The Study and Handling of Program Inputs in the Selection of Garbage Collectors [*]

Xipeng Shen Feng Mao Kai Tian Eddy Zheng Zhang
Computer Science Department
College of William and Mary, VA, USA
{xshen,fmao,ktian,eddy}@cs.wm.edu

Abstract

Many studies have shown that the best performer among a set of garbage collectors tends to be different for different applications. Researchers have proposed application-specific selection of garbage collectors. In this work, we concentrate on a second dimension of the problem: the influence of program inputs on the selection of garbage collectors.

We collect tens to hundreds of inputs for a set of Java benchmarks, and measure their performance on Jikes RVM with different heap sizes and garbage collectors. A rigorous statistical analysis produces four-fold insights. First, inputs influence the relative performance of garbage collectors significantly, causing large variations to the top set of garbage collectors across inputs. Profiling one or few runs is thus inadequate for selecting the garbage collector that works well for most inputs. Second, when the heap size ratio is fixed, one or two types of garbage collectors are enough to stimulate the top performance of the program on all inputs. Third, for some programs, the heap size ratio significantly affects the relative performance of different types of garbage collectors. For the selection of garbage collectors on those programs, it is necessary to have a cross-input predictive model that predicts the minimum possible heap size of the execution on an arbitrary input. Finally, by adopting statistical learning techniques, we investigate the cross-input predictability of the influence. Experimental results demonstrate that with regression and classification techniques, it is possible to predict the best garbage collector (along with the minimum possible heap size) with reasonable accuracy given an arbitrary input to an application. The exploration opens the opportunities for tailoring the selection of garbage collectors to not only applications but also their inputs.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory Management (Garbage Collection)

General Terms Performance, Experimentation

Keywords Cross-Input Program Analysis, Input-Specific Selection, Selection of Garbage Collectors, Profiling, Minimum Possible Heap Size

1. Introduction

Garbage collection (GC), as one of the major tasks in managed runtime environments, critically determines the efficiency of memory management and the resulted data locality. Consequently, a large body of research has proposed various types of GC techniques. Many studies have shown that the optimal garbage collector differs in different scenarios for different applications [7, 11, 15, 18–20, 22].

Based on those observations, researchers have proposed application-specific garbage collection, in which, a specialized GC algorithm is selected for each program. Example work includes static selection during compile time by Fitzgerald and Tarditi [11], dynamic switching of GC algorithms by Soman and others [20], and machine learning based selection by Singer and others [18]. Most of these techniques require the profiling of some typical runs of the application to attain either some application-specific information (such as, the minimum possible heap size), or more directly, the best GC algorithms. They have shown considerable performance improvement for applications running on Java Virtual Machines (JVM) or Common Language Runtime (CLR).

In this work, we concentrate on a different dimension of GC selection: the influence from program inputs. Given that most application-specific selections of garbage collectors depend on profiling results, a good understanding of

[*] This work is based on an earlier work: Influence of Program Inputs on the Selection of Garbage Collectors, in the Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (March 11–13) © ACM, 2009. http://doi.acm.org/10.1145/1508293.1508307.
Major extentions are on cross-input prediction (from half of an page to 4 pages). The extensions include the use of two statistical learning techniques (Classification Trees and Nearest Neighbor) for the prediction of the best garbage collector on each heap size, the demonstration of the performance benefits brought by the input-specific garbage collector selection through a systematic comparison, and the discussions on the application of the techniques for actual uses.

the input influence is essential: If the influence is negligible, profiling one run would suffice; otherwise, cross input adaptivity would be important for the selection of garbage collectors.

While most previous work has been focused on applications, the influence of inputs remains preliminarily explored. Some studies have briefly explored the influence, using few (typically two) inputs per application. The limited explorations have led to an unclear picture: Depending on the settings, some work has seen negligible influence from inputs [20], but some have shown more significant effects [18].

The objective of this work is to offer a more comprehensive understanding of the influence of inputs on the selection of garbage collectors. We conduct a series of systematic measurement of the effects of program inputs on the performance of GC. For 10 programs from 3 benchmark suites, we collect and create hundreds of different inputs. We measure the performance of totally 316,000 executions of those programs with 5 types of garbage collectors, 1580 different inputs, and 4 heap size ratios (the ratio between the used heap size and the minimum possible heap size.) In a rigorous manner, we analyze the influence of program inputs on the top set of garbage collectors and the combined effects with heap sizes. The analysis reveals the following findings.

First, inputs influence the relative performance of garbage collectors significantly. For most programs, the top set of garbage collectors varies significantly across inputs. So, in general, profiling one or few runs is inadequate for selecting the garbage collector that works well for most inputs. *Second*, despite that influence, certain consistency does exist across inputs for all the programs in our test set: When the heap size ratio is fixed, one or two types of garbage collectors are enough to stimulate the top performance of a program on all inputs. This consistency seems to suggest a potential solution to the input-sensitivity problem: profiling a number of different inputs and selecting the most popular top garbage collector for an application. *However*, the heap size factor complicates the problem further. On some programs, the heap size ratio shows significant influence on the relative performance of different types of garbage collectors. Therefore, for the selection of garbage collectors on those programs, it is necessary to have a cross-input predictive model that forecasts the minimum possible heap size of the execution on an arbitrary input. Finally, through statistical learning techniques (Classification Trees, Nearest Neighbor and Regression techniques), we verify the predictability of the minimum possible heap size and the best garbage collectors across inputs, showing the potential feasibility of the input-specific selection of garbage collectors.

In the rest of this paper, Section 2 describes the methodology of the experiments. Section 3 reports the measurement results and exposes three-fold findings. Section 4 describes the construction of predictive models and presents the results. Section 5 discusses the exploitation of the cross-input predictability. Section 6 reviews some related work. Section 7 concludes the paper with a short summary.

Table 1. Garbage collectors used in this work

Garbage collector	Description
GC1: GenCopy	a classic copying generational collector with a copying higher generation.
GC2: GenMS*	a copying generational collector with a non-copying mark-and-sweep mature space.
GC3: MarkSweep	a mark-and-sweep (non copying) collector.
GC4: RefCount	a reference counting collector with synchronous (non-concurrent) cycle collection.
GC5: SemiSpace.SS	a copying semi-space collector.

* : The default Jikes RVM configuration for the production distribution.

2. Methodology

To uncover the effects of program inputs on the selection of GC algorithms, we measure the running time of a sequence of executions of Java programs on different inputs, heap sizes, and garbage collectors. This section presents the experimental settings, describes the performance measurement scheme, and introduces the statistical approach used for data analysis.

2.1 Experimental Settings

The machine we use is equipped with Intel Xeon E5310 processors running Redhat Linux 2.6.9 at 1.6GHz. We use Jikes RVM [3] version 2.9.1 as our Java virtual machine. Among the various garbage collectors included in the memory management toolkit (MMTK) [6] coming with Jikes RVM, we select five of them that are stable for all the executions. Table 1 lists those garbage collectors.

We select 10 programs from 3 benchmark suites to form a mix of different types of applications, as shown in Table 2. Using part rather than all of the content in the suites is because of the difficulty in the creation and collection of inputs. We do not choose a benchmark if its input is too difficult to collect or create. Furthermore, it is common in the construction of a benchmark suite that some benchmarks are obtained by simplifying the original applications. Many input options of the original applications are disabled to make the benchmark interface simple. Given that input is the focus of this work, we select the 10 programs that are close to the original application in terms of the usage and interface.

