# A Study of Java Object Demographics

Richard Jones Chris Ryder

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

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

## Abstract

Researchers have long strived to exploit program behaviour in order to improve garbage collection efficiency. For example, by using a simple heuristic, generational GC manages short-lived objects well, although longer-lived objects will still be promoted to an older generation and may be processed repeatedly thereafter. In this paper, we provide a detailed study of Java object lifetimes which reveals a richer landscape than the generational view offers.

Allocation site has been claimed to be a good predictor for object lifetime, but we show that object lifetime can be categorised more precisely than 'short-lived/long-lived/immortal'. We show that (i) sites allocate objects with lifetimes in only a *small* number of *narrow ranges*, and (ii) sites cluster strongly with respect to the lifetime distributions of the objects they allocate. Furthermore, (iii) these clusterings are robust against the size of the input given to the program and (iv) are likely to allocate objects that are live only in particular phases of the program's execution. Finally, we show that, in contrast to previous studies, (v) allocation site alone is not always sufficient as a predictor of object lifetime distribution but one further level of stack context suffices.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection); C.4 [*Performance of Systems*]: Measurement techniques

***General Terms*** Measurement, Performance, Languages.

***Keywords*** Memory management, Garbage collection, Java.

## 1. Introduction

Object-oriented languages such as Java and C♯ supported by managed run-times are being deployed more and more widely in commercially important applications. All these run-time systems provide garbage collection (GC) [27]. GC overhead can be measured by overall execution time, pause-time, memory footprint and so on; the importance of each factor varies according to the environment and application concerned. Many solutions have been offered to address an ever-changing variety of requirements, but finding scenarios that defeat general strategies for GC is straightforward. No single collector/configuration can provide the best results for all programs [14, 9, 16, 40, 39].

Nevertheless, certain commonly made observations concerning the memory behaviour of objects (*object demographics*) widely hold. Ungar's dictum that 'most objects die young' [45], and its dual, that some objects may live for a very long time, is true for a wide range of applications, although not for all.

The observation that different objects may exhibit different behaviours has led GC researchers and application programmers to attempt to exploit these properties to improve performance. Region-based memory managers segregate objects in order to apply different management policies to different regions. Examples include segregation by age (generational GC [29, 45], older-first GC [43]), by size (large-object areas [50]) or by mortality [13, 9]. Phase behaviour has been addressed through 'hot swapping' between collectors [36, 32, 40], by adaptively varying tenuring thresholds in generational collectors [47, 3] and even explicit GC requests.

We believe that such 'one size fits all' collectors, that treat objects uniformly without attempting to match the behaviour of the user program (the *mutator*), miss an opportunity to improve memory management performance. We expect significant performance improvements will be achieved if behaviour patterns can be better captured and exploited. The prerequisites for success are that (a) characteristic behaviour can be recognised, and (b) predictors of this behaviour can be identified, and (c) such predictors of behaviour can be exploited efficiently.

In this paper, we provide a detailed study of the memory behaviour of a wide range of realistic Java benchmarks, drawn from the SPEC jvm98 [41] and DaCapo [5] benchmark suites. In particular, we focus on the distribution of lifetimes of objects allocated by a *site* (a point in the source code such as `x=new...`). Like earlier studies, we find an association between allocation site and both object lifetime distribution and phase behaviour. However, we make the following novel contributions.

- We identify a richer demographic landscape than the 'short-lived', 'long-lived' or 'immortal' classification of earlier studies such as [13, 9, 8].
- We refine this classification by considering separately the behaviour of objects allocated by the JIT compiler, the run-time system, Java library code as well as those allocated directly by the application program.
- Most importantly for practical GC implementation, we find that the sites of a single program run exhibit only a few distinct object-lifetime distributions, i.e. they cluster strongly.
- These clusterings are stable across different program inputs, making exploitation feasible.
- A very small number of lifetime distribution patterns dominate; these cannot all be classified by simple characterisations such as 'short-lived', 'immortal' and so on.
- Adding one further level of stack context significantly improves over allocation site alone as a predictor of object lifetime distribution for Java library classes.

Note that this paper focuses only on demographics; we do not provide here results of any implementation exploiting our observations.



*ISMM '08* June 7–8, 2008, Tucson, Arizona, USA.

***Structure of this paper*** Section 2 reviews related work and discusses the background to this research. Section 3 presents a case study of the object lifetime behaviour of a range of Java programs drawn from the SPEC jvm98 and DaCapo benchmark suites. Section 4 explains our clustering methodology in detail, and Section 5 shows that such clustering is robust against changes in the size of program inputs. We investigate calling context depth in Section 6 and conclude in Section 7.

## 2. Background and related work

The requirements made of the GC by applications running in different environments also 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. For example, studies of client-side, desktop applications [16, 40, 39] demonstrate significant differences in performance depending on which collector was used. Clearly, no 'one size fits all' solution is possible even within a single environment.

### 2.1 Adapting GC to mutator behaviour

In order to meet the challenges imposed by different applications, environments and user requirements, many different GC architectures and strategies have been adopted.

The simplest forms of regional organisation distinguish objects by their age or by their mortality. Where objects are predominantly managed by moving collectors, large objects are commonly allocated into a separate, stationary, large object area [50]. If it is known that an object is certain (or at least, highly likely) to live for the duration of the program, then it is better to avoid processing it altogether (although the collector must be able to identify its children); such objects may be kept in an immortal area [7].

The most common form of regional organisation is generational GC [29, 45]. Observing that most objects die young [45, 52, 21, 37, 4, 15], generational GC segregates objects by age into different regions, 'generations', in the heap. Different generations are collected at different frequencies (with younger ones collected more frequently than older ones), and may be managed by different collectors. Thus the principle underlying generational collection is to concentrate effort on reclaiming those objects most likely to be garbage. An important issue is how to decide when an object should be 'promoted' from a younger to an older generation. Too early risks objects not only dying soon after promotion (thereby causing too frequent collections of the older generation) but also encourages nepotistic promotion of the referents of tenured (i.e. promoted) garbage; too late demands either larger young generations (and consequently longer pause times for stop-the-world collectors) or more frequent young generation collections.

Solutions to the promotion problem have included multiple generations, variable-sized nursery generations [2] and different promotion thresholds (copy counts, allocation thresholds, etc) [51]. Recent variations on generational collection include older-first collection [43, 42, 19] and the Beltway framework [9].

It is also common for applications to exhibit phasic behaviour [49, 25]. Examination of the volume of the data live at any time reveals features such as ramps, plateaux, spikes and repetition to be common. The ability for GCs to adapt to changing mutator behaviour has been shown to offer performance improvements. One approach is to allow different collectors to be 'hot-swapped' at runtime [36, 32, 40]. Some generational GCs allow promotion thresholds to be varied dynamically in order to limit young generation pause-times [47, 3]. Programmers are also known to attempt to force a GC explicitly although for Java this only "suggests that the Java Virtual Machine expend effort toward recycling discarded objects" [18].

