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

**Entwurf und Analyse einer Scala Benchmark Suite für die Java Virtual
Machine**

Zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)
genehmigte Dissertation von Diplom-Mathematiker Andreas Sewe aus
Twistringen, Deutschland
April 2013 – Darmstadt – D 17



TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine
Entwurf und Analyse einer Scala Benchmark Suite für die Java Virtual Machine

Genehmigte Dissertation von Diplom-Mathematiker Andreas Sewe aus Twistringen, Deutschland

1. Gutachten: Prof. Dr.-Ing. Ermira Mezini
2. Gutachten: Prof. Richard E. Jones

Tag der Einreichung: 17. August 2012

Tag der Prüfung: 29. Oktober 2012

Darmstadt – D 17



For Bettina





Academic Résumé

November 2007 – October 2012 Doctoral studies at the chair of Prof. Dr.-Ing. Ermira Mezini, Fachgebiet Softwaretechnik, Fachbereich Informatik, Technische Universität Darmstadt

October 2001 – October 2007 Studies in mathematics with a special focus on computer science (*Mathematik mit Schwerpunkt Informatik*) at Technische Universität Darmstadt, finishing with a degree of Diplom-Mathematiker (Dipl.-Math.)



Acknowledgements

First and foremost, I would like to thank Mira Mezini, my thesis supervisor, for providing me with the opportunity and freedom to pursue my research, as condensed into the thesis you now hold in your hands. Her experience and her insights did much to improve my research as did her invaluable ability to ask the right questions at the right time. I would also like to thank Richard Jones for taking the time to act as secondary reviewer of this thesis. Both their efforts are greatly appreciated.

Time and again, I am astonished by the number of collaborators and co-authors whom I have worked with during these past five years: Mehmet Akşit, Sami Al-souri, Danilo Ansaloni, Remko Bijker, Walter Binder, Christoph Bockisch, Eric Bodden, Anis Charfi, Michael Eichberg, Samuel Z. Guyer, Kardelen Hatun, Mohamed Jmaiel, Jannik Jochem, Slim Kallel, Stefan Katzenbeisser, Lukáš Marek, Mira Mezini, Ralf Mitschke, Philippe Moret, Hela Oueslati, Nathan Ricci, Aibek Sarimbekov, Martin Schoeberl, Jan Sinschek, Éric Tanter, Petr Tůma, Zhengwei Qi, Alex Villazón, Dingwen Yuan, Martin Zandberg, and Yudi Zheng.

Others whom I have worked—albeit not written a paper—with are several talented and enthusiastic students: Erik Brangs, Pascal Flach, Felix Kerger, and Alexander Nickol; their work influenced and improved mine in various ways.

Still others have helped me realize the vision of a Scala benchmark suite for the Java Virtual Machine by graciously providing help with the suite’s various programs, the programs’ input data, or both: Joa Ebert, Patrik Nordwall, Daniel Ramage, Bill Venners, Tim Vieira, Eugene Yokota, and the late Sam Roweis. I am also deeply grateful to the DaCapo group for providing such an excellent foundation on which to build my Scala benchmark suite.

Of all the aforementioned, there are few to which I want to express my thanks in particular: Christoph Bockisch and everyone at the Dynamic Analysis group of the University of Lugano. Together with Michael Haupt, my colleague Christoph exposed me to the wonderful world of virtual machines in general and Jikes RVM in particular. He also supported me not only during my Diplom thesis and my first year at the Software Technology group, but also after he left us for the Netherlands, to assume a position as assistant professor at the Universeit Twente. Aibek, Danilo, and Walter from the University of Lugano helped my tremendously in developing and refining many of the tools needed for the numerous experiments conducted as part of this thesis. Moreover, they made me feel very welcome on my visit to what may be the most beautiful part of Switzerland.

Here at the Software Technology group, further thanks go to my fellow PhD candidates and post-docs for inspiring discussions, over lunch and beyond: Christoph Bockisch, Eric Bodden, Marcel Bruch, Tom Dinkelaker, Michael Eichberg, Vaidas Gasiunas, Ben Hermann, Sven Kloppenburg, Roman Knöll, Johannes Lerch, Ralf Mitschke, Martin Monperrus, Sebastian Proksch, Guido Salvaneschi, Lucas Satabin, Thorsten Schäfer, Jan Sinschk, and Jurgen van Ham. The quality of my quantitative evaluations in particular benefited from dozens of discussions I had with Marcel. I would also like to thank Gudrun Harris for her unfailing support in dealing with all things administrative and for just making the Software Technology Group a very pleasant place to work at. And for her excellent baking, of course.

My thanks also go to both the participants of and the reviewers for the Work-on-Progress session at the 2010 International Conference on the Principles and Practice of Programming in Java, held in Vienna; their suggestions and encouragement helped to turn a mere position paper into the thesis you now hold in your hands.

My parents Renate and Kai-Udo Sewe provided valuable support throughout all my life. Finally, I am indebted to Bettina Birkmeier for her encouragement and patience—not to mention countless hours of proof-reading.

Funding

Parts of my work have been funded by AOSD-Europe, the European Network of Excellence on Aspect-Oriented Software Development.¹ Other parts of the work have been funded by CASED, the Center for Advanced Security Research Darmstadt,² through LOEWE, the “Landes-Offensive zur Entwicklung Wissenschaftlich-ökonomischer Exzellenz.”

¹ See <http://www.aosd-europe.org/>.

² See <http://www.cased.de/>.