2.2 Input Collection

In the benchmark suites, most programs come with only one or two inputs, which are insufficient for a systematic study of input influence. We collect more inputs as shown in the sec-

Table 2. Benchmarks

Benchmark	Num of	Min heap	Input features	
	inputs	size (MB)	Total	Used
Compress^j	18	20–98	3	1
Db^j	100	16–31	11	2
Mpegaudio^j	30	16–20	3	1
Mtrt^j	100	15–49	2	2
Bloat^d	976	22–23	23	4
Fop^d	224	72–86	27	3
Euler^g	14	16–55	1	1
MolDyn^g	15	18–21	1	1
MonteCarlo^g	30	39–74	1	1
Search^g	8	21–21	2	1

j: JVM98 [2]; d: DaCapo [8]; g: Grande [1]

ond column of Table 2. For some programs, such as *Search*, we have a small number of inputs due to the special requirements on their inputs. During the collection, we try to ensure that the inputs are typical in the normal executions of the benchmarks. More specifically, we either collect the inputs by searching the real uses of the corresponding applications, or derive the inputs after getting a thorough understanding of the benchmark through reading its source code and example inputs. To make the benchmarks close to real applications, for some programs (*Bloat*, *Fop* and *Mtrt*), we enable some of their command-line options that were disabled by the benchmark suite interface.

2.3 Performance Measurement

In this work, we use the running time of an application as the performance metric. Because we are interested in the influence of the garbage collector selection on the entire execution, we did not use replay mode. The measured performance is not stead-state performance, but start-up performance. The running time is simply end-to-end execution time, consisting of all the time spent in both the application and the JVM.

In the experiments, we use 4 different heap size ratios. The heap sizes are multiples (1,2,4,8) of the minimum possible heap size for an application to run on an input. We measure the minimum possible heap size by conducting a binary search in a similar manner as Singer et al. do [18]. During the binary search, for a given input, we run the application on that input several times using a range of heap sizes (from 16MB to 500MB) to find the smallest size, on which the application can finish successfully. The granularity is 1MB. Note that some applications have different minimum possible heap sizes on different inputs. The third column in Table 2 shows the range of minimum possible heap sizes for every benchmark.

2.4 Statistical Performance Analysis

The goal of garbage collector selection is to select the best garbage collector—that is, to minimize the execution time of a Java application in our setting. However, as previous work suggests [12], it requires statistical analyses to compare running times of Java applications to eliminate the effects of random noises in Java virtual machines.

In this work, we adopt the approach that Georges et al. has described [12]. For every combination of *(program, input, heap size ratio, garbage collector)*, we execute it for 10 times. We then use the *Student's t*-distribution to compute the statistical *confidence interval* of the average execution time from the 10 runs. We use 90% as the confidence level (the significance level, α, is hence 0.1.) A confidence interval computed in this way shows the range that contains the true running time (i.e. the running time in a no-noise setting) with 90% probability. The key guideline of the rigorous analysis is that if the confidence intervals of two sets of runs overlap, the two sets are regarded as having no significant difference. In the context of garbage collector selection, two garbage selectors have similar performance for a program if there is an overlap between the two confidence intervals corresponding to the two garbage collectors.

This statistical analysis turns out to be vital for this work. The 10 measurements of a single combination often exhibit considerable variations in the experimental results. Those variations suggest that it would be difficult to draw reliable conclusions based on the comparison of average or minimum running times. The confidence intervals remove the effects of the variations in a large degree.

As an example, suppose for a given combination of *(program, heap size ratio, input)*, we measure its execution 5 times with GC1 used; the running times are S_1 ={22s, 22.1s, 21.9s, 22.2s, 21.8s}. We then use GC2 for the execution and get another 5 running times as S_2={21.1s, 20.8s, 20.7s, 20.7s, 22.8s}. Their average running times are respectively $mT_1 = 22s$ and $mT_2 = 21.2s$. Their confidence intervals are respectively $[20.5s, 23.5s]$, and $[19.7s, 22.8s]$. Although mT_2 is smaller than mT_1, their confidence intervals overlap with each other. So, according to the statistics theory, there is no significant difference between the two garbage collectors in terms of their effects on the program execution time. The difference between their average running times are likely caused by random noises rather than the difference between the two garbage collectors.

3. Measurement Results

This section presents three findings we obtain from the experiments. We first demonstrate that due to the influence program inputs impose on the performance of different garbage collectors, the set of the best garbage collectors rarely remain constant across inputs. The variations suggest the risks of traditional profiling-based garbage collector selection: A garbage collector selected by profiling the execution on one or few inputs may be an inferior or even the worst choice for other inputs. On the other hand, the results in Section 3.3 show that even with the influence from program inputs, it is typical for one or two garbage collectors to meet the needs

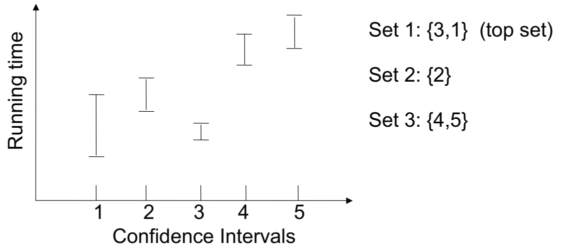


Figure 1. Illustration of the clustering scheme used for comparisons among confidence intervals.

of almost all inputs of an application. This phenomenon suggests some possible solutions to handle program inputs in the selection of garbage collectors. But the observations revealed in Section 3.4 indicate that, with the combined influence from heap sizes, some special treatment must be taken for some programs that are sensitive to heap size changes.

3.1 Metrics

Before describing the results, we first explain several concepts related to the metrics used in our data analyses.

Top Set of Garbage Collectors. In our experiment, there are 10 runs for every combination of *(program, input, heap size ratio, garbage collector)*. The 10 running times result in a confidence interval. So, for a fixed *(program, input, heap size ratio)* tuple, we have 5 confidence intervals, corresponding to the 5 garbage collectors used in the experiments. We cluster the 5 intervals into several sets based on their overlaps. The clustering works in an iterative way. It maintains a working list, initially including all the 5 confidence intervals. In each iteration, it selects the interval whose upper bound is the smallest among all the intervals in the working list as the seed of a new set. It then includes into that set all the intervals in the working list that overlap with that seed. Those intervals and the seed interval are then removed from the working list. This process continues until the working list becomes empty. At the end, every interval belongs to exactly one set, and no members of a set are significantly different from each other (according to the definition of the confidence interval.) The garbage collectors corresponding to the set constructed in the first iteration are the top performers among all 5 garbage collectors. They form the *top set of garbage collectors*.

Figure 1 illustrates the clustering scheme. The 5 intervals form 3 sets with the top set covering the first and third intervals. The second interval, for instance, forms a separate set because it is significantly different from *at least one* interval in each of the other two sets.

Coverage of a Garbage Collector. For a given program and heap size ratio, all the runs of the program on each input yield one top set of garbage collectors. The coverage of a garbage collector, *gc*, is the number of the top sets that include *gc* divided by the total number of top sets. For instance, there are 10 inputs and the top sets of 6 inputs include GC1. The coverage of GC1 is $6/10 = 0.6$.

Top Garbage Collector. The top garbage collector of a program is the collector with the largest coverage for that program.

3.2 Variations of the Top Set of Garbage Collectors

The runs of each combination of *(program, input, heap size ratio)* yields a top set of garbage collectors. The cross-input variations of those sets reflect the influence of program inputs on the relative performance of different garbage collectors.

The pie graphs in Figure 2 summarize the cross-input variations for every program and heap size ratio. Take *MonteCarlo* as an example. When r=1, for 60% of its inputs, the corresponding top sets of garbage collectors are $\{$GC1, GC3$\}$; for 75% of its other inputs, the top sets are $\{$GC1, GC2, GC3$\}$; for the remaining inputs, the top sets are always $\{$GC3$\}$. The three kinds of top sets correspond to the 3 pieces in the leftmost pie of *MonteCarlo* in Figure 2. When the heap size ratio becomes larger (r=2), the set $\{$GC3$\}$ becomes the top set for every input. The corresponding pie thus has no splits at all.

The number of pieces in a pie is equal to the number of unique top sets. Most pies in Figure 2 have some splits, showing that the top set of garbage collectors changes across inputs for most of the programs.