### 2.2 Parameters for adaptation

Both region-based organisation and collector adaptation to changing mutator behaviour require the existence of a metric to guide behaviour. In some cases, the metric and the behaviour is simple. For example, if the size of an object exceeds some (usually predefined) threshold, then it is allocated in a large object area. Note that this heuristic often encompasses three notions: (i) it is expensive to move large objects, (ii) large objects are likely to live for a long time (they take a long time to construct) and so some collectors allocate large objects directly into an old generation [48], and (iii) it is worth paying any additional costs of free-list allocation [6].

For other cases, the identification of suitable parameters and metrics is harder or more limited. For example, selection of immortal objects is often determined by the virtual machine implementer [7] and hence restricted to VM objects rather than application objects, although heuristics derived from analysis of program traces are also possible [10]. Selection is often broadly based, using the object's 'kind' (e.g. some prefix of its fully qualified Java class name) rather than by individual instance.

### 2.3 Demographic studies

Most approaches to tailoring the GC to the behaviour of the mutator are based on heuristics. Furthermore, these heuristics are usually generic, by which we mean that 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 (in order both to reduce expected pause-times and to reduce the overall amount of collection work done) by first allocating *all* objects in a nursery. Similarly, adaptive tenuring techniques and collector hot-swapping address only 'average' object behaviour. One exception to this generic approach creates a collector tailored to the program being compiled [44]. Nevertheless, the immediate corollaries of the generic adaptive strategies (for example, in the case of the generational hypothesis, that *some* objects do not die young) have not been explored in any depth.

One of the most thorough studies of memory behaviour for Java programs [15] is concerned with reference densities, proportions of array objects and so on, but does not match our intuition as programmers. Programs are designed with phases and behaviours that model problem domains. The big claim for object-orientation is that it helps to cross the chasm between design and implementation. For example, compilation requires syntax and semantic analysis, code generation and so on. The object-oriented compiler designer thinks in terms of classes and relationships between them that model 'real world' concepts; such as symbol tables, variables, types, intermediate representations and so on; of execution as a sequence of phases in which some kinds of activity dominate others; and not of the relative volume of array objects in the heap, for example.

In contrast, the focus of the DaCapo group is to demonstrate the broad range of the programs in their benchmark suite [5]. Programs are characterised by allocation volume and memory footprint, 4 MB nursery survival rate, heap composition (by object size) over time and pointer distance, and other metrics. Such heap-related metrics are important to designers of generational collectors.

Generational and other hypotheses and implementations, such as adaptive tenuring or hot-swapping, capture only the gross behaviour of programs. Program behaviour is not random. Intuitively, we believe that programs exhibit distinctive behaviour at a much finer grain and that the grain of this behaviour is related to the design of the program as realised (for object-oriented programs) by the relationships of classes and methods.

Given such predictions and an implementation that can exploit them, we expect significant performance gains to accrue. For example, the collector could both avoid processing live objects before their expected time of death (improving both throughput and pause times) and reclaim objects promptly after their death (fragmentation might be reduced if objects with similar lifetimes are allocated together). Shaham et al. [38] suggest a theoretic upper-bound on space savings of 39% through early reclamation of garbage but offers no implementation that might realise these savings.

### 2.4 Pretenuring

Four studies have taken a step in this direction. Cheng et al. [13] (CHL) gathered profiles from a generational collector for ML. They tag objects with the point in the program (*site*) that allocated them and inspect the tags of dead objects at each collection. From this profile, they identify those sites that allocate promoted objects consistently (i.e. where the volume exceeds a threshold of 80%) in their collector. This advice is then used to allocate objects from those sites directly into the old generation, thereby reducing GC times by 12–50%. Because their pre-tenuring threshold is a function of their particular collector configuration, the wider applicability of this study is reduced.

Harris [20] pretenures objects in a Java virtual machine using dynamic feedback from statistics gathered online. Dynamic pretenuring allows adaption to phase behaviour. The disadvantage of his sampling technique, based on overflows of local allocation buffers, is that it tends to oversample moderately large objects. Nevertheless, it demonstrates the feasibility of dynamic sampling. Harris also observed that some stack context distinguished tenured from non-tenured instances of some `java.util` objects.

Blackburn et al. [10] extend CHL's approach; they remove implementation dependency from pretenuring advice by normalising object lifetime as a multiple of the maximum volume of live objects at any time, *livesize*. They classify sites as generating predominantly short-lived, long-lived or 'immortal' objects, using a threshold heuristic of 20% of maximum live size for short-lived objects, and as immortal if objects die more than halfway between their birth and the end of the program (and hence not worth reserving copy space for). Their two-generation collector prevents any exploitation of a finer distinction of object age. Because their advice is implementation-independent, they can also combine advice from runs of different programs by ignoring application-specific data. Just using such 'build-time' advice improved performance by up to 8% in tight heaps. Combining build-time and application-specific advice led to reductions in GC time of up to 32%.

Marion et al. [28] leverage data mining of an object-lifetime knowledge bank and a simple static analysis of the 'micro-patterns' [17] of program source code to provide program-specific advice, achieving improvements in GC time of 6–77% for several SPEC jvm98 programs. Although the knowledge bank is derived off-line from data mining program traces, advice generation is quick and provides 'true' prediction (i.e. not derived from a past run of the same program). Although their technique tends to identify only sites allocating immortal (rather than other long-lived objects), it nevertheless supports our intuition as programmers.

## 3. Object lifetime behaviour

Other than the studies in Section 2.4 above, the weakness of most generational GC schemes is the 'one size fits all' assumption that underlies segregation by age. Thus, all objects are assumed to be short-lived until they prove otherwise, and no provision is made for different allocation sites to allocate objects with different lifetime characteristics, and no account is taken of phase behaviour. In this section, we examine in detail the object lifetime behaviour of programs drawn from the SPEC jvm98 [41] and DaCapo [5] benchmark suites and reveal a richer landscape of behaviour.

### 3.1 Data capture

We modified Jikes RVM to generate traces of allocation and death events, using the MemTrace system [28] and a calling context tree builder [11]. MemTrace modifies the Jikes RVM compilers in two ways. As each method is compiled, its allocation sites are recorded and the allocation routines modified to tag each object allocated with its site and the position in the calling context tree of the call containing the allocation and to emit an allocation record. One advantage of this approach is that it uses the same framework for data gathering as it does for specialisation of allocation in performance runs (as for example used by Marion et al. [28]).

The platforms used here were Jikes RVM [1], versions 2.4.6 for the work described in Section 6 and 2.3.3 in all other sections. Classes loaded at both build-time and run-time were compiled by the non-optimising baseline compiler.

We profile with a baseline configuration because our focus in this paper is on on application objects and their behaviour, rather than compiler behaviour. None of the Jikes RVM compilers perform any object allocation optimisation (such as object inlining), so the number of application objects allocated and the point in the program at which they die is independent of the choice of compiler. However, the optimising compiler allocates significantly more data than the baseline compiler, causing a large increase in the time taken to generate and analyse allocation traces. In small programs, the allocation performed by the optimising compiler can dominate, making it difficult to see the application behaviour. We wished to minimise the effect of compiler-allocated data on exaggerating the lifetime of a object (measured in bytes) — as the size of any other objects allocated, e.g. by the compiler, between its birth and death contributes to its lifetime. Hence using the baseline compiler gives two benefits, faster trace generation and clearer application behaviour. Technical note: one drawback of the compiler approach is that, without more extensive modifications, 5 sites that exclusively allocate array classes internal to Jikes RVM through `create` methods[1] hijacked by the compiler cannot be distinguished. Their presence cannot be discovered by stack walking, either. We also distinguish between scopes and categories of allocation. We record separately objects allocated by the compiler (*compiler scope*) and by the application (*default scope*). Similarly, we distinguish sites in Jikes RVM, Java library and other packages.

