Towards Intelligent Analysis Techniques for Object Pretenuring

Jeremy Singer, Gavin Brown, Mikel Luján, Ian Watson
University of Manchester, UK

ABSTRACT

Object pretenuring involves the identification of long-lived objects at or before their instantiation. It is a key optimization for generational garbage collection systems, which are standard in most high performance Java virtual machines. This paper presents a new study of factors that are used to indicate object lifespans. We adopt the information theory measurement of normalized mutual information to compare these various different factors in a common framework. A study of garbage collection traces from four standard Java benchmark programs shows that there is high dependence on some of these factors such as allocation site and object type. We also identify and measure new factors based on object-oriented metrics.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

General Terms

Measurement

Keywords

Object lifetime analysis, pretenuring, information theory

1. INTRODUCTION

Garbage collection (GC) is the process of automatic management of dynamically allocated memory [13]. Once this discipline was the province of academic and esoteric languages, but it has now become a mainstream computer systems issue. The growing significance of GC is largely due to the rise of managed programming languages, such as Java and C#, with their associated virtual machine (VM) systems.

In these managed environments, GC takes place during application execution, thus it is accountable for part of the overall execution time. The percentage of execution time spent in GC is generally around 5%, though it can be much higher for relatively small heap sizes. Time spent in GC is wasted as far as the application user is concerned. There is a need to minimise the overall GC time. In addition, for interactive applications, it is desirable to have frequent short GC pauses rather than occasional longer pauses. These requirements lead to the concept of *generational* GC.

The basic premise, known as the *weak generational hypothesis* [18], is that most objects die young. A simple generational collection scheme may operate as follows. The heap is split into two regions: the nursery and the mature space. Objects are initially allocated in the nursery and this region is garbage collected frequently. Since most objects die young, then they are collected in the nursery and their space is recycled. If objects survive for a long time (i.e. they are not dead when nursery GC takes place) then these long-lived objects are promoted to the mature space. This is known as object *tenuring*. The mature space is collected less frequently than the nursery, since it fills up more slowly and we expect that objects in the mature space will remain alive for a long time.

An optimization for generational GC is *pretenuring* [7, 6]. The task of a pretenuring scheme is to identify long-lived objects at or before their dynamic instantiation point and allocate such objects immediately into the mature space since they will probably survive the nursery. This saves the computational cost of processing the object in the nursery, and of copying it to the mature space. The scanning and copying of objects are generally the two most expensive operations that occupy significant GC time.

The crucial issue with pretenuring optimizations is that the analysis must be accurate. The aim is to minimise the number of pretenured objects that are not long-lived. They would only waste the mature space, which is collected infrequently. Furthermore they could artificially prolong the lives of objects to which they point. It is also necessary to consider the high cost of inter-generation references, implemented via write-barriers.

This paper classifies a range of existing indicators used to predict object lifetimes for pretenuring schemes. We describe how *information theory* can be used to evaluate relative performance of various features as indicators for object tenuring. We report on some preliminary studies that show strong promise for the application of machine learning techniques to this area.

The contributions of this paper are:

1. a formal framework based on information theory to

assess good indicators for object tenuring.

2. a study comparing the relevance of common type- and site-based characteristics for object tenuring.

3. an evaluation of a metrics-based approach as an indicator of object tenuring.

2. BACKGROUND

2.1 Pretenuring

We divide the pretenuring process into two distinct phases: an *analysis* phase followed by a *transformation* phase. The analysis phase has the task of predicting object lifetimes, based on input from a corpus of already-processed object lifetimes, and from the characteristics of the current object that is about to be allocated. The transformation phase acts on the analysis by optimizing object allocations to take advantage of short- and long-lived objects. In general, this means that objects are allocated in different sections of the heap based on their predicted lifetime. In this paper, we focus on the analysis problem.

Pretenuring analyses and object lifetime studies can be characterized according to several simple parameters. Studies either use *exhaustive tracing* or *statistical sampling* of dynamic object allocations. They have various levels of precision for predictions. Some schemes are *binary* (short/long). Others are *ternary* (short/long/immortal). Some are even *continuous* and give actual numerical estimates for object lifetimes. The evaluations presented in this paper are based on exhaustive tracing using binary predictions, where each prediction indicates whether an object should be tenured or not.