To understand the reasons for the input influence, we take program *Mtrt* (when r=1) as an example. Garbage collectors GC2 and GC3 are two of the most popular collectors in the top sets of *Mtrt*. However, the ranking between them changes across inputs. For 9% of the inputs of *Mtrt*, their top sets include GC3 but not GC2. For 23% of the inputs, the top sets include both. And for the other 68% inputs, the top sets include GC2 but not GC3.

Figure 3 (a) and (b) reveal the reasons for such cross-input differences. Figure 3 (a) shows the running times of *Mtrt* when it runs on the smallest heap size and either GC2 or GC3 is used. The inputs are ordered in the program's corresponding running times under GC2, from the shortest to the longest. The crosses ("x") in the figure indicate those inputs whose corresponding top sets of garbage collectors include GC3. Almost all of those inputs are among the smallest; the corresponding segments of the GC2 and GC3 curves are close to each other. As the input size increases, the gap between the two curves enlarges and GC3 becomes unfavorable after the 32nd input. One of the reasons for the enlarging gap is that as input becomes larger, the time spent by GC3 increases faster than the time by GC2, as shown in Figure 3 (b).

A more detailed analysis on Figure 3 (b) exposes the reason why the 9% inputs have GC3 but not GC2 in their top sets. Those inputs are the inputs 1 to 9 in the Figure 3. As shown by Figure 3 (b), GC2 and GC3 have similar GC

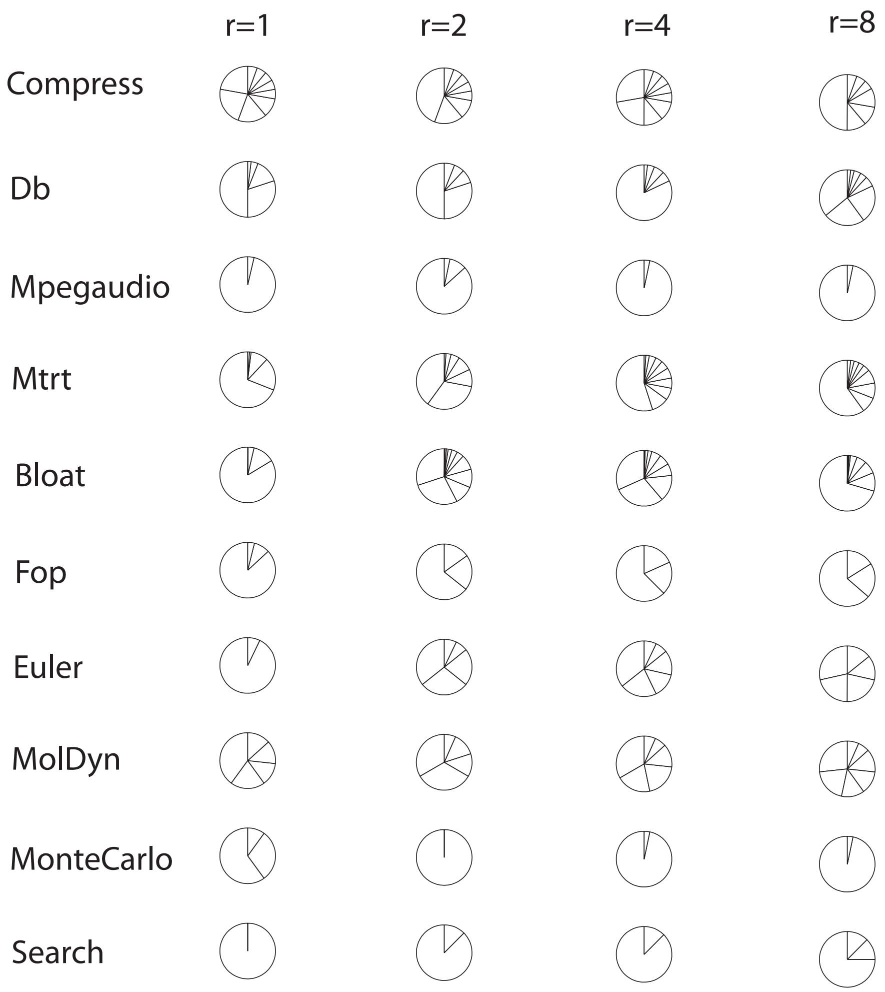


Figure 2. The influence of inputs on the top sets of garbage collectors. Each column of pies correspond to one heap size ratio (r is the ratio of the heap size to the minimum possible heap size.) Each piece in a pie shows the fraction of a program's inputs whose top sets of garbage collectors is equal to a particular set. The number of pieces in a pie equals the number of unique top sets of garbage collectors of the program under a given heap size ratio.

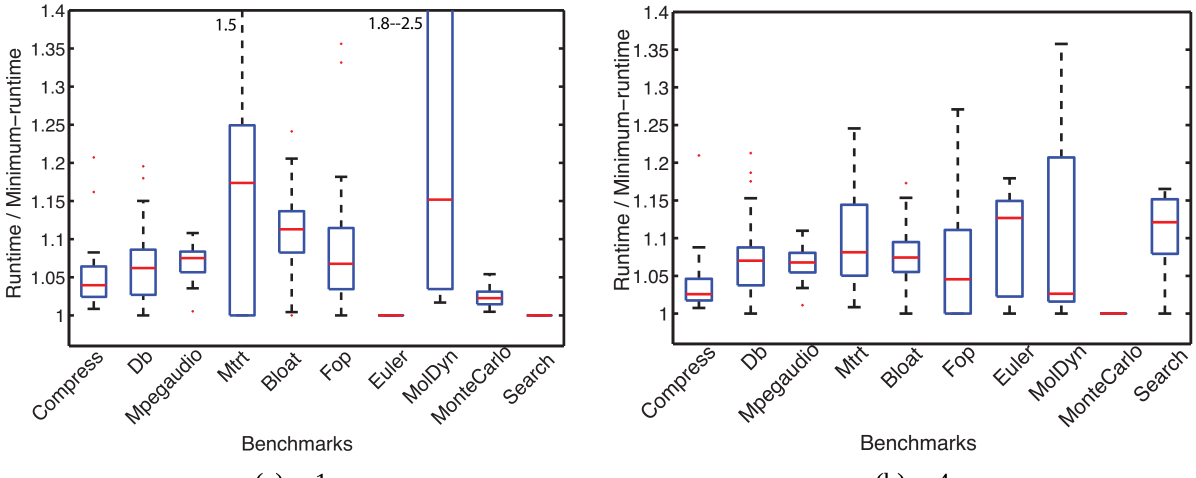
times on each of those inputs. So, the reason for the better performance of the program under GC3 than GC2 is because GC3 brings better data locality to the program and shortens the mutator running time. This explanation concurs with the performance on inputs 10 to 32, on which, although GC3 takes longer time to finish than GC2, the program performs similarly on the two garbage collectors.

Figure 3 (c) exposes the number of garbage collections. Although GC2 is invoked more often than GC3, an invocation of GC2 takes less time to finish than an invocation of GC3.

Implications to Garbage Collector Selection. The example of *Mtrt* reflects the potential risk of the existing approaches in profiling-based garbage collector selection. In those approaches, typically very few (one or two) inputs are used for profiling to select the best garbage collector. If the input used for profiling of *Mtrt* happens to be a small input, GC3 may be chosen as the best garbage collector. That decision would cause the program inferior performance on most large inputs, reflected by the gap between the two curves on Figure 3 (a).

The many splits in the pies in Figure 2 suggest that such risks exist for almost all programs. To demonstrate the potential severity of such risks, the boxplots in Figure 4 show the performance when the user happens to choose a garbage collector that, although appearing in the top sets of more than 20% inputs, is not in the top sets of the majority of the inputs. (The greater than 20% coverage suggests the nontrivial probability for the garbage collector to be chosen in a profiling-based selection.) We normalize the running time by the time achieved when the best garbage collector is used for every input.

