Rigorous Benchmarking in Reasonable Time

Tomas Kalibera
University of Kent, Canterbury
t.kalibera@kent.ac.uk

Richard Jones
University of Kent, Canterbury
r.e.jones@kent.ac.uk

Abstract

Experimental evaluation is key to systems research. Because modern systems are complex and non-deterministic, good experimental methodology demands that researchers account for uncertainty. To obtain valid results, they are expected to run many iterations of benchmarks, invoke virtual machines (VMs) several times, or even rebuild VM or benchmark binaries more than once. All this repetition costs time to complete experiments. Currently, many evaluations give up on sufficient repetition or rigorous statistical methods, or even run benchmarks only in training sizes. The results reported often lack proper variation estimates and, when a small difference between two systems is reported, some are simply unreliable.

In contrast, we provide a statistically rigorous methodology for repetition and summarising results that makes efficient use of experimentation time. Time efficiency comes from two key observations. First, a given benchmark on a given platform is typically prone to much less non-determinism than the common worst-case of published corner-case studies. Second, repetition is most needed where most uncertainty arises (whether between builds, between executions or between iterations). We capture experimentation cost with a novel mathematical model, which we use to identify the number of repetitions at each level of an experiment necessary and sufficient to obtain a given level of precision.

We present our methodology as a cookbook that guides researchers on the number of repetitions they should run to obtain reliable results. We also show how to present results with an effect size confidence interval. As an example, we show how to use our methodology to conduct throughput experiments with the DaCapo and SPEC CPU benchmarks on three recent platforms.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques.

Keywords Benchmarking methodology; statistical methods; DaCapo; SPEC CPU.

1. Introduction

Experimental evaluation is key to programming language and systems research. It has proved hard to do well. Programs are (possibly surprisingly) non-deterministic, and their execution times can vary significantly from run to run, or with different builds. Such variation makes it uncertain what the effect of, say, a particular optimisation might be. The challenge to experimental computer scientists is to deal with this uncertainty, and to provide reliable estimates of program speedup. The difficulty is to know how many experiments to run, and to minimise the cost in experiment time without compromising the validity of the results. We show that, in most cases, good experimental methodology is feasible without excessive cost.

We focus on execution time, the key measurement in, for example, 90 out of 122 papers presented in 2011 at PLDI, ASPLOS and ISMM, or published in TOPLAS (nos. 1–4) and TACO (nos. 1–2). Unfortunately, the overwhelming majority of these papers reported results in ways that seem to make their work impossible to repeat, or did not convincingly demonstrate their claims for performance improvement: 71 failed to provide any measure of variation (such as variance or a confidence interval) for their results. This is unparalleled in most other scientific and social scientific fields. It risks reporting misleading results.

These risks are real. Advances in performance in our field are often small (Mytkowicz et al [20] report a median of 10%) and so can fall within the bounds of measurement error. In a case study of Java VM/DaCapo benchmarking, Georges et al [9] show how poor methods of repeating and summarising experiments led to misleading results. Ignoring systematic bias in code layout in experiments with SPEC CPU benchmarks can also deliver misleading results [20]. Even if reported speedups are relatively large, a rigorous study should estimate the measurement error.

While it is disappointing that all this evidence seems to have had little impact on practice in our field, maybe it is understandable. Researchers find themselves faced with the task of running ever more experiments in order to deal with these problems. Thus they run multiple iterations of a benchmark for each VM execution and run each VM execution multiple times. Variations in Unix environment size and in link order, and randomised algorithms in compilers (producing different binaries for the same code) all seem to impose further requirements for repetition [1, 9, 15, 20]. For example, running the SPEC INT benchmarks to measure speedup of a compiler optimisation against a base, using 30 different linking orders, takes 3 days on a recent platform. Running the DaCapo 2006 and 2009 benchmarks [3] to compare two systems, using 20 executions and 10 iterations, also takes almost 3 days.

We renounce any catalogue of despair. We show that good experimental methodology is feasible: we can cater for variation in performance without excessive repetition in most cases. Our approach is to adapt experiment design to the problems that a particular platform and benchmark present. In our experience, research groups run the same benchmarks on the same systems for years. This is done for a good reason, as otherwise the performance changes due to, say, improvements of a garbage collector under study could be confused with performance changes due to operating system upgrades. In this setting, an initial investment into dimensioning benchmark experiments will pay off. As common in most statistical analyses of data, this does involve some manual steps. In most projects this is a one-off investment but, of course, any major change to the system would necessitate re-dimensioning.

ISMM'13, June 20–21, 2013, Seattle, Washington, USA.

Note that repetition is most needed where most non-determinism occurs in the experiment. We show how to establish the repetition counts necessary for any evaluation to be valid, and sufficient to provide the most precise result for a given experimentation budget. We show how to estimate the error bounds both to evaluate the performance of a single system and to compare execution times of a baseline and a new version of a system: such ratios are commonly used but hardly ever qualified with error bounds. As an example, we apply our methodology to a case study of the performance of DaCapo and SPEC CPU benchmarks. Our contributions are:

- We explain the shortcomings of statistical methods commonly used in our field. Instead, we offer a sound method based on effect sizes and confidence intervals.
- We provide an observational study of non-determinism in the DaCapo and SPEC CPU benchmarks.
- We offer a sound experimental methodology that makes best use of experiment time. We establish, both formally and in practice, the optimum number of repetitions of experiments to achieve the most precise results for a given experiment time.
- We compare our methodology with heuristics-based practice, and show that the latter often leads to either too few or too many experiments.
- We revisit the question of the effect of code layout on the performance of DaCapo and SPEC CPU and show that it is less important than prior work had shown.

2. Related Work

When running SPEC CPU benchmarks, Mytkowicz et al [20] found a number of sources of significant *measurement bias*, i.e. a systematic error in the measurement of systems that might favour one system over another. These included the Unix environment size and the link order (the order in which `.o` files are given to the linker), which affect the memory layout of data and code, and hence memory hierarchy (cache, virtual memory) performance. Mytkowicz et al suggest eliminating this bias by randomising the experimental set-up. Because of the expense of running benchmarks with many different link orders, they used only the 'training' sizes for the benchmarks. We repeated their experiments and confirmed the impact of link order for training sizes. However, our experimental methodology made it feasible to consider SPEC's full 'reference' sizes, where we found that, although there was variation, it was very small (Section 7.2).

Getting the experimental methodology right is crucial because, as Georges et al [9] demonstrate, different methodologies can lead to different conclusions. They advocate running multiple iterations of each Java benchmark within a single VM execution, and multiple VM executions. In each execution, a number of initial iterations is dropped to *warm up* the benchmark before it is deemed to have converged to steady state. They establish the warm-up number on the fly by finding the first window of N (say, 10) iterations that seems stable (the relative width of 95% confidence interval for the mean is less than 5%). The sample mean of such a window is used to summarise this execution of the VM.

The DaCapo'09 benchmark harness also tries to detect steady state automatically, but reports one iteration per VM execution. It keeps calculating relative variation (the ratio of standard deviation and mean) over windows of three iterations until it drops below 3%, at which point the benchmark is deemed to have converged: the time of one iteration is used as the summary. We observe that these two automated methods do not perform well, as they often lead to either too few experiments, hence failing to get to a steady state, or far too many (Section 6.3).

The compilation strategy of Java VMs is non-deterministic. Which methods are JIT-compiled and when is determined by sampling method execution. One way to reduce variation between VM executions is always to compile to the same plan. Ogata et al [22] use a sampling compiler to capture a compilation plan and then, for measurements, a replay compiler which compiles methods according to this plan. Georges et al [10] advocate compiler replay with multiple plans to avoid any bias introduced by a fixed compilation plan. They summarise with fixed-effects analysis of variance. However, this technique can be used only for very similar systems, where forcing the same compilation plan could still give results representative of real executions.

