Garbage Collection Should Be Lifetime Aware

Richard Jones and Chris Ryder
{R.E.Jones,C.Ryder}@kent.ac.uk

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK

Abstract. We argue that garbage collection should be more closely tied to object demographics. We show that this behaviour is sufficiently distinctive to make exploitation feasible and describe a novel GC framework that exploits object lifetime analysis yet tolerates imprecision. We argue for future collectors based on combinations of approximate analyses and dynamic sampling.

1 Introduction

Currently, much GC effort is, in a sense, wasted on preventing the premature reclamation of live objects. it has long been known that programs' 'object demographics' are not random, but objects are allocated, interact and die in particular patterns. We argue that understanding and exploiting this regularity offers the key to reducing tracing cost by concentrating GC effort on those regions of the heap in which few objects are live. Neither generational nor concurrent GC provides a good solution to the huge heaps expected in the near future. Generational collection addresses only 'youthful' objects: it handles very new, 'middle-aged' and 'immortal' objects poorly. Concurrent collectors cannot reclaim any space until their trace of live objects is complete.

Lifetime Aware GC, LA, is a new paradigm for allocation and collection. Its goal is to use object death-time predictions to lay out objects in the heap in death-order and, at each collection, to scavenge only those objects expected to have died. LA is thus distinct from both generational [21] and older-first collectors [20].

Two requirements must be met in order to construct an LA collector. First, it must be possible to identify good object death-time predictors. Ideally, such predictions would be perfect but we argue that such accuracy is neither possible nor necessary. Second, we require a GC architecture that can exploit these predictions and tolerate prediction errors in an efficient manner. Below we describe how we acquire good predictors, describe in detail the LACE architecture and outline future directions of this project.

Suppose the object allocator has very good (but not necessarily perfect) advice on the expected lifetime of an object that it is about to allocate. How could it take advantage of such advice? Let us consider a horticultural analogy.

Some farmers grow a variety of crops. Each crop has a different expected ripening time (i.e. from planting to harvesting) but this may vary between plants according to environment, genetics, etc.

Full-Collection farmers walk up and down a single greenhouse, harvesting any ripe plants, and planting new ones. Some FC farmers may move immature specimens in order to get a contiguous bed for new planting. All FC farmers do a lot of walking.

The Generational farmer has 2 greenhouses. She plants seeds of every variety in a single nursery greenhouse. When that is full, she harvests any ripe plants but also has to replant any longer ripening varieties in her mature greenhouse. Occasionally, she harvests the mature greenhouse as well, in the same way as the FC farmer. She does a lot of replanting (and walking in the mature greenhouse).

The smart Lifetime-Aware (LA) farmer builds several greenhouses. All the crops in a single greenhouse are expected to have the same ripening time. Each greenhouse has a number of beds. The farmer places a label in each bed indicating when he expects to harvest that bed (i.e. the ripening time plus some slack). He only plants seeds in a bed if the label allows time for them to ripen; otherwise he prepares a new bed. The LA farmer only ever harvests beds with expired labels. He never has to replant a crop of a variety expected not to be ripe, although very occasionally he might have to replant an unripe plant to another greenhouse with a different ripening time. He does not walk much.

Such a scenario can be modelled by modifying the Beltway GC framework [2], a generalisation of region-based, copying GC (we describe Beltway in more detail in Section 5). Greenhouses are implemented by Beltway *belts*, beds by *increments* and expected ripening times by the expected lifetimes of objects. Belts therefore model expected lifetimes rather than generations. Thus, *belt* is both an allocation abstraction and a characterisation of a collection policy (e.g. when to collect) rather than just a mechanism for delaying collection (c.f. generation). Now, instead of collecting the first increment from the *lowest* belt (as per Beltway), we collect expired increments in expiry date order from *all* belts.

2 Related work

The requirements made of the GC by applications running in different environments vary in the priorities they assign to different performance metrics but, even within a single domain of interest, the memory behaviour of applications may differ substantially. No 'one size fits all' solution is possible even within a single environment [7, 16]. The simplest forms of regional organisation distinguish objects by their age or by their mortality. Large objects are commonly allocated into a separate area. Objects known to the JVM implementer to be immortal may be kept in an immortal area [1].

Generational GC segregates objects by age, with different generations collected at different frequencies (younger more frequently). The principle underlying generational collection is to concentrate effort on reclaiming those objects most likely to be garbage. Variations on generational collection include older-first collection [20, 19, 8] and the Beltway framework [2].