When r=1, except *Euler* and *Search*, all programs show significant performance degradations. Programs *Mtrt* and



(a) r=1

(b) r=4

Figure 4. The potential performance degradation of input-oblivious garbage collector selection.

MolDyn show up to 1.5 and 2.5 times slowdown. When r=4, all programs except *MonteCarlo* exhibit significant slowdown.

The results draw the conclusion that selecting garbage collectors by profiling few inputs is subject to the risk of significant performance degradations.

3.3 Consistency of the Top Garbage Collector

The previous section shows that program inputs complicate the selection of garbage collections. The analysis in this section, on the other hand, shows another aspect of the measurement, and suggests a potential way to address the input influence.

In Figure 5, the largest piece in a pie shows the fraction of a program's inputs whose top sets of garbage collectors include the garbage collector that has the largest coverage. The other pieces in a pie show how other garbage collectors cover the remaining inputs. Take *MolDyn* as an example. When r=8, GC2 is the top garbage collector with 60% coverage. GC3 is the most popular one in the remaining 40% inputs, covering 82.5% of them. GC1 then covers the remaining inputs. Together, the 3 garbage collectors correspond to the three pieces in the rightmost pie of *MolDyn* in Figure 5.

In contrast to Figure 2, the pies in Figure 5 have fewer pieces. Out of the 40 pies, only one of the 40 pies have more than 2 pieces, and 15 of the pies consist of only 1 piece in each. For every program except *MolDyn*, there exist one garbage collector that can cover over 83% inputs of the program. Two garbage collectors are virtually enough to cover all inputs for all the programs.

Implications to Garbage Collector Selection. Figure 6 shows the performance degradation when the top garbage collector is used for all inputs to a program. Compared to Figure 4, the degradations become much smaller. Most programs have less than 3% degradations on most inputs. This result suggests that, given a fixed heap size ratio, using the top garbage collector is often sufficient for the selection of a reasonably good garbage collector.

However, if the heap size ratio changes, the problem becomes more complex. Before discussing the effects of heap sizes, we note that Figure 5 should not be used for understanding the influence of heap sizes. Some information it leaves out may cause misleading conclusions. For example, the r=4 pie of *Search* contains no portion of GC2 at all, even though GC2 has a coverage of 87.5% in that scenario. The reason for not having GC2 in the pie is because GC1 and GC3 together already form a full coverage of the pie.

3.4 Influence from Heap Sizes

The size of heap significantly influences the number of garbage collections that happen in an execution. Table 3 reports the average number of garbage collections in an execution of each program. The increase in heap size reduces the number of garbage collections. It also causes changes in the ranking of garbage collectors.

To demonstrate the influence from heap sizes on garbage collector selection, we examine how the coverage of a garbage collector changes across heap sizes. Figure 7 show the cross-heap-size changes of the coverage of the top garbage collector for a program. For example, *Mtrt* shows -40% changes in the r=1 case, indicating that by applying the top garbage collector obtained when r=2 to the executions when r=1, 40% more runs would suffer from significant performance degradations. (The significance is in the sense of statistical confidence.) Whereas, there are positive changes in the cases of r=4 and r=8 for *Mtrt*. The positive changes are also reflected by the rightmost 3 pies of *Mtrt* in Figure 5: The most popular garbage collector is GC3 for all three cases and its coverage increases as heap size increases. (We note again that Figure 5, even though showing some heap-related information, cannot be used for analyzing heap size

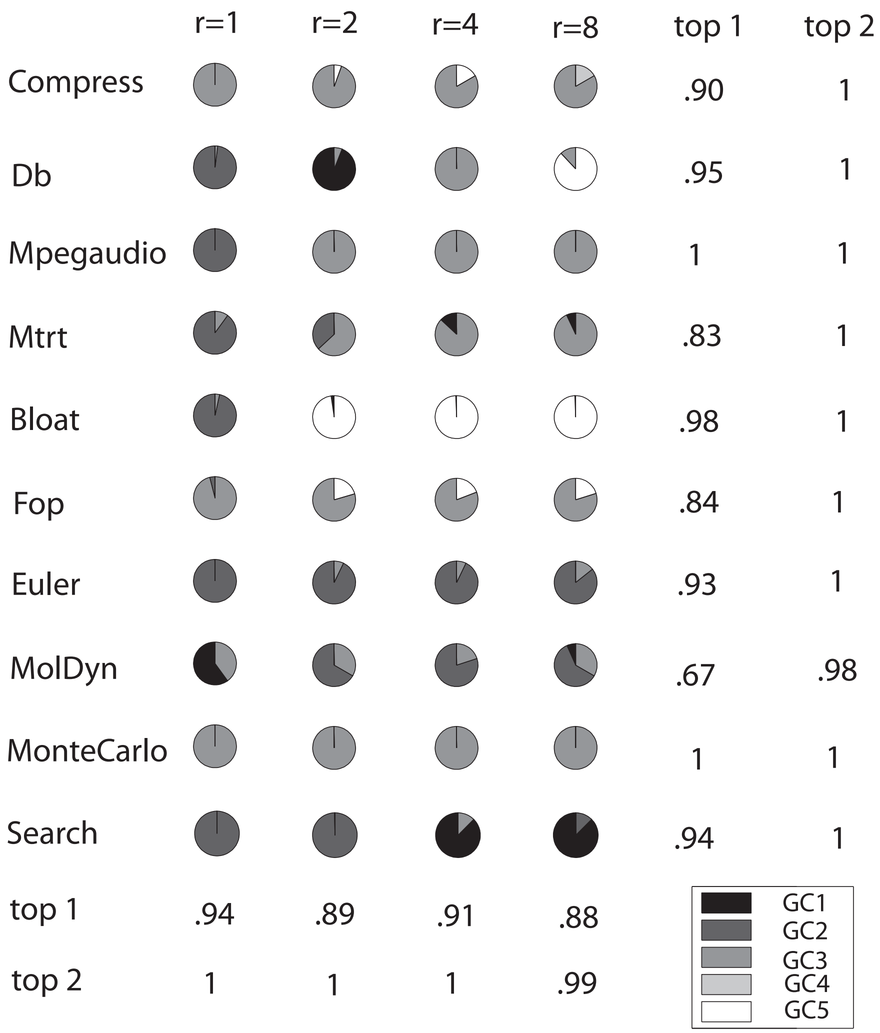


Figure 5. The influence of inputs on the top garbage collectors. The largest piece in a pie shows the fraction of a program's inputs whose top sets of garbage collectors include the overall top garbage collector. The other pieces in a pie show how other garbage collectors cover the remaining inputs. Each column of pies correspond to one heap size ratio (r is the ratio of the heap size to the minimum possible heap size.) The right-most two columns show the fraction of the most dominant garbage collector(s) averaged across the 4 heap sizes. The bottom two rows show the corresponding fraction averaged across benchmarks.

effects in general.) Some bars, like the r=4 bar in *Search*, are invisible because their values are 0.

There are small coverage changes on r=4 and r=8, but some large changes on r=1. The reason is that the heap is large enough in all the cases of (r=2,4,8). So as showed in Table 3, the number of garbage collections does not change as dramatically as between the (r=2) and (r=1) cases.

