A Rigorous Benchmarking and Performance Analysis Methodology for Python Workloads

Arthur Crapé Lieven Eeckhout
Ghent University, Belgium

***Abstract*—Computer architecture and computer systems research and development is heavily driven by benchmarking and performance analysis. It is thus of paramount importance that rigorous methodologies are used to draw correct conclusions and steer research and development in the right direction. While rigorous methodologies are widely used for native and managed programming language workloads, scripting language workloads are subject to ad-hoc methodologies which lead to incorrect and misleading conclusions. In particular, we find incorrect public statements regarding different virtual machines for Python, the most popular scripting language. The incorrect conclusion is a result of using the geometric mean speedup and not making a distinction between start-up and steady-state performance.**

In this paper, we propose a statistically rigorous benchmarking and performance analysis methodology for Python workloads, which makes a distinction between start-up and steady-state performance and which summarizes average performance across a set of benchmarks using the harmonic mean speedup. We find that a rigorous methodology makes a difference in practice. In particular, we find that the PyPy JIT compiler outperforms the CPython interpreter by 1.76× for steady-state while being 2% slower for start-up, which refutes the statement on the PyPy website that 'PyPy outperforms CPython by 4.4× on average' based on the geometric mean speedup and not making a distinction between start-up and steady-state. We use the proposed methodology to analyze Python workloads which yields several interesting findings regarding PyPy versus CPython performance, start-up versus steady-state performance, the impact of a workload's input size, and Python workload execution characteristics at the microarchitecture level.

I. Introduction

Benchmarking and performance analysis is at the foundation of computer architecture and computer systems research and development. A lack of rigorous methodologies may lead to incorrect and misleading conclusions which steer research and development in unfruitful directions. Benchmarking and performance analysis is a well understood problem for native programming language workloads (i.e., applications written in C or C++). In particular, architects know how to select representative regions of execution [19], they know how to deal with non-determinism [1], and they know how to compose a diverse and representative set of benchmarks [8]. Benchmarking managed language workloads (i.e., applications written in Java or C#) requires additional care because of the virtual machine which dynamically optimizes application code and automatically manages memory. State-of-the-art methodologies hence make a distinction between start-up and steady-state performance [3], [10].

The situation is quite different for scripting languages, such as Python, Javascript, PHP, Ruby, etc. Python is a particularly popular scripting language: it is the most popular scripting language and the third most popular programming language after C and Java, according to the TIOBE index as of July 2020.[1] Unfortunately, rigorous benchmarking and performance analysis methodologies are lacking for scripting language workloads, and different researchers and practitioners use different ad-hoc methodologies [14], [15], [21]–[23]: some seem to measure start-up performance while others seem to measure steady-state performance (without explicitly stating so), and most seem to lack statistical data analysis. In particular, the PyPy website[2] mentions that *'on average, the PyPy Just-in-Time (JIT) compiler is 4.4 times faster than the CPython interpreter'*. We find that this is an incorrect and misleading conclusion for at least two reasons.

First, this average performance number is based on the geometric mean speedup which misguides the overall conclusion. The geometric mean speedup implicitly assumes that the individual speedup numbers are log-normally distributed. We find that this is not the case using three statistical tests. In addition, the geometric mean speedup does not have a physical meaning. Instead, we argue that the harmonic mean should be used to compute the average speedup across a set of benchmarks. The physical meaning of the harmonic mean speedup is that it quantifies the average reduction in execution time. We find that PyPy is on par with CPython (in fact, PyPy is 3% slower than CPython) using the harmonic mean speedup, in contrast to what the geometric mean speedup suggests. In other words, using the correct mean speedup matters in practice and is critical to reach valid conclusions.

Second, the average performance number reported on the PyPy website does not make a distinction between start-up and steady-state performance. This is unfortunate and inappropriate because it does not acknowledge the fact that PyPy is a JIT compiler, i.e., PyPy dynamically identifies, compiles and optimizes frequently executed code. An inherent property of a JIT compiler is that it requires time to identify and optimize hot code, hence it is expected that performance will be modest initially before reaching higher levels of performance as more and more code gets optimized. It is therefore critical to make a distinction between start-up and steady-state performance when evaluating JIT compilers. Start-up accounts for library loading and JIT compilation overheads, whereas steady-state measures optimized code performance. We find that PyPy

[1] https://www.tiobe.com/tiobe-index/
[2] https://www.pypy.org/

outperforms CPython by 1.76× on average for steady-state while being on par (actually, 2% slower) for start-up.

In this paper, we propose a statistically rigorous benchmarking and performance analysis methodology for Python workloads. This methodology makes a distinction between start-up and steady-state performance. Start-up measures performance of a single benchmark execution across multiple invocations of the Python virtual machine (VM) assuming no libraries have been loaded by the system. Steady-state measures performance across multiple executions of the same benchmark within a single VM invocation so that library loading and compilation overheads are amortized. Average performance across benchmarks is computed using the harmonic mean speedup, rather than the geometric mean. (While we focus on Python in this paper, the contributions made in this paper are applicable to workloads written in other scripting languages as well as other programming languages.)

We subsequently use this methodology to analyze PyPy versus CPython performance, yielding various interesting findings. For example, we find that the PyPy website primarily focused on start-up performance, not highlighting PyPy's JIT compiler ability to optimize steady-state performance; PyPy's performance advantage over CPython improves with increasing input data set size; PyPy's performance advantage comes from a reduced dynamic instruction count which compensates for a lower IPC (useful instructions executed per cycle) compared to CPython; steady-state performance is substantially higher than start-up performance for PyPy compared to CPython and is due to reduced dynamic instruction count and improved IPC; Python performance is strongly inversely correlated with branch MPKI, more so than LLC MPKI — this suggests that branch behavior is a more critical contributor to Python performance than cache behavior.

In summary, we make the following contributions:

- We argue based on a statistical argument and physical meaning that the average speedup across a set of benchmarks should be computed using the harmonic mean and not the geometric mean, and we show that computing the correct mean matters in practice, i.e., using an inappropriate mean speedup may lead to incorrect and misleading conclusions.
- We propose a statistically rigorous performance evaluation methodology for measuring Python start-up and steady-state performance. Start-up performance includes library loading and JIT compilation overheads, whereas steady-state performance is largely determined by the application code and the JIT compiler's optimizations.
- We use this methodology to evaluate and analyze Python workload performance using the CPython interpreter and the PyPy JIT compiler. We provide a list of 10 findings regarding PyPy versus CPython performance, start-up versus steady-state performance, input size sensitivity, and Python workload execution characteristics at the microarchitectural level.

