# Scala $\overset{?}{\equiv}$ Java mod JVM

TECHNISCHE UNIVERSITÄT DARMSTADT

## On the Performance Characteristics of Scala Programs on the Java Virtual Machine

### 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 Towards a Scala Benchmark Suite

Previous investigations into the performance of Scala programs have been mostly restricted to micro-benchmarking. (The language's implementers themselves perform a number of so-called shoot-outs, each testing a particular language feature) While undeniably useful to the implementers of the Scala compiler, such micro-benchmarks 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. Thus, a full-fledged benchmark suite consisting of both Scala and Java programs is needed.

The following programs (along with potential input data) have been selected for inclusion in our Scala benchmark suite, developed as an extension to the popular DaCapo suite [2]. As of this writing, more than half of the implementations are finished and under evaluation (†).

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

lift The Lift web framework, which uses Scala's actor library to good effect (running its example application on the Tomcat Servlet container).

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

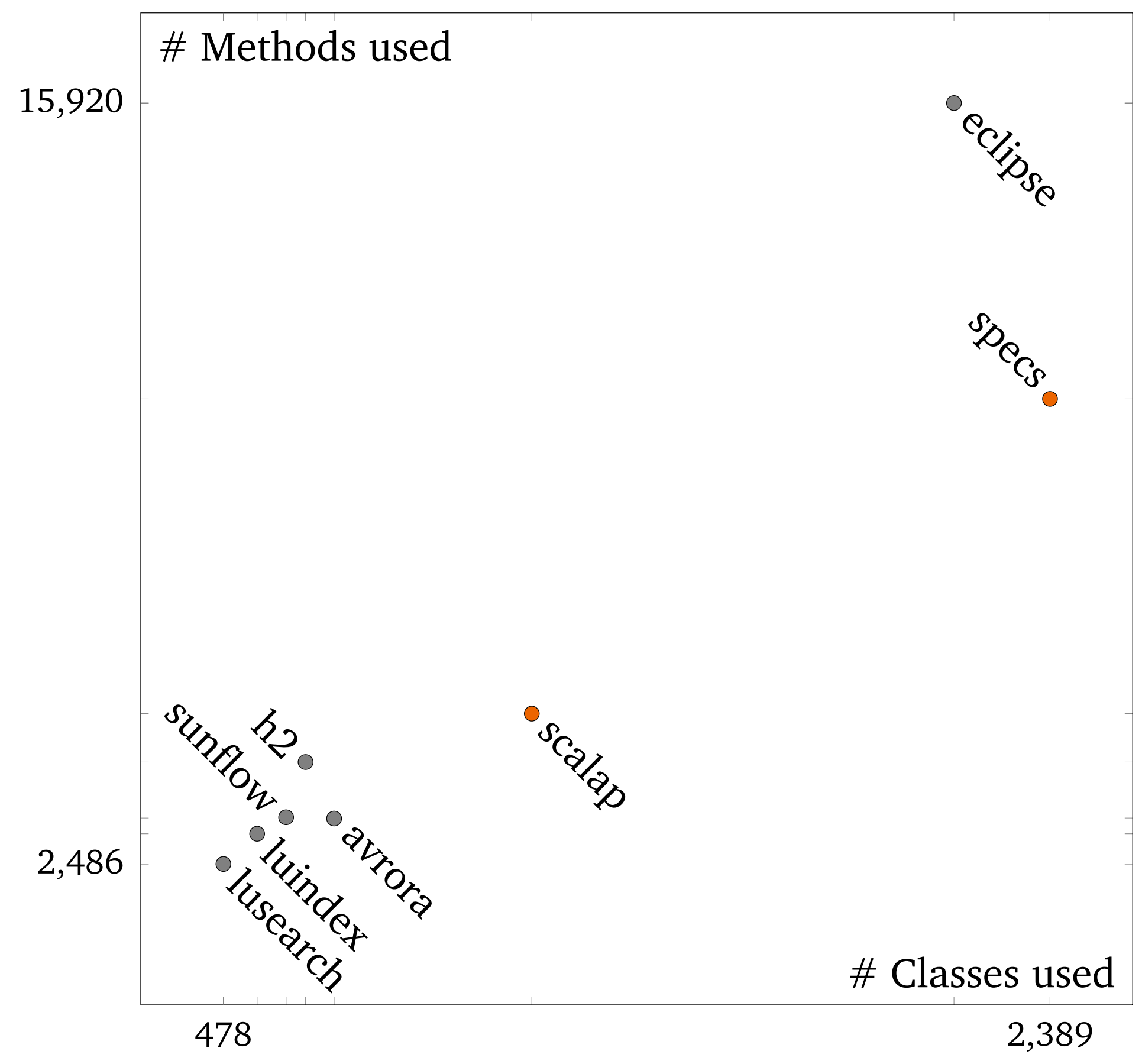
specs† Specs, another testing framework, which makes heavy use of embedded domain-specific languages (running its own test suite).