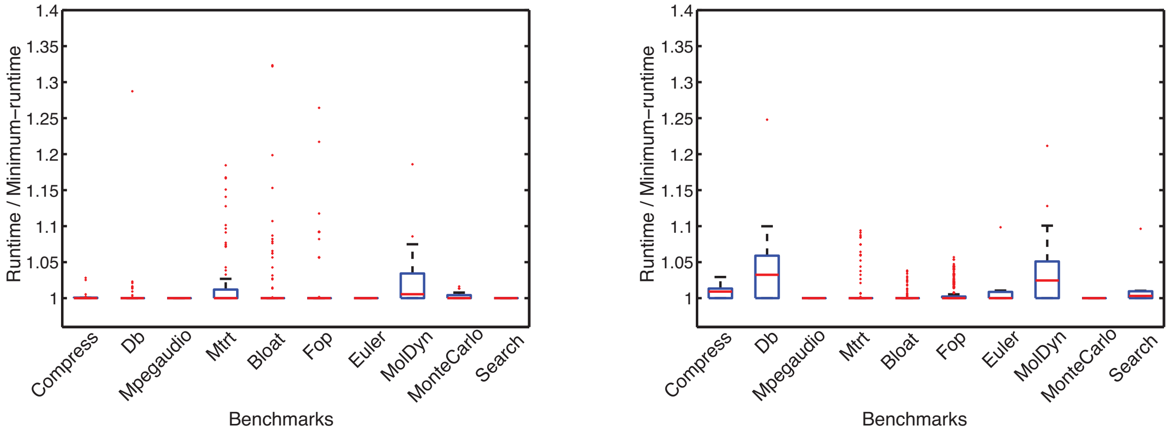
Figure 8 shows the similar barplots as Figure 7, except that the garbage collector that is used is obtained in the case of (r=1) rather than ($r = 2$.)

On both figures, the programs fall into two categories: Programs that are sensitive to heap size ratio changes—including *Db*, *Mpegaudio*, *Mtrt*, and *Bloat*, and the insensitive programs—including all other programs. For the insensitive programs, profiling multiple inputs on one heap size should be sufficient for the selection of garbage collectors.

But for the sensitive programs, it is necessary to profile on not only multiple inputs but also *multiple heap sizes*. But even with that, it is still not enough: To use the right garbage collector for a new run, we have to first determine the heap size ratio of the current run. To do that, we must know the minimum possible heap size of this run besides the given heap size. Unfortunately, as shown in Table 2, the minimum possible heap size may change across inputs. Therefore, to solve this problem, we need to have a cross-input predictive model that can forecast the minimum possible heap size for an arbitrary run of a program.

Table 3. The average number of garbage collections

Benchmark		Compress	Db	Mpegaudio	Mtrt	Bloat	Fop	Euler	MolDyn	MonteCarlo	Search
r=1	GC1	3.9	216.3	35.4	86.0	2.1	2.3	150.5	4.2	3.0	1076.6
	GC2	3.0	94.1	12.6	10.3	1.0	1.6	73.1	2.2	3.0	822.4
	GC3	2.0	23.5	5.8	5.6	0.0	1.0	18.3	1.0	3.0	548.0
	GC4	2.7	65.4	12.8	12.9	1.0	3.1	47.9	1.7	4.0	754.3
	GC5	3.7	116.3	19.3	53.0	2.0	2.0	71.8	3.6	3.0	1024.8
r=2	GC1	2.0	41.2	5.6	5.1	0.0	1.0	27.4	1.0	2.0	349.2
	GC2	2.0	29.2	4.4	4.3	0.0	1.0	19.2	1.0	2.0	299.7
	GC3	2.0	10.1	3.0	2.8	0.0	0.0	8.1	1.0	2.0	155.0
	GC4	2.0	16.3	3.8	4.0	0.0	0.0	10.8	1.0	2.0	212.4
	GC5	2.0	17.0	4.8	4.9	0.0	0.0	14.1	1.0	2.0	335.4
r=4	GC1	2.0	9.6	3.0	3.1	0.0	1.0	12.0	1.0	2.0	138.1
	GC2	2.0	9.1	3.0	3.0	0.0	1.0	11.4	1.0	2.0	138.1
	GC3	2.0	5.2	2.0	2.1	0.0	0.0	3.6	1.0	2.0	102.1
	GC4	2.0	7.9	2.2	2.6	0.0	0.0	7.3	1.0	2.0	85.0
	GC5	2.0	8.2	3.0	2.7	0.0	0.0	6.4	1.0	2.0	129.6
r=8	GC1	2.0	8.4	2.9	3.0	0.0	1.0	11.9	1.0	2.0	138.1
	GC2	2.0	8.4	2.9	3.0	0.0	1.0	11.3	1.0	2.0	138.1
	GC3	2.0	3.4	2.0	2.0	0.0	0.0	2.9	1.0	2.0	91.0
	GC4	2.0	7.8	2.0	2.6	0.0	0.0	6.8	1.0	2.0	52.7
	GC5	2.0	4.8	2.0	2.2	0.0	0.0	5.2	1.0	2.0	106.3



(a) r=1

(b) r=4

Figure 6. The potential performance degradation if the top garbage collector is used for all inputs.

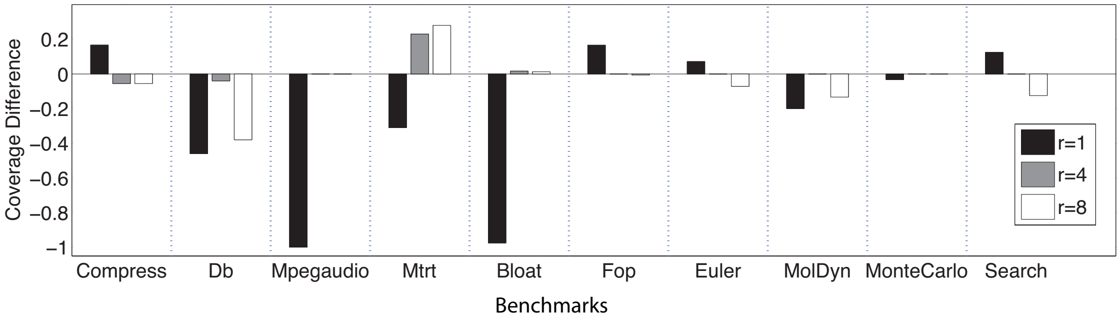


Figure 7. Coverage changes due to the changes in the heap size ratio. Each bar shows: *Coverage (gc, r=i) - Coverage (gc, r=2)*, (i=1,4,8), where, *gc* is the top garbage collector when r=2.

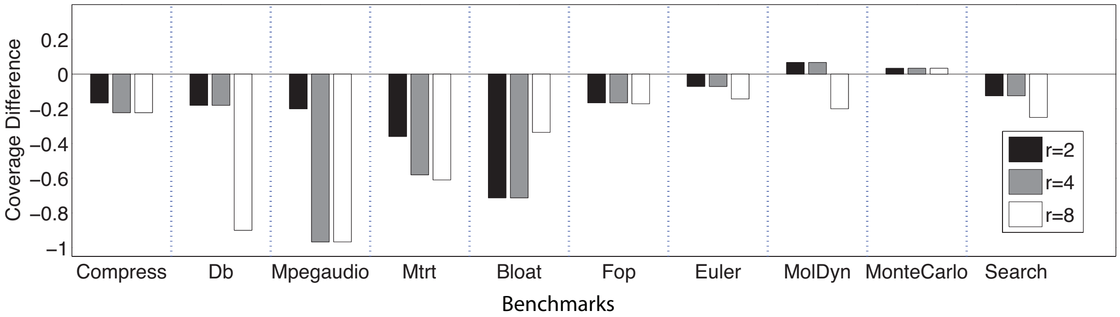


Figure 8. Coverage changes due to the changes in the heap size ratio. Each bar shows: *Coverage (gc, r=i) - Coverage (gc, r=1)*, (i=2,4,8), where, *gc* is the top garbage collector when r=1.

4. Cross-Input Predictability

Given the significant influence from program inputs, the second part of this work attempts to explore the possibility of addressing the influence by predicting it. Our objective is to investigate whether it is possible to construct a predictive model for an application to capture the relations between its inputs and their influence on the best GC algorithms. With such a model, by feeding the features of an arbitrary input into the model, we would be able to forecast the best GC algorithm for an execution of the program on that input.

4.1 Statistical Learning Techniques

We formulate the prediction of the best input-specific garbage collector for a given program as a classification problem. Each garbage collector is a class label. The training data is a set of pairs, $< I_i, GC_i >$, where I_i is the feature vector of a program input, and GC_i is the best garbage collector for the program's execution on that input (on a given size of heap.) The goal is to use the training data to construct a function that maps from input feature vectors to the best garbage collectors. The function is predictive in the sense that it reports the GC algorithm that suits the execution of the application on an arbitrary input.

