Intelligent Selection of Application-Specific Garbage Collectors

Jeremy Singer Gavin Brown
Ian Watson
University of Manchester, UK
{jsinger,gbrown,iwatson}@cs.man.ac.uk

John Cavazos
University of Edinburgh, UK
jcavazos@inf.ed.ac.uk

Abstract

Java program execution times vary greatly with different garbage collection algorithms. Until now, it has not been possible to determine the best GC algorithm for a particular program without exhaustively profiling that program for all available GC algorithms. This paper presents a new approach. We use machine learning techniques to build a prediction model that, given a single profile run of a previously unseen Java program, can predict a good GC algorithm for that program. We implement this technique in Jikes RVM and test it on several standard benchmark suites. Our technique achieves 5% speedup in overall execution time (averaged across all test programs for all heap sizes) compared with selecting the default GC algorithm in every trial. We present further experiments to show that an oracle predictor could achieve an average 17% speedup on the same experiments. In addition, we provide evidence to suggest that GC behaviour is sometimes independent of program inputs. These observations lead us to propose that intelligent selection of GC algorithms is suitably straightforward, efficient and effective to merit further exploration regarding its potential inclusion in the general Java software deployment process.

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

General Terms Performance, Experimentation

Keywords Machine learning, application-specific garbage collection

©ACM 2007. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in *Proceedings of the International Symposium on Memory Management*, 2007.

1. Introduction

1.1 Importance of GC

In managed runtime environments such as the Java Virtual Machine (JVM) and the Common Language Runtime (CLR), the total execution time of a user application includes both application execution and VM execution. The most time-consuming tasks performed by the VM are generally optimizing compilation and garbage collection (GC). In this paper, we largely ignore compilation time and focus on improving GC time.

GC has a significant effect on overall execution time for two reasons. There is a *direct* impact, which is apparent in the experiments conducted in this paper. In the most extreme case, one configuration (_213_javac, 32MB heap, SemiSpace collector) spent 89% of its execution time performing GC. The mean proportion of execution time spent in GC is 12.2%, for all 1566 experimental configurations we recorded. There is also an *indirect* impact of GC. This is caused by the manner in which the GC algorithm rearranges heap-allocated data after collection [12]. This can affect subsequent program execution time due to significant changes in the spatial locality of data.

It is well-known that different garbage collectors work best for different programs. The concept of selecting a specialized GC algorithm for each program is termed *application-specific garbage collection*.

Figure 1 shows that different programs perform better with different GC algorithms. These timings were measured using Jikes RVM 2.4.6 with a fixed 512MB heap for a selection of benchmarks from the DaCapo suite, on a lightly loaded Linux IA32 workstation with 1GB RAM. For each benchmark, execution results are reported as speedup percentages relative to the default GenMS GC scheme. Thus scores above 100 represent an improvement on the default scheme, whereas scores below 100 represent a degradation. There are six different GC algorithms tested for each benchmark program. Note that there is a wide variation in the different GC algorithms' performance. In relation to GenMS, each of the other five GC algorithms perform better for at least one program. The best case is the SemiSpace GC on the pmd benchmark, which is over 20% faster than the de-

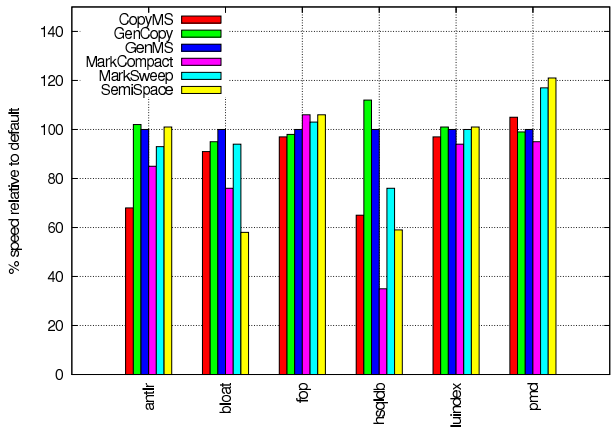

Figure 1. Relative performance of DaCapo benchmarks using various GC algorithms, normalized to the GenMS execution for that benchmark

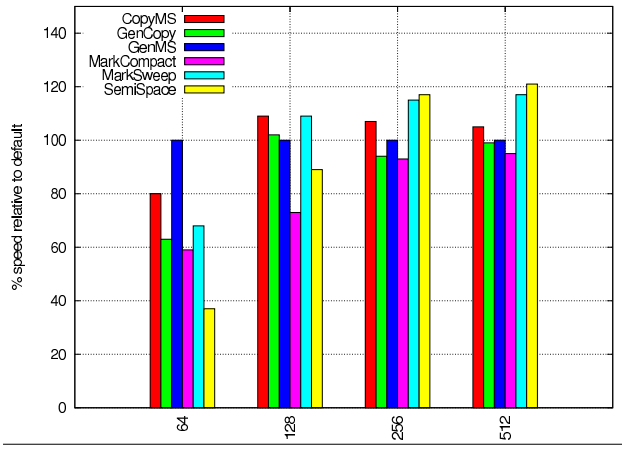

Figure 2. Relative performance of DaCapo pmd benchmark at different fixed heap sizes using various GC algorithms, normalized to the GenMS execution for that heap size

fault case. However it can also be seen that each of the five GC algorithms performs worse than GenMS for at least one program. The worst case is the MarkCompact GC on the hsqldb benchmark, which is almost 65% slower than the default case.

Figure 2 shows that the optimal GC algorithm for a single application can change depending on the size of the JVM heap. All these measurements were taken using the same experimental framework as above, only with various different fixed heap sizes and a single benchmark. Again it is clear to see that there is a wide variation in the relative execution times for the different GC algorithms.

A key point is that it is not at all obvious how to determine which GC algorithm works well with each program and heap size. It would be better to derive this kind of relationship automatically using machine learning techniques.

1.2 Application-Specific GC

To date, attempts to select automatically the optimal GC algorithm have been fairly restricted. They generally require exhaustive profiling of all different GC configurations for the candidate program. This exhaustive search technique is a common element for both static and dynamic selection of GC algorithms. *Static* selection [10] chooses a single GC algorithm for the entire program execution, whereas *dynamic* selection [17] uses several GC algorithms, and switches between them at runtime when certain heuristic thresholds are crossed.