The remainder of this paper is organized as follows. We first provide background on the Python programming language and its implementations (Section II). We then elaborate on how to compute average speedup across a set of benchmarks in a meaningful way (Section III). We describe our proposed rigorous performance evaluation methodology for Python workloads measuring start-up and steady-state performance (Section IV). After detailing our experimental setup (Section V), we comprehensively analyze Python performance along a number of dimensions including JIT compilation versus interpretation, start-up versus steady-state, and microarchitecture-level characteristics (Section VI). Finally, we discuss related work (Section VII) and conclude (Section VIII).

II. Background: Python

Python is a high-level general-purpose scripting language designed for programmer productivity. Python is dynamically typed and uses garbage collection to automatically manage memory. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming, and offers an extensive standard library.

Python is interpreted by default. CPython is the open-source reference interpreter, developed and maintained by a global community of programmers.[3] PyPy is an alternative implementation to CPython and leverages Just-in-Time (JIT) compilation to dynamically identify, compile and optimize frequently executed code (so-called hot code). Interpreters are easier to write and develop than JIT compilers. On the flip side, interpreters typically run slower, especially for long-running applications for which the JIT compiler has time to optimize the code [2]. Beyond CPython and PyPy, there exists a wide variety of implementations for (restricted subsets of) Python, including Cython (compilation of Python to C and C++), Jython (use of Java class library from a Python program), Numba (LLVM-based Python compiler), etc.

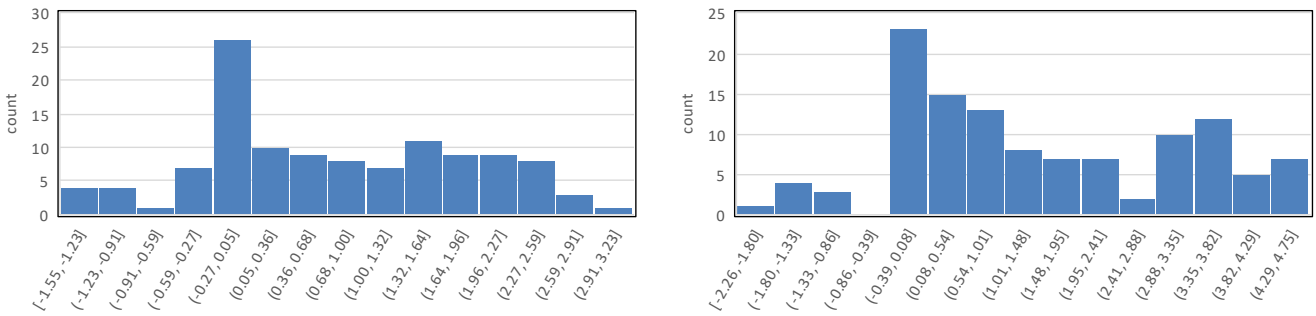
Python was conceived in the late 1980s, and several revisions were released including Python 2 in 2000 and Python 3 in 2008. There is no complete backward-compatible between Python 2 and 3. Python 2 was officially discontinued in 2020 of which Python 2.7 is the last Python 2 release. Since Python 2's end-of-life, only Python 3.5 and beyond are supported.

III. Computing Average Performance

Before describing how to rigorously and comprehensively benchmark and analyze Python performance, we first discuss how to compute average speedup numbers across a set of benchmarks in a meaningful way. How to compute average performance has been a topic of controversy and debate for several decades. Even today, performance engineers and computer architects do not seem to agree on how to summarize performance across a set of benchmarks.

The debate started in 1986 with Fleming and Wallace [9] arguing for the geometric mean. Smith [20] advocated the opposite, shortly thereafter. Cragon [5] also argues in favor of the arithmetic and harmonic mean. Hennessy and Patterson [12] describe the pros and cons of all three averages. More recently,

[3] https://www.python.org/



(a) Start-up.

(b) Steady-state.

Fig. 1: Histogram of the logarithm of speedups for PyPy versus CPython for start-up and steady-state performance. *Speedup does not follow a log-normal distribution.*

John [16] argued strongly against the geometric mean, which was counterattacked by Mashey [18]. Eeckhout [6] describes the underlying intuitions for the various means.

In this work, we argue against the geometric mean and in favor of the harmonic mean for computing the average speedup across a set of benchmarks. We do so based on two key arguments, a statistical argument and physical meaning, which we explain in the next two subsections.

A. Statistical Argument

Mashey [18] argues that speedup (the execution time on the reference system divided by the execution time on the enhanced system) is distributed following a log-normal distribution. A log-normal distribution means that the logarithm of the elements of a population are normally or Gaussian distributed. In contrast to a normal distribution, a log-normal distribution is skewed, i.e., its distribution is asymmetric and has a long tail to the right, whereas a normal distribution is symmetric and has zero skew. Normal distributions arise from aggregations of many additive effects (cf. central limit theory), whereas log-normal distributions arise from combinations of multiplicative effects. The assumption that speedup follows a log-normal distribution has some intuitive appeal: it is our common experience that some programs experience (much) larger speedups than others — in other words, there are outliers, hence the long tail to the right.

Assuming that speedup follows a log-normal distribution, the geometric mean is the appropriate mean for speedups:

$$GM = \exp\big(\frac{1}{n}\sum_{i=1}^{n}\ln(x_i)\big) = \sqrt[n]{\prod_{i=1}^{n} x_i},$$

with x_i the speedup for benchmark i. In this formula, it is assumed that x is log-normally distributed, or in other words, $\ln(x)$ is normally distributed. The average of a normal distribution is the arithmetic mean, hence, the exponential of the arithmetic mean over $\ln(x_i)$ computes the average for x. This equals the definition of the geometric mean, as shown on the right-hand side of the equation.

The geometric mean of speedup thus builds on the assumption that speedup is log-normally distributed. For the speedup values obtained in this work we verified whether speedup indeed follows a log-normal distribution. To do so, we performed three normality tests on the logarithm of the obtained speedup values: Shapiro-Wilk, D'Agostino and Anderson-Darling. The Shapiro-Wilk test tests the null hypothesis that a sample $x_1, \ldots, x_n$ comes from a normally distributed population. The test returns a p value, and if the p value is less than a chosen alpha level, say 0.05, the null hypothesis is rejected. D'Agostino's K^2 test is a normality test based on the sample's skewness and kurtosis. The Anderson-Darling normality test rejects the null hypothesis of normality if the test's statistic is larger than the normal distribution's critical value, and is generally considered a powerful test.