The problem of constructing such a mapping function is a typical statistical learning task. There are a large number of techniques developed in the realm of statistical learning for classification [13]. We choose Classification Trees and Nearest Neighbor (NN) methods, which represent two different learning strategies.

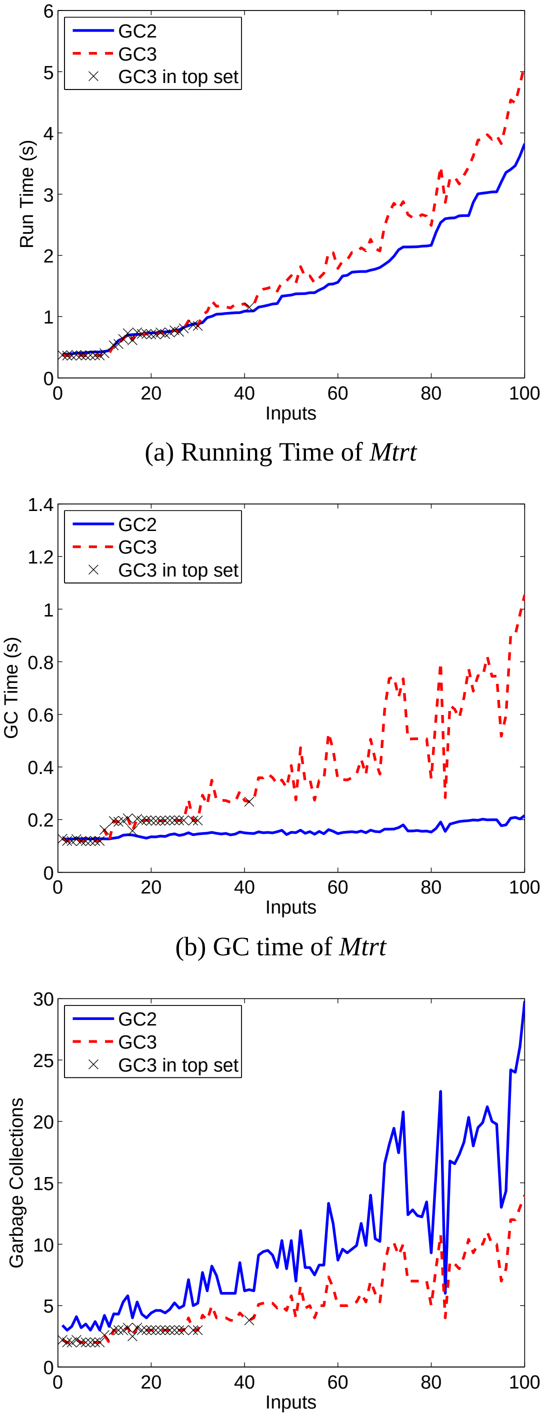
Classification Trees is a divide-and-conquer learning approach, which divides the input space into local regions, each having a class label. Figure 9 shows such an example. Each non-leaf node asks a question on the input features and each leaf node has a class label. The class of a new input equals to the label of the leaf node that the input falls in. The question asked in a non-leaf node is automatically selected in the light of entropy reduction—that is, the increase of class label purity of the data set after the data are split on their answers to the question in that node.

NN is a memory-based learning approach, which, for a test instance, finds its closest neighbor (in the feature space) among all training instances and uses that neighbor's class as the class of the test instance. NN requires the definition of the distance between instances. In the setting of GC prediction, the distance refers to the distance between the feature vectors of two program inputs.

Besides predicting the best garbage collectors, we also experiment cross-input prediction of the minimum possible heap size, a property that has been used in application-specific selection of garbage collectors [18, 20]. This task is a regression rather than classification problem, because heap size is a quantitative property. We select Regression Trees method for the model construction. Regression Trees is similar to Classification Trees, except that each leaf offers a quantitative value rather than a categorical class label. In our implementation, we use Least Mean Squares (LMS) to produce a linear model that fits the data in each leaf node. For every program, we use all its training runs, $\{< I_1, mhs_1 >, \cdots, < I_k, mhs_k >\}$ (mhs for minimum possible heap size), to construct such a regression tree. For a new input to the program, its mhs can be then predicted using the linear model inside the leaf node where this new input falls.

The reasons for us to select those three learning techniques are on multiple folds. First, the techniques handle both discrete and numeric features, both of which are common in program input features. Second, the techniques are simple and efficient, thus potentially applicable for practical uses. Finally, Classification Trees and Regression Trees have good interpretability. The trees can be converted into a set of rules that a human can easily understand and validate.

We stress that the objective of this work is to investigate the cross-input predictability of GC selection, laying the foundation for practical solutions to address input influence, rather than offering a completely practical solution. There are some complications in employing the cross-input prediction techniques for practical uses, such as how to make the training process transparent to users, and how to control



(a) Running Time of *Mtrt*

(b) GC time of *Mtrt*

(c) Number of Garbage collections of *Mtrt*

Figure 3. The average running time and GC time of *Mtrt* on GC2 and GC3 when r=1. The crosses ("x") indicate those inputs whose top sets of garbage collectors include GC3. As input becomes larger, the time spent by GC3 increases much faster than by GC2, turning GC3 from favorable to unfavorable.

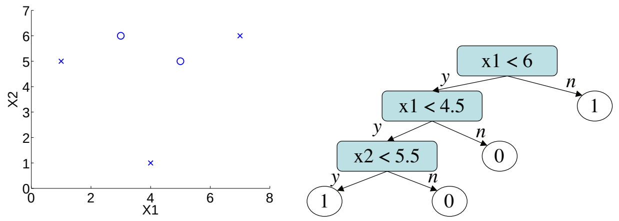


Figure 9. A training dataset (o: class 1; x: class 2; X1 and X2 are two input features) and the classification tree.

learning overhead. Section 5 discusses those complications and possible solutions.

4.2 Input Characterization

For cross-input prediction, it is necessary to convert a program input into a structured format with important input features contained. In this work, we use the eXtensible Input Characterization Language (XICL) [16] for the transformation. XICL provides an extensible way for a programmer to specify the input format and the potentially important input features for a program. An XICL translator, based on the XICL specification and program runtime values, automatically converts a program input into a well-formed feature vector. The right-most two columns in Table 2 show the numbers of features extracted by XICL translator before and after feature refinement.

XICL-based characterization has some limitations, such as the need for programmers' efforts. However, enhancement of input characterization is a research topic out of the focus of this paper: No matter how the input vectors are formed, this work explores whether the learning techniques can produce accurate cross-input predictive models.

4.3 Prediction Results

We experiment two approaches for input-specific selection of garbage collectors.

As shown in Figure 5, the best garbage collector for a program is quite stable across inputs for a given heap size ratio. Therefore, we may determine the top collector (i.e. the one having the largest coverage of inputs) of a program for each of a sequence of heap size ratios through some training runs. For a new run of the program, if we can predict the heap size ratio of the run, we may achieve good results by simply choosing the corresponding top garbage collector. This is the scheme adopted by our first approach for input-specific selection of garbage collectors. Its key is the prediction of minimum possible heap sizes from program inputs.

The second approach is similar to the first one, also requiring the prediction of the minimum possible heap size to determine the heap size ratio of the current run. The difference is that instead of selecting the top one garbage collector, the second approach predicts which of the top two collectors fits the current input better. The rationale is that although the top collector has high coverage of the inputs for most of the programs, its coverage is less than 85% for some programs like *Mtrt, Fop, MolDyn*. In contrast, the top 2 collectors cover over 98% of the inputs for every programs. Therefore by adaptively selecting the more suitable one of the the top 2 collectors for the current execution, the second approach could possibly achieve better results than the first approach does.