Stabiliser [7] is an LLVM-based compiler and runtime environment for code, stack and heap layout randomisation. It changes the layout randomly at regular intervals during one execution of a benchmark, in order to reduce the need for repeated execution. One benefit is that the sources of layout bias (such as the environment size [20] or link order) need not be identified by the experimenter. On the other hand, the results may include far more variation than in real systems, and hence can mislead. Our approach is less intrusive, but expects the experimenter to find potential sources of bias. If layout is the cause of bias, our methodology could be combined with Stabiliser. Our approach also applies to systems where online re-randomisation is not yet available, such as JVMs.

3. The Challenge of Reasonable Repetition

We have seen how variation can be introduced at several stages of a benchmark experiment (iteration, execution, compilation and so on). Three kinds of variables influence the outcomes of experiments. Values of *controlled variables* (such as the platform we choose, the heap size or compiler options) and how they impact the results are of interest for the evaluation. *Random variables* (such as the time between hardware interrupts or scheduling order on a multi-processor) change frequently in a random or non-deterministic manner. We are interested in the statistical properties of our results in face of random variables, but not in the individual values of these variables. *Uncontrolled variables* happen to be fixed for most or all of an experiment, but are beyond our control. If these impact the results, they cause bias and mislead. Hence, the experimenter's first task is to identify uncontrolled variables that impact results and modify the experimental system so that these become either controlled or random. For example, randomising link order turns an uncontrolled variable into a random one.

Experiment design is a statistical discipline which deals with how to run experiments efficiently given a set of controlled and random variables (see e.g. Maxwell & Delaney [18] for more details and references to the literature). The goal of benchmarking experiments is typically to estimate (a confidence interval for) the mean execution time of a given benchmark on one or more platforms, that is for a relatively small set of combinations of values of controlled variables. Note that the mean is a property of the underlying probability distribution of the population of random execution times. In practice we can never know this mean or the distribution, but a confidence interval can tell us something about what that mean might be. If we use sound experimental methodologies in our studies, each time constructing a 95% confidence interval for the mean, we can expect that overall in 95% of cases our intervals will have covered the true means. Often the goal is also to estimate speedup (65 of the 90 papers in our survey): such an estimate should also be qualified by a confidence interval.

The challenge we address here, and also the next step of the experimenter, is to design efficient experiments (repetitions and repetition counts) given the random variables present. A further challenge is to identify and get rid of the uncontrolled variables.

Think of a benchmarking experiment as a sequence of actions, starting with building the benchmark and system under test (e.g. a virtual machine or a compiler — we call this *compilation*) and

ending with providing a single execution time measurement. If this sequence included neither random nor uncontrolled variables turned to random, the design would be trivial — just run once and take the result. But in reality a number of random variables in the sequence will influence the measurement. Some take effect before the measured operation starts and influence it indirectly, others act during it.

This necessitates repeating the sequence a number of times, at least from the point where the first random variable takes effect. Suppose compilation was not random (it was deterministic and performance did not depend on code layout): in this case we would not have to repeat it. In contrast if, say, the start-up of a VM execution includes some random variation, then we must repeat VM *executions* but can do this with the same binary. We refer to points of potential repetition as *levels* of the experiment (not to be confused with a 'factor level' in ANOVA). The *highest* level is the first source of variation in the experiment sequence, e.g. compilation. The *lowest* level is the operation measured (e.g. an *iteration* of a benchmark).

Through repetition we get a number of measurements and typically we calculate a confidence interval. The more repetitions made, the narrower ('more precise') is the interval. At the very least, repetition must be done at the highest level that has random variation to avoid bias, but sometimes repeating at lower levels can reduce experimentation time without sacrificing precision.

In the rest of this paper, we consider how to design experiments that will deliver reliable results at the least cost in experimentation time. We explore how many repetitions are needed, at which levels, and for what price in terms of experimentation time. Although our approach is general, as an example we consider benchmarking with two suites, DaCapo and SPEC CPU, and three levels (repeating iteration, execution and compilation).

4. The Challenge of Summarising Results

As we observed above, it is still uncommon in our discipline to report results with any degree of statistical rigour despite the efforts of Georges, Mytkowicz and others [9, 10, 20]. Often the plausible argument is made that, if performance improvements are large (e.g. $2\times$ or more), there is no need for statistical machinery to prove that they are real. However, improvements reported in the programming languages field are often small enough (e.g. about 10% [20]) to necessitate some statistical demonstration that they do not come about by chance. Moreover, even a large speedup should come with error bounds estimate to allow rigorous quantification and comparison of different studies. Where researchers have used statistical techniques, these have often been significance tests.

Significance testing. Quantification of performance change with statistical significance [13, 16] tests whether it is likely that two systems have different performance. The decision is based on the probability that the observed difference (or a larger one) in the (sample) means of the two systems would occur if the (true) means were the same. This probability, the p-value, is compared against a pre-defined threshold, the significance level (i.e. 5%). If the p-value is smaller then the null hypothesis, that the two systems have the same performance, is rejected.

Problems with statistical significance testing have long been known. The method is deprecated in other disciplines [5, 6, 21, 25], and some journals explicitly require alternative methods [12]. First, significance testing does not provide the metric we are ultimately interested in, a reliable estimate of e.g. the ratio of the execution times of two systems. The test is also vulnerable to the number of measurements used. The larger the sample size is (the more measurements we have), the more unlikely even a very small difference becomes. In practice this means that a large sample size (and in our field it is easy to generate very large samples) will nearly always lead to the decision that there is a 'statistically significant' difference in performance, even if the true difference is so small that it is of little interest; statistical significance methods confuse sample size and practical relevance [6]. Statistical significance tests are also notoriously hard to interpret. This may be because they do not give us the answer to what we want to know but instead offer temptations, such as the belief that the p-value is actually the probability that the systems have the same performance. The interpretation of the results of statistical tests is sufficiently tricky that even some statistics textbooks have got it wrong (examples are given by Cohen [6]).

Visual tests. An alternative is to construct confidence intervals for the two systems under test and to examine whether they overlap. If they do not, then one can conclude that it is likely that the systems differ in performance [9, 16]. Jain [13] adds another step, falling back to a statistical test if the intervals overlap only slightly, i.e. if the centre of neither interval lies within the other. An advantage of the visual test is that it gives a clear measure of the size of the difference in mean performance, while the intervals also show the uncertainty of the systems in isolation. However, although this is a useful aid for an analyst, a visual test still does not tell us what we want to know (an estimate for the ratio of the performances of the two systems and its error bound) and, in contrast to a statistical test, lacks any rigorous semantics as its error is not known. It is actually rather conservative. For example, with 95% confidence intervals, the probability of error is not 5%, but less than 1% under the normality assumption [23].

Normality. Researchers in our field commonly assume that the Central Limit Theorem justifies their use of parametric methods such as a t-test or analysis of variance on data that is not normally distributed. Informally, the theorem states that the average of a sufficiently large number of independent and identically distributed (i.i.d.) random variables tends to follow a Normal distribution. Although the theorem does not fully justify this assumption, parametric methods have been found to be robust under various sets of conditions [2, 24]. Our summarising method is based on analysis of variance, and we provide a full derivation and discuss its assumptions in our technical report [14]; we also demonstrate an alternative non-parametric method.

Effect size. Better methods are available. In section 9.3, we show how to construct an *effect size* confidence interval. Summaries can be as simple as "we are 95% confident that system A is faster than system B by $5.5\% \pm 2.5\%$". Such a statement is more natural than those derived from significance testing and less open to misinterpretation: it quantifies the size of the change, gives its error bound and indicates how certain this result is.

> **RECOMMENDATION: Analysis of results should be statistically rigorous and in particular should quantify any variation. Report performance changes with effect size confidence intervals.**

In summary, sound experimental methodology is an increasing concern for the computer science research community. On one hand, it is clear that our field lags behind the standards expected by other sciences for reporting experimental evaluations. This has led to the foundation of *Evaluate Collaboratory* (`http://evaluate.inf.usi.ch/`) to promote better experimental practice. On the other hand, researchers are unclear as to how to make best use of their time to run and report experiments without sacrificing rigour. This paper is a contribution towards resolving that dilemma.