Different analyses group dynamic object allocations into *clusters* according to some criterion. The most common two criteria are *static allocation site* and *object type*. For instance, all objects allocated at program counter value x, or all objects allocated with type `Foo`, may be clustered together. Usually, pretenuring predictions are applied on a per-cluster (rather than per-object) basis. Such a generalization is made for efficiency reasons. This paper aims to study the most effective approaches to clustering dynamic object allocations, re-examining existing schemes as well as proposing new ones.

2.2 Information Theory

Information theory provides a rich and sound mathematical framework for analysis of data sources. Originally developed in the context of secure communications and cryptography, it has been applied in fields as diverse as machine learning, medical image processing, and financial market prediction. We explore how these ideas might be used to assess and improve object lifetime prediction.

The fundamental measure in this framework is *entropy*, which explicitly quantifies the information content in a given source of data: the more 'randomness' or unpredictability in the data source, the higher the entropy value. As an example consider a device producing symbols according to a random variable X, defined over a finite alphabet of possible symbols S_X. If we assume each successive symbol $s_i \in S_X$ is independent of the previous ones, the *unconditional entropy* is defined as,

$$H(X) = -\sum_{i=1}^{|S_X|} p(i)\log(p(i)) \tag{1}$$

where $p(i)$ is the probability of the ith symbol being produced. Note that all logarithms are taken to base 2. In practical terms, $p(i)$ can be calculated with frequency counts, i.e.:

$$p(i) = \frac{\text{number of occurrences of symbol } s_i}{\text{total number of symbols seen}} \tag{2}$$

In this work, we consider the produced symbols to be a common measurement on a sequence of dynamic object allocations. For instance we could measure the size of each newly created object. This gives us a stream of size values, for which we can calculate an entropy measure.

The *conditional entropy* measures the dependence between two different symbol streams. In our case, we could take two measurements on each element of a sequence of dynamic object allocations. For instance we could measure the size of each newly created object and its lifetime. These are two different random variables X and Y respectively with two different alphabets S_X and S_Y but there may be some dependence between them, which we can quantify by conditional entropy.

$$H(Y|X) = -\sum_{i=1}^{|S_X|} p(i) \sum_{j=1}^{|S_Y|} p(j|i)\log(p(j|i)) \tag{3}$$

This is the *first order conditional entropy*. The required probabilities can again be computed from frequency counts:

$$p(j|i) = \frac{\text{number of times } s_j \text{ occurs with } s_i}{\text{number of occurrences of } s_i} \tag{4}$$

First order conditional entropy has a minimum value of zero and a maximum value of $\log(|S_Y|)$. In the example above, it measures the uncertainty we have in the lifetime of an object given its size. If lifetime values are produced uniformly at random over the alphabet S_Y, then eq.(3) will converge in the limit to $\log(|S_Y|)$.

The *mutual information* between X and Y is a measure of the agreement, or correlation, between them. The mutual information is,

$$I(X;Y) = H(Y) - H(Y|X) \tag{5}$$

This is easily computed from the entropy measurements we have already described above. This measurement is symmetric, i.e. $I(X;Y) = I(Y;X)$, and quantifies the reduction in our uncertainty of Y, when the value of X is revealed. Unlike Pearson's correlation coefficient, which only detects *linear* correlations between random variables, mutual information can detect arbitrary *nonlinear* relationships.

$I(X;Y)$ can be normalized to a value between 0 and 1 by dividing it by $\min(H(X), H(Y))$. All mutual information values in this paper are normalized in this way. A high value of mutual information indicates that there is information in the symbol sequence to be exploited. A low value of mutual information indicates that there is little information, and therefore little opportunity for accurate prediction.

3. RELATED WORK

This section surveys existing work regarding pretenuring analysis and object lifetime studies. For each paper, we explicitly state their clustering criteria for dynamic allocations. Most existing schemes use site-based [6, 5, 14, 9, 7], type-based, [16, 11]. or context-based clustering [12, 3, 15]. We evaluate these clustering criteria in a common framework in Section 4.

Blackburn *et al.* [5, 6] present a scheme for static pretenuring of Java programs implemented in Jikes RVM. This scheme analyses offline object lifetime information produced by exhaustive tracing of object lifetimes. The clustering of objects is based on allocation sites classifying these into immortal, long-lived or short-lived. Loosely speaking, given an allocation site, this scheme would classify it, for example, as immortal when most objects allocated by the allocation site are classified as immortal. The actual definition of what 'most objects' means is done in terms of *homogeneity factors*. The scheme requires exploring the adequate values for these homogeneity factors by trial-and-error. Once an allocation site is given a classification, this is fixed and this prediction is specialized for those programs for which the object lifetime information has been analysed. Blackburn *et al.* [5] measure entropy for allocation sites to illustrate their homogeneity.