The prediction of minimum possible heap sizes is the same for the two approaches. We report the results in Section 4.3.1. Section 4.3.2 report the effectiveness of the two approaches in selecting the best garbage collectors and the corresponding improvement of program performance.

In all prediction experiments, we use leave-one-out strategy [13] for evaluation. Each time, we pick one input out of the whole input sets as the testing input, and use the rest for the construction of predictive models. The prediction accuracy is the average of the accuracy on all inputs.

4.3.1 Minimum Possible Heap Size

Table 4 contains the prediction accuracy of minimum possible heap sizes using the Regression Trees method. The accuracy is computed as $1 - (|S - \hat{S}|/S)$, where, S and $\hat{S}$ stand for the real and predicted minimum possible heap sizes respectively.

Different GC algorithms have different heap requirement. The table lists the prediction accuracy of each of the 5 GC algorithms. In the selection of GC algorithms, it is useful to know the maximum of the minimum possible heap sizes of all garbage collectors. The right-most column in Table 4 corresponds to the prediction on such maximum values.

For most of the programs, the prediction accuracy is larger than 95%. Program *Mtrt* and *Euler* have the lowest prediction accuracies: 86-91% for *Mtrt*, and 90-95% for *Euler*. The numbers of inputs to those two programs are relatively small. *Mtrt* has 100 but its input feature vectors have two dimensions; *Euler* has 14 inputs only. With more inputs, the prediction accuracy may become better. We note that program *Search* has even fewer inputs than those two programs, but its minimum possible heap size is a constant across inputs, which accounts for its high prediction accuracies. Overall, this experiment demonstrates that minimum possible heap size has good cross-input predictability, and Regression Trees method is an approach suitable for the prediction.

4.3.2 GC Selection

As mentioned at the beginning of Section 4.3, the first approach to input-specific GC selection simply selects the corresponding top GC after finding out the heap size ratio. The second approach needs the prediction of the better one between the top two garbage collectors. In this section, we first present the accuracy of the prediction of the best GC in the second approach, and then report the performance benefits brought by both approaches.

Table 5. Accuracy in predicting between top 2 garbage collectors

Benchmark	r=1		r=4	
	CT	NN	CT	NN
Compress	100	100	83.3	38.9
Db	100	100	100	100
Mpegaudio	100	100	100	100
Mtrt	90.9	81.8	100	100
Bloat	100	100	100	100
Fop	100	100	76.9	92.3
Euler	100	100	77.8	88.9
MolDyn	75.0	75.0	100	100
MonteCarlo	100	100	100	100
Search	100	100	75.0	25.0
Average	96.6	95.7	91.3	84.5

r: heap size/ the minimum possible heap size;
CT: Classification Trees; NN: Nearest Neighbor

GC Prediction Accuracy We investigate both the Classification Trees and the NN methods for the prediction of input-specific garbage collectors in the second approach.

Table 5 reports the prediction accuracy when the heap size ratios are 1 and 4. The two prediction methods, Classification Trees and NN, have similar overall prediction accuracy. When the heap size ratio is 1, the average prediction accuracies are both over 95% for both methods; the accuracies become 91.3% for the Classification Trees and 84.5% for the Nearest Neighbor methods when the heap size ratio becomes 4. The accuracy drop is mainly due to the more evident influence of inputs on garbage collector selection on the larger heap size ratio, as shown in Figure 5. The low accuracy of *Search* when the heap size ratio is 4 indicates that the Nearest Neighbor method is not suitable for that program.

We decide to use Classification Trees for the prediction of the second approach. Next, we report the performance benefits brought by the input-specific garbage collector selection.

Performance Improvement Figure 10 contains the performance of the benchmarks when different GC selection is used. Each benchmark corresponds to 4 bars, showing the ranges of the normalized running times of all the executions of the benchmark. The divider in the normalization is the running time of the application when the garbage collector best for the execution (among all of the 5 garbage collectors) is used. The first bar corresponds to the running times when the default GC algorithm is used. The second and third bars correspond to the input-specific GCs predicted by the two approaches described earlier in this section. The fourth bar shows the range when the real input-specific GCs (selected from the top 2 GCs) are used.

The results lead to the following points. First, the default GC algorithm causes significant performance degradation to some programs, compared to the performance achieved by the best garbage collector. On program *Fop*, for instance, the running time is 10% longer on average when the heap size

Table 4. Prediction accuracy of minimum possible heap size under each garbage collection algorithm and the overall maximum

Benchmarks	GC_1	GC_2	GC_3	GC_4	GC_5	Max
Compress	99.8	99.8	100	100	99.9	99.9
Db	98.1	97.4	98.2	97.0	97.8	98.2
Mpegaudio	100	98.1	96.3	96.0	99.6	96.8
Mtrt	86.1	90.5	87.4	90.5	89.7	90.7
Bloat	99.9	100	99.7	99.4	99.9	99.9
Fop	98.2	97.2	96.6	98.3	97.7	98.3
Euler	91.3	92.7	91.4	95.2	90.4	93.9
MolDyn	98.6	99.0	98.1	98.8	99.3	98.6
MonteCarlo	98.9	99.1	99.4	99.3	99.5	99.3
Search	100	100	100	100	100	100
Average	97.1	97.4	96.7	97.4	97.4	97.5

ratio is 1, and over 25% longer on average when the heap size ratio is 4. Most programs show more than 10% slowdown, especially on the large heap size ratio. On average, the mean degradation is about 4% and 8% on the two heap size ratios, and the maximum slowdown is 7% and 17% respectively. The more significant slowdown on the larger heap size ratio is due to the more complex distribution of the top garbage collectors as shown in Figure 5.

Second, the input-specific GC predicted by the two approaches show similar results, both outperforming the default GC significantly. The overall average slowdown becomes less than 1%; the maximum slowdown is cut to less than 5% on average. The results from the second approach are slightly worse than those from the first approach. It is because of the influence of the prediction errors.

Finally, the average running time using the real input-specific GC is very close to the possible minimum. This result is consistent with the high coverage of the top 2 garbage collectors as shown earlier in Figure 5.

In summary, cross-input predictive models are effective for the prediction of the minimum possible heap size. The input-specific garbage collector selections may improve the performance of the programs to the near-optimal.

5. Discussions

This section discusses the complications in exploiting the cross-input prediction in practical managed runtime environments. One of the difficulties is the need for profiling runs and a learning process. Sometimes, it is difficult to find a large set of representative inputs to do profiling and to build the predictive models. One possible solution is to move the learning process to the real uses of the application [14]. Static selection techniques [11, 18] may produce a small set of likely-best garbage collectors for an application during compile time. During real uses of the application, each time, the runtime environment picks one of the garbage collectors for execution and records the input feature vectors and the running times into a database. The database gradually grows and the predictive models can thus be constructed from it. Furthermore, multiple users may share their database to accelerate the learning process. This user-end incremental learning scheme eliminates the need for explicit profiling and circumvents the difficulty in collecting representative inputs.

To prevent model construction from hurting application performance, we can always do the construction during the idle time of the machine. To control the quality of the learned models, we can use the prediction errors on the data in the database as a confidence measurement to enable discriminative prediction—predict only when confident.

