Da Capo con Scala

Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine

Andreas Sewe Mira Mezini
Technische Universität Darmstadt
{sewe, mezini}@st.informatik.tu-darmstadt.de

Aibek Sarimbekov Walter Binder
University of Lugano
{aibek.sarimbekov, walter.binder}@usi.ch

Abstract

Originally conceived as the target platform for Java alone, the Java Virtual Machine (JVM) has since been targeted by other languages, one of which is Scala. This trend, however, is not yet reflected by the benchmark suites commonly used in JVM research. In this paper, we thus present the design and analysis of the first full-fledged benchmark suite for Scala. We furthermore compare the benchmarks contained therein with those from the well-known DaCapo 9.12 benchmark suite and show where the differences are between Scala and Java code—and where not.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Performance attributes; D.2.8 [*Metrics*]: Performance measures

General Terms Languages, Measurement, Performance

Keywords Benchmarks, dynamic metrics, Scala, Java

1. Introduction

While originally conceived as a target platform of the Java language only, the Java Virtual Machine (JVM) [31] has since been targeted by numerous programming languages, ranging from Ada to Z-code. Among the most popular of these are Clojure, Groovy, JRuby, Jython, and Scala, which have gathered a following in recent years, as they bring expressive constructs to a stable and portable platform.

However, the benchmark suites so often used in JVM research do not yet reflect this development. Not only do older benchmark suites like SPECjbb2005[1] and Java Grande [10] altogether ignore languages other than Java, but also are the two modern benchmark suites commonly used in research still firmly Java-focused: The SPECjvm2008 suite[2] does not contain a single benchmark written in a non-Java language. The DaCapo suite [7] contains only a single one, namely jython. But while a single such benchmark might serve as an interesting curiosity,[3] it does not allow for deeper insights into the execution characteristics of non-Java languages on the JVM. We have therefore set out to complement the existing Java benchmark suites with a new suite featuring a large set of non-Java applications.

The language we chose for our endeavour is Scala [34], a statically-typed language with roots in both functional and object-oriented programming. The reason for choosing a single language rather than several is simple: Any relevant benchmark suite needs to cover a broad selection of real-world applications; choosing a single language leads to a comprehensive yet cohesive suite. The reason for preferring Scala over other possible candidates like Clojure, Groovy, JRuby, or Jython is more involved: Of the aforementioned five languages, four are dynamically-typed. But this single language feature has significant impact [49] on the performance of the JVM, a machine which, until recently [41], has been specifically tailored towards a single, statically-type language, namely Java. In contrast, the execution characteristics of other statically-typed languages like Scala on the JVM are less well understood, in part due to the lack of a comprehensive benchmark suite. “Scala $\equiv$ Java $\bmod$ JVM?” thus becomes an interesting research question [45]: Does Scala code differ significantly from Java code[4] when viewed at the bytecode level?

In this paper we will therefore extensively analyze our Scala benchmark suite with respect to various dynamic metrics, e.g., the benchmarks’ instruction mixes, the code hotness at different levels, or the use of reflection and boxing. These metrics are computed for three distinct, but related reasons: First, to validate that the suite is indeed a valid Scala benchmark suite, i.e., that Scala code contributes significantly to the benchmarks’ executions. Second, to en-

[1] See `http://www.spec.org/jbb2005/`.

[2] See `http://www.spec.org/jvm2008/`.

[3] The jython benchmark, e.g., exhibits very many short-lived objects.

[4] For the sake of brevity, we refer to Java bytecode that was compiled from Java and Scala sources as “Java code” and “Scala code,” respectively.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10...$10.00

sure that the selected Scala benchmarks exhibit a varied set of execution characteristics. Third, to investigate whether the Scala benchmarks' execution characteristics differ from those of an established Java benchmark suite. All our metrics are hence chosen to be relevant to common JVMs but independent of any particular implementation. We will therefore focus exclusively on properties of bytecode rather than machine code. Moreover, our analysis is focused on code rather than memory and pointer behaviour; a detailed study of the latter is beyond the scope of this paper.

Our evaluation is concerned with comparing the execution characteristics of Scala and Java code;[5] it is explicitly not designed to compare the "performance of Scala" with the "performance of 'raw' Java." Such a study would require two sets of equivalent yet idiomatic applications written in both languages. Instead, our Scala benchmark suite, like the DaCapo suite we compare it against, consists of independently-developed, real-world applications.

The scientific contributions of this paper are two-fold:

1. The **design** of a new, comprehensive, and open-source Scala benchmark suite that complements the popular DaCapo benchmark suite [7].
2. The **analysis** of both suites with respect to various static and dynamic metrics.

This paper is structured as follows: First, Section 2 gives a brief overview of our evaluation setup. Next, Section 3 describes our benchmark suite and the design criteria we applied. Then, Section 4 describes the metrics we used to compare the Scala benchmarks to each other and to the Java benchmarks of the DaCapo suite as well as the results of this analysis. Section 5 discusses related work, before Section 6 concludes with a summary of our analysis. Finally, Section 7 suggests directions of future work. Two appendices offer additional notes and observations on building a benchmark suite and on the collection of dynamic metrics.

2. Evaluation Setup

In contrast to several other researchers [7, 21], we refrain entirely from using hardware performance counters to characterize the benchmark suite's workloads. Instead, we rely exclusively on metrics which are independent of both the specific JVM and architecture used, as it has been shown that, e.g., the JVM used can have a large impact for short-running benchmarks [21]. Another, pragmatic reason for using bytecode-based metrics is that doing so does not obscure the contribution of the source language and its compiler as much as the JVM and its just-in-time compiler would do.

To collect metrics which are independent of the specific JVM, we extended the JP2 calling-context profiler [42, 43] to collect execution frequencies for basic blocks. The rich profiles collected by the call-site aware JP2 enable us to derive all code metrics—with one exception (see below)—from a single, consistent set of measurements. We use a specially-written callback [42] that ensures that only the benchmark itself is profiled by JP2; this includes both the benchmark's setup and its actual iteration, but excludes JVM startup and shutdown as well as the benchmark harness. This methodology ensures that the results are not diluted by startup and shutdown code [19]. Also, the Scala benchmarks remain undiluted by Java code used only in the harness. More information on JP2 can be found in Appendix B.

We furthermore extended TamiFlex [8] to measure the use of reflection. All these measurements have been normalized with respect to a dummy benchmark, which simply does nothing during its iteration. This normalization step ensures that reflection used during JVM startup or shutdown or within the harness does not perturb the measurements.

All measurements were conducted on a Quad-Core AMD Opteron 8356 processor with 40 GiB of RAM running build 17.1-b03 of the Java HotSpot 64 bit Server VM (build 17.1-b03, JRE build 1.6.0_22-b04) in mixed mode. Externally multi-threaded benchmarks where run with the maximum number of threads. We used version 9.12 of the DaCapo benchmark suite and a pre-release version of our own Scala benchmark suite. All benchmarks safe one (factorie) in the latter suite are compiled for version 2.8.1 of the Scala language. This ensures that any difference between Scala benchmarks are due to the workload itself rather than due to differences in the compiler.

The internal validity of our findings is high; both JP2 and TamiFlex have been shown to collect stable and complete profiles [6, 8]. The chosen JRE, however, exerts some influence on the profiles, as both Scala and Java benchmarks spend a considerable portion of their execution in the JRE's code (cf. Section 3.3). This needs to be kept in mind when interpreting the results.

3. Benchmark Design

The Scala benchmark suite is based on the latest release (version 9.12, nicknamed "Bach") of the DaCapo benchmark suite, a suite already popular among JVM researchers which specifically strives for "ease of use" [7]. Table 1 summarizes the 12 Scala benchmarks we added to the 14 Java benchmarks;[6] thus, our suite is almost on par with the current release of the DaCapo benchmark suite and larger than its previous one, despite a more limited set of well-known applications written in Scala to choose from. To allow for easy experimentation with different inputs, several benchmarks come with more than the two to four input sizes (small, default, large, and huge) typical for the DaCapo benchmarks. This gives rise to 51 unique workloads, i.e., benchmark-input combinations. The DaCapo benchmark suite offers 44 such workloads.

[5] Scala also compiles to CIL [20]. However, an analysis of Scala's execution characteristics on the .NET platform is beyond the scope of this paper.

[6] See `http://www.scalabench.org/`.

Benchmark	Description	Inputs (#)	References
actors	Trading sample with Scala and Akka actors	tiny–gargantuan (6)	
apparat	Framework to optimize ABC, SWC, and SWF files	tiny–gargantuan (6)	
factorie	Toolkit for deployable probabilistic modeling	tiny–gargantuan (6)	[32]
kiama	Library for language processing	small–default (2)	[47]
scalac	Compiler for the Scala 2 language	small–large (3)	[44]
scaladoc	Scala documentation tool	small–large (3)	
scalap	Scala classfile decoder	small–large (3)	
scalariform	Code formatter for Scala	tiny–huge (5)	
scalatest	Testing toolkit for Scala and Java programmers	small–huge (4)	
scalaxb	XML data-binding tool	tiny–huge (5)	
specs	Behaviour-driven design framework	small–large (3)	
tmt	Stanford Topic Modeling Toolbox	tiny–huge (5)	[37]

Table 1. The 12 Scala benchmarks selected for inclusion in the benchmark suite.

3.1 Covered Application Domains

The external validity of our findings hinges on the benchmarks' representativeness. We have therefore chosen a large set of applications from a range of different domains as the basis of our benchmark suite. In fact, only two categories of application are completely absent from the Scala benchmark suite but present in the latest release of the DaCapo benchmark suite: client/server applications (tomcat, tradebeans, and tradesoap) and in-memory databases (h2). Of these categories, the earlier release (2006-10) of the DaCapo suite also covers in-memory databases (hsqldb). The absence of client/server applications from our suite is explained by the fact that all three such DaCapo benchmarks rely on either a Servlet container or an application server, a dependency which a Scala benchmark within this category will have to share. In fact, a benchmark based on the popular Lift web framework was designed [45] but then discarded, as its profile proved to be dominated by the Java-based container. The absence of in-memory databases is explained by the fact that, to the best of our knowledge, no such Scala application yet exists that is more than a thin wrapper around Java code.

While the range of domains covered is nevertheless broad, several benchmarks occupy the same niche. This was a deliberate choice made to avoid bias from preferring one application over another in a domain where Scala is frequently used: automated testing (scalatest, specs), source-code processing (scaladoc, scalariform), or machine-learning (factorie, tmt). In this paper, we will thus show that the inclusion of several applications from the same domain is indeed justified; in particular, the respective benchmarks exhibit a distinct instruction mix (cf. Section 4.1).

3.2 Code Size

Using established source code metrics [12], Blackburn et al. [7] argue that the DaCapo benchmarks exhibit "much richer code complexity, class structures, and class hierarchies" than their predecessors. Source code metrics, however, are less useful when assessing our Scala benchmarks

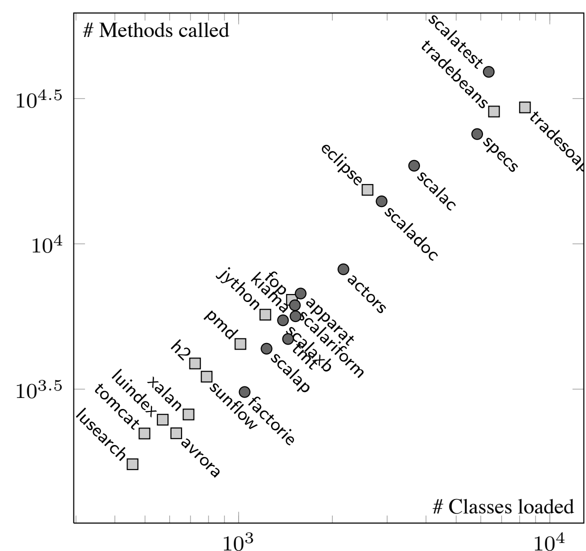

Figure 1. Number of classes loaded and method called at least once by the DaCapo (□) and Scala (●) benchmarks (default input size).

as there are, aside from various micro-benchmarks,[7] no predecessors to compare the benchmarks against. Also, source code metrics are difficult to compare across language boundaries. All metrics to compare Scala with Java code are thus based on bytecode rather than source code.