Jump *et al.* [14] also present a pretenuring scheme for Java programs implemented in Jikes RVM, but it is dynamic pretenuring. This scheme performs online statistical sampling of object lifetime per allocation site. Allocation sites are either classified as to be pretenured or not depending on whether the survival rate for the allocations is above a threshold. Again the scheme requires exploring the adequate values for the threshold which is done offline.

Harris [9] implements the first dynamic pretenuring scheme for Java programs in Sun's Research VM. Harris reports statistics on information that could be used at runtime to classify an allocation site as to be pretenured: object class, allocation site and dynamic call context up to a depth of five. Although object class and dynamic call context provide reasonable predictors of an object lifetime, the dynamic call context combined with allocation site reduces the misclassification rate. Nonetheless, Harris uses only allocation site to evaluate the runtime pretenuring scheme. Jump *et al.* [14] and Harris [9] differ on the sampling techniques to obtain object lifetime information and also on the backsampling strategies.

Cheng *et al.* [7] investigate a pretenuring scheme for ML programs using offline analysis and clustering on allocation sites. They find that a few allocation sites consistently generate long-lived objects while most allocations sites generated short-lived objects. Accordingly allocation sites with long-lived objects above a threshold of 80% (found empirically) are modified so that their objects are pretenured.

Huang *et al.* [11] also describe a dynamic pretenuring scheme for Java programs implemented in Jikes RVM, but they cluster objects according to object class (type) ignoring allocation site. Jump *et al.* [14] criticize this clustering choice by stating 'Type is not a good predictor of lifetime', although without reporting experimental data to illustrate it.

Shuf *et al.* [16] introduced the term *prolific types* to characterize those relatively few object types that account for large volumes of objects allocated in their set of Java benchmarks. Based on static analysis of the object lifetime traces, they propose the prolific hypothesis: prolific types create objects that die younger than objects of non-prolific type.

Inoue *et al.* [12] investigate the construction of object lifetime predictors taking into account the dynamic call context (up to depth 20) and object type. The predictor is created by associating those dynamic call contexts and object types for which every object created (and captured in the offline analysis) has the same lifetime.

Barrett and Zorn [3], and Seidl and Zorn [15] describe object lifetime predictors for C programs. Barrett and Zorn [3] use dynamic call contexts and object size to determine whether objects are short-lived or long-lived. On the other hand, Seidl and Zorn [15] only consider the dynamic call context but try to classify objects into one of four categories: highly referenced, not highly referenced, short-lived and others.

To sum up, none of the cited papers have used information theory to identify good predictors for tenuring behaviour. All these papers have disregarded some of the features and focus on others. They selected features on the basis of observed improvements in execution times for their particular GC implementations and benchmarks, and disregarded other features that did not lead to improvements. Thus, there is a lack of a common theoretical framework for the comparison of features as predictors for object tenuring behaviour.

4. EMPIRICAL STUDIES OF TENURING

This section evaluates existing pretenuring analysis design choices as well as new possibilities. The experiments operate by measuring correlation of various features with tenuring behaviour, using the Normalized Mutual Information (NMI) metric introduced in Section 2.2.

4.1 Experimental Framework

We obtain the object allocation data and tenuring information using Jikes RVM [1, 2] and MMTk [4]. Jikes RVM is a production-quality open-source optimizing VM, used extensively in research-based studies of Java programs. MMTk is an extensible research-oriented memory management toolkit. (Note that the majority of other pretenuring studies also use Jikes RVM and MMTk for their empirical evaluations.) All our experiments are conducted with Jikes RVM v2.4.6. We have instrumented the VM to record statistics about allocations in the nursery, and promotions from the nursery to the mature space. This information is dumped to trace files, which are postprocessed to extract the relevant information.

Statistics are gathered for a single run of each benchmark program, using the default production build of Jikes RVM for IA32 Linux, running in adaptive compilation mode. This incorporates a generational mark-and-sweep GC scheme. We fix[1] the initial and maximum heap size to be 50 MB (16 MB for the smaller health benchmark). We also fix the nursery size to be 12 MB (4 MB for health). The remainder of the heap is used for mature space, immortal space and large object space. All the trace information relates solely to objects and arrays allocated in the nursery space, which

[1]We also conducted our experiments using variable sized heaps and nurseries, which are the default for Jikes RVM. We found very little difference in our correlation results.

name	suite	description	input
_202_jess	SPECjvm98	expert system shell	s100
_213_javac	SPECjvm98	Java compiler	s100
_228_jack	SPECjvm98	parser generator	s100
health	JOlden	process simulator	6/128

Figure 1: Description of the benchmark programs used in this study

are candidates for promotion to the mature space.