All three tests reject the null hypothesis of normality for all the speedups reported in this work. The tests also reject the null hypothesis when focusing on just the start-up and steady-state speedup numbers. This is further illustrated in Figure 1 in which we report the logarithm of the speedup values for PyPy versus CPython under start-up and steady-state. The histograms confirm that the logarithm of the speedups does not follow a normal distribution. We hence conclude that speedup does not follow a log-normal distribution. This statistical reasoning puts to question whether the geometric mean of speedups is appropriate.

B. Physical Meaning

In addition to the statistical argument, there is another compelling argument against the geometric mean of speedups and in favor of the harmonic mean, based on physical meaning. Table I provides an illustrative example. Consider two benchmarks *A* and *B* that run equally long on the original system, i.e., their normalized execution time equals one. (An alternative interpretation is to say that both benchmarks are equally important to the experimenter, i.e., this is a system's perspective on computer performance.) Assume now that the optimization of interest does not affect performance for benchmark *A*, i.e., execution time with the optimization

TABLE I: Example to illustrate the lack of physical meaning for the geometric mean of speedups.

Benchmark	Execution time		Speedup
	original	*optimized*	
A	1	1	1
B	1	0.01	100
Geometric mean speedup			10
Harmonic mean speedup			1.98

equals the execution time of the original system. In contrast, the optimization improves performance by a factor $100\times$ for benchmark *B*, i.e., the execution time is reduced from one unit of time to 0.01 units of time. In other words, executing benchmarks *A* and *B* on the original system takes 2 time units while taking 1.01 time units on the optimized system.

The geometric mean speedup across the two benchmarks equals $10\times$. The intuitive understanding based on the geometric mean is thus that the optimization improves performance by $10\times$, which does not have any physical meaning, unfortunately. In contrast, the harmonic mean speedup does have a physical meaning, namely the execution time of benchmarks *A* and *B* is reduced by a factor $1.98\times$. This is exactly what is expected. Executing *A* and *B* takes 2 time units on the original system and 1.01 time units on the optimized system, which results in a speedup of $2/1.01 = 1.98\times$. This equals the harmonic mean speedup as computed over the individual benchmarks' speedup values. In other words, the physical meaning of the harmonic mean speedup S is that the execution time of a set of benchmarks is reduced by a factor $S\times$.

C. Does it Matter in Practice?

One may wonder whether computing the harmonic versus geometric mean speedup makes a difference in practice? Or, is this just a matter of academic interest?

We ran all the PyPerformance benchmarks out-of-the-box on the PyPy JIT compiler version 7.3.1 and the CPython interpreter version 3.6.9 on an Intel Core i7 system (experimental details follow in Section V). Computing the geometric mean speedup leads us to conclude that PyPy is $4.4\times$ faster than CPython. (This speedup number is exactly the same as the speedup number reported on the PyPy website.) However, computing the harmonic mean speedup leads us to conclude that PyPy yields a speedup of $0.97\times$ compared to CPython; or, in other words, PyPy is 3% slower than CPython! So the question now is: which mean speedup is correct? Is PyPy indeed much faster than CPython according to the geometric mean speedup? Or, is PyPy on par with CPython (and slightly slower) according to the harmonic mean speedup?

This result clearly illustrates that using the appropriate mean really makes a difference in practice, and using an inappropriate mean may lead to incorrect or misleading conclusions. Based on the statistical argument and the physical meaning as discussed above, we argue that the harmonic mean speedup leads to the correct answer. To further and more deeply analyze Python performance, we now describe how to evaluate startup and steady-state performance in a statistically rigorous manner.

IV. Measuring Start-Up and Steady-State Performance

The methodology we propose and use to evaluate Python workloads in this work is based on the statistically rigorous Java performance evaluation methodology by Georges et al. [10]. This methodology makes a distinction between start-up and steady-state performance. Before describing the methodology in detail, we first introduce some terminology and notation.

A. Terminology and Notation

We refer to a *VM invocation* as the initiation of the Python virtual machine (VM). The VM can be an interpreter (e.g., CPython) or a JIT compiler (e.g., PyPy). Within a single VM invocation, we can execute the Pyton benchmark multiple times. Each execution within a single VM invocation is called a *benchmark iteration*. If a benchmark is iterated multiple times within a single VM invocation, the first iteration of the benchmark will perform most of the library loading and JIT (re)compilation, and subsequent iterations will experience less (re)compilations. In other words, the first (few) iteration(s) will incur library loading and JIT compilation overheads whereas subsequent iterations will be running compiled and optimized code. Researchers and developers mostly interested in steady-state performance should therefore focus on the subsequent iterations of the benchmark, and not the first iteration. This might be of interest for example to evaluate a VM's intrinsic compilation and optimization performance. Researchers interested in start-up performance will typically run a benchmark only once to capture the first iteration. This might be of interest to evaluate a system's responsiveness and interactive performance for short-running Python workloads. In the following discussion, we will refer to x_{ij} as the measurement of the j-th benchmark iteration of the i-th VM invocation.

B. Start-Up Performance

Start-up performance quantifies how quickly a Python virtual machine can execute a relatively short-running Python program. As mentioned above, start-up performance is affected by library loading and JIT compilation, substantially more so than steady-state performance.

For measuring start-up performance, we follow a two-step methodology:

1) We measure the execution time of multiple VM invocations, each VM invocation running a single benchmark iteration. This results in p measurements x_{ij} with $1 \leq i \leq p$ and $j = 1$.
2) We dynamically compute the confidence interval at the 95% confidence level over the past i measurements. If the confidence interval is within 5% of the computed mean, or we have run more than $p = 30$ VM invocations, we stop the measurements. We compute the confidence interval using the Student t-statistic [17].

This procedure makes the implicit assumption that the different measurements (VM invocations) are independent.

This may not be true in practice, because the first VM invocation may change system state which persists past the first VM invocation, e.g., dynamically loading libraries in main memory. To reach independence across the different VM invocations, we clear the pagecache using the Linux operating system `drop_caches` utility. This is to unload the libraries from the system. In other words, start-up performance in this work includes the overhead of library loading. If a performance analyst or engineer would like to exclude the impact of library loading (because the system under test is assumed to have the libraries already dynamically loaded), one can decide to not clear the pagecache when measuring start-up performance.