The collector configuration was MMTk's SemiSpace collector. To capture object death events, we force full collections at 64 KB intervals; after each collection, we use the GCspy framework [33] to sweep through dead objects to log death events. Hence object allocation times are accurate, but death times are only accurate to this granularity. Although this exaggerates the lifetime of short-lived objects [8], we do not consider this to be problematic as no tracing collector could take advantage of greater precision.

### 3.2 Residency

In the remainder of this section, we discuss the lifetime behaviour of objects allocated by sites. We observe that many sites demonstrate similar behaviour (similar lifetime distributions) and we group these into clusters. We postpone discussion of our clustering algorithm until Section 4. First, we consider *residency*, the volume of data live (i.e. reachable from the program's roots by following chains of pointers) at any moment.

[1] In classes `VM_CodeArray`, `VM_WordArray`, `VM_AddressArray`, `VM_OffsetArray` and `VM_ExtentArray`.

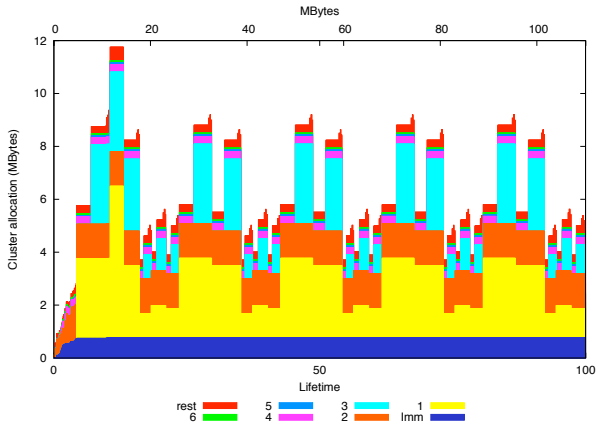



**Figure 1.** Residency against time for the major clusters for compress, speed 100, under Jikes RVM, BaseBase compiler.

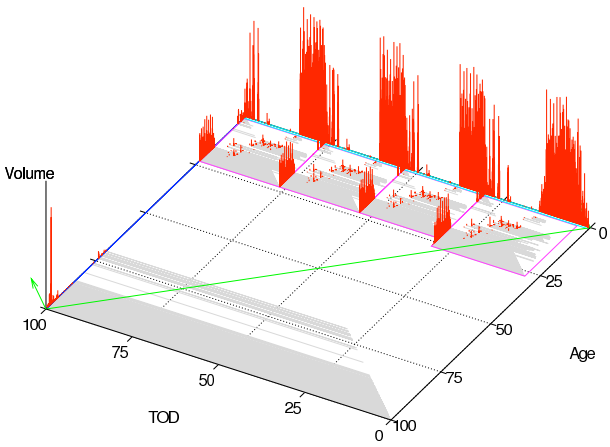



**Figure 2.** Lives of objects allocated by a single site for javac, input 100, under Jikes RVM, baseline compiler. The site accounts for nearly 5% of total allocation.

Figure 1 shows the residency of objects allocated by different (clusters of) allocation sites for SPEC jvm98 compress [2], using its largest input (100), running on Jikes RVM, with the baseline compiler used for both building the image and at run-time. Although compress is a comparatively simple program, its residency profile is particularly clear. Other benchmarks also show clear and particular residency profiles. In Figure 1, the $x$-axis, time, is labelled both as a percentage of total allocation and in megabytes allocated so far. The $y$-axis show the residency in megabytes at any point in the program's execution.

Earlier studies have shown such residency graphs either without any subdivision of the data allocated, or by stratifying the allocation by object size, pointer count or mutation distance [5]. In contrast, we show allocation grouped into clusters of sites whose objects display similar lifetime distributions. Thus, the dark band at the bottom of Figure 1, labelled 'Imm', represents those sites that only allocate immortal objects (i.e. objects that survive to the end of the program); the light band labelled '1' a cluster of objects that are neither short-lived nor very long-lived, and so on. A number of trends are apparent, and similar observations can be made for other benchmarks[3].

- Some objects, mostly but not exclusively created near the start of the program, live forever (cluster 'Imm'). Many of these are Jikes RVM data.
- There are 5 major phases (of just over 20 MB each). compress is a modified Lempel-Ziv compression algorithm, which iterates 5 times, compressing and uncompressing 5 tar files.
- Each of the major phases has sub-phases, each of which has further sub-phases, an so on.
- Data allocated by some 'long-lived' allocation sites appears to live for much longer than others.

A sophisticated collector might treat these groups of sites separately, placing objects allocated by different (groups of) sites in different regions and managing these regions with different policies. Note that this organisation would not necessarily be generational: regional segregation is not simply by age.

[2] compress is not a particularly interesting benchmark for GC research. Much of its behaviour is due to loops in the SPEC harness. However, its virtue here is that its comparatively simple behaviour reproduces clearly for monochrome printing.

[3] See www.cs.kent.ac.uk/projects/gc/demographics.

### 3.3 Object lifetimes

Let us drill down to consider the lifetime characteristics of a single site. Inspection of objects' lifetime behaviour shows that most sites allocate objects with a generational behaviour, i.e. most objects die young, as is to be expected. These objects are well handled by traditional generational collectors. However, the behaviour of other sites is more interesting, exhibiting 'GC-devils' [46] that do not conform to the generational hypothesis.

The 3d plot in Figure 2 shows the lives of objects allocated by a single site by javac, input 100. The origin is the top, right-hand corner. The $x$- and $y$-axes show the time at which an object died (TOD) and its age (both given as percentages of the total allocation of the program). Thus a bar of height $z$ at point $(x,y)$ indicates that $z$ bytes of data died at time $x$, having lived for $y$ bytes. Note that there can be no bars in the bottom right-hand triangle of the grid (below the diagonal line) since an object's time of death must be greater than its age. To make the life spans of objects clearer, we also plot a 'shadow' of each bar on the $xy$-plane: the shadow shows the period for which the objects represented by the vertical bar were alive.

The site in Figure 2 allocates an array of `char` for `Strings`. It is a significant allocation site for javac, accounting for nearly 6% of total allocation. Again, we can see four distinct phases: javac is Sun's JDK 1.02 compiler which, for input 100, compiles the JavaLex.java scanner four times. 85% of the objects this site allocates are very short lived (the large bars at the back, almost sitting on the $x$-axis), but a significant fraction of allocation lives until the end of each phase. A few objects (bottom, left-hand corner) are immortal. We examine this behaviour further in Section 6.

