Cross-Language Compiler Benchmarking

Are We Fast Yet?

Stefan Marr Benoit Daloze Hanspeter Mössenböck

Johannes Kepler University Linz, Austria

{stefan.marr, benoit.daloze, hanspeter.moessenboeck}@jku.at

Abstract

Comparing the performance of programming languages is difficult because they differ in many aspects including preferred programming abstractions, available frameworks, and their runtime systems. Nonetheless, the question about relative performance comes up repeatedly in the research community, industry, and wider audience of enthusiasts.

This paper presents 14 benchmarks and a novel methodology to assess the compiler effectiveness across language implementations. Using a set of common language abstractions, the benchmarks are implemented in Java, JavaScript, Ruby, Crystal, Newspeak, and Smalltalk. We show that the benchmarks exhibit a wide range of characteristics using language-agnostic metrics. Using four different languages on top of the same compiler, we show that the benchmarks perform similarly and therefore allow for a comparison of compiler effectiveness across languages. Based on anecdotes, we argue that these benchmarks help language implementers to identify performance bugs and optimization potential by comparing to other language implementations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Benchmarking, Languages, Virtual Machines

1. Introduction

Programming languages vary widely in design and programming models. Frameworks and community-specific philosophies cause further variations in how they are used. Despite these differences, when implementing languages, one wants to compare the performance to existing systems. Or to put it less formally: we'd like to know whether we're fast yet!

Copyright © 2016 ACM 978-1-4503-4445-6/16/10…$15.00
DOI: http://dx.doi.org/10.1145/10.1145/2989225.2989232

With the wide variety of languages, there is no trivial answer. It is also unclear whether any single answer is useful outside of a specific context. Nonetheless, the question is asked again and again, which leads us to search for a way to answer one specific question: *is one implementation of a widely used* set of core language abstractions *as fast as other implementations of the same set of abstractions?*

We conjecture that a commonly supported set of core language abstractions (incl. objects, polymorphic methods, closures, and arrays) is relevant for general application performance. Hence, it is necessary to optimize these abstractions to reach optimal performance. However, the set of abstractions is not universal. For example, programs in Haskell use language abstractions that are different from those in imperative languages. Another aspect is that optimizing only a set of core abstractions is not sufficient for optimal performance of all uses of a language. Programs typically use very expressive but language-specific abstractions, which need to be optimized as well. Nonetheless, a common set of abstractions likely minimizes the performance impact of different language semantics and enables a comparison.

This paper assesses whether we can use benchmarks written using only these core language abstractions for cross-language comparisons. We compare the performance of four object-oriented languages built on the same platform, namely the HotSpot JVM with the Graal just-in-time (JIT) compiler [16] and the Truffle language implementation framework [18]. We find that these implementations exhibit very similar performance for our set of 14 benchmarks. In addition, we compare 6 other language implementations to confirm that the results reflect the expected performance differences. Together with anecdotes about found performance bugs, the results indicate that our approach is adequate to determine the *compiler effectiveness*, i.e., the degree by which the core language abstractions are optimized by the compiler and the runtime system. The contributions of this work are:

- a novel methodology to compare the performance of different language implementations for a common set of core language abstractions
- a freely available set of benchmarks
- a characterization of the benchmarks and the aspects they measure based on language-independent metrics

- a performance evaluation of 10 implementations of 6 languages showing a wide range of optimization levels

2. Background

When evaluating programming language implementations, benchmarks can be used to compare a wide range of characteristics including computational performance and resource footprint. This paper focuses on computational performance in order to guide implementation and optimization.

Benchmark suites are typically built around specific languages or platforms. For the Java virtual machine (JVM), there exist commercial suites such as SPECjvm2008[1] and academic projects such as DaCapo [1] and DaCapo con Scala [13]. JavaScript has a set of competing projects with different goals including JetStream,[2] Octane,[3] and Kraken.[4] However, when it comes to comparing the performance of different programming languages, only the Computer Language Benchmarks Game[5] gained wider attention.

The remainder of this section discusses these suites, examines the current state of cross-language benchmarking, and identifies common metrics to characterize benchmarks.

2.1 Benchmark Suite Design

SPECjvm2008 SPECjvm2008 is designed to measure the performance of the JVM and its libraries, focusing on processor and memory usage. The benchmarks include text, numerical, and media processing. Shiv et al. [14] characterized the benchmarks based on system-level metrics. While these metrics are conceptually language-independent, they do not provide a higher-level insight into the benchmarks to get an intuition of language-level behavior.

DaCapo DaCapo [1] provides Java benchmarks and an experimental methodology to account for the JVM's dynamic compilation and garbage collection. Its goal is to enable research based on an open and easy-to-use suite that represents complex applications from a wide range of domains.

The benchmarks are characterized based on static code size and complexity metrics. Additionally, they determine the number of bytecodes in methods executed at least once, as well as instruction cache misses on the processor level during execution. To capture characteristics relevant for garbage collectors (GC), they also report metrics on object lifetime and demographics as well as heap composition. Furthermore, *principal component analysis* (PCA) [5] is used to assess the difference between benchmarks and argue that they cover a wide variety of dynamic behaviors.

DaCapo con Scala The work of Sewe et al. [13] uses a similar approach for a suite of application benchmarks for Scala, which also runs on the JVM. *DaCapo con Scala* aims at helping JVM engineers and researchers to optimize JVMs for languages besides Java.

Compared to DaCapo, here the focus is more on the code than on the memory behavior. Furthermore, they focus on architecture and VM-independent metrics by comparing metrics on the bytecode level, which is the common intermediate language for Java and Scala. The considered metrics include code size metrics, instruction mix statistics based on bytecodes, call site polymorphism, stack usage, method and basic block hotness, argument passing, use of reflection, and primitive boxing. PCA is used to argue that the benchmarks are different from each other and thereby provide relevant insights to assess a JVM's performance.

JetStream, Kraken, Octane In contrast to the JVM vendor-independent benchmark suites, vendors developed their own suites for JavaScript. Specifically, Apple has JetStream, Mozilla has Kraken, and Google has its Octane suite. The main difference between them seems to be the specific goal and weighting of workloads within the suites. The benchmarks themselves seem to be selected based on the application scenarios they represent. Furthermore, the suites seem to select benchmarks based on historic precedence, i.e., they include benchmarks that seem to have proven to be useful indicators in the past. While the suites provide a brief characterization of each benchmark, there does not seem to be a formal characterization based on metrics.

2.2 Cross-Language Benchmarks

While discussing the pros and cons of programming languages has a long tradition and performance is often a part of it, there does not seem to be any scientific effort in providing a comprehensive set of cross-language benchmarks.

