Java Performance Evaluation through Rigorous Replay Compilation

Andy Georges Lieven Eeckhout Dries Buytaert
Department of Electronics and Information Systems, Ghent University, Belgium
{ageorges,leeckhou}@elis.ugent.be, dries@buytaert.net

Abstract

A managed runtime environment, such as the Java virtual machine, is non-trivial to benchmark. Java performance is affected in various complex ways by the application and its input, as well as by the virtual machine (JIT optimizer, garbage collector, thread scheduler, etc.). In addition, non-determinism due to timer-based sampling for JIT optimization, thread scheduling, and various system effects further complicate the Java performance benchmarking process.

Replay compilation is a recently introduced Java performance analysis methodology that aims at controlling non-determinism to improve experimental repeatability. The key idea of replay compilation is to control the compilation load during experimentation by inducing a pre-recorded compilation plan at replay time. Replay compilation also enables teasing apart performance effects of the application versus the virtual machine.

This paper argues that in contrast to current practice which uses a single compilation plan at replay time, multiple compilation plans add statistical rigor to the replay compilation methodology. By doing so, replay compilation better accounts for the variability observed in compilation load across compilation plans. In addition, we propose matched-pair comparison for statistical data analysis. Matched-pair comparison considers the performance measurements per compilation plan before and after an innovation of interest as a pair, which enables limiting the number of compilation plans needed for accurate performance analysis compared to statistical analysis assuming unpaired measurements.

Categories and Subject Descriptors D.2.8 [*Software Engineering*]: Metrics—Performance measures; D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Experimentation, Measurement, Performance

Keywords Java, virtual machine, performance evaluation, benchmarking, replay compilation, matched-pair comparison

1. Introduction

Managed runtime systems, such as Java virtual machines, are challenging to benchmark because there are various factors affecting overall performance, such as the application and its input, the virtual machine (VM) and its configuration (memory management strategy, heap size, dynamic optimizer, thread scheduler, etc.), and also the hardware on which the benchmarking experiment is done. To complicate things even further, non-determinism leads to different performance results when running the same experiment multiple times. An important source of non-determinism comes from timer-based sampling for selecting candidate methods for JIT compilation and optimization. Timer-based sampling may result in different samples being taken across multiple runs of the same experiment, which may lead to different methods being optimized, and which in its turn results in execution time variability. Although some VMs use method invocation counters to find optimization candidates [13, 28], most VMs use timer-based sampling [1, 2, 11, 25, 34, 36]. In particular, VMs that use multiple levels of optimization rely exclusively on sampling for identifying optimization candidates because of the high overhead invocation counters introduce in optimized code.

Researchers are well aware of the difficulty in understanding and benchmarking managed runtime system performance. This is reflected in the growing body of work on Java performance analysis, see for example [3, 16, 18, 24, 35]. Some recent work focuses on experimental design [3, 5, 14], i.e., choosing an appropriate set of benchmarks and inputs, VMs, garbage collectors, heap sizes, and hardware platforms; other recent work focuses on data analysis [15], i.e., how to analyze and report the performance results in the presence of non-determinism.

One particularly interesting and increasingly widely used experimental design methodology is *replay compilation* [19, 29]. Replay compilation fixes the compilation load through a so called *compilation plan* which is determined from a profile run. The compilation plan then forces the VM to compile each method to a predetermined optimization level in the replay run. By doing so, replay compilation eliminates the non-determinism due to timer-based JIT optimization. This facilitates performance analysis.

Current practice in replay compilation uses a single compilation plan during replay. In this paper, we argue that the performance results obtained for one compilation plan may not be representative for other compilation plans, and may potentially lead to misleading results in practical research studies. We therefore advocate multiple compilation plans in order to better represent average behavior. We propose a matched-pair comparison as the statistically rigorous data analysis method when comparing design alternatives under replay compilation using multiple compilation plans. A matched-pair comparison amortizes part of the overhead introduced by multiple compilation plans.

In this paper, we make the following contributions:

- We show that different compilation plans lead to statistically significant execution time variability. The reason is that different compilation plans may lead to different methods being compiled to different optimization levels. And this execution time variability may lead to inconsistent conclusions across compilation plans in practical research studies.
- We advocate replay compilation using multiple compilation plans in order to capture the execution time variability across compilation plans. Multiple compilation plans result in a more rigorous replay compilation methodology compared to prior work which considers a single compilation plan during replay.
- We propose matched-pair comparison for analyzing the performance numbers obtained from replay compilation using multiple compilation plans. Matched-pair comparison considers the performance numbers for a given compilation plan before and after the innovation as a pair. In general, this yields tighter confidence intervals than statistical analysis assuming unpaired measurements. Or, for the same level of accuracy, i.e., for the same confidence interval size, fewer compilation plans are to be considered under matched-pair comparison.
- We demonstrate that for a given experimentation time budget, it is beneficial to consider more compilation plans rather than more runs per compilation plan.

Although the experimental results in this paper are obtained using Jikes RVM [1], we believe that the overall conclusions from this paper generalize to other VMs that use timer-based sampling for driving just-in-time compilation and optimization. In addition, we believe that these conclusions also generalize to VMs that use invocation counters when running multithreaded benchmarks; different thread schedules may lead to different methods being optimized to different levels of optimization at different points in time. Multiple compilation plans will capture these differences.

This paper is organized as follows. In Section 2, we first describe replay compilation as an experimental design methodology for managed runtime systems. Section 3 explains the experimental setup used in this paper. In Section 4 we evaluate the runtime variability across compilation plans at replay time, and how this may affect conclusions in practical research studies. Section 5 presents matched-pair comparison for the statistical analysis of performance numbers obtained from multiple compilation plans. In Section 6 we describe the overall framework for rigorous replay compilation using multiple compilation plans. We finally conclude in Section 7.

2. Replay Compilation

Replay compilation [19, 29] is a recently introduced experimental design methodology that fixes the compilation/optimization load in a Java virtual machine execution. As mentioned before, the motivation is to control non-determinism, and by doing so to facilitate performance analysis.

2.1 Replay compilation mechanism

Replay compilation requires a *profiler* and a *replayer*. The profiler, see Figure 1, records the profiling information used to drive the compilation decisions, e.g., edge counts, path and dynamic call graph information, etc., as well as the compilation decisions, e.g., method X was compiled at optimization level Y. Typically, researchers run multiple experiments yielding multiple profiles (p profiles in Figure 1), and a single *compilation plan* is determined from these profiles. The replayer then reads the compilation plan and upon the first invocation of a method, it induces the compilation plan by optimizing the method to the optimization level specified in the compilation plan. By doing so, the compilation/optimization load is fixed which forms the basis of comparison when evaluating the efficacy of an innovation of interest.

During the replay phase, a benchmark is iterated twice within a single VM invocation. The first iteration includes the compilation overhead according to the compilation plan — this is referred to as the *mix* run. The second iteration then is the timing run with adaptive (re)compilation turned off — this is called the *stable* run. In order to fix the garbage collection load in the stable run, a full-heap garbage collection is typically done between the mix and the stable runs.

2.2 Design options

Researchers typically select a single compilation plan out of a number of profiles, or, alternatively, combine these profiles into a single compilation plan. Some researchers pick the best profile as the compilation plan, i.e., the profile that

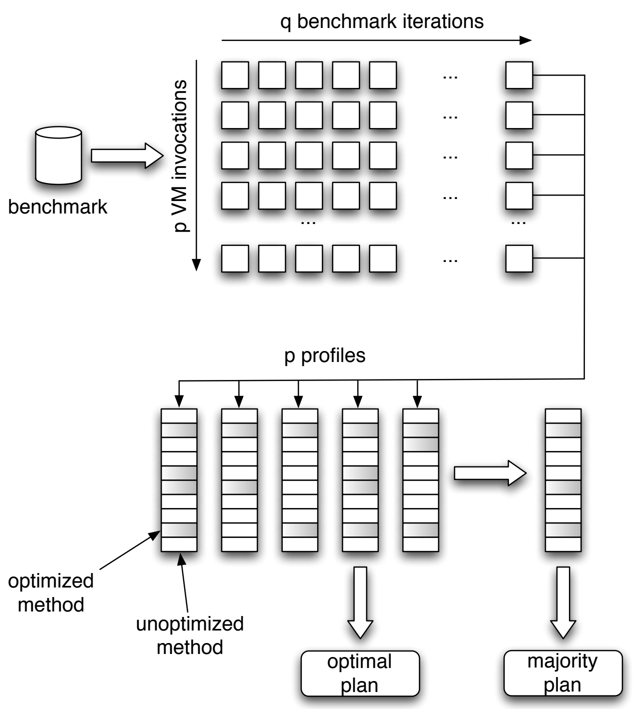


Figure 1: The profiling phase for replay compilation. There are p VM invocations, and each VM invocation runs q benchmark iterations, yielding p profiles. Current practice then selects a single compilation plan from these profiles: the optimal plan (yielding the fastest execution time), or the majority plan (combining all profiles through a majority vote).

yields the best overall performance, see for example [3, 6, 7, 10] — this is called the *optimal plan*. The motivation for using an optimal plan is to assess an innovation on a compilation plan that represents the best code quality. Others select the median optimization level observed across the profiles for each method [37]. Yet others select the methods that are optimized in the majority of the profiles, and set the optimization level for the selected methods at the highest optimization levels observed in the majority of the profiles, see for example [12, 19, 30, 29] — this is called the *majority plan*. And yet others select the methods observed in the intersection of multiple profiles [32].

Another design option relates to how many benchmark iterations (represented by q in Figure 1) to consider when collecting the profiles. As the benchmark is iterated multiple times without restarting the VM, more and more methods will be selected for JIT optimization, i.e., the code quality will steadily improve as more and more methods get optimized to higher levels of optimization. The question then is when to collect the profile across these iterations. One option could be to finish the profile collection after the first benchmark iteration, i.e., $q = 1$. Another option would be to collect the profile across multiple benchmark iterations, i.e., $q > 1$; this will result in a profile that represents better code quality. In this paper, we consider both options, using *1-iteration* ($q = 1$) and *10-iteration* ($q = 10$) compilation plans.

The final design option is how to configure the system setup (virtual machine configuration, garbage collection strategy, heap size, etc.) when collecting the profiles.