C. Steady-State Performance

Steady-state performance concerns long-running applications for which start-up is less relevant (or even irrelevant), i.e., the application's total running time (by far) exceeds start-up time. Since most of the library loading and JIT compilation/optimization is performed during start-up, steady-state performance is largely determined by the application code and the JIT compiler's optimization abilities.

The approach we take for steady-state performance is to execute or iterate a benchmark multiple times within a single VM invocation. Executing a benchmark multiple times without restarting the VM amortizes library loading time and enables the JIT compiler to further optimize the code. The question then is how many benchmark iterations to consider before we reach steady-state performance within a single VM invocation? The answer will likely be different for different benchmarks, hence we need a mechanism to determine when steady-state performance is reached within a single VM invocation.

We therefore use a three-step methodology for steady-state performance:

1) We consider one VM invocation within which we run the benchmark at most $q = 30$ times.
2) We determine the iteration s where steady-state performance is reached, i.e., once the coefficient of variation (CoV), or the standard deviation divided by the mean, of the last k iterations ($s - k + 1$ to s) is less than 2%. We run at most $q = 30$ iterations per VM invocation and we set $k = 4$.
3) We compute the mean $\bar{x}_i$ of the s benchmark iterations under steady-state ($i = 1$):

$$\bar{x} = \sum_{j=1}^{s} x_{ij}.$$

By re-executing the same benchmark within the same VM invocation, this steady-state procedure thus computes the average performance observed over a long period of time, providing a solid picture of steady-state performance. Note that the methodology by Georges et al. [10] considers multiple VM invocations across which average steady-state performance is computed. We consider a single VM invocation in this work for two reasons: (i) to limit experimentation time, and (ii) we find negligible variability across VM invocations for steady-state performance. (This is further affirmed by the limited variability observed for start-up performance across VM invocations, as we will discuss in the next section.)

TABLE II: Benchmarks and inputs considered in this work.

Benchmark	Input	Description
nqueens	10	N-queens solver
go	300 500 700	AI-driven board game
crypto_pyaes	default	AES block-cipher
richards	default	OS kernel simulation
mdp	default	graph node sorting
nbody	default	n-body solver
fannkuch	10 11	permutation game
pidigits	5,000 10,000 20,000 30,000	calculating π digits
pyflate	default	bzip2 decompression
raytrace	300 400 500 600 700	raytracing square image
spectral_norm	1,000 1,500 2,000 5,000	eigenvalue computation
regex_v8	default	regular expression on Web page

V. Experimental Setup

We now use the above statistically rigorous data analysis methodology to benchmark Python performance. The performance numbers reported in this work are obtained on an HP Envy notebook which features an Intel Core i7-6500U CPU running at 2.5 GHz with 12 GB of DRAM running at 1.6 GHz. The machine runs the Ubuntu 16.04.6 LTS (Xenial Xerus) operating system. We use Linux' `perf` utility to measure time and hardware performance counters.

We consider two Python runtimes, namely the CPython interpreter version 3.6.9 and the PyPy Just-in-Time (JIT) compiler version 7.3.1. We ran all the PyPerformance benchmarks[4] out-of-the-box in Section III. Many of these benchmarks are (very) short-running though: 48% and 81% of the benchmarks run for less than 0.1 seconds for CPython and PyPy, respectively, and all but two run for less than one second for both CPython and PyPy. Because evaluating start-up and steady-state performance for a JIT compiler requires sufficiently long execution times, we consider a subset of the PyPerformance benchmarks in the remainder of this paper. We select benchmarks based on the following two criteria: (i) conversion from Python 2 to Python 3 — converting benchmarks from Python 2 to Python 3 was straightforward for many benchmarks but not all, hence we excluded the benchmarks that were non-trivial to port for further consideration; and (ii) execution time — we withheld the benchmarks that run for at least

[4] `https://pyperformance.readthedocs.io/`

0.5 seconds and/or for which we could prolong the execution time to be more than 0.5 seconds by changing the input. Table II lists the benchmarks considered in the remainder of this paper. We consider multiple inputs for several benchmarks to evaluate input sensitivity. All benchmark-input pairs run for at least 0.5 seconds using CPython under start-up, and half the benchmarks run for more than ten seconds, and up to almost 6 minutes.

Recall that our statistical data analysis methodology uses specific parameters. In particular, for start-up performance, we take as many measurements as needed (with a cap of 30 measurements) so that the confidence interval is within 5% of the computed mean. For steady-state performance, we consider the last $k = 4$ measurements out of 30 so that the coefficient of variation is less than 2%. These specific methodological parameters were selected based on prior work by Georges et al. [10]. We verify that for our measurements, the confidence interval is within 2.6% on average (and at most 4.8%) of the mean for start-up, and the average coefficient of variation is around 0.4% on average (and at most 1.9%) for steady-state. We do not report the confidence intervals in the result graphs because they are hardly visible, while complicating the reading of the graphs.

Although the proposed analysis methodology requires multiple runs of the same benchmark to obtain stable performance results, this does not lead to an impractical experimental methodology. Recall that we run up to 30 VM invocations for start-up, and up to 30 benchmark iterations within a single VM invocation for steady-state. Of course, the fact that multiple runs are required prolongs experimentation time. Fortunately, we find that the number of required measurements is limited in practice. For start-up, we find that we need at most four VM invocations for most benchmarks for both CPython and PyPy. We note only two outliers requiring 6 VM invocations (go for PyPy) and 12 invocations (crypto for CPython). For steady-state, we note that there is limited performance variability for CPython and we hence need only the minimum number of four benchmark iterations. There is more performance variability across benchmark iterations for PyPy — not unexpectedly because of the JIT compiler — however, the number of iterations is still limited to 5.4 on average and at most 8 for richards. In other words, the overall conclusion is that this performance analysis methodology does not increase experimentation time dramatically. On average, the methodology increases experimentation time by around 5× compared to running a single instance per benchmark. While this is a non-negligible increase, it is manageable (especially on real hardware) while guaranteeing a solid performance analysis.

VI. Performance Analysis

We now perform a number of performance analyses, including evaluating PyPy versus CPython, input sensitivity, start-up versus steady-state performance, and microarchitectural analysis.

