continued sampling to dynamically track how the observed lifetimes of objects change as the application executes. This allows the technique to adapt as the application changes the way in which it uses objects.

*The work presented in this paper was performed while this author was at Sun Microsystems, Inc.

1.2 Object Sampling and Fingerprinting

Central to our sampling technique is the use of weak references. Simply put, a weak reference is a reference to an object with the property that the garbage collector does not consider this reference when determining the reachability of the referred-to object. As long as some other stronger reference keeps the object alive, the weak reference will continue to refer to the object. At some point, the collector determines that the object is only reachable from weak references of a given strength. For each weak reference to the object, an "imminent death" action is then performed, followed by clearance of the weak reference so that the referred object can subsequently be reclaimed. Weak references can be available as first-class constructs in the language, as is the case with the Java™ platform, or they can be an implementation-level construct visible only to the runtime system. Either kind of weak reference may be used with our approach so long as the one chosen cannot cause an object to become reachable again.

The Java platform [10] mandates four kinds or strengths of weak references under the package `java.lang.ref`:

- *soft* references meant to be used for implementing caches
- *weak* references meant to implement canonicalization data structures
- *final* references used to implement finalization
- *phantom* references designed to schedule actions once an object ceases to be reachable

Weak references to objects are processed in order of strength. Weak references, when processed, may be enqueued on a reference queue for further processing by the application. For our purposes, it is important to note that finalization of objects may result in those objects becoming strongly reachable again. Only phantom references are guaranteed not to resuscitate a dying object. This means that phantom references or VM-specific weak references are appropriate choices for implementing our proposal in a Java virtual machine.

As a motivating example, suppose we want to determine the average lifespan of certain objects. Specifically, we might want to know if instances of a certain class X tend to live longer than instances of a certain other class Y. Monitoring all allocations of classes X and Y to compute the exact lifespan statistics may be too costly to be practical in the production use of most programs. However, it is possible that knowing the lifespans of a small fraction of all X and Y instances, say one in every 1000 allocated, allows us to estimate lifespans for the entire population of X and Y instances with a useful degree of accuracy. We can sample X and Y instances effectively by attaching a weak reference to every 1000th X and Y instance allocated. The weak reference tracks the instances, and the associated "imminent death" action will inform us when they cease to be alive. Thus, we get access to both the birth and death events for a specifiable fraction of X and Y instances, from which we can estimate their relative lifespans.

More generally, this approach combines the use of weak references to track a sample of objects over their lifetimes with the idea of "fingerprinting" such objects, i.e., associating additional information with the objects. The fingerprint summarizes interesting aspects of the state of the system at the point when the object is allocated. A convenient place to store the fingerprint would be in association with the weak reference data structure.

Examples of the kind of information that one might include in the fingerprint are:

- the time at which the object is allocated. Allocation time may be measured in many ways: real time, CPU time, number of collections since the start of the application, number of bytes allocated. If we combine information from allocation with information gathered at other points in the program, we can infer reasonable bounds on object lifetimes.
- the allocation site in the program where the object is created. This may include a summary of information from one or more frames of the calling context for the allocation operation. Many different places in a program may allocate instances of the same class. Sometimes it might be useful to distinguish between these. For example, it might be that `String` objects allocated at one site (say, used for file names) tend to live much longer than `String` objects allocated at another site (say, created to print floating point numbers).
- the type of the object. In most object-oriented languages, this information does not need to be included in the fingerprint since it is stored in the object itself. For other languages where runtime type information is not readily available, it may be approximated by some other metric such as object size [3].
- in object-oriented languages, the type of the receiver to which the method performing the allocation is applied. [1]
- an identifier for the thread performing the allocation. Objects of the same type allocated at the same allocation site by different threads may serve different purposes and, hence, have different lifespans.

Weak references can be implemented relatively efficiently, and by varying the fraction of objects we sample, we can trade efficiency and statistical accuracy against each other. Further, in environments like the Java platform where support for weak references already exists, we can efficiently gather statistics on sampled objects as they become unreachable without having to explicitly examine the heap for dying objects after each collection.