Both these techniques have been implemented on research JVM systems (see Section 6 for full details). They demonstrate nontrivial speedups using application-specific GC as opposed to a default 'one-size-fits-all' GC algorithm. The major drawback of these techniques is that they require large numbers of profiling runs.

Our paper presents a novel approach that uses *machine learning* to select efficiently a good GC algorithm for a program that has not been seen before, by looking at how 'similar' programs perform with the whole range of GC algorithms.

There are several complications that need to be resolved to make this machine learning technique effective. We have seen that the optimal GC algorithm depends on other factors apart from just the program itself. For instance, as the JVM heap size varies, different GC algorithms have different relative changes in performance. In general, the generational collector with a mark-and-sweep mature space (GenMS) algorithm does best with a small heap, whereas other algorithms can outperform GenMS with larger heaps. Another question is whether the GC behaviour of a program could be dependent on that program's input. We consider this in detail in Section 4.3 although earlier work has found input to have negligible effects [17]. All our experimental work is conducted using Jikes RVM. This is a Java-in-Java VM, so VM activity (most notably adaptive compilation activity) also has an impact on GC. In the version of Jikes RVM that we used, the VM shares the same heap as the user application. It is possible to incorporate compilation activity into the GC profile, by profiling the initial execution of each program (one-off run). Alternatively, it is possible to ignore compilation activity by profiling the nth iteration of a program in a test harness (steady-state run) where we assume that all hot methods have been identified and recompiled as necessary in earlier iterations. Unless explicitly stated, the data in this paper concerns steady-state runs of benchmark programs. Another difficulty is that we need to identify suitable program features that can be used to detect similarities between programs for GC performance prediction. This feature selection task is not really intuitive. Section 3.1 describes our feature set in detail.

To summarize, our new approach works like this. We build a model that relates programs with optimal GC algo-

rithms. Then we examine a new program p that is not part of the model, find the most 'similar' program to p in our model based on its features, and select the GC algorithm that our model predicts. Our approach should be quicker and easier than existing profile-guided GC selection techniques. Note that in this paper, *optimum* is always in terms of mimimal overall execution time. We recognise that there are other definitions of optimization (such as minimization of space or power consumption) for which our technique would also be applicable. However we do not consider these throughout this paper.

1.3 Our contributions

This paper makes four main contributions:

- It is the first use of machine learning for application-specific GC selection.
- We achieve a 5% speedup for standard benchmark programs (averaged over all benchmarks and heap sizes) using GC algorithms predicted by our model, rather than using the default generational mark-and-sweep scheme in Jikes RVM.
- The paper quantifies the maximum possible speedup that an oracle GC predictor could achieve. This maximum speedup is 17% when averaged over all benchmarks and heap sizes, in relation to always using the default GC algorithm. No previous limits guidance was available for such a wide range of benchmarks and GC algorithms.
- Given a possible choice between N GC algorithms, our technique requires $O(1)$ profiling runs whereas previous techniques use $O(N)$ profiling runs. Our new approach could easily become part of an automated build/deployment process for managed software.

2. Background

This section considers a simple example scenario, to motivate and explain how machine learning can be applied to the problem of application-specific GC.

Suppose we know that program X executes in the shortest time with GC algorithm foo, and program Y executes in the shortest time with GC algorithm bar. We would like to be able to predict which GC algorithm to use for program Z, without having to profile the executions of Z with every possible GC algorithm.

Our approach is to take some simple measurements on X and Y, which we call the *features* of these programs. In this simplistic example, let the number of explicit invocations of `System.gc()` in the program source code be the only feature. Now we use standard machine learning techniques that relate these features with the best GC algorithms. We determine the best GC algorithms by running X and Y with all GC algorithms, and recording which GC algorithm causes each program to execute in the shortest amount of time. This gives us a table of *training data* that looks like Figure 3.

program	num calls of `System.gc`	best GC algorithm
X	4	foo
Y	1	bar

Figure 3. Training data table for GC prediction

```
if (num calls of System.gc > 2)
then foo else bar
```

Figure 4. Decision tree for GC prediction

We apply a decision tree generator to this training data, which produces a series of conditional statements to determine the best GC algorithm for a program. To continue our trivial example, the decision tree is shown in Figure 4.

Now we measure the same feature for program Z. Say Z calls `System.gc` three times. We feed this data into the decision tree, which therefore recommends that we use GC algorithm foo for Z.

This machine learning approach is beneficial if:

- it is relatively easy and quick to extract features from a program.
- we already have a large corpus of reliable training data that covers a broad range of programs and GC behaviours.
- the recommended algorithm performs better than the default algorithm. (Ideally, we would like the recommended algorithm to be the optimum.)

3. Intelligent Selection

The problem is actually more complicated than the scenario in Section 2. In our real-world system there are six garbage collection algorithms. Basically, a machine learning approach takes a description of a program execution and predicts which GC algorithm performs best for a given JVM heap size, without running all six GC algorithms on that program. In order to obtain program execution descriptions, we take some measurements of benchmarks. These act as *features* which are inputs for the machine learning algorithm.

3.1 Features

Various measurements can be used to characterize the different benchmark programs. We consider static program metrics (Section 3.1.1) which measure properties of the Java bytecode independent of any VM or GC algorithm, dynamic program metrics (3.1.2) which measure runtime behaviour of the program running on a specific JVM with specific GC algorithm, and VM metrics (3.1.3) which measure heap properties of the JVM. Some of the dynamic metrics are GC-invariant, whereas others are highly GC-specific.

3.1.1 Static Metrics

We adopt the Chidamber and Kemerer metrics suite for object-oriented programs [8]. This includes the following six measurements for each class: weighted methods per class (WMC); depth of inheritance tree (DIT); number of children (NOC); coupling between object classes (CBO); response for a class (RFC); and, lack of cohesion of methods (LCOM).

WMC has a unit weight, so it is simply a count of the number of methods in a class. DIT measures the length of the maximum path in the inheritance tree from the root `java.lang.Object` to that class. NOC counts the number of direct subclasses of the current class. CBO measures how often code in a class uses methods or instance fields defined in another class. Multiple accesses to the same class are only counted as a single coupling. The relationship is symmetric. Coupling occurs in both the used class and the user class. RFC counts how many methods are executed in response to a single method call, i.e. how many sub-methods a method executes in its body. This is summed over all methods in the class. LCOM measures the diversity of instance variable usage by different methods of a class. A low LCOM score means that most methods in the class use a similar subset of instance variables in that class. A high LCOM score means that most methods use disjoint subsets of the set of all instance variables in that class.