2.3 Discussion

A single compilation plan. Current practice in replay compilation considers a single compilation plan during replay. As we will show in this paper, this can be misleading. The reason is that a single compilation plan does not account for the variability observed in compilation load across multiple runs under non-deterministic VM executions. By consequence, a single compilation plan may not be representative for the average behavior seen by an end user. We therefore advocate using multiple compilation plans at replay time. This is consistent with our prior work [15] on using statistical data analysis for coping with non-determinism, which advocates using an average performance number along with a confidence interval computed from a number of benchmarking experiments instead of picking a performance number from a single experiment.

Non-determinism. Replay compilation, although it controls non-determinism to a large extent, does not completely eliminate non-determinism. There are a number of remaining sources of non-determinism that replay compilation does not control, e.g., thread scheduling. Different thread scheduling decisions in time-shared and multi-threading environments across different runs of the same experiment can affect performance. For example, different thread schedules may lead to different points in time where garbage is being collected leading to different data layouts which may affect memory system performance as well as overall performance. Also, various system effects, such as interrupts, introduce non-determinism when run on real hardware. To cope with the non-determinism, in our prior work [15], we recommended applying statistical analysis by computing the average performance number as well as confidence intervals across multiple experiments. By controlling non-determinism, replay compilation reduces the required number of measurements to reach statistically valid conclusions.

Replay compilation as experimental design. Replay compilation is an experimental design setup that may be appropriate for some experiments but inappropriate for others. It's up to the experimenter who has a good understanding of the system under measurement to determine whether replay compilation is an appropriate experimental design setup. Specifically, the implicit assumption for replay compilation is that the innovation under evaluation does not affect compilation decisions, i.e., the compiler/optimizer is assumed to make the same compilation/optimization decisions irrespective of the innovation under evaluation. This may or may not be a valid assumption depending on the experiment at hand.

2.4 Use-case scenarios

There are several use-case scenarios for which replay compilation is a useful experimental design setup. We enumerate a couple examples here as they are in use today — this enumeration illustrates the wide use of replay compilation as an experimental design setup for managed runtime systems.

JIT innovation. JIT research, such as compiler/optimizer innovation, may benefit from replay compilation as an experimental setup. Researchers evaluating the efficacy of a JIT innovation want to answer questions such as 'How does my innovation improve application code quality?' 'What is the compilation time overhead that the innovation incurs?' The problem at hand is that in a virtual execution environment with dynamic compilation, application code execution and compilation overhead are intermingled. The question then is how to tease apart the effect that the JIT innovation has on code quality and compilation time?

Replay compilation is a methodology that enables teasing apart code quality and compile time overhead, see for example Cavazos and Moss [12], who study compiler scheduling heuristics. The mix run provides a way of quantifying the overhead the innovation has on compilation time. The stable run provides a way of quantifying the effect of the innovation on code quality.

Innovation in profiling and JVM innovation A research topic that is related to JIT innovation is profiling, i.e., an improved profiling mechanism provides a more accurate picture for analysis and optimization. For example, Bond and McKinley [9] use replay compilation to gauge the overhead of continuously maintaining a probabilistic calling context in a Java virtual machine; the same research group uses similar setups in [8] and [10]. Because the compilation load and the resulting code quality is fixed, the stable run can be used for performance and overhead measurement. Similarly, Schneider et al. [31] optimize spatial locality by co-allocating objects in a generational garbage collector based on measurements obtained from hardware performance monitors that count cache misses. They use replay compilation to minimize the variability across multiple runs during measurement.

GC innovation. Garbage collection (GC) research also benefits from replay compilation. In fact, many recent garbage collection research papers use replay compilation as their experimental design methodology [3, 6, 7, 17, 19, 29, 30, 32, 37]. The reason why replay compilation is useful for garbage collection research is that it fixes the compiler load, and by doing so, it controls non-determinism which facilitates the comparison of garbage collection alternatives.

GC research often uses the optimal plan under replay compilation. The motivation for doing so is that if a GC strategy degrades mutator locality, this is likely to be exposed more by a compilation plan that represents higher code quality. Although an optimal plan is an important experimental design choice for GC research for this reason, it may not accurately represent user-perceived performance.

Other applications. There exist a number of other applications to replay compilation. Krintz and Calder [21] for example annotate methods with analysis information collected offline, similar to a compilation plan. These annotations significantly reduce the time to perform dynamic optimizations. Ogata et al. [27] use replay compilation to facilitate the debugging of JIT compilers.

3. Experimental Setup

Before studying replay compilation in great detail, we first describe our experimental setup. We discuss the virtual machine configurations, the benchmarks and the hardware platforms considered in this paper. Finally, we also detail the replay compilation setup.

3.1 Virtual machine configuration

We use the Jikes Research Virtual Machine (RVM) [1] which is an open source Java virtual machine written in Java. Jikes RVM employs baseline compilation to compile a method upon its first execution; hot methods are sampled by an OS-triggered sampling mechanism and subsequently scheduled for further optimization. There are three optimization levels in Jikes RVM: 0, 1 and 2. We use the February 12, 2007 SVN version of Jikes RVM in all of our experiments.

As Jikes RVM employs timer-based sampling to detect optimization candidates, researchers have implemented replay compilation in Jikes RVM to control non-determinism using so called advice files — an advice file is a compilation plan in this paper's terminology. An *advice file* specifies (i) the optimization level for each method compiled, (ii) the dynamic call graph profile, and (iii) the edge profile. Advice files are collected through a profile run: through command-line arguments, Jikes RVM can be instructed to generate an advice file just before program execution terminates. Then, in the replay run, Jikes RVM compiles each method in the advice file to the specified level upon a method's first invocation. If there is no advice for a method, the method is compiled using Jikes RVM's baseline compiler.

In some of our experiments we will be considering multiple garbage collection strategies across a range of heap sizes. This is similar in setup to what garbage collection research papers are doing — as mentioned before, many of these garbage collection papers also employ replay compilation as their experimental design. We consider five garbage collection strategies in total, all provided by the Jikes' Memory Management Toolkit (MMTk) [4]. The five garbage collection strategies are: (i) CopyMS, (ii) GenCopy, (iii) GenMS, (iv) MarkSweep, and (v) SemiSpace; the generational collectors use a variable-size nursery. We did not include the GenRC, MarkCompact and RefCount collectors from MMTk, because we were unable to successfully run Jikes with the GenRC and MarkCompact collectors for

benchmark	description	min heap size (MB)
compress	file compression	24
jess	puzzle solving	32
db	database	32
javac	Java compiler	32
mpegaudio	MPEG decompression	16
mtrt	raytracing	32
jack	parsing	24
antlr	parsing	32
bloat	Java bytecode optimization	56
fop	PDF generation from XSL-FO	56
hsqldb	database	176
jython	Python interpreter	72
luindex	document indexing	32
pmd	Java class analysis	64

Table 1: SPECjvm98 (top seven) and DaCapo (bottom seven) benchmarks considered in this paper. The rightmost column indicates the minimum heap size, as a multiple of 8MB, for which all GC strategies run to completion.

some of the benchmarks; and RefCount did yield performance numbers that are statistically significantly worse than any other GC strategy across all benchmarks.

3.2 Benchmarks

Table 1 shows the benchmarks used in this study. We use the SPECjvm98 benchmarks [33] (first seven rows), as well as seven DaCapo benchmarks [5] (next seven rows). SPECjvm98 is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmarks with the largest input set (`-s100`). The DaCapo benchmark suite is a recently introduced open-source benchmark suite; we use release version 2006-10-MR2. We use the seven benchmarks that execute properly on the February 12, 2007 SVN version of Jikes RVM. We use the default (medium size) input set for the DaCapo benchmarks.

In all of our experiments, we consider a per-benchmark heap size range, following Blackburn et al. [3]. We vary the heap size from a minimum heap size up to 6 times this minimum heap size, using increments of the minimum heap size. The per-benchmark minimum heap sizes are shown in Table 1.

3.3 Hardware platforms

Following the advice by Blackburn et al. [5], we consider multiple hardware platforms in our performance evaluation methodology: a 2.1GHz AMD Athlon XP, and a 2.8GHz Intel Pentium 4. Both machines have 2GB of main memory. These machines run the Linux operating system, version 2.6.18. In all of our experiments we consider an otherwise idle and unloaded machine.

3.4 Replay compilation setup

The compilation plans are computed by running a benchmark on the Jikes RVM using the GenMS garbage collector and a heap size that is 8 times the minimum heap size. In this paper, we compute profiles after (i) a single iteration of the benchmark within a single VM invocation (yielding *1-iteration* plans), and (ii) after 10 iterations of the benchmark within a single VM invocation (yielding *10-iteration* plans). We compute separate compilation plans per hardware platform. We perform a full GC between the mix and stable runs.

4. The Case for Multiple Compilation Plans

Having detailed our experimental setup, we now study the accuracy of selecting a single compilation plan for driving replay compilation. This study will make the case for multiple compilation plans instead of a single compilation plan.

This is done in four steps. We first demonstrate that different compilation plans can lead to statistically significantly different benchmark execution times. This is the case for both GC time and mutator time. Second, we provide a reason for this difference in execution time, by comparing the methods that are compiled under different compilation plans. Third, we present a case study in which we compare various garbage collection strategies using replay compilation as the experimental design setup. This case study demonstrates that the conclusions taken from practical research studies may be subject to the chosen compilation plan. Finally, we demonstrate that a majority plan (which combines multiple profiles into a single compilation plan) is no substitute for multiple compilation plans.

However, before doing so, we first need to explain the ANOVA statistical data analysis method which we will use throughout this section.

4.1 Statistical background: ANOVA

Single-factor ANOVA. *Analysis of Variance (ANOVA)* [20, 23, 26] separates the total variation in a set of measurements into a component due to random fluctuations in the measurements versus a component due to the actual differences among the alternatives. In other words, ANOVA separates the total variation observed in (i) the variation observed *between* each alternative, and (ii) the variation *within* the alternatives, which is assumed to be a result of random effects in the measurements. If the variation between the alternatives is larger than the variation within each alternative, then we conclude that there is a statistically significant difference between the alternatives; if not, the variability across the measurements is attributed to random effects.