Abstract

In the last decade, virtual machines (VMs) for high-level languages have become pervasive, as they promise both portability and high performance. However, these virtual machines were often designed to support just a single language well. The design of the Java Virtual Machine (JVM), for example, is heavily influenced by the Java programming language.

Despite its current bias towards Java, in recent years the JVM in particular has been targeted by numerous new languages: Scala, Groovy, Clojure, and others. This trend has not been reflected in JVM research, though; all major benchmark suites for the JVM are still firmly focused on the Java language rather than on the language ecosystem as a whole. This state of affairs threatens to perpetuate the bias towards Java, as JVM implementers strive to “make the common case fast.” But what is common for Java may be uncommon for other, popular languages. One of these other languages is Scala, a language with both object-oriented and functional features, whose popularity has grown tremendously since its first public appearance in 2003.

What characteristics Scala programs have or have not in common with Java programs has been an open question, though. One contribution of this thesis is therefore the design of a Scala benchmark suite that is on par with modern, widely-accepted Java benchmark suites. Another contribution is the subsequent analysis of this suite and an in-depth, VM-independent comparison with the DaCapo 9.12 benchmark suite, the premier suite used in JVM research. The analysis shows that Scala programs exhibit not only a distinctive instruction mix but also object demographics close to those of the Scala language’s functional ancestors.

This thesis furthermore shows that these differences can have a marked effect on the performance of Scala programs on modern high-performance JVMs. While JVMs exhibit remarkably similar performance on Java programs, the performance of Scala programs varies considerably, with the fastest JVM being more than three times faster than the slowest.



Zusammenfassung

Aufgrund ihres Versprechens von Portabilität und Geschwindigkeit haben sich virtuelle Maschinen (VMs) für höhere Programmiersprachen in der letzten Dekade auf breiter Front durchgesetzt. Häufig ist ihr Design jedoch nur darauf ausgelegt, eine einzige Sprache gut zu unterstützen. So wurde das Design der Java Virtual Machine (JVM) zum Beispiel stark durch das Design der Programmiersprache Java beeinflusst.

Trotz ihrer aktuellen Ausrichtung auf Java hat sich insbesondere die JVM als Plattform für eine Vielzahl von neuer Programmiersprachen etabliert, darunter Scala, Groovy und Clojure. Dieser Entwicklung wurde in der Forschung zu JVMs bisher jedoch wenig Rechnung getragen; alle großen Benchmark Suites für die JVM sind immer noch stark auf Java als Sprache anstatt auf die Plattform als Ganzes fokussiert. Dieser Zustand droht, die systematische Bevorzugung von Java auf lange Zeit festzuschreiben, da die JVM-Entwickler ihre virtuellen Maschinen für die häufigsten Anwendungsfälle optimieren. Was aber häufig für Java ist, muss keinesfalls häufig für andere populäre Sprachen sein. Eine dieser Sprachen ist Scala, welche sowohl funktionale als auch objekt-orientierte Konzepte unterstützt und seit ihrer Veröffentlichung im Jahre 2003 stetig in der Entwicklergunst gestiegen ist.

Welche Charakteristika Scala-Programme mit Java-Programmen gemein haben ist allerdings eine weitgehend ungeklärte Frage. Ein Beitrag dieser Dissertation ist daher das Erstellen einer Benchmark Suite für die Programmiersprache Scala, die mit modernen, etablierten Benchmark Suites für Java konkurrieren kann. Ein weiterer Beitrag ist eine umfassende Analyse der in der Suite enthaltenen Benchmarks und ein VM-unabhängiger Vergleich mit den Benchmarks der DaCapo 9.12 Benchmark Suite, die bisher bevorzugt in der Forschung zu JVMs eingesetzt wird. Diese Analyse zeigt auf, dass Scala-Programme nicht nur den Befehlssatz der JVM merklich anders nutzen, sondern auch, dass allozierte Objekte eine Lebensdauer-Verteilung aufweisen, die der funktionaler Sprachen nahekommmt.

Wie diese Dissertation weiterhin zeigt, haben diese Unterschiede einen deutlichen Effekt auf die Geschwindigkeit, mit der Scala-Programme auf modernen Hochleistungs-JVMs ausgeführt werden. Während verschiedene JVMs sich beim Ausführen von Java-Programmen als ähnlich leistungsfähig erweisen, sind die Leistungsunterschiede im Falle von Scala-Programmen beträchtlich; die schnellste JVM ist hierbei mehr als dreimal so schnell wie die langsamste.



Contents

1	Introduction	1
1.1	Contributions of this Thesis	2
1.1.1	The Need for a Scala Benchmark Suite	2
1.1.2	The Need for Rapid Prototyping of Dynamic Analyses	3
1.1.3	The Need for VM-Independent Metrics	4
1.2	Structure of this Thesis	4
2	Background	7
2.1	The Java Virtual Machine	7
2.2	The Scala Language	10
2.3	The Translation of Scala Features to Java Bytecode	11
2.3.1	Translating Traits	11
2.3.2	Translating First-Class Functions	14
2.3.3	Translating Singleton Objects and Rich Primitives	16
3	Designing a Scala Benchmark Suite	17
3.1	Choosing a Benchmark Harness	17
3.2	Choosing Representative Workloads	17
3.2.1	Covered Application Domains	19
3.2.2	Code Size	20
3.2.3	Code Sources	22
3.2.4	The dummy Benchmark	24
3.3	Choosing a Build Toolchain	25
4	Rapidly Prototyping Dynamic Analyses	27
4.1	Approaches	28
4.1.1	Re-using Dedicated Profilers: JP2	29
4.1.2	Re-purposing Existing Tools: TamiFlex	33
4.1.3	Developing Tailored Profilers in a DSL: DiSL	35
4.2	Discussion	40
5	A Comparison of Java and Scala Benchmarks Using VM-independent Metrics	43
5.1	The Argument for VM-independent, Dynamic Metrics	43