Most approaches to tailoring the GC to the behaviour of the mutator are based on generic heuristics, i.e. the same heuristic is applied to all objects, regardless of their class or the point in the program at which they were allocated. For example, generational GCs seek to exploit the weak generational hypothesis that *most* objects die young by first allocating *all* objects in a nursery. Adaptive tenuring techniques and hot-swapping collectors also address only 'average' object behaviour. We believe that programs exhibit distinctive behaviour at a much finer grain, related to the design of the program, and given good object lifetime predictions and an implementation that can

exploit them, significant performance gains can accrue by avoiding processing live objects before their expected time of death (improving both throughput and pause times) and reclaiming objects promptly after their death [15].

Four studies have taken a step in this direction. Cheng et al. [5] (CHL) gathered profiles from a generational collector for ML. By tagging objects with the program *site* that allocated them, they identify those sites that allocate objects promoted consistently by their collector. This advice is then used to allocate objects from those sites directly into the old generation. Because their pre-tenuring threshold is a function of their particular collector configuration, the wider applicability of this study is reduced. Blackburn et al. [3] remove this dependency from pretenuring advice by normalising object lifetime as a multiple of the maximum volume of live objects at any time. Harris [9] gathers feedback dynamically to pretenures objects. None of these techniques exploited any finer age distinction than 'short', 'long lived' or 'immortal' Hirzel et al. [11] find a correlation between object connectivity and object lifetime. They propose to use connectivity to convert the object graph into a tree of strongly connected components, thus removing the need for write barriers.

3 Experimental methodology

An *ideal* lifetime predictor would indicate, for each object allocated, precisely when it would die (thereby allowing its space to be reclaimed shortly thereafter). Unfortunately, ideal prediction is possible only in special cases. However, even a *good* predictor offers the promise of reductions in processing costs (e.g. unnecessary copying) and floating garbage by avoiding processing an object until soon after its death. Even if the prediction is 'wrong', either in the sense that the object turns out to be still alive or had died long before, the correctness of the collector is not compromised; it will simply either 'waste' effort. But this is what a generational collector would do in *all* cases.

We considered the role of allocation sites in phase behaviour (which sites allocate in which phases of the program?) and how well sites predicted object lifetimes. In contrast with previous work, we examine allocation patterns at a finer level of detail than 'short-lived/long-lived/immortal' and we find that the behaviour of very many sites—according to the program phases in which they participate or according to the lifetime distributions of the objects that they allocate—is strongly correlated. We further refine our predictors by considering *scope* (all objects are allocated on behalf of either the JIT compiler or the application—the run-time system merely serves these), and *package* (was the allocating method from Jikes RVM, a standard Java library or an application class). Such context is cheap to determine, and its exploitation requires neither specialisation of methods nor examination of threads' stacks. Our key results are

- Most sites allocate objects with lifetimes in only a small number of narrow ranges.
- Sites cluster strongly with respect to both the lifetime distributions of the objects they allocate, and the phases in which the site's objects live: 8 clusters account for almost all allocation.
- Clusterings are stable and largely unaffected by different program inputs.

The platform used was Jikes RVM version 2.3.1 with GNU Classpath 0.06, and benchmarks drawn from SPEC's jbb2000 [18] and jvm98 suites [17], and the new Da-

Capo suite, version β041020 [12]. We experimented with several different sizes of input: 1, 10 and 100 for jvm98, small, default and large inputs for DaCapo and different numbers of jbb2000 'warehouses' and threads.

To generate traces of allocation and death events, we modified Jikes RVM's compilers so that, as each allocating bytecode is compiled, a unique identifier *aid* is created, a map entry *(site, aid)* generated and additional instructions are emitted to write this *aid* into the object's header, We used a dynamic scoping mechanism to encode further context into an object's *aid* field when it is allocated without having to crawl the call stack. We thus distinguish allocation due to the compiler (either directly or indirectly), by Jikes RVM classes (`com.ibm.JikesRVM.*` or `org.mmtk.*` packages), Java libraries (`java.*` or `gnu.java.*`) and application classes. Objects allocated by the Jikes RVM in the boot-image are immortal and were not tracked.

The compiler-modification approach gave good performance and, importantly, allows the same framework to be used both to tag allocations in trace-gathering runs and to provide allocators with advice in performance runs. In order to capture death times, we force full heap GCs at 64KB intervals. Although this loses precision, little would be gained from collecting at finer intervals (although this granularity does exaggerate the space rental of short-lived objects). Performance was acceptable for trace gathering, extending the elapsed time for javac, for example, from approximately 10s to 3.5h (instead of a week with the Merlin trace gathering tool [10]).

