

Experimental Evaluation Methodology for the Era of No Steady Performance

JAROMÍR ANTOCH, Charles University, Czechia
WALTER BINDER, Università della Svizzera italiana, Switzerland
LUBOMÍR BULEJ, Charles University, Czechia
FRANÇOIS FARQUET, Oracle Labs, Switzerland
VOJTĚCH HORKÝ, Charles University, Czechia
ALEKSANDAR PROKOPEC, Oracle Labs, Switzerland
ANDREA ROSÀ, Università della Svizzera italiana, Switzerland
PETR TŮMA, Charles University, Czechia

Recent studies of virtual machine warm up have pointed out that even small deterministic microbenchmarks executed in tightly controlled circumstances often do not reach a steady state of peak performance. This impacts performance evaluation methodologies that focus on performance after warm up, because the lack of a steady state may violate common assumptions made when computing metrics such as the average performance or the confidence interval for that average.

Our work examines the reported lack of steady state in the context of comparatively larger virtual machine workloads. We document and analyze similar lack of steady state and argue that it should be considered an inherent property of these workloads rather than a fault. We introduce an updated performance evaluation methodology for workloads whose execution exhibits segments of steady state performance separated by sudden performance changes. Using the Renaissance benchmark suite for the Java Virtual Machine, we show that the methodology can produce confidence intervals that miss the true performance over 20 % less often than the existing methodologies.

CCS Concepts: • **General and reference** → **Performance; Measurement; Experimentation**.

Additional Key Words and Phrases: Java, Benchmark, Methodology

ACM Reference Format:

Jaromír Antoch, Walter Binder, Lubomír Bulej, François Farquet, Vojtěch Horký, Aleksandar Prokopec, Andrea Rosà, and Petr Tůma. 2026. Experimental Evaluation Methodology for the Era of No Steady Performance. *Proc. ACM Program. Lang.* 10, OOPSLA1, Article 128 (April 2026), 33 pages. <https://doi.org/10.1145/3798236>

1 Introduction

Performance evaluation of modern virtual machines often relies on executing benchmarks, or similar experimental workloads, whose performance characterizes the system under test. In an idealized performance evaluation scenario, that performance would be repeatable – the benchmark would exhibit the same performance when run repeatedly on the same platform in the same

Authors' Contact Information: [Jaromír Antoch](mailto:antoch@karlin.mff.cuni.cz), antoch@karlin.mff.cuni.cz, Charles University, Prague, Czechia; [Walter Binder](mailto:walter.binder@usi.ch), walter.binder@usi.ch, Università della Svizzera italiana, Lugano, Switzerland; [Lubomír Bulej](mailto:bulej@d3s.mff.cuni.cz), bulej@d3s.mff.cuni.cz, Charles University, Prague, Czechia; [François Farquet](mailto:francois.farquet@oracle.com), francois.farquet@oracle.com, Oracle Labs, Zurich, Switzerland; [Vojtěch Horký](mailto:horky@d3s.mff.cuni.cz), horky@d3s.mff.cuni.cz, Charles University, Prague, Czechia; [Aleksandar Prokopec](mailto:aleksandar.prokopec@oracle.com), aleksandar.prokopec@oracle.com, Oracle Labs, Zurich, Switzerland; [Andrea Rosà](mailto:andrea.rosa@usi.ch), andrea.rosa@usi.ch, Università della Svizzera italiana, Lugano, Switzerland; [Petr Tůma](mailto:petr.tuma@d3s.mff.cuni.cz), petr.tuma@d3s.mff.cuni.cz, Charles University, Prague, Czechia.



This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/).

© 2026 Copyright held by the owner/author(s).

ACM 2475-1421/2026/4-ART128

<https://doi.org/10.1145/3798236>

conditions. Prior work [3, 11, 14, 19, 23, 24, 32] shows that this idealized perspective does not hold in practice. Multiple effects complicate matters and therefore must be understood and accounted for to correctly execute and interpret the benchmark experiments.

One effect known to complicate measurements is warm up, which causes the initial benchmark performance to differ from the performance sustained over continuous execution. In managed environments, an obvious technical factor contributing to warm up is just-in-time compilation, which trades the initial expense of loading, profiling and compiling the application code for the ensuing benefit of executing optimized machine code. Another factor specific to managed environments is garbage collection. It typically requires multiple collection cycles to gradually tenure long lived objects and thus collect young objects more efficiently. Still other factors may include demand loading (of data or code) or priming of various caches at the hardware, system or application level.

To facilitate performance evaluation in the presence of warm up, benchmark workloads progress past warm up while repeating the same computation multiple times. The performance of each repetition is reported individually so that the measurements from the cold repetitions can be discarded in an experiment that is to investigate sustained performance.

Another effect that complicates measurements is run variability. Some aspects of the benchmark workload execution, such as the exact relative timing of concurrent operations, do not lend themselves to sufficient control to achieve entirely equivalent benchmark runs. Besides leading directly to different performance of individual workload repetitions, execution differences can also influence feedback-directed just-in-time compilation and therefore expand their impact to future workload repetitions that use the compiled code. Other resource allocation decisions that happen only once per run may have similar effect, impacting each run in a potentially different manner that mere workload repetition in one run can not characterize.

An accepted solution to run variability is an experiment design with multiple runs, each with fresh initialization, compilation and resource allocation decisions, and a corresponding statistical evaluation. A procedure described by Georges et al. [11] advocates averaging over measurements collected within each run and then computing a confidence interval over these averages. Kalibera et al. [23] present a more complex procedure with confidence intervals that take into account both variability of measurements within each run and variability between runs. The procedure is further extended by Kalibera et al. [22, 24, 26] with better variance estimates and an arbitrary depth of experiment structure. A similar procedure is used by Bakshy et al. [2] to compute standard error in experiments structured into batches collected across multiple hosts.

Among the complications not yet addressed is the observation made by Barrett et al. [3] in their detailed study of virtual machine warm-up behavior, which shows that even small deterministic benchmark workloads may not exhibit steady performance after warm up. The authors note that with such workloads, the widely used steady-state detection method from Georges et al. [11] incorrectly identifies steady state in most workloads without one. Furthermore, the methodology of Georges et al. [11] assumes a workload always reaches a steady state, even if it may take a very long time, which is an assumption the work of Barrett et al. [3] potentially contradicts. The more recent methodology by Kalibera et al. [24] accepts that a workload may not reach a steady state, but then recommends using the same single sample from each run, which is not efficient and may introduce bias.

In this paper, we argue that lack of steady performance may simply be a fact of life with workloads on modern virtual machines. We develop a new methodology that accounts for this fact – rather than evaluating performance under the common steady-state assumption, we model the workload

as a random process that transitions between segments of steady-state performance, and provide a method for summarizing such performance using confidence intervals. Finally, we demonstrate the benefits of the model in a realistic performance evaluation scenario. In specific terms, our contributions are:

- We extend the experiment of Barrett et al. [3] to include significantly larger code base and significantly longer run times, using the Renaissance benchmark suite [35] for the Java Virtual Machine. Larger code and longer run times provide additional information on the lack of steady performance beyond the scope of the original experiment.
- We use a technical analysis of a selected performance change to illustrate why we believe that achieving steady performance after warm up may not be a useful ambition, and propose an updated evaluation methodology to reflect this situation.
- We use simulation based on realistic statistical distributions of performance to quantify the difference in accuracy between the existing and updated evaluation methodologies, and to expose the impact of warm up detection mechanisms on the optimum experiment dimensions. For analytical computation, we show that the updated methodology produces confidence intervals that miss the true performance up to 20 % (normal quantiles) or 23 % (studentized quantiles) less often than the existing methodologies. For resampling, we show an improvement of up to 24 %, with the confidence intervals on average only 5.6 % wider than the ground truth data.

We proceed by investigating the long term behavior in the context of the Renaissance benchmark suite in Section 2, and delve into the technical reasons behind the observed behavior in Section 3. We present our updated statistical model in Section 4 and evaluate the accuracy of the confidence intervals based on the model in Section 5. Section 6 delves into the practical implications of warm up detection on the experiment dimensions. After summarizing the methodological recommendations in Section 7, we close the paper with a look at the threats to validity in Section 8, related work in Section 9, and conclusion in Section 10. Our data and scripts are also available in a reproducibility package online.

2 Analyzing Long Runs Performance of the Renaissance Suite

In order to avoid confounding the virtual machine behavior with workload idiosyncrasies, the study of Barrett et al. [3] deliberately relied on simple workloads. The bulk of the work used 6 benchmark workloads implemented in multiple programming languages, originally from the Computer Language Benchmarks Game project [13], with the Java variants (without harness) totaling 1500 LOC with no dependencies outside the Java Class Library. It is conceivable that some of the reported effects are more pronounced with smaller workloads, where the contribution of individual execution mechanisms to the overall workload performance tends to be relatively larger, we therefore reproduce the evaluation of Barrett et al. [3] with Renaissance. The Renaissance workloads (without harness) total around 69 000 LOC with 420 MB of external dependencies.¹

In their study, Barrett et al. [3] executed each workload with 2000 repetitions per run and 30 runs, runs typically took 200 s to 2000 s (3 min to 30 min). To adjust these dimensions for Renaissance, we refer to the continuous monitoring of the Graal compiler performance [7], where some workloads take more than 5 min to get past the initial burst of compilation activity. Furthermore, some Renaissance workloads rely on frameworks that may schedule disruptive activities with periods in tens of minutes, such as the 30 min interval for the Apache Spark Context Cleaner garbage

¹Another popular benchmark suite for the Java Virtual Machine is DaCapo [6], however, at the time of collecting the initial measurements, the suite was not updated for over a decade. We have considered adding DaCapo to this study after the recent releases, however, our brief experiments, available in the reproducibility package, indicated the results would be similar to Renaissance. Note that just the initial Renaissance measurements consumed around 3 years of bare metal hardware time.

collection.² To make sure we collect enough measurements to cover such long durations, we use 28 800 s to 30 600 s (8 h to 8.5 h) runs.³ We collect this and all subsequent measurements on dedicated Intel Xeon machines running at fixed 3.5 GHz with 32 GB DDR4-2666 RAM running Fedora Linux, Oracle Java 11 and Renaissance 0.14.1, with the usual precautions for measurement stability. A detailed configuration information is in the reproducibility package online.

Table 1. Workload classification with categories from [3]. The *Repetitions* columns give the average repetitions executed per run and the average repetition where steady state was reached. The *(Not) Steady* columns give the share of runs classified into their respective categories.

Benchmark	Runs	Repetitions		Steady			Not	Overall
		Total	Steady	Flat	W-Up	S-Dn	Stdy	
akka-uct	78	4230	0	96%	4%	0%	0%	good inconsistent
als	75	8712	5861	0%	21%	71%	8%	bad inconsistent
chi-square	78	32686	11502	0%	47%	53%	0%	bad inconsistent
db-shootout	77	6724	0	100%	0%	0%	0%	flat
dec-tree	77	39222	7420	0%	62%	38%	0%	bad inconsistent
dotty	78	24474	4811	0%	100%	0%	0%	warmup
finagle-chirper	75	15310	3222	0%	99%	1%	0%	bad inconsistent
finagle-http	77	9794	3567	8%	39%	53%	0%	bad inconsistent
fj-kmeans	80	10231	1033	0%	70%	30%	0%	bad inconsistent
future-genetic	76	13877	7950	0%	12%	88%	0%	bad inconsistent
gauss-mix	79	24504	6276	0%	78%	19%	3%	bad inconsistent
log-regression	74	55912	14651	0%	84%	14%	3%	bad inconsistent
mnemonics	78	8534	6059	0%	51%	31%	18%	bad inconsistent
movie-lens	77	4248	2508	5%	1%	88%	5%	bad inconsistent
naive-bayes	77	10670	7033	0%	23%	68%	9%	bad inconsistent
neo4j-analytics	77	9352	4542	5%	44%	48%	3%	bad inconsistent
page-rank	75	5399	105	92%	8%	0%	0%	good inconsistent
par-mnemonics	79	8743	4302	0%	53%	43%	4%	bad inconsistent
philosophers	78	24914	15544	0%	19%	81%	0%	bad inconsistent
reactors	78	3029	23	95%	4%	1%	0%	bad inconsistent
rx-scrabble	77	105792	9258	5%	44%	51%	0%	bad inconsistent
scala-doku	76	5663	2794	0%	0%	97%	3%	bad inconsistent
scala-kmeans	78	73562	31242	3%	45%	53%	0%	bad inconsistent
scala-stm-bench7	78	33637	14210	0%	14%	81%	5%	bad inconsistent
scrabble	75	64287	0	87%	13%	0%	0%	good inconsistent

The results of the experiment are summarized in Table 1. For reader convenience, we recapitulate the classification categories introduced by Barrett et al. [3] – a workload run is said to reach *steady state* if there was no significant change in performance of the last 500 repetitions, that steady state is further classified as *flat* if there was no significant change in performance before the last 500 repetitions, *warmup* if performance in the last 500 repetitions is never worse than in the preceding repetitions, and *slowdown* otherwise. Across multiple runs, a workload is said to behave

²<https://spark.apache.org/docs/latest/configuration.html>

³Not using exactly 8 h prevents inadvertent synchronization with system services that may operate on daily basis.

inconsistently if the classification among the runs differs, that inconsistency is denoted as *good inconsistency* if all runs are either flat or warmup and *bad inconsistency* otherwise.