The aim of the CK metrics suite is to measure the maintainability and object-orientation of program source code. In general, high scores for CK metrics are presumed to indicate complex, poorly maintained source code, that needs to be refactored. Conversely, low scores denote simple, maintainable, object-oriented code.

We used the source code figures from the DaCapo study [6] where possible, and used the same metrics reporting tool (ckjm [18]) for our supplementary benchmark programs. Each metric value is for a single class. To obtain a metric value for an entire program, we sum each metric value over over all the application classes (excluding standard Java libraries) that are loaded during benchmark execution. Our motivation to include these static metrics is that genuinely object-oriented code may be more allocation-intensive. Perhaps it is possible to predict allocation behaviour from the source code using such static metrics.

We note that these metrics are all absolute values. They could be translated into relative values by dividing each summed metric value by the number of classes in that benchmark program, to get an average per-class metric value.

3.1.2 Dynamic Metrics on Reference VM

We use an instrumented build of Jikes RVM to obtain certain object demographics for each benchmark program. These are independent of GC and heap size. They reflect program allocation behaviour in general. These metrics are: number of allocated objects; mean size of allocated objects; number of allocated arrays; mean size of allocated arrays; number of allocated bytes; number of bytes allocated in nursery; proportion of objects that are `java.lang.String` objects; and, bins for different ranges of array sizes (excluding very large arrays that are allocated in the large object space).

Our expectation is that different GC algorithms may be better for differently sized objects or arrays. Note that objects and arrays above a threshold size of 8KB are automatically placed in a separate large object space (LOS) that is not subject to the same GC regime as the standard heap. These LOS allocations are included in the figures for most of these metrics, however nursery-allocated bytes and array bins specifically exclude large objects and arrays. The allocation and collection of the LOS is orthogonal to the general purpose JVM heap, and is independent of the six MMTk GC schemes outlined in Section 3.3. Therefore our GC optimization scheme should not change the behaviour of the LOS.

Other object demographics are GC-dependent. We collect these using a reference GC algorithm for each heap size. In all our tests, the reference algorithm is generational mark-and-sweep (GenMS) which is the default setting in Jikes RVM. Recall that a generational collector has a nursery space, where objects are initially allocated, and a mature space, to which long-lived objects are promoted. GCs can be 'minor' which means that only the nursery space is collected, or 'major' which means that both the nursery and the mature space are collected. Generational collectors generally rely on a low proportion of allocated objects surviving the nursery to be promoted to the mature space. This is known as the 'weak generational hypothesis' which states that most objects die young [19]. Thus nursery survival rate might indicate whether the program is a suitable candidate for generational collection. However note that even programs with high survival rates can perform well with generational collection. The hsqldb program in the DaCapo suite illustrates this point [6]. These GC-dependent dynamic metrics are: nursery survival rate; number of major GCs; and, number of minor GCs.

Finally for dynamic metrics we run a 'reference' VM with various heap sizes and the default GenMS GC algorithm. For each heap size we use the default MMTk timing harness to record: overall execution time; total number of (major+minor) GCs; proportion of time spent in mutator; and, proportion of time spent in GC.

This information gives us an insight into the program's GC behaviour as the heap size changes. We note that these metrics all provide absolute figures. These scores may be turned into relative figures by dividing by number of allocated bytes or by total execution time. However, this relativization of the metrics does not improve prediction accuracy. In fact it has the opposite effect.

3.1.3 VM Metrics

We treat JVM heap size as an input metric. The initial heap size is specified to a JVM by the command line flag `-Xms`,

and the maximum heap size by `-Xmx`. We set both these parameters to the same value to ensure a fixed heap size throughout benchmark execution. As we have already noted, various GC algorithms perform relatively differently when heap size is varied, so this is an important feature. We also note that using runs with different heap sizes is an important technique for obtaining large amounts of training data for our machine learning technique. For instance, if we only have 20 benchmark programs, then this provides very little training data if we fix a standard heap size. With varied heap sizes, we could have 80 rows of training data, with 4 different heap sizes for each benchmark program. In fact, some benchmarks cannot run with small heap sizes, so there are fewer heap size choices for some benchmarks than others. The heap sizes we use in our experiments are 16, 32, 48, 64, 96, 128, 160, 192, 224, 256 and 512MB.

Soman et al [17] use a single metric to determine when to switch GC algorithms. Their metric is the ratio of the current heap size to the minimum possible heap size for each benchmark. For each program, they determine an optimal value for GC algorithm switch. Since they single out this metric as important, we include it as one of our features for the machine learning technique. This means that we need to determine the minimum possible heap sizes for each benchmark program. Note that our figures will be different to the minimum sizes in the DaCapo study [6] since they have separate heaps for VM objects and for application objects, and they only report the minimum size for the application heap. In contrast, we use the default Jikes RVM heap layout which shares a single heap for both VM and application objects. We obtain the minimum heap size for each benchmark to a megabyte granularity, by running each benchmark several times for a range of heap sizes between 8 and 512MB. Note that Jikes RVM fails to boot fully in less than 8MB. We record the smallest heap size for which the application successfully completes and there are no out-of-memory errors from the Jikes RVM optimizing compiler.

For each benchmark run (that generates a row of feature data in the training table) we calculate the ratio of the current heap size with the minimum possible heap size and use this ratio as an additional feature. So the VM-specific dynamic features are: current JVM heap size; and, ratio of current JVM heap size to minimum possible heap size in which this benchmark completes.

3.2 Benchmarks

Figure 5 shows the benchmarks used in all experiments. There is clearly a wide range of user applications from all different genres of general purpose software. Other benchmarks from these suites are excluded because they are too trivial (such as _200_check from SPECjvm98) or they do not run reliably on Jikes RVM (such as eclipse from DaCapo).