The selected benchmarks are of significant code size, comparable to those of the DaCapo benchmarks. As Figure 1 shows, even comparatively simple Scala programs like scalap consist of thousands of classes, although the number of methods actually called per class is, in general, slightly lower than for their Java counterparts. This is due to the translation strategy the Scala compiler employs [44]; a single Scala class often results in several Java classes with few methods apiece. This fact may have performance ramifica-

[7] See `http://www.scala-lang.org/node/360`.

tions, as class metadata stored by the JVM consumes a significant amount of memory [35].

For the Scala benchmarks, abstract and interface classes on average account for 13.8 % and 13.2 % of the loaded classes, respectively. For the Java benchmarks, the situation is similar: 11.3 % and 14.1 %. In case of the Scala benchmarks, though, 48.4 % of the loaded classes are marked final. This is in stark contrast to the Java benchmarks, where only 13.5 % are marked thusly. This discrepancy is in part explained by the Scala compiler's translation strategy for anonymous functions: On average, 32.8 % of the classes loaded by the Scala benchmarks represent such functions.

The methods executed at least once by the Scala benchmarks consist, on average, of just 2.9 basic blocks, which is much smaller than the 5.1 basic blocks found in the Java benchmarks' methods. Not only do methods in Scala code generally consist of less basic blocks, they also consist of less instructions, namely 17.3 on average, which is again significantly smaller than the 35.8 instructions per method of the Java benchmarks; on average, Scala methods are only half as large as Java methods.

3.3 Code Sources

For research purposes the selected benchmarks must not only be of significant size and representative of real-world applications, but they must also consist primarily of Scala code. This requirement rules out a large set of Scala programs and libraries as they are merely a thin wrapper around Java code. To assess to what extent our benchmarks are comprised of Java and Scala code, respectively, we thus categorize all bytecodes loaded by the benchmarks according to their containing classes' package names and source file attributes into one of five categories:

Java Runtime. Packages java, javax, sun, com.sun, and com.oracle; *.java source files

Other Java libraries. Other packages; *.java source files

Scala Runtime (Java code). Package scala; *.java

Scala Runtime (Scala code). Package scala;[8] *.scala

Scala application and libraries. Other packages, *.scala

Runtime-generated classes (proxies and mock classes) are categorized like the library that generated the class, even though the generated class typically resides in a different package than the generating library.

Based on the categorization, the inner circles in Figure 2 show how the loaded bytecodes are distributed among the five classes, with the circles' areas signifying the relative number of bytecodes loaded by the benchmarks. As can be seen, all benchmarks contain significant portions of Scala code, albeit for three of them (actors, factorie, and tmt) the actual application consists only of a rather small Scala kernel. Still, in terms of bytecodes executed rather than merely loaded, all but two benchmarks (actors, scalatest) spend at least two thirds of their execution within these portions, as is signified by the outer rings. The two exceptional benchmarks nevertheless warrant inclusion in a Scala benchmark suite: In the case of the actors benchmark, the Java code it primarily executes is part of the Scala runtime rather than the Java runtime. In the case of the scalatest benchmark, a vast portion of code loaded is Scala code.

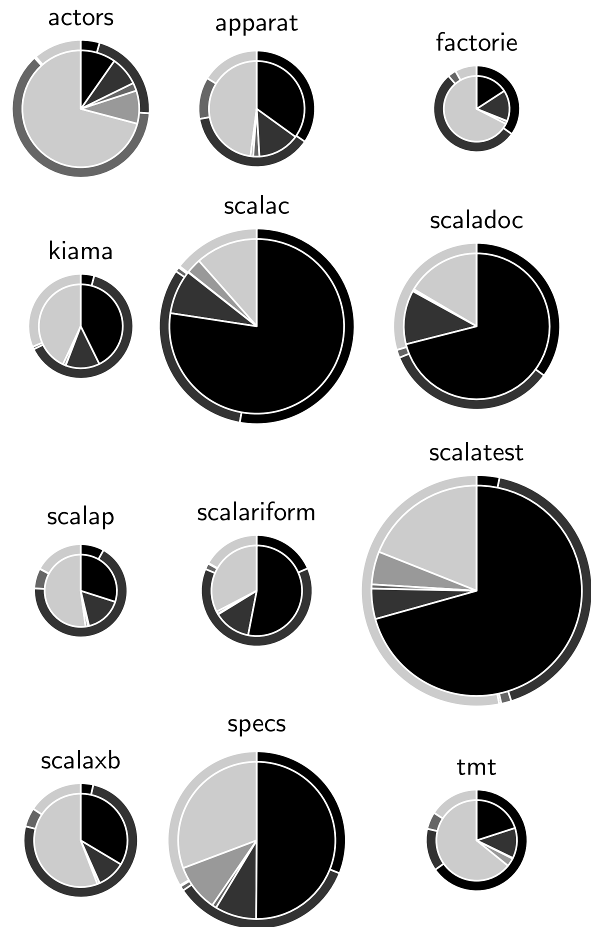

Figure 2. Bytecodes loaded and executed by each of the 12 Scala benchmarks (default input size): Java runtime (▢), Java libraries (▢), Scala runtime written in Java (■) and Scala (■), and Scala application (■).

Like the scalatest benchmark, the specs benchmark is particularly noteworthy in this respect: While it loads a large number of bytecodes belonging to the Scala application, it spends most of its execution elsewhere, namely in parts of the Scala runtime. This behaviour is explained by the fact that the workloads of both benchmarks execute a series of tests written using the ScalaTest and Specs testing frameworks, respectively. While the volume of test code is high, each test is only executed once and then discarded. This

[8] The package scala.tools was excluded; it contains, e.g., the Scala compiler and the ScalaDoc tool that are used as benchmarks in their own right.

behaviour places the emphasis on the JVM's interpreter or "baseline" just-in-time compiler as well as its class metadata organization. As such, this kind of behaviour is not well-covered by current benchmark suites like DaCapo or SPECjvm2008, but nevertheless of real-world importance since tests play a large role in modern software development.

Native method invocations are rare; on average, 0.44 % of all method calls target a native method. The actors benchmark (1.8 %), which makes heavy use of actor-based concurrency [29], and the scalatest benchmark (2.1 %), which uses the Java runtime library quite heavily, are the only notable outliers. These values are very similar to those obtained for the Java benchmarks; on average 0.49 % of method calls target native methods, with tomcat (2.0 %) and tradesoap (1.3 %) being the outliers. The actual execution time spent in native code depends on the used Java runtime and on the concrete execution platform, as none of the benchmarks analyzed in this paper contain any native code themselves. Since we focus on dynamic metrics at the bytecode level, a detailed analysis of the contribution of native code to the overall benchmark execution time is not within the scope of this paper. An initial study by one of the authors on native code execution, using the SPECjvm98[9] and SPECjbb2005 benchmarks, can be found elsewhere [5].

4. Benchmark Analysis

In the following, we analyze the Scala and DaCapo benchmark suites with respect to a variety of metrics all of which affect compilation and compiler optimizations. An analysis with respect to properties more relevant to a JVM's garbage collector [7, 16] than to its interpreter and compiler is subject to future work.

All these metrics are dynamic in nature. If we, e.g., say that a call site targets a single method only (cf. Section 4.2) and thus is particularly suitable for inlining, this means that the call site in question only had a single target method during the benchmark's execution; the call site may or may not be monomorphic in general. Also, any concrete JVM may be able to infer that a call site is de-facto monomorphic only in some of the cases, due to the inevitable limitations of any static analysis. However, we assume an idealized JVM for our metrics. The numbers we report are thus upper bounds of what a JVM can infer.

4.1 Instruction Mix

The instruction mix of a benchmark can serve as an indicator whether the application in question is, e.g., "array intensive" or "floating-point intensive;" moreover, it may show patterns discriminating Scala from Java code. We have thus used JP2 to obtain precise frequency counts for all Java bytecode instructions.

[9] See http://www.spec.org/jvm98/.

Unlike some related work on workload characterization, we neither consider all 156 instructions[10] individually [13, 14] nor do we group them manually [14, 19]. Instead, we apply principal component analysis (PCA) [36] to discern meaningful groupings of instructions. This approach offers a higher-level view of the instruction mix which is objective rather than being influenced by one's own intuition of what groupings might be meaningful.

The benchmarks are represented by vectors X in a 156-dimensional space whose components X_i are the relative execution frequencies of the individual instructions. As such high-dimensional vector spaces are hard to comprehend, we apply PCA to reduce the data's dimensionality. To this effect, we standardize each component X_i to zero mean, i.e., $Y_i = X_i - \overline{X}_i$. We do not standardize to unit variance, however. In other words, we use PCA with the covariance rather than the correlation matrix; this is justified as using the latter would exaggerate rarely-executed instructions whose execution frequency varies little across benchmarks (e.g., nop, multianewarray, floating-point coercions). PCA now yields a new set of vectors, whose uncorrelated components $Z_i = \sum_j a_{ij} Y_j$ are called principal components.

We discard those principal components that account for a low variance only, i.e., those which do not discriminate the benchmarks well, and retain 4 components that account for 58.9 %, 15.3 %, 6.4 %, and 5.6 % of the variance present in the data. Taken together, these four principal components explain more than 86.2 % of the total variance, whereas none of the discarded components accounts for more than 2.7 % of variance. Figure 3 depicts the so-called loadings, i.e., the weights $a_{ij} \in [-1, +1]$, of the four retained principal components.

We now proceed to interpret the retained principal components. For the first component, several instructions exhibit a strong positive correlation ($|a_{1j}| > 0.1$) with the first principal component: reference loads (aload), method calls (invoke...), and several kinds of method return (areturn, ireturn, and return). Likewise, one group of three instructions exhibits an equally strong negative correlation: integer variable manipulations (iinc, iload, and istore). This suggests that the first component contrasts interprocedural with intraprocedural control flow, i.e., method calls and their corresponding returns with "counting" loops controlled by integer variables. This is further substantiated by the fact that the if_icmpge instruction commonly found in such loops is also negatively correlated. The second principal component is governed by floating-point manipulations (fload, fmul, and fstore) and field accesses (getfield, putfield), all of which are positively correlated.

Figure 4 shows to what degree both the Scala and Java benchmarks are affected by these principal components. As

[10] This number is slightly smaller than the actual number of JVM instructions (201); mere abbreviations like aload_0 and goto have been treated as aload and goto_w, respectively. The wide modifier is treated similarly.