Fig. 1. Ad hoc examples of workload performance. Colors distinguish individual runs, a random sample of 1000 points per run is shown to reduce clutter, the top measurement percentile is removed to improve scale. Note how some runs exhibit sudden changes in performance hours into the run.

The interpretation of Table 1 requires some care, because some of the parameters of the automated classification used by Barrett et al. [3] were tuned to different workloads. While most runs are classified as eventually reaching steady state, the average repetition where steady state is reached is often relatively high, suggesting that the steady state (defined as a period of stable performance covering the last 500 repetitions) can sometimes represent only a relatively short and possibly transient part of the run (this is also illustrated in Figure 1, where some runs exhibit long periods of stability that nonetheless do not reach the end of the run). The overall classification indicates that with most workloads, reaching the steady state does not mean reaching the peak performance.

Long Runs Summary

The observations by Barrett et al. [3], which show that simple workloads often do not reach stable performance even after minutes of execution, also apply to hours of execution of the more complex workloads from the Renaissance suite. In some workloads, the runs exhibit *sudden changes* in performance *hours* into the run, including changes that regress from peak performance.

3 Investigating Disruptions to Stability

When a benchmark workload serves to assess performance, it is evidently desirable that performance after warm up reaches steady state. In fact, this intent is reflected in the workload design, where the individual repetitions typically perform the same computation on the same data, and measures such as stable random generator seeds are employed to remove potential variability. From this perspective, the behavior documented in Table 1 and Figure 1 can be considered faulty and warrant investigation. Here, we detail such investigation for one particular change in performance, which often appeared in our measurements at around 12 100 s (3.4 h) of execution. The change is illustrated in Figure 2.

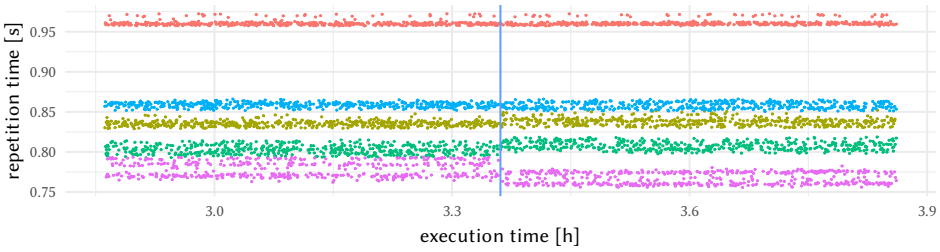


Fig. 2. Performance of the *chi-square* workload with change in some runs at around 12 100 s (3.4 h) of execution. Colors distinguish individual runs, a random sample of 1000 points per run is shown to reduce clutter, the top measurement percentile is removed to improve scale. Vertical line highlights the performance change.

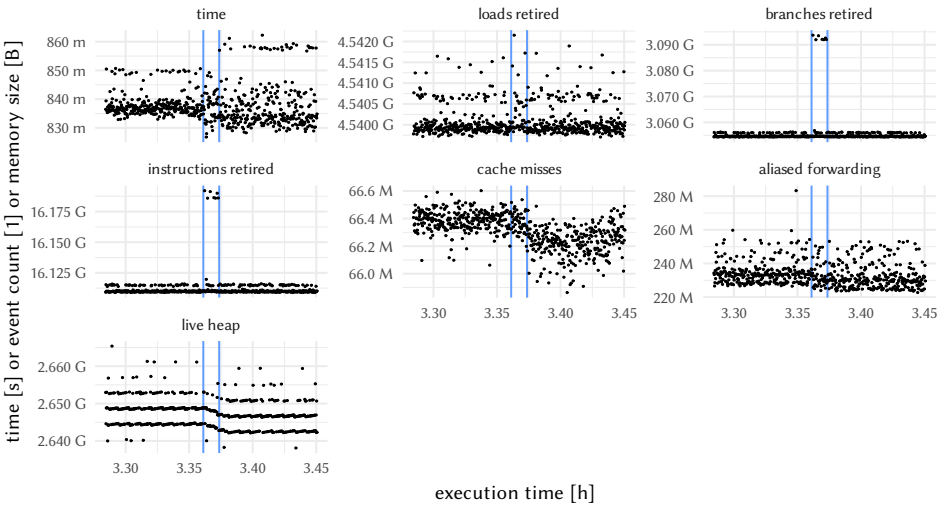


Fig. 3. A selection of metrics associated with a single run of the *chi-square* workload with performance change at around 12 200 s (3.4 h) of execution. Vertical lines indicate the start and the end of the disruptions associated with the performance change.

Our investigation proceeds in steps as follows:

- (1) We check whether the change is associated with the usual culprits such as dynamic compilation (we record the count of compilation events and the processor time consumed by dynamic compilation during each repetition) or garbage collection in response to memory shortage (we record the count of garbage collection events during each repetition and force garbage collection between repetitions). No extra dynamic compilation event and no extra garbage collection cycle occurs regularly around the time of the change.
- (2) Ruling out the usual culprits, we collect indicators to direct further investigation. Using a manual variant of vertical profiling [16], we collect the values of multiple system metrics, including the attributes exported by the standard JMX interfaces of the virtual machine and the available hardware performance counters, and examine whether these metrics change together with performance. Selected metrics are shown in Figure 3.

- (a) The hardware performance counters associated with static workload parameters, such as the counts of various retired instruction types, exhibit no durable shift, but may show a short burst of increased activity around the time of the performance change.
- (b) The hardware performance counters associated with the efficiency of miscellaneous processor optimization mechanisms, such as the count of last level cache misses, or the count of store forwarding events with aliased addresses, exhibit a shift around the time of the performance change. The magnitude and direction of the individual metric shifts does not necessarily correspond to the magnitude and direction of the performance change, but the aggregate effect of multiple shifts may.
- (c) The JMX attribute associated with the total live heap size fluctuates with an approximately regular period of around 25 s, and the average live heap size over the entire period exhibits a consistent drop of about 834 kB (0.03 %) around the time of the performance change.

Together, the metrics indicate a situation where the same code exhibits different performance after a change in data layout, similar to observations by Kalibera et al. [23] or Mytkowicz et al. [32].

- (3) Suspecting data layout changes, we examine heap-related activity in detail. An analysis of a verbose garbage collection log shows that the change in the total live heap size is associated with a change in the metaspace size, which shrinks by about 405 kB but is otherwise stable across multiple workload repetitions. Metaspace is a special part of the virtual machine heap used to store information related to loaded classes.
- (4) Investigating classloading activity in the verbose classloading log shows that the change in metaspace size is associated with unloading of multiple classes generated internally by the virtual machine to accelerate reflection. These classes are generated in the initial stages of the benchmark workload and are kept alive through soft references.
- (5) Objects kept alive using soft references are cleared by the garbage collector during regular garbage collection cycles using a platform-specific strategy. In our configuration, the default behavior is to keep objects alive for certain time from the last use of the reference. That time is computed from the maximum heap size, with 1 MB corresponding to 1 s. The measurements used 12 GB maximum heap size, which translates to a little bit over 12 100 s, the time of the observed performance change. The causal relationship can be confirmed by changing the maximum heap size, which duly shifts the time of the associated performance change.

The analysis offers several takeaways. Perhaps most notably, the performance change in question is caused by a combination of factors ranging from relatively high level actions (such as the use of reflection) through virtual machine implementation details (such as the internal logic of soft reference handling) down to machine code behavior (such as the sensitivity of memory access latencies to data-layout minutiae). This requires collecting and analyzing more than the usual amount of information, which may be expensive (more runs may be needed to collect everything) or even impossible (tools may not work or may change observed performance).

Hinging on a long sequence of causes, the performance change may also disappear (or appear) with arguably non-essential workload or platform changes, making the analysis results singular rather than general. A precise understanding of the underlying causes of one performance change may only have a limited value when considering other performance changes, other workloads or platforms.

Finally, the mechanisms that contributed to the performance change in our example are present in common software systems. Although adjusting the workload or the platform to prevent certain performance effects might be possible [4, 9, 12, 29, 34], this may make the observed performance

less representative of such software systems and thus defeat the very purpose of evaluating the benchmark workload performance.

Stability Disruptions Summary

Analyzing and (possibly) eliminating the individual performance changes that violate the steady state *may incur high expenses for low returns*. Instead, the performance evaluation methodologies methods should support *analyzing the observed performance in presence of these performance artifacts*.

4 Statistical Model with Steady-State Shifts

The existing performance evaluation methodologies recommend reporting performance with confidence intervals [11, 24], but the prescribed computation does not support systems that exhibit performance artifacts illustrated in Figure 1. To facilitate evaluation of such systems, we first define a statistical model of the benchmark workload performance, and then use the model to derive and validate suitable confidence interval computation methods. In the model, we assume that the system under test exhibits segments of steady state performance associated with specific execution conditions. The segments are separated by moments when the execution conditions change, notably at the start of each run and possibly at random times during each run, as illustrated in Figure 1.

More formally, we approximate the benchmark workload performance by a non-stationary random process, where the baseline workload performance after warm up is modulated by the impact of the execution-segment conditions, changing at independent random moments during every run with a constant arrival rate, and the impact of the initial run conditions, changing once at the start of every run. Both impacts are modeled as simple additive shifts of the baseline.

- We use indices r , s and m to denote runs, execution segments within a run, and measurements within an execution segment, respectively.
- Let R_r be iid (independent identically distributed) random variables that describe the impact of the initial run conditions of run r – the **run effect**, with **run variance** $Var(R)$ quantifying the variability *between* different runs.
- Let $S_{s(r)}$ be iid random variables that describe the impact of the execution-segment conditions of segment s of run r – the **segment effect**, with **segment variance** $Var(S)$ quantifying the variability *between* segments *within* a single run.
- Let $B_{m(rs)}$ be iid random variables that describe the workload performance observed in measurement m of segment s of run r – the **baseline performance**, with **baseline variance** $Var(B)$ quantifying the inherent variability between individual measurements *within* a single segment.
- We use λ to denote the segment arrival rate.

Each measurement X_{rsm} is then expressed as a sum of the baseline performance and the modulating conditions:

$$X_{rsm} = R_r + S_{s(r)} + B_{m(rs)} \quad (1)$$

A measurement experiment performs N_R runs, with $N_{S(r)}$ segments in run r , and $N_{M(rs)}$ measurements in segment s of run r . The values of $N_{S(r)}$ and $N_{M(rs)}$ are controlled indirectly through run duration and are related by the segment arrival rate λ . In this model, we compute the sample grand mean \bar{X} as an estimate of the average workload performance, using a shorthand notation for summations where $\sum_{rsm} \equiv \sum_{r=1}^{N_R} \sum_{s=1}^{N_{S(r)}} \sum_{m=1}^{N_{M(rs)}}$:

$$\begin{aligned}\bar{X} &= \frac{1}{N_R} \sum_r \bar{X}_{R(r)} & \bar{X}_{R(r)} &= \frac{1}{N_{S(r)}} \sum_s \bar{X}_{S(rs)} & \bar{X}_{S(rs)} &= \frac{1}{N_{M(rs)}} \sum_m x_{rsm} \\ \bar{X} &= \frac{1}{N_R} \sum_r \left(\frac{1}{N_{S(r)}} \sum_s \left(\frac{1}{N_{M(rs)}} \sum_m x_{rsm} \right) \right)\end{aligned}\quad (2)$$

Without loss of generality, we assume the run and segment effects center around zero, $E[R] = E[S] = 0$. We then have $E[\bar{X}] = E[B]$, in other words the grand mean intuitively estimates the baseline performance.⁴

Using the basic properties of variance, we can also compute the variance of the grand mean $Var(\bar{X})$. The equation relies on the iid properties of R , S and B , and for simplicity assumes $N_{S(r)}$ and $N_{M(rs)}$ to be the same in each run (N_{SR}) and segment (N_{MSR}):

$$Var(\bar{X}) = \frac{Var(R)}{N_R} + \frac{Var(S)}{N_R \cdot N_{SR}} + \frac{Var(B)}{N_R \cdot N_{SR} \cdot N_{MSR}} \quad (3)$$

In principle, our statistical model composes the contributions of the run effects, the segment effects, and the baseline performance in a manner similar to other hierarchical models applied in earlier performance evaluation methodologies [11, 23, 24], and, also in a manner similar to the earlier work, can be used to compute the asymptotic confidence interval for the grand mean. In our evaluation, we therefore only focus on the particularities of our statistical model that were not present in earlier work.

5 Evaluating Confidence Interval Accuracy

Central to our empirical evaluation is the question of confidence interval accuracy. Our statistical model from Section 4 differs from the existing hierarchical models in considering the steady-state shifts between the individual execution segments, we therefore look at whether the confidence intervals derived using our model are more accurate than confidence intervals derived using the existing models by Georges et al. [11] and Kalibera et al. [24], which use the sample mean to characterize the performance of each run, and then use the variance of the sample means to derive the confidence interval of the grand mean.