Although we will not describe the ANOVA mechanics in great detail — we refer to a textbook on statistics for a detailed description, see for example [20, 23, 26] — we rather explain the outcome of the ANOVA. A single-factor ANOVA splits up the total variation, sum-of-squares total (SST), observed across all measurements into a term, sum-of-squares due to the alternatives (SSA), that quantifies the variation between the alternatives, versus a term, sum-of-squares due to measurement errors (SSE), that quantifies the variation within an alternative due to random errors. Mathematically, this means that

$$SST = SSA + SSE.$$

The goal now is to quantify whether the variation SSA across alternatives is 'larger' in some statistical sense than the variation SSE within each alternative. A simple way for doing so is to compare the fractions SSA/SST and SSE/SST. A statistically more rigorous approach is to apply a statistical test, called the *F-test*, which is used to test whether two variances are significantly different. The *F statistic* is computed as

$$F = \frac{s_a^2}{s_e^2},$$

with

$$s_a^2 = \frac{SSA}{k-1}$$

and

$$s_e^2 = \frac{SSE}{k(n-1)}$$

with k alternatives, and n measurements per alternative. If the F statistic is larger than the critical value $F_{[1-\alpha;(k-1),k(n-1)]}$, which is to be obtained from a precomputed table, we can say that the variation due to differences among the alternatives is significantly larger than the variation due to random measurement noise at the $(1-\alpha)$ confidence level. A $(1-\alpha)$ confidence interval, i.e., a confidence interval with a $(1-\alpha)$ confidence level, means that there is a $(1-\alpha)$ probability that the variation due to differences between the alternatives is larger than the variation due to random noise. In this paper we will consider 95% confidence intervals.

ANOVA assumes that the variance in measurement error is the same for all the alternatives. Also, ANOVA assumes that the errors in the measurements for the different alternatives are independent and Gaussian distributed. However, ANOVA is fairly robust with respect to non-Gaussian distributed measurements, especially in case there is a balanced number of measurements for each of the alternatives. This is the case in our experiments: we have the same number of measurements per alternative; we can thus use ANOVA for our purpose.

Two-factor ANOVA. The ANOVA discussed above is a so called single-factor ANOVA, i.e., a single input variable (factor) is varied in the experiment. A two-factor ANOVA on the other hand allows for studying two input variables and their mutual interaction. In practicular, a two-factor ANOVA splits up the total variation between terms that quantify the variation for each of the two factors, i.e., SSA and SSB, as well as an interaction term between the factors, i.e., SSAB, and a term that quantifies the variation within each combination of factors due to random errors, i.e., SSE. Similarly to the above formulas, one can use the *F-test* to verify the presence of effects caused by either factor and the interaction between the factors.

Post-hoc test. After completing an ANOVA test, we may conclude that there is a statistically significant difference between the alternatives, however, the ANOVA test does not tell us between which alternatives there is a statistically significant difference. There exists a number of techniques to find out between which alternatives there is or there is not a statistically significant difference. One approach, which we will be using in this paper, is called the Tukey HSD (Honestly Significantly Different) test. The advantage of the Tukey HSD test over simpler approaches, such as pairwise Student's t-tests, is that it limits the probability of making an incorrect conclusion in case there is no statistically significant difference between the alternatives and in case most of the alternatives are equal but only a few are different. For a more detailed discussion, we refer to the specialized literature [20, 23, 26].

4.2 Execution time variability

We first study how benchmark execution time is affected by the compilation plan under replay compilation. To do so, we consider the following experiment. We collect 10 compilation plans per benchmark, and run each benchmark 10 times for each compilation plan. This yields 100 execution times in total per benchmark. This measurement procedure is done for both 1-iteration and 10-iteration plans, for 5 GC strategies and for 6 heap sizes. The goal of this experiment now is to quantify whether the execution time variability observed across these 100 measurements is determined more by the compilation plans than by the runtime variability per compilation plan.

Example. Figure 2 illustrates this experiment for a typical benchmark, namely jython — we observed similar results for other benchmarks. Violin plots are displayed which show the GC time and mutator time variability within a compilation plan (on the vertical axis). By comparing violin plots across compilation plans (on the horizontal axis) we get insight in how compilation plans affect execution time. The middle point in a violin plot shows the median, and the shape of the violin plot represents the distribution's probability density function: the wider the violin plot, the higher the density. The top and bottom points show the maximum and minimum values. This figure suggests that the *variability within a compilation plan is much smaller than the variability across compilation plans*.

Rigorous analysis. To study the execution time variability across compilation plans in a more statistically rigorous

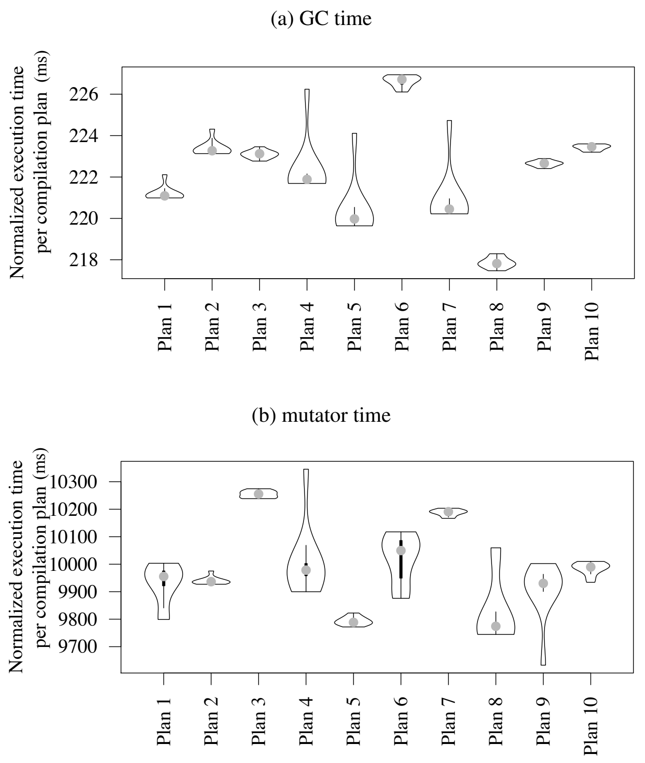


Figure 2: Violin plots illustrating the variability in (a) GC time and (b) mutator time within and across compilation plans for jython on the AMD Athlon, the GenMS garbage collector, and a 144 MB heap size; assuming a stable run and a 10-iteration compilation plan. Time is measured in milliseconds, and the difference between the highest and lowest value for GC and mutator time is 3.6% and 6.2%, respectively.

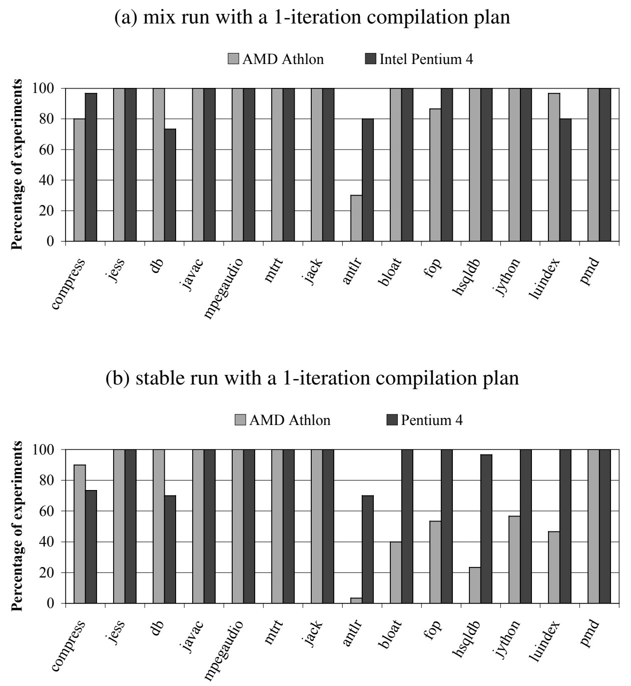


Figure 3: The fraction of experiments for which there is a statistically significant difference in total execution time across the ten 1-iteration compilation plans on the AMD Athlon XP and the Intel Pentium 4 platforms. The top and bottom graphs show results for mix and stable replay, respectively.

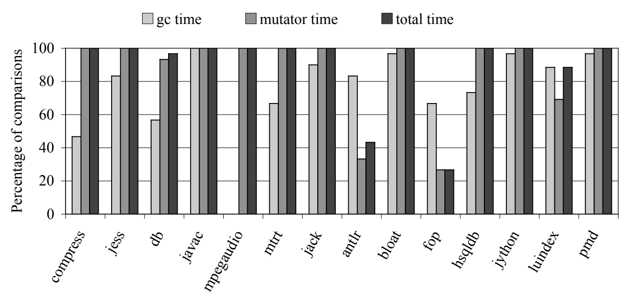
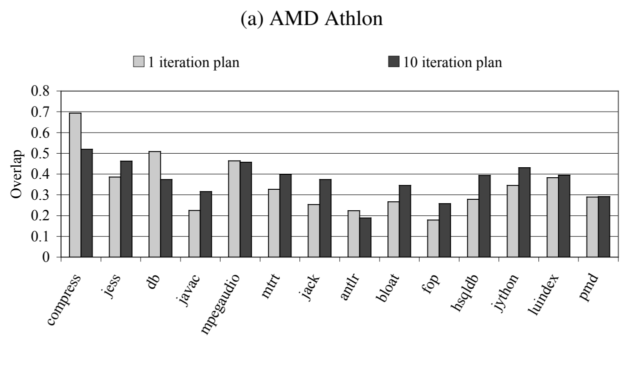


Figure 4: The fraction of experiments for which there is a statistically significant difference in GC, mutator and total time across compilation plans. These graphs assume 10-iteration plans and stable runs on the AMD Athlon platform.

manner, we now use a single-factor ANOVA in which the compilation plans are the alternatives. In other words, the ANOVA will figure out whether the execution time variability across these 100 measurements is due to random effects rather than due to the compilation plans.