The Computer Language Benchmarks Game. The Computer Language Benchmarks Game is the only widely recognized resource for benchmarks in different languages. Currently, it provides a collection of 14 benchmarks for 29 language implementations. The rules for benchmark implementations state that the same algorithm should be used and that an implementation should produce the same output. Beyond these two requirements, benchmarks can be implemented freely for each language. As a result, the Benchmark Game has a wide variety of implementations for the same benchmark in a single language. Variations include different choices for how to express the algorithm as well as fundamentally different approaches for instance by using parallel techniques instead of sequential code.

From a scientific perspective this means that the Benchmark Game's rules are too permissive and allow for a too great variation, which precludes many relevant conclusions from such benchmarks. For this paper, the goal is to determine whether a language implementation successfully opti-

[1] *SPECjvm2008*, `https://www.spec.org/jvm2008/`

[2] *Introducing the JetStream Benchmark Suite*, Apple Inc., access date: 2016-02-14, `https://webkit.org/blog/3418/introducing-the-jetstream-benchmark-suite/`

[3] *Octane*, Google, `https://developers.google.com/octane/`

[4] *Kraken*, Mozilla Foundation, `https://wiki.mozilla.org/Kraken`

[5] *The Computer Language Benchmarks Game*, Isaac Gouy, `http://benchmarksgame.alioth.debian.org/`

mizes the execution of a given program with a focus on the compiler. Thus, the benchmarks need to behave identical to allow us to draw conclusions about the used compiler.

Other Projects. To our knowledge, there are only few other projects that approach cross-language performance comparison systematically. This includes a comparison [6] of C++, Java, Go, and Scala based on a loop recognition algorithm [4]. This experiment gained a lot of attention from users of these languages and was strongly criticized for the way the benchmark was implemented for the different languages. This includes for instance criticism on the Java version's[6] inefficient use of the collection library and that the Go version[7] should better use arrays instead of maps and missed some caching that was introduced in the Java version. Thus, the benchmarks between the languages were not actually similar enough to allow for strong conclusions. From these examples we conclude that a cross-language benchmarking methodology requires a precise set of goals and rules, which the benchmarks need to follow carefully.

Another project is the TechEmpower Web Framework Benchmarks.[8] This project uses small web services to compare web frameworks and languages. These services serialize JSON, query a database, or answer fortune-cookie messages. The goal is to assess the performance of a complete web stack, including a language's runtime as well as a specific framework. While this in itself is of high practical value, it would not allow us to draw conclusions about the effectiveness of a specific compiler.

2.3 Metrics for Benchmark Characterization

Dufour et al. [3] established guidelines for dynamic metrics that can be used to characterize benchmarks, which are also used by DaCapo and DaCapo con Scala. The guidelines are meant to design metrics to summarize a benchmark in a concise informative manner, allow for a comparison of benchmarks, and guide compiler optimizations.

Dufour et al. identify five requirements. First of all, metrics need to be *unambiguous* so that it is clearly defined what is measured. Furthermore, to characterize benchmark behavior, the metrics need to be *dynamic*, i.e., relate to run-time aspects. They also need to be *robust*, i.e., not overly dependent on program input, and *discriminating* such that behavioral changes in a program are reflected in the metrics. Finally, they should be *machine independent*. For the purpose of this paper, the requirements need to be extended also to be reasonably language-independent (cf. section 4).

[6] *Scala/Java Shootout*, Jeremy Manson, access date: 2016-02-11, `http://jeremymanson.blogspot.co.at/2011/06/scala-java-shootout.html`

[7] *Profiling Go Programs*, Russ Cox, access date: 2016-02-11, `http://blog.golang.org/profiling-go-programs`

[8] *TechEmpower Web Framework Benchmarks*, TechEmpower Inc., `https://www.techempower.com/benchmarks/`

The metrics identified by Dufour et al. include size measurements such as the number of classes of an application, the loaded classes, and the amount of loaded, executed, and compiled bytecodes. The behavioral metrics include characteristics of the branching behavior and control flow changes, the density of array, floating point, and pointer operations, as well as metrics on the polymorphism of object-oriented behavior with respect to call site density, receiver, and target polymorphism. Additionally they also collect a number of metrics on allocation behavior and object sizes. We use these metrics as a template for defining language-independent metrics for this paper.

3. A Methodology for Cross-Language Benchmarks

This section defines our goals for the performance comparison of different languages. It further defines the *core* language abstractions to be used by the benchmarks and how it can be mapped on common object-oriented languages. Finally, it specifies requirements for the implementation of benchmarks to ensure the performance comparison gives insights into the effectiveness of the compiler and relevant parts of the runtime system.

3.1 Goals

The goal of this work is to create a foundation for comparing the performance of a set of *core* language abstractions between languages. We believe this is useful for early-stage language implementations to assess the performance. Furthermore, we believe it can enable researchers to compare their research platform of choice with established industrial-strength systems and draw reliable conclusions about the compiler's effectiveness. Specifically, we aim for a benchmark suite with the following characteristics:

Relevant Abstractions Benchmarks focus on abstractions that are likely to be relevant for the peak performance of a wide range of applications in different languages, namely basic abstractions such as objects, arrays, and closures.

Portable The benchmarks rely only on language abstractions that are part of the *core* language and can be mapped straightforwardly to target languages. Furthermore, they rely on a minimal set of primitive operations to ensure portability. Finally, the size of the benchmarks is practical. The size needs to balance porting effort and workload representativeness compared to application-level benchmarks.

However, it is a non-goal to provide application-level benchmarks. We believe that this would be neither practical nor result in realistic results for cross-language comparisons. Languages come not only with a syntax and a set of libraries, but also with a *philosophy* that influences program design and thus determines the *performance sweet spot* for a language implementation. We assume this to be different between languages and thus, a cross-language application-level benchmark would either test too many different aspects

to be meaningful, or use the language in a way that is too restricted to represent the behavior of applications in that language (cf. section 3.2).

Focus on Compiler Effectiveness The benchmark suite focuses on assessing compiler effectiveness. This means, the degree by which the core language abstractions are optimized by the compiler and the runtime system. Hence, comparing the performance of standard libraries or efficiency of GCs are non-goals. Individual benchmarks can be sensitive for instance to the performance of the GC, but a detailed assessment is outside the scope of this project.

To exercise the compilers abilities to optimize complex code, the benchmark suite includes smaller and larger benchmarks. However, it excludes purely synthetic microbenchmarks, e.g., for simple field accesses, since it requires larger benchmarks to assess effectiveness of for instance inlining.

Ease of Use To facilitate adoption, a simple and reliable methodology for executing benchmarks needs to be defined. Furthermore, the setup requirements need to be minimal.

3.2 The *Core* Language

For portability of benchmarks across different languages, we select a set of common abstractions that forms a *core* language to be used by the benchmarks. While we compare object-oriented (OO) languages, the benchmarks can be implemented in other languages as well as long as the polymorphic nature of the benchmark code can be expressed.