5. Benchmarks and Platforms

Benchmarks For JVM experiments we use DaCapo 2006 and 2009 benchmarks (2006-10-MR2 and 9.12-bach) running on OpenJDK 7 (version 7u2, build 13, November 17, 2011) compiled with

Table 1. Platforms Used in the Case Study.

	Linux		CPU	GHz	LLC	Mem.
P1	3.0.0	64bit	4x16(x1) AMD Opteron	2.1	12M L3	64G
P2	2.6.38	64bit	2x4(x2) Intel Xeon	2.27	8M L3	12G
P3	3.0.0	64bit	1x4(x2) Intel Core i7	3.4	8M L3	16G
P4	2.6.35	64bit	1x2(x1) Intel Core 2	2.4	4M L2	4G
P5	2.6.35	32bit	1x1(x2) Intel Pentium 4	3.2	1M L2	4G

gcc version 4.7. These benchmarks are widely used in garbage collection and VM research. We report only those benchmarks that run without crashing on the VM. We run the 'large' and 'small' sizes of each workload, using the default production settings of the VM and letting the benchmark harness scale the workloads to all available processors.

For gcc experiments we use CINT (integer benchmarks) from the SPEC CPU2006 benchmark suite, version 1.2. SPEC CPU benchmarks are widely used for C/Fortran compiler and CPU performance measurements. We build the benchmarks with gcc 4.7 using the O3 optimisations, and run the 'train' (smallest) and the 'reference' (standard) sizes.

Platforms We use 5 different platforms, each running a version of Ubuntu Linux. In Table 1, '2x4(x2)' denotes a system with 2 physical processors, each with 4 cores and 2-way hyper-threading. Platforms P1 and P2 have non-uniform memory access. We disable all system services that might interfere with measurements.

6. Repeating Iterations

Researchers are typically interested in steady state performance, so we restrict our study to this case. Performance in the steady state should be 'somewhat' stable, without clear trends, and particularly without any obvious overhead of VM or application initialisation. We cannot take *live* measurements before this state is reached.

We identify an initialised state and an independent state of benchmark execution. We call a state *independent* if the execution times of the benchmark iterations are (statistically) independent and identically distributed. A state is *initialised* — the lower bar — when iterations are no longer subject to obvious and significant initialisation overhead. Such overhead may be due to dynamic linking, filling I/O buffers for data/code, or just-in-time compilation. Independence means that the duration of an iteration is not affected by earlier iterations in the same execution. By definition, 'independent' implies 'initialised'. We believe that most researchers would regard an independent state as 'steady', and so i.i.d. is a well-defined sufficient condition for the steady state. We also believe that 'initialised' would be widely accepted as a necessary condition for a steady state.

Random factors, such as context switches, scheduling order or Java heap layout, can affect performance, so repetition at the iteration level or higher is needed. Repeating iterations is experimentally cheaper since there is no need to wait for a new execution (or higher level operation) to reach a steady state.

Note that it does not makes sense to repeat measurements unless the system has reached an independent state. If measurements are not i.i.d., the variance and confidence interval estimates will be biased. The first question to ask is, therefore, *does a benchmark reach an independent state and, if so, after how many iterations?*

Aside: some researchers might repeat statistically dependent iterations and then include, say, their average in further summary [9]. This approach is not incorrect if the results are correctly interpreted, but the risk of misinterpretation is high. It redefines what is measured. For example, rather than asking "how long does it take to run 1 iteration", it asks "how long does it take to run 10 iterations" (divided by 10). Any variance then relates to the '10 iterations' rather than the 'one iteration'. This approach always requires repetition at a higher level to avoid bias and to form the confidence interval. We would not encourage this practice.

6.1 Independent State

Our first study is to investigate whether DaCapo benchmarks reach independent state. We run three executions of each benchmark with 300 iterations per execution (note that we do not expect researchers to run this many iterations). DaCapo and other Java benchmark suites (such as the SPEC JVM ones) allow iterations to be repeated within a single VM execution. On the other hand, SPEC CPU benchmarks provide only one measurement per execution of a benchmark binary: we address this later.

In the first step, we inspect run-sequence plots (of iteration duration against iteration number), looking for an iteration after which the data seem stable, that is with no regularities or patterns. We always take the maximum of the three executions, but in most benchmarks the executions agree very closely. We discard the unstable prefix. In the second step we check whether the remaining data are statistically independent. If they are, we have found the point at which the benchmark iterations become independent. Otherwise, we conclude that the benchmark does not reach an independent state in reasonable time (running 300 iterations of many of the large-size DaCapo benchmarks takes far more time than is feasible for a particular experiment).

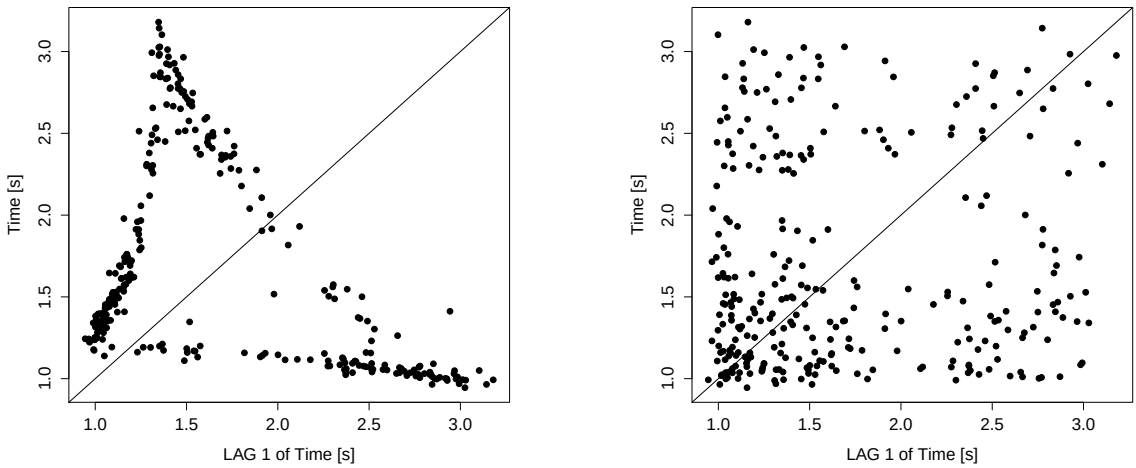
To reduce time, we support this manual process with an interactive R script. In the first step, the user clicks on a run-sequence plot to indicate the point at which the data seem stable (this takes a few seconds per benchmark execution). In the second step, the script displays three plots for each benchmark execution: an auto-correlation function (ACF) plot, a lag plot, and a run-sequence plot with the consecutive measurements connected (details below). Each of these plots can reveal dependencies, and each is offered in two versions — one for the measured data and one for that data randomly reordered. The two versions make the interpretation easier: the experimenter simply looks for a systematic, significant difference between the real and the randomised plots. With some practice this takes less than a minute per benchmark (for all 3 executions).