Figure 3 shows the percentage of the 30 experiments per benchmark (there are 5 GC strategies and 6 heap sizes) for which the ANOVA reports there is a statistically significant difference in total execution time at the 95% confidence level between the various compilation plans for a 1-iteration plan. The top graph in Figure 3 shows the mix run results, whereas the bottom graph shows the stable run results; there are two bars per benchmark for the Intel Pentium 4 and AMD Athlon machines, respectively. For the majority of the benchmarks, there is a statistically significant difference in execution times across multiple compilation plans. For several benchmarks, the score equals 100% which means that the execution times are significantly different across all compilation plans. The difference tends to be higher for the mix runs than for the stable runs for most of the DaCapo bench-



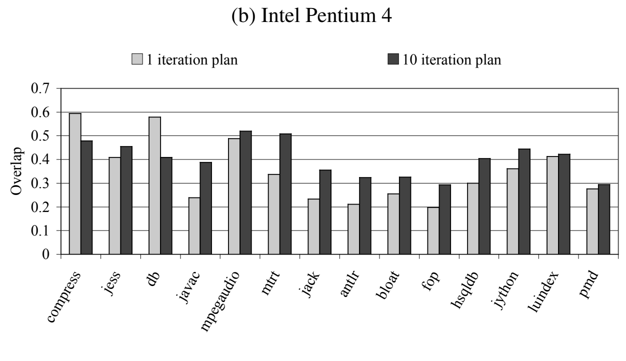


Figure 5: Average overlap across compilation plans on (a) the AMD Athlon platform, and (b) the Intel Pentium 4 platform, for the 1-iteration and 10-iteration compilation plans.

marks on the AMD platform. This suggests that performance seems to be more similar across compilation plans in the stable run.

Figure 4 shows similar results for the 10-iteration compilation plans, but now we make a distinction between GC, mutator and total time, and we assume the stable run. (Although Figure 4 only shows results for the AMD Athlon, we obtained similar results for the Intel Pentium 4.) We conclude that even under 10-iteration compilation plans there still is a large fraction of experiments for which we observe *statistically significant execution time differences across compilation plans*. And this is the case for GC, mutator and total time.

4.3 Compilation load variability

Now that we have shown that different compilation plans can result in statistically significantly different execution times, this section quantifies why this is the case. Our intuition tells us that the varying execution times are due to different methods being compiled at different levels of optimization across compilation plans. To support this hypothesis we quantify the relative difference in compilation plans.

To do so, we determine the Method Optimization Vector (MOV) per compilation plan. Each entry in the MOV represents an optimized method along with its (highest) optimization level; the MOV does not include an entry for baseline compiled methods. For example, if in one compilation plan, the method `foo_1` gets optimized to level 1, method `foo_2` gets optimized to level 0, and method `foo_3` gets only baseline compiled, then the MOV looks like `[(foo_1,1);(foo_2,0)]`. In another compilation plan, method `foo_1` gets optimized to optimization level 1 as well, whereas method `foo_2` gets baseline compiled, and `foo_3` gets optimized to level 0, then the MOV looks like `[(foo_1,1);(foo_3,0)]`. Comparing the two compilation plans can then be done by comparing their respective MOVs. This is done by counting the number of (method, optimization level) pairs that appear in both MOVs, divided by the total number of methods appearing in both compilation plans. In the above example, the overlap metric equals 1/3, i.e., there is one common (method, optimization level) pair that appears in both MOVs, namely `(foo_1,1)` and there are three methods optimized in at least one of the compilation plans. An overlap metric of one thus represents a perfect match, and a zero overlap metric represents no match.

Figure 5 quantifies the overlap metric per benchmark computed as an average across all (unique) pairs of 10 compilation plans — there are $C_{10}^2 = 45$ unique pairs of compilation plans over which the average overlap metric is computed. We observe that the overlap is rather limited, typically under 0.4 for most of the benchmarks. There are a couple benchmarks with relatively higher overlap metrics, see for example compress and db. These benchmarks have a small code footprint and therefore there is a higher probability that the same methods will get sampled across multiple profiling runs of the same benchmark. We conclude that the *significant performance differences across compilation plans are due to compilation load differences*.

4.4 Case study: Comparing GC strategies

We now study whether different compilation plans can lead to different conclusions in practical research studies. In order to do so, we consider a case study that compares GC strategies using replay compilation as the experimental design — this reflects a widely used methodology in GC research, see for example [3, 6, 7, 17, 19, 29, 30, 32, 37]. GC poses a complex space-time trade-off, and it is unclear which GC strategy is the winner without detailed experimentation.

We use the same data set as before. There are 14 benchmarks (7 SPECjvm98 benchmarks and 7 DaCapo benchmarks), and we consider 5 GC strategies and 6 heap sizes per benchmark. For each benchmark, GC strategy and heap size combination, we have 10 measurements per compilation plan for both the mix and stable runs; and we consider 1-iteration and 10-iteration plans. We then compute the average execution time along with its 95% confidence interval across these 10 measurements, following the statistically rigorous methodology described in our prior work [15] — specifically, we use ANOVA in conjunction with the Tukey HSD test to compute the simultaneous 95% confidence intervals. This yields the average execution time along with its confidence interval per GC strategy and heap size, for each

		compilation plan i		
		H_0^i is not rejected	H_0^i is rejected, $A > B$	H_0^i is rejected, $B > A$
	H_0^j is not rejected	*agree*	*inconclusive*	*inconclusive*
compilation plan j	H_0^j is rejected, $A > B$	*inconclusive*	*agree*	*disagree*
	H_0^j is rejected, $B > A$	*inconclusive*	*disagree*	*agree*

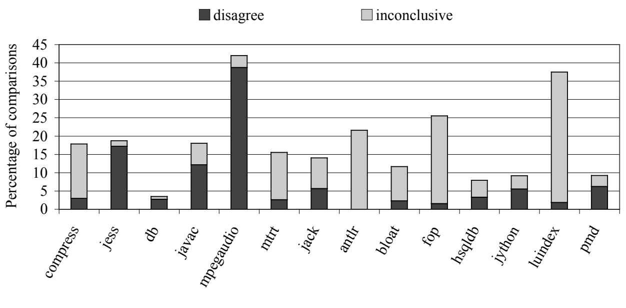
Table 2: Classifying pairwise GC comparisons when comparing compilation plans; A and B denote GC strategies, and $A > B$ means A outperforms B.

benchmark and compilation plan. We then compare these averages and confidence intervals by doing a pairwise comparison across compilation plans. The goal of this comparison is to verify whether different compilation plans lead to consistent conclusions about the best GC strategy for a given heap size and benchmark.

When comparing two compilation plans, we compare the execution times per pair of GC strategies (per heap size) and classify this comparison in one of the three categories: *agree*, *disagree* and *inconclusive*, see also Table 2. For a given compilation plan p_i and GC strategies A and B, we define the null hypothesis as $H_0^i \equiv \mu_{p_i}^A = \mu_{p_i}^B$. The null hypothesis states that GC strategies A and B achieve the same mean execution time under compilation plan p_i. Hence, if the null hypotheses H_0^i and H_0^j for compilation plans p_i and p_j are rejected, and if in both cases the same GC strategy outperforms the other, then the comparison is classified as an *agree*. This means that both compilation plans agree on the fact that GC strategy A outperforms GC strategy B (or vice versa) in a statistically significant way. In case both compilation plans yield the result that both GC strategies are statistically indifferent, i.e., for neither compilation plan the null hypothesis is rejected, we also classify the GC comparison as an *agree*. If on the other hand both compilation plans disagree on which GC strategy outperforms the other one, then we classify the comparison as *disagree*. In case the null hypothesis is rejected for one compilation plan, but not for the other, we classify the GC comparison as *inconclusive*, i.e., there is no basis for a statistically valid conclusion.

1-iteration plans. Figure 6 shows this classification per benchmark for the total execution time under mix and stable replay for 1-iteration compilation plans. The *disagree* and *inconclusive* categories are shown as a percentage — the *agree* category then is the complement to 100%. For several benchmarks, the fraction disagree comparisons is higher than 5%, and in some cases even higher than 10%. The mpegaudio benchmark is a special case with a very high disagree fraction although it has a very small live data footprint: the reason is that the various GC strategies affect performance through their heap data layout — see also later for a more rigorous analysis. For many benchmarks, the fraction inconclusive comparisons is larger than 10%, for both the mix and stable runs, and up to 20% and higher for several benchmarks. In other words, in a significant amount of cases, *different compilation plans do not agree on which GC strategy performs best.*

(a) mix run



(b) stable run

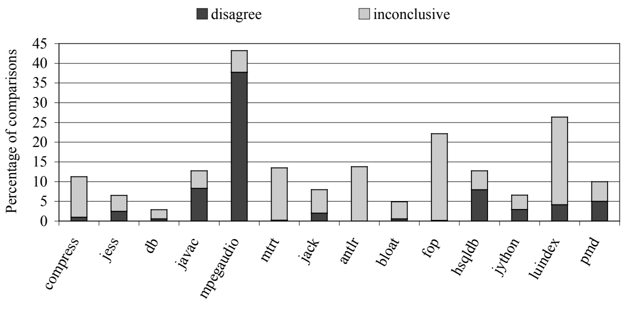


Figure 6: Percentage inconclusive and disagreeing comparisons on the AMD Athlon using 1-iteration compilations plans, under (a) mix replay and (b) stable replay.

10-iteration plans. Figures 7 and 8 show the percentage of inconclusive and disagreeing comparisons for GC time and mutator time, respectively, assuming stable replay and 10-iteration compilation plans. Although compilation plans mostly agree on the best GC strategy in terms of GC time (see Figure 7) — for some benchmarks, such as jess, bloat, fop and mpegaudio, all compilation plans agree — this is not the case for all benchmarks, see for example antlr and the 64MB to 96MB heap size range. In contrast to the large fraction of agrees in terms of GC time, this is not the case for mutator time, see Figure 8. For some benchmarks, the fraction disagrees and inconclusives can be as large as 13%

