# A Microbenchmark Case Study and Lessons Learned

Joseph (Yossi) Gil Keren Lenz * Yuval Shimron
Department of Computer Science
The Technion—Israel Institute of Technology
Technion City, Haifa 3200, Israel

## Abstract
The extra abstraction layer posed by the virtual machine, the JIT compilation cycles and the asynchronous garbage collection are the main reasons that make the benchmarking of Java code a delicate task. The primary weapon in battling these is replication: "billions and billions of runs", is phrase sometimes used by practitioners. This paper describes a case study, which consumed hundreds of hours of CPU time, and tries to characterize the inconsistencies in the results we encountered.

## Keywords
benchmark, measurements, steady-state

## 1. Introduction
Science is all about the generation of new and *verifiable* truths. In experimental sciences, this amounts to *reproducible* experimentation. In experimental computer science, reproducibility may seem easy, since we are accustomed to deterministic and predictable computing systems: We expect that hitting a key labeled 'a' shall always, in a bug-free environment, produce the letter 'a' on the screen.

It came therefore as a surprise to the authors of this paper that in our attempt to benchmark a certain small JAVA [1] function, a task known as *micro-benchmarking*, we encountered *inconsistent* results, even after neutralizing effects of well known perturbing factors such as garbage collection, just in time compilation, dynamic loading, and operating system background processes.

This paper tells the story of our micro-benchmarking endeavors, and tries to characterize the inconsistencies we encountered. Little effort is spent on trying to explain these; we believe this task requires a dedicate study which would employ different research tools (simulation and hardware probes come to mind). Our hope is that this report would contribute to better understanding of the phenomena we describe and promote the development of methods to eliminate these.

*contact author: lkeren@cs.technion.ac.il

*VMIL '11* October 24, 2011, Portland, Oregon.

### 1.1 Background: Benchmarking and Microbenchmarking of Java
Benchmarking of computer systems is a notoriously delicate task [9]. The increasingly growing abstraction gap between programs and their execution environments, most notably virtual machines, forces benchmarking, which could, in the early days, be carried out by counting program instructions, to use methods used in experimental, exact and social sciences.

Other issues brought about by the widening abstraction layer include the fact that the same benchmark on different platforms may yield different, and even contradictory results [4]. In addition, different compilers may apply different optimizations to source code, and therefore comparing two alternatives compiled with one compiler can lead to different results than the comparison of the same alternatives compiled using a different compiler.

Micro-benchmarking, measuring the performance of a piece of code (as opposed to assessing the performance of an application) is even more challenging. There are several subtle issues that can cause a benchmarker to draw wrong conclusions from an experiment. One such aspect is the use of dead code: micro-benchmarks often do nothing but calling the benchmarked code. Compilers may recognize such pattern and eliminate parts of the benchmark code, leading to code that runs faster than expected. In order to avoid that, it is not enough just to call the benchmarked code, but some extra code has to be introduced in the benchmark. However, doing so leads to a benchmark which measures the performance of the original and the extra code, introducing noise into the measurement.

Micro-benchmarking of JAVA functions raises its own set of intriguing issues.[1] There are two major hurdles for JAVA microbenchmarks: First, the Just In Time (JIT) compiler, may change the code as it executes. Secondly, the Garbage Collector (GC) may asynchronously consume CPU cycles during the benchmark. Multithreading may also be a concern, given the fact that an ordinary Java Virtual Machine (JVM) invocation, even of a non-threaded program, spawns a dozen of threads in addition to the main thread.

JIT compiles methods to different optimization levels, and is driven by timer-based sampling. Thus, in different runs different samples may be taken, leading to different methods being optimized at different levels, and to variations in running time of a benchmark across different JVM invocations, even on a single host [2].

[1] See Clikc's JavaOne'02 presentation, "How NOT To Write a Microbenchmark?", www.azulsystems.com/events/-javaone_2002/microbenchmarks.pdf

The first iterations of the benchmarked code include a large amount of dynamic compilation. Later iterations are usually faster both since they include less compilation and because the executed code is compiled and optimized [7]. The common wisdom of dealing with the presence of the JIT compiler is by conducting, prior to the beginning of the benchmark, *warm-up* executions of the benched code. The warm-up stage should be sufficiently long to allow the JIT compiler to fully compile and optimize the benchmarked code.

In dealing with the GC, one can allocate sufficiently large heap space to the JVM to reduce the likelihood of GC. Also, the intermittent nature of the GC effect on the results can be averaged by repeating the benchmark sufficiently many times.

The main tool of the trade is then replication: "billions and billions of runs", is the phrase sometimes used by practitioners. But, as we discovered, there are certain contributions to inconsistency which could not be mitigated by simple replication.

## 1.2 Findings

Having eliminated as much as we could the effects of JIT compilation, GC cycles and operating system and other environmental noise, we conducted benchmarking experiments consisting of a very large number of runs. Our case study revealed at least four different factors that contributed to inconsistency in the results:

1. *Instability of the Virtual Machine.* We show that different, seemingly identical invocations of the virtual machine may lead to different results, and that this difference (which may be in the order of $3\%$) is statistically significant. The impact of this instability may be more significant when two competing implementations are compared.

   This variation *increases* with the abstraction level, in the sense that when the JIT compiler is disabled, the discrepancy between different VM invocations increases.

   Observe that naive replication, within the same VM cannot improve the accuracy of the results. The remedy is in replication of the VMs.

2. *Multiple Steady States.* We observed that in some invocations, the VM converges to a certain steady state, and then, within the same invocation, leaps into a different steady state. The consequence of this phenomenon is that an averaged measurement within a single VM invocation may be misleading in not reflecting or characterizing these leaps.

3. *Multiple Simultaneous Steady States.* Further, we observed that in some invocations, the VM converges to two or more simultaneous steady states, in the sense that a given measurement has high probability of assuming one of number of distinct results, and small probability of assuming any intermediate value. A simple average of these results may be even more misleading report of the benchmarking process.

4. *Prologue Effects.* Finally, we demonstrate that the runningtime of the benchmarked code may be significantly affected by execution of *unrelated* prologue code, and that this effect persists even if the benchmarked code is executed a great number of times. This means that one cannot reliably benchmark two distinct pieces of code in a single program execution.

   The more disturbing conclusion is that the timing results obtained in a clean benchmarking environment may be meaningless when the benchmarked code is used in an application. Even if the application executes this code numerous times, the (unknown) application prologue effects may persist, rendering the benchmarked results meaningless.

