$\mathbf{Scala} \stackrel{?}{\equiv} \mathbf{Java} \bmod \mathbf{JVM}$ — On the Performance Characteristics of Scala Programs on the Java Virtual Machine

Andreas Sewe
Software Techology Group
Technische Universität Darmstadt
Darmstadt, Germany
sewe@st.informatik.tu-darmstadt.de

ABSTRACT

In recent years, the Java Virtual Machine has become an attractive target for a multitude of programming languages, one of which is Scala. But while the Scala compiler emits plain Java bytecode, the performance characteristics of Scala programs are not necessarily similar to those of Java programs. We therefore propose to complement a popular Java benchmark suite with several Scala programs and to subsequently evaluate their performance using VM-independent metrics.

1 Introduction

While originally conceived as target of the Java programming language only, the Java Virtual Machine (JVM) [1] has since become a target for hundreds of programming languages, the most prominent ones arguably being Clojure, Groovy, Jython, JRuby, and Scala. The JVM can therefore rightly be considered a *Joint* Virtual Machine.

Targeting such a joint virtual machine offers a number of engineering benefits to language implementers: After more than 15 years of research and development the Java platform is very mature. Moreover, it is not only mature but portable, wide-spread, and offers a staggering amount of libraries to choose from. Last but not least, the platform is backed by several high-performance JVMs. Alas, simply targeting the JVM does not always result in performance as good as Java's; existing JVMs are primarily tuned with respect to the performance characteristics of Java programs.

Of the five languages mentioned above, four languages share one key characteristic: Clojure, Groovy, Python, and Ruby are all dynamically typed. As this single language feature has been identified as the biggest performance bottleneck, the Java Community Process has put forth a specification request (JSR 292) to "[Support] Dy-

namically Typed Languages on the Java^{TM} Platform," i.e., to close the semantic gap between dynamically-typed source languages and Java bytecode.

While a semantic gap undoubtedly exists for statically-typed source languages like Scala [2] as well, it is less clear what the bottlenecks are. This work-in-progress therefore aims to shed light on the performance characteristics of Scala programs. In particular, we will answer the following three questions: Are the performance characteristics of Scala programs, from the JVM's perspective, similar or dissimilar to those of Java programs? If they are dissimilar, what are the assumptions that implementers of a JVM have to reconsider? And are Scala programs sufficiently different to warrant special treatment—as the dynamically-typed languages now receive?

2 Characterising the Performance of Scala Programs

Previous investigations into the performance of Scala programs have been mostly restricted to micro-benchmarking.[1] While such benchmarks are undeniably useful to the implementers of the Scala compiler, who have to decide between different code generation strategies for a given language feature, they are less useful to implementers of a Java VM, who have to deliver good performance across a wide range of real-world programs, only some of which are written in Scala. Our research will therefore assume the latter's viewpoint, in turn making the following contributions:

1. A benchmark suite of Scala programs developed as an extension to the popular DaCapo benchmark suite [3].
2. The definition of VM-independent metrics to characterise the performance of Scala programs.
3. A VM-independent comparison of the performance characteristics of Scala programs and Java programs.

2.1 Towards a Scala Benchmark Suite

The following programs (along with potential input data) have been selected for inclusion in the benchmark suite. As of October 2010, half of the implementations are stable (marked †); Figure 1 on page 3 relates their size to the DaCapo benchmarks'.

kiama† The Kiama library for language processing (compiling and interpreting the Obr and ISWIM languages, respectively).

lift The Lift web framework (running its example application).

scalac† The "New" Scala compiler (compiling and optimising the Scalaz library).

scalap† A Scala classfile disassembler (disassembling a complex classfile).

scalatest ScalaTest, a testing framework supporting various testing styles, including JUnit and TestNG integrations (running its own test suite).

specs† Specs, another testing framework, which makes heavy use of embedded domain-specific languages (running its own test suite).

tmt The Stanford Topic Modeling Toolbox, a natural language processing framework driven by Scala scripts (learning a model using Latent Dirichlet Allocation).

[1] The language's implementers themselves perform a number of so-called shoot-outs, each testing a particular language feature: `http://www.scala-lang.org/node/360`.