We have chosen four benchmarks from the pretenuring study by Blackburn *et al.* [6]. They are studied in most Java-based pretenuring papers. The benchmarks are described in Figure 1. Blackburn *et al.* [6] identify these particular programs as benefitting from pretenuring optimizations to a greater extent than other benchmarks from the same suites. Note that all our experiments include runtime activity from Jikes RVM, since this is a Java-in-Java VM. This means that VM objects are also candidates for pretenuring optimizations. Blackburn *et al.* show that this can lead to greater optimization opportunities.

The trace files contain the following features for each object allocation: object type; static allocation site; dynamic calling context; object size; tenuring information. These features are cheap to generate and small enough to be stored efficiently. Other more complex features could be used, but they may be too expensive to obtain and store given the real-time constraints of typical GC schemes, so they would not be used in practice.

4.2 Benchmark Characteristics

Figure 2 presents the number of bytes allocated in the nursery space for each benchmark program. These include object and array allocations, for both VM and application code. The graph also shows the number of bytes promoted from the nursery to the mature space, and again how this figure is split between objects and arrays. Note that in general, a higher proportion of tenured space is due to objects rather than arrays. This is because only small, relatively short-lived arrays are allocated in the nursery. Larger arrays are allocated directly in the Jikes RVM large object space, which is not part of this empirical study. We take advantage of this tenuring preference for objects in Section 4.4, which focuses on object-specific metrics rather than array-based metrics for pretenuring.

4.3 Evaluation of Standard Clustering Techniques

This section evaluates how well the standard clustering techniques correlate with object tenuring behaviour. We include several variations of clustering that we have identified in the literature. We use the information theory measurement of NMI to report correlation. Recall that an NMI score of 0 indicates no correlation, and 1 indicates perfect correlation.

The left-most column for each benchmark in Figure 3 shows how object type (including arrays) correlates with tenuring, for all types.

We have already discussed the notion of prolific types [16]. The hypothesis suggests that these are generally short-lived and should not be tenured. We measure NMI scores for type versus tenuring for the ten types responsible for the

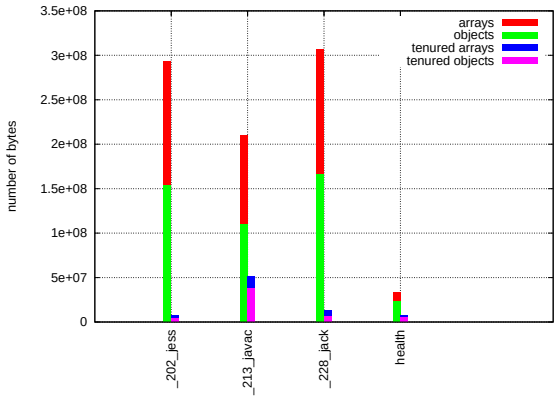

Figure 2: Number of bytes allocated in nursery and promoted from nursery for various benchmarks

most allocations for each benchmark. We manually select these as prolific types. The second column for each benchmark in Figure 3 shows these results. We note that the NMI scores are generally reduced (compared with the scores when including non-prolific types) due to the elimination of singleton clusters, which tend to inflate NMI scores. The NMI score is reduced when different dynamic allocations within a single cluster have different tenuring behaviour. Within a singleton cluster, only one tenuring behaviour is possible so these cannot serve to degrade the NMI score. Note that _213_javac has a high NMI value for prolific types. The original paper [16] has its best results on this benchmark, so this would confirm their findings.

The third column for each benchmark in Figure 3 shows how static allocation site correlates with tenuring information. Note that site is always a more accurate indicator of tenuring policy than type. This is a quantitative confirmation of the assertion of Jump *et al.* [14] that type is not as good as site. In fact site subsumes type, since only objects of a single type can ever be allocated at any one static allocation site.

To parallel the prolific types idea above, we investigate whether *prolific sites* are also good indicators of tenuring policy. For each benchmark, we identify the 25 static allocation sites that are responsible for the most allocations and we measure the correlation for these prolific sites alone. The fourth column for each benchmark in Figure 3 shows the results. Again there is a slight reduction in NMI scores due to the elimination of singleton clusters. However site seems to be a better indicator of tenuring behaviour than type, in the prolific case as well as the general case.