## 1.3 Related Work

Non-determinism of benchmarking results is a well studied area. It is well known that modern processors are chaotic in the mathematical sense, and therefore analyzing the performance behavior of a complex program on a modern processor architecture is a difficult task [3]. Eechout et al. [6] showed that benchmarking result highly depends on the virtual machine; results obtained for one VM may not be obtained by another VM. Blackburn et al [4] showed that a JAVA performance evaluation methodology should consider multiple heap sizes and multiple hardware platforms, in addition to considering multiple JVMs.

Several attempts were made to suggest methodologies for benchmarking. One particularly interesting and increasingly widely used methodology *replay compilation* [10], which is used to control the non-determinism that stems from compilation and optimization. The main idea is the creation of a "compilation plan", based on a series of training runs and then the use of this plan to deterministically apply optimizations.

Georges, Buytaert and Eeckhout [7] describe prevalent performance analysis methodologies and point their statistical pitfalls. The paper presents a statistically robust method for measuring both startup time and steady-state performance.

Blackburn et al. [5] recommend methodologies for performance evaluation in the presence of non-determinism introduced by dynamic optimization and garbage collection in managed languages. Their work stresses the importance of choosing a meaningful baseline, to which the benchmark results are compared, and controlling free variables such as hosts, runtime environments, heap size and warm-up time. To deal with non-determinism the authors suggests three strategies: (i) using replay compilation, (ii) measuring performance in steady state, with JIT compiler turned off, and (iii) generating multiple results and apply statistical analysis to these.

Mytkowicz, Diwan, Hauswirth and Sweeney [12] discuss factors which cause measurement bias, suggest statistical methods drawn from natural and social sciences to detect and avoid it. To avoid bias, the paper suggests a method which is based on applying a large number of randomized experimental setup, and using statistical methods to analyze the results. The method for detecting bias, causal analysis, is a technique used for establishing confidence that the conclusions drawn from the collected data are valid.

Lea, Bacon and Grove [11] claim that there is a need to establish a systematic methodology for measuring performance, and to teach students and researchers this methodology.

Georges et al. [8] present a technique for measuring processor-level information gathered through performance counters and linking that information to specific methods in a JAVA program. They argue that "different methods are likely to result in dissimilar behavior and different invocations of the same method are likely to result in similar behavior".

**Outline.** *The remainder of this paper is organized as follows. In Section 2 we describe the setting of our experimental evaluation, the hardware and software used, and the benching environment.*

*Section 3 shows that different, seemingly identical, invocations of the JVM may converge to distinct steady states. This finding is further examined in Section 4, which also shows that the discrepancy is greater when the JIT compiler is not present. Section 5 shows that even a single JVM invocation may have multiple steady states, and that, further, these multiple steady states may be simultaneous. The impact of prologue execution is the subject of Section 6. Section 7 concludes and suggests directions for further research.*

# 2. Settings of the Benchmark

## 2.1 Hardware and Software Environment

To minimize effects of disk swapping and background processes on benchmarking, we selected a computer with a large RAM and multiple cores. Measurements were conducted on a *Lenovo ThinkCentre* desktop computer, whose configuration is as follows: an *Intel Core 2 Quad CPUQ9400* processor (i.e., a total of four cores), running at *2.66 GHz* and equipped with *8GB* of RAM. Hosts used in specific experiments are described below.

During benchmarking all cores were placed in "performance" mode, thereby disabling clock rate and voltage changes.

On this computer, we installed *Ubuntu version 10.04.2 (LTS)* and the readily available open JAVA development kit (*OpenJDK, IcedTea6 1.9.8*) including version 1.6.0_20 of the JAVA Virtual Machine (specifically *6b20-1.9.8-0ubuntu1~10.04.1*). Other software environments used in specific experiments are described below.

To minimize interruptions and bias due to operating system background (and foreground) processes, all measurements took place while the machine was running in what's called "single-user" mode (`telinit 1`), under root privileges in textual teletype interaction (no GUI), and with networking disabled.

## 2.2 Benchmark Algorithm

Our atomic unit of benchmarked code was defined as an instance of a class implementing interface `Callable` (defined in package `java.util.concurrent`). This interface defines a single no-arguments method, which returns an `Object` value.

Given an implementation of `Callable` our benchmark algorithm works in three stages: (i) *estimation*, (ii) *warm-up*, and (iii) *measurement*. The purpose of the estimation phase is to obtain a very rough estimate of execution time of the parameter. In the warm-up stage, this estimation is used for warming up the given `Callable`, i.e., iterating it until the JIT compiler has realized its full potential, for a specified warm-up period (five seconds in our experiments). In the measurement phase, the running time of the parameter is measured repeatedly. Each such measurement executes the parameter $m$ times, where $m$ is so selected that the measurement duration is roughly that of a parameter $\tau$.

All phases are based on making a number of sequences of iterations of the `Callable` object parameter. Each iteration sequence is monitored for garbage collection cycles, JIT compilation cycles, and class loading/unloading events by probing the MX-beans found in class `ManagementFactory` (which is defined in package `java.lang.management`). The timing result of a sequence is discarded if any of these events happen. Further, the detection of JIT compilation cycles late in the warm-up period, increases its duration. And, the repeated detection of garbage collection cycles triggers shortening of a measurement sequence so as to decrease the likelihood of further interruptions of this sort.

## 2.3 Benchmarked Code

At the focus of our attention we placed the implementation of class `HashMap`, arguably one of the most popular classes of the JAVA runtime environment. We used version 1.73 of the code, (dated March 13, 2007) due to Doug Lea, Josh Bloch, Arthur van Hoff and Neal Gafter. To preclude the option of pre-optimization of code present in JAVA libraries, we benchmarked a *copy* of the code in our benchmarking library.

Within this class, we concentrated in the throughput of function `V get(Object key)` which retrieves the value associated with a key stored in the hash table. Listing 1 depicts this function's code.

**Listing 1** Java code for benchmarked function (function `get` in class `HashMap`)

```
1 public V get(Object key) {
2   if (key == null) return getForNullKey();
3   int hash = hash(key.hashCode());
4   for (
5     Entry<K,V> e=table[hash & table.length-1];
6     e != null;
7     e = e.next
8   ) {
9     Object k;
10    if (
11          e.hash == hash
12      &&
13          ( (k = e.key) == key
14            ||
15              key.equals(k)
16          )
17    )
18      return e.value;
19  }
20  return null;
21 }
```

This function includes iterations, conditionals and logical operations, array access and pointer dereferencing. Examining function `hash` which is called by `get` in line 3 (Listing 2), we see that the benchmarked code includes also bit operations.