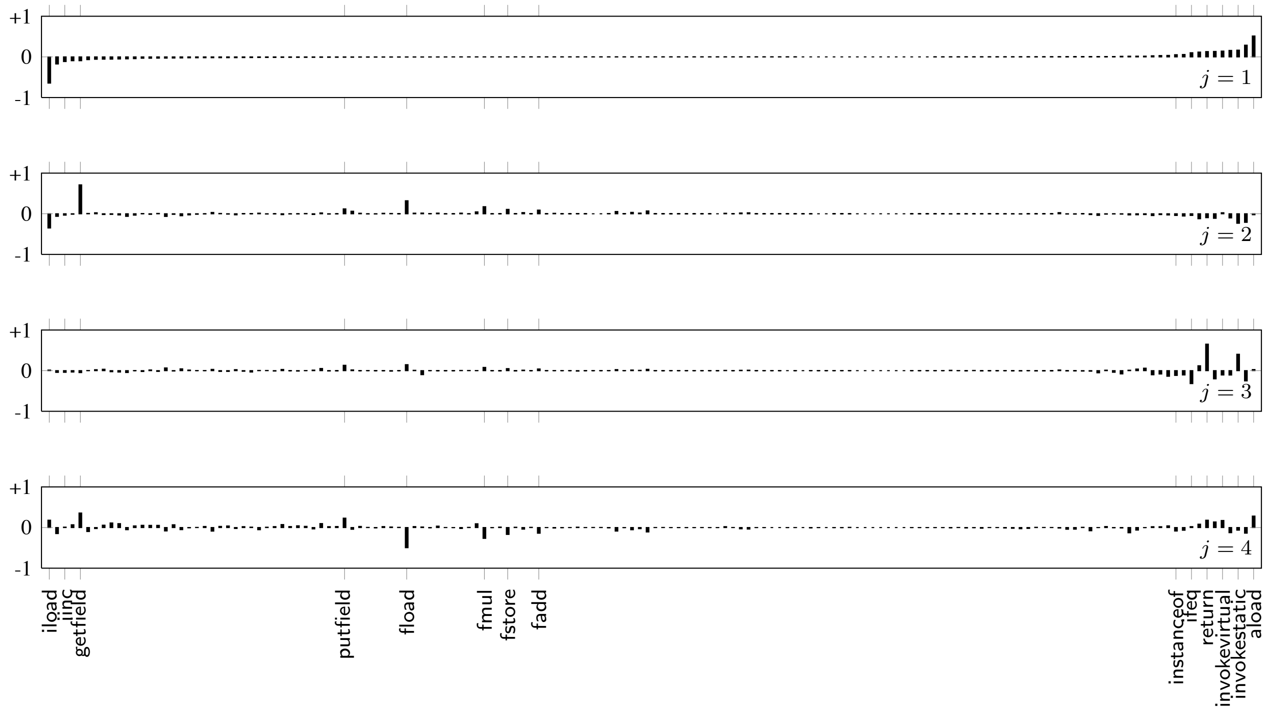

Figure 3. The top four principal components that account for 86.2 % of variance in the benchmarks' instruction mixes.

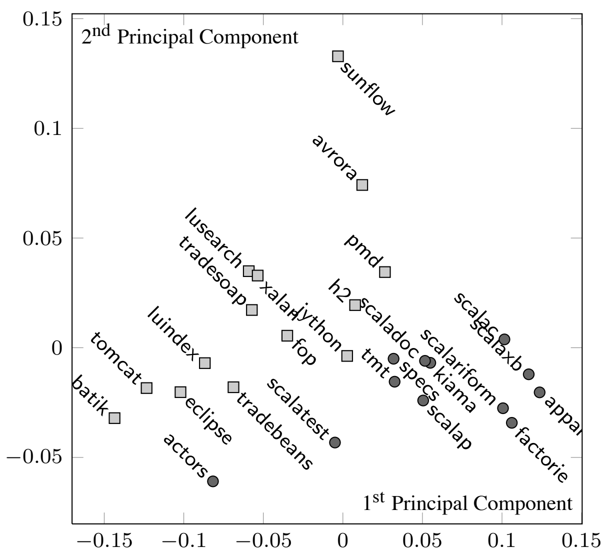

Figure 4. The DaCapo (▫) and Scala benchmarks (•) with respect to the first and second principal component.

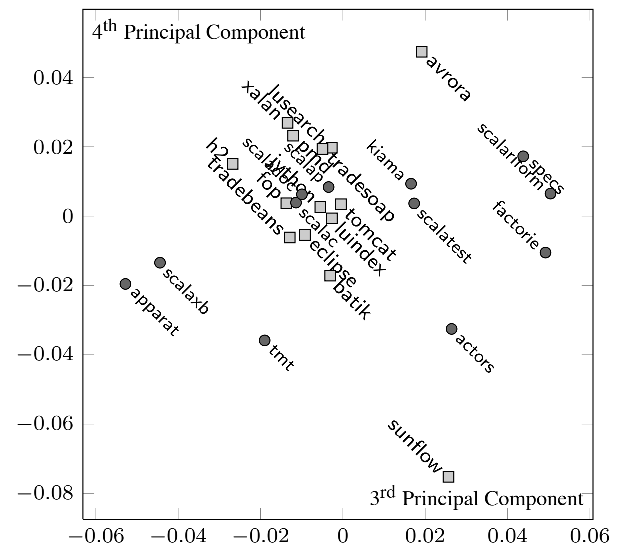

Figure 5. The DaCapo (▫) and Scala benchmarks (•) with respect to the third and fourth principal component.

can be seen, all Scala benchmarks with the exception of actors exhibit high values for the first principal component, the scalatest benchmark with its heavy usage of the Java runtime (cf. Section 3.3) being a borderline case. This shows that these Scala benchmarks strongly favor interprocedural over intraprocedural control flow. In fact, they do so to a larger degree than most of the considered Java benchmarks.

The third principal component correlates positively with calls that are dynamically dispatched (invokevirtual and invokeinterface) and negatively with calls that are statically dispatched (invokespecial and invokestatic). Moreover, other forms of dynamic type checks (checkcast, instanceof) also contribute negatively to this component. The fourth principal component again correlates (negatively) with various floating-point operations, but is otherwise hard to grasp

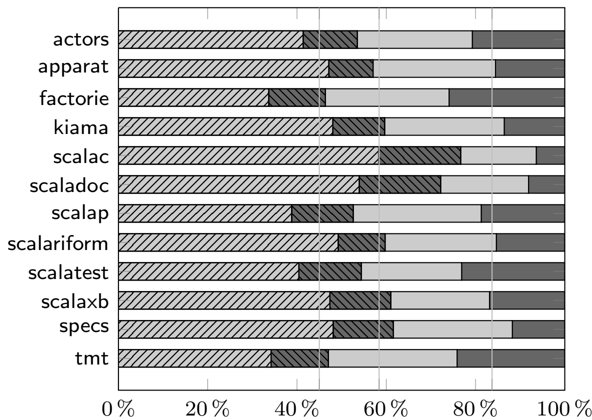

Figure 6a. The relative number of callsites using invokevirtual (▨), invokeinterface (▧), invokespecial (□), and invokestatic (■) instructions for the Scala benchmarks.

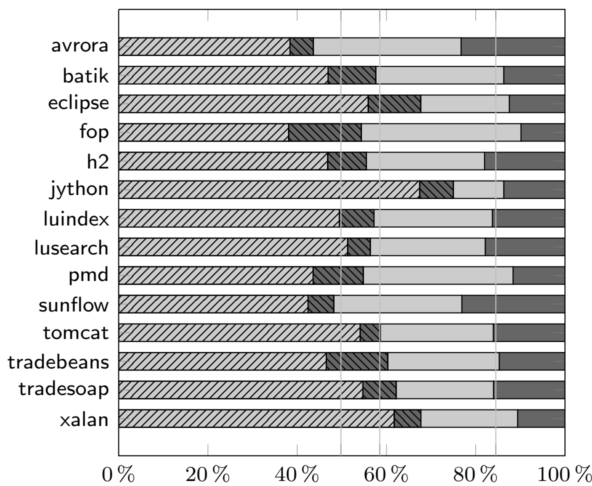

Figure 6b. The relative number of callsites using invokevirtual (▨), invokeinterface (▧), invokespecial (□), and invokestatic (■) instructions for the Java benchmarks.

intuitively. Still, as Figure 5 shows, it has significant discriminatory power, in particular with respect to the Scala benchmarks.

What is noteworthy in Figure 4 and particularly in Figure 5 is that benchmarks from the same application domain, e.g., factorie and tmt (machine-learning), nevertheless exhibit a different instruction mix. This justifies their joint inclusion into the suite.

4.2 Call Site Polymorphism

In object-oriented languages like Java or Scala, polymorphism plays an important role. For the JVM, however, call sites that potentially target different methods pose a challenge, as dynamic dispatch hinders method inlining, which is an important optimization. At the level of Java bytecode, such dynamically dispatched calls take the form of

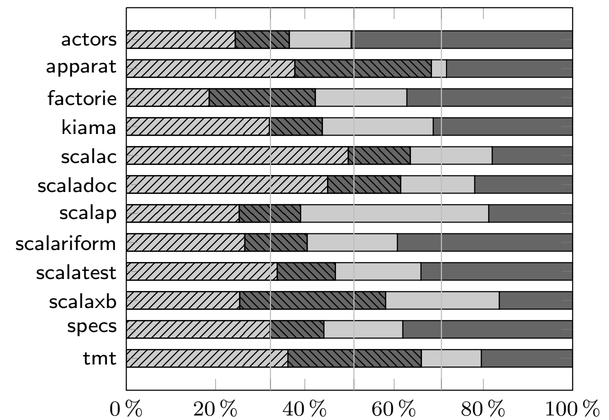

Figure 7a. The relative number of calls made using invokevirtual (▨), invokeinterface (▧), invokespecial (□), and invokestatic (■) instructions for the Scala benchmarks.

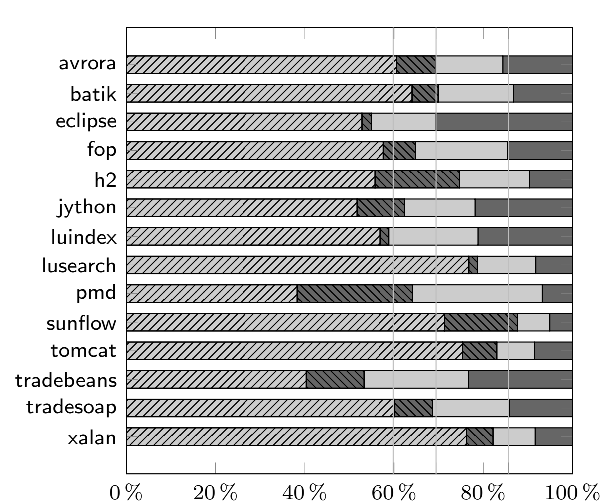

Figure 7b. The relative number of calls made using invokevirtual (▨), invokeinterface (▧), invokespecial (□), and invokestatic (■) instructions for the Java benchmarks.

invokevirtual or invokeinterface instructions, whereas the invokespecial and invokestatic instructions are statically dispatched. Figures 6a and 6b contrast the relative occurrences of these instructions in the Scala and Java benchmarks.

Here, only instructions executed at least once have been counted; dormant code is ignored. The numbers are remarkably similar for the Scala and Java benchmarks, with the vertical bars denoting the respective arithmetic mean, with only a slight shift from invokevirtual to invokeinterface instructions due to the way the Scala compiler unties inheritance and subtyping [44, Chapter 2] to implement mixins.

But the numbers in these figures provide only a static view. At runtime, the instructions' actual execution frequencies may differ. Figures 7a and 7b thus show the relative number of actual calls made via the corresponding call sites.

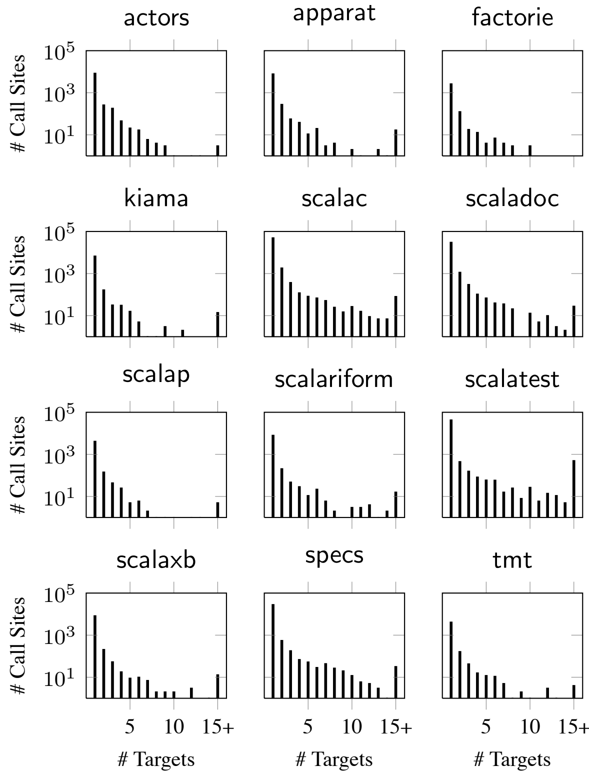

Figure 8a. The number of dynamically-dispatched call sites targeting a given number of methods for the Scala benchmarks.