Some patterns of behaviour turn out to be remarkably similar, across benchmarks and across sizes of input. For example, it is common for sites to allocate objects that may or may not be short lived but all die together: imagine a loop that allocates objects and adds them to a collection (such as a list) which dies at the end of the loop. Figure 3 shows such a cluster of sites for DaCapo's hsqldb benchmark (default size). These 4 sites account for nearly 17% of allocation (each of the large vertical bars accounts for nearly 1%) but over 94% of *space rental* (the product of volume and lifetime, a useful measure of the importance of an object to the memory management system).

Another common pattern is for sites to allocate data with a bimodal lifetime pattern — we see this in each of the phases of javac in Figure 2 and in Figure 4. The latter figure shows a cluster

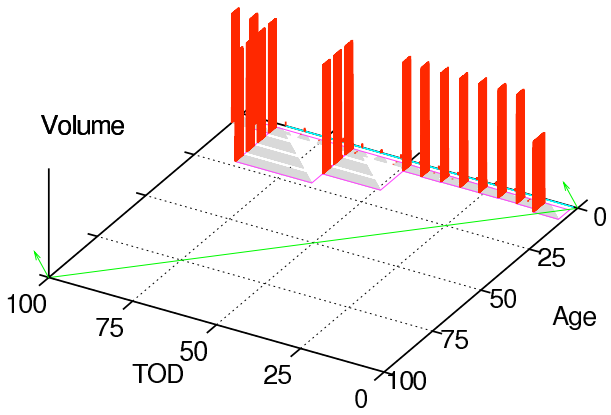



**Figure 3.** Lives of objects allocated by a cluster of sites for hsqldb, default input, under Jikes RVM, BaseBase compiler. These sites account for approximately 16% of total allocation.

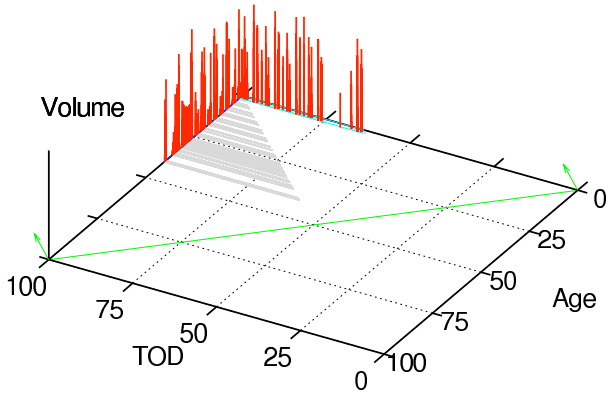



**Figure 4.** Lives of objects allocated by a cluster of sites for fop, under Jikes RVM, BaseBase compiler. These sites account for over 19% of total allocation. Short-lived and immortal and immortal allocations for this cluster are 8.29% and 9.27% respectively.

of 18 sites for DaCapo fop (default size), that account for 19% of allocation (16% of space rental). These sites are active in the last third of the program's execution, and allocate, in nearly equal proportions, objects that either survive until the end of the program or are discarded quite soon.

Graphs like these confirm — and explain precisely why — simple generational strategies miss opportunities for better object management.

## 4. Lifetime clusters

All but the most trivial programs have far too many allocation sites to treat each one separately. For example, the javac benchmark (for input 100) with the Jikes RVM BaseBaseSemiSpace configuration uses 1173 allocation sites; it would clearly be impractical to treat each site specially. In this section, we consider how to characterise the lifetimes of objects allocated by a site, and how to identify sites whose objects exhibit similar lifetime patterns. In the next section, we consider the extent to which lifetime behaviour is independant of the input to the program.

### 4.1 Lifetime density functions

We calculate from a program trace the lifetime of each object allocated, and summarise each site as a histogram ($N = 2000$ buckets)

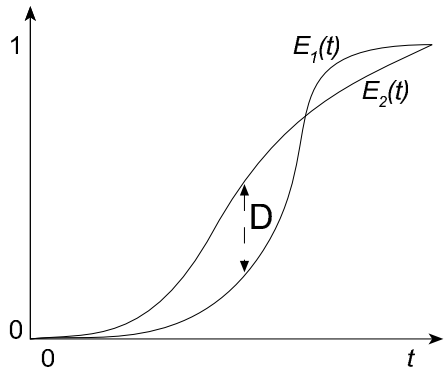



**Figure 5.** Kolmogorov-Smirnov Two-sample test.

of lifetimes. Thus, we obtain a *lifetime density function*, $ldf_s(t)$, an estimate of the volume of data allocated by a site $s$ that have lifetime $t$. Visually, the *ldf* is a projection onto the $yz$-plane of the 3d diagrams above.

### 4.2 Cluster analysis

Evidence from prior work [21] suggests that, because computation tasks are achieved through the collaboration of sets of objects, those objects have similar memory behaviour: "objects that are allocated together, die together". We wish to partition (cluster) the set of sites of a program by the memory behaviour of the objects they allocate. We characterise each site by its *ldf*, and consider two sites to behave the same if their *ldf*s are similar. Thus the input to our cluster analysis is a set of (sites and) *ldf*s and a confidence level; the output is a partitioning of the set. A good clustering method will lead to a partitioning where the intra-partition similarity of *ldf*s is high but the inter-partition similarity is low [24].

We compare *ldf*s statistically, treating them as probability density functions. We test, with a certain confidence $p\%$, whether two *ldf*s are similar, i.e. the chance that the difference between them is significant is less than $p\%$. We use the Kolmogorov-Smirnov Two-sample test (*KS2*) [12]. The advantages of this test are that it is non-parametric and *distribution-free*: it does not matter what the underlying distribution is — this is important since we do not expect lifetimes to be normally distributed, for example. A further advantage of KS2 is that it is computationally cheap as the null hypothesis can often be rejected after comparing only the first few points of each distribution.

KS2 compares two sample distributions by calculating the largest gap $D$ between their cumulative distribution curves, in our case

$$E_s(t) = \int_0^t ldf_s(t)$$

Formally, KS2 rejects the null hypothesis, that the two samples come from the same distribution, with confidence $p\%$ if the $D$-statistic $D = max|E_1(t) - E_2(t)|$ exceeds a critical value, which depends on the significance level $p$ required and the size of the samples (see Figure 5). Our greedy, 'gravitational' algorithm (Figure 6) places each site in the cluster of the most important site in its neighbourhood. We choose our definition of importance to be space rental, since objects that are larger or that live longer, put more pressure on the memory manager. However, we get similar results if we consider volumes (see Figure 7).

First, we identify and remove all sites allocating only immortal objects, since these would be allocated in a region not subject to garbage collection. These immortal sites are assigned to a single cluster, $C_0$. Then, we consider each 'mortal' site $s$ in order of space rental (i.e. the sites most important to the memory manager were

```
add immortal sites to cluster 0;
sort mortal sites by space rental;
foreach mortal site {
  foreach cluster in clusterList {
    if (match(site, cluster)) {
      add site to cluster;
      next site;
    }
  }
  add site to new cluster;
}

match (site, cluster) {
  if mortality(site) != mortality(cluster)
    return false;
  return KS2test(site, cluster, SIGNIF);
}
```

**Figure 6.** The clustering algorithm.

considered first). $s$ is added to the first cluster $C$ for which $ldf(s)$ is similar to $ldf(C)$ according to KS2 at the 1% significance level. If no similar cluster is found, $s$ is added to a new cluster $C'$, and that cluster is appended to our list of clusters. The $ldf$ of a cluster is set to be that of the first site added to it.