**Listing 2** JAVA code for function `hash` (invoked from `get` in class `HashMap`)

```
static int hash(int h) {
  h ^= h >>> 20 ^ h >>> 12;
  return h ^ h >>> 7 ^ h >>> 4;
}
```

Our measurements never pass a **`null`** argument, so there is no need to examine the code of function `getForNullKey`. Also, we only exercise `get` in searching for keys which are *not present* in the hash table. The call `key.equals(k)` is thus executed only in the case that the cached hash value (field `hash` in the entry) happens to be equal to the hash value of the searched key. This rare event does not happen in our experiments, so, the call `key.equals(k)` is never executed.

All in all, the benchmarked code is presented in its entirety in listings 1 and 2. Notably, the benchmarked code does not include any

virtual function calls—foregoing virtual optimization techniques. Viable optimization techniques include e.g., branch prediction, prefetching, inlining, and elimination of main memory access operations. Also, the code does not allocate memory, nor does it change the value of any reference. As it turns out, there was not a single garbage collection cycle in a sequence of one billion consecutive calls to function `get`.

The precise manner in which `get` was exercised was as follows:

1. Fix $n = 1,152$.
2. Make $n$ pairs of type $\langle$key, value$\rangle$ , where both the `String` key and the `Double` value are selected using a random number generator started with a fixed seed.
3. Create an empty `HashMap<String, Double>` data structure and populate it with these pairs.
4. Create an array of $n$ fresh keys of type `String`, where keys are selected using the same random number generator.
5. The benchmarked procedure, `getCaller()`, then iterates over the keys array, calling `get` for each of its elements.
6. The *throughput* of function `get` is defined as the running-time of `getCaller()` divided by $n$.

## 3. No Single Steady State

We conducted $v = 7$ independent *invocations* of our benchmarking program, i.e., each invocation uses a freshly created VM. These invocations were consecutive, with a script invoking the VM $v$ times.

In each invocation the VM carried out $r = 20,000$ *iterations*, i.e., $r$ executions of the measurement procedure described above (after a single estimation stage followed by a single warm-up stage). We shall call such an iteration a *measurement session*. A measurement session returns a *measurement result* (result for short).

A measured result of $t$ (for function `getCaller()`) gives a throughput of $t/n$ for function `get()`. Also, a measurement session of length $\tau$ calls `getCaller()` about $\tau/t$ times. The throughput in our experiment was about 15.3ns per each `get()` call.

Overall, in our case, $\tau/t \approx 6,000$. If the time of each of the iterations of `getCaller()` were independent, then, by the *central limit theorem* the $r$ results collected in an invocation should be close to a normal distribution.

Overall, in the experiment described in this section, the number of executions of function `get`, was about

$$v \times r \times \frac{100\text{ms}}{15.3\text{ns} \times 1,152} \approx 0.8 \times 10^9$$

Table 1 summarizes the essential statistics of the distribution of measurement results in our experiment. (The "mad" column represents the median of absolute deviations from the median—a statistics which is indicative of the quality of the median statistics and robust to outliers.)

The values in each column, i.e., the same statistics of different invocations, appear quite similar. The median is close to the mean; these statistics indicate a throughput of about 15.2–15.3ns/operation. The standard deviation is about 0.1ns, which is about 0.7% of the mean.

| No. | Mean | $\sigma$ | Median | Mad | Min | Max |
|---|---|---|---|---|---|---|
| 1 | 15.27 | 0.0802 | 15.27 | 0.0509 | 14.95 | 15.69 |
| 2 | 15.17 | 0.0768 | 15.17 | 0.0454 | 14.74 | 15.69 |
| 3 | 15.24 | 0.0726 | 15.24 | 0.0451 | 14.93 | 15.67 |
| 4 | 15.38 | 0.0727 | 15.38 | 0.0430 | 15.06 | 15.81 |
| 5 | 15.29 | 0.0867 | 15.30 | 0.0495 | 14.95 | 15.68 |
| 6 | 15.10 | 0.0722 | 15.11 | 0.0420 | 14.83 | 15.51 |
| 7 | 15.24 | 0.0929 | 15.25 | 0.0559 | 14.87 | 15.60 |

**Table 1: Essential statistics (in ns/operation) of the distribution of measurement results in seven consecutive invocations of a 20,000 iterations measurement, $\tau = 100$ms .**

The extreme values appear to be well behaved in that they deviate from the mean by about 5%.

The mean throughput in the different invocations spans a 0.28ns $\approx$ 2% range which may seem reasonable. A graphical plot of the distribution of the same results, as done in Figure 1, reveals a somewhat different picture. A quick look at the figure, suggests that the results in two different invocations are drawn from close, yet *distinct*, distributions.

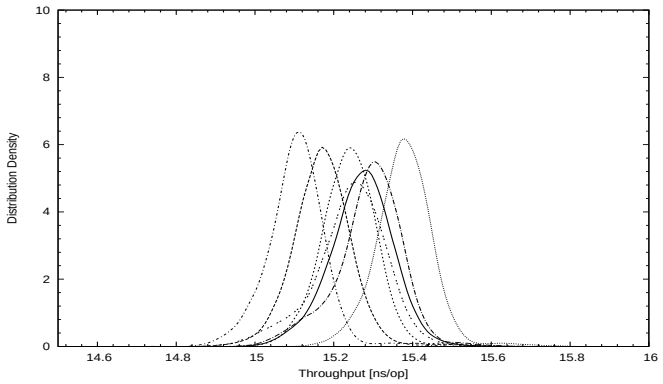



**Figure 1: Distribution of benchmark results in seven invocations of a 20,000 iterations experiment, $\tau = 100$ms**

Each curve in the figure represents the distribution function of the $r$ values obtained in a single invocation. Here, and henceforth, all curves were smoothed out by using a *kernel density estimation*, where a Gaussian was used as the windowing function, and where a bandwidth free parameter was $h$ (lower values means less estimation). The value of $h$ selected in all figures is

$$h = \min_i \frac{4}{3r}^{1/5} \sigma_i,$$

where $i$ ranges over all distribution plotted in the same figure, $\sigma_i$ being the standard deviation of the $i^{th}$ distribution. (In normal, Gaussian distribution, the optimal value of $h$ is $(4/3r)^{1/5}\sigma$.)

Also, here and henceforth, the smoothed distribution was scaled down by multiplying the density by $1/r$, so as to make the total area under the curve 1.

A quick "back of an envelope" analysis confirms that the difference between the means of the different invocations is statistically significant.