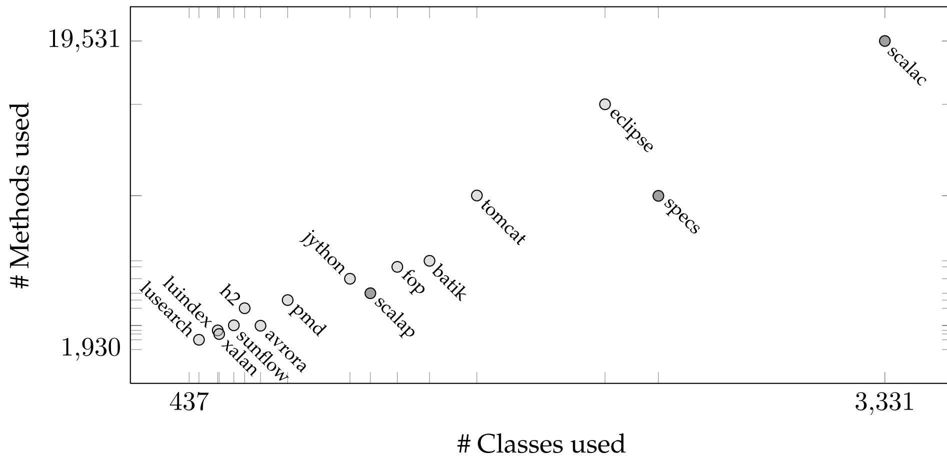

Figure 1: The size and complexity of 15 benchmark programs (excluding harness) written in Java (○) and Scala (◉), respectively.

A few of the above benchmarks incorporate a significant amount of code written not in Scala but in plain Java. This choice is deliberate, as it reflects current practice; candidates either employ Scala facades to Java libraries (scalatest, specs) or run on an infrastructure written entirely in Java (lift). The following table summarises this for a selection of Scala benchmarks.

Benchmark	# Method Calls		
	Java JRE	Java (other)	Scala
scalac†	7.29%	0.22%	92.49%
scalap†	29.83%	0.04%	70.13%
specs†	89.99%	0.06%	9.95%

2.2 Towards VM-Independent Benchmark Comparisons

Possible metrics to compare benchmarks in a VM-independent fashion are based on object demographics or the structure of the static and dynamic call graphs. Hereby, metrics based on object demographics have been used extensively to characterise the DaCapo benchmarks [3]; thus, we will sketch a few metrics of the latter group below.

Two of the most effective optimisations a JVM can perform are adaptive recompilation and method inlining. Just how effective these optimisations are is determined, to a large degree, by the program's weighted dynamic call graph; the larger the weight of a vertex, the more profitable is recompiling the corresponding method; the larger the weight of an edge, the more profitable is inlining the corresponding call. Each of these optimisations, however, comes at a cost. Any dynamic metric must thus be related to a static metric which reflects the cost of performing said optimisations. In either case, it is essential for the purpose of our study to discern the influence of code written in Scala from code written in Java within the same benchmark program.

One metric of particular interest is the number of tail-calls which Scala programs exhibit. While the JVM does not yet support the notion of hard tail calls and thus will not guarantee tail-call optimisation, such optimisations are often assumed to be necessary to fully support functional languages on the JVM. The degree to which tail-calls are used in the aforementioned benchmarks determines whether such an

optimisation would also be beneficial to existing programs, whether written in Scala or Java. In particular, this metric would shed some light on the Scala compiler's effectiveness in eliminating tail-calls (cf. Section 3.1).

3 Future Directions

In the following we will outline a few research directions into which we will embark once the above contributions have been made.

3.1 Optimising Compiler vs. Optimising VM

The semantic gap between Scala source code and Java bytecode is wider than the gap between Java source and bytecode. It is therefore likely that the peculiar nature of the bytecode derived from Scala sources inhibits some of the optimisations a production JVM will perform on Java programs.

The Scala compiler scalac is thus able to perform several optimisations on its own: method inlining, escape analysis (for closure elimination), and tail call optimisation. All these optimisations have traditionally been the domain of the JVM. Working offline, however, the compiler can spend considerably more time optimising. It does not have access to online profiles, though. The key question is thus whether the semantic gap is wide enough to warrant the re-implementation of optimisations within the compiler or whether the VM is the proper place for these optimisations.

3.2 JVM vs. Common Language Runtime

Scala targets a second platform besides the JVM, namely the Common Language Runtime (CLR). This gives rise to further questions: Do the answers to the above questions carry over to the CLR? If so, what makes such a generalisation possible?

Acknowledgments

Thanks go to the entire team behind the DaCapo benchmark suite, who have provided us with a rock-solid foundation to work on.

This work was supported by CASED (www.cased.de).

References

[1] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[2] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2008.

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Portland, Oregon, USA, 2006.