Required Abstractions The set of required abstractions is:

- objects with fields, could also be records or structs
- polymorphic methods on user-defined classes or types
- closures, i.e., anonymous functions with read and write access to variables in their lexical scope
- an array-like abstraction, ideally with a fixed size
- strings, integers, and doubles
- automatic memory management, i.e., garbage collection

For some languages, a mapping to these abstractions is trivial. For other languages, we have guidelines.[9] For example, Java does not support writing to variables in the outer scope of a closure. Therefore, we follow the common practice to use an array as a workaround.

The legend of fig. 3 lists all basic operations used by the benchmarks, which are thus part of the required abstractions.

Excluded Abstractions Not permitted is the use of:

- built-in data structures such as hash tables, dictionaries, stacks, or queues
- object-identity-based hash
- non-local returns, except in an '`if`' or to implement iteration functions on collections
- flow control in loops with `continue`, `break`, etc, except to implement iteration functions on collections

These abstractions are excluded, because they are not portable and cannot be mapped easily to other constructs. For instance, JavaScript does not provide the notion of an object-identity-based hash, and the differences between collections in different languages can be a source of performance variation, which is outside the scope of this work. As replacement for built-in data structures, we provide a portable collection library.

Relevance of the* Core *Language Since the chosen abstractions are widely supported by different languages, we conjecture them to be widely used in programs as well. While these abstractions excludes languages-specific ones that are relevant for general program performance, these abstractions are integral for the overall performance of languages that support them. As such, we see an optimal implementation of this *core* language as *a necessary condition* to achieve optimal application performance. However, depending on a specific language and its features, optimizing this core language is not a sufficient condition.

3.3 Requirements for Benchmark Implementations

To ensure that these benchmarks assess compiler effectiveness, it is necessary to define how benchmarks are ported between languages (cf. section 2.2).

As Identical as Possible. The first and foremost requirement is that benchmarks must be as *identical* as possible between languages. This is achieved by relying only on a widely available and commonly used subset of language features and data types. These language features also need to be used consistently between languages. For example, to iterate over arrays, a benchmark should use a `forEach()` method that takes a closure, or should fall back to a `for-each` loop construct if available in the language. We prefer the use of a `forEach()` method, because it is commonly supported, is a high-level abstraction, and exercises the compiler's abilities. Another example is the semantics of arrays. Languages such as Java and Smalltalk offer only fixed-sized arrays, while JavaScript and Ruby have dynamically-sized arrays. To ensure that the executed operations are as identical as possible, we use arrays as fixed-sized entities in all languages. This ensures that we benchmark the same operations, and expose all compilers to the same challenges.

Idiomatic Use of Language. To stay within the performance sweet spot of language implementations, the second most important requirement is to use a language as *idiomatically* as possible, i.e., following best practices. As a guideline for idiomatic language usage, we rely on widely used lint and code style checking tools for the languages. In some cases this might be a tradeoff with the requirement to be as identical as possible. We prefer here the more idiomatic solution if the result is close enough in terms of use of control

[9] `https://github.com/smarr/are-we-fast-yet#guidelines`

structures, closures, and other performance-related aspects, and stays within the *core* language. Simple idiomatic differences include that a comparison in Java '`if (node != null)`' is written simply as '`if node`' in Ruby.

This also includes structural information not present in all languages, e.g., types and information about immutability. Thus, in languages such as Java, we use the `private` and `final` keywords on fields and methods to provide hints to the compiler. We consider this idiomatic and desirable language-use to enable optimizations.

Well-typed Behavior. The benchmarks are designed to be well-typed in statically-typed languages, and behave well-typed in dynamic languages to ensure portability and identical behavior. For dynamic languages this means that initialization of fields and variables avoids suppressing possible optimizations. For example, fields that contains primitive values such as integers or doubles are not initialized with pointer values such as `null` but with the appropriate `0`-value to ensure that the implementation can optimize these fields.

Fully Deterministic and Identical Behavior. The benchmarks are designed to be fully deterministic and show identical behavior in the benchmark's code on all languages. This means, repeated executions of the benchmark take the same path through the benchmark code. On the one hand, this is necessary to ensure reproducible results, and on the other hand, this is required to ensure that each language performs the same amount of work in a benchmark. However, differences of language semantics and standard libraries can lead to differences in the absolute amount of work that is performed by a benchmark.

4. Benchmark Characterization

The benchmark suite presented in this paper consists of 5 larger commonly used benchmarks and 9 smaller ones. The benchmarks were selected based on their code size and anecdotal relevance, i.e., whether other suites include them as well. Table 1 lists all benchmarks with a brief description. We characterize them based on metrics that are designed to be language agnostic, and show that especially the larger benchmarks show a wide range of different characteristics.

4.1 Metrics

To characterize the benchmarks, we use size, structural, and behavioral metrics. However, we disregard purely static metrics because they can be misleading in the context of dynamic compilation. Static metrics would include code that is not executed, only influencing minor aspects such as load and parse time, which are not relevant to assess compiler effectiveness. Furthermore, we select metrics that are designed to be language agnostic, i.e., that lead to similar results across languages. This is similar to the approach of Dufour et al. [3] and Sewe et al. [13] that select metrics that are independent for instance of the underlying execution environment. The following sections detail the selected metrics and discuss their cross-language properties.

All metrics are measured for SOMns,[10] a Newspeak implementation based on Truffle [18]. The values correspond to what would be measured for languages such as Java.

4.1.1 Code Size

To give an intuition about the size of benchmarks, table 1 lists for each benchmark the following metrics.

Executed Lines of Code Compared to the classic notion of lines of code (LOC), we count only lines of code that have been executed at least once to measure the dynamic size of the program instead of its static size. This metric is sensitive to language differences for defining classes and methods. In languages such as Ruby or JavaScript, class or method definitions would be counted as imperative statements. For Java or Newspeak, these are declarative definitions that are not included in the number of executed LOC metric. Thus, we exclude them from this metric.

Classes The number of classes includes only classes of which at least one method was executed. It is stable across languages with the exception of JavaScript which does not use classes in the version of the language used for this work.

Executed Methods Similar to counting executed LOC, we count methods that have been executed at least once. This metric is stable across languages, with the exception of built-in classes, e.g., for integers or strings. Since the benchmark structure is fixed, the set of defined methods is fixed, too.

Per Iteration Methods In addition to executed methods, we further distinguish methods that are executed for each benchmark iteration. Thus, we separate out code that was only executed once during startup or shutdown. The number of per iteration methods indicates the methods that get likely compiled during benchmarking.

4.1.2 Dynamic Metrics

The measured dynamic metrics characterize in more detail the behavior of the benchmarks. For each metric, we determine and count the number of lexical sites and the number of times the corresponding behavior has been executed.