All DaCapo benchmark executions use the default input set. All SPECjvm98 benchmark executions use the default size 100 input set. The pseudojbb benchmark uses the default 7000 transactions setting. The JOlden benchmark executions use the default command line settings, with the exception of the treeadd benchmark. In this case, we altered the `levels` parameter from 20 to 22. After this alteration, the treeadd program takes around 1s for total execution time, whereas before it ran in 0.05s, which is too short for meaningful feature collection. All benchmark executions take at least 0.1s. The mean overall execution time across all benchmark executions is 4.0s.

benchmark	suite	description
_201_compress	SPECjvm98	zip compression algorithm
_202_jess	SPECjvm98	expert system shell
_205_raytrace	SPECjvm98	raytracer
_209_db	SPECjvm98	database system
_213_javac	SPECjvm98	Java compiler
_222_mpegaudio	SPECjvm98	MP3 decoder
_227_mtrt	SPECjvm98	multi-threaded raytracer
_228_jack	SPECjvm98	parser generator
pseudojbb	SPECjbb2000	database engine
antlr	DaCapo	parser generator
bloat	DaCapo	bytecode optimizer
fop	DaCapo	print formatter
hsqldb	DaCapo	relational database engine
luindex	DaCapo	text indexing tool
pmd	DaCapo	Java source code analyzer
bh	JOlden	mathematical computation
bisort	JOlden	sorting
em3d	JOlden	mathematical computation
health	JOlden	process simulation
mst	JOlden	minimum spanning tree
perimeter	JOlden	image processing
power	JOlden	process simulation
treeadd	JOlden	tree traversal
tsp	JOlden	graph optimization
voronoi	JOlden	image processing

Figure 5. Table of benchmarks used in experiments

From the DaCapo paper [6], it is clear to see that the DaCapo benchmarks stress the GC system far more than the SPECjvm98 and pseudojbb benchmarks. Similarly the JOlden benchmarks are small though pointer-intensive. Our aim is to have a wide range of GC behaviour for our training corpus.

3.3 GC algorithms

All our experiments are conducted using Jikes RVM 2.4.6. This is an open-source Java-in-Java virtual machine. It was originally developed by IBM Research but now is actively maintained by a worldwide community of academics and software engineers [1, 2]. The memory management subsystem of Jikes RVM is known as the memory management toolkit (MMTk). This is a highly modular, retargetable GC framework [5]. At VM build time, it is possible to specify which MMTk GC algorithm should be incorporated with the target VM. Of the available MMTk GC algorithms, there are

only six algorithms that work reliably with Jikes RVM 2.4.6. These are:

1. CopyMS: copying mark-and-sweep collector
2. GenMS: generational scheme with mark-and-sweep mature space
3. GenCopy: generational scheme with copying mature space
4. MarkSweep: simple mark-and-sweep collector
5. MarkCompact: simple compacting collector
6. SemiSpace: semispace copying collector

For the production-quality build configuration of Jikes RVM, the default selected GC algorithm is GenMS. In our experiments, we build six VMs that have identical build configurations except that each incorporates a different GC algorithm from the above MMTk range. Note that there is no fundamental reason why a single VM image cannot incorporate support for multiple GC algorithms. There are some engineering complications regarding write barriers and runtime compilation, but it should be possible to solve these issues.

3.4 Predictor Organization

The prediction problem is six-class, since there are six possible GC algorithms. A single predictor does not work very well in this case. The difficulty in predicting accurate answers increases exponentially with the class of the prediction problem. So a single predictor is not scalable if the number of GC algorithms increases, for instance if new algorithms were developed for MMTk.

Instead we reduce the six-class problem into a complex predictor that is a series of simpler two-class (binary) problems. The complex predictor is organized according to the schematic diagram in Figure 6. Each circle represents a binary predictor. Each arrow represents a prediction. Arrows with no children represent final predictions from the complex predictor.

Given a feature vector, the complex predictor first checks a filtering GenMS predictor to determine whether the GenMS GC algorithm is suitable for this program. If 'yes', we take GenMS as the final prediction. If 'no', then the feature vector is supplied to a full tournament predictor (shown in right hand box of Figure 6) where the remaining five GC algorithms 'play off' to see which one wins the tournament. Each round of the tournament is a single predictor that selects between two GC algorithms, to say which GC algorithm would give shorter overall execution time (these figures are obtained from the timings database). The winner then plays the next GC algorithm, until all GC algorithms have competed at least once in the tournament and the winning algorithm is the final prediction.

The order of GC algorithms in the tournament does have a slight effect on the outcome of the prediction. We evaluated several arbitrary orderings (without considering all $m!$ orderings of the m GCs) and chose the ordering that gave the best results. This is (GenCopy, CopyMS, MarkCompact, MarkSweep, SemiSpace).

The GenMS GC algorithm gets special treatment since it is the default GC algorithm in Jikes RVM. GenMS is most likely to give the fastest overall execution time for many programs, so we hope it will be the easiest to predict accurately. Thus we make this the first choice in our complex predictor. Note that we did consider a six-way tournament predictor, treating GenMS in the same way as the other five GC algorithms, but the prediction results were generally less accurate.

Each individual binary predictor is a decision tree generated by the C4.5 algorithm, a series of if/then/else tests on feature values. The predictors are automatically generated using the weka system [20], then interfaced together with custom perl scripts.

4. Evaluation

4.1 Experimental Method

We harvest values for static features from the DaCapo paper [6] as far as possible. The JOlden benchmarks are not considered in this paper, so we use the same tool as the DaCapo researchers (ckjm) to gather the static feature values for JOlden. We gather GC-independent dynamic data for each benchmark by running that benchmark using an instrumented build of Jikes RVM with a reference setup, namely GenMS GC with a heap size of 256MB. We gather GC-dependent dynamic data from another instrumented build of Jikes RVM using the reference GenMS collector, with specified heap sizes. We gather execution time information for all benchmarks, for all GC algorithms and all appropriate heap sizes using the GC-customized production builds of Jikes RVM as described above.

In an attempt to eliminate adaptive compilation activity from our dynamic and timing measurements, we take all measurements from the fifth iteration of a benchmark program in a test harness loop. By this stage, hot methods should have been recompiled and optimizing recompilation should be negligible. We refer to this situation as 'steady-state' benchmark execution. Even though the measurements are intended to be steady-state, they are still nondeterministic due to the inevitable noise in a managed adaptive runtime system. To compensate for this, we take five measurements for each feature and select the features from the run with the median execution time of these five.

In this way, we generate two databases. There is a database of feature vectors for each (benchmark, heapsize) combination, and a database of execution times for each (benchmark, heapsize, GC algorithm) combination. We identify the GC algorithm that gives the shortest overall execution time for each (benchmark, heapsize) combination.