4 Object lifetimes

A large object that lives for a long time incurs a greater GC cost than a small, short-lived one. This cost depends on the object's size (it occupies space in the heap, each reference field must be scanned, and the object may be copied) and the number of times the memory manager processes it. *Space rental* [3], the product of an object's size and its lifetime, provides an estimate of this cost to a simple, non-generational, tracing collector. We focus attention on (groups of) sites with high space rental.

We wanted to examine to what extent site predicts object lifetime. We characterise a site by its *lifetime density function*: the probability density function of the lifetimes of its objects allocated. From a program trace, we obtain a lifetime density function, $ldf_s(t) = v_s(t)$ for a site s where $v_s(t)$ = fraction of volume (bytes) of objects allocated by s with lifetimes in the range $(t, (t+\delta t)]$. Time is measured in bytes allocated.

Our results show that most sites allocate objects with a narrow range of lifetimes. Even better, other than for very short-lived data, many longer lived objects allocated by a site tend to die at the same point in the program (often this behaviour is repeated in phases). Such sites are good candidates for special treatment.

Programs typically have many hundreds or a few thousand sites. From an implementation viewpoint, a 1:1 mapping between site and allocator is undesirable: the heap would become badly fragmented if each allocator were to allocate into a different region, and cache performance might become an issue. We used the Kolmogorov-Smirnov Two-sample test [4] to cluster similar distributions. The advantages of this test are that it is computationally cheap, non-parametric and *distribution-free*: it does

	All sites			All packages					Jikes		Application			Java library		
				default			compiler		default		default			default		
Benchmark	1	4	8	1	4	8	1	4	1	4	1	4	8	1	4	8
compress	0.7	83.6	91.9	0.7	84.2	91.8	5.4	99.8	9.2	99.1	0	92	100	28	73.7	74.8
jess	0.4	99.1	99.9	0.4	99.2	99.9	0.2	96.9	9.7	99.9	0	99.8	100	13	84.7	87.4
raytrace	0.6	97.9	99.3	0.6	98.1	99.3	3	99.6	8.4	91.5	0	99.5	100	10.3	88	93.7
db	1	14.9	42.2	1	14.1	41.5	4.9	99.9	9.5	96.4	0	99.9	100	1	10.4	98.7
javac	0.7	88	89.4	0.7	88.1	89.5	1.3	100	3.7	99.7	0.1	30.6	99.8	0.8	98.4	98.8
mpegaudio	13.7	68.5	77.6	19.2	88	98.2	1.8	74.1	11.5	97.9	95.4	98.7	98.8	27.1	74.3	93.1
mtrt	0.6	97.5	99.2	0.6	97.8	99.3	3.6	100	7.3	86.3	0	99.2	100	6.5	92.3	96
jack	0.4	97.3	98.6	0.4	97.5	99	2.1	100	1	26.1	0.1	98.7	99.9	0.3	98.8	99.6
jbb2000	0.4	47.5	95.5	0.4	92.9	95.4	0.2	100	2.7	73.7	0	44.2	100	0.7	90.4	91.6
antlr	0.3	0.8	91.6	0.3	88.7	91.6	0.1	100	5.8	99.5	0	99.2	100	0.2	90.3	99.9
bloat	0.1	99.1	100	0.1	99.7	100	0.1	100	5.5	99.6	0	100	100	0.1	100	100
fop	5	58.1	60.9	5.4	56.6	85.4	0.2	95.5	9.5	98.2	21	52.3	99.7	1.8	7.3	80.9
hsqldb	0.2	6.5	91.3	0.2	6.5	99.9	0.2	100	2.9	100	0	99.8	100	0.1	100	100
jython	0.1	99.8	99.9	0.1	99.8	99.9	0.4	100	13.5	100	0	100	100	0.3	99.7	99.9
pmd	0.2	29.2	30.8	0.2	30.1	30.9	0.4	99.9	6.8	99.6	0	38.7	100	0.1	100	100
ps	0.4	1	100	0.4	1	100	0.4	100	0	100	7.4	100	100	0.8	99.9	100

Table 1: The volume of allocation (as percentages) due to the top 1, 4 or 8 lifetime clusters for large program inputs (speed 100 for jvm98, 8 warehouses and 2 threads for jbb2000, and 'large' for DaCapo).

not matter what the underlying distribution is — this is important as lifetimes are not normally distributed.

Table 1 shows the volume of allocation due to the top few clusters of sites. The number of statistically distinct lifetime density functions is only a small fraction of the number of allocation sites. The top 8 clusters account for over 90% of allocation in all but 5 benchmarks. These clusters also account for an average of 97.7% of space rental. Furthermore, when compiler, Jikes RVM and application allocation are clustered separately, only 4 clusters account for over 99% of both all compiler and all Jikes RVM space rental. We conclude that sites cluster sufficiently tightly to exploit with just a small number of allocators.