To increase certainty in our findings, we repeated the experiment with $\tau = 1,000\text{ms} = 1\text{sec}$, except that for practical reasons, we set $v = 5$. The resulting distributions are depicted in Figure 2.

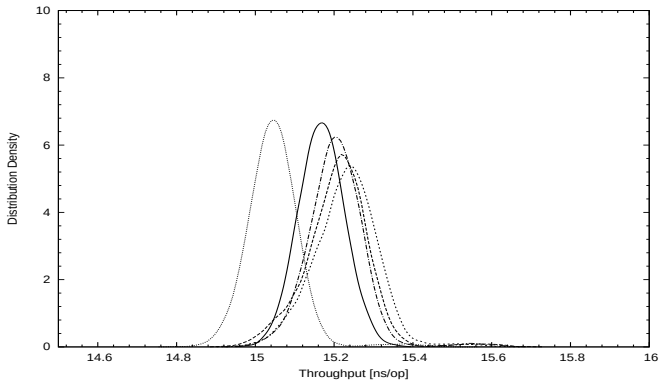



**Figure 2: Distribution of benchmark results in five invocations of a 20,000 iterations experiment, $\tau = 1,000$ms**

To appreciate the scale, note that each invocation this time executed function `get` $\approx 1.1$ billion times. The actual benchmarked function, i.e., the function which wraps `get`, calling it $n = 1,152$ times, is called about 60,000 times in each of the 20,0000 iterations.

The overall picture portrayed by Figure 1 and Figure 2 is that invocations of the JVM, even in our fairly clean benchmarking does not converge to a single steady state. It seems as if the mythical single convergence point is replaced by multiple steady states, or even a range of convergence centers. The figures suggest that in this particular case, this range spans about $2\%$ of the measured value.

The consequence is that repeated invocations of a single benchmark shall yield different results, even if all factors are kept equal. Further, when one compares competing algorithms or implementations, this error accumulates and may bring about wrong conclusions.

It is interesting to see that each of the distributions plotted in Figure 1 resembles normal distribution. Figure 3 compares one such distribution with that of a normal distribution with the same mean and standard deviation.

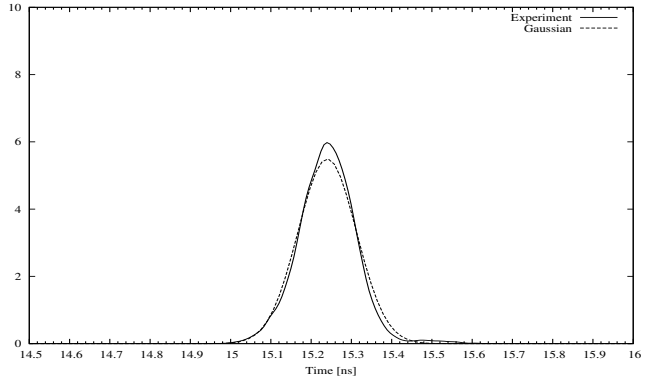



**Figure 3: Distribution of benchmark results in an invocation of a 20,000 iterations experiment, $\tau = 100$ms, compared to a Gaussian with the same mean and standard deviation.**

Evidently, the experimental distribution is even more peaked than a Gaussian. Applying a JarqueBera test we obtained that it very highly unlikely $p < 10^{-5}$ that the measurement sessions' results were drawn from a normal distribution. This fact is an initial indication that the measurement sessions' results are *not* independent.

Still, we may conservatively assume that the distribution of the results is normal, for e.g., computing the standard error. This is useful for estimating accuracy of the mean, even for small number of iterations. In selecting, for example, $r = 10$, the standard deviation would be about $0.1$ns $\times \sqrt{10/9}$, i.e., a standard error $\approx 0.06$ns which gives about $0.2\%$ measurement error. In selecting a greater number of iterations the measurement error could be further decreased, i.e., for $r = 100$, executing in about ten seconds, the measurement error would be about $0.1\%$.

The difficulty in applying this analysis for benchmarking is that different invocations lead to slightly different distributions centered at different values. The inherent $2\%$ discrepancy of these cannot be improved by increasing $r$. Further, as we shall see below, even a single invocation of the JVM may have multiple convergence points.

# 4. Examining the No Single Steady State Judgment

One may hypothesize that the discrepancy of the results can be explained by the fact that the invocations were consecutive. The CPU, hardware cache and other factors may have a long "learning" curve. These can, presumably, act together so that the computing carried out in one experiment somehow improves the results obtained in the subsequent experiment, despite execution in a different process of the operating system and in a distinct address space.

A quick look at Table 1 indicates that this could not be the case, since later invocations do not yield lower values than those of earlier invocations. The respective statistics of the longer experiment (Figure 2), provided in Table 2 do not portray a different picture.

| No. | Mean | $\sigma$ | Median | Mad | Min | Max |
|---|---|---|---|---|---|---|
| 1 | 15.17 | 0.0585 | 15.17 | 0.0392 | 14.95 | 15.40 |
| 2 | 15.21 | 0.0859 | 15.21 | 0.0483 | 14.88 | 15.73 |
| 3 | 15.23 | 0.0828 | 15.23 | 0.0510 | 14.90 | 15.68 |
| 4 | 15.05 | 0.0687 | 15.04 | 0.0391 | 14.75 | 15.57 |
| 5 | 15.21 | 0.0756 | 15.20 | 0.0429 | 14.92 | 15.69 |

**Table 2: Essential statistics (ns/operation) of the distribution of measurement results in five consecutive invocations of a 20,000 iterations measurement, $\tau = 1,000$ms.**

We see in Table 2 that the centers of the distributions are the same as in Table 1. The standard deviation and other statistics are also about the same. And, Table 2 *does not* suggest that a certain invocation is favorably affected by earlier invocations.

Is the absence of the single steady state an artifact of instability of the triggering of optimization algorithms within the JIT? To answer this question, we repeated the experiment while disabling the JIT compiler (i.e., by passing `-Djava.compiler=NONE` flag to the JVM). The resulting distributions are depicted in Figure 4.

Comparing the figure with Figure 1 and Figure 2, we see that in the absence of the JIT compiler the distributions and their steady states are even more clearly separated. Examining Table 3, which summarizes the respective statistics of these distributions, strengthens our conclusion.