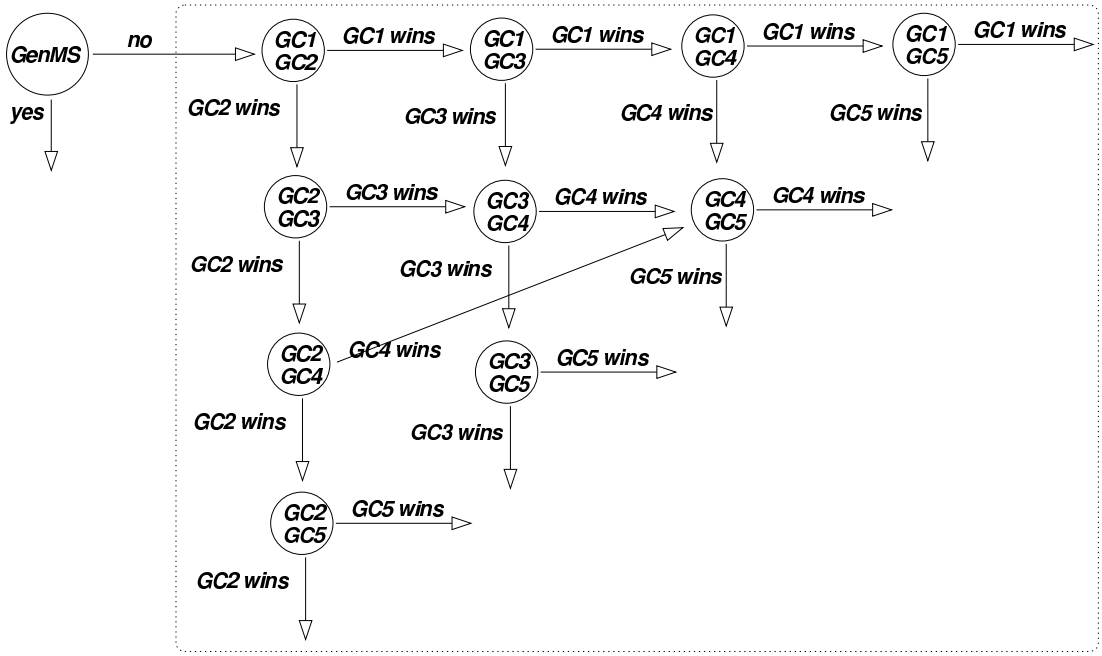

Figure 6. Structure of complex predictor for application-specific GC, showing the initial GenMS filter predictor and the tournament component

This optimal GC algorithm is the answer that our complex predictor should be able to predict.

We postprocess the database of feature vectors and send it to the weka machine learning system [20]. We generate decision trees using the C4.5 algorithm, particularly the `weka.classifiers. trees.J48` implementation. This decision tree learner does not require its inputs to be normalized and generally produces accurate predictors.

We perform leave-out-one cross-validation (LOOCV) for all our evaluations. This means that we separate our database of feature vectors into two sections. One section is known as the *training data*. This contains data for all benchmarks except one. The other section is known as the *testing data*. This only contains data for the various executions of a single benchmark. We apply the weka system using the feature vectors in the training data to construct a prediction model, then assess the accuracy of this model on the testing data. This LOOCV approach should be similar to how a prediction model would work, given a Java program that had not been used to construct the model, i.e. a previously unseen program.

For each (benchmark, heapsize) combination, we have a prediction generated by the LOOCV predictor model. We can convert these predictions into execution times by looking up the actual execution time for this (benchmark, heapsize, predicted GC algorithm) combination in our timings database. We can compare this predictive execution time with the default execution time by examining the time for (benchmark, heapsize, GenMS) in the timings database. This is how we measure the performance of our complex predictor model in the following section.

4.2 Results

Figure 7 shows how our predictor performs, in relation to the default GenMS GC algorithm and the optimal GC selection, which would require an oracle predictor. We can determine the optimal GC selection since we have a complete execution timings database so we can see which GC algorithm gives the minimum overall execution time for each (benchmark, heapsize) combination. In the graph, a score of 100 represents the default GenMS execution. A lower score represents a slowdown and a higher score represents a speedup. The results are reported as the mean score for all benchmarks at each given heap size. Note that for the smaller heap sizes, only a handful of benchmarks are included. The minimum heap size for which all benchmarks can execute is 128MB. The mean speedup over all benchmarks and heap sizes is 5% for our complex predictor, compared with a theoretical optimum limit of 17% for an oracle predictor.

An exhaustive profiling approach would require at least one execution run of each (benchmark, heapsize) test for each GC algorithm, which would make six runs of Jikes RVM. Our intelligent approach requires two execution runs of each (benchmark, heapsize) test. The first is a profiling run with an instrumented JVM to collect all the necessary dynamic feature information. (In fact, we have two profiling runs with instrumented JVMs but there is no technical reason to prevent their unification.) The second is a profiling run

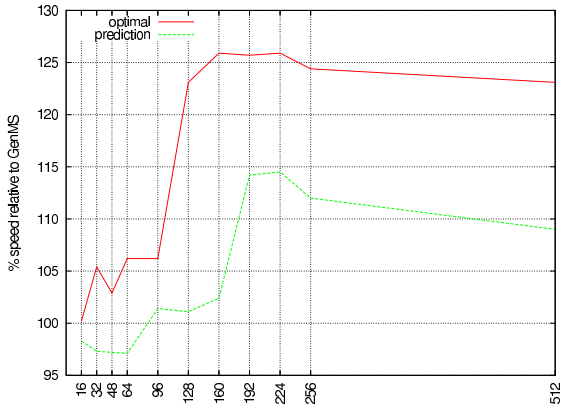

Figure 7. Results of complex predictor, with GenMS filter and 5-way tournament, for all benchmarks described in Section 3.2 at various heap sizes between 16 and 512MB

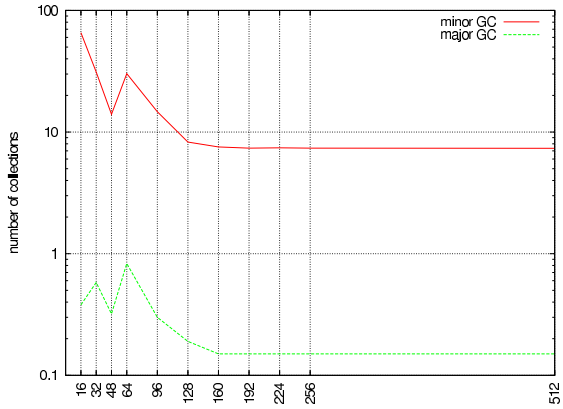

Figure 8. Mean number of GCs at various heap sizes at various heap sizes between 16 and 512MB

with a default GenMS JVM to collect the timing information for this reference GC. So our approach requires two runs of Jikes RVM. Thus, roughly speaking, we have reduced the amount of profiling required by 66%. This does not consider the fact that we need to use multiple runs and take median values (we assume this would be the same in both approaches). Also it does not take into account the slowdown that occurs in Jikes RVM for instrumentation and data gathering. However this is not overly significant.