In principle, assessing the accuracy of the confidence intervals entails running a large number of experiments with known baseline performance and checking the confidence intervals constructed from measurements against this baseline. However, this is not possible in realistic experimental conditions, where the baseline performance is not known and estimating it with sufficient accuracy is likely to be prohibitively expensive – with the experiments we consider in the context of Renaissance, we are talking about executing workloads in multiple runs of several hours in each of many experiments.

To work around the experimental constraints, we base our evaluation on simulation, similar to Bakshy et al. [2]. Prior to the evaluation, we use extensive measurements in realistic conditions to collect the performance parameters of the individual benchmark workloads. In the evaluation, we use these parameters to generate data for a large number of simulated measurement experiments. Because we generate data using a stochastic process with known parameters, we can assess what

⁴The sample grand mean computation uses no weights, making all runs and all segments equally important. Statistically efficient alternatives under standard independence assumptions include weighing by sample count or weighing by inverse variance [33]. Equal weights are appropriate when the run and segment durations are incidental to the estimated performance. When sample sizes are not independent of sample values, weighting can introduce bias – this can occur for example when each run executes for the same amount of time and thus runs that observe higher sample values produce fewer samples.

proportion of the confidence intervals covers the known baseline performance and thus evaluate the confidence interval accuracy.

In the statistical model from Section 4, the benchmark workload performance is characterized by distributions B (baseline), S (segment effect) and R (run effect), and the segment arrival rate λ . To recover these characteristics from the preparatory measurements, we perform the following steps:

Collecting measurements. We collect enough measurements to reasonably characterize the benchmark workload performance on the system under test. Specifically, we want to capture the B , S and R distributions with sufficient accuracy to preserve practically relevant features such as modes, and aim to observe potential steady-state shifts with periods of up to hours. The conservative bounds derived by Massart et al. [30] inform us that 66 observations are enough to reconstruct a distribution function with at most 20 % error with 99 % probability. The exponential distribution similarly suggests 4.6 h suffices to observe a random steady-state shift with an arrival rate of 1 h^{-1} with 99 % probability. Combining both and adding some reserve, we settle at 25 days of measurements of each workload, split into 8 h runs (with less than 5 % difference in individual workloads due to assorted experiment interruptions).

Removing warm up data. We drop measurements from the first 60 min of each run. The step is designed to move beyond typical warm up with a large margin, and avoid more complex warm up detection mechanisms that could interact with other measurement properties and therefore introduce confounding factors. Again, in practice, performance of a repeating workload should be reasonably relevant after that much time.

Removing outliers. In a simple sliding window, we drop measurements that are below the 5 % quantile or above the 95 % quantile by more than 10 % of the inter-quantile range. The step is designed to facilitate subsequent change detection, which is sensitive to outliers [3], the exact parameters were chosen through visual inspection.

Checking trends. We check that there is no overall trend in the measured performance. We use bootstrap [10] over runs to compute the 99 % confidence interval for the slope of a least squares linear regression between the duration from the start of the run and the measured performance, and check that the interval includes zero.

Identifying execution segments. We apply the PELT algorithm [27] to detect changes in mean or variance, in a manner similar to that of [3], except for the choice of the penalty. The formula for the penalty described by Barrett et al. [3] yielded penalty values that were, on our data, on average about 3 times higher than penalties chosen through visual inspection. Unlike Barrett et al. [3], who considered many different systems under test, we could afford to choose the penalties through visual inspection for each individual workload (by identifying a point just after the knee on the penalty-vs-segment count curve).

Approximating distributions. We construct the empirical distributions of baseline performance \hat{B} , segment effects \hat{S} and run effects \hat{R} from the individual measurements, segment means and run means, respectively. Finally, we apply the Gaussian kernel density smoothing with bandwidth determined using UCV [36].

Figures 4 to 6 show examples of the \hat{B} , \hat{S} and \hat{R} distributions constructed from the workload measurements on Figure 1 (while that figure shows only some measurements to reduce clutter, all measurements were used to construct the distributions).

With the parametrized statistical model for each benchmark workload, we generate artificial data that approximate measurements in realistic conditions simply by drawing from the three distributions, B , R and S , with the number of samples determined by the experiment dimensions and the segment arrival rate λ . An example of the generated artificial data for the three workloads from Figure 1 are in Figure 7.

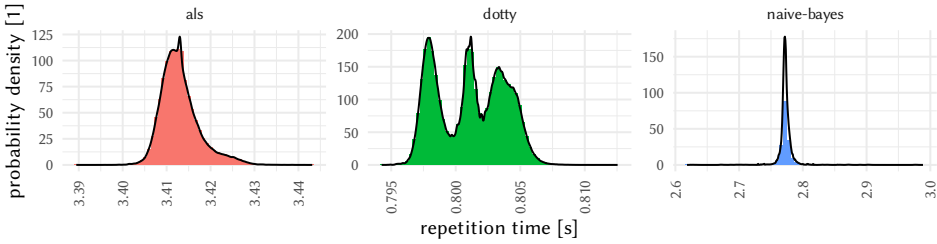


Fig. 4. Empirical distributions of the baseline workload performance \hat{B} for the workloads from Figure 1.

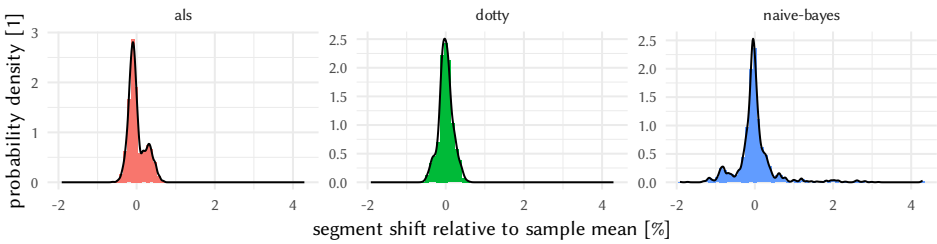


Fig. 5. Empirical distributions of the segment effects \hat{S} for the workloads from Figure 1.

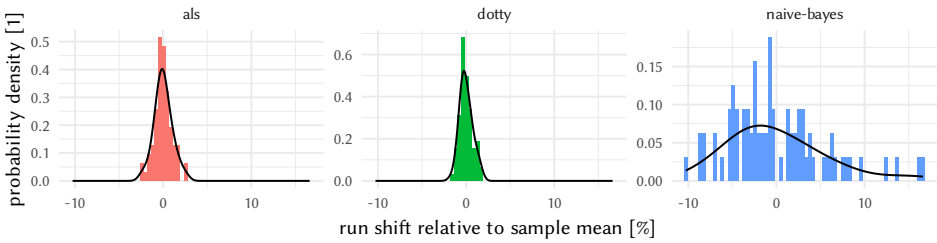


Fig. 6. Empirical distributions of the run effects \hat{R} for the workloads from Figure 1.

We conclude the evaluation by generating a large number of artificial measurements and using several methods to compute a 99% confidence interval for the grand mean across these measurements. Then, we look at the average interval width and at how often the intervals miss the true baseline performance, expressed as a miss rate.

Because the simulation is intended to assess the confidence interval accuracy in realistic experiments, we generate measurements approximating three realistic experiment dimensions – a short experiment, with a total duration of 1 h and 2 min warm up, a medium experiment, with a total duration of 8 h and 5 min warm up, or a long experiment, with a total duration of 24 h and 10 min warm up. Other generated experiment parameters – the benchmark workload and the run and sample count – are then drawn from a uniform distribution to fit the dimension. Note that the accuracy of the statistical estimates made by the respective confidence interval computation methods usually depends on the amount of data available, the mix of experiment dimensions in the generated measurements may therefore influence the overall standing of each method.

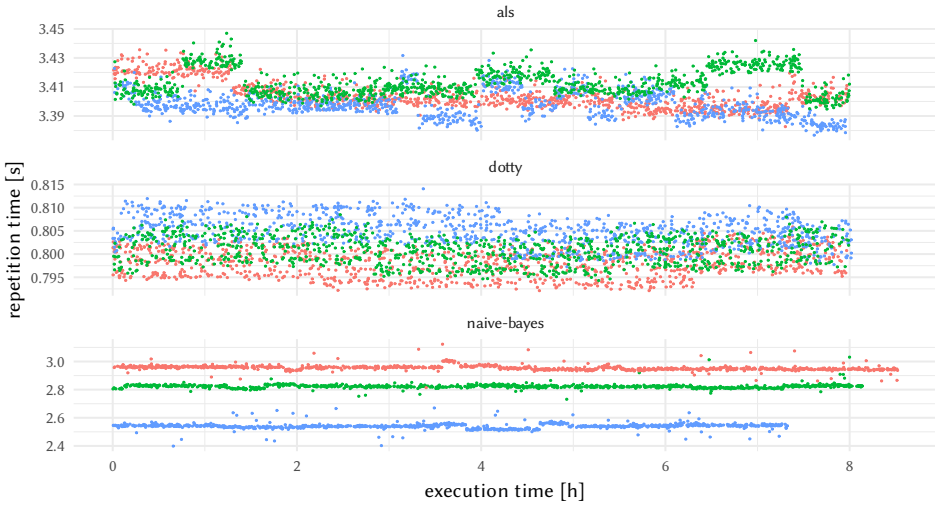


Fig. 7. Artificial data generated by simulating the performance of workloads from Figure 1 using the statistical model from Section 4 and the distributions from Figures 4 to 6. Colors distinguish individual runs, a random sample of 1000 points per run is shown to reduce clutter.

5.1 Analytical Computation Methods

For analytical computation methods, we generate enough measurements to distinguish miss rate changes of 1‰ to 2‰, which is around 250 000 simulated experiments per benchmark workload. The evaluated methods are:

Analytical Computation with Runs Only. Following [11, 24], we estimate the baseline performance using a two-stage average – we first average measurements within each run (ignoring segments), and then average these run means across runs. We use the sample variance of the run means to construct the confidence interval for the baseline performance estimate. For the exact formulas, see Appendix A.1. We refer to this computation method as *segment ignorant* in further text and with the *Run* acronym in formulas and tables.

Analytical Computation with Segments and Runs. Following Equation (2), we estimate the baseline performance using a three-stage average – we first average measurements within each segment, then average segment means within each run, and finally average run means across runs. Following Equation (3), we estimate $Var(B)$ from the pooled variance of individual measurements around the segment means, $Var(S)$ from the pooled variance of segment means around the run means, and $Var(R)$ from the variance of the run means. We then use the combined variance from Equation (3) to construct the confidence interval for the baseline performance estimate. For the exact formulas, see Appendix A.2. For brevity, we refer to this computation method as *segment aware* in further text and with the *Seg* acronym in formulas and tables.

Normal and Studentized Quantiles. For the mean and variance estimates from both methods, we compute the confidence interval for the mean performance at confidence level γ as the $\frac{1-\gamma}{2}$ and $\frac{1+\gamma}{2}$ quantiles of a normal distribution with the same mean and variance, and a corresponding Student’s t distribution with $N_R - 1$ degrees of freedom.

Table 2. Miss rates and confidence interval widths for 99% confidence intervals computed using normal quantiles with segment-ignorant and segment-aware methods. Bold deltas are statistically significant using Wilson proportion test for rates and paired Mann–Whitney–Wilcoxon rank sum test for widths. Benchmarks with notable results only, see Appendix B for a complete table.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	21.87 (21.1-22.6)	26.16 (25.4-27.0)	$\times 1.20$	30.69	29.81	-0.88
akka-uct	19.79 (19.1-20.5)	22.04 (21.3-22.8)	$\times 1.11$	16.82	16.28	-0.55
future-genetic	20.1 (19.4-20.8)	21.64 (20.9-22.4)	$\times 1.08$	20.48	20.26	-0.22
scrabble	20.24 (19.5-21.0)	21.27 (20.5-22.0)	$\times 1.05$	47.71	47.47	-0.24
movie-lens	36.73 (35.8-37.7)	37.6 (36.6-38.6)	$\times 1.02$	17.08	16.99	-0.09
reactors	23.33 (22.6-24.1)	23.77 (23.0-24.6)	$\times 1.02$	18.45	18.32	-0.13
...						
scala-kmeans	187.4 (185.4-189.4)	187.38 (185.4-189.4)	$\times 1.00$	91.32	91.33	+0.01
...						
dotty	21.92 (21.2-22.7)	21.78 (21.0-22.5)	$\times 0.99$	9.16	9.18	+0.01
log-regression	17.06 (16.4-17.7)	16.91 (16.3-17.6)	$\times 0.99$	27.01	27.03	+0.02

Table 3. A variant of Table 2 computed using studentized quantiles.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	10.09 (9.6-10.6)	12.38 (11.8-13.0)	$\times 1.23$	41.67	40.50	-1.17
akka-uct	8.53 (8.1-9.0)	9.76 (9.3-10.3)	$\times 1.14$	22.87	22.18	-0.69
future-genetic	9.3 (8.8-9.8)	10.08 (9.6-10.6)	$\times 1.08$	27.61	27.32	-0.28
scrabble	9.08 (8.6-9.6)	9.63 (9.1-10.1)	$\times 1.06$	64.00	63.67	-0.33
reactors	10.04 (9.5-10.6)	10.37 (9.9-10.9)	$\times 1.03$	25.18	25.03	-0.16
movie-lens	19.34 (18.7-20.1)	19.96 (19.3-20.7)	$\times 1.03$	23.30	23.18	-0.11
db-shootout	15.23 (14.6-15.9)	15.55 (14.9-16.2)	$\times 1.02$	78.83	77.88	-0.95
naive-bayes	10.53 (10.0-11.1)	10.73 (10.2-11.3)	$\times 1.02$	93.08	93.04	-0.04
philosophers	10.26 (9.8-10.8)	10.42 (9.9-10.9)	$\times 1.02$	41.67	41.73	+0.06
...						
scala-kmeans	167.51 (165.6-169.4)	167.43 (165.5-169.4)	$\times 1.00$	116.47	116.48	+0.01
...						
log-regression	7.64 (7.2-8.1)	7.58 (7.1-8.0)	$\times 0.99$	36.07	36.10	+0.03

The results of the simulation are shown in Tables 2 and 3. Overall, we can observe that the segment-aware and segment-ignorant computations yield quite similar results in terms of miss rate and interval width for most benchmarks, however, there are several notable observations:

Miss Rate with Normal vs Studentized Quantiles. The computations use a 99% confidence level, which should correspond to 1% miss rate. Across our simulation, the average miss rate was around 3.0% when using normal quantiles and around 1.8% when using studentized quantiles. Informally, the use of Student’s t distribution in the confidence interval computation compensates for inaccurate variance estimate with low run count. Although our simulated measurements are not drawn from a normal distribution and the confidence interval construction therefore does

not necessarily lead to the Student's t distribution, we can still consider the difference between the two miss rates to reflect the loss of accuracy in the $Var(\bar{X})$ estimate due to low run count.