- 5.2 Profilers 44
- 5.3 Threats to Validity 46
- 5.4 Results 48
 - 5.4.1 Instruction Mix 48
 - 5.4.2 Call-Site Polymorphism 54
 - 5.4.3 Stack Usage and Recursion 62
 - 5.4.4 Argument Passing 64
 - 5.4.5 Method and Basic Block Hotness 70
 - 5.4.6 Use of Reflection 73
 - 5.4.7 Use of Boxed Types 76
 - 5.4.8 Garbage-Collector Workload 77
 - 5.4.9 Object Churn 82
 - 5.4.10 Object Sizes 85
 - 5.4.11 Immutability 86
 - 5.4.12 Zero Initialization 90
 - 5.4.13 Sharing 93
 - 5.4.14 Synchronization 95
 - 5.4.15 Use of Identity Hash-Codes 99
- 5.5 Summary 102

6 An Analysis of the Impact of Scala Code on High-Performance JVMs 105

- 6.1 Experimental Setup 105
 - 6.1.1 Choosing Heap Sizes 106
 - 6.1.2 Statistically Rigorous Methodology 110
- 6.2 Startup and Steady-State Performance 111
- 6.3 The Effect of Scala Code on Just-in-Time Compilers 115
- 6.4 The Effect of Method Inlining on the Performance of Scala Code 122
- 6.5 Discussion 136

7 Related Work 139

- 7.1 Benchmark Suites 139
- 7.2 Workload Characterization 143
- 7.3 Scala Performance 147

8 Conclusions and Future Directions 151

- 8.1 Directions for Future Work 151

Bibliography 157

List of Figures

3.1	Classes loaded and methods called by the benchmarks	21
3.2	Bytecodes loaded and executed (Scala benchmarks)	23
3.3	Report generated by the dacapo-benchmark-maven-plugin	26
4.1	Sample calling-context tree	30
4.2	Sample output of JP2	31
4.3	Architecture of TamiFlex	35
4.4	Sample output of TamiFlex	36
5.1	The top four principal components	50
5.2	The first and second principal component	51
5.3	The third and fourth principal component	52
5.4	The first four principal components	53
5.5a	Call sites using different instructions (Java benchmarks)	55
5.5b	Call sites using different instructions (Scala benchmarks)	56
5.6a	Calls made using different instructions (Java benchmarks)	57
5.6b	Calls made using different instructions (Scala benchmarks)	58
5.7a	Histogram of call-site targets (Java benchmarks)	59
5.7b	Histogram of call-site targets (Scala benchmarks)	60
5.8a	Histogram of call targets (Java benchmarks)	61
5.8b	Histogram of call targets (Scala benchmarks)	62
5.9	Maximum stack heights	63
5.10a	Stack-height distribution (Java benchmarks)	65
5.10b	Stack-height distribution (Scala benchmarks)	66
5.11	Distribution of the number of floating-point arguments	67
5.12a	Distribution of the number of reference arguments (Java benchmarks)	68
5.12b	Distribution of the number of reference arguments (Scala benchmarks)	69
5.13a	Method and basic-block hotness (Java benchmarks)	71
5.13b	Method and basic-block hotness (Scala benchmarks)	72
5.14	Use of reflective method invocation	74
5.15	Use of reflective object instantiation	75
5.16	Use of boxed types	77
5.17a	Survival rates (Java benchmarks)	79
5.17b	Survival rates (Scala benchmarks)	80
5.18	Allocations and pointer mutations	83

5.19	The dynamic churn-distance metric	84
5.20a	Churn distances (Java benchmarks)	85
5.20b	Churn distances (Java benchmarks)	86
5.21a	Object sizes (Java benchmarks)	88
5.21b	Object sizes (Java benchmarks)	89
5.22a	Use of immutable instance fields	90
5.22b	Use of immutable fields	91
5.23a	Use of immutable objects	92
5.23b	Use of immutable classes	93
5.24a	Necessary and unnecessary zeroing (Java benchmarks)	94
5.24b	Necessary and unnecessary zeroing (Scala benchmarks)	95
5.25a	Shared objects with respect to read accesses	96
5.25b	Shared objects with respect to write accesses	97
5.25c	Shared types	98
5.26	Objects synchronized on	99
5.27a	Nested lock acquisitions (Java benchmarks)	100
5.27b	Nested lock acquisitions (Scala benchmarks)	100
5.28	Fraction of objects hashed	101
6.1a	Startup execution time (Java benchmarks)	111
6.1b	Startup execution time (Scala benchmarks)	112
6.2a	Steady-state execution time (Java benchmarks)	113
6.2b	Steady-state execution time (Scala benchmarks)	114
6.3a	Methods optimized by OpenJDK 6	117
6.3b	Methods optimized by OpenJDK 7u	117
6.3c	Methods optimized by Jikes RVM	118
6.4a	Steady-state execution time with tuned compiler DNA (Java bench.)	122
6.4b	Steady-state execution time with tuned compiler DNA (Scala bench.)	123
6.5a	Bytecodes optimized by OpenJDK 6	124
6.5b	Bytecodes optimized by OpenJDK 7u	124
6.5c	Bytecodes optimized by Jikes RVM	125
6.6	Bytecodes optimized by Jikes RVM over time	127
6.7a	Amount of inline expansion in OpenJDK 6	128
6.7b	Amount of inline expansion in OpenJDK 7u	128
6.7c	Amount of inline expansion in Jikes RVM	129
6.8a	Speedup achieved by tuned inlining over steady state (Java bench.)	131
6.8b	Speedup achieved by tuned inlining over steady state (Scala bench.)	132
6.9a	Speedup achieved by inlining over startup (Java benchmarks)	134
6.9b	Speedup achieved by inlining over startup (Scala benchmarks)	134
6.10a	Speedup achieved by inlining over steady state (Java benchmarks)	135