Method Calls We measure the observed variability at call sites and count the number of observed receiver types. To make the metric language-independent, classic operators such as '+' or '*' are excluded from the method call count. This is necessary to ensure that results for languages such as Smalltalk and Ruby are comparable to Java or JavaScript. Since benchmarks have identical code structure in all languages, the type hierarchies and the methods on objects are identical, too. With these properties, the metric is stable across languages for method calls on user-defined types.

[10] `https://github.com/smarr/SOMns#readme`

Macro Benchmarks		Exec. LOC	Classes	Exec. Methods	Per Iter. Methods
CD	Airplane collision detector simulation [7]	356	16	43	41
DeltaBlue	Classic object-oriented constraint solver [15]	387	20	99	75
Havlak	Loop recognition algorithm [6]	421	18	110	87
Json	JSON string parsing	232	14	56	56
Richards	Classic operating system kernel simulation [11]	279	12	47	47
	Micro Benchmarks				
Bounce	Simulation of a box with bouncing balls	42	5	11	11
List	List creation and traversal	30	2	9	9
Mandelbrot	Classic Mandelbrot computation	39	0	2	2
NBody	Classic n-body simulation of solar system	105	3	14	14
Permute	Generate permutations of an array	33	3	13	13
Queens	Solver for eight queens problem	36	3	13	13
Sieve	Sieve of Eratosthenes	22	3	9	9
Storage	Tree of arrays to stress GC	23	4	10	10
Towers	Towers of Hanoi	42	2	12	12

Table 1. Benchmark suite and code size metrics. The numbers exclude elements shared by all benchmark.

Closure Applications Similar to method calls, we measure the number of lexical closures observed at a closure application site. A closure application site is the lexical point where a closure is executed. This metric is stable with respect to closures defined in benchmarks and executed at application sites in the benchmark code. However, it is sensitive to the availability and implementation details of, for instance, `forEach()` within the different standard libraries.

Maximum Stack Height As an indication for the recursive behavior, we measure the maximal observed stack height, i.e., the number of method activations on the runtime stack. This metric is sensitive to differences in the standard libraries, e.g., the use of `forEach()` instead of a lexical loop.

Loops We count loops that have been activated at least once. Furthermore, we count the number of times a loop body has been executed. This metric is sensitive to the availability of `forEach()` methods. We report the numbers for a language with such methods. For languages such as Java, which has a separate for-each loop construct, the number of loops is higher. The number of loop activations is however the same, because `forEach()` is implemented with some looping construct and is counted as such.

Branches We count the number of control flow branches that have been taken at least once. This includes `if`-branches, but also operations that have control flow semantics such as short-cutting `or` and `and` operators where the right-hand expression is executed conditionally. Furthermore, we count how often each branch is taken. This metric is stable between languages with the same caveats as method calls.

Allocations For array allocations, we track the number of arrays created as well as their overall size. For objects, we track the number of objects created and the number of declared fields. The array metrics are stable between languages as long as languages support array creation with a predefined size. The object metrics are sensitive to the way objects and/or classes are defined, which might not be done statically. For example in JavaScript and Ruby objects are constructed dynamically instead of declaratively.

Object Field Accesses We count object field reads and writes that were executed at least once. Furthermore, we report the number of accesses per iteration. This metric is expected to be stable between languages with only minor variations. For instance, Newspeak, Smalltalk, and JavaScript do not have the notion of constants, which thus need to be modeled for instance as object fields. In Newspeak, one way to model them is as final fields of an object in an outer lexical scope, which means they are counted as field accesses.

Variable Accesses We report the number of sites of variable reads and writes that were executed at least once. This includes variables in methods and closures. Furthermore, we report the number of accesses per iteration. This metric is stable between languages with only minor variations. One source of variation is the availability of `forEach()` methods on arrays. If `forEach()` or a method to iterate with an index is not available, additional local variables are used to read an array element and store it for use in the loop body.

Array Accesses We count the sites of array reads and writes that were executed at least once. Furthermore, we count the number of array reads and writes per iteration. This metric is stable between languages, assuming that array operations such as copying are tracked in detail so that each read and write access is reported.

Basic Operations Basic operations are also known as primitives or built-in functions. We include comparisons, arithmetic and bit operations, reading the size of strings or arrays, and string operations. Since the complexity of these

operations range from simple integer additions, which can be mapped directly to a processor instruction, up to trigonometric functions, or string comparisons, which require complex algorithms, we categorize them in groups with similar properties as shown in the legend of fig. 3.

We count operations that were executed at least once, as well as the number of executions per iteration. These metrics are stable across languages for the used operations in the benchmark code, i.e., differences in the standard library of languages can lead to minor variations.

4.2 Results

To characterize the benchmarks briefly, this section presents and discusses them based on the chosen metrics.

Polymorphism and Stack Usage Table 2 shows that the smaller benchmarks do not have polymorphic method calls. For the larger benchmarks, the polymorphism varies widely. For closure applications, we see a higher degree of polymorphism. Overall however, the degree of method and closure polymorphism remains small compared to application-level benchmarks, e.g., of DaCapo.

Stack usage, indicated by the maximum observed stack height is overall low as well. The only exception is the Havlak benchmark, which traverses a graph recursively.

Control Flow Instructions The relative frequency of control flow operations is depicted in fig. 1. This groups method calls, closure applications, branches, and loop iterations. Method calls and closure applications are further categorized into monomorphic and polymorphic to detail the elements that might require additional optimization.

Allocations and Accesses The relative frequencies of array and object allocations as well as read and write accesses to them are shown in fig. 2. For most benchmarks, field reads are the most common operations of this group. However, smaller benchmarks can show different behavior.

Basic Operations The relative frequency of basic operations is depicted in fig. 3. From this plot one can see high-level characteristics of the benchmarks. For instance, the behavior of Mandelbrot and NBody is dominated by floating point operations. The JSON parsing benchmark has a high number of string operations, and the collision detector (CD) benchmark has many floating point comparisons.

Benchmark Variation To validate that the benchmarks in the suite differ from each other, we apply a principal component analysis (PCA) on all metrics as it is done by Blackburn et al. [1] and Sewe et al. [13]. The analysis takes 69 different measurements per benchmark as input. This includes code size as well as structural and dynamic characteristics. A large fraction of the measurements are the different groups of basic operations and their structural and dynamic characteristics. The PCA determines from these 69 metrics 14 correlated variables, of which the first four account for a cu-

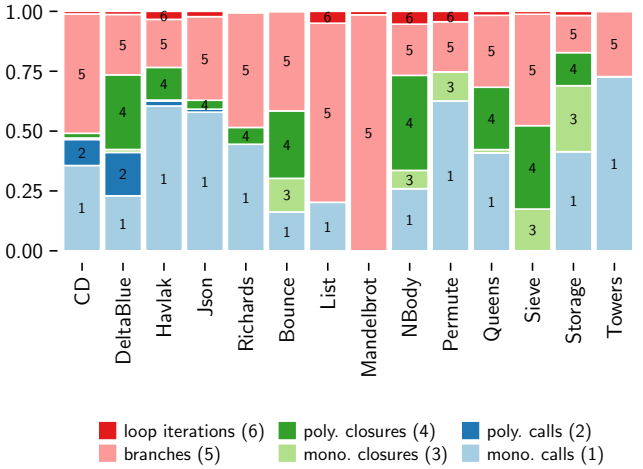

Figure 1. Control flow instructions including method calls, closure applications, branches, and loop iterations.