4.3 Discussion

A possible concern is that the number of GCs may be minimal at larger heap sizes such as 512MB. Figure 8 shows how the mean number of garbage collections varies with heap size for the reference GenMS collector, over all benchmarks. Note the logarithmic scale on the y-axis. Also note that the spikes in the curves are due to new benchmarks being introduced at various heap sizes. Above 128MB all benchmarks are included.

At larger heap sizes, very few benchmarks require major GC collection. From these statistics it may be argued that learning about the best GC algorithm is not valid at such large heap sizes. However, there are still large variations in benchmark performance with different GCs. This is mostly due to the relative efficiencies of the allocators rather than the collectors. However learning based on allocator performance is definitely valid, since allocators are GC specific. We notice that at large heap sizes, GC algorithms that employ simple bump-pointer style allocators are favoured over more complicated schemes.

The problem with machine learning techniques is that they are highly dependent on training data to be representative and to have broad coverage. We have already noted that the hsqldb benchmark has extraordinary behaviour, as acknowledged in the DaCapo survey paper [6]. From our investigations, hsqldb is quite unlike the other benchmarks. For instance, when the JVM starts with a small heap which is permitted to grow in size towards a limit of 1GB, the hsqldb benchmark increases the heap to 345MB, whereas no other benchmark increases past 128MB. Thus our complex predictor is unable to make accurate GC predictions for this unusual program. The hsqldb benchmark works best with GenMS and GenCopy predictors, but according to our complex predictor model, it should do better with SemiSpace. In fact, SemiSpace gives hsqldb an execution time of around 1.75 times the GenMS time for 512MB. If we eliminate hsqldb from our results, then the mean speedup over all benchmarks and heap sizes rises to 7%, with the optimal speedup for an oracle predictor rising to 18%. Such problems should be solved with a larger training corpus.

This application-specific GC prediction is not useful if GC behaviour is highly input-dependent. So far we have only profiled each benchmark with a single set of input data. In order to determine whether GC behaviour will vary with inputs, we run some DaCapo benchmarks using the small input set, rather than the default, to examine differences in GC behaviour. We use all GC algorithms with the DaCapo small inputs and determine the optimal GC algorithm for each program. This is the information we need to construct an oracle predictor for the small inputs. Now we use the small input oracle predictor to predict the best GC algorithm, and test this on the DaCapo benchmarks running with default input sizes. If the DaCapo benchmark GC behaviour is highly input-dependent then it should not be possible to get any performance improvement from using the GC predictions based on small input execution to select GC algorithms for default input execution. Figure 9 shows the results. For one heap size, there is a slight speedup. For the other four heap sizes, there are slowdowns, with a 26% slowdown at 32MB being the most severe. The graph also shows the maximum possible speedup from exhaustive profiling of all default input executions. The mean optimal speedup is 4%.

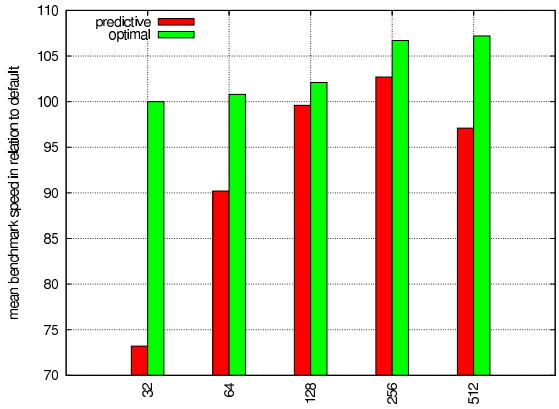

Figure 9. DaCapo benchmark speeds for various heap sizes, when training with small inputs and testing with default inputs using an oracle predictor

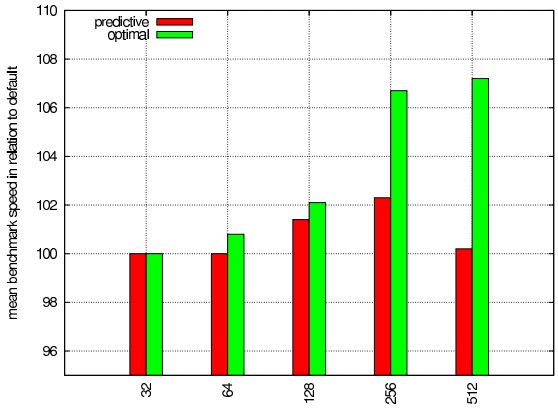

Figure 10. DaCapo benchmark speeds for various heap sizes, when training with one-off runs and testing with steady-state runs using oracle predictor

From a careful examination of the relative GC algorithm performance, we have found that across input sets, the optimum algorithm often changes, but the worst-case algorithm rarely differs. So testing on a different input set to the training data should avoid the worst-case GC, but it is not always possible to predict the best GC. This GC prediction problem shares the common weakness of all ahead-of-time profiling techniques, that it is always going to be heavily reliant on representative input data for the profiling executions.

Recall that all our program executions are from steady-state runs in test harnesses. To assess the impact of JVM optimizing compiler activity, we train with one-off program executions and test on steady-state. Similar to the above investigation, we run all one-off benchmark executions to construct an oracle predictor, then use this one-off oracle predictor to predict the best GC algorithm for steady-state runs. If the DaCapo benchmark GC behaviour changes greatly from the first execution of a program to the nth consecutive execution then there should be no performance improvement by using GC predictions obtained from the the one-off executions to select GC algorithms for the steady-state executions. Figure 10 shows the results. For three heap sizes, there is a slight speedup. The other two remain unchanged. The mean speedup is 1%, compared with a theoretical optimum of 4%. This is mostly due to the fact that optimizing compiler activity, if it accounts for a significant proportion of allocations, will tend to favour the default GenMS GC algorithm. The optimizing compiler allocates many short-lived objects during its analysis phases, ideal for generational collection. For program executions in which optimizing compilation activity is insignificant, the same GC algorithm should be best for both one-off and steady-state executions.

5. Analysis of Prediction Schemes

Machine learning techniques are simply automatic heuristic generation tools. It is often instructive for systems researchers to examine the heuristics created by such tools, both as a sanity check, and to gain insight into how predictors work.