To exploit these behaviour patterns without having to gather trace data for every program input size, we want to be able to use data gathered for a particular program from one input for a different input. Although we cannot hope for expected lifetimes of clusters to remain constant as input grows, we do find that allocation sites share the same clusters from one input to another.

Table 2 compares stability of the top 8 clusters of the jvm98 and DaCapo benchmarks across three different input sizes. If, for every cluster i of input A, every site of cluster i appears in a single cluster j of input B, then the clusterings are equivalent and ARI=1 (e.g. across all jbb2000 configurations). With the exception of mpegaudio, all ARIs are sufficiently close to 1 to conclude that the clusterings change little across inputs.

This suggests that a feasible implementation strategy for LA collection might be to acquire clustering data from a single training run. These clusters would indicate which

SPEC jvm98	100:10	10:1	DaCapo	large:default	default:small
compress	1.000	0.998	antlr	1.000	0.892
db	0.998	1.000	bloat	0.946	0.859
jack	0.991	0.973	fop	1.000	0.987
javac	0.975	0.997	hsqldb	1.000	0.983
jess	0.991	0.996	jython	0.819	0.953
mpegaudio	0.642	0.638	pmd	0.957	0.984
mtrt	0.989	0.989	ps	0.949	0.976
raytrace	0.980	0.993			

Table 2: Adjusted Rand Index of top 8 clusters for three different input sizes (speeds 1, 10 and 100 for jvm98, and small, default and large for DaCapo).

sites had similar behaviours, and therefore should be allocated under the same policy. A lightweight run-time sampling (from a *few clusters* rather than *many sites*) could then be used to measure lifetimes, or to report when most objects allocated on a belt are dead; this also has the benefit of being able to respond to phase changes.

5 The LACE framework

The LACE collector is built on top of Beltway, but with different rules for collection of increments and promotion of objects that survive a collection. Beltway groups objects into one or more regions (*increments*), held on queues (*belts*), that are collected independently. Increments, the unit of collection, are collected in first-in-first-out order: the increment at the front of the lowest numbered belt is always collected first, and any survivors are copied to the last increment on that or a higher numbered belt. Beltway can thus be configured as any copying semispace, generational or older-first collector as well as a number of novel collectors [2]). A key insight behind Beltway is the separation of age (by varying the number of increments allowed on a belt) from incrementality (increments provide the unit of collection).

Beltway has been modified in a number of ways to support LA. For LA, we tune GC to application behaviour by associating a *policy* with one of more of each of the top n (n=8) allocation site clusters (with other sites associated with a default policy). Practically, each policy is mapped to a Beltway/LA belt. A policy includes the expected *time to die* (TTD) of the objects allocated by that cluster(s). It may also include *how* we expect cluster objects to die. For some clusters, objects allocated have varying lifetimes but all objects die at the same point (e.g. at the end of a phase). For other clusters, all objects share similar lifetimes but do not die together. We do not require predictions to be accurate. It is therefore possible that the GC discovers a few live objects when it collects an increment. A policy therefore also specifies how such survivors are to be handled. There are a number of possibilities.

- It is common for survivors to be immortal: these may be copied to an immortal belt.
- Other clusters exhibit a small number of distinct object lifetimes: here survivors may be copied to a belt corresponding to the next expected lifetime of this cluster
- If we have no better information, objects may be copied to the default belt or to a belt that is only collected as a last resort.

Using a mechanism similar to that used to tag objects for tracing (Section 3), we modified the compiler to pass a belt number to the allocator. The allocator uses the belt's policy to allocate the object into an increment on that belt.

- For clusters (belts) that expect all objects allocated within a phase to die together, we allocate a fresh increment stamped with a *time of death*, $TOD = now + TTD$ for that belt. All subsequent allocations for that cluster are made to the same increment until $now > TOD$, after which an allocation causes this increment to be *closed* and a new increment to be appended to the belt.
- For clusters in which objects are predicted to share lifetimes but not to die together (such as typical young generation objects), the increment is stamped with a TOD sufficient to allow the increment to hold many objects. The increment is only closed when an object is allocated at a time such that $now + TTD > TOD$.

In either case, the collector is invoked when we run out of space and all increments with $TOD \leq now$ are collected in increasing TOD order, with any survivors promoted according to their belt's policy.