Clustering reveals that applications exhibit only a few statistically distinct lifetime behaviours. Figure 7 shows the coverage (as percentages of the total volume allocated or of space rental used) by the top 32 allocation sites or clusters. The curves are geometric means for the DaCapo benchmarks; those for the much smaller SPEC jvm98 benchmarks are very similar. Aggregating sites into just a few clusters accounts for almost all allocation. Overall, only a small number of clusters account for almost all space rental used. A few more clusters are needed to account for the same fraction of volume. Unsurprisingly, fewer partitions dominate in Jikes RVM itself than in the applications (particularly by volume). In the compiler, just two *sites* dominate; these allocate arrays for reference maps for the collector.

An important property of the clustering algorithm is that it preserves the common, generational behaviour. For example, one strongly generational cluster of antlr captures 88% of allocation (and 10% space rental). Within this cluster, 92% lives for less than 332 KB and almost all the remainder is dead within 12 MB.

## 5. Input sizes

So far, object lifetime distributions have been revealed to cluster well, and most distributions turn out to comprise a small number of distinct behaviours (i.e. two-dimensional polygons in the 3D diagrams). It would clearly be unrealistic if, in order to exploit these behaviour patterns, it was necessary to gather trace data for every program input size. It would be better if it were possible to use data gathered from one input to a particular program to inform the allocator for a different input. Unfortunately, the lifetime behaviour of an allocation site varies from input to input: for example, compare the behaviour of sites from hsqldb with the default input (Figure 3) with the behaviour of the same sites using the 'small' input (Figure 8).

However, the key question is not 'does an allocation site generate the same lifetime behaviour regardless of input' — for it clearly does not — but 'do allocation sites share the same clusters from one input to another?' Our intuition is that they will, because the objects they allocate collaborate to perform the same task, regardless of the size of that task.

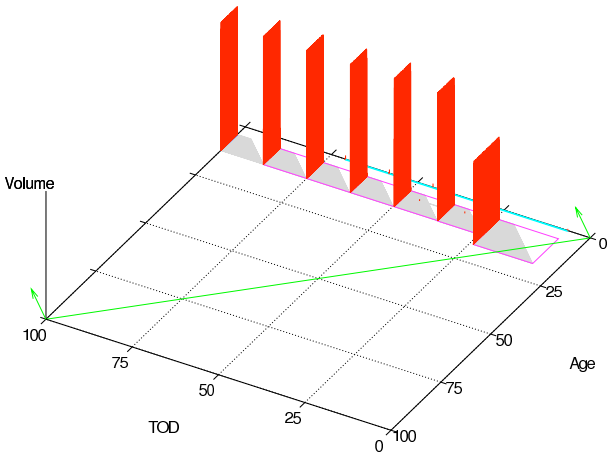



**Figure 8.** Lives of objects allocated by the same cluster of sites for hsqldb as in Figure 3 but using the 'small' input. These sites account for over 15% of total allocation.

We applied our clustering algorithms to each of the SPEC jvm98 and DaCapo benchmarks[4], and compared the clusterings derived from different input sizes provided. It is not unusual for major clusters to contain just 1 site, in which case the answer to our question is simple, but in general, it is non-trivial to compare different clusterings: there is no 'ideal' clustering method [30]. Further, all the indices of similarity of which we are aware have undesirable properties. They may assume that both clusterings use the same data set, be sensitive to the size of the data sets or the number of partitions, may have ranges that are difficult to interpret or make comparision between the comparison of one pair of clusterings and another difficult.

Rand Indices [34] measure the similarity between two clusterings. Given two partitions $U$ and $V$ of a set, count $(a)$ the number of pairs of items placed in the same clusters in $U$ and in $V$; $(b)$ those in the same cluster in $U$ but in different clusters in $V$; $(c)$ those in different clusters in $U$ but in the same cluster in $V$; and $(d)$ those in different clusters in $U$ and in $V$. The Rand Index, $(a+d)/(a+b+c+d)$, compares the number of agreements $a+d$ with the number of disagreements $b+c$. Its value is 1 when two partitions agree completely, but unfortunately its expected value is not 0 for two random partitions. For this, we require the Adjusted Rand Index (ARI, details omitted but see [23]).

For the purposes of comparing clusterings of allocation sites, only the most dominant sites are of interest. We compared the stability of the top $N$ clusters for each of our benchmarks, for $N = 16$, 32 or all clusters. We found that the stability of the most significant clusters (e.g. the top 16) is better than that of less significant clusters (e.g. top 32 or all clusters)[5]. Table 1 compares stability of the top 16 clusters of 4 DaCapo benchmarks, including the best and the worst. Each row compares small v. large, small v. default, and default v. large inputs, for application objects, Jikes RVM objects (excluding the compiler), library objects (excluding those used by the compiler), and all compiler objects. If, for every cluster $i$ of input $A$, every site of cluster $i$ appears in a *single*

[4] antlr, jython, pmd and ps ran successfully under Jikes RVM 2.3.3 and used different workloads for their default and large inputs.

[5] Computing the ARI requires that the underlying data sets for both inputs are identical; we therefore add to each clustering where necessary an extra partition consisting of any sites that were present in the clustering for one input but not the other.