Lag plots and ACF plots (also called correlograms) are commonly used to detect whether a time-series data set is random or not. Given a series Y, a *lag plot* for a given lag h plots the points (Y_t, Y_{t-h}). Interpretation is easy: any pattern detected in a lag plot indicates some dependency. We check lag plots for lags 1–4, using both iteration order and randomly reordered data (all plotted on one screen by our script). For example, the lag plots in Figure 1(a) show strong auto-dependency in iterations of lusearch9.[1]

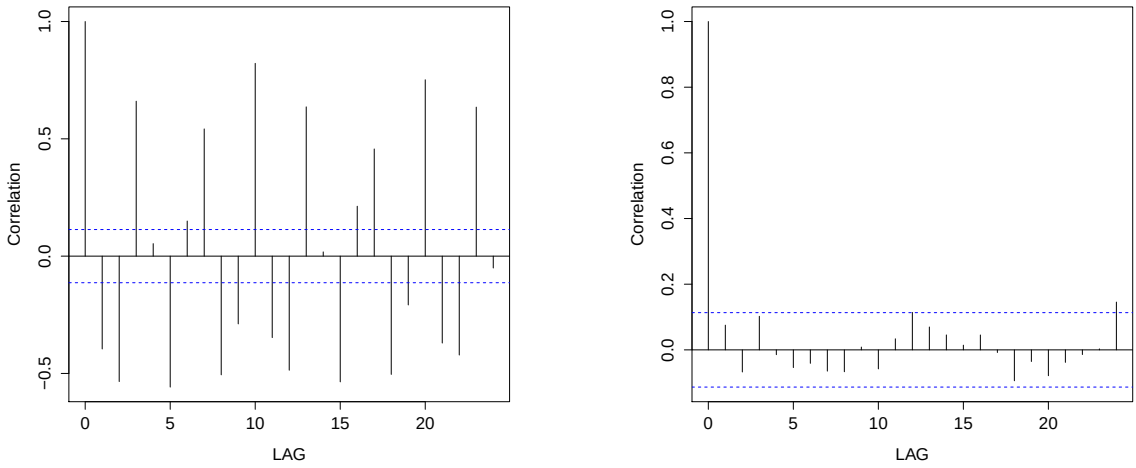
An *auto-correlation plot* shows, for each lag h, the correlation of the series Y_t with its lagged version Y_{t-h}: $cor(Y_t, Y_{t-h}) = E[(Y_t - \mu_Y)(Y_{t-h} - \mu_Y)]/\sigma_Y^2$, where μ and σ^2 are the mean and variance of Y. By definition, the value is always between -1 and 1. It is 1 for lag $h = 0$, but for larger values of h, independent data should have correlations mostly small in absolute value and in the range shown between the horizontal dotted lines (which bound the values expected from random noise) in Figure 1(b). Any systematic structure in the correlations, even if small, is an additional indication of a dependency. Figure 1(b) shows the ACF for the same lusearch9 data as the lag plot.

We applied this method to 'small' and 'large' DaCapo benchmarks on platforms P1 and P2. Table 2 shows which combinations reached an independent state. We found that the (in)dependence patterns agreed in most cases for different benchmark executions. However, results for 'small' sizes disagreed significantly with those for 'large' sizes: clearly one cannot use the 'small' sizes as a shortcut to identify the number of iterations required by 'large' bench-

[1] Lusearch9 stands for lusearch from DaCapo 2009.



(a) Lag plots show how one measurement (x-coordinate) depends on its preceding measurement (y-coordinate). Any pattern in the plot suggests a strong dependency (left figure); a more random scatter indicates no dependency (right figure).



(b) ACF plots show correlation against the x^{th} preceding measurements. Large correlations (outside the dashed lines) or any pattern in the correlations marks a dependency (left figure). In contrast, the right-hand figure shows no dependency.

Figure 1. Iteration durations for lusearch9 (left) and randomly re-ordered (right), large size, Jikes RVM running on P2.

marks to achieve independence. The patterns also disagree between platforms.

Over half the DaCapo/OpenJDK benchmarks reach an independent state. Column 2 of Table 3 shows the number of iterations required for this *independent warmup*. These figures are JVM dependent: Jikes RVM gave different results.

> **RECOMMENDATION: Use this manual procedure just once to find how many iterations each benchmark, VM and platform combination requires to reach an independent state.**

In our previous research, manual inspection of detailed performance data also helped to reveal bugs that did not lead to crashes, thus saving not only experimentation but also debugging time.

6.2 Initialised State

Many benchmarks do not reach an independent state in reasonable time (Table 2). So how should we run these benchmarks? Most have strong auto-dependencies: a gradual drift in times, trends (gradual increase and decrease), state changes (abrupt change in results after some number of iterations), systematic transitions between durations (e.g. odd-numbered iterations show one time and even-numbered ones another), and so on. By choosing which iterations to take, we influence the result significantly (by *tens* of percent). This is problematic for on-line algorithms that often choose different iterations in different runs, platforms or VMs — they use an expensive methodology to distinguish performance differences of only a few percent, but the algorithm incorporates noise many times larger.

> **RECOMMENDATION: If a benchmark does not reach an independent state in a reasonable time, take the same iteration from each run.**

The trends tend to be consistent across runs, and so we would take the first iteration for which each benchmark is initialised, i.e. the largest initialised warmup over all VMs and platforms in our experimental setting. Column 1 of Table 3 shows the initialised warmups of all the 'large' benchmarks on our platforms, established manually through inspection of the first 50 iterations of each execution (from their run-sequence plots). This took only a few seconds per benchmark execution.

Table 2. Independent State in DaCapo/OpenJDK.

	Small		Large	
	P2	P1	P2	P1
avrora9	x	x	x	x
bloat6	-	-	x	x
chart6	-	-	-	x
eclipse6	x	x	x	x
eclipse9	x	x	x	x
fop6	x	-	x	-
fop9	-	-	-	-
h29	x	x	x	x
hsqldb6	-	-	x	-
jython6	x	x	-	-
jython9	-	x	-	-
luindex6	x	x	-	-
luindex9	-	x	x	x
lusearch6				
lusearch9	x	x	x	x
pmd6	-	-	-	x
pmd9	-	-	-	x
sunflow9	-	x	x	-
tomcat9	x	-	-	x
tradebeans9	x	x	-	x
tradesoap9	x	x	-	x
xalan6	-	-	x	x
xalan9	-	-	x	x

x reached independent state
- did not reach independent state

Table 3. Number of Iterations to Warmup DaCapo/OpenJDK.

	Platform P1				Platform P2			
	Initial.	*Indep.*	*Harness*	*Georges*	*Initial.*	*Indep.*	*Harness*	*Georges*
avrora9	2	128	4	1	3	8	3	6
bloat6	2	3	9	∞	2	4	8	∞
chart6	10	88	10	7	3		4	1
eclipse6	3	14	5	11	5	7	7	4
eclipse9	3	9	4	1	2	14	4	1
fop6	6		6	4	10	180	7	8
fop9	6		10	20	6		9	16
h29	0	19	3	0	3	0	4	0
hsqldb6	3		4	1	6	6	8	15
jython6	3		5	2	3		5	2
jython9	3		4	1	3		4	1
luindex6	6		4	48	13		4	8
luindex9	11	19	7	8	10	85	7	8
lusearch9	3	5	5	247	2	37	5	33
pmd6	3	134	4	1	7		4	1
pmd9	3	48	5	2	5		5	3
sunflow9	3		10	∞	0	0	20	∞
tomcat9	5	39	8	8	9		6	8
tradebeans9	2	5	4	5	2		4	1
tradesoap9	3	5	5	1	2		4	1
xalan6	2	2	29	∞	6	13	15	139
xalan9	3	9	8	42	3	31	5	2

6.3 Experimentation Time Savings

Table 3 shows number of iterations for warmup determined by the DaCapo'09 harness and by Georges' method [9]. We show maximums over three runs. We observe that the heuristics do not do very well. There are cases when they give a warmup longer than the independent warmup (e.g. lusearch9 on P1), which would waste experimentation time. In other cases they give a warmup shorter than the initialised warmup (e.g. luindex6 on P2), making any results prone to initialisation noise and hence unusable. This is not to pick particularly on these two heuristics. Automated on-line heuristics attempt to take a decision after a few iterations, as they are designed for real runs. This renders them less reliable than our once per benchmark/JVM/platform manual method where we look at 300 iterations. The heuristics sometimes detect independence too late, but will always waste time on benchmarks that never reach independence.