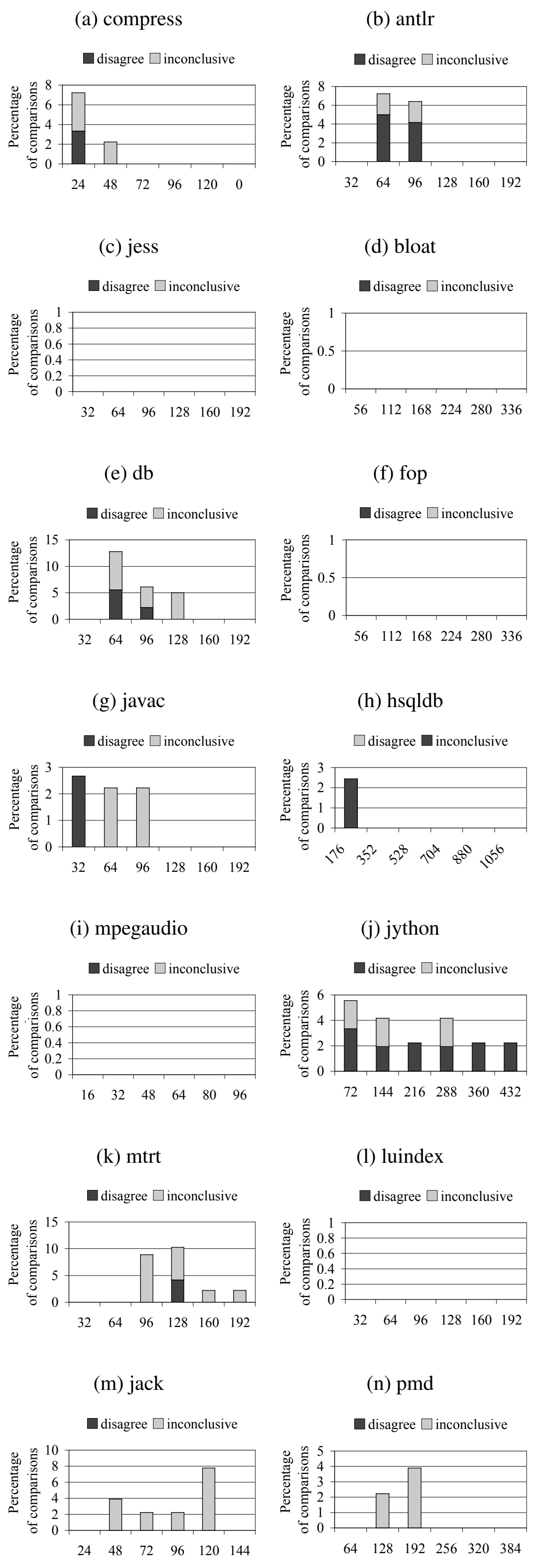


Figure 7: Percentage inconclusive and disagreeing comparisons for *GC time* under stable replay; heap size appears on the horizontal axis in each of the per-benchmark graphs.

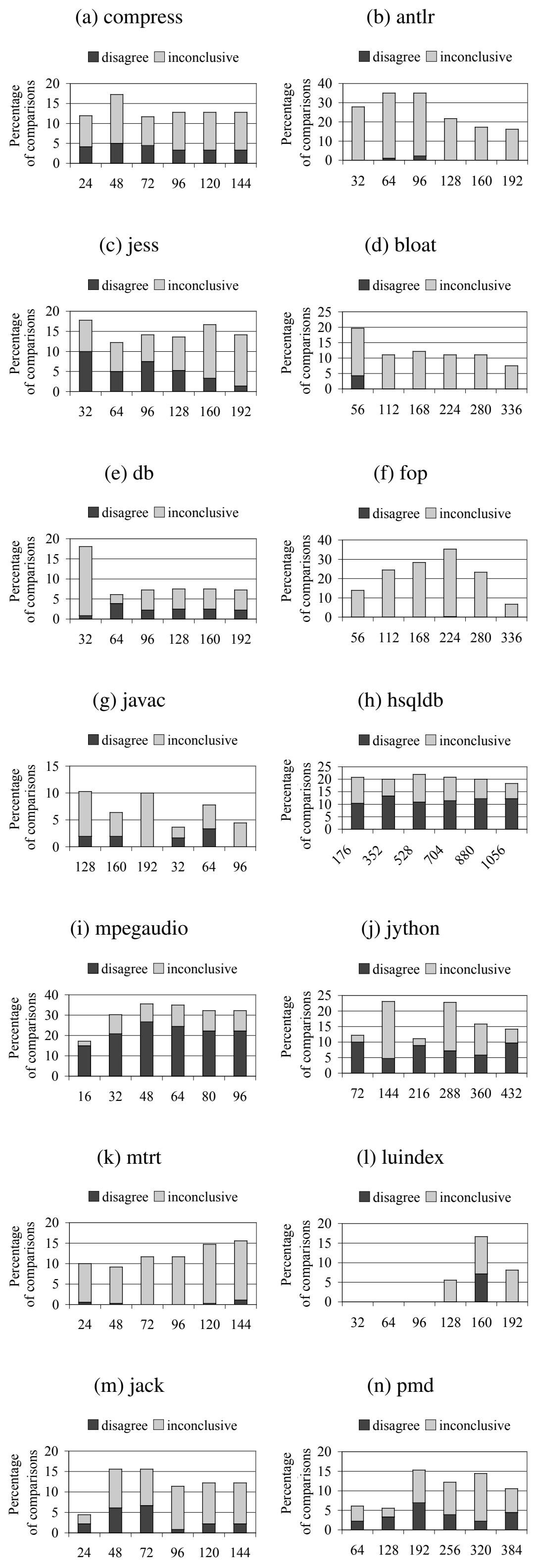


Figure 8: Percentage inconclusive and disagreeing comparisons for *mutator time* under stable replay; heap size appears on the horizontal axis in each of the per-benchmark graphs.

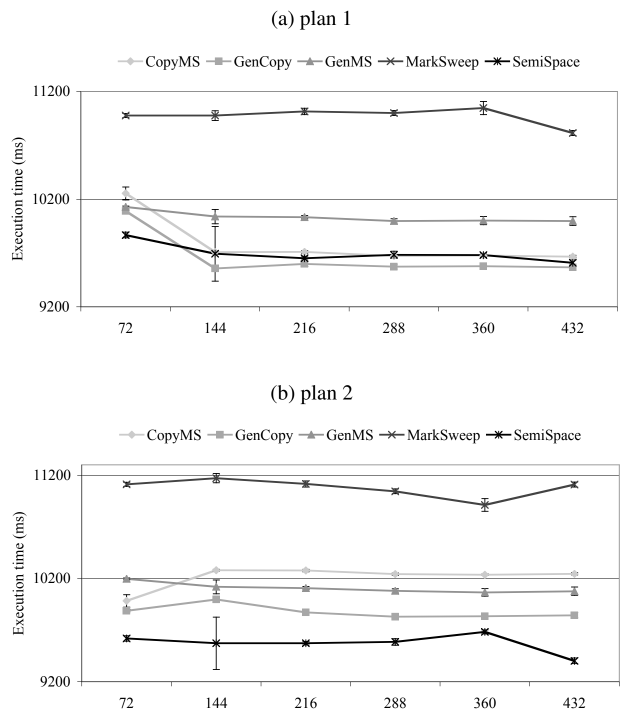


Figure 9: Comparison between the mutator execution times for *jython* using two different 10-iteration compilation plans as a function of the heap size for five garbage collectors. We show the mean of 10 measurements for each plan and the 95% confidence intervals.

(hsqldb) and 35% (fop), respectively. (Again, mpegaudio is a special case for the same reason as above.)

To further illustrate the differences across compilation plans, we now compare the mutator execution times of *jython* for each of the five garbage collectors for two different compilations plans obtained after running the benchmark for 10 iterations. Figure 9 shows that for the first plan, there is no clear winner given that there are only small performance difference between CopyMS, GenCopy and SemiSpace. However, the second plan shows a very different picture, in which SemiSpace is the best collector by more than 3% for some heap sizes.

Analyzing mutator time. The high fraction disagrees and inconclusives for mutator time in the above experiment raises an important question: is this observation a result of the effect that the GC strategy has on the data layout of the mutator, or in other words, is the GC strategy one of the main contributors to the high fraction disagrees and inconclusives? Or, is this observation simply a result of the performance variability observed across compilation plans and does the GC strategy not affect mutator time?

To answer this question we employ a two-factor ANOVA with the two factors being the GC strategy and the compilation plan, respectively. The two-factor ANOVA then reports whether the variability in mutator time is due to the GC strategy, the compilation plan, their mutual interaction, or random noise in the measurements. Figure 10 shows the percentage of the total variability in mutator time under stable replay that is accounted for by the garbage collector (SSA), the compilation plan (SSB), their interaction (SSAB), and the residual variability (SSE) due to random measurement noise. For almost all benchmarks, the garbage collector has a significant impact on mutator time. The same is true for both the compilation plans and the interaction between these two factors. For all benchmarks, except for mtrt, garbage collection accounts for over 15% of the observed variability in these experiments, and for many benchmarks, GC accounts for more than 60% of the total variability. Remarkably, the GC strategy affects mutator time quite a lot for mpegaudio, accounting for over 50% of the observed variability, even though no time is spent in GC during the stable run. This explains the earlier finding for mpegaudio: its performance is very sensitive to the data layout.

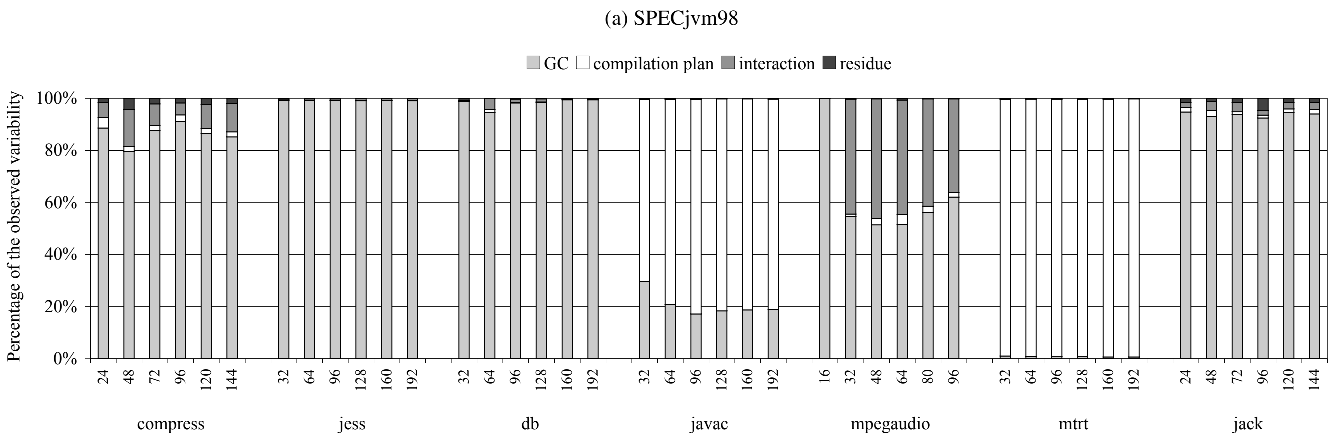
These results show that both factors, the GC strategy and the compilation plan, as well as their mutual interaction, have a significant impact on the observed variability. Most importantly for this study, we conclude that the GC strategy has a significant impact on the mutator time variability in this experiment, and thus the answer to the above question is that the large fraction of disagrees and inconclusives for mutator time is in part due to GC, and is not just a result of the variability observed across compilation plans.