- 6.10b Speedup achieved by inlining over steady state (Scala benchmarks) . . 136
- 6.11a Steady-state execution time w.r.t. inlining (Java benchmarks) 137
- 6.11b Steady-state execution time w.r.t. inlining (Scala benchmarks) 138

- 7.1 Benchmark suites used for JVM research 140

- 8.1 The website of the Scala Benchmark Project 152



List of Tables

3.1	The 12 benchmarks of the Scala benchmark suite	18
4.1	Approaches to prototyping dynamic analyses	42
5.1	Summary of garbage collection simulation results	78
5.2	Allocations and 1 MiB survival rates (Scala benchmarks)	82
5.3	Categorized median churn distances (Scala benchmarks)	87
6.1	Minimum required heap sizes	107
6.2	Optimal heap sizes	109
6.3	Compiler DNA for Jikes RVM	120
6.4	Time spent in Jikes RVM compiler phases	121



List of Listings

2.1a	The <code>Logger</code> trait and an implementation of it in Scala	11
2.1b	The <code>Logger</code> trait from Listing 2.1a translated into Java	12
2.2a	The <code>Decorations</code> trait composed with a class in Scala	12
2.2b	The mixin composition of Listing 2.2a translated into Java	13
2.3	Various features of Scala and their translation into Java	15
4.1	XQuery script computing a benchmark’s instruction mix	32
4.2	DiSL class instrumenting object allocations	37
4.3	Runtime class managing the shadow heap	38
4.4	DiSL class instrumenting hash-code calls	39
4.5	Runtime classes keeping track of per-object hash-code calls	41
5.1	The four categories of “under-the-hood” objects (cf. Listing 2.3)	81



1 Introduction

In recent years, managed languages like Java and C# have gained much popularity. Designed to target a virtual rather than a “real” machine, these languages offer several benefits over their unmanaged predecessors like C and C++: improved portability, memory safety, and automatic memory management.

The popularity of the aforementioned languages has caused much expenditure of effort [Doe03] in making them run on their underlying virtual machines, the Java Virtual Machine (JVM) [LYBB11] and the Common Language Runtime (CLR) [ECM10], respectively, as fast as their predecessors ran on “real” ones. This effort led to significant advances in just-in-time compilation [Ayc03] and garbage collection [BCM04]. Ultimately, it resulted in virtual machines (VMs) that are highly optimized, yet portable and mature.

Consequently, these VMs have become an attractive target for a plethora of programming languages, even if they, as is the case for the Java Virtual Machine,¹ were conceived with just a single source language in mind. As of this writing, the JVM is targeted by literally dozens of languages, of which Clojure, Groovy, Python (Jython), Ruby (JRuby), and Scala are arguably the most prominent. Just as the CLR is a *common* language runtime, today the JVM can rightly be considered a *joint* virtual machine.

Targeting such a mature and wide-spread joint virtual machine offers a number of benefits to language implementers, not least among them the staggering amount of existing library code readily available to the language’s users. Alas, just targeting the JVM does not necessarily result in performance as good as Java’s; existing JVMs are primarily tuned with respect to the characteristics of Java programs. The characteristics of other languages, even if compiled for the same target, may differ widely, however. For example, dynamically-typed source languages like Clojure, Groovy, Python, and Ruby all suffer because the JVM was built with only Java and its static type system in mind. The resulting semantics gap causes significant performance problems for these four languages, which have only recently been addressed by Java Specification Request 292 [RBC⁺11] and the dedicated **invokedynamic** instruction [Ros09, TR10] specified therein.

Similar bottlenecks undoubtedly exist for statically-typed source languages like Scala [OSV10]. For these languages, however, it is much less clear what the bottle-

¹ As the name suggests, the Common Language Runtime was conceived as the target of many languages, albeit with a bias towards imperative and object-oriented ones [Sin03].

necks are. In fact, the designers of Scala claim that "the Scala compiler produces byte code that performs every bit as good as comparable Java code."²

In this thesis, I will therefore explore the performance characteristics of Scala code and their influence on the underlying Java Virtual Machine. My research is guided by the golden rule of performance optimization: "Make the common case fast." But what is common for Java code may be rather uncommon for some other languages. Thus, the two key questions this thesis will answer are the following:

- "Scala $\stackrel{?}{\equiv}$ Java mod JVM" [Sew10]. In other words, is Scala code, when viewed from the JVM's perspective, similar or dissimilar to Java code?
- If it is dissimilar, what are the assumptions that JVM implementers have to reconsider, e.g. about the instruction mix or object demographics of programs?

1.1 Contributions of this Thesis

Answering the aforementioned questions requires a multi-faceted research effort that has to address several needs: first, the need for a dedicated Scala benchmark suite; second, the need for rapidly prototyping dynamic analyses to facilitate characterization of the suite's workloads; and third, the need for a broad range of VM-independent metrics. The contribution of this thesis lies thus not only in answering the two questions above, but also in the creation of research tools and infrastructure that satisfies these needs.