The second question to ask is, therefore, *how many iterations should be run with benchmarks that do reach an independent state?* We can run a benchmark to independence and then collect a number of iterations, or we can repeatedly run it only to its initialised state and collect one iteration. The former method can save experimentation time if there is higher variation between iterations rather than between executions, the initialised warmup plus the VM initialisation before the first iteration is large, and the independent warmup is small. We examine the tradeoffs between the levels to repeat below.

7. Repeating Executions

The lowest level at which we can repeat a SPEC CPU benchmark is executing a binary; thus the issues are similar to those faced when repeating DaCapo iterations. For DaCapo, the interest is whether there are any random factors that impact results at the execution level. Hence, we discuss repeating executions for these two suites separately.

7.1 Variation in Execution (DaCapo)

We focus on DaCapo benchmarks that reach an independent state by their 11th iteration or sooner. If a benchmark does not reach independence by this time, we simply run it to its initialised state, and take only one live iteration. Our approach is however independent of such a threshold.

To find out if there is random variation exclusively at the execution level, we ran 30 executions of each benchmark, each with 40 iterations, on OpenJDK/P1. We compare the execution variation with the iteration variation (Table 4). By *iteration variation* we mean the variation of iterations within a single execution. By *execution variation* we mean the variation between means of executions. A non-trivial execution variation that is much larger than the iteration variation shows that there are random factors that impact results at the execution level that we need to handle. The variations in Table 4 are normalised by the mean. We define these measures mathematically in Section 9.4 and give more sophisticated estimates in Section 9.2.

Table 4 shows that lusearch9 has very high execution variation (30%) and much higher than the iteration variation (3%). Xalan6 and bloat6 also have high variation at execution level, as does xalan9 to some extent. The execution variations of the remaining three benchmarks are below 0.5%, so we conclude that they do not have significant random variation at the execution level.

7.2 Initialisation, Independence (SPEC CPU)

SPEC CPU benchmarks can run only one iteration per execution. Under SPEC rules each binary should be executed 3 times (5 executions were used in [20]). The benchmarks are quite long

Table 4. Percentage Variation.

	bloat6	*eclipse9*	*lusearch9*	*tradebeans9*	*tradesoap9*	*xalan6*	*xalan9*
Iteration	14.1	0.8	3.3	1.5	0.8	7.0	3.5
Execution	3.7	0.4	30.3	0.4	0.4	9.1	1.0

Measured with DaCapo/OpenJDK on P1.

running, so we first check if repetition is really needed, i.e. if there is any initialisation noise (necessitating warmup) and if the measurements are i.i.d.. For this we ran 30 executions of each 'large' CINT benchmark binary on platforms P3 and P4. On P3, all the measurements from each benchmark were i.i.d. so warmup was unnecessary, allowing us to execute each binary only once. On P4, 10 out of 12 CINT benchmarks had the same nice property.

The exceptions were mcf and gobmk, both of which still can be immediately considered in an initialised state, but then are auto-dependent through at least the first 20 executions (mcf increasing, gobmk decreasing execution times). It would be infeasible with these two benchmarks to reach an independent state as the 20 iterations already take nearly 4 hours (even if they became independent later, which we did not check). As in the case of the DaCapo benchmarks that do not reach independent state in reasonable time, we would just take the first execution in the initialised state, that is the first execution.

Thus, in summary, on our two platforms, it is reasonable to use only one execution of each benchmark binary rather than the default of 3 (2 as warmup), thereby saving about 7 hours of experiment time.

It may be regarded as questionable whether reaching a steady state is even desirable with the SPEC benchmarks. Although the SPEC rules require repetition, the start-up performance of these benchmarks may be closer to their real usage. Our numbers suggest that this discussion is a distraction: based on results on our platforms, only one execution is necessary.

8. Repeating Compilation

If any random variation is due to compilation of our VM, compiler or benchmark, we must repeat the compilation and evaluate multiple binaries. The same applies if performance depends on code layout, in which case we should randomise the layout to avoid bias.

To investigate the performance implications of code layout we patched the `gcc` compiler to randomise the order of functions within each module (source file) compiled, the order of modules compiled/linked and the order of functions globally during link-time optimisations (LTO), a recent feature of `gcc`. We use LTO with the SPEC CPU benchmarks thus randomising their layout fully. OpenJDK does not yet build with LTO, but its modules are linked in large batches, so the layout is substantially randomised by our patch as well. This randomises the VM itself but not the application code's layout, and thus there is no direct runtime overhead.

8.1 DaCapo

To check whether DaCapo/OpenJDK benchmarks are sensitive to code layout, we compared relative variation in 30 executions, each with a different binary, against variation in 30 executions of the same binary on platform P1. We always took the 10^{th} iteration from each execution. Of 24 DaCapo benchmarks, 8 had a variation over binaries larger than that over executions, but the difference was never more than a single percentage point, except for antlr6 where the variation was 9% over binaries and 3% over executions. We also

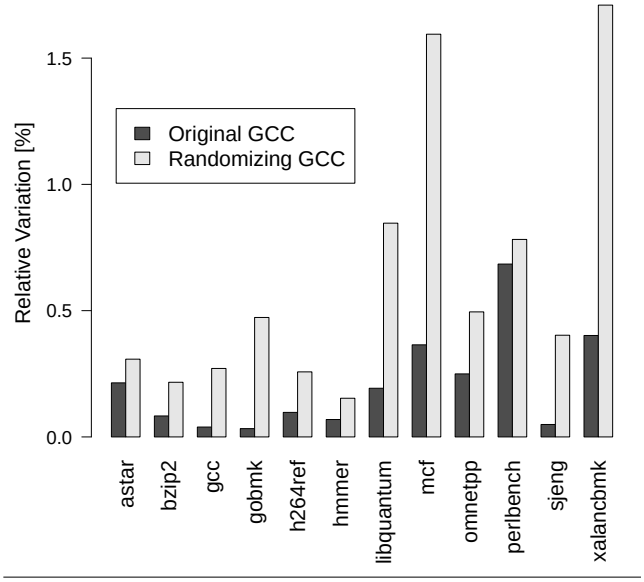


Figure 2. Relative variation with randomising and original `gcc` (reference size).

carried out a similar experiment with Jikes RVM, randomising the layout of the VM classes, but the variation over binaries was only up to one percentage point larger than that over executions.

In summary, while code layout performance impact has been reported for other systems [11, 15, 20], we found little evidence of this problem with DaCapo on our platforms despite looking hard.

8.2 SPEC CPU

We repeated Mytkowicz et al's experiment with link order [20] on platforms P3, P4 and P5 to see if our systems' performances are sensitive to link order. We ran all the 'training' CINT benchmarks, with 30 binaries differing in link order, each binary 5 times, and taking the 5th execution from each. We calculated the relative variation for each benchmark. As a baseline, we repeated exactly the same experiment without link order randomisation. We confirmed the earlier result that nearly all the benchmarks had a higher variation with the link order randomisation than without, and some by a large margin (e.g. xalancbmk 4% vs. 1% and libquantum 1.9% vs. 0.3% on P3). Furthermore, we found that some benchmarks had statistically significantly worse performance with randomised link order (xalancbmk by 4% and libquantum by 2.6%). An experimenter would clearly have to decide if the goal of their study is performance, independent of code layout (as we believe is commonly the case), or the performance of a given layout. Note that the `gcc` compiler would not normally change the code layout (these experiments were performed without LTO).

However, real benchmarking must use reference size benchmarks. Does code layout impact them in the same way? The experiments above cannot be run with reference sizes, because the benchmarks run for too long. However, we showed in Section 7.2 that it suffices to run each binary just once with the reference size. So we ran every benchmark 30 times on platform P3, each time for a different binary. We repeated this once for randomised builds and once for non-randomised (the default) in order to compare the variations again. As we used LTO for our randomised binaries, we also used it with the baseline binary. With each benchmark the variation was higher with the randomised layout (Figure 2), although the variation was quite small overall (1.7% with xalancbmk, 1.6% with mcf and below 1% with all the other benchmarks). Only with

xalancbmk and mcf was the performance with randomisation consistently worse – by 3.3% with xalancbmk and by 6.8% with mcf (both statistically significant).