4.5 Majority plan

It follows from the above analyses that multiple compilation plans should be used in replay compilation instead of a single compilation plan. These compilation plans are obtained from multiple profiling runs. Some researchers however, have been using a majority compilation plan which captures information from multiple profiles within a single compilation plan. A majority plan reduces the number of experiments that need to be conducted compared to multiple compilation plans, while (presumably) accounting for the differences observed across compilation plans.

We now evaluate whether a majority plan can be used as a substitute for multiple compilation plans. For doing so, we again use a single-factor ANOVA setup with the GC strategy being the factor. This yields us a mean and a confidence interval per GC strategy and per heap size; this is done for the majority plan on the one hand, and for multiple compilation plans on the other hand. We subsequently perform pairwise GC comparisons between the majority plan against the multiple compilation plans, and classify these comparisons in the *agree*, *disagree* and *inconclusive* categories.

Figure 11 shows that the majority plan and the multiple plans may disagree under both mix and stable replay. In addition, the conclusion is inconclusive in a substantial fraction of the cases, i.e., one of the approaches claims there is no difference between the alternatives, whereas the other does



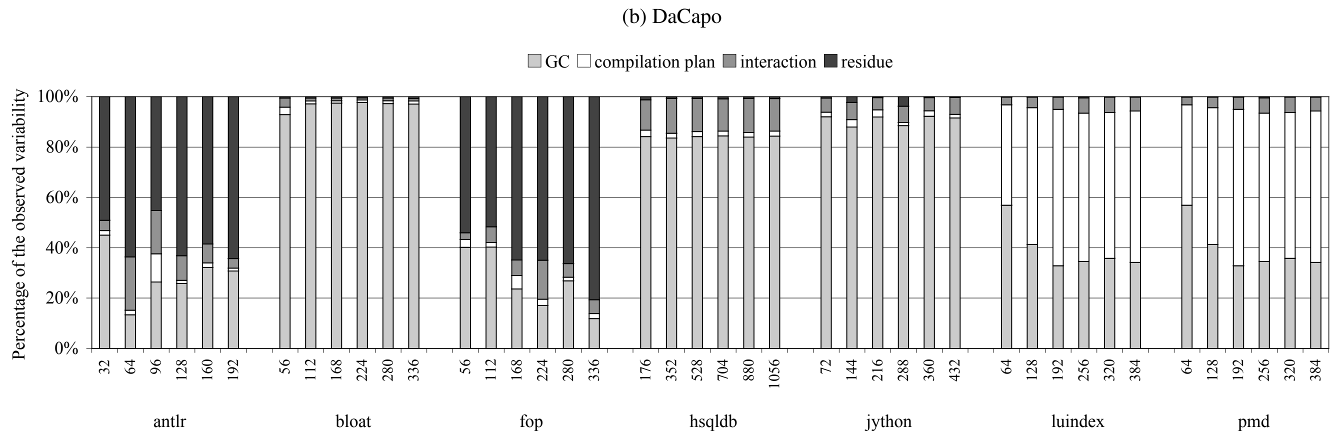


Figure 10: Percentage of the variability in mutator time accounted for by (i) the GC strategy, (ii) the compilation plan, (iii) the interaction between the GC strategy and the compilation plan, and (iv) the residual variability, for 10-iteration compilation plans on the Athlon XP under stable replay.

find a significant difference. We thus conclude that *a majority plan cannot serve as a proxy for multiple compilation plans*.

4.6 Summary

As a summary from this section, we conclude that (i) different compilation plans can lead to execution time variability that is statistically significant, and we observe this variability in GC time, mutator time and total time, and, in addition, we observe this for compilation plans obtained from multi-iteration profiling runs as well as from single-iteration profiling runs; (ii) the reason for this runtime variability is the often observed difference in the methods and their optimization levels appearing in the compilation plans; (iii) different compilation plans can lead to inconsistent conclusions in practical research studies; and (iv) a majority plan is not an accurate proxy for multiple compilation plans. For these reasons we argue that, in order to yield more accurate performance results, *replay compilation should consider multiple compilation plans instead of a single one at replay time*.

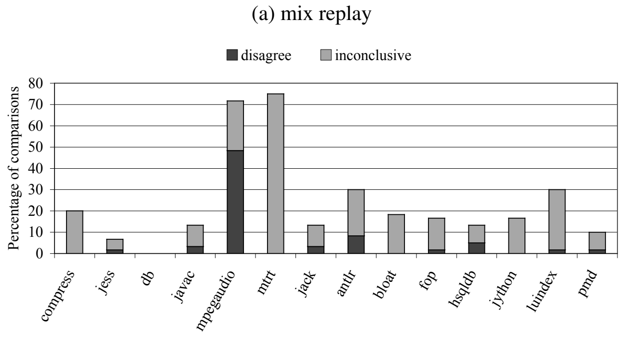
5. Statistical Analysis

Now that we have reached the conclusion that rigorous replay compilation should consider multiple compilation plans, we need statistically rigorous data analysis for taking statistically valid conclusions from these multiple compilation plans.

5.1 Multiple measurements per compilation plan

As mentioned before, the performance measurements for a given compilation plan are still subject to non-determinism. Therefore, it is important to apply rigorous data analysis when quantifying performance for a given compilation plan [15]. Before analyzing the data in terms of whether an innovation improves performance, as will be explained in the following section, we first compute the average execution time per compilation plan. Assume we have k measurements $x_i, 1 \leq i \leq k$, from a population with expected value μ and variance σ^2. The mean of these measurements $\bar{x}$ is computed as

$$\bar{x} = \frac{\sum_{i=1}^{k} x_i}{k}.$$



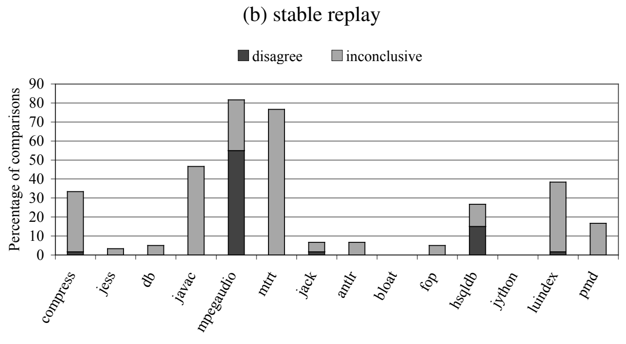


Figure 11: Percentage disagreeing and inconclusive comparisons under (a) mix replay, and (b) stable replay for all benchmarks when comparing a majority plan versus multiple 10-iteration compilation plans on the AMD Athlon platform.

The central limit theory states that, for large values of k (typically $k \geq 30$), $\bar{x}$ is approximately Gaussian distributed with expected value μ and standard deviation $\sigma/\sqrt{k}$, provided that the samples $x_i, 1 \leq i \leq k$, are (i) independent and (ii) come from the same population with expected value μ and finite standard deviation σ. In other words, irrespective of the underlying distribution population from which the measurements x_i are taken, the average measurement mean $\bar{x}$ is approximately Gaussian distributed if the measurements are taken independently from each other. To reach independence in our measurements, we consider the approach described in our prior work [15]: we discard the first measurement for a given compilation plan and retain the subsequent measurements — this assumes that the libraries are loaded when doing the measurements and removes the dependence between subsequent measurements.

5.2 Matched-pair comparison

Comparing design alternatives and their relative performance differences is of high importance to research and development, more so than quantifying absolute performance for a single alternative. When comparing two alternatives, a distinction needs to be made between an experimental setup that involves corresponding measurements versus a setup that involves non-corresponding measurements. Under replay compilation with multiple compilation plans there is an obvious pairing for the measurements per compilation plan. In particular, when evaluating the efficacy of a given innovation, the performance is quantified before the innovation as well as after the innovation, forming an obvious pair per compilation plan. This leads to a so called *before-and-after* or *matched-pair* comparison [20, 23].

To determine whether there is a statistically significant difference between the means before and after the innovation, we must compute the confidence interval for the *mean of the differences* of the paired measurements. This is done as follows, assuming there are n compilation plans. Let $\bar{b}_j, 1 \leq j \leq n$, be the average execution time for compilation plan j *before* the innovation; likewise, let $\bar{a}_j, 1 \leq j \leq n$, be the average execution time for compilation plan j *after* the innovation. We then need to compute the confidence interval for the mean $\bar{d}$ of the n difference values $\bar{d}_j = \bar{a}_j - \bar{b}_j, 1 \leq j \leq n$.

The *confidence interval* for the mean of the differences $[c_1, c_2]$ is defined such that the probability of the expected value δ of the differences falls between c_1 and c_2 equals $1 - \alpha$; α is called the *significance level*, and $(1 - \alpha)$ is called the *confidence level*. Because the significance level α is chosen a priori, we need to determine c_1 and c_2 such that $Pr[c_1 \leq \delta \leq c_2] = 1 - \alpha$. Typically, c_1 and c_2 are chosen to form a symmetric interval around δ, i.e., $Pr[\delta < c_1] = Pr[\delta > c_2] = \alpha/2$. Applying the central-limit theorem as explained in the previous section, which states that $\bar{a}$, $\bar{b}$, and, by consequence, $\bar{d}$ are Gaussian distributed, we thus find that

$$c_1 = \bar{d} - z_{1-\alpha/2}\frac{s_{\bar{d}}}{\sqrt{n}}$$

$$c_2 = \bar{d} + z_{1-\alpha/2}\frac{s_{\bar{d}}}{\sqrt{n}},$$

with $s_{\bar{d}}$ the standard deviation of the difference values computed as follows:

$$s_{\bar{d}} = \sqrt{\frac{\sum_{i=1}^{n}(\bar{d}_i - \bar{d})^2}{n-1}}.$$

The value $z_{1-\alpha/2}$ is defined such that a random variable Z that is Gaussian distributed with expected value $\mu = 0$ and variance $\sigma^2 = 1$ has a probability of $1 - \alpha/2$ to be smaller than or equal to $z_{1-\alpha/2}$. The value $z_{1-\alpha/2}$ is typically obtained from a precomputed table.