As can be seen, by including various pieces of information in the fingerprints, and by controlling sampling rates, we can estimate:

- how many objects are allocated of each type (class)
- how many objects are allocated at each allocation site
- how long each category (class, allocation site) of objects tend to live

Furthermore, we can refine the above kind of information either on a per-thread basis or through correlated class information. The uses of such statistical information include optimizations in the memory system as well as offering a source of feedback to the programmer allowing the program to be better optimized or debugged. For instance, in the memory system, one might:

- allocate (or pretenure) certain categories of objects directly into specific generations.
- promote objects to selected generations.
- better place and move objects within generations, especially in the context of systems based on the Train Algorithm [9].
- choose which objects to allocate in thread-local versus global areas of the memory system.

[1] Our colleague, Tim Harris, suggested this idea.

Using the data gathered about reclaimed and still-live objects and their lifespans, we can improve the efficiency of generational collectors through better object placement, reduction in the copying of objects, and reduced collection of individual generations.

1.3 Roadmap

In Section 2, we briefly review work related to our proposal. In Section 3, we offer our own design for a dynamic object-sampling and fingerprinting framework applicable to a generational framework. Finally, we consider future work and conclude.

2. RELATED WORK

Garbage collection has a long history. Two excellent surveys of the area are available from Paul Wilson [19] and Richard Jones [11]. The earliest reports on generational collectors can be traced to David Moon [13], Henry Lieberman and Carl Hewitt [12] and their implementation of a collector for Lisp. They observed that segregating young objects and concentrating collection on those objects yields a more efficient memory system. Another pioneer in generational collection is David Ungar who developed a simple, two-generation collector where the heap is split into an allocation area or eden and a tenured object space [15]. Ungar succinctly sums up the generational hypothesis: "most young objects die young." A good account of the benefits of generational collectors may be found in [1].

Unfortunately, there are limits to the generational hypothesis. While the observation generally holds true of recently allocated objects, there is no stronger statement supporting the idea that the older an object is the longer it is likely to live. Ungar and Jackson improved the efficiency of a two-generation collector for Smalltalk by explicitly segregating some kinds of objects and by using a dynamically computed threshold for deciding when to promote objects [16, 17]. Barrett and Zorn extended this idea further by collapsing the two generations and using a movable "threatening boundary" to separate the spaces for short- and long-lived objects [2].

In the absence of information about the lifespans of objects, a number of proposals have been put forward to improve the efficiency of older generations. Most recently, work by Will Clinger and Lars Hansen, and by Darko Stefanovic, Eliot Moss, and Kathryn McKinley has examined the idea of "oldest-first" collectors [6, 14]. Our proposal differs in that we attempt to improve the collector by having knowledge about the objects in the heap. In this sense, it is orthogonal to "oldest-first" collection techniques and may even improve these techniques by placing objects where they are likely to be collected shortly after dying.

David Hanson developed a `malloc` implementation for the C language that allows for explicit placement of allocation requests [7]. Barrett and Zorn extended this work by developing a technique for profiling the allocation behavior of programs [3]. The two metrics they used were allocation size (since C lacks appropriate type information) and a summary of the state of the call stack at the time of allocation. Using the profile information generated from tracing the applications, they augmented the `malloc` library with a database consisting of allocation sites where objects always die quickly. This technique allows them to separate long lived objects from short lived ones, simultaneously reducing fragmentation, allowing the short lived objects to be allocated together on pages, and allowing these pages to be freed at once.

Seidl and Zorn have studied the notion of pretenuring objects based on how the objects are referenced and what the observed lifespans of the objects are [20]. Using profile-driven feedback from sample application runs, they classify objects allocated at given sites in the program into categories: short-lived, highly referenced, not highly referenced, and other. Combining this information together with information about the state of the call stacks at the time of allocation, they use two predictors, path point prediction and stack contents prediction, to place objects in the heap to maximize the efficiency with which the program uses virtual pages of memory.