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

A few of the above benchmark incorporate a significant amount of code written not in Scala but in plain Java. This choice is deliberate, as it reflects current practice; Scala programs spend a considerable amount of time within the JRE itself.

| Benchmark | # Calls | | |
|---|---|---|---|
| | Java JRE | Java (other) | Scala |
| avrora | 3.29% | 96.71% | n/a |
| eclipse | 15.64% | 84.36% | n/a |
| h2 | 26.07% | 73.93% | n/a |
| luindex | 25.69% | 74.31% | n/a |
| lusearch | 28.43% | 71.57% | n/a |
| scalap† | 29.83% | 0.04% | 70.13% |
| specs† | 89.99% | 0.06% | 9.95% |
| sunflow | 16.96% | 83.04% | n/a |
| | # Bytecodes executed | | |
| avrora | 2.57% | 97.43% | n/a |
| eclipse | 9.36% | 90.64% | n/a |
| h2 | 26.57% | 73.43% | n/a |
| luindex | 17.91% | 82.09% | n/a |
| lusearch | 19.78% | 80.22% | n/a |
| scalap† | 51.32% | 0.11% | 48.57% |
| specs† | 94.91% | 1.33% | 3.76% |
| sunflow | 0.052% | 99.48% | n/a |

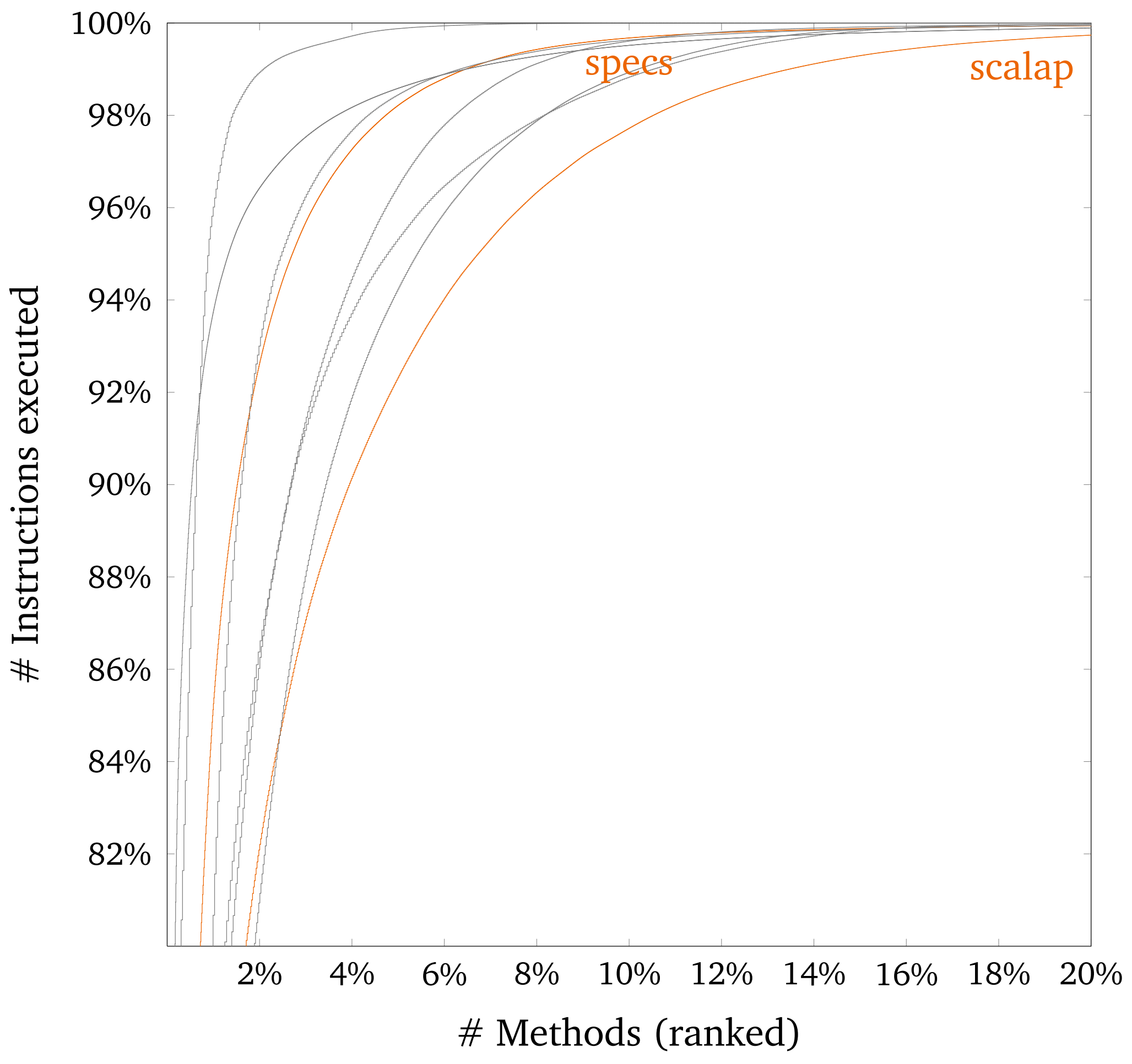
The following figure relates the benchmarks' sizes to the DaCapo benchmarks'. As can be seen, even simple Scala programs like scalap consist of thousands of classes, although the number of (called) methods per class is, in general, lower than for their Java counterparts.