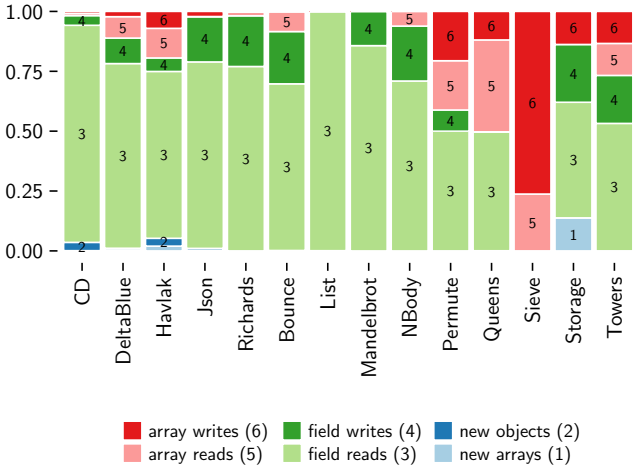

Figure 2. Allocation and accesses for objects and arrays.

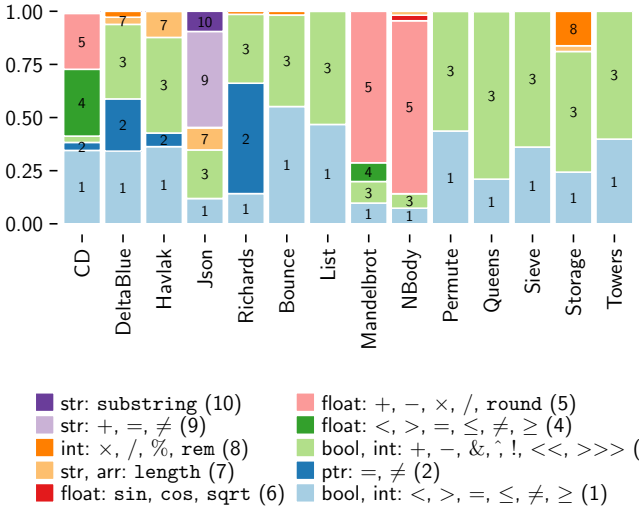

Figure 3. Mix of basic operations grouped by similar complexity.

	Method Calls				Closure Application				
	Monomorphic		Polymorphic		Monomorphic		Polymorphic		
Benchmark	sites	calls	sites	calls	sites	applies	sites	applies	max. stack
CD	166	75,892,212	3	23,151,272	2	1,345,202	3	4,188,512	35
DeltaBlue	171	6,219,974	23	4,896,692	13	336,022	4	8,432,152	37
Havlak	256	79,581,122	3	2,720,556	1	531,748	5	17,938,956	1,717
Json	163	32,115,418	2	678,200	0	0	3	2,070,004	43
Richards	108	95,521,018	2	10,000	1	7,600	3	15,027,604	31
Bounce	31	17,736,018	0	0	2	15,300,000	3	30,756,004	30
List	28	17,271,018	0	0	1	3,002	1	3,002	37
Mandelbrot	10	16	0	0	2	4	0	0	15
NBody	38	8,500,102	0	0	2	2,500,012	2	13,000,052	24
Permute	25	51,982,018	0	0	2	10,090,000	2	16,004	44
Queens	28	29,088,018	0	0	1	960,000	2	18,504,004	56
Sieve	19	60,018	0	0	1	30,000,000	2	60,006,004	26
Storage	23	32,774,018	0	0	2	21,840,000	2	10,924,004	75
Towers	27	39,390,018	0	0	2	18,002	1	1,202	34

Table 2. Statistics on method calls, closure application, and stack usage.

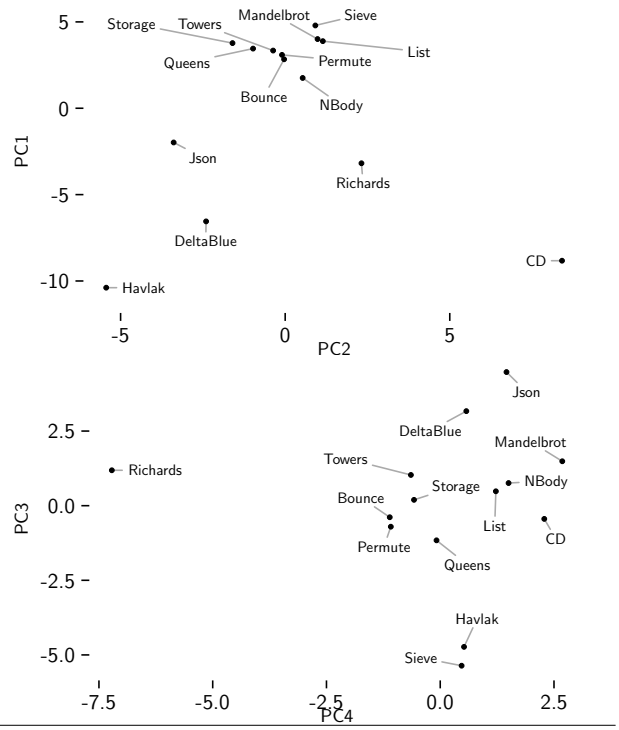

Figure 4. PCA Scatter plot. Components 1 to 4 (PC1 to PC4) of the principal component analysis over all obtained metrics, which account for 72% of the overall variance. Closer benchmarks are more similar to each other.

mulative variance of 72%. This means, these four components characterize the main differences of the benchmarks with respect to the measured metrics.

We use these four components to visualize the relation between the benchmarks in fig. 4. As shown in the plots, the larger benchmarks differ from each other significantly on the four main axes. The smaller benchmarks however show more similar characteristics, which is indicated by them being grouped closer together. While there is less variation between them in this analysis, we include them all in the performance evaluation of the next section because in our experience, each of them uses language features in ways that are different enough to identify performance issues.

5. Performance Measurements

This section analyses the performance results to assess whether our benchmarks allow us to compare compiler efficiency across different languages.