Cheng, Harper, and Lee examine several techniques for improving the efficiency of generational garbage collection in the context of the TIL compiler and runtime system for ML [5]. This work uses profile-driven feedback from sample runs to pretenure long-lived objects to the older of a two-generation heap. To aid in pretenuring decisions, their system tracked the observed object lifetimes for each allocation site. Tracking information on a per-allocation site basis is much simpler than summarizing information about the dynamic call stack. Further, much of the benefits of the latter can be gained through inlining of methods containing allocation sites into calling contexts.

All of these approaches rely on off-line, full-run profiling of applications. These approaches make the basic assumption that the profile data is representative of how objects allocated at particular allocation sites behave. This assumption is limiting for a number of reasons. First, programs, especially concurrent ones, rarely run repeatedly with precisely the same behavior. Second, as programs scale to large sizes (for example, ones with heaps greater than 1 GB), the lifespans of objects and their relative proportions shift. Finally, as programs execute, they go through phases. Objects allocated at a given allocation-site may exhibit different lifespans depending on the phase in which they are allocated. Profile-driven feedback is unable to track such shifts in behavior within a running application. Our approach using weak references to fingerprint sampled objects avoids these problems with negligible overhead on the performance of the application.

3. DESIGN

Pretenuring or the automatic placement of objects at allocation time breaks down into the following set of tasks or choices:

1. Object selection or sampling
2. The fingerprinting mechanism
3. The level and kind of data gathered for each object
4. The summarization of gathered data
5. Method (re)adaptation

In this section, we outline approaches to each of these steps.

The platform we will use to implement our approach to dynamic adaptive tenuring is the ResearchVM.[2] This platform provides a number of facilities including:

- the ability to configure the number, size, and policies of generations forming the heap
- support for both application-level and VM-specific classes of weak references
- support for method recompilation including a number of optimizations such as the inlining of methods

The basic memory structure of the ResearchVM is described in [18].

[2]The Sun Laboratories Virtual Machine for Research or ResearchVM is currently being used as the Java virtual machine in the Java™ 2 Standard Edition for the Solaris™ Operating Environment and was formerly known as the ExactVM.

3.1 Sampling Techniques

Since sampling is done when objects are allocated and since the tenuring decisions are made at allocation sites, it makes sense that any decision on how to implement object sampling affects the allocation code at these sites. Because allocation is a frequent activity, it is good to sample objects in a manner that does not impact the common case.

The key to obtaining low overhead sampling is to avoid introducing overhead on the allocation of non-sampled objects. The ResearchVM memory system already contains several mechanisms by which this can be achieved:

- Local allocation buffers (LABs). We can chose to sample only the objects that cause a LAB refill. This is a low-pressure point in the allocation system. Care must be taken in this case since large objects tend to overflow LABs more often than small objects, resulting in skewed samples.
- Custom-allocator routines can be used to achieve cheap sampling for specific classes.
- The allocator routines already take a thread identifier as an argument. By comparing this identifier against a global variable or mask, we could obtain thread-specific sampling. To cover multiple threads, the global variable can be slowly rotated among the threads. Alternatively, we can have a per-thread flag indicating if a thread should be considered for sampling.

The most common technique for sampling will likely use the first approach. One might think that we could overcome the bias mentioned for large objects by sampling the first object allocated once the local allocation buffer is refilled. Unfortunately, this alternative introduces a different kind of bias since allocation of large objects, for example, character arrays, are often correlated with the allocation of other objects, such as the `String` objects managing those arrays. On the other hand, one could argue that such a bias is appropriate for large objects since these objects incur higher costs when copied by the garbage collector. In any event, customization of allocator routines is handled by having a per-class allocation function. So, another approach to the bias issue is to support several LABs per thread and to use custom-allocators to allocate different sizes of objects.

3.2 Fingerprinting

We have already discussed the importance of weak references as a mechanism for associating information with objects as they are allocated. The ResearchVM supports the four application-level weak references mandated by the Java Language Specification as well as several kinds of VM-level weak references. Examples of the latter kind include the hash tables supporting the interning of strings and the maintenance of loader constraints generated through class loading, resolution, and verification. For our purposes, the appropriate choices of weak references are either phantom references or VM-level references since neither of these kinds may result in a dying object to become reachable again when their "imminent death" operations are performed. Phantom references offer several advantages:

- the `PhantomReference` class is easily extended to include the data associated with the object.
- the collector need only queue the phantom references for processing instead of processing them during a collection phase.
- more of the tenuring infrastructure can be expressed in the Java programming language and is, thus, more portable to other Java virtual machines.