LA uses more increments than Beltway but still requires collection to be complete, that is all garbage including cyclic garbage to be collected eventually. This means that any cyclic garbage structure will eventually be promoted to a single increment. This has two consequences: survivor policies must not contain cycles and this 'final' increment must be sufficiently large to handle the worst case (half the size of the heap if the increment is collected by copying). We therefore allow increments to grow (up to a maximum size) in units of pages.

As in Beltway, we use *frames*, 2^n-aligned contiguous regions to allow a fast write-barrier that records inter-increment references. However, increments may comprise more than one frame since a 1:1 mapping would exhaust a 32-bit address space. In the worst case, this might lead to a slower slow-path in the write-barrier. However, we expect good write-barrier performance for two reasons. First, there is ample evidence that most pointer 'lengths' are short and hence such writes will be caught by the fast-path (the source and target will be in the same frame). Second, because we expect our lifetime predictions to be reasonably accurate and because objects that die at similar times will tend to be allocated to the same increment, we expect few references from other increments, and hence expect remsets to be small.

6 Conclusions

We argue that next generation GC should exploit mutator behaviour. Analysis of program traces shows that program sites allocate objects with distinctive and largely consistent behaviour, that sites cluster strongly with respect to the lifetime distributions of the objects they allocate, and that these clusterings are robust against changes of input. We described LACE, a garbage collector framework that can exploit program knowledge to both avoid unnecessary copying and reduce floating garbage, but importantly tolerates imprecise allocation advice.

We plan to investigate how on-line sampling can be combined with our static cluster analysis. We also intend to investigate static analysis techniques, both to identify

clusters of similar sites and to identify points in the program where it is *likely but not guaranteed* that objects allocated by a site will be dead. We also believe that phase boundary analysis, e.g. [14], might be profitably combined with our approach.

Acknowledgements The authors are grateful for the support for this work from IBM through IBM Faculty Partnership awards and through EPSRC grants GR/R42252 and GR/R57140. Any opinions, findings, conclusions, or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

References

1. S.M. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *Int. Conf. on Software Engineering*, 2004.
2. S.M. Blackburn, R.E. Jones, K.S. McKinley, and J.E.B. Moss. Beltway: Getting around garbage collection gridlock. In *Programming Languages Design and Implementation*, 2002.
3. S.M. Blackburn, Sharad Singhai, Matthew Hertz, K.S. McKinley, and J.E.B. Moss. Pretenuring for Java. In *Object-Oriented Systems, Languages and Applications*, 2001.
4. Chakravarti, Laha, and Roy. *Handbook of Methods of Applied Statistics*, .I. Wiley, 1967.
5. P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *Programming Languages Design and Implementation*, 1998.
6. S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. *European Conf. on Object-Oriented Programming*, LNCS 1445, 1998. Springer.
7. R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *Int. Symp. on Memory Management*, 2000.
8. L.T. Hansen and W.D. Clinger. An experimental study of renewal-older-first garbage collection. In *Int. Conf. on Functional Programming*, 2002.
9. T. Harris. Dynamic adaptive pre-tenuring. In *Int. Symp. on Memory Management*, 2000.
10. M. Hertz, S.M. Blackburn, K.S. McKinley, J.E.B. Moss, and D. Stefanovic. Error-free garbage collection traces: How to cheat and not get caught. In *Int. Conf. on Measurements and Modeling of Computer Systems*, 2002.
11. M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the Connectivity of Heap Objects. In *Int. Symp. on Memory Management*, 2002.
12. http://ali-www.cs.umass.edu/DaCapo/gcbm.html. *DaCapo benchmark suite*.
13. L. Hubert and P. Arabie. Comparing clusterings. *Journal of Classification*, 1985.
14. P. Nagpurkar, C. Krintz, M. Hind, P. Sweeney, and V.T. Rajan. Online phase detection algorithms. In *Int. Symp. on Code Generation and Optimization*, 2006.
15. R. Shaham, E. Kolodner, and M. Sagiv. Estimating the impact of liveness information on space consumption in Java. In *Int. Symp. on Memory Management*, 2002.
16. S. Soman, C, Krintz, and D, Bacon. Dynamic selection of application-specific garbage collectors. In *Int. Symp. on Memory Management* 2004.
17. Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, 1999.
18. Standard Performance Evaluation Corporation. *SPECjbb2000 (Java Business Benchmark) Documentation*, 2001.
19. D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, Univ. Massachusetts, 1999.
20. D. Stefanović, K.S. McKinley, and J.E.B. Moss. Age-based garbage collection. In *Object-Oriented Systems, Languages and Applications*, 1999.
21. D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5), 1984.