Segment Impact in Segment-Ignorant Computation. Although the segment-ignorant computation does not consider segments explicitly, they are still reflected in the $Var(\bar{X})$ estimate. With very short runs ($N_{M(r)} \ll 1/\lambda$), the start of most segments will coincide with the start of a run and the segment effects S will be mostly indistinguishable from the run effects R . On the opposite end of the spectrum, with very long runs ($N_{M(r)} \gg 1/\lambda$), S will be similarly indistinguishable from the baseline workload performance B . The sweet spot where the segment-aware computation can differ substantially from the segment-ignorant computation are therefore experiments where the runs are long enough to include multiple segments, but not so long that the impacts of the segments entirely average each other out.

Workloads with High Segment Variance. The segment-ignorant computation gives significantly worse miss rate for the akka-uct, future-genetic and scala-doku workloads than the segment-aware computation, with 8 % to 23 % more misses. These three workloads have a high segment variance $Var(S)$ relative to the run variance $Var(R)$, and a low segment arrival rate λ , which increases the relative importance of the segment variance component in the run mean variance from Equation (3). In situations where the segment-ignorant computation underestimates the run mean variance, the potentially more accurate pooled variance estimate of the segment-aware computation may lead to a more accurate result for the run mean variance.

Bimodal Run Effects. Even with the studentized quantiles, both computation methods exhibit high miss rate with low run count for the scala-kmeans workload, and to a smaller degree also for the db-shootout, finagle-chirper, gauss-mix, movie-lens and page-rank workloads. These workloads have a bimodal distribution of the run effects R and an accurate estimate of the run variance $Var(R)$ therefore requires a high run count.

On the flip side, the studentized quantiles give very low miss rates with low run count for the log-regression, mnemonics and par-mnemonics workloads. For these workloads, the computation overcompensates for inaccurate variance estimate with low run count, the effect is associated with high kurtosis of mostly unimodal distribution of the run effects R .

Interpretation of Segment Weight. In the segment-aware computation, each segment contributes with equal weight to the run mean. In the segment-ignorant computation, the entire run can be modeled with a mixture distribution where the segment contribution is determined by the segment length. The variance of the run means $Var(\bar{X}_R^{Run})$ depends on the relative segment lengths and is generally higher than the variance of the run means computed from the segment means $Var(\bar{X}_R^{Seg})$ unless the segments are of equal length. With random segment lengths, the segment-aware computation therefore tends to produce more narrow and more centered confidence intervals.

Impact of Segment Detection Accuracy. With generated measurements, we can also compare the effect of using the ground truth information about segment boundaries vs using the segment boundaries detected in the generated measurements. Perhaps somewhat counterintuitively, the miss rate improvements of the segment-aware computation tend to be smaller when using the ground truth information about segment boundaries. This can be explained by considering the behavior of the segment detection algorithm – especially with long runs, the algorithm can label outlying measurements as (short) segments, overestimating the segment arrival rate λ and therefore reducing the relative importance of the segment variance component in the run mean variance from Equation (3).

Analytical Computation Summary

The segment-aware confidence interval computation delivers *equal or better miss rate* than the segment-ignorant computation, with only *very small impact on the average confidence interval width*, which makes it a suitable default method in the presence of segments. For workloads with high segment variance, the method significantly reduces the miss rate. Regardless of the method, workloads with bimodal run effects require high run counts for accurate confidence intervals. On average, the methods typically achieve worse miss rate than what the confidence level of 99 % would warrant even with the studentized quantiles.

5.2 Resampling Computation Methods

For resampling computation methods, which are more computationally expensive than the analytical methods, we generate enough measurements to distinguish miss rate changes of 1 % to 2 %, which is around 11 000 simulated experiments per benchmark workload. We generate more measurements for the *akka-uct*, *future-genetic* and *scala-doku* workloads, which yielded the largest miss rate differences between the segment-ignorant and segment-aware analytical computations. The evaluated methods are:

Resampling with Runs Only. We perform two-stage bootstrap by resampling runs, and then measurements within each resampled run (ignoring segments). Specifically, our bootstrap sample \bar{X}^* is an average of resampled run means $\bar{X}_{R(r^*)}^*$, and each resampled run mean $\bar{X}_{R(r^*)}^*$ is an average of resampled measurements $x_{r^*s^*m^*}$. During resampling, r^* is drawn uniformly with replacement from $1 \dots N_R$, and (s^*, m^*) is drawn uniformly with replacement from $\{(s, m) : s \in 1 \dots N_{S(r^*)}, m \in 1 \dots N_{M(r^*s)}\}$. For brevity, we refer to this computation method as *segment ignorant* in further text and with the *Run* acronym in formulas and tables.

Resampling with Segments and Runs. We perform three-stage bootstrap by resampling runs, then resampling segments within each resampled run, and finally resampling measurements within each resampled segment. Specifically, our bootstrap sample \bar{X}^* is an average of resampled run means $\bar{X}_{R(r^*)}^*$, each resampled run mean $\bar{X}_{R(r^*)}^*$ is an average of resampled segment means $\bar{X}_{S(r^*s^*)}^*$, and each resampled segment mean $\bar{X}_{S(r^*s^*)}^*$ is an average of resampled measurements $x_{r^*s^*m^*}$. During resampling, r^* is drawn uniformly with replacement from $1 \dots N_R$, s^* is drawn uniformly with replacement from $1 \dots N_{S(r^*)}$, and m^* is drawn uniformly with replacement from $1 \dots N_{M(r^*s^*)}$. For brevity, we refer to this computation method as *segment aware* in further text and with the *Seg* acronym in formulas and tables.

In both methods, we use 33 000 replicas and compute the 99 % confidence interval for the grand mean using the expanded percentile method from [18]. The results are shown in Table 4.

Overall, we can observe that the difference between the segment-aware and segment-ignorant confidence intervals with the resampling computation methods is similar to that of the analytical computation methods. In particular, the observations on the segment impact in segment-ignorant computation, high segment variance, and bimodal run effects also apply for the results in Table 4. Additionally:

Interval Location around Sample Mean. Unlike the analytical computations, the resampling computation does not necessarily center the confidence intervals around the sample mean, and thus provides more accurate interval location for workloads with skewed distribution of the initial run conditions R . This is illustrated in Figure 8 on the *scala-kmeans* workload.

Hit Rates and Relative Widths. Comparing Tables 2 to 4 further reveals that on average, the miss rate of the resampling computation (2.4 %) is between that of the analytical computations with normal quantiles (3.0 %) and studentized quantiles (1.8 %). Similarly, the average relative

Table 4. Miss rates and confidence interval widths for 99% confidence intervals computed using normal quantiles with segment-ignorant and segment-aware methods. Bold deltas are statistically significant using Wilson proportion test for rates and paired Mann–Whitney–Wilcoxon rank sum test for widths. Benchmarks with notable results only, see Appendix B for complete table.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	15.83 (15.0-16.7)	19.64 (18.7-20.6)	$\times 1.24$	32.84	32.14	-0.70
akka-uct	12.79 (12.1-13.6)	14.71 (13.9-15.5)	$\times 1.15$	18.84	18.44	-0.40
future-genetic	15.06 (14.1-16.1)	17.25 (16.2-18.4)	$\times 1.15$	22.55	22.22	-0.33
dotty	17.82 (14.9-21.3)	20.05 (17.0-23.7)	$\times 1.12$	9.96	9.90	-0.06
scrabble	15.14 (12.5-18.4)	16.95 (14.1-20.3)	$\times 1.12$	52.25	51.74	-0.51
naive-bayes	17.58 (14.7-21.0)	19.39 (16.4-23.0)	$\times 1.10$	74.46	73.88	-0.58
...						
scala-kmeans	176.73 (167.8-186.0)	176.99 (168.1-186.3)	$\times 1.00$	92.83	92.74	-0.08
...						

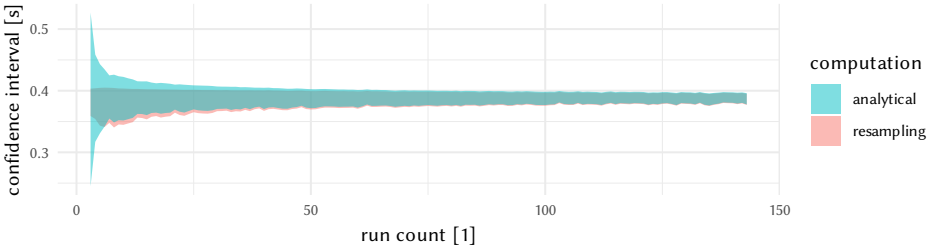


Fig. 8. Average confidence interval location for experiments with different run counts of the *scala-kmeans* workload. Colors distinguish the computation method used to obtain the interval.

confidence interval width of the resampling computation (5.0%) is between that of the analytical computations with normal quantiles (4.7%) and studentized quantiles (6.3%).

For reference, we also perform simulated experiments to determine the empirical distribution of the grand mean \bar{X} . In the context of the resampling computation methods, this constitutes the source of the ground truth for the confidence interval width. The results, shown in Table 5, indicate that the segment-aware intervals are, on average, 5.6% wider than this baseline.

Resampling Computation Summary

Similar to the analytical computation, the segment-aware confidence interval computation delivers *equal or better miss rate* than the segment-ignorant computation, with only *very small impact on the average confidence interval width* and significant reduction in the miss rate for workloads with high segment variance. Regardless of the method, workloads with bimodal run effects require high run counts for accurate confidence intervals. On average, the methods achieve worse miss rate than what the confidence level of 99% would warrant despite the intervals being slightly wider than the empirical baseline.

6 Handling Warm Up

A typical benchmark experiment does not immediately exhibit peak performance. Instead, it initially exhibits a period of an anomalous (often, but not always, reduced) performance, denoted as warm

Table 5. Segment-aware 99 % confidence interval width relative to centered 99 % inter quantile interval of the grand mean \bar{X} . Differences in braces were not found statistically significant using paired Mann–Whitney–Wilcoxon rank sum test. Sorted by relative interval width.

Benchmark	Relative Width	Benchmark	Relative Width	Benchmark	Relative Width
reactors	×1.13	finagle-http	×1.06	dec-tree	×1.05
akka-uct	×1.10	movie-lens	×1.06	mnemonics	×1.04
scala-doku	×1.08	als	×1.06	par-mnemonics	×1.03
scrabble	×1.08	dotty	×1.06	scala-stm-bench7	×1.03
fj-kmeans	×1.07	gauss-mix	×1.06	neo4j-analytics	(×1.03)
future-genetic	×1.07	philosophers	×1.06	log-regression	×1.03
db-shootout	×1.07	chi-square	×1.06	scala-kmeans	(×0.96)
page-rank	×1.07	naive-bayes	×1.05		
finagle-chirper	×1.06	rx-scrabble	×1.05		

up. During warm up, one-off mechanisms such as priming of processor, system and application caches, demand loading of code and data, just-in-time compilation, and stabilization of resource allocation algorithms including power management and load balancing, take place and thus impact performance.

While the warm up performance itself may be of practical interest (for example when considering the reaction time of interactive applications, or when minimizing the response latency of newly allocated service containers in the cloud), Renaissance and other benchmark suites are geared towards workloads that should mostly execute past warm up in practice. This is reflected in the benchmark harness design, which executes the same workload repeatedly to get past warm up, and gives rise to the question of when warm up completes.