The disadvantage is that phantom references occupy space in the heap and may affect the rate at which collections occur. This disadvantage may be lessened in two ways: the sampled object and its phantom weak reference object could be allocated in one operation and if the ratio of sampled objects to non-sampled objects is low then the impact should be minimal. The advantage of VM-level weak references include:

- the lack of constraints on the format in which the fingerprinting information is represented. For example, we can maintain arrays of such references and process these arrays with bulk operations.
- the fact that these weak references are allocated outside the heap.

The primary disadvantage to this approach is that the resulting system is entirely VM-specific.

3.3 Gathering Statistics

There are several ways in which statistics on objects can be gathered:

- at the time an object is allocated and sampled
- when a sampled object becomes unreachable and dies
- in the event that the weak references are linked together, by iterating over the list of sampled objects

Combining the first two approaches allows us to incrementally maintain information about object lifespans without having to traverse potentially long data structures. For a simple, two-generation heap, the statistics may take the form of a mean object lifespan and deviation. More generally, though, it may take the form of one or more histograms to support tenuring decisions with more than one generation. The data on lifespans may be used to determine when objects allocated at a given site live long enough to justify being placed in a particular generation.

For sampled objects that are still living, we can best gather their birth time statistics in a histogram at the time they are allocated. As sampled objects die, we can remove their contribution to the appropriate bucket of the histogram, in the process maintaining an up-to-date histogram of the birth times and counts of reachable sampled objects. Using this histogram together with the current time, we can calculate the distribution of lifespans for objects allocated at a given site in the program. For sampled objects that become unreachable, we can gather their lifespan information in a second histogram directly. Combining these two distributions, we can easily determine where best to allocate or promote a given object.

3.4 Representation of Statistics

Allocation-site information may include static information such as the program counter of the allocation instruction or the method containing it as well as dynamic information summarizing the top frames of the call-stack, the identity of the thread allocating the objects, or the receiver-object of the method in which the allocation occurs. All of these may be used to refine the summary data gathered.

However, the two key pieces of information required for making tenuring decisions include the object's age and the site in the program at which it is allocated. As mentioned above, there are a

number of different metrics that may be used to measure age and lifespan. What is important is that these metrics are nondecreasing and simple to calculate. Following common practice, we will use ages and lifespans calculated in terms of the number of bytes allocated. Concerning allocation-site information, we have considered using summary information from the allocating thread's call stack. However, preliminary studies by our colleague, Tim Harris, indicate that simply tracking the allocation-site is a sufficient discriminator for our purposes [8]. Further, in the presence of inlining and dynamic recompilation, we have the ability to gain the benefits of call stack information without incurring the cost of having to examine thread stacks when allocating sampled objects.

We have proposed that the appropriate data structures for making tenuring decisions in a configurable generational framework are histograms of either birth time or lifespan distributions. The organization of these histograms must have fine enough resolution to distinguish the ages of objects in the individual generations. Further, the distributions of ages represented by the buckets need to reflect the relative rates and frequencies with which the individual generations are collected. One simple approximation of this might be to scale the per-bucket distributions logarithmically. Finally, we must support both adaptation to changing program behavior as well as some level of hysteresis to ensure that decisions, once made, are stable for a while. This goal may be achieved by either periodically purging lifespan data gathered about a particular allocation site—for example, when a new tenuring decision is made as to where to place objects allocated at that site—or by "aging" the information in the histogram—for example, with the use of a decay function.

3.5 Program Adaptation

There are a number of ways in which lifespan information may be used to improve the placement of objects:

- at allocation of an object
- when promoting an object
- when moving an object within a generation

All three of these decisions can be made through dynamic checks on the available lifespan statistics of a given object. For example, promotion of objects allocated at a given site might be improved if we know that these objects exhibit a bimodal behavior—for example, either dying young or living forever—and so move surviving objects directly to the oldest generation.