As can be seen in the table, there is a 9% range span between the best result, in which the average throughput was 212ns/operation, and the worst result of 231.04ns/operation. This difference

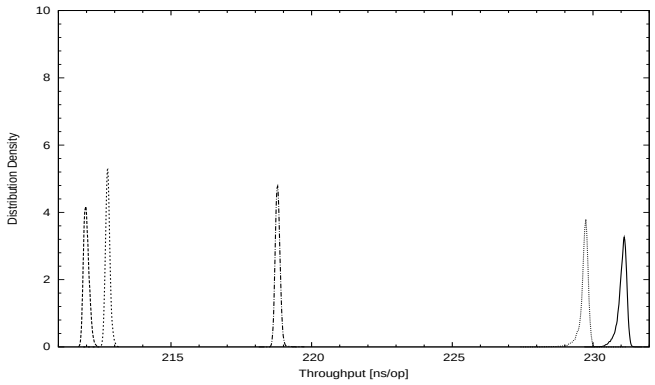



**Figure 4: Smoothed distribution function of benchmark results, JIT compiler disabled, in five invocations of a 20,000 iterations measurement, $\tau = 100$ms**

| No. | Mean | $\sigma$ | Median | Mad | Min | Max |
|---|---|---|---|---|---|---|
| 1 | 231.04 | 0.1642 | 231.07 | 0.0888 | 229.69 | 231.76 |
| 2 | 212.00 | 0.0980 | 211.99 | 0.0651 | 211.71 | 212.44 |
| 3 | 212.76 | 0.0819 | 212.75 | 0.0508 | 211.89 | 213.20 |
| 4 | 229.68 | 0.1868 | 229.72 | 0.0747 | 227.42 | 231.19 |
| 5 | 218.80 | 0.0921 | 218.79 | 0.0554 | 218.12 | 219.74 |

**Table 3: Essential statistics (in ns/operation) of the distribution of measurement results, JIT compiler disabled, in five consecutive invocations of a 20,000 iterations measurement, $\tau = 100$ms.**

is even more significant considering the fact that the coefficient-of-variance (CoV), defined as the standard deviation divided by the mean, is smaller in the absence of the JIT. The greatest CoV in Table 3 is $0.07\%$, while the smallest CoV in tables 1 and 2 is $0.39\%$.

As argued by Blackburn el al. [4], the choice of architecture may affect the reported results substantially. Therefore, we repeated the experiments on six additional different computing systems. Table 4 summarizes the hardware and software characteristics of the additional systems on which the experiments were conducted.

| Id | CPU | Clock [GHz] | RAM [GB] | OS | JVM | Compiler |
|---|---|---|---|---|---|---|
| **A** | Intel Pentium M | 1.70 | 2 | Linux Mint Xfce Edition | OpenJDK (IcedTea6 1.8.7) 1.6.0_18 | OpenJDK 1.6.0_18 |
| **B** | Intel Pentium 4 | 2.80 | 2 | Ubuntu 8.04 | HotSpot Client VM, 1.6.0_24-b07 | Sun 1.6.0_24 |
| **C** | Intel Core 2 Duo T5870 | 2.00 | 2 | Ubuntu 10.04 | OpenJDK (IcedTea6 1.9.9) 1.6.0_20 | Eclipse Helios |
| **D** | Intel Core 2 Duo T7500 | 2.20 | 4 | Ubuntu 11.04/64 | OpenJDK 64-Bit Server VM | OpenJDK 1.6.0_22 |
| **E** | Intel Core i3 | 2.93 | 2 | Linux Mint 11 (Katya) | OpenJDK (IcedTea6 1.10.2) 1.6.0_22 | Eclipse Helios |
| **F** | Intel Core 2 Duo T9600 | 2.80 | 4 | Ubuntu 10.10 | OpenJDK (IcedTea6 1.9.9) 1.6.0_20 | OpenJDK 1.6.0_20 |

**Table 4: Hardware and software characteristics of computing systems used for additional experiments**

Note that all systems were operating under the Gnu/Linux operating system, and were all equipped with 2GB or more of internal memory. The main variety is the range of CPU models, along with their respective cache levels.

We carried out $v = 7$ VM invocations on each of systems A—F, where each invocation includes 20,000 iterations of measurements spanning $\tau = 100ms$ each. The distribution density functions are plotted in Figure 5.

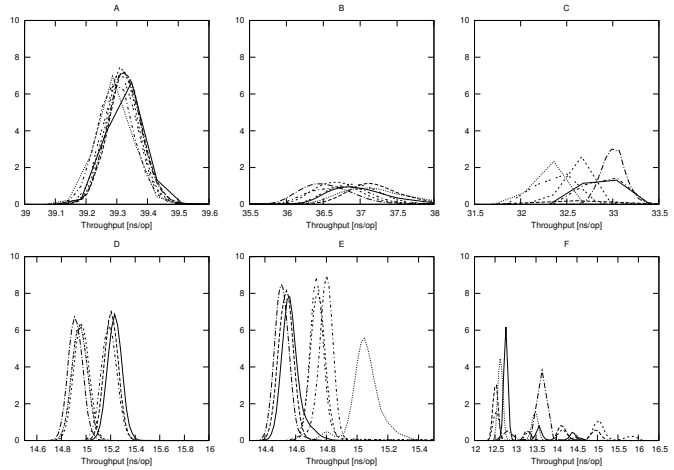



**Figure 5: Distribution of benchmark results, in six additional different computing systems, $v = 7$, $r = 20,000$, $\tau = 100$ms**

On the least modern system A, we see almost identical, overlapping distributions. The mean of all these distributions are identical up to three significant digits (39.4ns/op, while the standard deviation is $0.2\%$ of this value. The fact that similar results for different invocations were obtained in this system indicates that the multiple steady-state phenomenon is not due to an inherent problem in our benching system, but rather a behavior which is observed only in certain environments.

In the slightly more modern system B, the distribution are further apart, with the means varying between 36.6ns/op and 37.2ns/op (a $1.6\%$ range), while the standard deviations being about $1\%$ of the mean.

We clearly have distinct steady states in systems C, D, and E. In system D, there are two distinct multiple steady states, one at 15.2ns and another at 14.95ns. These are about $1.5\%$ apart. In system E there appear to be three distinct steady states which are $1.7\%$ apart. System F exhibits the most interesting behavior: we see on it that there are multiple *local* maxima of the distribution density function.

Overall, the curves in Figure 5 lead us to conclude that the phenomena of absence of a single steady state for a different VM invocations on the same platform is not unique to our specific benchmarking environment, and that it is typical of more modern architectures.

# 5. Shift of Steady State

Having examined the host system, it is only natural to wonder whether the benchmarked code, i.e., function `get` may contribute its share to the indeterminism we encountered. To this end, benchmarking was repeated for four other functions written specifically for this study:

1. `arrayBubbleSort()`, which randomly shuffles the values in a fixed array of **`int`**s of size $\ell_1$, and then resorts these using a simple bubble sort algorithm.