Nevertheless, repeating with different code layouts is still useful to avoid bias if the goal is to evaluate layout-independent performance. It can be done quite cheaply as there is no need for per-execution warmup. The only added cost is re-compiling a benchmark before execution, which is comparably cheap as these benchmarks are long running. However, the experimenter needs to be aware that layout randomisation can impact performance *consistently*, and not just introduce noise. To find out the case on a given platform, one would have to run initial experiments.

9. Multi-level Repetition

With many experiments (such as those DaCapo benchmarks that reach an independent state), we want to repeat executions and take multiple measurements from each. Adding repetition at the highest level will always increase the precision of the result (narrow the confidence interval). But, in some cases, increasing repetition at a lower level may do so more cheaply. The trade-off is between the degree of variation caused at each level and the cost of repetition. Repetition at the iteration level is cheap (once a benchmark has reached independence, the added cost is just the measured time of that new iteration). Repetition at the execution level is more expensive because, as we add a new execution, we have to wait for it to reach independence — the cost depends on the length of the independent warmup and also on application initialisation before the first iteration (non-trivial in e.g. tradebeans9 and tradesoap9). The cost of repetition at the compilation level is the time to compile, which again can be non-trivial (e.g. OpenJDK takes about 20 minutes to compile on our platforms).

These trade-offs can be formulated mathematically in order to determine the optimum number of repetitions that should be performed at each level to get the most precise result for a given experimentation time. The inputs for this optimisation are the costs (independent warmup, time to compile, etc.) and measurements from an *initial experiment*. From these measurements, one can estimate the variances at the different levels. The outputs are the optimum repetition counts (at all but the highest level) to use for *real experiments* later. As long as the variances in the real experiments are similar to those in the initial experiments, the repetition counts remain optimal. The interval estimation, however, uses only estimates/data from the real experiments, so that the results are sound even if the variances change.

Such a method has been derived for three-level experiments [15]. Here we generalise that method to an arbitrary number of levels and make a technical improvement to the variance estimators. Kalibera and Tuma [15] used statistically biased variance estimators when calculating the repetition counts, and hence the counts established were not optimal (the expected value of a biased estimator of the sample variance is not equal to the true variance of the underlying distribution). Their variance estimator of the sample mean was also biased, making the confidence interval too wide. Too wide an interval would lead to wasting experimental time and increase the chance of failing to detect a true difference in systems. Full proofs and derivations are available in our technical report [14] and more background can be found in McCulloch et al [19] or Searle et al [26]. In Section 9.4 we show how to apply our general method to find optimum repetition counts for DaCapo.

9.1 Initial experiment

Let us consider repetition at levels 1 (the lowest) to n (the highest). First, run an initial experiment. We denote parameters of the initial experiment in sans serif font (e.g. 'r_1') and those for the real experiment in serif font (e.g. 'r_1'). Choose the repetition counts (exclusive of any warm-up iterations needed), $\mathsf{r}_1, \ldots, \mathsf{r}_n$, at each of these levels to be some arbitrary yet sufficient value; 20 may be a good choice but use 30 if possible. If there are many levels for the initial experiment, reduce experimental time by using fewer repetitions (say, 10) at lower levels if you must. It makes sense only to include a level at the top of the hierarchy (n) where you know some repetition is needed. For example, we ruled out the impact of VM code layout on DaCapo benchmarks earlier (on our platform), so we would not include compilation.

Including other levels ($n-1$ and below) is purely for optimisation of the experimental time, as repetition there is never needed for correctness. If including all levels where repetition is possible would be infeasible, design several initial experiments with different inner levels omitted (though always including the highest level), e.g. just compilation and iterations, but not executions. If the optimal repetition count at any level in these partial experiments ends up being 1, it is best not to repeat at that level in the real experiment.

In the initial experiment, gather the costs of repetition at each level, $\mathsf{c}_1, \ldots, \mathsf{c}_{n-1}$, i.e. the time added exclusively by that level. The dimensioning process assumes that these costs do not change much between experiments, which follows our experience. With a 3-level experiment (iterations, executions and compilations), we have:

- c_1 time to get an iteration (iteration duration)
- c_2 time to get an execution (time to independent state)
- c_3 time to get a binary (build time)

In this initial experiment, also take the measurement times, which we denote $\mathsf{Y}_{j_n \ldots j_1}$, where $j_1 = 1 \ldots \mathsf{r}_1$ to $j_n = 1 \ldots \mathsf{r}_n$. These are indexed by the experiment levels (highest to lowest), e.g. $Y_{2,1,3}$ would be the third non-warmup iteration time from the first execution of the second binary in that 3-level experiment.

Calculate arithmetic means of these measurements for different levels. For instance, the mean across experiments at all but the highest level (for which the $j_n{}^{\text{th}}$ repetition is used) is denoted $\overline{\mathsf{Y}}_{j_n \underbrace{\bullet \cdots \bullet}_{n-1}}$, i.e. indexes that vary are denoted by bullets.

9.2 Variance Estimators

After running the initial experiments, we calculate n unbiased variance estimators, $\mathsf{T}_1^2, \ldots, \mathsf{T}_n^2$ from the repetition counts r_i and the measurements $\mathsf{Y}_{j_n \ldots j_1}$. These estimators describe how much each level contributes independently to variability in the result. First, calculate S_i^2, the *biased* estimator of the variance at each level i, $1 \le i \le n$:

$$\mathsf{S}_i^2 = \frac{1}{\prod_{k=i+1}^{n} \mathsf{r}_k} \frac{1}{\mathsf{r}_i - 1} \sum_{j_n=1}^{\mathsf{r}_n} \cdots \sum_{j_i=1}^{\mathsf{r}_i} \left(\overline{\mathsf{Y}}_{j_n \ldots j_i \underbrace{\bullet \cdots \bullet}_{i-1}} - \overline{\mathsf{Y}}_{j_n \ldots j_{i+1} \underbrace{\bullet \cdots \bullet}_{i}} \right)^2 \quad (1)$$

Then obtain each T_i^2 as follows:

$$\begin{aligned} \mathsf{T}_1^2 &= \mathsf{S}_1^2, \\ \forall i \,.\, 1 < i \le n, \mathsf{T}_i^2 &= \mathsf{S}_i^2 - \frac{\mathsf{S}_{i-1}^2}{\mathsf{r}_{i-1}}. \end{aligned} \quad (2)$$

If $\mathsf{T}_i^2 \le 0$ (or at least very small — note T_i^2 denotes an estimator, *not* some value squared), then this level of the experiment induces little variation so repetitions at this level can be removed from the real experiment. This is semantically equivalent to running the initial experiment again with fewer levels.

9.3 Real Experiment: Confidence Interval

Once we have these variance estimators and costs, the optimum numbers of repetitions at levels 1 to $n-1$ for the *real* benchmarking experiments on the same platform are $r_1, \ldots, r_{n-1}$:

$$\forall i \,.\, 1 \le i < n,\ r_i = \left\lceil \sqrt{\frac{\mathsf{c}_{i+1}}{\mathsf{c}_i} \frac{\mathsf{T}_i^2}{\mathsf{T}_{i+1}^2}} \right\rceil . \tag{3}$$

In the real experiment, we use these optimal repetition counts, $r_i, \ldots, r_n$. Note that this formula does not give the optimum repetition count for the highest level. This is because the optima found $(r_1, \ldots, r_{n-1})$ are independent of the number of repetitions chosen at that level. More repetitions can always be added at that level during the real experiment to improve the results' precision and the counts already found will remain optimal.