Earlier work such as [11], and benchmarks such as [6], equate warm up with performance disruptions that increase sample variance, and thus detect end of warm up by waiting for sliding window variance to shrink below a heuristic threshold. This approach, however, does not cope well with the typical behavior of just-in-time compiled workloads, where isolated compilation bursts yield periods of stable performance interleaved with sudden performance shifts, as illustrated in Figure 9. Rather than focusing purely on performance variability, we therefore define warm up more broadly as an initial execution phase where the system behavior is not relevant to the goals of the particular performance evaluation experiment. We illustrate the methodological shift due to our warm up definition on the observed compilation progress.

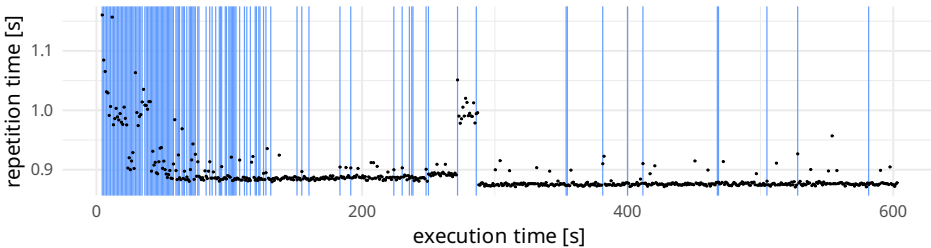


Fig. 9. Ad hoc example of periods of stable performance interleaved with performance shifts in warm up of the *scala-stm-bench7* workload. Blue vertical lines denote repetitions where more than one method compilation completed, the top measurement permille is removed to improve scale.

6.1 Compilation Progress Based Warm Up

When the Renaissance benchmark suite is used to determine the performance of code produced by just-in-time compilation, warm up must last long enough for the performance relevant code to be compiled. Because the compilation prioritizes frequently executed code [15], waiting for warm up to complete equates to waiting for compilation activity to subside (because just-in-time compilation is requested for all frequently executed code, any left-over code is not frequently executed, and therefore unlikely to be relevant to performance). This behavior is illustrated in Figure 10.

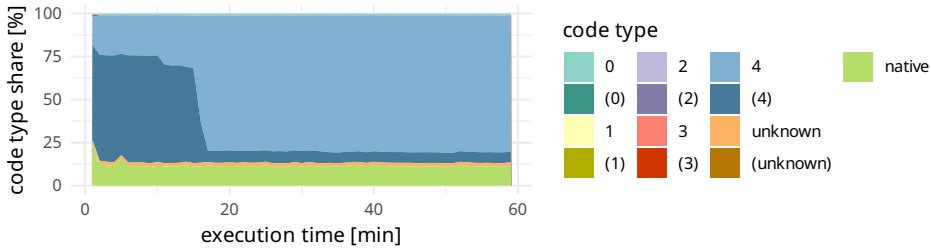


Fig. 10. Ad hoc example of the code type evolution during warm up of the *scala-stm-bench7* workload. Color denotes code type, numerical levels correspond to the OpenJDK compilation log (0 for JNI wrapper code, 1 to 3 for C1 compiled code with different profiling levels, 4 for C2 compiled code), *unknown* denotes dynamically installed code with no associated compilation record, *native* denotes code of the virtual machine itself, parentheses indicate code that was not observed running in the last 1 h of an 8 h execution.

In the execution profile in Figure 10, a significant share of the warm up happens in the first minutes, where the share of the virtual machine code (which includes the workload executing in the interpreter) decreases to around 12% of the execution time, and the share of the code produced by the C2 compiler grows to around 85%. However, most of that code is replaced in subsequent compilations and only around 27% is used for the remaining workload execution. A significant share of those subsequent compilations happens around 17 min, afterwards the share of the code produced by the C2 compiler is around 86% and around 92% of that code is used for the remaining workload execution. In this case, an experiment used to determine the performance of code produced by the C2 compiler should therefore warm up for at least 2 min if both interim and stable code is of interest, and at least 17 min if only stable code is relevant.

To highlight the difference between warm up criteria based on performance variability and on compilation progress, we collect the same execution profiles as in Figure 10 for 10 8.5 h runs of each Renaissance workload and compute the warm up time using three different criteria. In Table 6, the *CoV* column shows the warm up time based on performance variability from [11], with a workload considered warm if the coefficient of variation of 10 subsequent repetitions drops below 0.02. The *% Stable* and *% Last Tier* columns show the warm up time based on compilation progress – both compute shares of a specific code type across 1 min windows and consider the workload warm when that share reaches 95% of the maximum observed across the entire execution. The *% Stable* column considers code that is not replaced in subsequent compilations, effectively defining warm workload as *workload whose code will no longer change due to compilation*. The *% Last Tier* column considers code that is produced by the C2 compiler, effectively defining warm workload as *workload whose code relies as much as possible on the C2 compiler*.

Table 6 offers several observations:

Table 6. Warm up times computed using different criteria. Average times, with min-max interval in parentheses.

Benchmark	Warm Up [s] per Criteria		
	CoV	% Stable	% Last Tier
akka-uct	123 (29-459)	120 (120-540)	120 (120-120)
als	43 (33-54)	120 (120-120)	180 (180-180)
chi-square	15 (13-22)	120 (60-120)	120 (120-120)
db-shootout	1446 (117-15587)	120 (120-120)	120 (120-120)
dec-tree	27 (23-44)	120 (120-120)	240 (240-240)
dotty	68 (59-72)	180 (180-240)	420 (300-480)
finagle-chirper	45 (42-50)	930 (420-3360)	120 (120-120)
finagle-http	21 (20-30)	390 (120-2100)	120 (120-120)
fj-kmeans	8 (5-9)	60 (60-60)	60 (60-60)
future-genetic	4 (3-6)	60 (60-120)	120 (120-120)
gauss-mix	17 (14-20)	60 (60-60)	180 (120-240)
log-regression	75 (39-102)	120 (120-120)	180 (180-240)
mnemonics	13 (11-16)	60 (60-60)	120 (120-180)
movie-lens	68 (65-124)	120 (120-2580)	180 (180-240)
naive-bayes	38 (30-47)	120 (120-120)	120 (120-180)
neo4j-analytics	50 (49-58)	120 (120-120)	180 (120-180)
page-rank	37 (30-39)	2490 (2460-2580)	120 (120-120)
par-mnemonics	10 (10-10)	60 (60-60)	120 (120-120)
philosophers	9 (4-448)	120 (60-2460)	60 (60-60)
reactors	442 (43-1308)	2880 (780-24120)	150 (120-300)
rx-scrabble	4 (3-6)	60 (60-19980)	120 (120-120)
scala-doku	23 (22-23)	14100 (13920-14280)	6120 (240-12360)
scala-kmeans	1 (1-2)	60 (60-60)	60 (60-120)
scala-stm-bench7	7 (4-32)	2130 (900-8460)	120 (120-180)
scrabble	62 (13-393)	60 (60-60)	60 (60-60)

Performance Stability Does Not Reflect Code Maturity. Most workloads are considered warm using the performance variability criterion well before the compilation progress criterion, suggesting that the former may not be a good choice for investigating code produced by just-in-time compilation.

Tuning Required. Even when working with a single benchmark suite, heuristic thresholds in warm up criteria may require workload-specific tuning. This is illustrated on the *db-shootout* workload, whose repetition times form a multimodal distribution with a high coefficient of variation. The warm up time based on performance variability therefore fluctuates even though the warm up time based on compilation progress is stable.

Inherent Detection Overhead. Evaluating the warm up criteria requires data from a window of multiple repetitions. A workload is considered warm only after the entire window meets the given criteria, thus all repetitions within the window are warm and the size of the window represents an inherent overhead in terms of extra repetitions performed beyond the warm up point. This holds even for more complex methods [38, 39]. Some workloads exhibit a wide min-max range of warm up times, indicating that using a conservative warm up time constant across multiple workload runs, suggested by Kalibera et al. [24], entails similar overhead.

6.2 Optimum Experiment Dimensions with Segments and Warm Up

In an experiment that performs N_R runs and collects a fixed number of measurements N_{MR} in each run, the choice of N_R and N_{MR} influences the variance of the grand mean \bar{X} and therefore the accuracy of the performance evaluation. Kalibera et al. [24] give a formula for the choice of experiment dimensions that minimizes the variance of the grand mean:

$$nr_i = \left\lceil \sqrt{\frac{Cost_{i+1}}{Cost_i} \times \frac{Var(L_i)}{Var(L_{i+1})}} \right\rceil \quad (4)$$

In Equation (4), i ranges over the levels of hierarchy in the experiment – in our case, $i = 1$ refers to measurements in a segment, $i = 2$ refers to segments in a run, and $i = 3$ refers to runs in an experiment. The $Var(L_i)$ notation denotes the variance of the random variables that describe the impact of the individual levels of hierarchy on the baseline workload performance – in our case, L_1 corresponds to B , L_2 corresponds to S , and L_3 corresponds to R from Section 4. Finally, $Cost_i$ refers to the cost associated with measurements at level i .

Kalibera et al. [24] assume a model where the experiment dimensions can be controlled independently. In our case, however, the number of segments in a run and the number of measurements in a segment are controlled only indirectly through run duration and are related by the segment arrival rate λ , making it impossible to apply Equation (4) directly.

As a workaround, we can rely on our observation made with the segment-ignorant computation, where, depending on the duration of the runs, the segment effects S are attributed partially to the run effects R and partially to the baseline workload performance B . For the extremes of very short and very long runs, we can substitute $Var(L_1)$ and $Var(L_2)$ in Equation (4) as follows:

- For very short runs ($N_{MR} \ll 1/\lambda$), $Var(L_1) = Var(B)$ and $Var(L_2) = Var(R) + Var(S)$.
- For very long runs ($N_{MR} \gg 1/\lambda$), $Var(L_1) = Var(B) + Var(S)$ and $Var(L_2) = Var(R)$.

To put these alternatives in perspective, we note that our workloads exhibit change rates λ around 1×10^{-3} to 1×10^{-4} and repetition times on the order of 0.1 s to 10 s. The point halfway between the alternatives, where on average one segment arrival is observed in each run, is typically reached with runs of a few hundred seconds (minimum 23 s for *log-regression*, maximum 25 min for *page-rank*). Our experiment dimensions assume a total duration of 1 h to 24 h and a warm up of 5 min to 10 min, most simulated experiments will therefore fall into the short runs alternative and the segment effects S will mostly fuse with the run effects R .

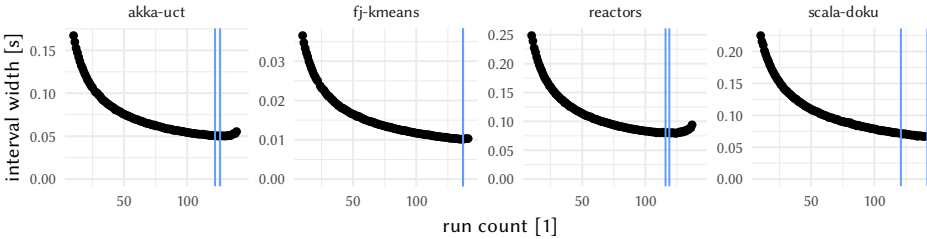


Fig. 11. Ad hoc examples of the centered 99 % inter quantile interval width of the grand mean \bar{X} for 24 h experiments with different run counts. Blue vertical lines denote the number of runs recommended by Equation (4) for very short runs (right) and very long runs (left).

Figure 11 compares the results of Equation (4) with the true 99 % inter quantile interval width of the grand mean \bar{X} . The examples indicate that the recommended experiment dimensions fall into a flat region of the interval width curve, where erring slightly in favor of fewer longer runs does not

increase the interval width much. When substituting the short runs alternative to Equation (4), we typically have the cost ratio in the order of hundreds and the variance ratio at most in the order of ones, resulting in a recommendation to collect at most tens of measurements within each run. Importantly, this is close to the inherent overhead of the warm up detection methods highlighted in Section 6.1 – in practical scenarios, the number of measurements collected within each run may therefore be driven by the warm up detection rather than the recommendation from Equation (4).

Handling Warm Up Summary

Merely reaching stable performance may not meet the purpose of the warm up, and *the warm up duration changes significantly depending on the purpose*. For warm up criteria based on performance variability and on compilation progress, *the warm up duration is often not stable between runs*, representing a complication for experiments that determine a common warm up time for multiple runs. The overhead of the warm up detection methods may represent a more significant factor in dimensioning experiments than the variance contributed by the levels of hierarchy in the experiment.

7 Practical Methodology

For reader convenience, we summarize the relevant steps of our work in the form of practical methodology, akin to the recommendations of Georges et al. [11] and Kalibera et al. [24]. We consider experiments that assess performance of repetitive mostly stable workloads, such as DaCapo [6], Renaissance [35], or similar experimental workloads used in performance evaluation of modern virtual machines.

To reflect the characteristics of the workloads, including warm up and steady state shifts, we perform initial methodology calibration steps, which ensure we compute meaningful statistical estimators to report performance. The calibration steps collect diagnostic measurements – in simple scenarios, these measurements may suffice for subsequent evaluation, while in more complex experiments, the same calibration can apply to reasonably similar experiment components that collect additional measurements. We use *italic* for calibration examples that illustrate the quantitative parameters in the context of our experiments from Section 5.