The above assumes that the number of compilation plans is sufficiently large, i.e., $n \geq 30$ [23]. In case there are less than 30 compilation plans, $\bar{d}$ can no longer be assumed to be Gaussian distributed. Instead, it can be shown that the distribution of the transformed value $t = (\bar{d} - \delta)/(s_{\bar{d}}/\sqrt{n})$ follows the *Student's* t-distribution with $n - 1$ degrees of freedom. The confidence interval can then be computed as:

$$c_1 = \bar{d} - t_{1-\alpha/2;n-1}\frac{s_{\bar{d}}}{\sqrt{n}}$$

$$c_2 = \bar{d} + t_{1-\alpha/2;n-1} \frac{s_{\bar{d}}}{\sqrt{n}},$$

with the value $t_{1-\alpha/2;n-1}$ defined such that a random variable T that follows the Student's t distribution with $n - 1$ degrees of freedom has a probability of $1-\alpha/2$ to be smaller than or equal to $t_{1-\alpha/2;n-1}$. As with the $z_{1-\alpha/2}$ value from above, also the $t_{1-\alpha/2;n-1}$ value is typically obtained from a precomputed table. It is interesting to note that as n increases, the Student's t-distribution approaches the Gaussian distribution.

Once the confidence interval is computed, we then verify whether the confidence interval includes zero. If the confidence interval includes zero, we conclude, at a given $(1-\alpha)$ confidence level, that the measured differences are not statistically significant. If not, there is no statistical evidence to suggest that there is no statistically significant difference. Note the careful wording here. There is still a probability α that the observed differences are due to random effects in the measurements and not due to differences between the alternatives. In other words, we cannot assure with a 100% certainty that there is an actual difference between the compared alternatives. In some cases, taking such 'weak' conclusions may not be very satisfactory but it is the best we can do given the statistical nature of the measurements.

6. Rigorous Replay Compilation

Figure 12 illustrates the overall replay compilation methodology that we advocate when comparing two alternatives. We start by collecting n compilation plans. For each of these compilation plans we then collect k performance numbers for both the 'before' and the 'after' experiments, and subsequently compute an average performance number per compilation plan before the innovation, $\bar{b}_j$, as well as after the innovation, $\bar{a}_j$. The differences between the alternatives per compilation plan, $\bar{d}_j = \bar{b}_j - \bar{a}_j$, then serve as input to the matched-pair comparison as explained in the previous section.

The replay methodology proposed in the previous section is more rigorous than current practice because it includes multiple compilation plans. The downside is that this methodology implies that more experiments need to be run. We now need to collect performance numbers for multiple compilation plans instead of a single compilation plan. This may be time-consuming.

Fortunately, only a limited number of compilation plans need to considered. The reason is that matched-pair comparison leverages the likely observation that the variability in relative performance difference between the alternatives is smaller than the variability observed across the compilation plans. More precisely, as we observed in Section 4, the variability in performance between different compilation plans can be large. However, the intuition is that the variability in relative performance across alternatives *for a given compilation plan* is not very large. A compilation plan leading to high performance for one alternative is likely to also yield high performance for the other alternative, even if the absolute performance is different. In our experiments, we found this to be the case in general, as will be shown later. We will exploit this property to limit the number of compilation plans that need to be considered while maintaining high accuracy and tight confidence intervals.

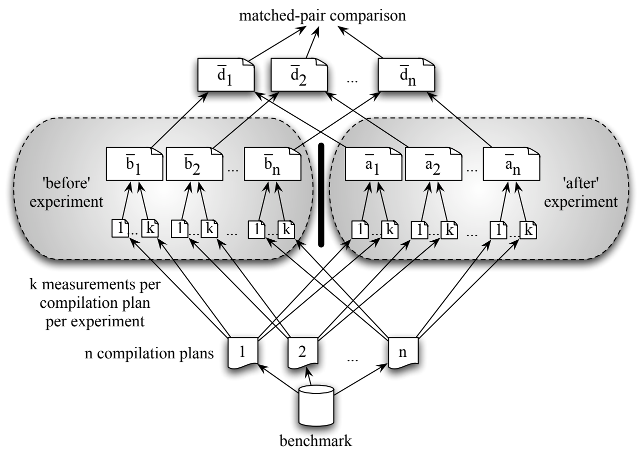


Figure 12: Replay compilation methodology using multiple compilation plans.

The underlying reason is that a matched-pair comparison exploits the property of paired or so called corresponding measurements. To better understand this important property, we first need to explain how to compare two alternatives in case of non-corresponding measurements.

6.1 Non-corresponding measurements

Consider two alternatives and respective measurements $x_{1j}, 1 \leq j \leq n_1$ and $x_{2j}, 1 \leq j \leq n_2$; assume there is no correspondence or pairing. We now need to compute the confidence interval of the difference of the means. We first need to compute the averages $\bar{x}_1$ and $\bar{x}_2$ for the two alternatives. The difference of the means equals $\bar{x} = \bar{x}_2 - \bar{x}_1$. The standard deviation of the difference of the means equals

$$s_x = \sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}},$$

with s_1 and s_2 the standard deviation for the two respective alternatives.

We can now compute the confidence interval $[c_1, c_2]$ for the difference of the means, again based on the central limit theory:

$$c_1 = \bar{x} - z_{1-\alpha/2} \cdot s_x$$

$$c_2 = \bar{x} + z_{1-\alpha/2} \cdot s_x.$$

If the resulting confidence interval includes zero, we can conclude that, at the confidence level chosen, there is no significant difference between the two alternatives.

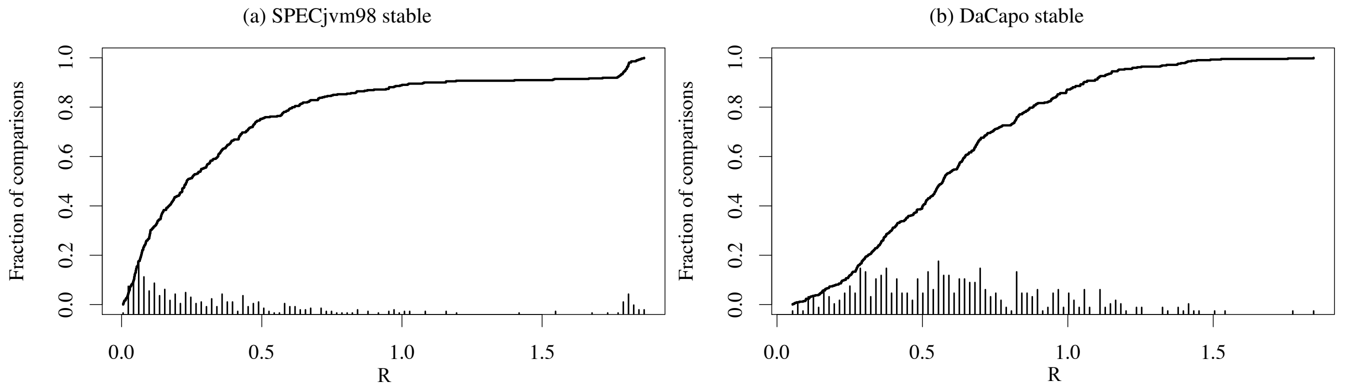


Figure 13: Cumulative distribution of the ratio R in confidence interval width between matched-pair comparison versus non-corresponding measurements statistics.

6.2 Comparison between corresponding and non-corresponding measurements

Let's now compare the confidence interval computed for corresponding measurements versus non-corresponding measurements. Assume $n_1 = n_2 = n$, and $b_i = x_{2i}$ and $a_i = x_{1i}$. Recall the confidence interval for corresponding measurements equals:

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \frac{s_{\bar{d}}}{\sqrt{n}},$$

or

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \sqrt{\frac{\sum_{i=1}^{n}(\bar{d}_i - \bar{d})^2}{n(n-1)}}.$$

For non-corresponding measurements, the confidence interval for the difference of the means equals:

$$c_{1,2} = \bar{d} \pm z_{1-\alpha/2} \cdot \sqrt{\frac{\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2}{n(n-1)} + \frac{\sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2}{n(n-1)}}.$$

Comparing both confidence intervals boils down to comparing

$$\sum_{i=1}^{n}(\bar{d}_i - \bar{d})^2$$

for the corresponding measurements, versus

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2$$

for the non-corresponding measurements. Writing $\bar{d}_i$ as $\bar{b}_i - \bar{a}_i$, and $\bar{d}$ as $\bar{b} - \bar{a}$, enables expanding the expression for the corresponding measurements to:

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2 - 2 \cdot \sum_{i=1}^{n}(\bar{a}_i - \bar{a})(\bar{b}_i - \bar{b}).$$

By consequence, if the term

$$\sum_{i=1}^{n}(\bar{a}_i - \bar{a})(\bar{b}_i - \bar{b})$$

is positive, then the confidence interval for the corresponding measurements is smaller than the confidence interval for the non-corresponding measurements. In other words, corresponding measurements result in tighter confidence intervals if the performance variation is large across the compilation plans, i.e., $\bar{a}_i - \bar{a}$ and $\bar{b}_i - \bar{b}$ are large, and if the relative performance variation is limited across compilation plans when comparing two alternatives.

To illustrate this finding empirically through our GC case study, we compute the ratio R:

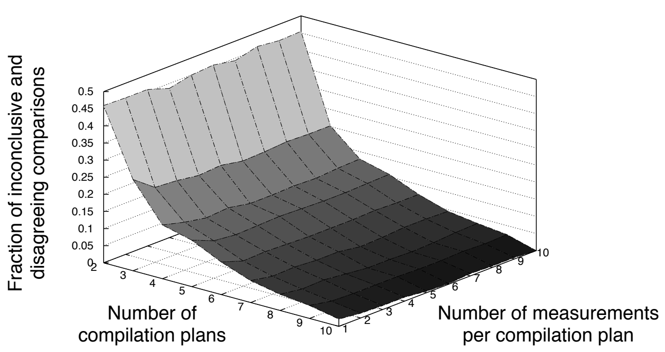
$$R = \frac{\sum_{i=1}^{n}(\bar{d}_i - \bar{d})^2}{\sum_{i=1}^{n}(\bar{a}_i - \bar{a})^2 + \sum_{i=1}^{n}(\bar{b}_i - \bar{b})^2},$$

across all benchmarks and all heap sizes, for all pairwise GC strategy comparisons. If R is smaller than one, this means that the confidence interval computed through matched-pair comparison is smaller than the confidence interval computed through statistics assuming non-corresponding measurements. Figure 13 shows the cumulative distribution of R. The various graphs show that in the majority of the cases, matched-pair comparison indeed results in smaller confidence intervals of the difference of the means. For example, for DaCapo and stable replay, for over 85% of the cases, matched-pair comparison results in a smaller confidence interval.