2. `listBubbleSort()`, which randomly shuffles the values in a fixed linked list of `Integer`s of size $\ell_2$ (without changing the list itself), and then resorts these (again, without modifying the list) using the same bubble sort algorithm.

3. `xorRandoms()`, which computes the X-OR of $\ell_3$ consecutive pseudo random integers drawn from JAVA's standard library random number generator.

4. `recursiveErgodic()`, which applies many recursive calls involving the creation and destruction of lists of integers, all implemented using the type `ArrayList<Integer>` to create an array of pseudo-ergodic sequence of length $\ell_4$.

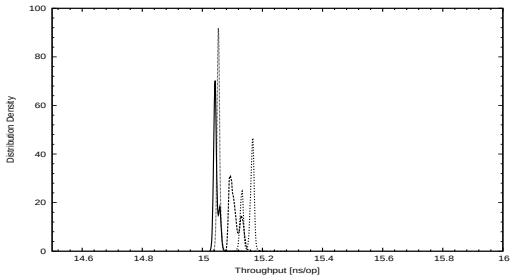
The implementation of these made the results of the computation externally available so as to prevent an optimizing compiler from optimizing the loops out. Note that each of the four functions has its own particular execution profile: Function `arrayBubbleSort` is characterized best by array access operations, while pointer dereferencing is the principal characterization of `listBubbleSort`. Function `xorRandoms()` involves a lot of integer operations and is therefore mostly CPU bound, while memory allocation is the main feature of `recursiveErgodic()`.

The values $\ell_1$, $\ell_2$, $\ell_3$ and $\ell_4$ were so selected to make the running-time of each of these four functions about the same as the function that wraps `get()`. Thus, for $\tau = 100$ms we have again about 6,000 calls of the benchmarked function, and, we expect again of the sessions' results to be normally distributed. Further, to make it easier for humans to compare the results, we selected integers $n_1, n_2, n_3$ and $n_4$ and divided the resulting running-time values by these numbers so as to make the "throughput" values close to those of the throughput of `get()`.

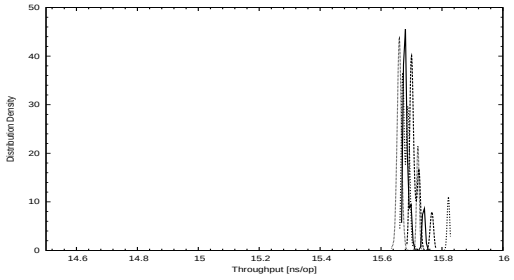
We conducted $v = 3$ VM invocations of each of these functions. As usual, each such invocation constituted $r = 20,000$ iterations of measurement sessions of $\tau = 100ms$. Each measurement session thus included about 60,000 executions of the benchmarked function. The distribution density curves for this experiment are plotted in Figure 6.

Benchmarking function `recursiveErgodic()` (Figure 6(d)) gives similar results to what we have seen before: Gaussian like distribution in each invocation, but different means in different invocations, i.e., the lack of single steady state.

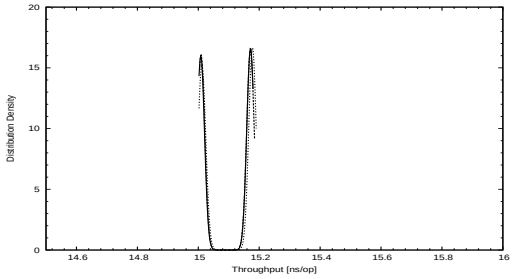
Most interesting is Figure 6(c), depicting the distribution density of the measurements of function `xorRandoms()`. The three different curves of the three different invocations are so close to each other that they cannot be distinguished. This similarity is encouraging, since it tells us that the variations between the different invocations that encountered are not an artifact of the benchmarking environment. Moreover, we can conclude that the bell-shape distribution of results is not due to the usual fluctuations in measurement errors, but is inherent to the performance of the benchmarked function. Said differently, the time performance of (some) benchmarked functions obeys a Gaussian like distribution, whose variation is significantly greater than measurement errors introduced by the measurement environment. Also interesting is the fact that the distribution density has two very sharp peaks, one at the level of 15ns/op



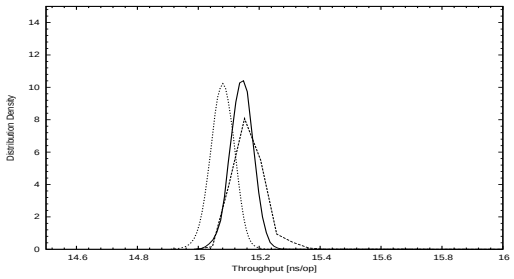

(a) Function `arrayBubbleSort()`




(b) Function `listBubbleSort()`




(c) Function `xorRandoms()`

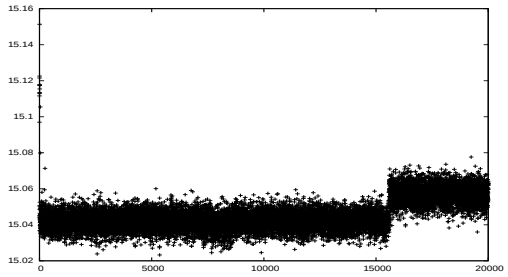



(d) Function `recursiveErgodic()`

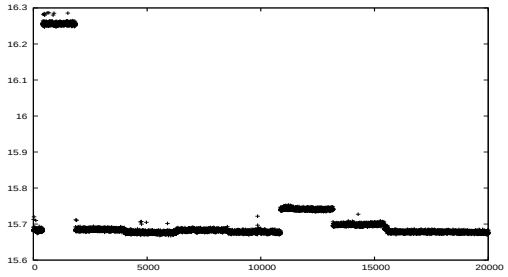
**Figure 6: Distribution density of throughput of four auxiliary JAVA functions, $v = 3$, $r = 20,000$, $\tau = 100$ms.**

throughput and the other at 15.2ns/op. Similar, multi-peaked distribution occurs also in `arrayBubbleSort`. The peaks are not located at identical locations in different executions but they are even sharper (observe that the different $y$-axis scales).