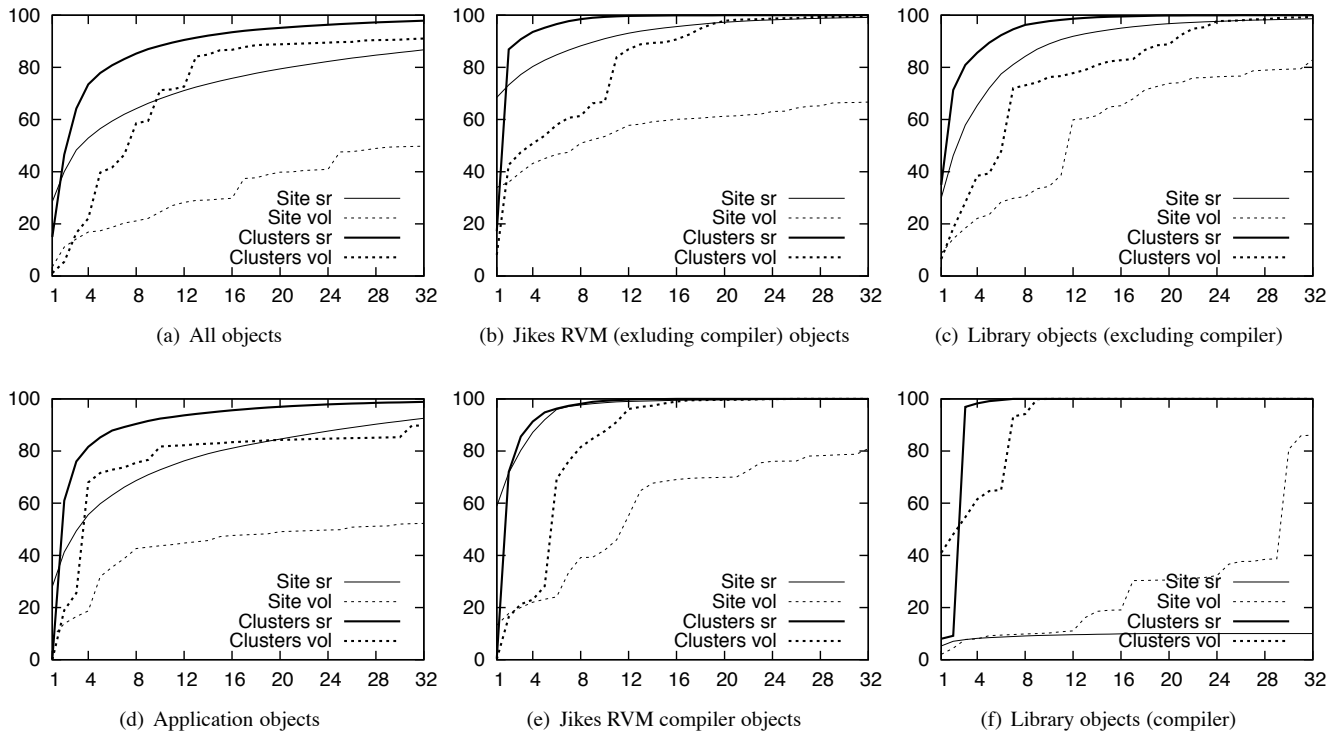



**Figure 7.** Geometric mean of site (thin lines) and cluster (thick lines) coverage by space rental (solid) and volume (dashed) for the DaCapo benchmark suite, segregating allocation by category. Each figure shows the fraction of that category's allocation covered ($y$-axis) by the top $x$ clusters.

| Size | antlr | | | | jython | | | | pmd | | | | ps | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Comp | App | Java | VM | Comp | App | Java | VM | Comp | App | Java | VM | Comp | App | Java | VM |
| S-D | 0.850 | 0.472 | 0.972 | 0.978 | 0.717 | 0.987 | 0.998 | 0.961 | 0.870 | 0.994 | 1.000 | 0.960 | 0.959 | 0.973 | 0.959 | 0.906 |
| S-L | 0.764 | 0.441 | 0.927 | 0.979 | 0.739 | 0.740 | 0.997 | 0.972 | 0.717 | 0.987 | 0.998 | 0.961 | 0.811 | 0.753 | 0.957 | 0.893 |
| D-L | 0.999 | 1.000 | 0.988 | 1.000 | 0.999 | 1.000 | 0.987 | 0.991 | 0.820 | 0.980 | 1.000 | 0.973 | 0.572 | 0.668 | 0.994 | 0.916 |

**Table 1.** Adjusted Rand indices for cluster comparisons for 4 DaCapo benchmarks (including the best and the worst). Each row compares small v. large (S-L), small v. default (S-D), and default v. large (D-L) inputs, for application objects, Jikes RVM objects (excluding the compiler), library objects (excluding those used by the compiler), and all compiler objects.

cluster $j$ of input $B$, then the clusterings are equivalent and ARI=1; otherwise $0 \leq \text{ARI} < 1$.

Java library clusters are highly correlated across all input sizes for all the DaCapo benchmarks, as are Jikes RVM clusters. Compiler clusters correlate moderately well for the top 16 clusters, but deteriorates significantly if more clusters are considered. Application clusters also tend to correlate well, except for antlr using the small input. We conclude that sites that tend to allocate objects with lifetime distributions that are similar to each other, tend to do so regardless of the size of the input to the program.

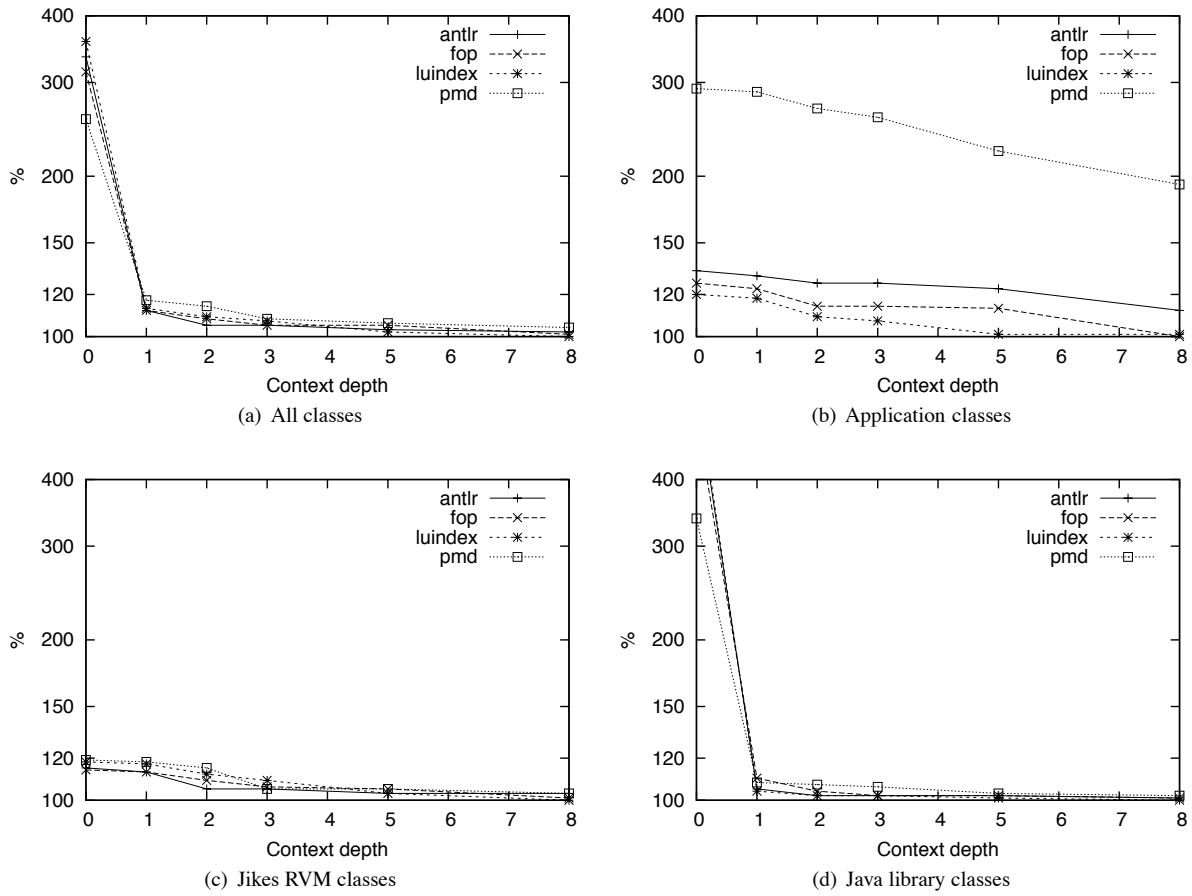
The stability of clustering across inputs suggests that a feasible implementation strategy for collection might be to acquire clustering data from a single training run. These clusters would indicate which sites had similar *ldf*s, and therefore should be allocated under the same policy.