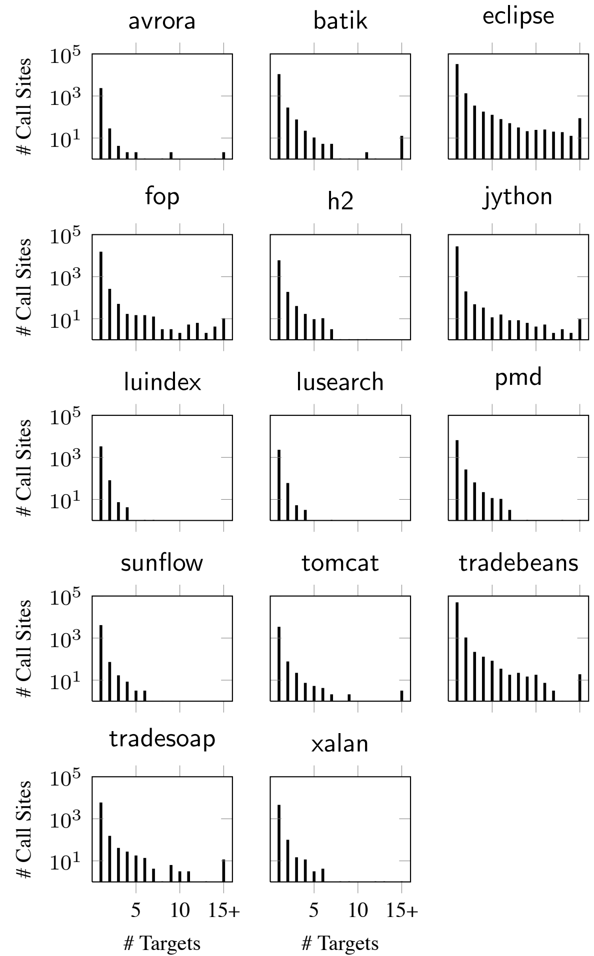

Figure 8b. The number of dynamically-dispatched call sites targeting a given number of methods for the Java benchmarks.

Here, the numbers for the Scala and Java benchmarks diverge, with the Scala benchmarks exhibiting a disproportionate amount of calls made by invokeinterface and invokestatic instructions. The former is again explained by the use of mixins, the latter by the use of singleton objects in general and by companion objects in particular. The divergence is even more pronounced, specifically with respect to calls made by invokeinterface, when comparing the numbers for Scala with results obtained for older Java benchmark suites [22]. In general, Scala code benefits more than Java code from techniques for the efficient execution of the invokeinterface instruction [1].

Polymorphic calls involve dynamic binding, since the target method may depend on the runtime type of the object receiving the call. In their analysis [19], Dufour et al. therefore distinguish between the number of target methods and the number of receiver types for polymorphic call sites, as there are typically more of the latter than of the former; not all subclasses override all methods.

Both dynamic metrics are relevant to different compiler optimizations [19]: The number of receiver types for polymorphic call sites is relevant for inline caching [25], an optimization technique commonly used in runtimes for dynamically-typed languages. Modern JVMs, however, rely on virtual method tables in combination with method inlining. We thus do not investigate the number of receiver types in this paper,[11] but focus on the number of target methods for polymorphic call sites. This information is essential for method inlining, one of the most effective compiler optimizations for the JVM. If a call site has only a single target method, the target method can be (speculatively) inlined. Moreover, even if a call site has more than one target, inlining is still possible with appropriate guards in place [15]. Only if the number of possible targets grows too large, inlining becomes infeasible.

Figures 8a and 8b (respectively 9a and 9b) show histograms of the polymorphic call sites, presenting the number of call sites (respectively the number of calls for call sites) with $x \geq 1$ target methods. These statistics comprise all exercised call sites corresponding to invokevirtual or invokeinterface instructions. Call sites that are never ex-

[11] The Scala compiler relies on a similar compilation technique using Java reflection and polymorphic inline caches for structural types [18]. However, this technique is not applied within the JVM itself.

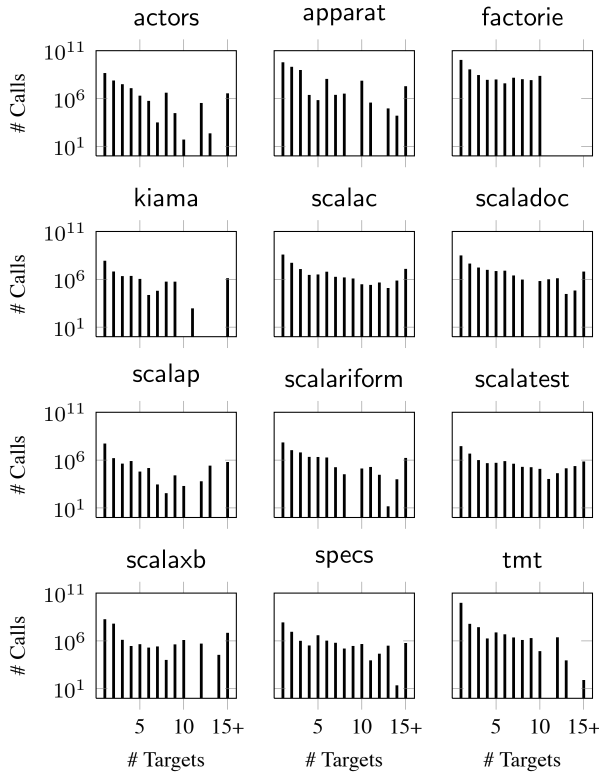

Figure 9a. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Scala benchmarks.

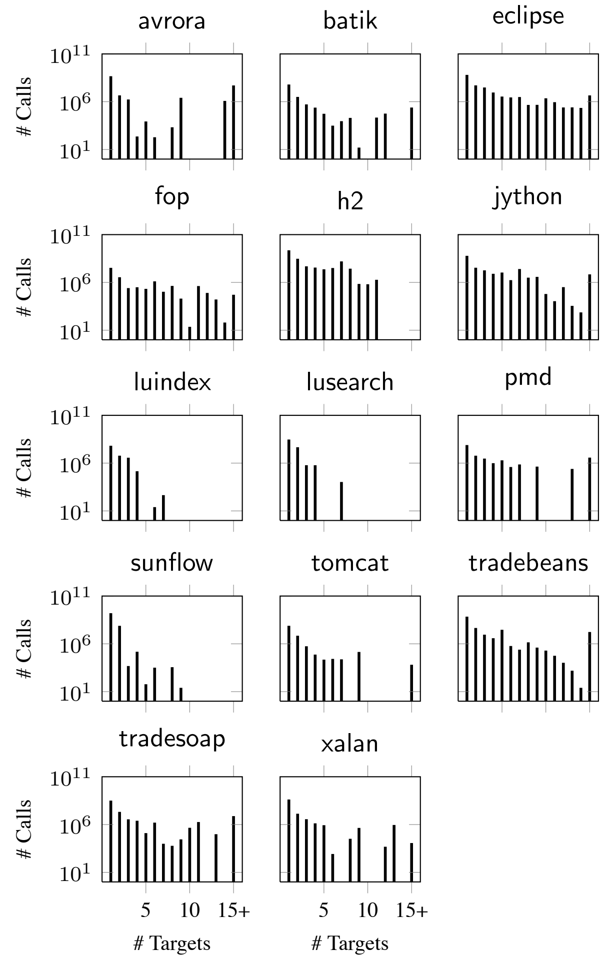

Figure 9b. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Java benchmarks.

ecuted by the workload are excluded; also, call sites corresponding to invokestatic and invokespecial bytecodes are excluded, as they are trivially monomorphic.

In Figures 8a and 8b, the actual number of invocations at a call site is not taken into account; call sites are merely counted. If a call site S, e.g., targets two methods M and N, whereby M is invoked m_S times and N is invoked n_S times, then call site S will be counted once for the bar at $x = 2$; the actual numbers m_S and n_S will be ignored. In contrast to Figures 8a and 8b, Figures 9a and 9b do take the actual number of invocations into account. That is, in the previous example, call site S contributes $m_S + n_S$ to the bar at $x = 2$. Figures 8a and 8b thus correspond to Figures 6a and 6b, whereas Figures 9a and 9b in turn correspond to Figures 7a and 7b. Whereas the former histograms show the possibility of inlining, the latter show its possible effectiveness.

As the analysis shows, there exist marked differences between the individual benchmarks, while the differences between the Scala and DaCapo benchmark suites are less pronounced. Still, dynamic dispatch with many different targets per call site used less by the Java benchmarks than by most Scala benchmarks except for factorie. However, megamorphic call sites with 15 or more targets are exercised almost all benchmarks in both the Scala and the DaCapo benchmark suites, i.e., by 11 out of 12 and 10 ouf of 14 benchmarks, respectively.

In general, polymorphism plays a larger role for the Scala benchmarks than for their Java counterparts, but this does not diminish the effectiveness of inlining significantly: Call sites with few targets in general and monomorphic call sites in particular account for a large part of potentially polymorphic method calls: For the Scala benchmarks, on average 97.1 % of callsites are monomorphic and they account for 89.7 % of the overall method calls. For the Java benchmarks on average 97.8 % of callsites are monomorphic and account for 91.5 % of calls.

The Scala benchmarks, however, show a higher variance (6.0 %) than the Java benchmarks (3.2 %) with respect to the number of monomorphic calls: The portion of such calls ranges from 76.4 % (apparat) to 99.2 % (tmt) rather than from 83.9 % (h2) to 96.0 % (xalan). With respect to this metric, our Scala benchmark suite is therefore at least as diverse as the DaCapo benchmark suite.

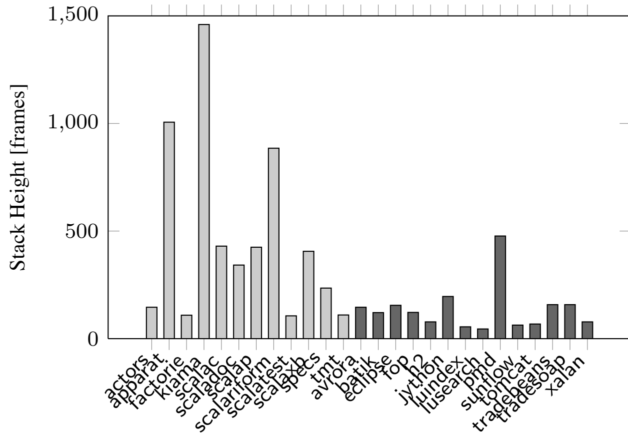

Figure 10. The maximum stack height required by the Scala (■) and Dacapo (□) benchmarks (default input size).

4.3 Stack Usage and Recursion

On the JVM, each method call creates a stack frame which, at least conceptually, holds the method's arguments and local variables, although some of them may be placed in registers instead (cf. Section 4.4). As most modern JVMs are unable to re-size a thread's call stack at runtime, they have to reserve a sufficiently large amount of memory whenever a new thread is created. If the newly-created thread does not require all the reserved space, memory is wasted; if it requires more space than was reserved, a StackOverflowException ensues.

As Figure 10 shows, the stack usage of the Scala benchmarks is significantly higher than for the Java benchmarks. However, for both benchmark suites the required stack size varies widely across benchmarks: For the Scala benchmarks, it ranges from 110 (tmt) to 1,460 frames (kiama), with an average of about 472. For the Java benchmarks, these numbers are more stable and significantly lower, ranging from 45 (lusearch) to 477 frames (pmd), with an average of 137.