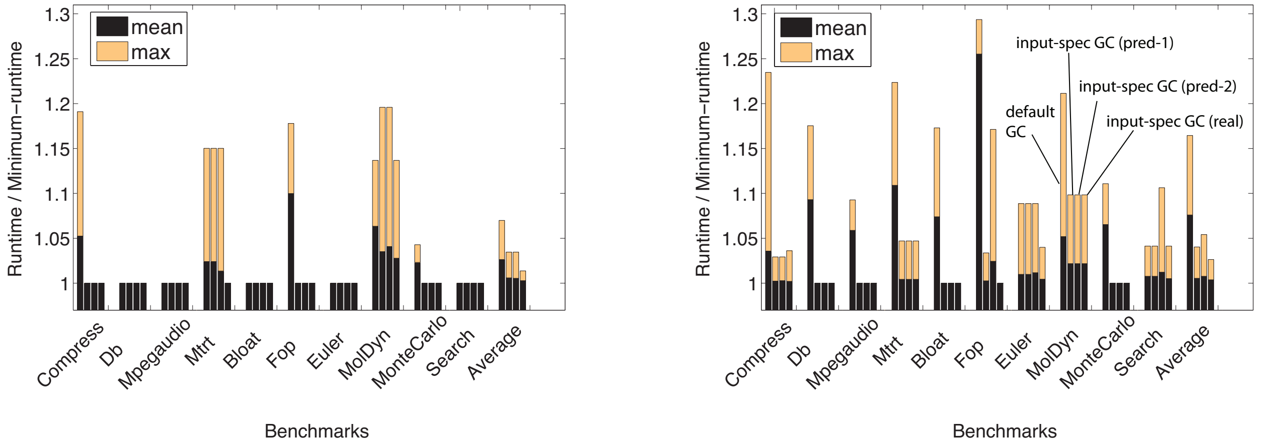
6. Related Work

We are not aware of any previous work that has systematically studied the influence of inputs on GC selection with a large set of inputs. Neither have we found any work on *cross-input* prediction of minimum possible heap sizes.

In this section, we first compare this work with previous studies on the selection of garbage collectors, and then review the prior explorations on handling the influence of program inputs on other program optimizations.

Selection of Garbage Collectors A number of previous studies have shown that for different applications, the best performer among a set of garbage collectors are different [7]. As early as a decade ago, researchers have started the comparison of the performance of different GC algorithms on different applications. Examples include the comparison between mark-and-sweep and copying-based GC algorithms from Zorn [22] and Smith and Morrisett [19].

More recent studies fall into two categories based on whether the technique allows static or dynamic selection of GC. Fitzgerald and Tarditi [11] propose a profiling-based approach for static GC selection. In their approach, the best GC for an application is selected during compile time, based on the profiling results of multiple runs of the application on a sample input with different GC algorithms enabled. Singer et al. [18] use machine learning techniques to predict the best GC for an application.



(a) heap as large as minimum possible size

(b) heap four times as large as minimum possible size

Figure 10. Comparison of normalized running times. The lower the better. The four bars (left to right) of each benchmark correspond to the running time of the benchmark when the garbage collector uses the default GC algorithm, the predicted input-specific GC selected by approaches 1 and 2, and the real input-specific GCs.

Some other research conducts finer-grained GC selection by allowing the switch of GC algorithms in the middle of an execution. Printezis [15] proposes a scheme to enable the dynamic selection between mark-and-sweep and mark-and-compact GC algorithms to manage the mature space in a generational scheme. The technique relies on a simple heuristic on heap space fragmentation. Soman et al. [20] develop a scheme on Jikes RVM to select and switch GC algorithms dynamically, based on annotations inserted into the bytecode of class files. The annotations include the minimum possible heap size of the switching points for each application, determined by some profiling runs. The studies conducted in our work has the potential to compliment the dynamic selection by offering cross-input prediction of the minimum possible heap size and possibly switching points.

Cross-Run Program Optimization There have been some explorations on cross-input program behavior prediction, mostly in the areas of locality studies on application written in traditional C/C++ languages.

In 1991, Wall [21] conducts a study to measure how well a profile from one run describe the behavior of a different run. He finds that there is a significant departure between profiles of the different runs, and using a perfect profile may do sometimes factors of better than using a profile of a different run.

Some of more recent studies measure the influence of data sets on program behavior for benchmark design [5, 10]. Ding and Zhong [9] describe an approach to predict program data locality across inputs. Arnold et al. [4] have proposed the repository-based approach for adaptive optimizations in JVM. Shen et al. show the cross-input predictability of locality phase sequence by representing a phase sequence in a regular expression [17]. Shen and Mao show the cross-input predictability of program basic block frequencies on some C/C++ programs [16]. They have recently proposed an evolvable scheme to tailor the optimizations in JVM to each input of an application [14].

Finally, the statistical analysis conducted in this paper is enlightened by Georges et al. [12]. The analysis has proved to be vital: It has corrected some conclusions we obtained merely from the average values of the running times.

7. Conclusions

This paper presents a set of experiments and analyses on uncovering the influence of program inputs on the selection of garbage collectors. The study draws the following conclusions:

- Inputs influence the relative performance of garbage collectors significantly, causing large variations of the top set of garbage collectors across inputs. Profiling one or few runs is typically insufficient for selecting the garbage collector that works for most inputs.
- But for most programs, when the heap size ratio is fixed, one top garbage collector may work best for over 80% inputs, and two would cover more than 96% inputs for all of the programs. In that scenario, profiling many runs on a sequence of different inputs and picking the best one can work reasonably well.
- The heap size ratio may affect the relative performance of garbage collectors significantly for some programs. It is therefore important to distinguish between heap-size-sensitive programs from the insensitive ones. For the former, profiling on one heap size should be enough; but for

the latter, it is necessary to have a cross-input predictive model that predicts the minimum possible heap size of the execution on an arbitrary input.

- Through regression techniques, it is possible to accurately predict minimum possible heap sizes across program inputs. Classification techniques may help determine the best garbage collector for an arbitrary input. The results suggest the promise of input-specific selection of garbage collectors.

8. Acknowledgment

We are especially grateful to Steve Blackburn for his insightful suggestions and great help to the final version of this paper. We thank the anonymous reviewers of VEE'09 for their invaluable comments. This material is based upon work supported by the National Science Foundation under Grant No. CSR-0720499 and CCF-0811791. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

References

[1] Java Grande benchmark. http://www2.epcc.ed.ac.uk/javagrande/.

[2] Spec jvm98. http://www.spec.org/jvm98/.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P.F. Sweeney. Adaptive optimization in the Jalapeno JVM. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Minneapolis, MN, October 2000.

[4] M. Arnold, A. Welc, and V.T. Rajan. Improving virtual machine performance using a cross-run profile repository. In *the Conference on Object-Oriented Systems, Languages, and Applications*, 2005.

[5] P. Berube and J. N. Amaral. Benchmark design for robust profile-directed optimization. In *Standard Performance Evaluation Corporation (SPEC) Workshop*, 2007.

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

[7] S. M. Blackburn, P. Cheng, and K. S. McKinley. Myths and realities: the performance impact of garbage collection. *SIGMETRICS Perform. Eval. Rev.*, 32(1), 2004.

[8] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2006.

[9] C. Ding and Y. Zhong. Predicting whole-program locality with reuse distance analysis. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–257, San Diego, CA, June 2003.

[10] L. Eeckhout, H. Vandierendonck, and K. D. Bosschere. Quantifying the impact of input data sets on program behavior and its applications. *Journal of Instruction-Level Parallelism*, pages 1–33, 2003.

[11] R. Fitzgerald and D. Tarditi. The case for profile-directed selection of garbage collection. In *Proceedings of the International Symposium on Memory Management*, 2000.

[12] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2007.

[13] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning*. Springer, 2001.

[14] F. Mao and X. Shen. Cross-input learning and discriminative prediction in evolvable virtual machine. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2009.

[15] 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*, 2001.

[16] X. Shen and F. Mao. Modeling relations between inputs and dynamic behavior for general programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2007.

[17] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. *Journal of Parallel and Distributed Computing*, 67(7), 2007.

[18] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *Proceedings of the International Symposium on Memory Management*, 2007.

[19] F. Smith and G. Morrisett. Comparing mostly-copying and mark-sweep conservative collection. In *Proceedings of the International Symposium on Memory Management*, 1998.

[20] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *Proceedings of the International Symposium on Memory Management*, 2004.

[21] D. Wall. Predicting program behavior using real or estimated profiles. In *Proceedings of PLDI*, Toronto,Canada, June 1991.

[22] B. Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Proceedings of ACM Conference on Lisp and Functional Programming*, 1990.