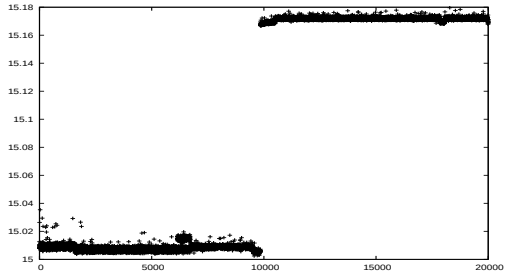
This multi-peak phenomenon repeats itself also in Figure 6(b), so it could not be a coincidence. To understand this phenomenon better, we plotted the measurement result vs. session number for each of the four functions (Figure 7). To conserve space, the figure is restricted to the *first* invocation of each function.



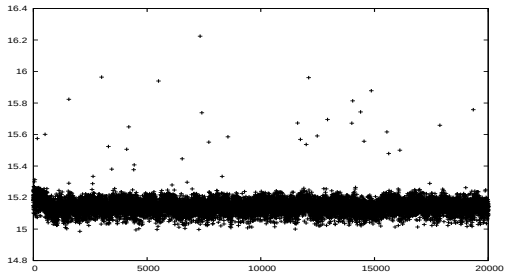

(a) Function `arrayBubbleSort`




(b) Function `listBubbleSort`




(c) Function `xorRandoms`




(d) Function `recursiveErgodic`

**Figure 7: Measurement result (throughput ns/ops) vs. session No. of the four auxiliary benchmarked functions.**

Figure 7(d) shows that the results obtained in the $r$ measurement sessions of function `recursiveErgodic()` are distributed more or less as expected, with the center of fluctuations not changing throughout the invocation.

Functions `arrayBubbleSort()`, `listBubbleSort()` and `xorRandoms()` are different in that the center of fluctuation, or "steady state" so to speak, changes during the run; these changes are always "step wise" rather than gradual.

The multiple steady states observed in Figure 6 correspond to these "quantum" leap of the center of fluctuations. Examining Figure 6(c) together with Figure 7(c) suggests that the quantum leap in function `xorRandoms` always occurs at about the same measurement session during the invocation.

In contrast, the local maxima of the distribution functions the different invocations of `arrayBubbleSort` and `listBubbleSort` do not coincide. This fact is an indication that the leaps of center of fluctuation are not necessarily deterministic. To appreciate this point better, consider Figure 8, in which the measurement session result is plotted against the session number for the *second* invocation of `listBubbleSort`.

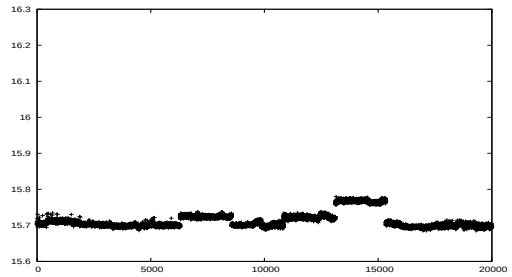
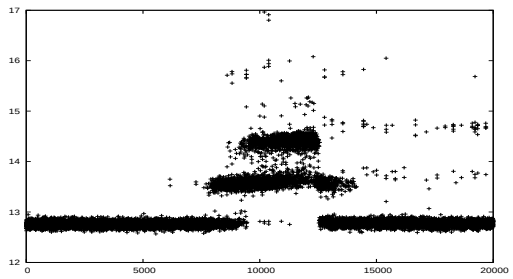


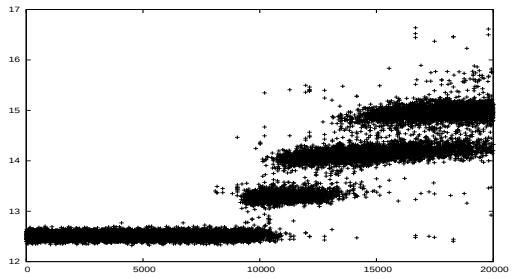

**Figure 8: Measurement result (throughput ns/ops) vs. session No. of the second invocation of benchmark of function `listBubbleSort`.**

Now, the comparison of the figure with Figure 7(b), depicting the first invocation of function `listBubbleSort()`), clearly indicates that the leaps between the different steady states do not occur at the same point during the invocation.

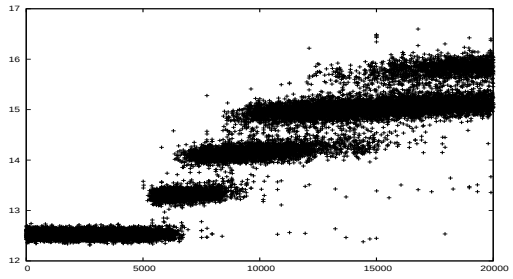
Recall that in the previous section, multiple local maxima were observed also in the distribution of results of benchmarking function `getCaller()`. It is therefore interesting to examine the progression of these results with time, as depicted in Figure 9.



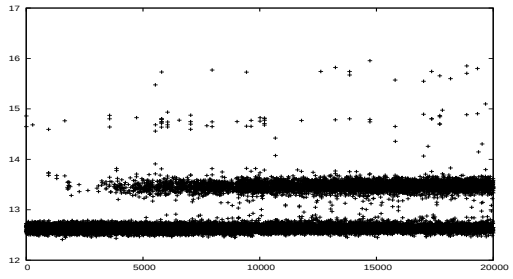
(a)



(b)



(c)



(d)

**Figure 9: Measurement result (throughput ns/ops) vs. session No. of the four invocations of benchmark of function `getCaller` on system F (see Table 4 and Figure 5 above),** $r = 20,000$**,** $\tau = 100$**ms.**

As in Figure 7, we see that ***(i)*** the execution time of the benchmarked function `getCaller()` on System F fluctuates about multiple steady states; ***(ii)*** these steady states change in time; and, ***(iii)*** the leaps between steady states occur in a seemingly non-deterministic fashion. However, unlike what we have seen before, Figure 9 shows that multiple steady states may occur *simultaneously*, e.g., there are two equally weighted steady states during most of the invocations depicted in Figure 9.

## 6. The Effect of Prologue Runs

One of the frustrating issues in our attempts of benchmarking function `get()` was our suspicion that the results seemed to have changed a bit when minor modifications were applied to portions of the enveloping code, e.g., code in charge of command line arguments processing, measurement, bookkeeping and tuneup. To eliminate this yet unverified suspicion, all our experiments were carried out within a fixed version of the enveloping code.

Further, to check whether code executed prior to benchmarking may affect the results, we carried out an experiment in which actual benchmarking commenced after a *prologue* session in which $m$ non-benching calls were made to foreign code. To make this foreign code similar to the benchmarked code, the calls in the prologues were made to an alternative function `getPrimeCaller`, which in turn made $n = 1,152$ calls to an alternative function named `getPrime()`. Function `getPrime()` does a similar, failing search on an alternative implementation of class `HashMap`, and its execution time is similar to that of `get()`.