The question is thus what gives rise to this significant increase in stack usage, the prime suspects being infrastructure methods inserted by the Scala compiler on the one hand and recursion on the other hand. To assess to what extent the programmer's use of the latter contributes to the benchmarks' stack usage, we make use of the calling context profiles collected by JP2. In the presence of polymorphism, however, a recursive method call is dynamically dispatched and may or may not target an implementation of the method identical to the caller's. This, e.g., often occurs when the Composite pattern is used.

Figures 11a and 11b, which depict the distribution of stack heights for each of the benchmarks, therefore use an extended definition of recursion. In this definition we distinguish between "true" recursive calls that target the same implementation of a method and plain recursive calls, which may also target a different implementation of the same method. Now, to compute the histograms of Figures 11a

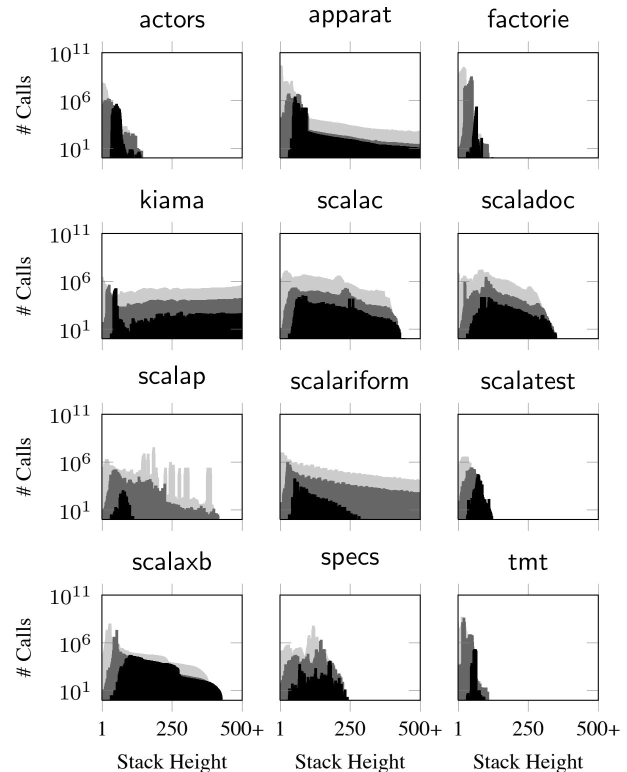

Figure 11a. The distribution of stack heights upon a method call for the Scala benchmarks: all method calls (□), recursive calls (■), and recursive calls to the same implementation (■).

and 11b, at every method call the current stack height x is noted. All method calls contribute to the corresponding light grey bar, whereas only recursive calls contribute to the dark grey bar, and only "true" recursive calls contribute to the black one.

The scalariform benchmark, e.g., makes no "true" recursive calls that target an implementation identical to the caller's at a stack height beyond 287, but still makes plenty of recursive calls targeting a different implementation of the selfsame method. In this case, the phenomenon is explained by the benchmark traversing a deeply-nested composite data structure. In general, this form of recursion is harder to optimize, as dynamic dispatch hinders, e.g., tail call elimination.

For all benchmarks, recursive calls indeed contribute significantly to the stack's growth, up to its respective maximum size, although for two Scala (scalap and scalariform) and one Java (pmd) benchmark this is not a direct consequence of "true" recursive calls, but of dynamically-dispatched ones. For all other benchmarks, however, there exists a stack height x at which all calls are "truly" recursive. This shows that, ultimately, it is the more extensive use of recursion by Scala programmers rather than of infrastructure methods by the Scala compiler that leads to the observed high stack usage.

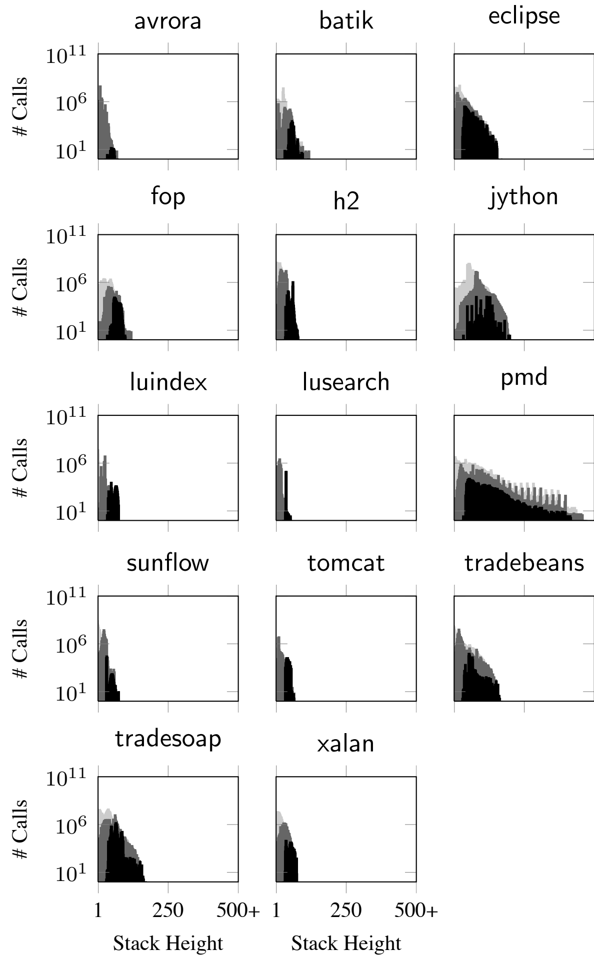

Figure 11b. The distribution of stack heights upon a call for the Java benchmarks: all method calls (▢), recursive calls (■), and recursive calls to the same implementation (■).

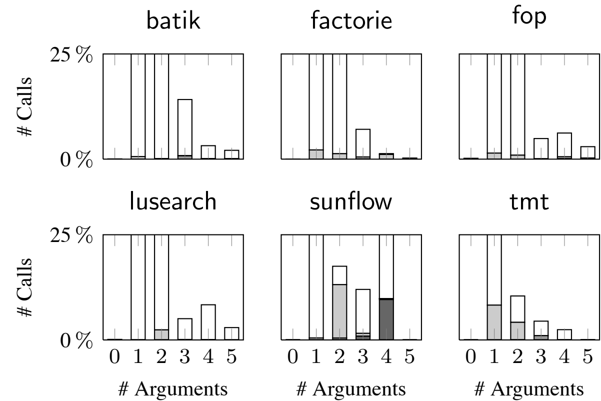

Figure 12. Distribution of the number of floating-point arguments passed upon a method call: none (□), 1 (▢), 2 (■), 3(■), 4 (■), and 5 or more (■).

4.4 Argument Passing

A method's stack frame not only stores a method's local variables, but it also contains the arguments passed to the method in question. Now both the number and kind of arguments passed upon a method call can have a performance impact: Large numbers of arguments lead to spilling on register-scarce architectures; they cannot be passed in registers alone. Different kinds of arguments may need to be passed differently; on many architectures, e.g., floating point numbers occupy a distinct set of registers.

As not all benchmarks in the DaCapo and Scala benchmark suites make much use of floating-point arithmetic, we first focus on the six benchmarks for which at least 1 % of method calls carries at least one floating-point argument: the Java benchmarks batik (1.9 %), fop (3.4 %), lusearch (2.6 %), and sunflow (25.1 %) and the Scala benchmarks factorie (5.1 %) and tmt (13.5 %).

The histograms in Figure 12 depict the number of floating-point arguments passed upon a method call, in relation to the overall number of arguments. The bars' shadings correspond to the number of floating-point arguments; the darker the shade, the more arguments are of either float or double type. As can be seen, not only do sunflow and tmt most frequently pass floating-point arguments to methods, but these two benchmarks are also the only ones where a noticeable portion of calls passes more than one floating-point argument: In the case of sunflow, four-argument methods are frequently called with three floating-point arguments, indicated by the dark portion of the respective bar. In the case of tmt, three-argument methods occasionally have two floating-point arguments (1.0 % of all calls).

The relation of floating-point and integral arguments is not the only dimension of interest with respect to argument passing: The histograms in Figures 13a and 13b thus depict the number of reference arguments passed upon an method call, in relation to the overall number of arguments, i.e., of primitive and reference arguments alike. Here the bars' shading corresponds to the number of reference arguments; the darker the shade, the more of the arguments are references rather than primitives.

Figures 13a and 13b distinguish between calls with an implicit receiver (invokevirtual, invokeinterface, and invokespecial) and calls without one (invokestatic). Both figures display the amount of arguments attributed to the former group above the "x-axis," whereas that attributed to the latter is displayed below the "x-axis." Taken together, the bars above and below the axis show the distribution of reference arguments for all calls to methods with x arguments. The receiver of a method call (if any) is hereby treated as a reference argument as well. This is in line with the treatment this receives from the virtual machine; it is simply placed "in local variable 0" [31].

For Scala and Java benchmarks alike, almost all methods have at least one argument, be it explicit or implicit, viz., this. This is in line with earlier findings on Java bench-

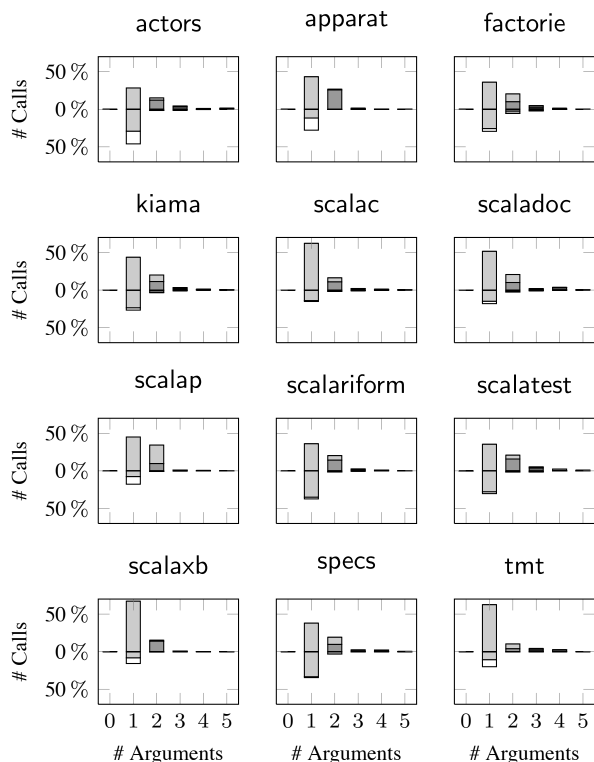

Figure 13a. Distribution of the number of reference arguments passed upon a method call by the Scala benchmarks: none (□), 1 (■), 2 (■), 3 (■), 4 (■), and 5 or more (■).

marks [14]. But while the maximum number of passed arguments can be as large as 21 for Scala code (specs) and 35 for Java code (tradebeans, tradesoap), on average only very few arguments are passed upon a call: 1.04–1.47 for Scala code and 1.69–2.43 for Java code. In particular, the vast majority of methods called by the Scala benchmarks has no arguments other than the receiver; they are simple "getters." This has an effect on the economics of method inlining: the direct benefit of inlining, i.e., the removal of the actual call, clearly outweighs the possible indirect benefits, i.e., the propagation of information about the arguments' types and values which, in turn, facilitates, e.g., constant folding.

This marked difference between Scala and Java benchmarks is of particular interest, as the Scala language offers special constructs, namely implicit parameters and default values, to make methods with many arguments less inconvenient to the programmer. But while methods taking many parameters do exist, they are rarely called.