Define Warm Up.

Warm up must be defined prior to the initial methodology calibration, which assumes only warm measurements are used. Section 6 illustrates how the warm up definition, and consequently the warm up duration, is determined by performance experiment objectives. Use one of the existing warm up detection methods based on steady state [5, 11, 38, 39] for experiments that aim to characterize stable performance. Use warm up detection from Section 6.1 for experiments specifically concerned with JIT compiled code.

Estimate Segment Variability.

Goal. Estimate the contribution of the segment variability to the overall measurement variability.

Procedure. Select the minimum steady state shift rate the experiment must have power to detect, λ_{min} . Collect measurements from a small number of runs N_R^{Long} whose length is sufficient to likely observe the steady state shifts.

The choice of N_{Long} is connected to the likelihood of failing to observe steady state shifts even if the system can exhibit such behavior. With independent arrival assumptions from Section 4, a run has at least 50 % chance of encountering a segment in $\ln(2)/\lambda_{min}$ measurements. With $N_R^{Long} = 5$ runs of such duration, or a single run of 5 times such duration for segment detection, the probability of encountering a segment at least once already exceeds 96 %.

As an example from our experiments, the median steady state shift rate was 2.2×10^{-4} , suggesting a segment should be observed with 50 % chance over 3139 measurements.

Use the PELT algorithm [27] to identify segments as do Barrett et al. [3] and Traini et al. [38], with the penalty value determined by the location of the knee on the penalty-vs-segment count curve as done by Traini et al. [38] and in Section 5.

Output. Estimate of segment variance computed using Equation (6) from Appendix A.2, used in subsequent decisions.

Estimate Run and Measurement Variability.

Goal. Estimate the contribution of the run and measurement variability to the overall measurement variability.

Procedure. Collect measurements from a small number of runs N_R^{Short} whose length exceeds warm up by a small number of observations N_M^{Short} to estimate run and measurement variability.

The choices of N_R^{Short} and N_M^{Short} are related to the accuracy of the variance estimates. With normality assumptions, 30 observations are often cited as sufficient for a reasonably accurate estimate [11], but the choice depends on the measurement distribution.

As an example from our experiments, which use realistic distributions, as few as $N_R^{Short} = 25$ runs typically achieved a relative error less than 5 %.

Output. Estimates of run and measurement variances computed using Equations (5) and (7) from Appendix A.2, used in subsequent decisions.

Decide on Experiment Dimensions.

Goal. Select the number and length of runs to balance the individual components of performance variability towards narrow confidence intervals.

Procedure. The purpose of the performance experiment may dictate some experiment dimensions, such as including long runs to increase the likelihood of observing rare events. Use Equation (4) for guidance on the ratio of experiment dimensions that balances the individual components of the performance variability. Use Equation (3) together with analytical confidence interval computation to estimate the confidence interval width for specific experiment dimensions, and, conversely to estimate the dimensions needed for specific interval width.⁵⁶

Decide on Confidence Interval Computation Method.

Goal. Decide on segment-aware vs segment-ignorant and analytical vs resampling computation for the confidence intervals to avoid complexity.

Procedure. Equation (3) determines whether segment-aware computation is beneficial. When $Var(R) \gg Var(S)/N_{SR}$, where $Var(R)$ and $Var(S)$ are the run and segment variance and N_{SR} is the average number of segments in a run, the segment variability contribution is negligible and segment-ignorant computation suffices. The segment-aware computation is most beneficial when the ratio of segment variance to run variance is high. Section 5 indicates the choice of segment-aware computation is a safe default.

As an example from our experiments, the akka-uct, future-genetic, scala-doku and scrabble workloads, which benefit from segment-aware computation per Tables 2 and 3, also had the four

⁵Section 5 relied on very large experiment dimensions to make sure the accuracy evaluation is reliable, however, this is not mandated by our methodology.

⁶The recommendations of both Georges et al. [11] and Kalibera et al. [24] assume equal number of measurements in all runs. The segment-aware computations from Appendix A.2 can handle runs of various lengths. Although such experiment configurations will not minimize the variance of the average overall performance per Equation (4), Figure 11 suggests complementing mostly short runs with several long runs can achieve reasonably narrow confidence intervals while also capturing information on long term workload behavior that only long runs can provide.

highest $\text{Var}(S)/\text{Var}(R)$ ratios, respectively 36 %, 58 %, 367 % and 40 %, while the workloads that did not benefit typically had the ratio below 10 %.

Use standard histogram plots of the measurements from the earlier steps (as in Figures 4 to 6) to decide on analytical vs resampling computation. Although more computationally expensive, the resampling methods work better when dealing with asymmetric distributions of the run and segment effects. The analytical computation is faster and conveniently handles extreme confidence levels. Section 5 indicates the choice of resampling methods is a safe default.

Collect Measurements.

Multiple runs involving virtual machine restarts, as well as other platform-specific precautions [20], are needed to fully characterize the workload performance. The Randomized Multiple Interleaved Trials methodology [1] helps avoid bias from time-varying experimental conditions.

Sanity Checks.

Use scatter plots (as in Figures 1, 7 and 9) to identify non-stationary performance artifacts, such as trends, which render the goal of reporting the average overall performance meaningless. Resource metrics such as memory consumption may reveal trends before they become visible in the time measurements, this was the case with the reactors workload in our experiments due to a leak that was since fixed in Renaissance.

8 Threats to Validity

Probably the most pertinent question concerning the validity of our results is how much our statistical modeling assumptions match reality. As is common in the existing models [11, 24], we treat measurement samples as independent identically distributed random variables. A sufficiently complex system can violate this assumption with almost arbitrary implications, however, extreme scenarios would often constitute behavior of limited practical relevance.

Also similar to existing models, we assume the run and segment effects can be modeled as additive shifts of the baseline performance. Neither is generally guaranteed, however, a visual examination of the measurements suggests mostly additive shifts are quite common. We also note that the hierarchical nature of the evaluation uses the individual measurement samples as the most averaged component of the observed performance, dampening the effect of other potential distribution changes on the average performance.

In some computations, we assume averages will tend to normal distribution, with compensation for small sample counts. Our experiments, however, do not rely on normality, and show the effect of non-normal measurement distributions on the confidence interval accuracy.

Specific to our model is the assumption that execution-segment conditions change at independent random moments with a constant arrival rate. Directly, we only use this assumption to derive the expected number of segments in a run, however, it also has a more subtle implication, making the arrivals, and therefore the segment lengths, independent of the segment shifts. This contradicts the behavior of the PELT algorithm for identifying the execution segments, which is more likely to detect shorter segments with larger shifts. In the extreme, the PELT algorithm may confuse short bursts of outliers for execution segments, which may not match the intended meaning of segments. Filtering outliers prior to identifying the execution segments reduces this concern, and was also done in prior work that relied on the PELT algorithm [3, 38].

The very first execution segment of a run constitutes a special case, as it starts together with the run, with (necessarily) independent segment shift. As a result, our model shows that for low segment arrival rates relative to warm up, it is more efficient to initiate a new run early rather than

wait for a new segment to arrive (because a new run will also initiate a new segment). For this observation to extend to reality, the distribution of the segment shifts at the end of the warm up would have to match the same distribution in later execution – unfortunately, this match is hard to test even with our wealth of measurements, in part because the results of any such test very much depend on the adopted distribution similarity metric, but also because there is an element of circular reasoning involved – if the role of the warm up is to get past transitory workload performance, then any difference between segment shifts at the end of the warm up and later execution is, by definition, an indication of insufficient warm up. We believe that in practice, the fluid nature of the warm up detection methods will prevent accurate enough comparisons between warm up time and segment arrival rate, and longer measurements will still be needed anyway to examine the very existence of execution segments – but the effect may explain why the lack of stability lamented by Barrett et al. [3] does not attract more attention in measurement studies with enough runs.

The assumption of independent execution-segment conditions is also related to the notion of the true performance estimated by measurement. Our work targets experiments whose goal is to estimate the expected performance across runs and segments whose durations are incidental rather than meaningful. When that is the case, the computation isolates the estimate of the potential performance from the impact of run and segment duration. In settings where run or segment durations are themselves meaningful and should be factored into the definition of true performance, alternative weighted grand-mean estimators may be more appropriate. Such estimators exhibit different finite-sample variance and therefore different confidence-interval coverage behavior.

Finally, the assumption of independent execution-segment conditions can be examined from the perspective of mechanisms that cause the segments to appear. Some, such as dynamic compilation, are likely to exhibit initially high arrival rate, which should gradually subside, and are best handled as warm up. We note, however, that even dynamic compilation may cause performance changes after long periods of repetitive execution, due to high invocation thresholds used to trigger compilation, or even indefinitely repeating performance changes, for example in applications that generate code, or due to artifacts such as deep loops. Other mechanisms, such as the memory layout changes described in Section 3, can also keep triggering performance changes indefinitely, simply because execution will likely involve cycles of allocation and (potentially copying) garbage collection. Additionally, mechanisms beyond the scope of our experiments, such as the scheduling algorithms employed on heterogeneous multicore processors, can also interact with the benchmark workload in complex ways leading to perpetual performance fluctuations.

Where external validity is concerned, the usual platform disclaimers apply – while our general experience is that workloads of similar complexity on platforms with dynamic compilation and garbage collection exhibit similar behavior, a definitive confirmation for a specific workload and platform can only be given by additional experiments. For more arguments supporting our conclusions, we note that dependency of workload performance on (transient) memory layout, which is among the causes of performance changes we consider, has been reported repeatedly [9, 23, 32]. We also suggest that some parameters of our workloads, such as the segment lengths and shifts, are perceived relative to the baseline performance and are therefore likely to retain their relative sizes – for example, much shorter segments or much smaller shifts would become difficult to separate from baseline sample variability, while much longer segments or much larger shifts would make the baseline performance impractical to observe.

Our simulation uses a mixture of experiment dimensions chosen to resemble experiments conducted on a time scale of several minutes to several hours. Some of our quantitative conclusions depend on this mixture and may become more or less pronounced with other experiment dimensions. Our reproducibility package includes a breakdown of the quantitative results that shows this

dependency, with some obvious observations – for example, when the experiment dimensions favor many very short runs, the difference between segment-aware and segment-ignorant methods becomes negligible because there is not enough time for segments to arrive.

9 Related Work

This paper directly follows the methodology work in related literature [2, 11, 19, 24, 25], which has helped establish the practice of designing experiments that collect multiple measurement samples from multiple runs (and possibly multiple compilations, multiple deployments, and so on) to provide complete information about system performance, and which has advocated the use of confidence intervals, among other statistical tools, to report such information. In our work, we stay with this practice, but extend it in the light of findings reported by Barrett et al. [3] and later Traini et al. [38], which both point out that many benchmarks do not reach steady state within reasonable time.

Our observations on the relationship between warm up detection and experiment dimensions are connected to the prior work [5, 17, 21, 28, 38–41], which documents the open issues in warm up detection and the impact of experiment dimensions on result accuracy. Notably, the warm up detection tends to focus on the stability of the collected performance metrics. Code maturity, which we also use for warm up detection, was investigated in the context of profiling by Mueller et al. [31].

Finally, besides the studies pointing out the lack of steady state in existing benchmarks, our point that we may need to give up on steady state is connected to work documenting issues with benchmark quality, such as [8], and to studies that show how minute workload properties impact performance [14, 32] and how complex it may be to do something about it [9, 12].

10 Conclusion

Our work extends the findings of Barrett et al. [3] and Traini et al. [38] on the lack of steady state in deterministic microbenchmarks to larger workloads, represented by the Renaissance benchmark suite [35]. In our measurements, most workloads do not reach steady state consistently across runs, and when an apparent steady state is detected, it can represent only a fraction of the run (Table 1).

By documenting sudden shifts in performance that occur after several hours of stable execution (Figure 1), we show that reaching steady state is not merely a matter of executing the workload for a few more minutes. An analysis of a selected anomaly (Section 3) illustrates how multiple mechanisms at different levels of abstraction can interact to cause a performance change, suggesting that exhaustively analyzing and eliminating every performance anomaly is not a practical goal for workloads on contemporary virtual machines.

To evaluate workloads that exhibit such shifts, we propose a statistical model in which performance exhibits multiple segments of steady state separated by sudden changes (Section 4), and compute confidence intervals for the average performance under this model. In simulations calibrated by extensive measurements (Section 5), the segment-aware methods achieve equal or better confidence interval coverage than the existing segment-ignorant methods. The largest improvements appear in workloads with high segment variance – for *scala-doku*, the miss rate for 99% confidence intervals drops from 26.16% to 21.87% with normal quantiles, from 12.38% to 10.09% with studentized quantiles, and similarly for *bootstrap* (Tables 2 to 4).

Finally, we emphasize scope. No finite methodology can guarantee that all possible performance anomalies will be observed, and it is always possible to construct scenarios where changes appear well beyond any reasonable experiment dimensions. Our goal is therefore not to prove a complete coverage of anomalies, but to provide a measurement and reporting methodology that remains meaningful when anomalies in the shape of sudden performance shifts do occur within the experiment horizon. Experiment dimensions should be chosen to reflect the minimum anomaly rate the experiment must have power to observe.