Figure 4 shows how dynamic calling context correlates with tenuring information. The graph shows that correlation increases with the context depth. However, the improvements level off after around four methods in the context. In comparison with static allocation site, context alone is not a good indicator. The situation is probably worse for Java programs than for ML (in which the scheme was originally evaluated) since Java methods may have more allocations than ML functions, so it is less likely that all allocations in a single context will have the same tenuring behaviour.

Figure 5 shows how static allocation site concatenated

with dynamic calling context correlates with tenuring behaviour. This gives the best correlation of all the standard schemes we have measured until now.

4.4 Clustering Based on Object Features

As outlined above, objects appear to be more important than arrays since they are more likely to be tenured. In this section we distinguish between object and array allocations. We want to address the problem of how to generalize over objects. This will allow pretenuring analyses to make predictions for *previously unseen* objects, given that they can relate such objects to existing objects in their training corpus. For instance, if we know that objects of type X are always tenured, then if type Y is similar to type X, will objects of type Y also be consistently tenured? The key issue here is determining how to measure similarity between objects and types. It is not enough to use any general object similarity metric. We need to measure whether a given metric correlates with object lifetime.

First we investigate how object *size* correlates with tenuring behaviour. We distinguish between class-based objects and arrays. Figure 6 shows that neither object size nor array size are good indicators of pretenuring behaviour for our four benchmarks, with the possible exception of object size for health. We need to use alternative metrics of similarity between classes if we are to predict tenuring behaviour accurately.

To extrapolate object tenuring behaviour based on type, we use type-based metrics obtained from the Chidamber and Kemerer suite [8]. This is a set of metrics specifically designed to measure the object-oriented nature of source code. We measure the metrics using the ckjm tool [17] for the benchmarks and the VM, creating a set of scores for each class file. We use these metrics to cluster together 'similar' object types, and measure their correlation with tenuring behaviour. The metrics are described in Figure 7.

We measure the NMI for each of these type-based metrics against the tenuring behaviour of objects. A NMI score close to 1 should indicate that the metric is a good predictor for pretenuring.

Figure 8 shows the results for this study. Some metrics (such as weighted methods per class) always seem to correlate consistently with tenuring information whereas other metrics (coupling between object classes) are less consistent. Note that Hirzel *et al.* [10] claim that object connectivity is a good indicator of object lifetime. Connectivity is indirectly related to one or more CK metrics.

5. CONCLUSIONS

This paper has shown how information theory can be used to evaluate indicators for pretenuring analyses. We have compared different existing dynamic allocation clustering schemes, under a sound and unified framework. The allocation site combined with context contains the most information among the available schemes on the set of benchmarks. We have introduced and measured a novel concept of identifying objects with similar tenuring behaviour based on the use of object-oriented metrics to determine similarity. We aim to build on this work by constructing predictors that use such features to learn effective pretenuring policies.

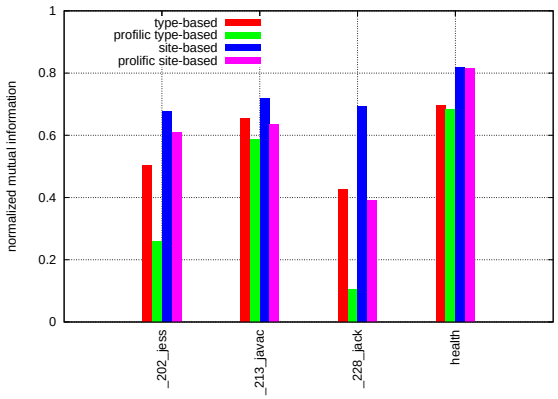

Figure 3: Correlation of various allocation clustering schemes with tenuring behaviour