4.5 Method and Basic Block Hotness

Any modern JVM with a just-in-time compiler adaptively optimizes application code, focusing its efforts on those parts that are "hot," i.e., executed frequently. Regardless of whether the virtual machine follows a traditional, region-based [24, 48], or trace-based [3] approach to compilation,

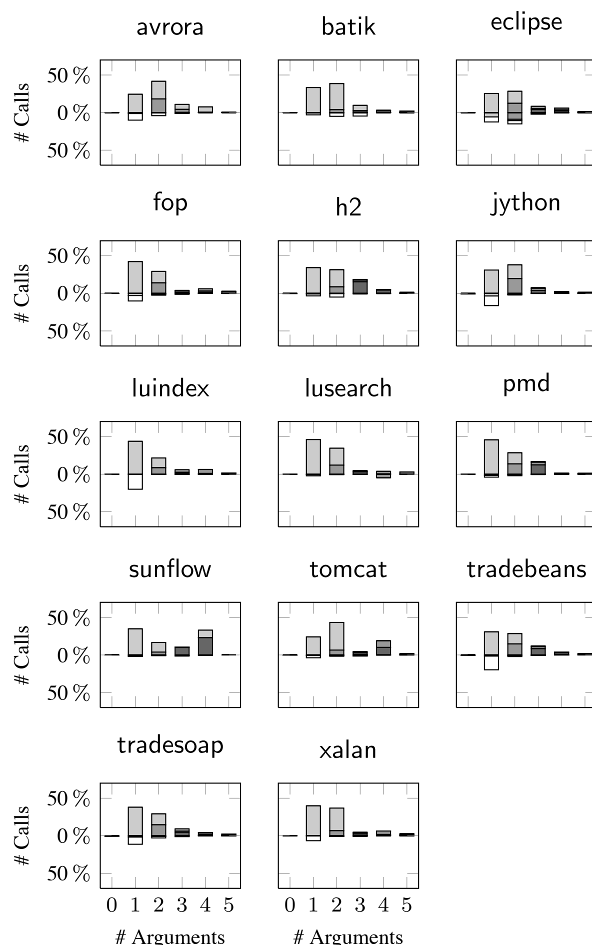

Figure 13b. Distribution of the number of reference arguments passed upon a method call by the Java benchmarks: none (□), 1 (■), 2 (■), 3 (■), 4 (■), and 5 or more (■).

pronounced hotspots are fundamental to the effectiveness of adaptive optimization efforts. It is thus of interest to which extent the different benchmarks exhibit such hotspots.

In contrast to Dufour et al., who report only "the number of bytecode instructions responsible for 90 % execution" [19], our metric is continuous. Figures 14a and 14b report to which extent the top 20 % of all static bytecode instructions in the code contribute to the overall dynamic bytecode execution. A value of 100 % on the x-axis corresponds to all instructions contained in methods invoked at least once; dormant methods are excluded. However, basic blocks that are either dead or simply dormant during the benchmark's execution in an otherwise live method are taken into account, as they still would need to be compiled using a traditional approach. A value of 100 % on the y-axis simply corresponds to the total number of all executed bytecodes.

Current JVMs typically optimize at the granularity of methods rather than basic blocks and are thus often unable to optimize just the most frequently executed instructions or

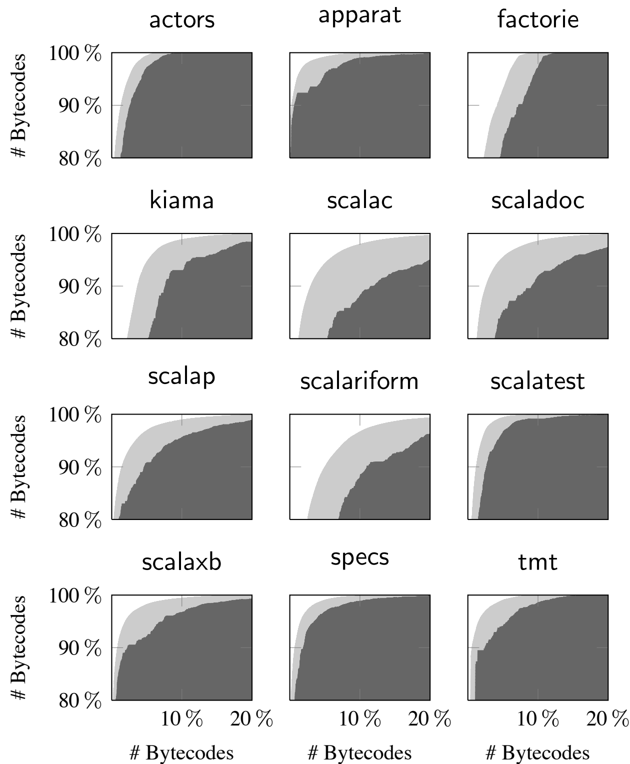

Figure 14a. Cumulative number of executed bytecodes for the most frequently executed bytecodes when measured at the granularity of basic blocks (■) or methods (■).

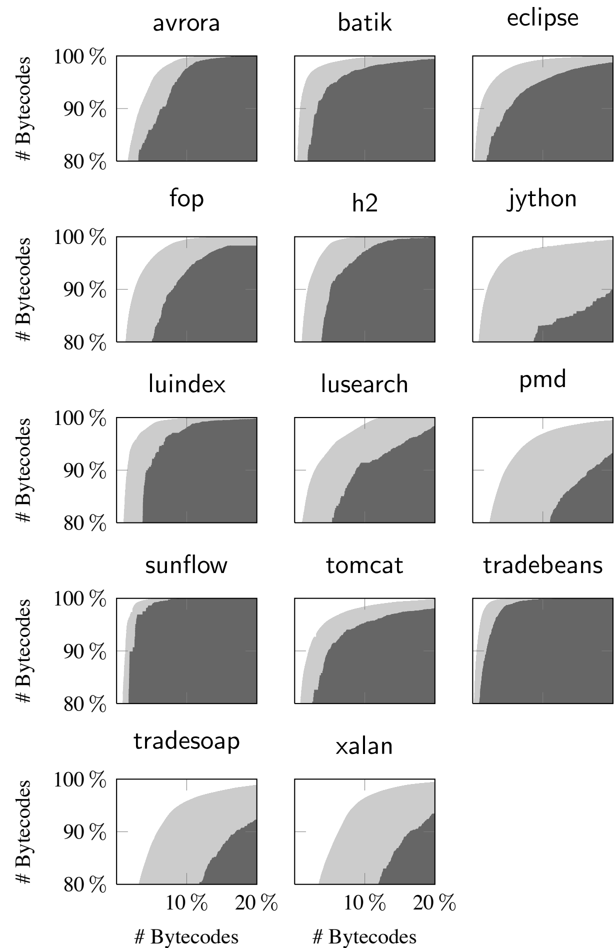

Figure 14b. Cumulative number of executed bytecodes for the most frequently executed bytecodes when measured at the granularity of basic blocks (■) or methods (■).

basic blocks. To reflect this, Figures 14a and 14b also report the extent to which the hottest methods are responsible for the execution. The two data series are derived as follows:

Basic block hotness. The basic blocks (in methods executed at least once) are sorted in descending order of their execution frequencies. Each basic block b_i is then plotted at $x = \sum_{j=1}^{i} \text{size}(b_j)$ and $y = \sum_{j=1}^{i} \text{size}(b_j) \cdot \text{freq}(b_j)$, where $\text{size}(b_j)$ is the number of bytecodes in b_j and $\text{freq}(b_j)$ is the number of times b_j has been executed.

Method hotness. The methods (that are executed at least once) are sorted in descending order of the overall number of bytecodes executed in each method. Each method m_i is then plotted at $x = \sum_{j=1}^{i} \sum_{b \in B_j} \text{length}(b)$ and $y = \sum_{j=1}^{i} \sum_{b \in B_j} \text{length}(b) \cdot \text{freq}(b)$, where B_j are the basic blocks of method m_j.

For the actors and scalap benchmarks, e.g., only 1.4 % and 1.5 % of all bytecode instructions, respectively, are responsible for 90 % of all executed bytecodes. However, beyond that point, the basic block hotness of the two benchmarks' differs considerably. Moreover, the method hotness of these two benchmarks is also different, the discrepancy between basic block and method hotness being much larger for scalap than for actors: A method-based compiler would need to compile just 2.7 % of actor's bytecodes to cover 90 % of all executed instructions, whereas 4.7 % of all instructions need to be compiled for scalap to achieve the same coverage.

In general, the discrepancy between basic block and method hotness is quite pronounced. This is the effect of methods that contain both hot and cold basic blocks, i.e., some that are frequently executed and some that are not. Four Java benchmarks (jython, pmd, tradesoap, and xalan) with a larger than average number of basic blocks per method suffer most from this problem. The remaining Java benchmarks exhibit patterns similar to the Scala benchmarks. Both region-based [24, 48] and trace-based compilation [3] can be employed to lessen the effect of such temperature drops within methods.

4.6 Reflection

While reflective features that are purely informational, e.g., runtime-type information, do not pose implementation challenges, other introspective features like reflective invoca-

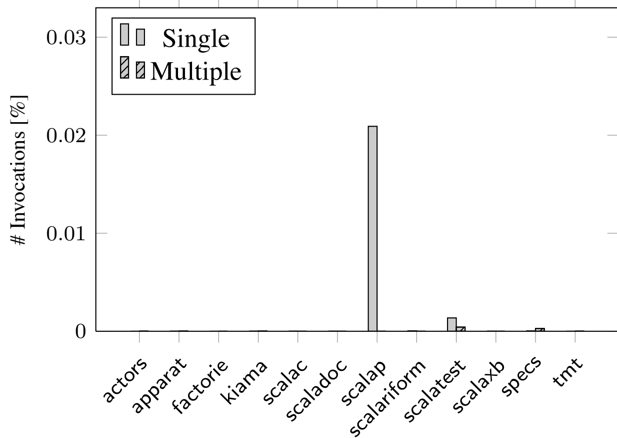

Figure 15a. Number of methods invoked reflectively by the Scala benchmarks with a single or multiple Method instances per call site, normalized to the number of all method invocations.

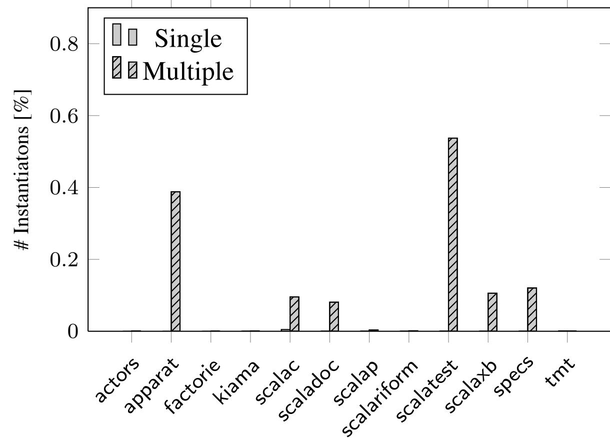

Figure 16a. Number of objects instantiated reflectively by the Scala benchmarks with a single or multiple Class instances per call site, normalized to the number of all object instantiations (new).

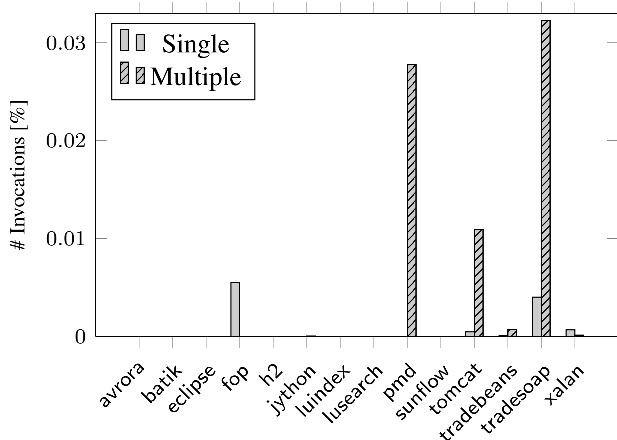

Figure 15b. Number of methods invoked reflectively by the Java benchmarks with a single or multiple Method instances per call site, normalized to the number of all method invocations.