Acknowledgments

This work has been supported by the Chips-JU Horizon Europe project NexTArc (101194287, 9A25008), the SNF Scientific Exchanges project “Benchmark Evolution Analysis” (IZSEZ0 229176), and the USI FIR project “Understanding and Mitigating Performance Variability on Managed Runtimes”.

Data Availability Statement

The paper is accompanied by a reproducibility package, available online at <https://doi.org/10.5281/zenodo.18758902>. The package contains complete data collected during measurements and simulations described in the paper, together with complete scripts used to browse and process the data and generate all tables and figures used in the paper. The package also includes a standalone implementation of the computations described in Appendix A.

References

- [1] Ali Abedi and Tim Brecht. 2017. Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE)*. ACM, L'Aquila, Italy, 287–292. doi:10.1145/3030207.3030229
- [2] Eytan Bakshy and Eitan Frachtenberg. 2015. Design and Analysis of Benchmarking Experiments for Distributed Internet Services. In *Proceedings of the 24th International Conference on World Wide Web (WWW)*. IW3C2, 108–118. doi:10.1145/2736277.2741082
- [3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount, and Laurence Tratt. 2017. Virtual Machine Warmup Blows Hot and Cold. *Proceedings of ACM on Programming Languages* 1, OOPSLA (2017), 52:1–52:27. doi:10.1145/3133876
- [4] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. *ACM SIGPLAN Notices* 41, 6 (2006), 158–168. doi:10.1145/1133255.1134000
- [5] Martin Beseda, Vittorio Cortellessa, Daniele Di Pompeo, Luca Traini, and Michele Tucci. 2025. A Kernel-Based Approach for Accurate Steady-State Detection in Performance Time Series. arXiv:2506.04204. arXiv:2506.04204 doi:10.48550/arXiv.2506.04204
- [6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiederemann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st International Conference on Object-Oriented Programming, Systems Languages, & Applications (OOPSLA)*. ACM. doi:10.1145/1167473.1167488
- [7] Lubomír Bulej, Vojtech Horký, Michele Tucci, Petr Tůma, François Farquet, David Leopoldseider, and Aleksandar Prokopec. 2023. GraalVM Compiler Benchmark Results Dataset (Data Artifact). In *Companion of the 14th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 65–69. doi:10.1145/3578245.3585025
- [8] Diego Costa, Cor-Paul Bezemer, Philipp Leitner, and Artur Andrzejak. 2021. What’s Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering* 47, 7 (2021), 1452–1467. doi:10.1109/TSE.2019.2925345
- [9] Charlie Curtsinger and Emery D. Berger. 2013. Stabilizer: Statistically Sound Performance Evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 219–228. doi:10.1145/2451116.2451141
- [10] Bradley Efron and R. J. Tibshirani. 1994. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, New York. doi:10.1201/9780429246593
- [11] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. In *Proceedings of the 22nd International Conference on Object-Oriented Programming, Systems Languages, & Applications (OOPSLA)*. ACM, 57–76. doi:10.1145/1297027.1297033
- [12] Andy Georges, Lieven Eeckhout, and Dries Buytaert. 2008. 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, 367–384. doi:10.1145/1449764.1449794
- [13] Isaac Gouy. 2025. The Computer Language Benchmarks Game. <https://benchmarksgame-team.pages.debian.net/benchmarksgame>.

- [14] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. 2006. Relative Factors in Performance Analysis of Java Virtual Machines. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE)*. ACM, 111–121. doi:10.1145/1134760.1134776
- [15] Tobias Hartmann, Albert Noll, and Thomas Gross. 2014. Efficient Code Management for Dynamic Multi-Tiered Compilation Systems. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ)*. ACM. doi:10.1145/2647508.2647513
- [16] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 251–269. doi:10.1145/1028976.1028998
- [17] Sen He, Glenna Manns, John Saunders, Wei Wang, Lori Pollock, and Mary Lou Soffa. 2019. A Statistics-Based Performance Testing Methodology for Cloud Applications. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 188–199. doi:10.1145/3338906.3338912
- [18] Tim Hesterberg. 2014. What Teachers Should Know about the Bootstrap: Resampling in the Undergraduate Statistics Curriculum. arXiv: 1411.5279.
- [19] T. Hoefler and R. Belli. 2015. Scientific Benchmarking of Parallel Computing Systems: Twelve Ways to Tell the Masses When Reporting Performance Results. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 1–12. doi:10.1145/2807591.2807644
- [20] Vojtěch Horký, Peter Libič, Antonin Steinhauser, and Petr Tůma. 2015. DOs and DON'Ts of Conducting Performance Measurements in Java. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 337–340. doi:10.1145/2668930.2688820
- [21] Nils Japke, Martin Grambow, Christoph Laaber, and David Bernbach. 2025. μ OpTime: Statically Reducing the Execution Time of Microbenchmark Suites Using Stability Metrics. arXiv:2501.12878. arXiv:2501.12878 doi:10.48550/arXiv.2501.12878
- [22] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Automated Detection of Performance Regressions: The Mono Experience. In *Proceedings of the 20th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 183–190. doi:10.1109/MASCOT.2005.18
- [23] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. 2005. Benchmark Precision and Random Initial State. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. SCS, 484–490.
- [24] Tomas Kalibera and Richard Jones. 2013. Rigorous Benchmarking in Reasonable Time. *ACM SIGPLAN Notices* 48, 11 (2013), 63–74. doi:10.1145/2555670.2464160
- [25] Tomas Kalibera and Richard Jones. 2020. Quantifying Performance Changes with Effect Size Confidence Intervals. arXiv:2007.10899. arXiv:2007.10899 doi:10.48550/arXiv.2007.10899
- [26] Tomas Kalibera and Petr Tuma. 2006. Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results. In *Formal Methods and Stochastic Models for Performance Evaluation*, András Horváth and Miklós Telek (Eds.), Number 4054 in Lecture Notes in Computer Science. Springer, 63–77. doi:10.1007/11777830_5
- [27] R. Killick, P. Fearnhead, and I. A. Eckley. 2012. Optimal Detection of Changepoints With a Linear Computational Cost. *J. Amer. Statist. Assoc.* 107, 500 (2012), 1590–1598. doi:10.1080/01621459.2012.737745
- [28] Christoph Laaber, Stefan Würsten, Harald C. Gall, and Philipp Leitner. 2020. Dynamically Reconfiguring Software Microbenchmarks: Reducing Execution Time without Sacrificing Result Quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 989–1001. doi:10.1145/3368089.3409683
- [29] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2011. Dthreads: Efficient Deterministic Multithreading. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. ACM, 327–336. doi:10.1145/2043556.2043587
- [30] P. Massart. 1990. The Tight Constant in the Dvoretzky-Kiefer-Wolfowitz Inequality. *The Annals of Probability* 18, 3 (1990), 1269–1283. doi:10.1214/aop/1176990746
- [31] Rene Mueller, Maria Carpen-Amarie, Matvii Aslandukov, and Konstantinos Tovletoglou. 2024. The Cost of Profiling in the HotSpot Virtual Machine. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR)*. ACM, 112–126. doi:10.1145/3679007.3685055
- [32] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong!. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 265–276. doi:10.1145/1508244.1508275
- [33] R.C. Paule and J. Mandel. 1982. Consensus Values and Weighting Factors. *J. Res. Nat. Bur. Standards* 87, 5 (1982), 377. doi:10.6028/jres.087.022

- [34] Andrej Pečimůth, David Leopoldseider, and Petr Tůma. 2025. A Pragmatic Approach to Replay Compilation. In *Proceedings of the 9th International Workshop on Modern Language Runtimes, Ecosystems, and VMs (MoreVMs)*, Jonathan Edwards, Roly Perera, and Tomas Petricek (Eds.), Vol. 134. Leibniz-Zentrum für Informatik, 3:1–3:4. doi:10.4230/OASiCS.Programming.2025.3
- [35] Aleksandar Prokopec, Andrea Rosà, David Leopoldseider, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 17. doi:10.1145/3314221.3314637
- [36] Bernard W. Silverman. 2018. *Density Estimation for Statistics and Data Analysis*. Routledge. doi:10.1201/9781315140919
- [37] Ole Tange. 2025. GNU Parallel.
- [38] Luca Traini, Vittorio Cortellessa, Daniele Di Pompeo, and Michele Tucci. 2022. Towards Effective Assessment of Steady State Performance in Java Software: Are We There Yet? *Empirical Software Engineering* 28, 1 (2022), 13. doi:10.1007/s10664-022-10247-x
- [39] Luca Traini, Federico Di Menna, and Vittorio Cortellessa. 2024. AI-Driven Java Performance Testing: Balancing Result Quality with Testing Time. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 443–454. doi:10.1145/3691620.3695017
- [40] Antonio Trovato, Luca Traini, Federico Di Menna, and Dario Di Nucci. 2025. AMBER: AI-Enabled Java Microbenchmark Harness. In *Proceedings of the IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 762–766. doi:10.1109/ICST62969.2025.10988925
- [41] Tom Wallace, Beatrice Ombuki-Berman, and Naser Ezzati-Jivan. 2023. Identification and Classification of JMH Microbenchmark States Using Time Series Analysis. In *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 101–105. doi:10.1145/3578245.3584694

Received 2025-10-08; accepted 2026-02-17

A Computation Details

This appendix gives the details of the computations used to estimate the mean and variance statistics from samples in Sections 5 and 7, also available as a standalone implementation in the reproducibility package. We recall and extend the notation from Section 4:

- Let N_R stand for the **total number of runs**, and r denote the **run index**, $r \in 1 \dots, N_R$.
- Let $N_{S(r)}$ stand for the **number of segments** within run r , and s denote the **segment index** within run, $s \in 1 \dots, N_{S(r)}$.
- Let $N_{M(rs)}$ stand for the **number of measurements** within segment s of run r , and m denote the **index of measurement** within segment s of run r , $m = 1 \dots, N_{M(rs)}$.

We use shortcuts for totals, useful in mean computations:

- Let $N_{M(r)} = \sum_s N_{M(rs)}$ denote the **total number of measurements** within run r .
- Let $N_S = \sum_r N_{S(r)}$ denote the **total number of segments** in the experiment.
- Let $N_M = \sum_r N_{M(r)}$ denote the **total number of measurements** in the experiment.

We use shortcuts for degrees of freedom, useful in variance computations:

- $df_R = N_R - 1$ for run-level variance estimates.
- $df_{S(r)} = N_{S(r)} - 1$ for segment-level variance estimates in run r .
- $df_{M(rs)} = N_{M(rs)} - 1$ for measurement-level variance estimates in segment s of run r .
- $df_S = \sum_r df_{S(r)} = N_S - N_R$ for segment-level variance estimates pooled across the experiment.
- $df_M = \sum_{rs} df_{M(rs)} = N_M - N_S$ for measurement-level variance estimates pooled across the experiment.

A.1 Analytical Computation with Runs Only

Consistent with [11, 24], we compute the sample mean performance of each run disregarding segments, $\bar{X}_{R(r)}^{Run}$, and then the grand sample mean \bar{X}^{Run} of these run sample means, as the estimate of the baseline performance.

$$\bar{X}_{R(r)}^{Run} = \frac{1}{N_{M(r)}} \sum_{s,m} x_{rsm} \quad (\text{run sample mean for run } r)$$

$$\bar{X}^{Run} = \frac{1}{N_R} \sum_r \bar{X}_{R(r)}^{Run} \quad (\text{grand sample mean})$$

We use the sample variance of the run sample means, $Var(\bar{X}_R^{Run})$, as an estimate of $Var(\bar{X})$, and use it to construct the confidence interval.

$$\widehat{Var}^{Run}(\bar{X}) \approx Var(\bar{X}_R^{Run}) = \frac{1}{df_R} \sum_r (\bar{X}_{R(r)}^{Run} - \bar{X}^{Run})^2$$

A.2 Analytical Computation with Segments and Runs

Consistent with Equation (2), we compute the sample mean performance of each segment, $\bar{X}_{S(rs)}^{Seg}$, then the run sample means $\bar{X}_{R(r)}^{Seg}$ of these segment sample means, and then the grand sample mean \bar{X}^{Seg} of these run sample means, as the estimate of the baseline performance.

$$\begin{aligned}\bar{X}_{S(rs)}^{Seg} &= \frac{1}{N_{M(rs)}} \sum_m x_{rsm} && \text{(segment sample mean for segment } s \text{ in run } r) \\ \bar{X}_{R(r)}^{Seg} &= \frac{1}{N_{S(r)}} \sum_s \bar{X}_{S(rs)}^{Seg} && \text{(run sample mean for run } r) \\ \bar{X}_R^{Seg} &= \frac{1}{N_R} \sum_r \bar{X}_{R(r)}^{Seg} && \text{(grand sample mean)}\end{aligned}$$

To estimate the variance components from Equation (3), we start by computing the sample variances at the measurement, segment and run levels.