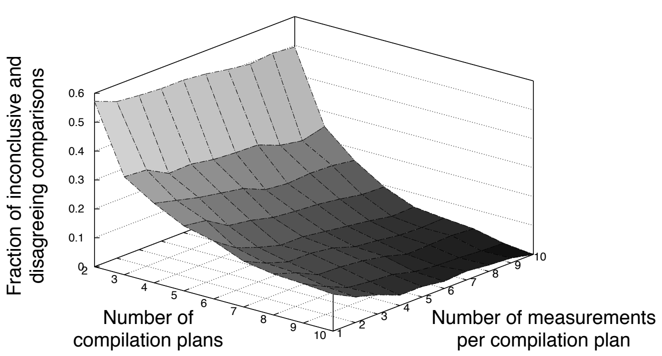
6.3 Number of compilation plans

The above analysis shows that matched-pair comparison for analyzing the performance results from multiple compilation plans is likely to result in tighter confidence intervals than using non-corresponding measurements statistics. This

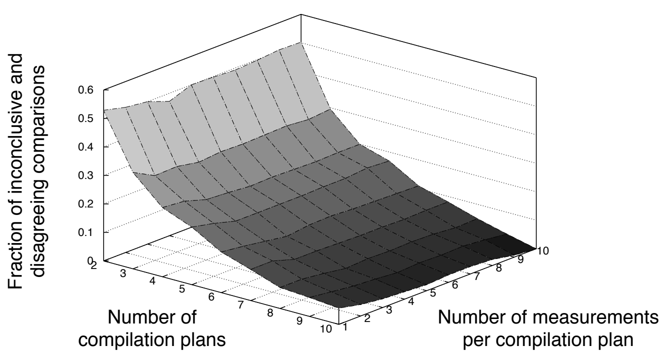
(a) SPECjvm98 stable total execution time



(b) DaCapo stable total execution time



(c) SPECjvm98 stable mutator execution time



(d) DaCapo stable mutator execution time

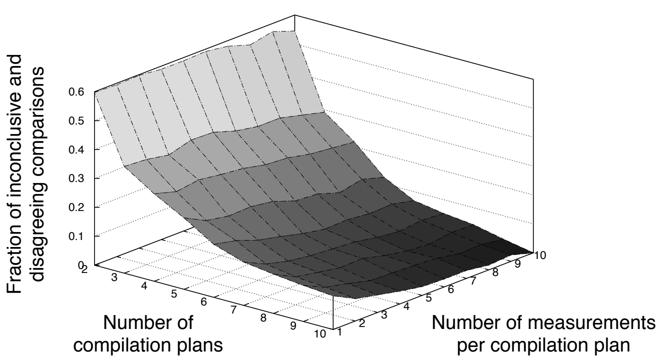


Figure 14: Exploring the trade-off between the number of compilation plans versus the number of measurements per 10-iteration compilation plan as measured on the AMD Athlon platform.

observation has an important implication. It means that for the same level of accuracy, i.e., for the same confidence interval size, fewer compilation plans need to be considered when using matched-pair comparison statistics instead of non-corresponding measurements statistics. Or, in other words, under matched-pair comparison, the number of compilation plans that are needed to obtain tight confidence intervals is limited.

We now leverage this observation to find a good trade-off between the number of compilation plans versus the number of measurements per plan to obtain accurate performance numbers. For exploring this trade-off, we again consider our GC case study in which we consider 5 GC strategies and 6 heap sizes. We now pairwise compare GC strategies per heap size through matched-pair comparison. Figure 14 shows the fraction inconclusive and disagree conclusions (averaged across all SPECjvm98 and DaCapo benchmarks) as a function of the number of compilation plans and the number of measurements per plan for the stable run. We show results for both total execution time and mutator execution time. The reference point is the setup for which we consider 10 compilation plans and 10 runs per compilation plan. In other words, a point (x, y) in this graph shows the fraction inconclusive and disagree comparisons for x compilation plans and y measurements per plan compared to 10 compilation plans and 10 measurements per plan. We observe that the fraction inconclusive and disagree conclusions quickly decreases with even a limited number of, say 4 or 5, compilation plans. At the same time, the fraction inconclusive and disagree conclusions is fairly insensitive to the number of measurements per compilation plan. In other words, for a given experimentation time budget, it is beneficial to *consider multiple compilation plans rather than multiple measurements per compilation plan*.

7. Summary

Replay compilation is an increasingly widely used experimental design methodology that aims at controlling the non-determinism in managed runtime environments such as the Java virtual machine. Replay compilation fixes the compilation load by inducing a pre-recorded compilation plan at replay time. The compilation plan eliminates the non-determinism due to timer-based sampling for JIT optimization.

Current practice typically considers a single compilation plan at replay time, albeit a plan may have been derived from multiple profiles. The key observation made in this paper is that a single compilation plan at replay time does not sufficiently account for the variability observed across different profiles. The reason is that different methods may be optimized at different levels of optimization across different compilation plans. And this may lead to inconsistent conclusions across compilation plans in practical research studies. In addition, we have shown that a majority compilation plan is no proxy for using multiple compilation plans. We therefore advocate replay compilation using multiple compilation plans so that the performance number obtained from replay compilation is a better representative for average performance.

The statistical data analysis that we advocate under replay compilation with multiple compilation plans is matched-pair comparison. Matched-pair comparison considers the before and after experiments for a given compilation plan as a pair, and by doing so, achieves tighter confidence intervals in general than assuming unpaired measurements. The reason is that replay compilation leverages the observation that the variability in the performance differences between two design alternatives is likely smaller than the variability across compilation plans. By consequence, replay compilation with multiple compilation plans and matched-pair comparison limits the number of compilation plans that need to be considered, and thus limits the experimentation time overhead incurred by multiple compilation plans.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments — their detailed suggestions greatly helped us improving this paper. Andy Georges and Lieven Eeckhout are Postdoctoral Fellows of the Fund for Scientific Research–Flanders (Belgium) (FWO–Vlaanderen). Dries Buytaert was supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT).

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *OOPSLA*, pages 47–65, Oct. 2000.

[2] BEA. BEA JRockit: Java for the enterprise. Technical white paper. http://www.bea.com, Jan. 2006.

[3] S. Blackburn, P. Cheng, and K. S. McKinley. Myths and reality: The performance impact of garbage collection. In *SIGMETRICS*, pages 25–36, June 2004.

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

[5] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, Oct. 2006.

[6] S. M. Blackburn, M. Hertz, K. S. McKinley, J. E. B. Moss, and T. Yang. Profile-based pretenuring. *ACM Trans. Program. Lang. Syst.*, 29(1):2, 2007.

[7] S. M. Blackburn and A. L. Hosking. Barriers: Friend or foe? In *ISMM*, pages 143–151, Oct. 2004.

[8] M. D. Bond and K. S. McKinley. Continuous path and edge profiling. In *MICRO*, pages 130–140, Dec. 2005.

[9] M. D. Bond and K. S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, Oct. 2007.

[10] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, Oct. 2006.

[11] D. Buytaert, A. Georges, M. Hind, M. Arnold, L. Eeckhout, and K. De Bosschere. Using HPM-sampling to drive dynamic compilation. In *OOPSLA*, pages 553–568, Oct. 2007.

[12] J. Cavazos and J. E. B. Moss. Inducing heuristics to decide whether to schedule. In *PLDI*, pages 183–194, June 2004.

[13] M. Cierniak, M. Eng, N. Glew, B. Lewis, and J. Stichnoth. The open runtime platform: A flexible high-performance managed runtime environment. *Intel Technology Journal*, 7(1):5–18, 2003.

[14] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *OOPSLA*, pages 169–186, Nov. 2003.

[15] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, pages 57–76, Oct. 2007.

[16] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE*, pages 111–121, June 2006.

[17] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *PLDI*, pages 364–375, June 2006.

[18] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA*, pages 251–269, Oct. 2004.

[19] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *OOPSLA*, pages 69–80, Oct. 2004.

[20] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis*. Prentice Hall, 2002.

[21] C. Krintz and B. Calder. Using annotations to reduce dynamic optimization time. In *PLDI*, pages 156–167, May 2001.

[22] B. Lee, K. Resnick, M. D. Bond, and K. S. McKinley. Correcting the dynamic call graph using control-flow constraints. In *CC*, pages 80–95, March 2007.

[23] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[24] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. Javana: A system for building customized Java program analysis tools. In *OOPSLA*, pages 153–168, Oct. 2006.

[25] V. Sundaresan, D. Maier, P. Ramarao, and M. Stoodley. Experiences with multithreading and dynamic class loading in a Java just-in-time compiler. In *CGO*, pages 87–97, Mar. 2006.

[26] J. Neter, M. H. Kutner, W. Wasserman, and C. J. Nachtsheim. *Applied Linear Statistical Models*. McGraw-Hill, 1996.

[27] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: Improving debuggability of a just-in-time compiler. In *OOPSLA*, pages 241–252, Oct. 2006.

[28] M. Paleczny, C. Vick, and C. Click. The Java Hotspot server compiler. In *JVM*, pages 1–12, Apr. 2001.

[29] N. Sachindran, and J. E. B. Moss. Mark-copy: fast copying GC with less space overhead. In *OOPSLA*, pages 326–343, Nov. 2003.

[30] N. Sachindran, J. E. B. Moss, and E. D. Berger. MC2: high-performance garbage collection for memory-constrained environments. In *OOPSLA*, pages 81–98, Oct. 2004.

[31] F. T. Schneider, M. Payer, and T. R. Gross. Online optimizations driven by hardware performance monitoring. In *PLDI*, pages 373–382, June 2007.

[32] S. Soman, C. Krintz, and D. F. Bacon. Dynamic selection of application-specific garbage collectors. In *ISMM*, pages 49–60, June 2004.

[33] Standard Performance Evaluation Corporation. SPECjvm98 Benchmarks. http://www.spec.org/jvm98.

[34] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a Java just-in-time compiler. In *TOPLAS*, 27(4):732–785, July 2005.

[35] P. F. Sweeney, M. Hauswirth, B. Cahoon, P. Cheng, A. Diwan, D. Grove, and M. Hind. Using hardware performance monitors to understand the behavior of Java applications. In *VM*, pages 57–72, May 2004.

[36] J. Whaley. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 78–87, June 2000.

[37] T. Yang, M. Hertz, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Automatic heap sizing: taking real memory into account. In *ISMM*, pages 61–72, June 2004.