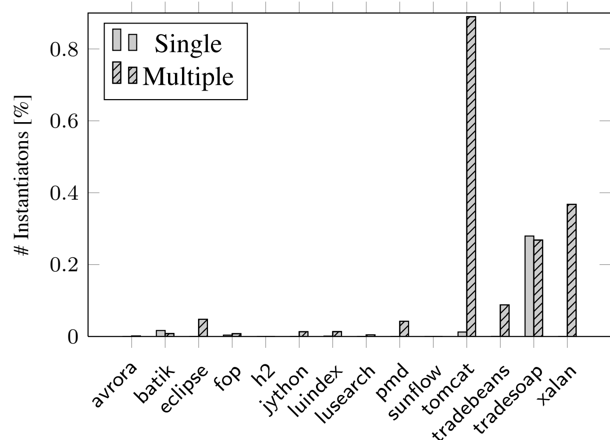

Figure 16b. Number of objects instantiated reflectively by the Java benchmarks with a single or multiple Class instances per call site, normalized to the number of all object instantiations (new).

tions or instantiations are harder to implement efficiently by a JVM [40]. It is thus of interest to what extent Scala code makes use of such features, in particular as Scala's structural types are compiled using a reflective technique [18]. We have thus extended TamiFlex [8] to gather information about the following three usages of reflection: method calls (Method.invoke), object allocation (Class.newInstance, Constructor.invoke, and Array.newInstance), and field accesses (Field.get, Field.set, etc.).

We first consider reflective invocations. What is of interest here is not only how often such calls are made, but also whether a call site exhibits the same behaviour throughout. If a call site for Method.invoke, e.g., consistently refers to the same Method instance, partial evaluation [9] might avoid the reflective call altogether. For the Scala and DaCapo benchmark suites, respectively, Figures 15a and 15b depict the number of reflective method invocations with a single or multiple Method instances per call site. These numbers have been normalized with respect to the number of overall method calls (invokevirtual–invokeinterface bytecode instructions). As can be seen, few benchmarks in either suite perform a significant number of reflective method invocations. Even for those benchmarks (scalap, pmd, tomcat, and tradesoap), reflective invocations account for at most 0.03 % of invocations. In the case of scalap, these invocations are almost exclusively due to the use of structural types within the benchmark. This also explains why only a single Method instance is involved [18]. Should the use of reflec-

tion to implement structural types be supplanted by the use of invokedynamic, these metrics remain meaningful, as similar caching techniques can be applied as an optimization.

We next consider reflective object instantiations, i.e., calls to Class.newInstance, Constructor.invoke, as well as Array.newInstance. Again, for reasons explained above, we distinguish between call sites referring to a single meta-object and call sites referring to several.[12] Figures 16a and 16b depict the number of reflective instantiations for the Scala and DaCapo benchmark suites, respectively. The numbers have been normalized with respect to the number of overall allocations (new bytecode instructions).

The surprisingly large number of reflective instantiations by several Scala benchmarks can be attributed to the creation of arrays via scala.reflect.ClassManifest, where a single call site instantiates numerous arrays of different type. This is an artifact of Scala's translation strategy, as one cannot express the creation of generic arrays in Java bytecode without resorting to reflection.

Unlike reflective invocations and instantiations, reflective field accesses are almost absent from both Scala and Java benchmarks. The only notable exception is eclipse, which uses reflection to write to a few hundred fields.

4.7 Boxed Types

Boxing of primitive types like int or double may incur significant overhead; not only is an otherwise superfluous object created, but simple operations like addition now require prior unboxing of the boxed value. We have therefore measured to which degree the Java and Scala benchmarks create boxed values. To this end, we distinguish between the mere request to create a boxed value by using the appropriate valueOf factory method and the actual creation of a new instance using the boxed type's constructor; usage of the factory method allows for caching but may impede JIT compiler optimizations [11].

Figures 17a and 17b show how many boxed values are requested and how many are actually created. The counts have been normalized with respect to the number of all object allocations (new bytecode instructions). Note that we have not counted boxed values created from strings instead of unboxed values, as the intent here is rather different, namely to parse a string. While for the Java benchmarks (Figure 17b), boxing accounts only for very few object creations, this is not true for many of the Scala benchmarks (Figure 17a); almost all of the objects created by, e.g., the tmt benchmark are boxed primitives.

In general, the caching performed by the factory methods is effective. Only for factorie and tmt, a significant number of requests (calls to valueOf) to box a value result in an actual object creation; this is due to the fact these two benchmarks operate with floating-point values, for which

[12] We treat the Class instance passed to Array.newInstance as the meta-object here.

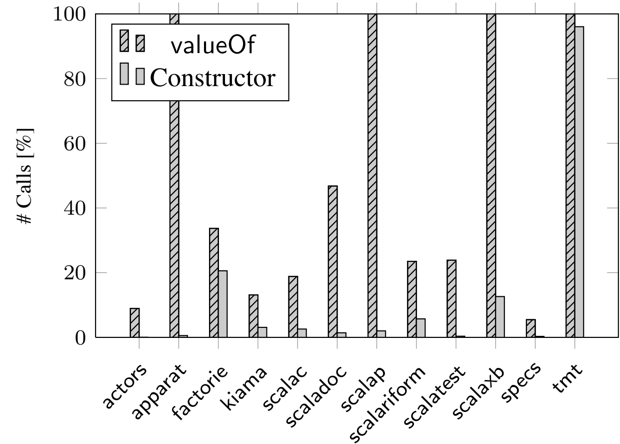

Figure 17a. Boxed instances requested (valueOf) and created (Constructor) by the Scala benchmarks, normalized to the number of all object allocations (new).

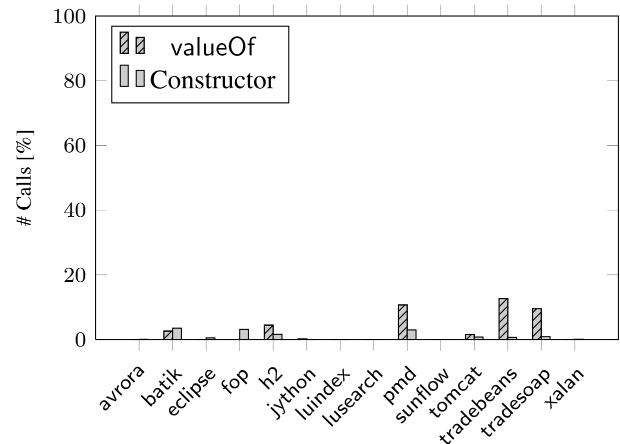

Figure 17b. Boxed instances requested (valueOf) and created (Constructor) by the Java benchmarks, normalized to the number of all object allocations (new).

the corresponding valueOf factory methods do not perform caching. That being said, the Scala benchmarks in general both create and request more boxed values than their Java counterparts. Extensive use of user-directed type specialization [17] may be able to decrease these numbers, though.

Another interesting fact about the usage of boxed values is that they are created at a few dozen sites only within the program, the fop Java benchmark being the only exception with 4,547 call sites of a boxing constructor.

5. Related Work

Blackburn et al. [7] created the DaCapo benchmark suite to improve upon the state-of-the-art of Java benchmark suites at that time. Their selection of metrics puts a strong emphasis on the benchmarks' memory and pointer behaviours, whereas the present paper concentrates on the difference

between Scala and Java code, leaving a detailed analysis of memory and pointer behaviour to future work (cf. Section 7). While the creators of the DaCapo benchmark suite also report code-related metrics, only the static metrics reported, e.g., the number of loaded classes or the coupling between them, are JVM-independent; the reported dynamic metrics like instruction mix (cf. Section 4.1) or method hotness (cf. Section 4.5) have been measured in a JVM-dependent fashion. Like others [26], Blackburn et al. use principal component analysis to demonstrate the benchmarks' diversity.

Dufour et al. [19] computed a range of VM-independent metrics for a selection of Java workloads. Unlike in the present paper, the authors' primary objects of study are the metrics, not the benchmarks. While Dufour et al. were trying to assess the usefulness of the various metrics in distinguishing the benchmarks, we use metrics to distinguish between the benchmarks themselves, in particular with respect to the Scala / Java code dichotomy.

Shiv et al. [46] analyzed the SPECjvm2008 benchmark suite both qualitatively and quantitatively. The majority of their quantitative evaluation, however, is based on JVM- and architecture-dependent metrics. The authors furthermore offer a brief comparison of the SPECjvm2008 and SPECjvm98 benchmark suites.

Daly et al. [14] conducted an analysis of the Java Grande benchmark suite [10] using VM-independent metrics. Their analysis focuses on the benchmark's static and dynamic instruction mix and considers the 201 instructions both individually and by manual assignment to one of 22 groups. In contrast to Daly et al., we apply principal component analysis to automatically extract groupings, which we then use to show that there a differences not only between individual benchmarks, but also between the Scala and Java benchmarks as a whole.

The number of actual invocations of native methods is a dynamic metric that can be obtained by incrementing a counter at runtime. Gregg et al. [22] used an instrumented version of the Kaffe virtual machine in order to gather this metric. Thus, their approach is not portable and provides only a very coarse-grained view of where CPU time is actually spent. Some researchers provided a more detailed breakdown of where CPU time is spent in Java workloads [23, 30]; however, they likewise had to sacrifice portability by directly modifying a JVM.

Hoste and Eeckhout [26] used principal component analysis to show that workload characterization is most effectively done in a microarchitecture-independent fashion; instead of relying on a particular microarchitecture, metrics should instead be defined with respect to an idealized one. In the present paper, we apply this tenet by using metrics defined with respect to Java bytecode, it being arguably the natural microarchitecture of JVMs.

To assess the performance impact of run-time types, Schinz [44, Chapter 6] performed some experiments with earlier (circa 2005) versions of two Scala applications that are part of our benchmark suite: scalac[13] and scalap. These experiments, however, were limited to evaluating different translation strategies for run-time types.

Recent work by Richards et al. [39] and Ratanaworabhan et al. [38] applied an approach similar to ours to analyze the dynamic behaviour of JavaScript; the authors use platform-independent metrics to characterize, e.g., the instruction mix, call site polymorphism, and method hotness. The key difference, besides the considered languages, is that our work resulted in a full-fledged benchmark suite researchers can re-use. Only Ratanaworabhan et al. compare their findings with established JavaScript benchmarks suites.

Jibaja et al. [27] recently compared the memory behaviour of two managed languages, namely PHP and Java, to guide language implementers. From the significant similarities found between benchmarks of the PHP and SPECjvm98 benchmark suites, the authors infer that PHP would benefit from garbage collector designs similar to those successful in modern JVMs.

6. Summary and Conclusions

The Java Virtual Machine is no longer targeted by the Java language alone but by a variety of programming languages. The benchmark suites used within the JVM research community, however, do not yet reflect this trend. The benchmark suite presented in this paper addresses this issue for the Scala language. They have been shown to a offer a varied set of workloads from a broad range of application domains.

To summarize, our analysis of the Scala benchmark suite led to the following findings, which may be of interest to both the developers of JVMs and to the developers of the Scala compiler and its associated libraries.

Instruction Mix. Scala and Java programs differ significantly in their instruction mix.

Call Site Polymorphism. Although polymorphism plays a larger role for Scala than for Java code, the overwhelming number of callsites is effectively monomorphic and accounts for the majority of calls. Inlining is thus expected to be as effective for Scala code as its is for Java code.

Stack Usage and Recursion. Scala applications require significantly more space on the call stack than their Java counterparts. Recursive method calls to varying target implementations contribute significantly to this.

Argument Passing. The vast majority of method calls in Scala code target parameter-less "getters;" methods with more than one arguments are rarely called. This negatively affects the economics of method inlining, as the

[13] At that time, it was still called nsc, the New Scala Compiler.

optimization propagates less information into the inlined method.

Method and Basic Block Hotness. Hotspots in Scala and Java code are similarly distributed. However, Scala code seems to be slightly easier for method-based compilation to cope with.