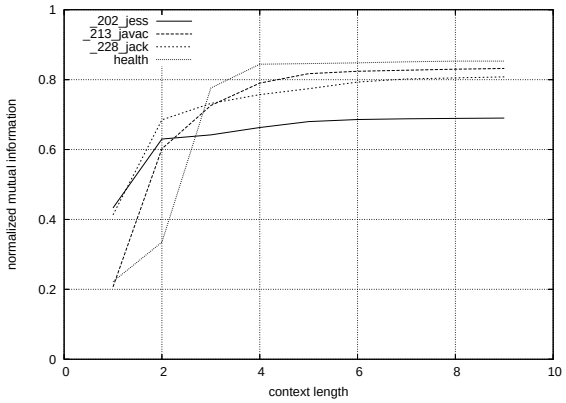

Figure 4: Correlation of context with tenuring behaviour

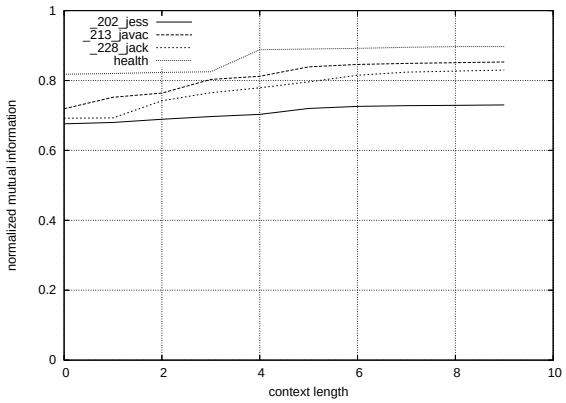

Figure 5: Correlation of site and context with tenuring behaviour

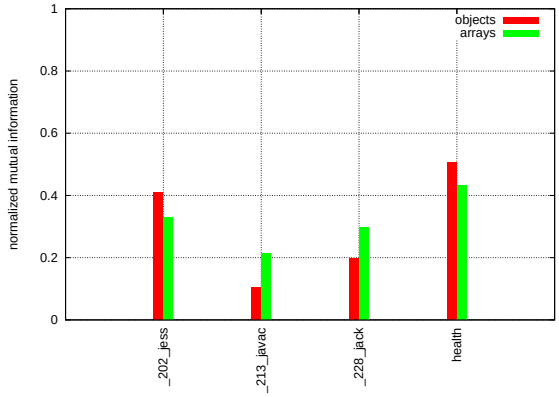

Figure 6: Correlation of object and array size with tenuring behaviour

name	description
WMC	weighted methods per class
DIT	depth of inheritance tree
NOC	number of children
CBO	coupling between object classes
RFC	response for a class
LCOM	lack of cohesion in methods

Figure 7: Metrics used to group types

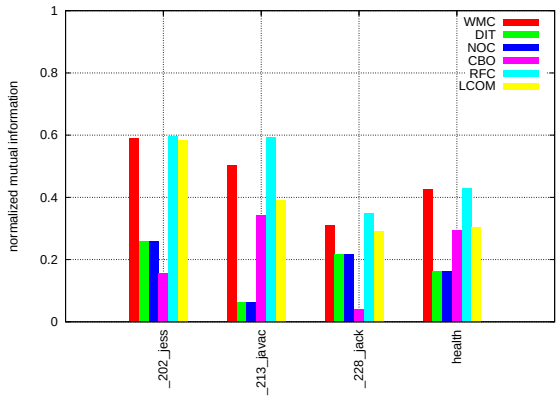

Figure 8: Correlation of various CK metrics with pretenuring behaviour

6. REFERENCES

[1] B. Alpern et al. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, Feb 2000.

[2] B. Alpern et al. The Jikes research virtual machine project: Building an open source research community. *IBM Systems Journal*, 44(2):1–19, Feb 2005.

[3] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI*, pages 187–196, 1993.

[4] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE*, pages 137–146, 2004.

[5] 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.

[6] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss. Pretenuring for Java. In *OOPSLA*, pages 342–352, 2001.

[7] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *PLDI*, pages 162–173, 1998.

[8] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[9] T. L. Harris. Dynamic adaptive pre-tenuring. In *ISMM*, pages 127–136, 2000.

[10] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ISMM*, pages 36–49, 2002.

[11] W. Huang, W. Srisa-an, and J. M. Chang. Dynamic pretenuring schemes for generational garbage collection. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 133–140, 2004.

[12] H. Inoue, D. Stefanovic, and S. Forrest. On the prediction of Java object lifetimes. *IEEE Transactions on Computers*, 55(7):880–892, 2006.

[13] R. Jones and R. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, 1996.

[14] M. Jump, S. M. Blackburn, and K. S. McKinley. Dynamic object sampling for pretenuring. In *ISMM*, pages 152–162, 2004.

[15] M. L. Seidl and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *ASPLOS*, pages 12–23, 1998.

[16] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *POPL*, pages 295–306, 2002.

[17] D. Spinellis. ckjm—Chidamber and Kemerer Java metrics, 2005. `http://www.spinellis.gr/sw/ckjm/`.

[18] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.