After these $m$ *prologue calls*, benchmarking of `getCaller()` carried out as usual. The value of $m$ was deliberately kept small. We maintained $m \leq 100$. Recall that in a single benchmarking session whose length is $\tau = 100$ms, about $6,000$ calls are made to function `getCaller()`; in a benchmarking invocation with reasonably large $r$, both the number of prologue calls and the total prologue time are infinitesimally smaller than benchmarking calls and benchmark time.

We measured the average throughput in running $r = 1,000$ measurement sessions after $m$ prologues runs. The results are depicted in Figure 10.

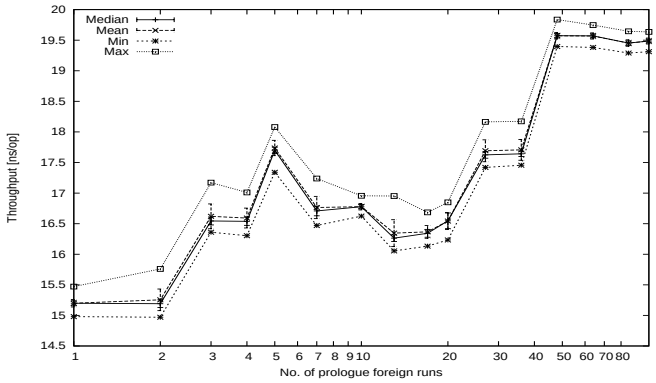



**Figure 10: Throughput vs.** $m$**, number of foreign prologue calls,** $r = 1,000$**,** $\tau = 100ms$**.**

The picture portrayed by the figure is worrying; $m = 50$ prologue calls, whose total duration is a millisecond or two, were sufficient to make a very noticeable impact on an invocation whose duration is about 100 seconds. The figure demonstrates an increase by about $30\%$, from $15.2$ns to $19.5$ns. Further, a noticeable increase, albeit smaller is made even after $m = 3$ prologues calls.

Does this prologue effect ever wear out? Figure 11 plots our measurements after $m = 50$ prologue calls for increasing values of $r$.

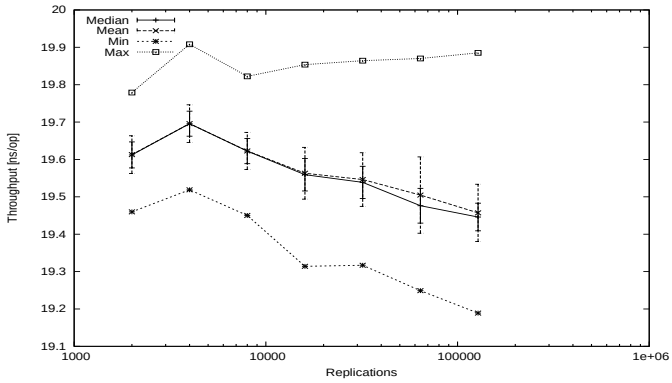



**Figure 11: Throughput vs. No. of runs after $m = 50$ foreign prologue runs, $\tau = 100ms$.**

We see in the figure that the effect of prologue runs does not wear out even after making $r = 128,000$ measurement sessions. (The duration of this last execution was about 4 hours.)

## 7. Discussion and Further Research

Section 3 showed, again, what many have previously observed—that a restart of the JVM, may yield different results. In attempt to study and understand this better (Section 4), we found that the problem, which is aggravated in more modern CPUs, is not entirely the blame of the JIT compiler as some may have thought before[2]; surprisingly, diversity and inconsistency of results are greater when the JIT compiler is disabled.

Delving deeper, we demonstrated that even a *single* JVM invocation may have multiple convergence points, and this multiplicity may even occur simultaneously. It may be argued that this multiplicity can be compensated for by sufficiently large restarts of the virtual machine. The concern is that micro-benchmarking of a certain code is used primarily to predict the impact of this code when integrated in a larger program. Brute force replication of the benchmark, without understanding the underlying issues, may yield results which are meaningless for an enveloping application program.

Worse, Section 6 demonstrated that the benchmarking results can be affected by prologue execution however infinitesimally short it is. This later finding, if not understood better (it could be a subtle artifact of "compilation planning") may cast a doubt on the value of benchmarking for the evaluation of real programs.

Our works stops short of trying to understand the effects we saw: such a task requires an analysis of the deep stack of hardware and software abstractions. But, perhaps before such a study starts, more experimentation is required: one needs to examine the effects of other prologue executions, no-JIT runs on other machines, with, and without prologue executions, study other benchmarked code, with and without JIT, etc. Of course, judicious planning is an absolute necessity in finding a meaningful and effective research path in this exponentially huge design space.

## 8. References

[2] see e.g., J. Bloch's "Performance Anxiety", Devoxx 2010, `http://www.parleys.com/#st=5&id=2103`


[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, 1996.

[2] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Proc. of the 17th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'02)*.

[3] H. Berry, D. Gracia Pérez, and O. Temam. Chaos in computer performance. *Chaos*, 16(1):013110–013110–15, 2006.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In P. L. Tarr and W. R. Cook, eds., *Proc. of the 21st Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'06)*.

[5] S. M. Blackburn, K. S. McKinley, R. Garner, C. Hoffmann, A. M. Khan, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovik, T. VanDrunen, D. von Dincklage, and B. Wiedermann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[6] L. Eeckhout, A. Georges, and K. De Bosschere. How java programs interact with virtual machines at the microarchitectural level. In R. Crocker and G. L. S. Jr., eds., *Proc. of the 18th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'03)*.

[7] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous Java performance evaluation. In R. P. Gabriel and D. Bacon, eds., *Proc. of the 22nd Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'07)*.

[8] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in Java workloads. In Vlissides and Schmidt [13], pp. 270–287.

[9] S. Hazelhurst. Truth in advertising : reporting performance of computer programs, algorithms and the impact of architecture and systems environment. *South African Computer Journal*, 46:24–37, 2010.

[10] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: improving program locality. In Vlissides and Schmidt [13], pp. 69–80.

[11] D. Lea, D. F. Bacon, and D. Grove. Languages and performance engineering: method, instrumentation, and pedagogy. *Sigplan Notices*, 43:87–92, 2008.

[12] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*.

[13] J. M. Vlissides and D. C. Schmidt, eds. *Proc. of the 19th Ann. Conf. on OO Prog. Sys., Lang., & Appl. (OOPSLA'04)*, Vancouver, BC, Canada, Oct. 2004. ACM SIGPLAN Notices 39 (10).