Reflection. Although the Scala compiler resorts to using reflection to translate structural types, reflective invocations are not a significant performance bottleneck in Scala applications. Scala is thus much less likely to benefits from the invokedynamic instruction than dynamically-typed languages like JRuby or Jython.

Boxed Types. While the Scala compiler already tries to avoid boxing, Scala programs nevertheless request and create significantly more boxed values than Java programs. Therefore, canonicalization or similar optimizations are crucial.

Our evaluation suggests that the execution characteristics of Scala code do differ from Java code in several ways. However, this does not invalidate optimizations performed by current JVMs; instead, it suggests that fine-tuning may be sufficiently profitable.

7. Future Work

No benchmark suite stays relevant forever. Like Blackburn et al. have done for the DaCapo benchmark suite [7], we plan to maintain the benchmark suite, incorporate community feedback, and extend the suite to cover further application domains once suitable Scala applications emerge. We will also maintain and extend the toolchain we used to build the Scala benchmark suite (cf. Appendix A), so that it becomes easier for other researchers to build their own benchmarks, possibly for further languages beyond Java or Scala.

Over the years, many researchers have investigated the memory behaviour of Java programs on a range of benchmark suites and with a variety of methods [7, 16, 28]. In the present paper, we concentrate on the structure and behaviour of the code itself. However, we do plan to investigate where the differences are between Scala and Java code with respect to memory behaviour in the future.

While the Scala library offers dedicated support for the actor-model of concurrency, a surprisingly small number of benchmarks (actors, apparat, scalac, scaladoc, scalatest, tmt) is multi-threaded. A detailed analysis of the execution characteristics of concurrent Scala program on the JVM requires further research. The actors benchmark contained in our suite, however, with its usage of different actor implementations, may prove a good starting point.

The Scala distribution also supports the Microsoft .NET platform.[14] For practical reasons, however, we have restricted our analysis to a single platform, namely the JVM. A detailed analysis of Scala's execution characteristics on the .NET platform is beyond the scope of the present paper, but subject to future work.

[14] See http://www.scala-lang.org/node/168.

A. Building a Benchmark Suite

The entire benchmark suite is built using Apache Maven,[15] a build management tool whose basic tenet rings particularly true in the context of a research benchmark suite: build reproducibility. The use of a central artifact repository mirrored many times worldwide, in particular, ensures that the benchmark suite can be built reproducibly in the future.

We have developed a dedicated Maven plugin that not only packages a benchmark according to the DaCapo suite's requirements but also performs a series of integration tests on the newly-built benchmark. It furthermore retrieves but keeps separate all transitive dependencies of the benchmark and its harness. Just as the benchmark suite, the Maven plugin is Open Source and freely available for download.

B. Collecting Dynamic Metrics with JP2

The measurements presented in this paper have been obtained with our open-source profiler JP2[16] [42, 43], which produces a calling context tree (CCT) [2] representing overall program execution. JP2 is a reimplementation and extension of the profiler JP [4, 6, 33]. It produces a *complete* CCT that distinguishes between different *callsites* and counts how often each *basic block* is executed in each context. JP2 has been designed for *compatibility* with standard JVMs and supports a variety of *output formats* for the generated profile. In the following text, we briefly summarize the key properties, including the remaining *limitations*, of JP2.

Completeness of the CCT. We consider the CCT to be complete if it represents, after an initial JVM bootstrapping phase, every method call where either the caller or the callee has a bytecode representation. The JVM bootstrapping phase finishes before the program's main method is invoked. This definition ensures that invocations of native methods by bytecode and callbacks from native code into bytecode are also included in the CCT. Furthermore, implicit method invocations, such as for class loading and class initialization, must be properly represented.

Distinguishing between Callsites in the CCT. Some calling context profilers like JP do not distinguish between individual callsites within a method; if two callsites in a method invoke the same target method, these invocations will be indistinguishable in the CCT. However, if callsites cannot be distinguished, several useful metrics like call site polymorphism (cf. Section 4.2) cannot be computed from the resulting profiles. JP2 solves this issue by associating each CCT node with two keys, a unique identifier of the target method and an identifier of the callsite within the calling method.

[15] See http://maven.apache.org/.

[16] See http://jp-profiler.origo.ethz.ch/.

Counting the Execution of Basic Blocks of Code. While JP only counts the number of executed bytecodes for each calling context, the latest version of JP2 keeps a separate counter for each basic block of code in a method. That is, each CCT node stores an array of counters, one for each basic block in the corresponding method. Together with the CCT itself and the loaded classfiles, these counts are sufficient to compute a variety of dynamic metrics.

For the purpose of this paper, only bytecodes that may non-sequentially change the control flow (e.g., goto, ifeq, return, athrow, etc.) end a basic block. In particular, method invocations as well as instructions which may throw exceptions implicitly (e.g., aastore) do not end a basic block. We have shown in previous work that this has little to no impact on the profiles' accuracy [6].

Compatibility with Standard JVMs. JP2 is implemented with ASM,[17] a light-weight bytecode manipulation framework. The profiler keeps structural modifications of classfiles to a minimum; it only modifies method bodies and inserts entries in the classfile constant pool. These transformations are neither visible through the Java reflection API nor do they interfere with stack introspection. Only a few instance fields are inserted into java.lang.Thread to reduce the overhead of thread-local variables used by JP2.

JP2 relies on native method prefixing, a JVMTI feature introduced with Java 6, in order to profile the invocation of native methods. Native methods are renamed (by adding a prefix) and wrapped with Java methods that will invoke the corresponding prefixed native methods. Although these transformations change the classfile structure, the Java class library and the JVM have been designed to properly deal with this issue (e.g., in the code for stack introspection). However, in some JVM releases, there are still a few native methods that cannot be prefixed.

Output formats. JP2 offers several plugins to serialize the CCT upon program termination. The plugin used for the measurements presented in this paper stores the CCT in an XML format. Together with the stored classfiles, converted to XML by ASM, we were able to compute all dynamic metrics using XQuery by cross-referencing between classfiles and CCT [43].

Limitations. JP2 currently does not distinguish between different classloaders. That is, if a polymorphic callsite invokes two different target methods with the same name and signature that are defined in distinct classes bearing the same name but defined by distinct classloaders, the two targets will be represented by the same CCT node. Such a situation may yield corrupt profiles, but was not encountered for any of the evaluated benchmarks.

Depending on the JVM used, prefixing may not work with a few native methods (e.g., registerNatives). Invocations of these methods will not be present in the CCT.

[17] See `http://asm.ow2.org/`.

Acknowledgments

We are deeply grateful to "e.e d3si9n," Joa Ebert, Patrik Nordwall, Daniel Ramage, Bill Venners, Tim Vieira, and the late Sam Roweis for their assistance with the various programs and input data that ultimately made it into our benchmark suite. We are also grateful to the DaCapo group for building such an excellent foundation on which we could build our benchmark suite.

We would furthermore like to thank both the participants of and the reviewers for the Work-on-Progress session at the 8th International Conference on the Principles and Practice of Programming in Java for suggestions and encouragement. Eric Bodden, Michael Eichberg, Ivan Jibaja, Kathryn McKinley, Nate Nystrom, and Jan Sinschek provided valuable comments on the contents of this paper.

This work was supported by the Center for Advanced Security Research Darmstadt (www.cased.de) and the Swiss National Science Foundation.

References

[1] B. Alpern, A. Cocchi, S. Fink, and D. Grove. Efficient implementation of Java interfaces: invokeinterface considered harmless. In *Proceedings of the 16th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the 10th Conference on Programming Language Design and Implementation (PLDI)*, 1997.

[3] M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2010.

[4] W. Binder. A portable and customizable profiling framework for Java based on bytecode instruction counting. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, 2005.

[5] W. Binder, J. Hulaas, and P. Moret. A quantitative evaluation of the contribution of native code to Java workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2006.

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

[7] 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 (OOPSLA)*, 2006.

[8] E. Bodden, A. Sewe, J. Sinschek, M. Mezini, and H. Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, 2011.

[9] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999.

[10] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings of the ACM 1999 Conference on Java Grande*, 1999.

[11] Y. Chiba. Redundant boxing elimination by a dynamic compiler for Java. In *Proceedings of the 5th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2007.

[12] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994. ISSN 0098-5589.

[13] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, May 2007. ISSN 0038-0644.

[14] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande Forum benchmark suite. In *Proceedings of the 2001 joint ACM-ISCOPE Conference on Java Grande*, 2001.

[15] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-oriented Programming (ECOOP)*, 1999.

[16] S. Dieckmann and U. Hölzle. A study of the allocation behavior of the SPECjvm98 java benchmarks. In *Proceedings of the 13th European Conference on Object-oriented Programming (ECOOP)*, 1999.

[17] I. Dragos and M. Odersky. Compiling generics through user-directed type specialization. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-oriented Languages and Programming Systems (ICOOOLPS)*, 2009.

[18] G. Dubochet and M. Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-oriented Languages and Programming Systems (ICOOOLPS)*, 2009.

[19] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2003.

[20] *Common Language Infrastructure (CLI)*. ECMA International, 5th edition, December 2010.

[21] L. Eeckhout, A. Georges, and K. De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th Conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2003.

[22] D. Gregg, J. Power, and J. Waldron. A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 17: 757–773, 2005.

[23] N. M. Hanish and W. E. Cohen. Hardware support for profiling Java programs. In *Proceedings of the Workshop on Hardware Support for Objects and Microarchitectures for Java*, 1999.

[24] R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th International Symposium on Microarchitecture (MICRO)*, 1995.

[25] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-oriented Programming (ECOOP)*, 1991.

[26] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27:63–72, 2007.

[27] I. Jibaja, S. Blackburn, M. Haghighat, and K. McKinley. Deferred gratification: Engineering for high performance garbage collection from the get go. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC)*, 2011.

[28] R. E. Jones and C. Ryder. A study of Java object demographics. In *Proceedings of the 7th International Symposium on Memory Management (ISMM)*, 2008.

[29] R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2009.

[30] G. Lashari and S. Srinivas. Characterizing Java application performance. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003.

[31] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[32] A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. *Advances on Neural Information Processing Systems*, 2009.

[33] P. Moret, W. Binder, and A. Villazón. CCCP: Complete calling context profiling in virtual execution environments. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2009.

[34] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, 2nd edition, 2010.

[35] K. Ogata, D. Mikurube, K. Kawachiya, S. Trent, and T. Onodera. A study of Java's non-Java memory. In *Proceedings of the 25th Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010.

[36] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2:559–572, 1901.

[37] D. Ramage, E. Rosen, J. Chuang, C. D. Manning, and D. A. McFarland. Topic modeling for the social sciences. In *NIPS*

Workshop on Applications for Topic Models: Text and Beyond, 2009.

[38] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development*, 2010.

[39] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[40] I. Rogers, J. Zhao, and I. Watson. Approaches to reflective method invocation. In *Proceedings of the 3rd Workshop on Implementation, Compilation, Optimization of Object-oriented Languages, Programs and Systems (ICOOOLPS)*, 2008.

[41] J. R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2009.

[42] A. Sarimbekov, P. Moret, W. Binder, A. Sewe, and M. Mezini. Complete and platform-independent calling context profiling for the Java virtual machine. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, 2011.

[43] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schöberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the java virtual machine. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011.

[44] M. Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, EPFL, Switzerland, September 2005.

[45] A. Sewe. $\mathrm{Scala} \stackrel{?}{\equiv} \mathrm{Java} \bmod \mathrm{JVM}$. In *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, volume 692 of *CEUR Workshop Proceedings*, 2010.

[46] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings of the SPEC Benchmark Workshop*, 2009.

[47] A. M. Sloane. Experiences with domain-specific language embedding in Scala. In *Proceedings of the 2nd International Workshop on Domain-Specific Program Development (DSPD)*, 2008.

[48] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the 16th Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[49] C. Thalinger and J. Rose. Optimizing invokedynamic. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2010.