All of the above measurements were conducted using JP [1], a tool for VM-independent, complete calling-context profiling. (Due to technical limitations, momentarily only a subset of benchmarks is covered.)

### 2 Towards VM-Independent Benchmark Comparisons

In order to make general claims about the similarities of Scala and Java with respect to performance, VM-independent metrics are needed. Moreover, these metrics needs to be relevant in the sense that they correlate with either optimisation opportunities themselves or with the cost of exploiting said opportunities. Two such metrics are related in the following figure.

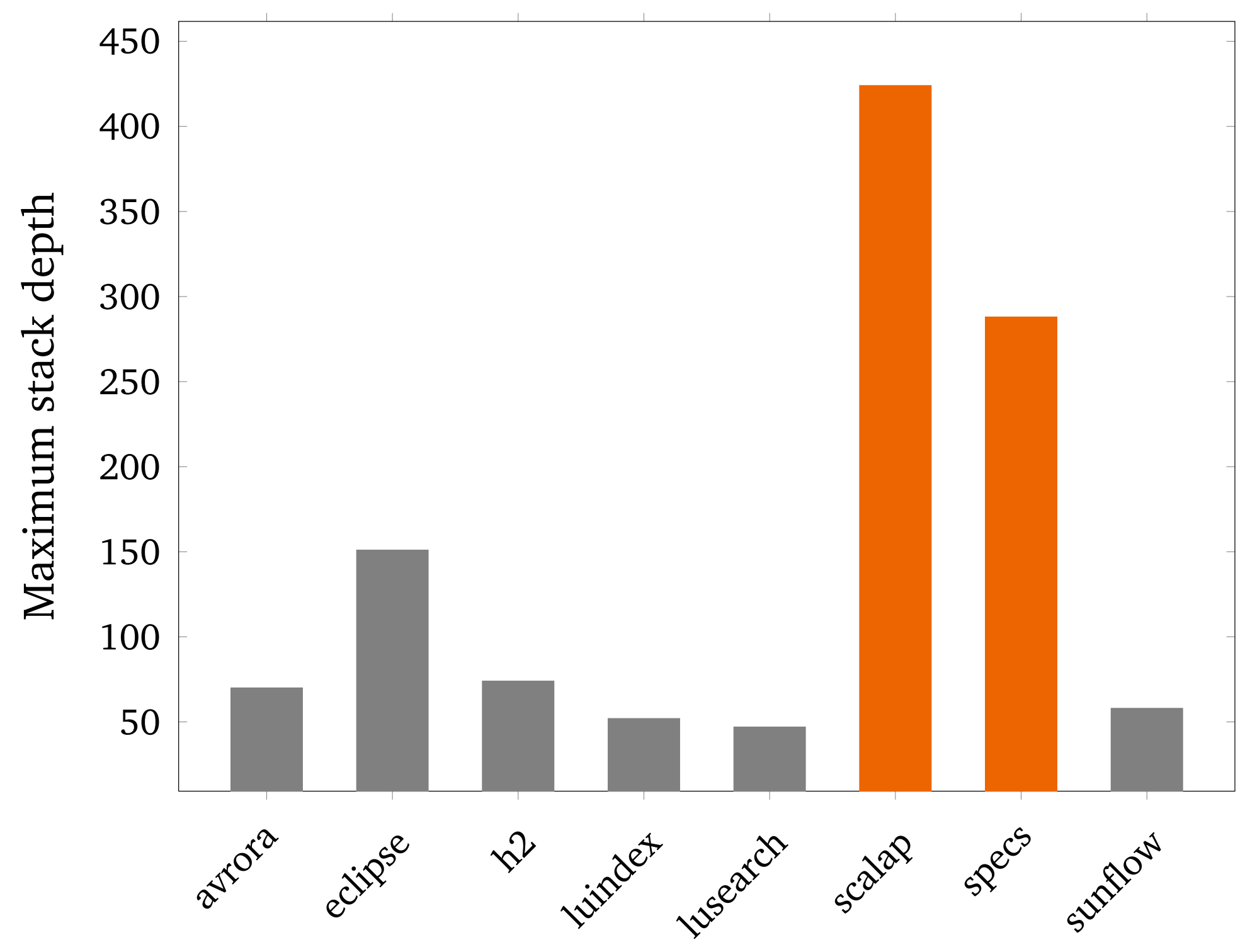


One novel metric measures the methods' **argument usage**. (We have already begun collecting information on argument usage with MAJOR [3].) As Scala supports higher-order functions, emulating them by means of function objects, it is of particular interest how a method's arguments are used, e.g., to guide inlining decisions. Consider the following example.

```
public void example(String[] args) {
    Function2 lt = new Example$$anonfun$1();
    Ordering ltOrdering =
        ordering.fromLessThan(lt);
    sorting.quickSort(args, ltOrdering);
}
```

When compiling the above method, inlining quickSort has a potentially large pay-off, as knowledge about the used Ordering can be propagated to the point of its actual usage. But this optimisation opportunity hinges on the fact that quickSort invokes methods on its ltOrdering argument—rather than passing it along or simply storing it.

Another metric of 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. Moreover, this metric would shed some light on the Scala compiler's effectiveness in eliminating tail-calls itself. Regardless, the maximum stack depth of Scala programs is often much higher than for Java programs.