## 6. Context

Allocation site offers several advantages as a predictor. It encompasses both the type of the object allocated and the static type of the object receiving this reference, reference density, size of scalar objects (and of arrays if they are always allocated at the same size at this site) and one level of context in the call graph. Earlier studies have found that the site provides sufficiently accurate predictions of the behaviour of Java [10] and ML [13] programs, although C programs [4, 51] require deeper calling context (since programmers commonly wrap calls to `malloc()` in order to catch failure). However, this style of programming also appears in object-oriented programs as allocation by factory methods, wrappers and so on. In this section, we investigate the influence of calling context on object lifetime distribution.

We observed in Section 3 that it is common for the lifetime distribution of objects allocated by a single site to exhibit a small number of distinct patterns: Figure 4 provides a typical example. Might the use of site as a predictor conflate distinct behaviours that deeper calling context would separate?

By recording in object headers the point of the allocation in the calling context tree (as well as site ID), we can identify the precise call chain to each object allocation. To obtain calling context, we incorporated Bond and McKinley's patch to Jikes RVM, version 2.4.6, into our MemTrace system. However, we use *precise* rather than probabilistic [11] calling contexts. Obtaining calling context tree information required the use of a modification of




(a) All classes

(b) Application classes

(c) Jikes RVM classes

(d) Java library classes

**Figure 9.** Lifetime distribution variance Dacapo benchmarks, default input. The $x$-axis is calling context depth and the $y$-axis is variance as a percentage of the variance for $k = \infty$ (log scale).

the Jikes RVM BaseAdaptive compiler; however, this configuration performs no code optimisation. Thus, classes loaded at both build-time and run-time continue to be compiled as if by the non-optimising baseline compiler.

By tracking calling context, each allocation can be mapped to a sequence $\langle s, m_0, m_1, \ldots \rangle$ where $s$ is the allocating bytecode in a method $m_0$, which was called by $m_1$, and so on. For simplicity, we shall continue to use the term 'site' to mean the sequence $\langle s, m_0, m_1, \ldots \rangle$. Consider two allocations, with sequences $\langle s, m_0, m_1, \ldots \rangle$ and $\langle t, n_0, n_1, \ldots \rangle$. For a calling context of depth $k$, we define both allocations to be made by the same 'site' if and only if $s = t$ and $m_i = n_i$ for all $i = 0 \ldots k$. We now explore $\langle s, m_0, m_1, \ldots, m_k \rangle$ as the lifetime predictor, rather than simply the method and byte-code offset (i.e. $\langle s, m_0 \rangle$), for different values of $k$. We denote the complete call chain by $\langle s, m_0, \ldots, m_\infty \rangle$.

Our goal is to identify good predictors for object lifetime behaviour that can be exploited to guide GC. An *ideal* 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, a *good* predictor would provide a narrow estimate of the lifetime range it expects of the object. That is, it would predict that a fraction $p$ of objects allocated by a site have lifetimes of between $l_1$ and $l_2$ bytes: the better the predictor, the larger $p$ and the smaller the interval $[l_2, l_1]$. The variance of the object lifetime distribution of a site is a measure of the range of the lifetimes of the objects it allocates.

The conclusion of earlier studies was that calling context $k = 0$ suffices for Java programs but C programs require larger value for $k$ to avoid conflating different behaviour patterns. The depth $k$ necessary can be determined by considering the variances of the site lifetime distributions generated for increasing values of $k$. We define the *weighted variance* of a set of a partitoned distribution to be the weighted sum of the variances of its partitions.

THEOREM 1 (Weighted Variance). *Consider a distribution $X$, of size $|X|$ and variance $var_X$. Suppose $\{X_0|X_1|\ldots X_n\}$ is a partitioning of the distribution. Then the weighted variance of the partition is no larger than the variance of the whole distribution.*

$$var_X \geq \sum_{i=0}^{n} \frac{|X_i|}{|X|} var_{X_i}$$

The effect of increasing context depth $k$, is to split some 'sites' and their object lifetime distributions into two or more. Categorising objects by 'site' is a partitioning of the set of objects allocated, and increasing $k$ leads to a finer partitioning. As $k$ increases, the number of sites also increases monotonically but the weighted variance of the site lifetime distributions decreases monotonically. The pertinent question is, after what depth $k$ does this variance tend to reach a limit?

Figure 9(a) shows the weighted standard deviation of allocation sites for increasing depths of calling context $k$, shown as a percentage of the weighted standard deviation for $k = \infty$ (i.e. using the complete call chain). For all of the DaCapo benchmarks analysed here[6], there is a significant improvement in variance when the calling context depth $k$ is greater than zero, and further small improvements are also seen at $k = 2$. However, increasing calling context depth does not appear to offer any significant further improvements.

The evidence that a calling context of $k = 1$ offers significant improvements in variance appears to contradict the conclusions of earlier studies. To explore this further we study separately sites in application classes, JikesRVM classes and Java library classes.

Figure 9(b) shows that for sites in application classes there is little reduction in variance as $k$ grows, which agrees with earlier studies. Figure 9(c) shows that sites in Jikes RVM classes, much like those in application classes, do not benefit from a calling context of $k \leq 8$. Furthermore it appears that even with $k = \infty$, Jikes RVM sites show only a 20% reduction in variance. However, Figure 9(d) shows that sites in Java library classes show very significant reductions in variance of between 2.3× for pmd and 5× for luindex are achieved with only one level of context ($k = 1$). Increasing beyond $k = 1$ appears to offer little further reduction. One explanation for this improvement is that many Java library classes perform allocation in response to application requests. With a calling context of $k = 0$ only the library behaviour can be seen, but the library may be controlled in many different ways by the application classes, and by using a calling context of $k \geq 1$ the application behaviour can be seen in addition to the library behaviour. The allocation site in Figure 2, from the internal GNU classpath `java.lang.String` constructor `String([char], int, int, boolean)`, is a good example. The method is invoked both from all `String` constructors, but also from `String`'s string concatenation methods. Treating all paths as one conflates distinct behaviours.

## 7. Conclusions

No one size of collector fits all applications. We believe that a key to improved memory management performance is to match the collector to the mutator. Through a study of a wide range of realistic Java benchmarks, we find an association between allocation site and both object lifetime distribution and phase behaviour. From the perspective of allocation sites, there is a richer demographic landscape than the usual generational classification of 'short-lived', or 'long-lived' or 'immortal'. We further refine our classification by considering separately the behaviour of objects allocated by the JIT compiler, the run-time system, Java library code as well as those allocated directly by the application program.

Most importantly for practical GC implementation, we demonstrate through a statistically rigorous analysis that the sites of programs exhibit only a few distinct object-lifetime distributions, that is, they cluster strongly. A very small number of clusters dominate allocation, whether measured by volume of allocation or space rental (the product of an object's size and age). Essential to exploitation, these clusterings are stable across different program inputs, thus making the prospect of specialised allocators realistic. We describe the design of such a collector in [26].

Finally, we investigate the extent to which calling context effects object lifetime. In common with earlier studies for Java, we find that simply using allocation site is sufficient for application and Jikes RVM classes. However, in contrast with previous studies, we find that allocation site and one further level of context significantly reduces variance in the lifetime distributions of objects allocated by Java library classes.