1.1.1 The Need for a Scala Benchmark Suite

First and foremost, answering any questions about a language's performance characteristics requires rigorous benchmarking. Previous investigations into the performance of Scala code were mostly restricted to micro-benchmarking. While such micro-benchmarks are undeniably useful in limited circumstances, e.g. to help the implementers of the Scala compiler decide between different code-generation strategies for a language feature [Sch05, Section 6.1], they are mostly useless in answering the research questions stated above, as micro-benchmarks rarely reflect the common case. Implementers of high-performance JVMs, however, need a good understanding of what this common case is in order to optimize for it.

² See <http://www.scala-lang.org/node/25#ScalaCompilerPerformance>.

Consequently, a new Scala benchmark suite must be developed which is on par with well-respected Java benchmark suites like the SPECjvm2008 or DaCapo suites. It must offer “relevant and diverse workloads” and be “suitable for research” [BMG⁺08]. Alas, the authors of the DaCapo benchmark suite estimated that to meet these requirements they “spent 10,000 person-hours [...] developing the DaCapo suite and associated infrastructure” [BMG⁺08].³ Developing a benchmark suite single-handedly⁴ therefore requires effective tools and infrastructure to support the design and analysis of the suite. The contributions of this thesis are thus not restricted to the resulting Scala benchmark suite but encompass also necessary tools and infrastructure, e.g. the build toolchain used.

1.1.2 The Need for Rapid Prototyping of Dynamic Analyses

A high-quality Scala benchmark suite and the tools to build one are only one prerequisite in answering the research questions. Just as essential are tools to analyze the individual benchmarks, both to ensure that the developed suite contains a diverse mix of benchmarks and to compare its Scala benchmarks with the Java benchmarks from an already existing suite. The former kind of analysis in particular is largely exploratory; as new candidate benchmarks are considered for inclusion, they are compared to existing benchmarks using a broad range of metrics. Thus, tools are needed that allow for rapid prototyping of such analyses.

While Blackburn et al. [BGH⁺06] resort to modifying a full-fledged Java Virtual Machine, I favour an approach that uses VM-independent tools to collect raw data. This approach has the distinct advantage that the tools are more likely to remain applicable to other, newer benchmark suites. This is in stark contrast to the modifications⁵ performed by Blackburn et al. While their original results remain reproducible, new ones cannot be produced. With such VM-dependent tools a comparison of a new benchmark suite with an older, established one would therefore in all likelihood be impossible.

With respect to tool development, this thesis contributes three case studies using different approaches to rapidly prototype VM-independent dynamic analyses: re-using dedicated profilers [SSB⁺11], re-purposing existing tools, and developing tailored profilers in a domain-specific language [ZAM⁺12]. The first

³ These numbers pertain to the 2006-10 release of the DaCapo benchmark suite; the 9.12 release is larger still, increasing the number of benchmarks from 11 to 14.

⁴ While the relevant publication [SMSB11] mentions four authors, the actual development of the benchmark suite itself laid solely in the hands of the author of this thesis.

⁵ These modifications consist of two sets of patches, one for Jikes RVM 2.4.5 and one for Jikes RVM 2.4.6, none of which is directly applicable to newer versions of the virtual machine.

and third approach hereby highlight the strength of domain-specific languages, XQuery [BCF⁺10] respectively DiSL [MVZ⁺12], when it comes to concisely describing one's metrics. What is common to all three approaches is that they require significantly less development effort than writing a profiler from scratch—be it a VM-dependent or VM-independent one.

1.1.3 The Need for VM-Independent Metrics

To answer the two main questions, I need to characterize and contrast Java and Scala workloads. To do so, a benchmark suite and tools to prototype the desired dynamic analyses are certainly necessary but not sufficient. What is also needed is a broad selection of metrics to subject the workloads to. These metrics should not only cover both code-related and memory-related behaviour, but they should also be independent of any specific VM. Their VM-independence guarantees that the characterizations obtained are indeed due to the intrinsic nature of real-world Java and Scala code, respectively, rather than due to the implementation choices of a particular VM.

In the area of metrics, the contribution of this thesis consists of several novel, VM-independent metrics, which are nevertheless all tied to optimizations commonly performed by modern JVMs, e.g. method inlining. New metrics are, however, introduced only in situations not adequately covered by established metrics; in all other situations, already established metrics have been used to facilitate comparison of my results with those of others.

1.2 Structure of this Thesis

The remainder of this thesis is structured as follows:

Background

Chapter 2 provides the necessary background on the Java Virtual Machine, with a particular focus on its instruction set, namely Java bytecode. The chapter furthermore contains a brief description of the Scala languages and its most relevant features as well as an outline of those features' translations into Java bytecode, insofar as it is necessary to understand the observations made in Chapter 5.

Designing a Scala Benchmark Suite

Next, Chapter 3 describes the design of the Scala benchmark suite developed for this thesis. It discusses the selection criteria for its constituent benchmarks and the

application domains covered by them. Furthermore, it briefly discusses the build toolchain developed for the benchmark suite's creation.

Rapidly Prototyping Dynamic Analyses

The subsequent Chapter 4 describes three approaches to rapidly prototype dynamic analyses, which can then be used to measure various performance characteristics of benchmarks: re-using dedicated profilers, re-purposing existing tools as profilers, and developing tailored profilers in a domain-specific language.