### 3 Future Directions

As the semantic gap between Scala source code and Java bytecode is wider than the gap between Java source and bytecode, the trade-offs involved in the **optimising compiler vs. optimising VM** decision needs to be investigated anew. Unlike the Java compiler, the Scala compiler already performs several optimisations on its own: method inlining, escape analysis (for closure elimination), and tail call optimisation. It is an open question, however, whether the semantic gap is wide enough to warrant such re-implementations of optimisations within the compiler or whether the VM remains the proper place for optimisations.

Also, Scala targets a second platform besides the JVM, namely the Common Language Runtime (CLR). This choice, **JVM vs. Common Language Runtime**, offers the unique opportunity to put all our findings to the test: Are they specific to a single platform or is a generalisation possible?

### References

[1] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39:47–79, 2009.

[2] 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 *Proceedings of the 21st Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, Portland, Oregon, USA, Oct. 2006.

[3] A. Villazón, W. Binder, P. Moret, and D. Ansaloni. MAJOR: rapid tool development with aspect-oriented programming. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 125–128, New York, NY, USA, 2009.

### Acknowledgments

CASED

### Contact Information

Andreas Sewe
sewe@st.informatik.tu-darmstadt.de
Technische Universität Darmstadt
Hochschulstr. 10
64289 Darmstadt, Germany