[6] Under Jikes RVM 2.4.6, for default input sizes, only antlr, bloat, fop, luindex and pmd ran successfully. MemTrace stresses the GC system and other DaCapo programs fell victim to the well-known GC-map bug. We have not yet had time to port our results to version 2.9 which we understand has resolved this bug.

***Acknowledgements*** We thank the anonymous reviewers for their thoughtful comments and suggestions. We 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. We thank IBM Research for making the Jikes RVM system available, and Michael Bond for providing his calling context tree mechanism.

## References

[1] B. Alpern, C.R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J.J. Barton, S.F. Hummel, J.C. Sheperd, and M. Mergen. Implementing Jalapeño in Java. In OOPSLA [31], 314–324.

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

[3] D.A. Barrett and B. Zorn. Garbage collection using a dynamic threatening boundary. Computer Science Technical Report CU-CS-659-93, University of Colorado, July 1993.

[4] 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*, 187–196, Albuquerque, NM, June 1993. ACM Press.

[5] S. Blackburn, R. Garner, K.S. McKinley, A. Diwan, S.Z. Guyer, A. Hosking, J.E.B. Moss, and D. Stefanovic. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06 ACM Conference on Object-Oriented Systems, Languages and Applications*, Portland, OR, October 2006. ACM Press.

[6] S.M. Blackburn, P. Cheng, and K.S. McKinley. Myths and reality: The performance impact of garbage collection. In *Sigmetrics - Performance 2004, Joint International Conference on Measurement and Modeling of Computer Systems*, June 2004.

[7] S.M. Blackburn, P. Cheng, and K.S. McKinley. Oil and water? high performance garbage collection in Java with MMTk. In *ICSE 2004, 26th International Conference on Software Engineering*, Edinburgh, May 2004.

[8] S.M. Blackburn, M. Hertz, K.S. Mckinley, J.E.B. Moss, and T. Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, 29(1):1–57, 2007.

[9] S.M. Blackburn, R.E. Jones, K.S. McKinley, and J.E.B. Moss. Beltway: Getting around garbage collection gridlock. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, 153–164, Berlin, June 2002. ACM Press.

[10] S.M. Blackburn, S. Singhai, M. Hertz, K.S. McKinley, and J.E.B. Moss. Pretenuring for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, 342–352, Tampa, FL, October 2001. ACM Press.

[11] M.D. Bond and K.S. McKinley. Probabilistic calling context. In *Object-Oriented Programming, Systems, Language and Applications (OOPSLA'07)*, Montreal, Canada, 2007. ACM Press.

[12] Chakravarti, Laha, and Roy. *Handbook of Methods of Applied Statistics*, volume I. John Wiley and Sons, 1967.

[13] 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*, Montreal, June 1998. ACM Press.

[14] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software Practice and Experience*, 24(6), 1994.

[15] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In Eric Jul, editor, *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98, Lecture Notes in Computer Science* 1445, 92–115, Brussels, July 1998. Springer-Verlag.

[16] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In [22].

[17] J. Gil and I. Maman. Micro patterns in Java code. In *Object-Oriented Programming, Systems, Language and Applications (OOPSLA'05)*, 97–116, San Diego, CA, 2005. ACM Press.

[18] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1997.

[19] L.T. Hansen and W.D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the 2002 ACM SIGPLAN International Conference on Functional Programming (ICFP02)*, 247–258, Pittsburgh, PA, 2002. ACM Press.

[20] T. Harris. Dynamic adaptive pre-tenuring. In [22].

[21] B. Hayes. Using key object opportunism to collect old objects. In *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, 33–46, Phoenix, Arizona, October 1991. ACM Press.

[22] T. Hosking, editor. *ISMM 2000 Proceedings of the Second International Symposium on Memory Management*, Minneapolis, MN, October 2000. ACM Press.

[23] L. Hubert and P. Arabie. Comparing clusterings. *Journal of Classification*, 2:193–218, 1985.

[24] A.K. Jain, M.N. Murty, , and P.J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(31):264–323, 1999.

[25] M.S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, December 1997.

[26] R.E. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2006)*, Nantes, France, July 2006.

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

[28] S. Marion, R.E. Jones and C. Ryder. Decrypting the Java gene pool: Predicting objects' lifetimes with micro-patterns. In [35], 67–78.

[29] 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.

[30] M. Meilă. Comparing clusterings — an information based distance. *Journal of Multivariate Analysis*, 98(5), May 2007.

[31] *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, Denver, CO, October 1999. ACM Press.

[32] T. Printezis. Hot-Swapping between a Mark&Sweep and a Mark&Compact Garbage Collector in a Generational Environment. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.

[33] T. Printezis and R.E. Jones. GCspy: An adaptable heap visualisation framework. In *OOPSLA'02 ACM Conference on Object-Oriented Systems, Languages and Applications*, 343–358, Seattle, WA, November 2002. ACM Press.

[34] W.M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66:846–850, 1971.

[35] M. Sagiv, editor. *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, Montréal, Canada, October 2007. ACM Press.

[36] P.M. Sansom. Combining copying and compacting garbage collection. In *Fourth Annual Glasgow Workshop on Functional Programming*. Springer-Verlag, 1992.

[37] P.M. Sansom and S.L. Peyton Jones. Generational garbage collection for Haskell. In *Record of the 1993 Conference on Functional Programming and Computer Architecture*, Glasgow, June 1993. ACM Press.

[38] R. Shaham, E. Kolodner, and M. Sagiv. Estimating the impact of liveness information on space consumption in Java. In *ISMM'02 Proceedings of the Third International Symposium on Memory Management*, 64–75, Berlin, June 2002. ACM Press.

[39] J. Singer, G. Brown, and I. Watson. Intelligent selection of application-specific garbage collectors. In [35], 91–102.

[40] S. Soman, C. Krintz, and D. Bacon. Dynamic selection of application-specific garbage collectors. Technical Report 2004–09, UCSB, January 2004.

[41] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, March 1999.

[42] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, University of Massachusetts, 1999.

[43] D. Stefanović, K.S. McKinley, and J.E.B. Moss. Age-based garbage collection. In [31], 370–381.

[44] S.P. Thomas. Garbage collection in shared-environment closure reducers: Space-efficient depth first copying using a tailored approach. *Information Processing Letters*, 56(1):1–7, October 1995.

[45] D.M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, April 1984.

[46] D.M. Ungar and F. Jackson. Outwitting GC devils: A hybrid incremental garbage collector. In *OOPSLA/ECOOP '91 Workshop on Garbage Collection in Object-Oriented Systems, Addendum to OOPSLA'91 Proceedings*, October 1991.

[47] 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.

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

[49] P.R. Wilson, M.S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of International Workshop on Memory Management*, *Lecture Notes in Computer Science* 986, Kinross, Scotland, September 1995. Springer-Verlag.

[50] T. Yuasa and M. Hagiya. Kyoto Common Lisp report. Technical report, Teikoku Insatsu Publishing, Kyoto, 1985.

[51] B.G. 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, October 1998.

[52] B.G. Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, March 1989. Technical Report UCB/CSD 89/544.