Recalculate the variance estimator S_n^2 in the same way as before but using the optimal repetition counts and the measurements from the real experiment. Note that although S_n^2 is a biased estimator of the variance at the highest level, it is the right estimator to use in the confidence interval formula (4): technical details can be found in our technical report [14]. Then calculate as before the arithmetic means, $\overline{Y}_{j_n \underbrace{\bullet \cdots \bullet}_{n-1}}$ and $\overline{Y}_{\underbrace{\bullet \cdots \bullet}_{n}}$ (denoted $\overline{Y}$ hereafter).

The asymptotic confidence interval with confidence $(1-\alpha)$ is:

$$\overline{Y} \ \pm \ t_{1-\frac{\alpha}{2},\nu} \sqrt{\frac{S_n^2}{r_n}} = \tag{4}$$

$$\overline{Y} \ \pm \ t_{1-\frac{\alpha}{2},\nu} \sqrt{\frac{1}{r_n(r_n-1)} \sum_{j_n=1}^{r_n} \left(\overline{Y}_{j_n \underbrace{\bullet \cdots \bullet}_{n-1}} - \overline{Y} \right)^2}$$

where $t_{1-\frac{\alpha}{2},\nu}$ is the $(1-\frac{\alpha}{2})$-quantile of the t-distribution with $\nu = r_n - 1$ degrees of freedom.

Observe that, for a single-level experiment, the interval is the standard asymptotic interval based on Student's t distribution (used in most statistical literature and elsewhere [9, 13, 16]). Note also that the multi-level interval is the same as if we had used a single-level interval for the means of all data from all but the highest level (e.g. binary means).

RECOMMENDATION: For each benchmark/VM/platform, conduct a dimensioning experiment to establish the optimal repetition counts (equation 3) for each but the top level of the real experiment. Re-dimension only if the benchmark/VM/platform changes.

9.4 DaCapo Executions vs. Iterations

We applied our optimisation method to suitable DaCapo benchmarks, i.e. those with random variation at the execution level and which reach independence in a reasonable time. In Section 7.1 we identified these as bloat6, lusearch9, xalan6 and xalan9 (on P1). In Section 8.1 we found no random variation at the compilation level with these benchmarks, so we are optimising a 2-level experiment, looking for the optimum number of iterations per execution.

Cost c_1 is the iteration duration: we use the average of all live measurements. Cost c_2 is the time to the first live measurement, which includes VM startup, application startup, and iterations for independent warmup. We take the average from all executions (we instrumented DaCapo to print the current time when an iteration starts, relative to VM start). We show the results in Table 5. We normalise the variance estimates T_1^2 and T_2^2 by the mean iteration duration giving variations $\mathsf{t}_i = \sqrt{\mathsf{T}_i^2}/\overline{\mathsf{Y}}$. They are very similar to the biased estimators $\sqrt{\mathsf{S}_i^2}/\overline{\mathsf{Y}}$ in Table 4, so the less sophisticated method we used for that table worked quite well. Count r_1 is the optimum number of live iteration measurements to take from each of the benchmarks. Note that lusearch9 has an optimum of only 1 measurement: it has a very high execution variation so experimenter time is much better spent repeating whole executions rather than iterations.

Table 5. Optimum Iteration Count with DaCapo.

	c_1[s]	c_2[s]	t_1[%]	t_2[%]	r_1
bloat6	35.5	110.0	14.0	2.7	**10**
lusearch9	1.7	12.3	3.4	30.3	**1**
xalan6	10.8	24.6	7.2	8.9	**2**
xalan9	6.7	71.8	3.5	0.8	**15**

10. Measuring Speedup

Typically, we want to compare two systems, e.g. one with a new optimisation against a base system without, and usually in terms of the ratio of their execution times. Of the 90 papers from our survey that evaluated execution time, 65 reported execution time ratios. Unfortunately, confidence intervals for the execution time ratio are rarely shown in our field (only 3 papers from our survey attempted that). Here we show one way to calculate such intervals and derive how to choose repetition counts.

The interval we show is by Fieller [8] and has been known since the 1950s, though it has not to our knowledge been used before for computer performance evaluation. The calculation of another confidence interval for the ratio has been proposed for computer simulations [17]. That interval is based on the delta method, but their case was sampling from a finite population rather than infinite which we need here. Also, their interval depends on the normal distribution of the ratios (more details in Cochran [4]). The Fieller interval which we show here does not make this assumption. Our technical report [14] includes more details.

10.1 Confidence Interval

A confidence interval for the ratio can be constructed as follows. Let $\overline{Y}$ be the average of all live measurements from the old system and $\overline{Y'}$ that from the new system. We estimate the ratio as $\overline{Y'}/\overline{Y}$. We can calculate an asymptotic $(1-\alpha)$ confidence interval as:

$$\frac{\overline{Y}\cdot\overline{Y'} \mp \sqrt{\left(\overline{Y}\cdot\overline{Y'}\right)^2 - \left(\overline{Y}^2 - h^2\right)\left(\overline{Y'}^2 - h'^2\right)}}{\overline{Y}^2 - h^2} \tag{5}$$

where

$$h = \sqrt{t_{\frac{\alpha}{2},\nu}^2 \frac{S_n^2}{r_n}} \qquad h' = \sqrt{t_{\frac{\alpha}{2},\nu}^2 \frac{S_n'^2}{r_n}}$$

The variance estimators for the old and new system, S_n^2 and $S_n'^2$ are derived as in Sections 9.2 and 9.3. h and h' are the half-widths of the confidence intervals for the single systems (Section 9.3).

10.2 Repetition Counts

We have shown how to establish repetition counts for single systems in isolation. With a little algebra, it can be shown how the relative half-widths of the single-system intervals relate to the half-width of the interval for the ratio. Let e, e' be the relative half-widths of the systems in isolation:

$$e = h/\overline{Y} \qquad e' = h'/\overline{Y'}$$

The half-width of the interval for the ratio, $\mathbf{e}$, is

$$\mathbf{e} = \frac{\overline{Y'}}{\overline{Y}} \cdot \frac{1}{1-e^2} \cdot \sqrt{e^2 + e'^2 - e^2 e'^2}$$

We are normally interested only in narrow intervals for the single systems, say with a half-width below 10%. Hence, we can approx-

Table 6. Suggested Repetitions with DaCapo.

	Warmup	*Iterations*	Executions for precision 1%	1.5%	2.5%	5%
avrora9	2	1	14	18	5	
bloat6	3	10	99	46	18	7
chart6	10	1	8	5		
eclipse6	3	1	33	16	8	
eclipse9	3	1	6			
fop6	6	1	10	6		
fop9	6	1	12	7		
h29	0	1	13	7	5	
hsqldb6	3	1	12	7		
jython6	3	1	15	8	5	
jython9	3	1	10	6		
luindex6	6	1	8	5		
luindex9	11	1	9	6		
lusearch9	3	1	∞	∞	548	139
pmd6	3	1	52	25	11	5
pmd9	3	1	5			
sunflow9	3	1	70	33	14	6
tomcat9	5	1	14	8	5	
tradebeans9	2	1	24	12	6	
tradesoap9	3	1	5			
xalan6	2	2	311	140	52	15
xalan9	9	15	10	6		

imate the second term: $(1 - e^2) \approx 1$. Similarly, $e^2 e'^2 \approx 0$. Hence,

$$\mathbf{e} \approx \frac{\overline{Y'}}{\overline{Y}} \sqrt{e^2 + e'^2} \tag{6}$$

Note that, if we were comparing one system with itself, the relative width of the ratio interval would be $\sqrt{2}$ times wider than the interval for the system itself. If two systems have similar performance and their intervals are no wider than ϵ relative to the mean, then the ratio interval would be no wider than $\epsilon\sqrt{2}$. The better (faster) the new system is, the narrower will be the confidence interval for the ratio.

Finally, and fortunately, this result says that optimising the number of repetitions for a single system, as described in previous sections, also optimises for the ratio of execution times in two systems.

RECOMMENDATION: Always provide effect size confidence intervals for results (equation 4 for single systems or 5 for speedups).