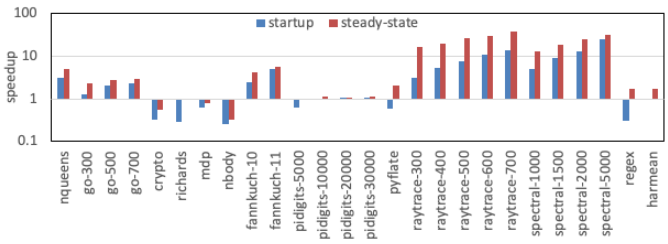


Fig. 2: Speedup PyPy versus CPython. *PyPy leads to a speedup of 1.76× and 0.98× compared to CPython for steady-state and start-up performance, respectively.*

A. PyPy versus CPython

We first compare the PyPy JIT compiler against the CPython interpreter, while making a distinction between start-up and steady-state performance, see Figure 2. PyPy leads to a harmonic mean speedup of 1.76× compared to CPython for steady-state performance, and a harmonic mean speedup of 0.98× for start-up performance. In other words, PyPy beats CPython for steady-state performance while being 2% slower (almost on par) for start-up performance. More specifically, PyPy leads to significant speedups for start-up for many benchmarks (up to 25.3× for spectral-5000) while severely degrading performance for others: 0.25× speedup (or, 4× slowdown) for nbody and around 0.3× speedup (or, 3.3× slowdown) for crypto, richards and regex. PyPy leads to significant speedups for steady-state and up to 36.8× for raytrace-700; performance degrades compared to CPython for nbody (3.1× slowdown) and crypto (1.8× slowdown).

Finding #1: PyPerformance primarily measures start-up performance. It is interesting to compare the above performance numbers against the performance numbers reported on the PyPerformance website, as described in Section III-C. Running the PyPerformance benchmarks out-of-the-box leads to a harmonic mean speedup for PyPy against CPython of 0.97×. This number is obtained by running each benchmark for a single iteration within a single VM invocation. And while we are considering a subset of the benchmarks here (with different inputs in some cases), the harmonic mean speedup of 0.97× is remarkably close to the harmonic mean speedup of 0.98× under start-up conditions. This suggests that PyPerformance primarily measures start-up performance and not steady-state performance. The fact that most PyPerformance benchmarks are (very) short-running, as discussed before, provides further evidence for this finding.

Finding #2: PyPy outperforms CPython for steady-state performance. The above performance analysis further reveals that the PyPy JIT compiler significantly outperforms the CPython interpreter for steady-state performance (i.e., harmonic mean speedup of 1.76×). This is not unexpected since the PyPy JIT compiler needs time to identify frequently executed code for compilation and optimization.

Finding #3: PyPy is on par with CPython for start-up performance. We find that PyPy yields similar start-

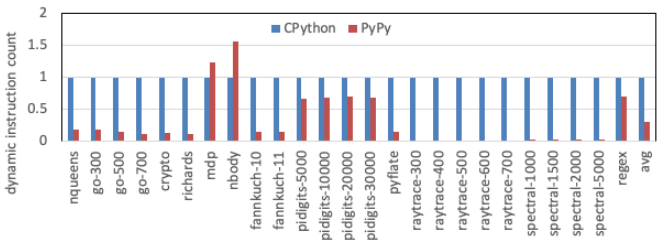


Fig. 3: Dynamic instruction count under steady-state. *The dynamic instruction count is on average 3.2× smaller for PyPy than for CPython.*

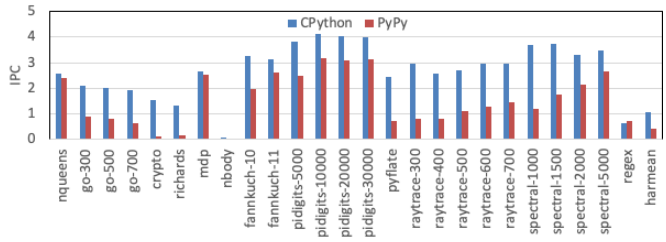


Fig. 4: IPC under steady-state. *The IPC is on average 2.5× lower for PyPy than for CPython.*

up performance to CPython on average, and in fact leads to slightly inferior performance (2% slower). The reason is that the JIT compiler is unable to speed up the frequently executed code and/or the compilation/optimization overhead does not get amortized by the improved code quality.

Finding #4: Different benchmarks favor different VMs. It is interesting to note that different applications favor different VMs. In particular, raytrace, spectral, fannkuch, nqueens and go favor PyPy for both start-up and steady-state. In contrast, crypto, richards, mdp and nbody favor CPython for both start-up and steady-state. For pidigits and regex, we note that CPython yields highest performance for start-up while PyPy yields highest performance for steady-state. This suggests that the JIT compilation cost gets amortized as the benchmark executes long enough. This observation also implies that system performance can be improved by selecting the best performing VM on a per-application basis, e.g., use PyPy for raytrace and spectral, versus use CPython for crypto and nbody. Further, selecting a particular VM for different scenarios may also improve performance, e.g., for pyflate and regex, using CPython for start-up and PyPy for steady-state yields the highest performance.

B. Analyzing JIT versus Interpreter Performance

We now dive deeper to understand where the performance difference between PyPy and CPython is coming from. To do so, we analyze dynamic instruction count and IPC (useful instructions executed per cycle). Recall that dynamic instruction count N and IPC determine overall performance, assuming that clock frequency f is constant, cf. Iron Law of Performance: execution time $T = N \times 1/IPC \times 1/f$. Figures 3 and 4 report dynamic instruction count and IPC, respectively, under steady-state.

Finding #5: The reduction in dynamic instruction count compensates for the decrease in IPC for PyPy versus CPython under steady-state. The dynamic instruction count is on average 3.2× smaller for PyPy compared to CPython, see Figure 3. We note a dramatic reduction in dynamic instruction count for most benchmarks using PyPy, and up to 75.2× (raytrace). This is due to the JIT compiler which optimizes the application code and eliminates the interpreter loop from the dynamic instruction stream. For only two benchmarks (mdp and nbody) do we note an increase in dynamic instruction count. This is likely due to the overhead introduced by the JIT compiler.

In contrast, we note that PyPy leads to lower IPC compared to CPython, by 2.5× on average, see Figure 4. We note a reduction in IPC for all but one benchmark. The reduction is as high as 14.4× for crypto from 1.54 for CPython to 0.11 for PyPy. The overall conclusion is that the reduction in dynamic instruction count (by 3.2×) amortizes the decrease in IPC (2.5×) which leads to a net performance improvement for PyPy compared to CPython for steady-state (by 1.76×).