$$\begin{aligned}Var(X_{M(rs)}) &= \frac{1}{df_{M(rs)}} \sum_m (x_{rsm} - \bar{X}_{S(rs)}^{Seg})^2 && \text{(for measurements in segment } s \text{ of run } r) \\ Var(\bar{X}_{S(r)}^{Seg}) &= \frac{1}{df_{S(r)}} \sum_s (\bar{X}_{S(rs)}^{Seg} - \bar{X}_{R(r)}^{Seg})^2 && \text{(for segment sample means in run } r) \\ Var(\bar{X}_R^{Seg}) &= \frac{1}{df_R} \sum_r (\bar{X}_{R(r)}^{Seg} - \bar{X}_R^{Seg})^2 && \text{(for run sample means)}\end{aligned}$$

We estimate $Var(B)$ using the pooled variance of the individual measurements around the segment sample means:

$$\widehat{Var}^{Seg}(B) = \frac{1}{df_M} \sum_{r,s} df_{M(rs)} Var(X_{M(rs)}) \quad (5)$$

We estimate $Var(S)$ using the pooled variance of the sample segment means around the respective sample run means, subtracting the estimated contribution of the measurement variance and capping the estimate at zero for cases where the estimated contribution exceeds the segment variance:

$$\widehat{Var}^{Seg}(S) \approx \max\left(0, \frac{1}{df_S} \sum_r df_{S(r)} \left(Var(\bar{X}_{S(r)}^{Seg}) - \frac{1}{N_{S(r)}} \sum_s \frac{\widehat{Var}^{Seg}(B)}{N_{M(rs)}}\right)\right) \quad (6)$$

We estimate $Var(R)$ as the variance of the sample run means, subtracting the estimated contributions of the measurement and segment variances and capping the estimate at zero for cases where the estimated contributions exceed the run variance:

$$\widehat{Var}^{Seg}(R) \approx \max\left(0, Var(\bar{X}_R^{Seg}) - \frac{1}{N_R} \sum_r \frac{\widehat{Var}^{Seg}(S)}{N_{S(r)}} - \frac{1}{N_R} \sum_r \left(\frac{1}{N_{S(r)}} \sum_s \frac{\widehat{Var}^{Seg}(B)}{N_{M(rs)}}\right)\right) \quad (7)$$

Finally, we use the combined variance from Equation (3) as an estimate of $Var(\bar{X})$, and use it to construct the confidence interval.

Given that the sample variance of the run sample means $Var(\bar{X}_R^{Seg})$ already naturally combines $Var(R)$ with the appropriately scaled contributions from both $Var(S)$ and $Var(B)$, the gradual computation of the variance estimate may appear needlessly complicated. Note, however, that the gradual computation can provide a more accurate estimate by virtue of using more observations for some of the variance components and resolving cases of negative variance component estimates.

B Complete Evaluation Results

Table 7. A complete version of Table 2.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	21.87 (21.1-22.6)	26.16 (25.4-27.0)	$\times 1.20$	30.69	29.81	-0.88
akka-uct	19.79 (19.1-20.5)	22.04 (21.3-22.8)	$\times 1.11$	16.82	16.28	-0.55
future-genetic	20.1 (19.4-20.8)	21.64 (20.9-22.4)	$\times 1.08$	20.48	20.26	-0.22
scrabble	20.24 (19.5-21.0)	21.27 (20.5-22.0)	$\times 1.05$	47.71	47.47	-0.24
movie-lens	36.73 (35.8-37.7)	37.6 (36.6-38.6)	$\times 1.02$	17.08	16.99	-0.09
reactors	23.33 (22.6-24.1)	23.77 (23.0-24.6)	$\times 1.02$	18.45	18.32	-0.13
db-shootout	26.52 (25.7-27.4)	26.87 (26.1-27.7)	$\times 1.01$	58.32	57.53	-0.80
naive-bayes	22.38 (21.6-23.2)	22.62 (21.9-23.4)	$\times 1.01$	68.88	68.85	-0.03
philosophers	21.81 (21.1-22.6)	22 (21.3-22.8)	$\times 1.01$	31.11	31.15	+0.05
chi-square	22.07 (21.3-22.8)	22.12 (21.4-22.9)	$\times 1.00$	101.34	101.34	0.00
dec-tree	24.96 (24.2-25.8)	24.98 (24.2-25.8)	$\times 1.00$	23.92	23.92	0.00
neo4j-analytics	25.45 (24.7-26.3)	25.47 (24.7-26.3)	$\times 1.00$	36.62	36.62	0.00
mnemonics	18.1 (17.4-18.8)	18.1 (17.4-18.8)	$\times 1.00$	22.69	22.69	0.00
scala-kmeans	187.4 (185.4-189.4)	187.38 (185.4-189.4)	$\times 1.00$	91.32	91.33	+0.01
als	21.29 (20.6-22.0)	21.28 (20.6-22.0)	$\times 1.00$	12.67	12.69	+0.02
gauss-mix	28.93 (28.1-29.8)	28.92 (28.1-29.8)	$\times 1.00$	361.99	361.99	0.00
finagle-http	21.75 (21.0-22.5)	21.73 (21.0-22.5)	$\times 1.00$	16.50	16.50	0.00
fj-kmeans	25.18 (24.4-26.0)	25.17 (24.4-26.0)	$\times 1.00$	8.21	8.21	0.00
rx-scrabble	21.18 (20.5-21.9)	21.17 (20.4-21.9)	$\times 1.00$	10.87	10.88	+0.01
page-rank	26.58 (25.8-27.4)	26.56 (25.7-27.4)	$\times 1.00$	33.16	33.16	0.00
finagle-chirper	28.49 (27.6-29.4)	28.45 (27.6-29.3)	$\times 1.00$	57.67	57.68	+0.01
par-mnemonics	20.58 (19.9-21.3)	20.53 (19.8-21.3)	$\times 1.00$	40.78	40.79	0.00
scala-stm-bench7	22.96 (22.2-23.7)	22.87 (22.1-23.7)	$\times 1.00$	12.85	12.87	+0.02
dotty	21.92 (21.2-22.7)	21.78 (21.0-22.5)	$\times 0.99$	9.16	9.18	+0.01
log-regression	17.06 (16.4-17.7)	16.91 (16.3-17.6)	$\times 0.99$	27.01	27.03	+0.02

Table 8. A complete version of Table 3.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	10.09 (9.6-10.6)	12.38 (11.8-13.0)	$\times 1.23$	41.67	40.50	-1.17
akka-uct	8.53 (8.1-9.0)	9.76 (9.3-10.3)	$\times 1.14$	22.87	22.18	-0.69
future-genetic	9.3 (8.8-9.8)	10.08 (9.6-10.6)	$\times 1.08$	27.61	27.32	-0.28
scrabble	9.08 (8.6-9.6)	9.63 (9.1-10.1)	$\times 1.06$	64.00	63.67	-0.33
reactors	10.04 (9.5-10.6)	10.37 (9.9-10.9)	$\times 1.03$	25.18	25.03	-0.16
movie-lens	19.34 (18.7-20.1)	19.96 (19.3-20.7)	$\times 1.03$	23.30	23.18	-0.11
db-shootout	15.23 (14.6-15.9)	15.55 (14.9-16.2)	$\times 1.02$	78.83	77.88	-0.95
naive-bayes	10.53 (10.0-11.1)	10.73 (10.2-11.3)	$\times 1.02$	93.08	93.04	-0.04
philosophers	10.26 (9.8-10.8)	10.42 (9.9-10.9)	$\times 1.02$	41.67	41.73	+0.06
als	9.85 (9.4-10.4)	9.89 (9.4-10.4)	$\times 1.00$	17.21	17.24	+0.03
dotty	10.34 (9.8-10.9)	10.37 (9.9-10.9)	$\times 1.00$	12.30	12.32	+0.02
rx-scrabble	9.94 (9.4-10.5)	9.97 (9.5-10.5)	$\times 1.00$	14.57	14.58	+0.01
fj-kmeans	11.95 (11.4-12.5)	11.98 (11.4-12.5)	$\times 1.00$	11.15	11.15	0.00
finagle-chirper	16.33 (15.7-17.0)	16.36 (15.7-17.0)	$\times 1.00$	77.90	77.91	+0.02
scala-stm-bench7	11.17 (10.6-11.7)	11.18 (10.6-11.7)	$\times 1.00$	17.18	17.21	+0.03
page-rank	13.68 (13.1-14.3)	13.68 (13.1-14.3)	$\times 1.00$	45.12	45.12	0.00
dec-tree	12.35 (11.8-12.9)	12.35 (11.8-12.9)	$\times 1.00$	32.12	32.12	0.00
gauss-mix	21.01 (20.3-21.8)	21.01 (20.3-21.8)	$\times 1.00$	486.48	486.47	0.00
chi-square	10.69 (10.2-11.2)	10.68 (10.2-11.2)	$\times 1.00$	136.06	136.07	0.00
scala-kmeans	167.51 (165.6-169.4)	167.43 (165.5-169.4)	$\times 1.00$	116.47	116.48	+0.01
par-mnemonics	9.28 (8.8-9.8)	9.27 (8.8-9.8)	$\times 1.00$	55.14	55.15	0.00
neo4j-analytics	12.95 (12.4-13.5)	12.92 (12.4-13.5)	$\times 1.00$	49.38	49.39	+0.01
mnemonics	8.25 (7.8-8.7)	8.23 (7.8-8.7)	$\times 1.00$	30.60	30.60	0.00
finagle-http	9.94 (9.4-10.5)	9.92 (9.4-10.4)	$\times 1.00$	22.42	22.42	0.00
log-regression	7.64 (7.2-8.1)	7.58 (7.1-8.0)	$\times 0.99$	36.07	36.10	+0.03

Table 9. A complete version of Table 4.

Benchmark	Miss Rate ‰			Width ‰		
	Seg (99% CI)	Run (99% CI)	Δ	Seg	Run	Δ
scala-doku	15.83 (15.0-16.7)	19.64 (18.7-20.6)	$\times 1.24$	32.84	32.14	-0.70
akka-uct	12.79 (12.1-13.6)	14.71 (13.9-15.5)	$\times 1.15$	18.84	18.44	-0.40
future-genetic	15.06 (14.1-16.1)	17.25 (16.2-18.4)	$\times 1.15$	22.55	22.22	-0.33
dotty	17.82 (14.9-21.3)	20.05 (17.0-23.7)	$\times 1.12$	9.96	9.90	-0.06
scrabble	15.14 (12.5-18.4)	16.95 (14.1-20.3)	$\times 1.12$	52.25	51.74	-0.51
naive-bayes	17.58 (14.7-21.0)	19.39 (16.4-23.0)	$\times 1.10$	74.46	73.88	-0.58
als	18.2 (15.3-21.7)	19.67 (16.6-23.3)	$\times 1.08$	13.87	13.79	-0.08
log-regression	21.23 (18.0-25.0)	22.61 (19.3-26.5)	$\times 1.07$	29.34	29.21	-0.13
rx-scrabble	16.55 (13.8-19.9)	17.58 (14.7-21.0)	$\times 1.06$	11.81	11.78	-0.03
scala-stm-bench7	20.49 (17.4-24.2)	21.62 (18.4-25.4)	$\times 1.05$	14.06	13.98	-0.08
dec-tree	20.89 (17.7-24.6)	21.94 (18.7-25.7)	$\times 1.05$	25.83	25.82	-0.02
movie-lens	29.29 (25.5-33.6)	30.33 (26.5-34.7)	$\times 1.04$	18.79	18.69	-0.10
finagle-chirper	17.38 (14.5-20.8)	17.9 (15.0-21.4)	$\times 1.03$	60.46	60.38	-0.08
finagle-http	18.12 (15.2-21.6)	18.64 (15.7-22.1)	$\times 1.03$	17.95	17.91	-0.04
mnemonics	19 (16.0-22.5)	19.42 (16.4-23.0)	$\times 1.02$	24.99	24.96	-0.02
fj-kmeans	16.81 (14.0-20.2)	17.16 (14.3-20.6)	$\times 1.02$	8.88	8.87	-0.01
page-rank	16.46 (13.7-19.8)	16.8 (14.0-20.1)	$\times 1.02$	35.54	35.55	+0.01
par-mnemonics	21.01 (17.9-24.7)	21.44 (18.2-25.2)	$\times 1.02$	44.51	44.48	-0.03
chi-square	18.67 (15.7-22.2)	19.01 (16.0-22.5)	$\times 1.02$	110.66	110.60	-0.06
reactors	11.36 (9.1-14.2)	11.53 (9.3-14.4)	$\times 1.02$	21.57	21.53	-0.03
philosophers	17.79 (14.9-21.2)	18.05 (15.1-21.5)	$\times 1.01$	34.03	33.73	-0.30
gauss-mix	19.7 (16.6-23.3)	19.87 (16.8-23.5)	$\times 1.01$	372.91	372.97	+0.06
neo4j-analytics	24.31 (20.9-28.3)	24.48 (21.1-28.4)	$\times 1.01$	39.54	39.52	-0.02
scala-kmeans	176.73 (167.8-186.0)	176.99 (168.1-186.3)	$\times 1.00$	92.83	92.74	-0.08
db-shootout	13.06 (10.6-16.1)	13.06 (10.6-16.1)	$\times 1.00$	62.56	62.45	-0.11