11. Good Repetition Counts

As we have shown, the required and optimum numbers of repetitions depend on the platform, VM, and benchmark.

Table 6 summarises the repetition counts we established for the DaCapo benchmarks on platform OpenJDK/P1. The highest experimental level here is execution — the more executions we take, the narrower confidence interval we get. The table shows approximately how many repetitions would be needed to get a confidence interval with a half-width that is within 1%, 1.5%, 2.5% or 5% of the mean. We do not show counts of fewer than 5 executions as they could hardly be used to get the variance estimate right (the confidence interval uses only the variance estimate at the highest level, so it is fine to have smaller repetition counts at the other levels). If 1000 iterations are not enough for a given precision, the table shows the ∞ symbol. The number of executions (highest-level repetitions) can be established on-line, by adding repetitions until the confidence interval is sufficiently narrow.

Table 7. Suggested Repetitions SPEC CPU.

	Executions = Builds for precision 0.5%	1%	1.5%	2%
astar	4 (5)			
bzip2	4 (5)			
gcc	4 (5)			
gobmk	6			
h264ref	4 (5)			
hmmer	3 (5)			
libquantum	14	6		
mcf	42	13	7	5
omnetpp	7			
perlbench	12	5		
sjeng	6			
xalancbmk	48	14	8	6

Table 7 summarises the repetition counts for SPEC CPU benchmarks on platform P3, compiling with a layout randomising `gcc`. The counts are those required to provide 95% confidence intervals with half-widths of 0.5%, 1%, 1.5% and 2% of the mean; each binary is executed exactly once. Five of the benchmarks are so stable that fewer than 5 executions already give a half-width of 0.5%. We would still run 5 executions of these, though, to get the confidence interval estimate. For half-widths of 1% and higher, we again do not show repetition counts below 5. The benchmarks are much more stable than DaCapo. Xalancbmk and mcf have consistently worse performance with randomisation — if benchmarking of a fixed layout is sought, the variance would be smaller and it would suffice to run 5 executions of both for a 0.5% interval half-width.

12. Summary

Rigorous performance evaluation requires benchmarks to be built, executed and measured multiple times in order to deal with random variation in execution times. Researchers should provide measures of variation when reporting results.

Benchmarks such as DaCapo or SPEC CPU require very different repetition counts on different platforms before they reach an initialised or independent state. Iteration execution times are often strongly auto-dependent: i.e. the benchmark does not reach a steady state, and hence automatic detection of steady state, such as that used in the DaCapo harness or the method recommended by Georges [9], is not applicable. By choosing different iterations in different runs, these heuristics can create an error of tens of percent. We believe that currently proposed or implemented heuristics have proved insufficient to detect independence accurately. We show that manual identification of independence is both necessary and provides a feasible technique, when applied as a one-off analysis for each system. Accurate and robust automation of this inspection is an open problem.

One benefit of our technique is that it made it feasible to repeat earlier experiments on the effect of code layout [20], but using the reference size of SPEC CPU benchmarks. In contrast to the earlier experiments that could use only training sizes, we find the effect of code layout to be small for reference sizes. Similarly, we found no significant impact on the performance of DaCapo benchmarks when we used the `gcc` compiler to randomise the code layout of the HotSpot JVM.

To capture variation, experiments need to be repeated. Experimentation time can be reduced by repetition at multiple levels (e.g.

compilation, execution, iteration) rather than always repeating at the highest level (e.g. compilation). However, there is no need to repeat at a level if the variation introduced by that level is small (e.g. at the compilation level when the effect of code layout is small). We provide a statistically rigorous method that identifies the optimal number of repetitions to perform at each level for a given experimentation budget. Our method saves experimenter time. Although this dimensioning experiment is expensive, it does not need to be repeated unless the system (e.g. benchmark/VM/platform) changes. For most research groups, this investment will be amortised over a few years. We have applied our method to the DaCapo and SPEC CPU benchmarks on several platforms. However, it is essential that experimenters do not use our dimensioning results at face value but apply our method to their systems, where their results are likely to differ.

RECOMMENDATION: Benchmark developers should include our dimensioning methodology as a one-off per-system configuration requirement.

We exhort researchers to report confidence intervals for their results and show how to derive these for experiments repeated at multiple levels, both for single systems and for reporting speedups between systems. Our methods reported here correct, generalise and extend earlier work [15]; a full description and proofs are available in our technical report [14].

Acknowledgements We thank the anonymous reviewers for their thoughtful comments and suggestions which have improved the presentation of this work. We are also grateful to Howard Bowman and to members of the Evaluate Collaboratory for many useful discussions. Finally, we are grateful for the support of the EPSRC through grant EP/H026975/1.

References

[1] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of Java. In *Proceedings of the 17th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2002.

[2] S. Basu and A. DasGupta. Robustness of standard confidence intervals for location parameters under departure from normality. *Annals of Statistics*, 23(4):1433–1442, 1995.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 169–190. ACM, 2006.

[4] W. G. Cochran. *Sampling Techniques: Third Edition*. Wiley, 1977.

[5] R. Coe. It's the effect size, stupid: What effect size is and why it is important. In *Annual Conference of the British Educational Research Association (BERA)*, 2002.

[6] J. Cohen. The Earth is round (p < .05). *American Psychologist*, 49(12):997–1003, 1994.

[7] C. Curtsinger and E. D. Berger. Stabilizer: Statistically sound performance evaluation. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*. ACM, 2013.

[8] E. C. Fieller. Some problems in interval estimation. *Journal of the Royal Statistical Society*, 16(2):175–185, 1954.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2007.

[10] A. Georges, L. Eeckhout, and D. Buytaert. Java performance evaluation through rigorous replay compilation. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2008.

[11] D. Gu, C. Verbrugge, and E. Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance Workshop, OOPSLA*, 2004.

[12] C. Hill and B. Thompson. Computing and interpreting effect sizes. In *Higher Education: Handbook of Theory and Research*, volume 19, pages 175–196. Springer, 2005.

[13] R. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, 1991.

[14] T. Kalibera and R. E. Jones. Quantifying performance changes with effect size confidence intervals. Technical Report 4–12, University of Kent, 2012.

[15] T. Kalibera and P. Tuma. Precise regression benchmarking with random effects: Improving Mono benchmark results. In *Proceedings of Third European Performance Engineering Workshop (EPEW)*, volume 4054 of *LNCS*. Springer, 2006.

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

[17] Y. Luo and L. K. John. Efficiently evaluating speedup using sampled processor simulation. *IEEE Computer Architecture Letters*, 3(1):6–6, 2004.

[18] S. E. Maxwell and H. D. Delaney. *Designing Experiments and Analyzing Data: a Model Comparison Perspective*. Routledge, 2004.

[19] C. E. McCulloch, S. R. Searle, and J. M. Neuhaus. *Generalized, Linear, and Mixed Models*. Wiley, 2008.

[20] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2009.

[21] S. Nakagawa and I. C. Cuthill. Effect size, confidence interval and statistical significance: a practical guide for biologists. *Biological Reviews*, 82(4):591–605, 2007.

[22] K. Ogata, T. Onodera, K. Kawachiya, H. Komatsu, and T. Nakatani. Replay compilation: Improving debuggability of a just-in-time compiler. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*. ACM, 2006.

[23] M. E. Payton, M. H. Greenstone, and N. Schenker. Overlapping confidence intervals or standard error intervals: What do they mean in terms of statistical significance? *Journal of Insect Science*, 3(1996), 2003.

[24] D. Rasch and V. Guiard. The robustness of parametric statistical methods. *Psychology Science*, 46(2):175–208, 2004.

[25] R. M. Royall. The effect of sample size on the meaning of significance tests. *American Statistician*, 40(4):313–315, 1986.

[26] S. R. Searle, G. Casella, and C. E. McCulloch. *Variance Components*. Wiley, 1992.