Experimental Setup. We use 10 language implementations of which four are implemented on the Graal and Truffle platform. These languages are JavaScript with Oracle's Graal.js[11] implementation, Ruby with JRuby+Truffle,[12] Newspeak with SOMns, as well as Smalltalk with TruffleSOM.[13] To complement these languages, we include Java 8 based on Oracle's HotSpot JVM as a baseline. Furthermore, we use Node.js, which uses Google's V8 JavaScript VM[14] as another comparison point for a common language. For Ruby, we also include MRI, i.e., the reference implementation, JRuby,[15] and Rubinius[16] to see how the benchmarks behave on implementations with a wide range of different optimiza-

[11] *Graal.js*, `http://www.oracle.com/technetwork/oracle-labs/program-languages/index.html`

[12] *JRuby+Truffle – a High-Performance Truffle Backend for JRuby*, `https://github.com/jruby/jruby/wiki/Truffle`

[13] *TruffleSOM*, `https://github.com/SOM-st/TruffleSOM`

[14] *Chrome V8*, Google, `https://developers.google.com/v8/`

[15] *JRuby*, `http://jruby.org/`

[16] *Rubinius*, `http://rubini.us/`

tion levels. Finally, we include Crystal,[17] a statically-typed derivate of Ruby as a statically compiled language without JIT compilation but with garbage collection, which is currently required for the benchmarks.

Benchmarking Methodology. JIT compilation, GC, and the general non-determinism of the underlying system are accounted for by repeating benchmarks within the same process for a number of times. For the slowest VMs (Ruby and Rubinius), we measure 100 iterations for each benchmark. For JRuby, which is faster, we measure 250 iterations. For the other languages, we measure 3000 iterations. We chose 3000 iterations, because the Graal compilation threshold is at 1000 method invocations, which means that the main method of the benchmark harness is compiled after 1000 iterations, and this may trigger follow-up compilations. For the slower implementations, this was impractical, but they do not require the same degree of precision.

The results are the averages over all data points for each benchmark, ignoring the first n iterations that show signs of compilation based on manual inspection. Thus, the reported performance corresponds to the peak performance, which is dominated by the effectiveness of the compiler and aspects such as object and closure representation.

To facilitate a simple and reliable execution of benchmarks, we use ReBench[18] to document benchmarking parameters and execute the experiments.

The benchmark machine has two 4-core Intel Xeons E5520, 2.26 GHz with 8 GB RAM and runs Ubuntu Linux with kernel 3.13. Truffle languages use GraalVM 0.12. The other languages use Oracle Java HotSpot 1.8.0_81, Node.js 6.2, Crystal 0.17.4, Ruby MRI 2.3.1, Rubinius 3.14, JRuby and JRuby+Truffle 9.1.2.0.

5.1 Cross-Language Benchmarking Evaluation

To assess whether our approach allows us to answer the question whether a set of core language abstractions in one implementation is as fast as in another one, we compare the results for Graal.js, JRuby+Truffle, SOMns, and TruffleSOM. These are four languages that all use the Graal compiler as part of the GraalVM. Thus, the compiler is the same, and the variable parts of the experiment are the language semantics and implementation choices. This means, if the benchmarks allow for a comparison between languages, we should see identical or close to identical performance for these implementations.

Figure 5 shows an aggregated overview of the results for all benchmarks as a boxplot. It shows that the four Truffle languages are grouped closely together. Taking SOMns as the base line, using the geometric mean over all benchmarks, Graal.js is about 9%, TruffleSOM about 10%, and JRuby+Truffle about 10% slower. Considering the major differences in language semantics between these languages, we consider this difference small and a strong indication for the suitability of our approach. As a comparison, Node.js with its highly optimizing V8 JavaScript VM is about 42% slower than SOMns, which reveals different design choices in the compiler, as we detail in section 5.2.

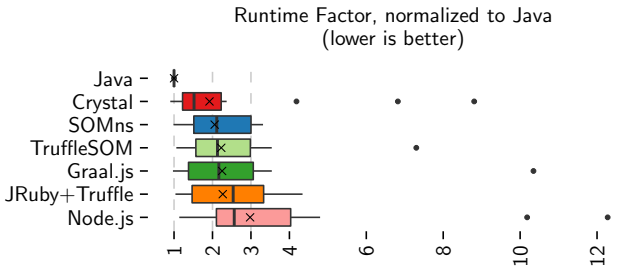

Figure 5. Boxplot over 14 benchmark results per language implementation. Normalized to Java. Boxplot indicates median with a vertical line, and geometric mean with an ×.

Figure 6 shows all benchmarks. We briefly discuss some of the outliers to support the claim that our approach is suitable to determine the effectiveness of the used compiler.

The first outlier is for TruffleSOM on the CD benchmark. The slow part of the benchmark accesses the value of an element in the red-black tree. Truffle's object model [17] optimizes the element for the different types it is used with, which currently leads to a megamorphic access that is not yet correctly optimized in TruffleSOM.

The second significant outlier is visible for the Havlak benchmark on Graal.js and Node.js. In this case, the hash table implementation of the benchmark shows an access pattern that causes both JavaScript implementations to represent an array internally sparsely with a hash table. Arguably, JavaScript programmers are unlikely to implement similar data structures manually so that this heuristic is useful for the common use cases in JavaScript.

Other minor performance outliers are visible for SOMns on the List and NBody benchmarks. They show optimization potential in SOMns' custom object model, which is not based on the Truffle framework.

Based on our analysis of the outliers, we find indications that the performance differences are based on different runtime representation choices. However, we do not find significant performance differences that are attributable to the compiler itself, which would be unexpected. From these results, we conclude that the benchmarks allow for cross-language comparison and can reach the same performance independent of the language.

Conclusion With this experiment for four languages on top of the same compiler, we find an average performance variation of 10% between the languages. Considering the significant differences between these languages, we see the results as indication that our approach can be used to assess the compiler effectiveness across languages.