A Comparison of Java and Scala Benchmarks Using VM-independent Metrics

Chapter 5 constitutes the main contribution of this thesis: a detailed comparison of Java and Scala code using a variety of VM-independent metrics, both established metrics and novel ones. The novel metrics are furthermore defined precisely. This chapter encompasses both code- and memory-related metrics.

An Analysis of the Impact of Scala Code on High-Performance JVMs

Unlike the previous chapter, Chapter 6 compares Java and Scala code in terms of their performance impact on a broad selection of modern Java virtual machines. Prompted by the results of the comparison, it furthermore contains an in-depth investigation into the performance problems exhibited by a particular JVM, namely the Jikes RVM. As this investigation shows, Scala performance of said VM suffers from shortcomings in the optimizing compiler which cannot be explained solely by a poorly tuned adaptive optimization system or inlining heuristic.

Related Work

Chapter 7 reviews related work in the areas of benchmark-suite design and workload characterization. It furthermore gives an overview of current research on improving Scala performance, i.e. on lessening the performance impact of Scala code.

Conclusions and Future Directions

Chapter 8 concludes this thesis and discusses directions for future work made possible by the benchmark suite developed as part of this thesis. In particular, it gives an outlook on the next release of the DaCapo benchmark suite, tentatively called version 12.x, which will apply some of the lessons learned during the genesis of this thesis.



2 Background

This chapter provides the background necessary to follow the discussion in the rest of this thesis in general and in Chapter 5 in particular. First, Section 2.1 introduces the Java Virtual Machine and its instruction set. Next, Section 2.2 describes the key features of the Scala language, whereas Section 2.3 sketches how these features are translated into the JVMs instruction set.

2.1 The Java Virtual Machine

The Java Virtual Machine (JVM) [LYBB11] is an abstract, stack-based machine, whose instruction set is geared towards execution of the Java programming language [GJS⁺11]. Its design emphasizes portability and security.

Instruction Set

The JVM's instruction set, often referred to as Java bytecode, is, for the most part, very regular. This makes it possible to formalize many of its aspects like the effects individual instructions have on a method's operand stack [ES11]. At the simplest level, the instruction set can be split into eight categories:

Stack & Local Variable Manipulation: `pop`, `pop2`, `swap`, `dup`, `dup_x1`, `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2`, `ldc`, `ldc_w`, `ldc2_w`, `aconst_null`, `iconst_m1`, `iconst_0`, `iconst_1`, `iconst_2`, `iconst_3`, `iconst_4`, `iconst_5`, `lconst_0`, `lconst_1`, `fconst_0`, `fconst_1`, `fconst_2`, `dconst_0`, `dconst_1`, `bipush`, `sipush`, `iload`, `lload`, `fload`, `dload`, `aload`, `iload_0`, `iload_1`, `iload_2`, `iload_3`, `lload_0`, `lload_1`, `lload_2`, `lload_3`, `fload_0`, `fload_1`, `fload_2`, `fload_3`, `dload_0`, `dload_1`, `dload_2`, `dload_3`, `aload_0`, `aload_1`, `aload_2`, `aload_3`, `istore`, `lstore`, `fstore`, `dstore`, `astore`, `istore_0`, `istore_1`, `istore_2`, `istore_3`, `lstore_0`, `lstore_1`, `lstore_2`, `lstore_3`, `fstore_0`, `fstore_1`, `fstore_2`, `fstore_3`, `dstore_0`, `dstore_1`, `dstore_2`, `dstore_3`, `astore_0`, `astore_1`, `astore_2`, `astore_3`, `nop`

Arithmetic & Logical Operations: `iadd`, `ladd`, `fadd`, `dadd`, `isub`, `lsub`, `fsub`, `dsub`, `imul`, `lmul`, `fmul`, `dmul`, `idiv`, `ldiv`, `fdiv`, `ddiv`, `irem`, `lrem`, `frem`, `drem`, `ineg`, `lneg`, `fneg`, `dneg`, `iinc`, `ishl`, `lshl`, `ishr`, `lshr`, `iushr`, `lushr`, `iand`, `land`, `ior`, `lor`, `ixor`, `lxor`, `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, `dcmpg`

Type Checking & Coercions: `checkcast`, `instanceof`, `i2l`, `i2f`, `i2d`, `l2i`, `l2f`, `l2d`, `f2i`, `f2l`, `f2d`, `d2i`, `d2l`, `d2f`, `i2b`, `i2c`, `i2s`

Control Flow (intra-procedural): `ifeq`, `ifne`, `iflt`, `ifge`, `ifgt`, `ifle`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpge`, `if_icmpgt`, `if_icmple`, `if_acmpeq`, `if_acmpne`, `ifnull`, `ifnonnull`, `goto`, `goto_w`, `tableswitch`, `lookupswitch`, `jsr`, `jsr_w`, `ret`

Control Flow (inter-procedural): `invokevirtual`, `invokespecial`, `invokestatic`, `invokeinterface`, `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, `return`, `throw`

Memory Allocation: `new`, `newarray`, `anewarray`, `multianewarray`

Memory Accesses `getstatic`, `putstatic`, `getfield`, `putfield`, `arraylength`, `iaload`, `laload`, `faload`, `daload`, `aaload`, `baload`, `caload`, `saload`, `iastore`, `lastore`, `fastore`, `dastore`, `aastore`, `bastore`, `castore`, `sastore`

Synchronization: `monitorenter`, `monitorexit`

As the JVM is a stack machine, a large number of instructions exist to manipulate the operand stack. Many of these simply push a (typed) constant onto the operand stack. In addition to the operand stack, there exists a set of variables local to the current method activation. The operand stack and local variables may or may not be kept in main memory; for performance reasons, an optimizing just-in-time compiler is often able to keep most operands and variables in hardware registers. Unlike the virtual machine's variables, however, these registers are a scarce resource.