C. Input Sensitivity

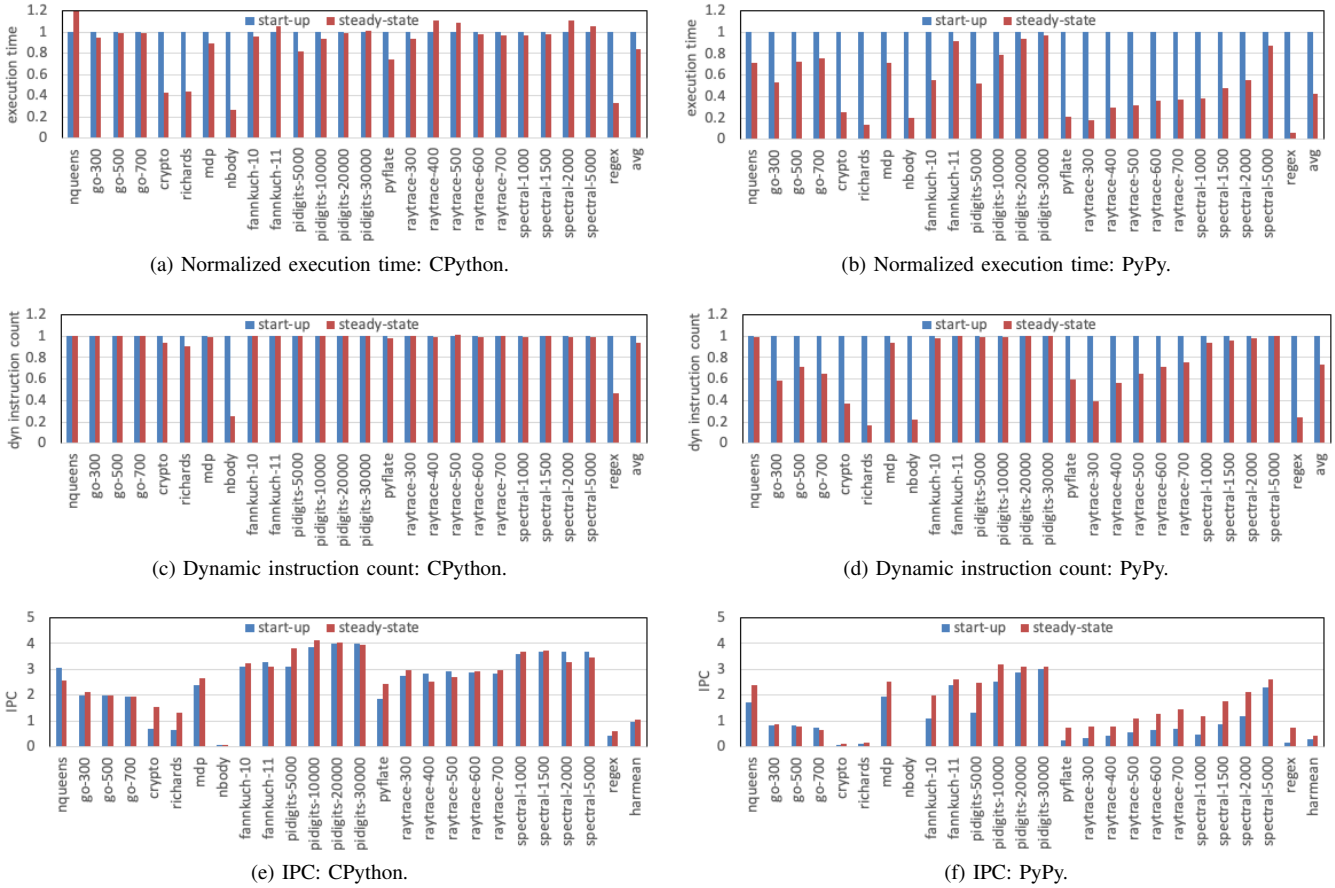
It is interesting to analyze PyPy versus CPython performance as a function of input size. The longer the execution takes, the higher the opportunity for the JIT compiler to optimize the code.

Finding #6: PyPy speedup over CPython increases with increasing input size. Figure 2 allows us to analyze PyPy versus CPython performance as a function of input size. (We consider different inputs for five benchmarks: go, fannkuch, pidigits, raytrace and spectral.) We find that PyPy indeed leads to higher speedups compared to CPython for increasingly large inputs, for both start-up and steady-state. For example, the speedup increases from 16.4× (smallest input) to 36.8× (largest input) for raytrace under steady-state. Note that for pidigits under start-up, PyPy leads to a 1.61× performance slowdown compared to CPython for the smallest input, while yielding a 1.09× speedup for the largest input.

D. Start-Up versus Steady-State Performance

We now compare steady-state performance against start-up performance for both PyPy and CPython, see Figure 5 which shows normalized execution time in the top row, normalized dynamic instruction count in the middle row and IPC in the bottom row; CPython results are reported in the left column and PyPy results are shown in the right column.

Finding #7: Steady-state performance is substantially higher than start-up performance for PyPy compared to CPython. The key observation from this analysis is that the gap between start-up and steady-state performance is substantially higher for PyPy compared to CPython. We report a harmonic mean speedup of 1.95× when comparing steady-state performance against start-up performance for PyPy, versus a harmonic mean speedup of 1.12× for CPython. The PyPy



(a) Normalized execution time: CPython.

(b) Normalized execution time: PyPy.

(c) Dynamic instruction count: CPython.

(d) Dynamic instruction count: PyPy.

(e) IPC: CPython.

(f) IPC: PyPy.

Fig. 5: Start-up versus steady-state performance for CPython and PyPy: execution time (top row), dynamic instruction count (middle row) and IPC (bottom row) for one iteration under start-up and steady-state. Execution time and dynamic instruction count results are normalized to start-up. *Steady-state performance is substantially higher than start-up performance for PyPy (1.95× harmonic mean speedup) compared to CPython (1.12× harmonic mean speedup).*

JIT compiler dynamically optimizes hot code for long-running applications which leads to a performance boost compared to start-up performance. This is not the case for the CPython interpreter, or at least, to a lesser extent. Note that CPython also yields higher performance for steady-state compared to start-up for some benchmarks. This seems to suggest that CPython also performs some form of dynamic code optimization in the interpreter loop. Clearly, the performance boost is not comparable to PyPy's JIT compiler.