5.1 Decision Tree to Predict Generational Collection

The first question we ask is: Do automatically generated decision trees make sense to the domain expert?

Note that for the LOOCV technique employed in our evaluation above, a different decision tree would be generated for each benchmark, since each benchmark has a slightly different training set of data. In this section, we train on all the benchmarks at once.

We choose to analyse a new prediction problem that has not been considered previously in this paper—whether or not a (benchmark, heapsize) combination is amenable to generational GC. We group the six GC algorithms into two groups. The Gen group of generational collection schemes comprises GenCopy and GenMS. The NonGen group of non-generational collection schemes comprises CopyMS, SemiSpace, MarkSweep and MarkCompact. We use the same benchmarks and feature sets as above, only now the classification is binary—either one of the Gen schemes gives the shortest time, or one of the NonGen schemes does. This simplified study has some practical value. Fitzgerald and Tarditi state that the most significant choice affecting performance is whether to use a generational collector [10].

We use all the benchmarks to create a C4.5 decision tree. Weka performs cross-validation by randomly selecting half of the input data for training, and the other half for testing. The generated decision tree gets 87% accuracy on the test data. So it is very good at predicting which programs will perform well under a generational collection scheme. Note that as a whole, 37% of the entries should be NonGen and 63% Gen.

Figure 11 shows the decision tree for this problem. It only includes five features. These are as follows:

```
dynamic_num_bytes <= 91306040
|static_lack_of_cohesion_of_methods <= 5: Gen
|static_lack_of_cohesion_of_methods > 5
||dynamic_num_minor_gcs <= 6: NonGen
||dynamic_num_minor_gcs > 6: Gen
dynamic_num_bytes > 91306040
|static_lack_of_cohesion_of_methods <= 47371
||dynamic_arrays_size_u128B <= 0.11: Gen
||dynamic_arrays_size_u128B > 0.11
|||ratio_curr_to_min_heap <= 15.515152: Gen
|||ratio_curr_to_min_heap > 15.515152: NonGen
|static_lack_of_cohesion_of_methods > 47371: NonGen
```

Figure 11. Decision Tree for Generational Collection

1. number of allocated bytes during program execution
2. value of the static LCOM metric from the CK suite
3. number of minor GCs during program execution
4. number of arrays allocated with sizes between 64 and 128 bytes, during program execution
5. ratio of current fixed heap size to minimum heap size in which this program will execute successfully.

A notable absentee is the 'nursery survival rate' feature. A low nursery survival rate indicates a program that satisfies the weak generational hypothesis [19] that most objects die young, which is the original basis for generational collection. However, as others have noted [6], even programs that have a high nursery survival rate do well with generational collection. Our decision tree shows that generational collection is actually dependent on other less intuitive program features.

5.2 Feature Selection

The 'curse of dimensionality' is a common problem in machine learning. A large number of features makes the learning problem take longer, and often leads to inferior solutions. Therefore it is useful to reduce the feature set to a subset of most relevant features. These should correlate well with the classifier, in this case with the optimal GC algorithm. Information theory provides tools to identify these features. The weka system uses the *information gain* metric to measure the significance each feature has in determining the class of the observation. When using the information gain metric on this Gen/NonGen classification problem, the top five features ranked in order of importance are:

1. total execution time with GenMS GC
2. number of allocated arrays during program execution
3. mean size of object in bytes
4. number of allocated bytes during program execution
5. value of the static LCOM metric from the CK suite

```
dynamic_num_bytes <= 91306040
|static_lack_of_cohesion_of_methods <= 5: Gen
|static_lack_of_cohesion_of_methods > 5: NonGen
dynamic_num_bytes > 91306040
|static_lack_of_cohesion_of_methods <= 47371: Gen
|static_lack_of_cohesion_of_methods > 47371: NonGen
```

Figure 12. Decision Tree for Generational Collection after Feature Selection

A decision tree trained using just these features achieves 1% higher accuracy than the previous decision tree on its test set. The new decision tree is shown in Figure 12.

So, there are interesting issues here for GC researchers. Nursery survival rate appears to be relatively unimportant in determining whether generational collection is applicable. Instead, a combination of other features seems to be more useful.

For future work, it would be good to analyse each of the predictors in the tournament setup, to determine which features are most important for identifying suitable programs for each GC algorithm. There will probably be different key features for each GC algorithm. Perhaps we could use customised feature subsets for each predictor component in the tournament.

6. Related Work

6.1 Application-Specific GC

There is a great deal of previous work that shows how different programs perform better under different GC regimes [21, 16, 10, 14, 17]. Three recent papers that apply to Java and select from a range of GC algorithms are reviewed in depth below.

Fitzgerald and Tarditi [10] advocate static selection of a GC algorithm. They choose the best GC algorithm at program compile time, since they compile Java programs into native code ahead-of-time using the Marmot optimizing compiler. They suggest having profiling runs for a program with sample input data, for all available GC algorithms. Then they select the GC algorithm that gives the fastest overall execution time for the program. Our approach shares the static GC selection philosophy, although we do not require exhaustive profiling and we use a wider range of GC algorithms.

Printezis [14] describes a system that dynamically selects between mark-and-sweep and mark-and-compact GC algorithms for the mature space in a generational scheme. He shows how to switch between these two algorithms at GC time, and presents a simple heuristic based on heap space fragmentation to determine when to switch. Our approach is static rather than dynamic, but we use a more complex heuristic based on over 20 features, and we have a wider range of GC algorithms.

Soman et al [17] extend Jikes RVM to perform dynamic switching of GC algorithms. Effectively they fragment the

JVM heap into smaller heaplets, each of which is collected according to a different GC algorithm. They perform exhaustive profiling for all benchmark and heap size combinations. They find that either one GC algorithm works best for all heap sizes in some programs, or else there is a heap size *switching point* below which one GC algorithm performs best, and above which another GC algorithm is better. Our experimental data largely confirms their findings. Their only input to a prediction model is the ratio of the current heap size to the minimum heap size for that particular program. They store minimum heap size and switching point for each program as bytecode annotations in the class files for that program. These values are read by the JVM at class loading time. When the VM changes the size of the heap, it checks to see if it has crossed the switching point and changes GC algorithm if necessary. They give details on how each GC-specific heaplet can transfer to every other GC algorithm.