Arithmetic and logical operations mimic the machine instructions offered by most modern architecture. They deal with both integers and floating-point values of 32 bit and 64 bit width, represented in the conventional two's-complement and IEEE 754 formats, respectively.

Intra-procedural control-flow is covered by instructions for conditional and unconditional jumps, including multi-way jumps (**`tableswitch`**, **`lookupswitch`**). Inter-procedural control-flow covers both method calls and returns (see below) and the raising of exceptions.

Memory allocation distinguishes between scalar objects and arrays. Despite the presence of an instruction that seemingly creates multi-dimensional arrays (**`multianewarray`**), the JVM lacks true multi-dimensional arrays; instead, they are represented as array of arrays, with each component array being an object in its own right.

Memory accesses also distinguish between reads and writes to a scalar object's fields and to an array's elements. Moreover, there exists a third mode of memory accesses, namely accesses to static fields, which are not associated with any object and typically reside at a fixed memory location.

The JVM has built-in support for synchronization; every object can be synchronized on, i.e. serve as a lock. Not every lock and unlock operation is represented by an explicit instruction, though; **synchronized** methods implicitly attempt to acquire the receiver's lock.

Classfiles

Each Java class is stored in a so-called classfile, which contains the bytecode of all methods declared by that class¹ as well as information about the class's declared fields and their default values. Literals referred to from the methods' bytecode are kept in a so-called constant pool, which is not directly accessible by the program itself, but rather serves the purpose of the JVM. Likewise, the symbolic references representing the names and descriptors of methods, fields, and classes are kept in the constant pool.

While the Java source language supports nested and inner classes, these are translated to top-level classes by the Java compiler, `javac`, and stored in classfiles of their own. Similarly, interfaces are stored in their own classfiles.

Types

Unlike many other assembly languages, Java bytecode is statically typed. Most instructions operating on either operand stack or local variables therefore exist in five flavours, operating on references or primitive values of type **int**, **long**, **float**, or **double**, respectively. While the Java source language also knows other **int**-like types, e.g. **byte** and **short**, at least arithmetic instructions do not distinguish between them; they always operate on integers of 32 bit width. Instructions to read from and write to an array, however, do distinguish **byte**, **short**, and **char**, with **boolean** values being treated as **byte**-sized. For the purposes of arithmetic the **boolean** type is treated like **int**, however.

Method Calls

The JVM instruction set has four dedicated instructions for performing method calls: **invokevirtual**, **invokeinterface**, **invokespecial**, and **invokestatic**.²

¹ Constructors and static initializers are treated as special methods named `<init>` and `<clinit>`, respectively.

² Java 7 introduced the **invokedynamic** instruction [LYBB11] to better support dynamically-typed languages targeting the JVM. As it is used by neither the Java nor Scala benchmarks analyzed in this thesis, it is not discussed here.

The first two instructions are used for dynamically-dispatched calls, whereas the latter two are used for statically-dispatched ones. The **invokevirtual** instruction is used for most method calls found in Java programs [GPW05] (cf. Section 5.4.2); these calls are polymorphic with respect to the receiver's type and can be handled efficiently by a vtable-like mechanism. As the name suggests, the **invokeinterface** instruction can be used for calling methods defined in an interface, although for performance reasons using **invokevirtual** is preferable if the receiver's class is known and not just the interface it implements; in that case, vttables are applicable. In the general case, however, vttables are not applicable to interface methods, although there exist techniques [ACFG01] which can alleviate much of the performance cost associated with the more flexible **invokeinterface** instruction. The **invokespecial** instruction is used in three different circumstances, in each of which the target implementation is statically known: constructor calls, **super** calls, and calls to **private** methods. In contrast to the **invokestatic** instruction, which handles calling **static** methods, **invokespecial** calls have a dedicated receiver object.

2.2 The Scala Language

As the JVM specification states, “implementors of other languages are turning to the Java virtual machine as a delivery vehicle for their languages,” since it provides a “generally available, machine-independent platform” [LYBB11]. From among the dozens of other languages, I have selected the Scala language [OSV10] for further study, as it exhibits some characteristics which may significantly impact execution on the underlying Java Virtual Machine:

Static typing The Scala language offers a powerful static type system, including a form of mixin-based inheritance. (To help the programmer harness the type system's power, the language provides a local type-inference mechanism.)

First-class functions Scala supports first-class functions and seamlessly integrates them with the method-centric view of object-oriented languages.

Fully object-oriented In Scala, every value is an object and hence an instance of a class. This includes integer values (instances of `Int`) and first-class functions (instances of `Function0`, `Function1`, etc.).

Scala can thus be considered a mix of object-oriented and functional language features. In the following section, I will explore how the aforementioned three characteristics influence the translation from Scala source code to Java bytecode.

```
1 trait Logger {
2   def log(msg: String)
3 }
4
5 class SimpleLogger(out: PrintStream) extends Logger {
6   def log(msg: String) { out.print(msg) }
7 }
```

Listing 2.1a: The Logger trait and an implementation of it in Scala

Note that other characteristics of the language have little impact on the language's execution characteristics. For example, the language's syntactic flexibility, which makes it suitable for the creation of embedded domain-specific languages, has virtually no impact on the resulting bytecode that is executed by the Java Virtual Machine.