[17] *The Crystal Programming Language*, `http://crystal-lang.org/`

[18] *ReBench*, `https://github.com/smarr/ReBench`

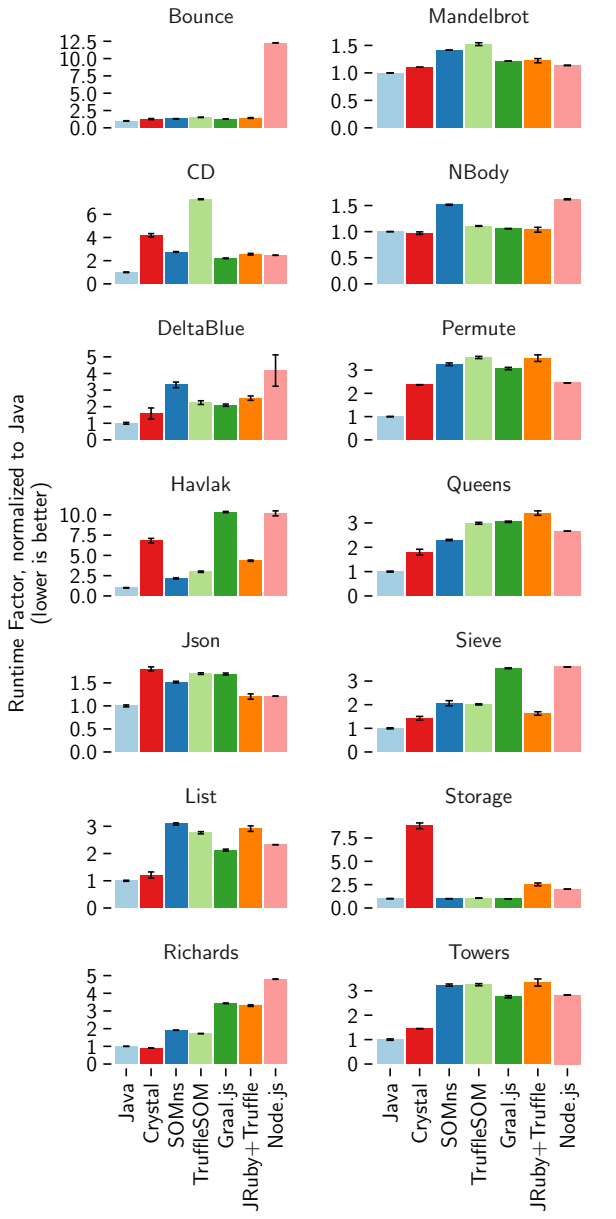

Figure 6. Benchmark results for the fast implementations.

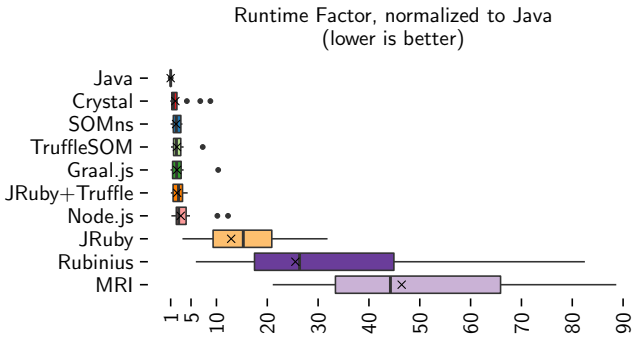

Figure 7. Boxplot over 14 benchmark results per language implementation including slower implementations.

5.2 Overall Performance

To complete the discussion, we briefly analyze the performance of the remaining language implementations. Figure 7 is similar to fig. 5 but includes the slower implementations.

Overall, for the 14 benchmarks we can conclude that Java on the HotSpot VM reaches the highest peak performance. The second fastest language implementation is Crystal, which uses LLVM as a compiler backend. Performance-wise it is close to Java, but shows for instance on Storage (a GC-bound benchmark) a 8.7x overhead, because it uses the conservative Boehm collector [2].

The Truffle-based implementations reach the range of about 2x slower than Java on these benchmarks. Node.js however shows slightly worse performance and has two outliers. The first outlier is the Bounce benchmark, which exemplifies the main reason for V8 not reaching better overall performance. Specifically, its inlining heuristic is not as aggressive as Graal's. Since V8 is designed for the browser, where startup and warmup behavior is crucial, the implementation chooses different tradeoffs than Hotspot and the Graal compiler, which are more geared towards long-running server applications. The second outlier for Node.js is the Havlak benchmark, for which V8's array heuristic, similar to Graal.js' is not optimized for using an array as backing storage for a hash table, causing the VM to choose a sparse hash-table-based representation for the array itself, which leads to a significant slowdown.

Using the geometric mean over all benchmarks, JRuby, which compiles to JVM bytecodes, is on average 14x slower than Java. Rubinius, with its LLVM-based custom baseline JIT compiler is about 26x slower, and MRI, which is a classic bytecode interpreter is about 46x slower than Java. Here we see the pattern observed for many language implementations: the interpreter is about one to two orders of magnitude slower than a highly optimizing VM. Code produced by a JIT compilers, depending on the available optimizations, is typically only up to one order of magnitude slower than the best observed performance.

6. Performance Engineering Anecdotes

When developing and optimizing language implementations, every new benchmark can expose new and unexpected behavior. From that perspective, this benchmark suite is not different. However, as an indication for the spectrum that the benchmarks cover and as an indication that they can support implementers, we report on concrete anecdotes.

As shown in the previous section, well-optimized implementations of different languages can reach close to identical performance on these benchmarks. Assuming that the same holds for a new language, the benchmarks can be used to identify concrete optimization potential. As an example, when starting with the experiments, we noted that JRuby+Truffle was consistently slower on all benchmarks.

The Graal compiler failed to optimize JRuby+Truffle's argument arrays because some type information was not propagated correctly. After identifying the issue, it was possible to fix this performance bug in Graal itself, and JRuby+Truffle reached the expected performance.

Other performance outliers indicated similar issues that can be considered performance bugs in the language implementations. In the case of JRuby+Truffle, the benchmarks helped to identify issues with unnecessary array initializations as well as a virtual call in performance-critical string handling code that could be avoided.

For some languages however, it is not as clear-cut. As discussed earlier, Graal.js and Node.js show a performance anomaly for the Havlak benchmark, because the array that is used as backing storage for a hash table is 'optimized' to use a hash-table-based implementation itself. Here the heuristic determines that the array is used as a sparse storage and in these languages the most common use case is similar to a hash table, which the implementations optimize for.

More clear examples are Node.js' outlier on Bounce and Richards. Bounce is the extreme example of V8's tradeoff choices for less aggressive inlining. V8 does currently not inline the use of `forEach()` in the benchmark,[19] which causes the high overhead. For Richards, V8 missed an optimization opportunity for comparisons with `null`, which is fixed in V8, but not yet in the used Node.js.[20]

7. Discussion

Influence of Language Differences on Peak Performance One of the major issues of cross-language benchmarking is the difference in language semantics and standard libraries. Thus, the question is how one can compare implementations despite these differences. For example, Ruby and JavaScript have variable-length arrays in the language, while Java and Smalltalk have fixed-sized arrays. From our results and the comparably minor differences in peak performance between the Truffle languages, we conclude that such differences have only minor impact on peak performance in the presence of a highly optimizing compiler and runtime.

Furthermore, with our methodology we focus on the effectiveness of the compiler by minimizing differences, e.g., by using a custom collection library. As a result, the main difference that remains, which we have not assessed in detail in this work, is how the dynamic features between languages influence peak performance. With dynamic features we refer for instance to possible changes in object structure or class hierarchy, which might need to be checked at run time. However, we assume that the majority of performance-relevant checks are moved out of performance-sensitive loops. Again, this is supported by the narrow performance range reached by four sufficiently different languages in our experiments.