In contrast, our approach has only dealt with static GC selection on fixed size heaps to date. It may be possible to extend our ideas to dynamic GC switching. It would be good to see if we could learn the heap size switching point using our prediction techniques. Then, we could use our existing data to predict the heap size switching point for a previously unseen benchmark, again avoiding the requirement for exhaustive profiling. Their speedup figures are similar to ours for most cases.

6.2 Machine Learning and GC

There has been little previous work on the application of machine learning to garbage collection. One seminal paper [3] uses machine learning techniques (particularly reinforcement learning) for GC in a JVM. Their paper presents a broad overview of how machine learning may be applied in the field of GC. In a preliminary case study, they describe a tool that addresses the issue of *when* to collect rather than *how* to collect, for a concurrent GC algorithm. They use reinforcement learning to compute optimal points in program execution for GC to occur. They have a small number of features based on heap space usage. They demonstrate a performance advantage from the learning system.

6.3 Tournament Prediction

Although tournament prediction is a term used in computer architecture, in that field it generally refers to a set of parallel predictors with a selector mechanism that chooses the result of the most accurate predictor. The Alpha 21264 branch prediction unit is constructed in this manner [11]. Our tournament is different since it runs a series of individual predictors, where each prediction result determines which predictor to run in the next round. In earlier work on instruction scheduling, one of us used a similar tournament prediction scheme to select an optimal schedule of instructions for a basic block in an Alpha 21064 simulator [13].

In the machine learning community, Beygelzimer et al [4] have recently proposed a new scheme to reduce a k-class prediction problem into $2k - 1$ binary prediction problems. Their scheme is called *filter tree*. They describe it as a 'single elimination tournament.' Like our tournament predictor, they have individual rounds that produce a binary result, with the winners going through to the next round. However whereas Beygelzimer et al build one large model to drive the whole tournament, we train many individual trees that each correspond to a decision node in the tournament.

7. Conclusions

This paper has demonstrated that application-specific garbage collectors can be predicted with custom predictors. We show a 5% speedup over the default execution time, compared with a 17% maximum possible speedup, averaged over all benchmarks and heap sizes. This is coupled with a 66% reduction in profiling time. We are not too disappointed about poorer results with smaller heap sizes. For tiny heaps, specialized algorithms such as MC^2 [15] are more appropriate. In addition, a larger training corpus and more careful feature selection could potentially improve prediction accuracy a great deal.

In deciding on a single default GC algorithm, the Jikes RVM development team certainly made the most appropriate choice. GenMS is the most common 'best-performing' algorithm on all our benchmark tests. Perhaps GenMS is the best-performing algorithm because it is most finely tuned, since it is the default. Or perhaps GenMS is the default since it is the best-performing algorithm. An adequate distinction between cause and effect may not be possible in this case. In fact, both arguments may be true.

Each of the six GC algorithms gives optimal performance for several (benchmark, heapsize) combinations. The three GC algorithms that give optimal performance for the most experiments are GenMS, GenCopy and SemiSpace. The SemiSpace GC is particularly popular for large heap sizes, and our complex predictor correctly selects SemiSpace in many such cases.

In order for this technique to be properly effective, the GC prediction scheme requires access to a large corpus of training data. Obviously, it takes a significant amount of time to profile each of the training programs with each of the GC algorithms, in addition to collecting feature information. We anticipate that this high cost is incurred at JVM install time, which users expect to be a lengthy process. Thus training should be a one-off cost, although retraining may be necessary if system parameters change dramatically at a later date (for instance after a RAM upgrade). We envisage that such technology may become commonplace, and that a 'GC auto-tuning' phase will be a standard step in the Java software development lifecycle.

For future work, we hope to extend these intelligent GC selection techniques to dynamic switching of GC algorithms, similar to [17]. However rather than having a statically determined switching criterion, we could have an

online learning system that is updated on-the-fly with data from the GC system, adapting to dynamic phase changes in program. We are convinced that there are many other applications of machine learning in GC. Andreasson et al [3] mention some of these issues and sketch a possible ML implementation. Brecht et al [7] use simple heuristics to control heap growth policies. We could easily replace such static heuristics with learned policies. Dieckmann and Hölzle [9] describe how GC implementations generally have 'a waggonload of knobs and levers which impact performance, but tuning is difficult since the right settings depend on the characteristics of the executed program.' GC algorithm selection is only the first problem—once we have an algorithm there are many parameters to set for this GC. For instance, Sun's HotSpot JVM has many different GC command line options. A few minor changes result in large variation in benchmark execution times. We feel that machine learning is particularly useful for generational style collection, since there are so many parameters to vary. These values are generally set according to contrived heuristics, which could easily be replaced by machine learning techniques in both offline and online scenarios.

Acknowledgements We thank the anonymous referees of this paper for their extremely helpful suggestions. This work was in part funded by the EPSRC Portfolio Award GR/S61270/01.

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] E. Andreasson, F. Hoffmann, and O. Lindholm. To collect or not to collect? Machine learning for memory management. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 27–39, 2002.

[4] A. Beygelzimer, J. Langford, and P. Ravikumar. Multiclass classification with filter trees, 2007. `http://hunch.net/~jl/projects/reductions/mc_to_b/invertedTree.pdf`.

[5] S. M. Blackburn, P. Cheng, and K. S. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, pages 137–146, 2004.

[6] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[7] T. Brecht, E. Arjomandi, C. Li, and H. Pham. Controlling garbage collection and heap growth to reduce the execution time of Java applications. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 353–366, 2001.

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

[9] S. Dieckmann and U. Hölzle. A study of the allocation behaviour of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 92–115, 1998.

[10] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collectors. In *Proceedings of the 2nd International Symposium on Memory Management*, pages 111–120, 2000.

[11] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 3rd edition, 2003.

[12] X. Huang et al. The garbage collection advantage: improving program locality. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 69–80, 2004.

[13] J. E. B. Moss et al. Learning to schedule straight-line code. In *Neural Information Processing Systems Conference*, 1997.

[14] T. Printezis. Hot-swapping between a mark&sweep and a mark&compact garbage collector in a generational environment. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*, pages 171–184, 2001.

[15] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC^2: high-performance garbage collection for memory-constrained environments. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 81–98, 2004.

[16] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the 1st International Symposium on Memory Management*, pages 68–78, 1998.

[17] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the 4th International Symposium on Memory Management*, pages 49–60, 2004.

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

[19] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, 1984.

[20] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2nd edition, 2005.

[21] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 87–98, 1990.