In the case of allocation, though, we can do better if we are able to recompile the methods containing the allocation sites where we wish to set a particular tenuring policy. For example, the ResearchVM provides mechanisms for scheduling a bytecode method to be recompiled. However, there are some cases where this is not possible. Examples include:

- the interpreter
- the reflection service
- `java.lang.Object::clone()`

The first two cases are typically not significant since objects of many types are allocated at the same site or sites. The third case can be handled by dynamically checking the lifespan information for an object but it is neither a frequent case nor is it clear that there is any relationship between the lifespan of an object and a copy of that object. So, for our purposes, the main impact of tenuring decisions is on dynamically compiled code and in the internal mechanisms of the per-generation collectors. When a change in policy is determined, the methods dependent on a given allocation-site are recompiled by the VM to implement the new object-placement policy. One reason to have different thresholds for changing decisions is to prevent the same methods from being repeatedly recompiled. In general, though, the result of this process only affects the performance of a running application and not its correctness.

4. CONCLUSION

We have presented a proposal for dynamically selecting and collecting information about object behavior in a generational framework. The proposal makes good use of facilities readily available in runtime systems like those provided by Java virtual machines: in particular, it makes use of weak references and of the ability to dynamically recompile methods. Further, we have evidence that the runtime overheads for the sampling technique and data management are low for a two-generation collector. Our plan is now to extend this technique to our dynamically configurable generational framework.

5. ACKNOWLEDGEMENT

We would like to thank Tim Harris for his excellent work in applying our methods to a two-generation system and demonstrating their practicality. In particular, his performance and tracing studies, his method for graphing object lifetimes, and his idea for using a small range of objects in the older generation to decide when to reverse pretenuring decisions helped clarify and validate our proposal.

In addition to Tim Harris, we would like to thank Steve Heller for many fruitful discussions. many

6. REFERENCES

[1] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[2] D. A. Barrett and B. Zorn. Garbage collection using a dynamic threatening boundary. In *Proceedings of SIGPLAN'95 Conference on Programming Languages Design and Implementation*, volume 30 of *ACM SIGPLAN Notices*, La Jolla, CA, June 1995. ACM Press.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, pages 187–196, Albuquerque, NM, June 1993. ACM Press.

[4] Y. Bekkers and J. Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[5] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[6] W. D. Clinger and L. 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.

[7] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, Jan. 1990.

[8] T. Harris. Dynamic adaptive pre-tenuring. In *Proceedings of the International Symposium on Memory Management*, Oct. 2000.

[9] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [4].

[10] Java™ 2 platform, standard edition, version 1.3 api specification. Available at: `http://java.sun.com/j2se/1.3/docs/api/index.html`.

[11] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[12] H. Lieberman and C. E. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983. Also report TM–184, Laboratory for Computer Science, MIT, Cambridge, MA, July 1980 and AI Lab Memo 569, 1981.

[13] D. A. Moon. Garbage collection in a large LISP system. In G. L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–245, Austin, TX, Aug. 1984. ACM Press.

[14] D. Stefanović, K. S. McKinley, and J. E. B. Moss. Age-based garbage collection. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 370–381, Denver, CO, Oct. 1999. ACM Press.

[15] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 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.

[16] D. M. Ungar and F. Jackson. Tenuring policies for generation-based storage reclamation. *ACM SIGPLAN Notices*, 23(11):1–17, 1988.

[17] D. M. Ungar and F. Jackson. An adaptive tenuring policy for generation scavengers. *ACM Transactions on Programming Languages and Systems*, 14(1):1–27, 1992.

[18] D. White and A. Garthwaite. The GC interface in the EVM. Technical Report SML TR–98–67, Sun Microsystems Laboratories, Dec. 1998.

[19] P. R. Wilson. Uniprocessor garbage collection techniques. In Bekkers and Cohen [4].

[20] B. Zorn and M. Seidl. Segregating heap objects by reference behavior and lifetime. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, Oct. 1998.