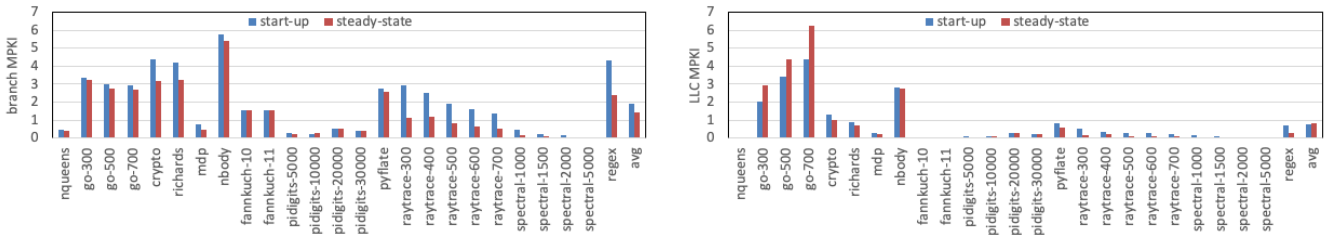
Finding #8: Performance improvement during steady-state is a result of decreasing dynamic instruction count and increasing IPC. There is a multiplicative effect on overall steady-state performance: dynamic instruction count decreases *and* IPC increases. This is especially the case for PyPy: dynamic instruction count decreases by on average 1.36× and IPC increases by on average 1.46×. This leads to a cumulative performance improvement of 1.95×.

E. Microarchitectural Analysis

We now explore the reason for the substantial increase in IPC for PyPy under steady-state compared to start-up. We therefore analyze branch MPKI (misses per thousand instructions) and LLC MPKI, see Figure 6.

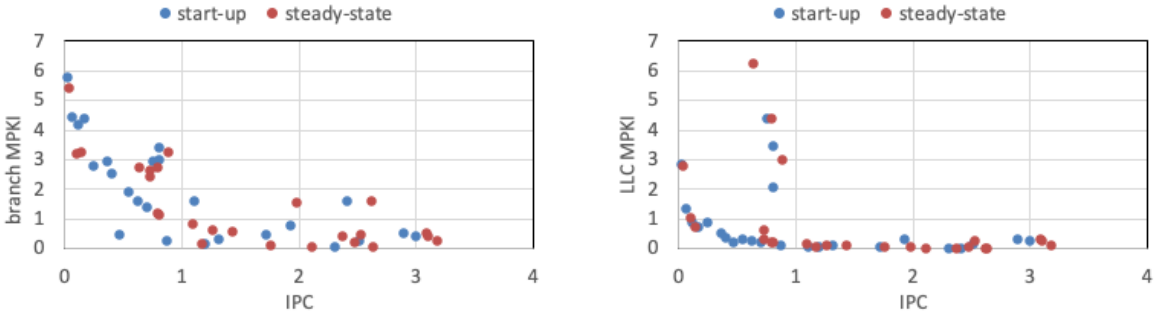
Finding #9: PyPy's JIT compiler decreases branch MPKI as well as LLC MPKI. We observe that branch MPKI is substantially lower for steady-state compared to start-up for all benchmarks. In other words, PyPy's JIT compiler is able to significantly improve the branch behavior of the hot code. On average, the JIT compiler reduces the branch MPKI by a factor 1.33× and up to 3× for spectral and 2.6× for raytrace. We observe a somewhat similar trend for LLC MPKI: we observe a reduction in LLC MPKI for all but one benchmark, namely go for which LLC MPKI increases 1.45×.

Finding #10: Python performance strongly correlates with branch prediction behavior. It is interesting to correlate branch MPKI against IPC, see Figure 7(a). We note an inverse correlation between branch MPKI and IPC. There is a clear trend: all benchmarks with a MPKI above 2 have an IPC below 1, and all benchmarks with an IPC above 1 have a branch MPKI below 2. In particular, we note the highest branch MPKI (5.4) and lowest IPC (0.04) for nbody. Likewise, the



(a) Branch MPKI. (b) LLC MPKI.

Fig. 6: Branch and LLC MPKI for PyPy start-up and steady-state performance. *PyPy's JIT compiler decreases branch MPKI for all benchmarks while reducing LLC MPKI for all but one benchmark.*



(a) Branch MPKI versus IPC. (b) LLC MPKI versus IPC.

Fig. 7: Correlating branch and LLC MPKI against IPC. *IPC inversely correlates with branch MPKI; the correlation is less pronounced between LLC MPKI and IPC.*

branch MPKI is high (3.2) and IPC is low (0.11 and 0.15) for crypto and richards, respectively. This suggests that the branch predictor has a substantial impact on performance. This is further affirmed by noting that the branch MPKI for these Python benchmarks is fairly high in absolute numbers.

The (inverse) correlation between LLC MPKI and IPC is less pronounced, see Figure 7(b), but still we note that benchmarks with a high LLC MPKI, such as go and nbody have a fairly low IPC (well) below 1. However, there are many benchmarks with an IPC below one with an LLC MPKI below one, which suggests that other processor components, such as the branch predictor, are responsible for the low IPC.

VII. Related Work

Python performance analysis. Ismail and Suh [15] perform a detailed simulation-based overhead analysis of the Python programming language compared to a native language such as C. They categorize the overhead contributors in terms of language features (e.g., name resolution, function setup and cleanup, error checking, garbage collection, boxing/unboxing) versus interpreter operations (e.g., C function calls, dispatching bytecode instructions, object allocation, stack operations). They conclude that name resolution and function setup and cleanup are the language features which incur the highest overhead; C function calls and dispatching byte code instructions are the interpreter features incurring the highest overhead. In addition, they explore the interaction with microarchitecture and heap size by providing various sensitivity analyses. In their experimental methodology, they 'warm up each benchmark by running it twice followed by running it three times for evaluation'. This seems to suggest that what they characterize is steady-state performance, more so than start-up performance. However, this ad-hoc methodology does not provide a statistically rigorous assessment whether it characterizes start-up or steady-state performance. The Python benchmarking methodology proposed in this paper is complementary to this prior work, providing a comprehensive and statistically rigorous approach for benchmarking both start-up versus steady-state performance. It would be interesting to see how the language feature and interpreter operation overheads differ between start-up versus steady-state.