[19] `https://bugs.chromium.org/p/v8/issues/detail?id=5041`

[20] `https://bugs.chromium.org/p/v8/issues/detail?id=5042`

Language Independence of Metrics As discussed in section 4.1, the various metrics have different degrees of language independence and thus, the measured numbers may vary between different languages. However, with our methodology, we require the benchmark code to be as identical as possible across languages. Furthermore, as our evaluation shows, a highly optimizing compiler can achieve very similar performance for different languages. This means, for the main purpose of this paper, the variability of metrics between languages does not have a major impact on the resulting performance.

Generalizability of Results As argued in section 3.1, the used benchmarks are not application benchmarks. Applications tend to have less hot code and exhibit more variety in their code patterns and dynamic behavior [9, 10]. Nonetheless, the used benchmarks allow for a comparison between language implementations and languages. We conjecture that they are a useful tool to optimize the core elements of many languages as indicated by our results and engineering anecdotes (cf. section 6). We consider that optimizing for these benchmarks can be a necessary condition for good application performance. However, it is not a sufficient condition. The benchmarks only cover *core* language abstractions and are not comprehensive with respect to any particular investigated language.

8. Related Work

As detailed in section 2.2, existing approaches to cross-language benchmarking have all different focuses and do not allow for a comparison of the compiler effectiveness between languages. Examples include the Computer Language Benchmarks Game (CLBG), the work of Hundt [6], and the TechEmpower Web Framework benchmarks. To our knowledge, this project is the first to provide a systematic approach and precise rules to enable such an assessment.

Other work focuses on characterizing differences of languages. Sarimbekov et al. [12] examine the CLBG benchmarks to determine their behavior for Clojure, Java, JRuby, and Jython based on dynamic metrics. Li et al. [8] do a similar study adding Scala and additional applications for each language. Such studies guide language implementations and optimizations at a high level. Our work however makes compiler efficiency comparable across languages.

9. Conclusion

Comparing implementations of different languages has been a recurring issue in academia and industry. This paper presents a novel methodology and freely available benchmark suite[21] to compare the effectiveness of compilers across languages by ensuring that the workload exposed to each language is as identical as possible. This is achieved by using a common *core* language in these benchmarks. We

[21] *Are We Fast Yet?*, `https://github.com/smarr/are-we-fast-yet`

conjecture that optimizing this core language is a necessary condition to reach peak performance for applications because it consists of basic and common abstractions. However, optimizing this core language is not a sufficient condition to reach peak performance, because applications will use language-specific abstractions, too.

This paper characterizes 14 benchmarks based on metrics and shows that the benchmark suite contains a wide variety of dynamic behaviors. Furthermore, it uses these benchmarks to compare 10 language implementations. We find that implementations for JavaScript, Newspeak, Ruby, and Smalltalk based on the Graal and Truffle platform reach on average very similar peak performance, close to being only 2x slower than Java. This indicates that these benchmarks can be used to compare performance across different languages. Based on performance engineering anecdotes, we argue that these benchmarks provide insights for language implementers since they allow the comparison of languages and thereby provide a concrete performance goal to be reached. In conclusion, this paper presents an approach to assess whether a set of *core* language abstractions is as fast as other implementations of the same abstractions.

Currently, the benchmarks are designed to compare the compiler effectiveness across languages. They could be extended to also assess, e.g., garbage collection and standard library design. Our work could further be used to assess fundamental performance differences between languages in the presence of highly optimizing compilers, e.g., considering tradeoffs for language design and compilation technology for statically or dynamically-typed languages as well as statically and dynamically-compiled implementations.

Acknowledgments

We would like to thank Danilo Ansaloni and Andreas Wöß for discussions and comments on early versions of this paper, Vyacheslav Egorov and the V8 team for triaging and fixing bugs, and the Oracle Labs VM Research group for their support. Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31.

References

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. of OOPSLA*, pages 169–190. ACM, 2006. doi: 10.1145/1167473.1167488.

[2] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proc. of PLDI*, pages 197–206. ACM, 1993. doi: 10.1145/155090.155109.

[3] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic Metrics for Java. In *Proc. of OOPSLA*, pages 149–168. ACM, 2003. doi: 10.1145/949305.949320.

[4] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997. doi: 10.1145/262004.262005.

[5] K. Hoste and L. Eeckhout. Microarchitecture-Independent Workload Characterization. *IEEE Micro*, 27(3):63–72, 2007. doi: 10.1109/MM.2007.56.

[6] R. Hundt. Loop Recognition in C++/Java/Go/Scala. In *Proc. of Scala Days*, 2011. URL `https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf`.

[7] T. Kalibera, J. Hagelberg, P. Maj, F. Pizlo, B. Titzer, and J. Vitek. CDx: A Family of Real-time Java Benchmarks. *Concurrency and Computation: Practice and Experience*, 23 (14):1679–1700, 2011. doi: 10.1002/cpe.1677.

[8] W. H. Li, D. R. White, and J. Singer. JVM-hosted Languages: They Talk the Talk, but Do They Walk the Walk? In *Proc. of PPPJ*, pages 101–112. ACM, 2013. doi: 10.1145/2500828.2500838.

[9] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *Proc. of WebApps*. USENIX, 2010.

[10] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proc. of PLDI*, pages 1–12. ACM, 2010. doi: 10.1145/1809028.1806598.

[11] M. Richards. Bench, 1999. URL `http://www.cl.cam.ac.uk/~mr10/Bench.html`.

[12] A. Sarimbekov, A. Podzimek, L. Bulej, Y. Zheng, N. Ricci, and W. Binder. Characteristics of Dynamic JVM Languages. In *Proc. of VMIL*, pages 11–20. ACM, 2013. doi: 10.1145/2542142.2542144.

[13] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *Proc. of OOPSLA*, pages 657–676. ACM, 2011. doi: 10.1145/2048066.2048118.

[14] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 Performance Characterization. In *Proc. of SPEC Benchmark Workshop*, pages 17–35. Springer, 2009. doi: 10.1007/978-3-540-93799-9_2.

[15] M. Wolczko. Benchmarking Java with Richards and Deltablue, 2013. URL `http://www.wolczko.com/java_benchmarking.html`.

[16] T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One VM to Rule Them All. In *Proc. of Onward!*, pages 187–204. ACM, 2013. doi: 10.1145/2509578.2509581.

[17] A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. An Object Storage Model for the Truffle Language Implementation Framework. In *Proc. of PPPJ*, pages 133–144. ACM, 2014. doi: 10.1145/2647508.2647517.

[18] T. Würthinger, A. Wöß, L. Stadler, G. Duboscq, D. Simon, and C. Wimmer. Self-Optimizing AST Interpreters. In *Proc. of DLS*, pages 73–82, 2012. doi: 10.1145/2384577.2384587.