Ilbeyi et al. [14] propose a cross-layer performance evaluation methodology to analyze meta-tracing JIT performance at the application, framework, interpreter, JIT compiler and microarchitecture levels. Meta-tracing JIT frameworks decouple the language definition from the VM internals to reduce the effort for developing custom JIT compilers. This prior work considers the RPython meta-tracing JIT, and compares CPython versus PyPy while not explicitly stating whether it focuses on start-up or steady-state performance. In fact, it seems that the standard PyPerformance benchmarking approach is adopted, which according to our findings focuses on start-up performance. Our work is complementary to this work and it would be interesting to analyze how the cross-layer

characteristics differ between start-up and steady-state.

JavaScript benchmarking. JavaScript is another popular scripting language, specifically targeting interactive web pages. Several studies have analyzed JavaScript performance. Tiwari and Solihin [22] analyze the performance overheads of JavaScript compared to C/C++. Southern and Renau [21] characterize the overheads of deoptimization in the V8 JavaScript engine. They run each 'benchmark configuration' multiple times which suggests they evaluate start-up performance (i.e., multiple VM invocations and one benchmark execution per invocation, per our terminology); they report geomean mean performance numbers across a set of benchmarks. Zhu et al. [23] propose and characterize server-side JavaScript benchmarks using the Node.js framework. They focus on steady-state performance, which is arguably the typical state of operation for server-side applications. We believe that the contributions made in this paper can help solidify benchmarking methodologies for scripting languages other than Python, such as JavaScript and beyond.

Java performance evaluation. There exists an extensive body of work on Java performance evaluation, upon which this paper builds. Eeckhout et al. [7] point out that Java performance characteristics can be vastly different depending on the application's input size due to the JIT compiler's ability to optimize long-running workloads. The DaCapo project offered a benchmark suite and proposed a solid experimental design methodology for benchmarking Java workloads [4]. Georges et al. [10] provided a statistically rigorous data analysis methodology making a distinction between start-up and steady-state performance. Replay compilation was introduced to evaluate a JIT compiler's inherent performance characteristics under steady-state [11], [13]. This paper borrows several concepts in experimental design and data analysis from this body of prior work, including the focus on start-up versus steady-state performance and the use of statistical data analysis. The conclusion that the harmonic mean speedup is a meaningful way to summarize average performance is a novel contribution of this paper, as well as the characterization and analysis of Python workloads.

VIII. Conclusion

Benchmarking methodology plays a critical role in computer architecture and computer systems research and development by creating a common ground for evaluating ideas and products. Sound methodology relies on sound experimental design and rigorous data analysis. Unsound methodology may retard or misdirect innovation; a sound methodology accelerates and steers research and innovation in the right direction.

While rigorous methodologies are now widely understood and widely used for native and managed programming language workloads, we find that sound methodologies for workloads written in scripting languages are lacking. In particular, we find that for Python, the most popular scripting language, current methodologies are ad-hoc, which leads to incomplete, misleading or outright incorrect conclusions. The use of geometric mean speedup provides an incorrect performance assessment of existing Python implementations (PyPy versus CPython). Not making a distinction between start-up and steady-state performance draws an incomplete or misleading performance picture.

This paper proposed a statistically rigorous benchmarking and performance analysis methodology for Python workloads, which makes a distinction between start-up and steady-state performance and which uses the harmonic mean speedup to summarize average performance across a set of workloads. Using this methodology we comprehensively evaluate PyPy versus CPython performance, start-up versus steady-state performance, the impact of a workload's input size, and Python workload execution characteristics at the microarchitecture level.

Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was supported in part by European Research Council (ERC) Advanced Grant agreement no. 741097, and FWO projects G.0434.16N and G.0144.17N.

References

[1] A. Alameldeen and D. Wood, "Variability in architectural simulations of multi-threaded workloads," in *Proceedings of the Ninth International Symposium on High-Performance Computer Architecture (HPCA)*, Feb. 2003, pp. 7–18.

[2] M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney, "A survey of adaptive optimization in virtual machines," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 449–466, Feb. 2005.

[3] S. 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*, vol. 51, no. 8, pp. 83–89, Aug. 2008.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2006, pp. 169–190.

[5] H. G. Cragon, *Computer Architecture and Implementation*. Cambridge University Press, 2000.

[6] L. Eeckhout, *Computer Architecture Performance Evaluation Methods*, ser. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2010.

[7] L. Eeckhout, A. Georges, and K. De Bosschere, "How Java programs interact with virtual machines at the microarchitectural level," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, Oct. 2003, pp. 169–186.

[8] L. Eeckhout, H. Vandierendonck, and K. De Bosschere, "Workload design: Selecting representative program-input pairs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2002, pp. 83–94.

[9] P. J. Fleming and J. J. Wallace, "How not to lie with statistics: The correct way to summarize benchmark results," *Communications of the ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986.

[10] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," in *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, Oct. 2007, pp. 57–76.

[11] A. Georges, L. Eeckhout, and D. Buytaert, "Java performance evaluation through rigorous replay compilation," in *Proceedings of the Annual ACM SIGPLAN Conference on Object-Oriented Programming, Languages, Applications and Systems (OOPSLA)*, Oct. 2008, pp. 376–384.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003.

[13] 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 *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA*, 2004, pp. 69–80.

[14] B. Ilbeyi, C. F. Bolz-Tereick, and C. Batten, "Cross-layer workload characterization of meta-tracing JIT VMs," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Oct. 2017, pp. 97–107.

[15] M. Ismail and G. E. Suh, "Quantitative overhead analysis for Python," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2018, pp. 36–47.

[16] L. K. John, "More on Finding a Single Number to Indicate Overall Performance of a Benchmark Suite," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 4, pp. 1–14, Sep. 2004.

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

[18] J. R. Mashey, "War of the benchmark means: Time for a truce," *ACM SIGARCH Computer Architecture News*, vol. 32, no. 4, pp. 1–14, Sep. 2004.

[19] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2002, pp. 45–57.

[20] J. E. Smith, "Characterizing computer performance with a single number," *Communications of the ACM*, vol. 31, no. 10, pp. 1202–1206, Oct. 1988.

[21] G. Southern and J. Renau, "Overhead of deoptimization checks in the V8 JavaScript engine," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016, pp. 75–84.

[22] D. Tiwari and Y. Solihin, "Architectural characterization and similarity analysis of Sunspider and Google's V8 Javascript benchmarks," in *Proceedings of the IEEE International Symposium on Performance Analysis (ISPASS)*, Apr. 2012, pp. 221–232.

[23] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural implications of event-driven server-side Web applications," in *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec. 2015, pp. 762–774.