2.3 The Translation of Scala Features to Java Bytecode

In this section, I will outline how the most important features of Scala are compiled to Java bytecode.³ This description is based on the translation strategy of version 2.8.1 of the Scala compiler, which is the version used to compile the benchmarks from my Scala benchmark suite. That being said, version 2.9.2, as of this writing the latest stable release, does employ the same translation strategy at least for the features discussed in this section.

2.3.1 Translating Traits

Consider the simple trait shown in Listing 2.1a, which only declares a method without providing its implementation. Such a trait simply translates into the Java interface shown in Listing 2.1b. The `Logger` trait is thus completely interoperable with Java code; in particular, Java code can simply implement it, even though the interface originated as a Scala trait. An implementation of the `Logger` trait/interface only needs to provide the missing method implementations. This is exemplified by the `SimpleLogger` class in Listing 2.1a and its Java counterpart in Listing 2.1b. Note how the Scala compiler translates the constructor's parameter into a **final** field; this is just one example of the Scala language's preference for immutable

³ For the sake of readability, equivalent Java source code rather than bytecode will be shown.

```
1 public interface Logger {
2     void log(String msg);
3 }
4
5 public class SimpleLogger implements Logger, ScalaObject {
6     private final out;
7     public SimpleLogger(PrintStream out) { this.out = out; }
8     public void log(String msg) { out.print(msg); }
9 }
```

Listing 2.1b: The Logger trait from Listing 2.1a translated into Java

```
1 trait Decorations extends Logger {
2     abstract override def log(msg: String) {
3         super.log("[log] " + msg)
4     }
5 }
6
7 class DecoratedLogger(out: PrintStream) extends SimpleLogger(out)
8     with Decorations
```

Listing 2.2a: The Decorations trait composed with a class in Scala

data structures (cf. Section 5.4.11). Also note that `SimpleLogger` implements an additional marker interface: `ScalaObject`.

So far, Scala's traits seem to offer little more than Java interfaces. But unlike interfaces, traits can also provide implementations for the methods they declare. In particular, traits can modify the behaviour of the base class they are mixed into. This makes it possible to write traits like the one shown in Listing 2.2a. Here, the `Decorations` trait decorates the output of any `Logger` it is mixed into. This is exemplified in the following interaction with the Scala console, `scala`:

```
1 > val logger = new SimpleLogger(System.err)
2 > logger.log("Division by zero")
3 Division by zero
4 > val decoratedLogger = new DecoratedLogger(System.err)
5 > decoratedLogger.log("Division by zero")
6 [log] Division by zero
```

```

1 public interface Decorations extends Logger, ScalaObject {
2     void log(String);
3     void Decorations$$super$log(String);
4 }
5
6 public abstract class Decorations$class {
7     public static void $init$(Decorations) { }
8     public static void log(Decorations delegator, String msg) {
9         // invokeinterface
10        delegator.Decorations$$super$log("[log] " + msg);
11    }
12 }
13
14 public class DecoratedLogger extends SimpleLogger
15     implements Decorations, ScalaObject {
16     public void log(String msg) {
17         Decorations$class.log(this, msg); // invokestatic
18     }
19     public final void Decorations$$super$log(String msg) {
20         super.log(msg); // invokespecial
21     }
22     public DecoratedLogger(PrintStream out) {
23         super(out);
24         Decorations$class.$init$(this);
25     }
26 }

```

Listing 2.2b: The mixin composition of Listing 2.2a translated into Java

The translation of the `DecoratedLogger` class's mixin composition with the `Decorations` trait is shown in Listing 2.2b. As can be seen, the Java version of the `DecoratedLogger` class implements the `Decorations` Java interface, which complements the `log` method with a second method used internally for super-calls: `Decorations$$super$log`. When the user calls the `log` method, `DecoratedLogger` delegates the call first to the `Decorations` trait's implementation class: `Decorations$class`. As the implementation of the trait's functionality resides in a `static` method, this happens using the `invokestatic` instruction. The said implementation method in turn delegates back to the `DecoratedLogger`'s `Decorations$$super$log` method, but this time using the `invokeinterface` in-

struction. The delegator class can then decide anew to which mixin, if any, to delegate the call. In Listing 2.2a, there is no further trait mixed into `DecoratedLogger`; thus, the class's own implementation, a super-call using `invokeSpecial`, is executed.

Methods like `Decorations$$super$log` always transfer control back to the delegator, which is the only class that knows about the composition of mixins in its entirety and can therefore act as a switchboard for the mixins' super-calls. Now, `Decorations$$super$log` is defined in the `Decorations` interface, which requires the use of the `invokeInterface` bytecode instruction since the exact type of the mixin's superclass is unknown in the mixin's implementation class. This alternation of `invokeStatic` and `invokeInterface` calls is typical of the delegation-based translation scheme the Scala compiler uses for mixin composition. While the scheme allows for separate compilation and avoids code duplication [Sch05], when the same trait is mixed into many classes it produces megamorphic call sites, i.e. call sites which target many different implementations. This, however, need not be a performance problem, as the initial call (`invokeStatic`) is likely to be inlined during just-in-time compilation. Inlining thereby propagates type information about the delegator to the delegatee, which the compiler can use in turn to devirtualize and subsequently inline the previously megamorphic call (`invokeInterface`). This propagates the type information further yet. The resulting cascade can thus theoretically inline all implementation of the various mixins overriding a method. In practice, though, the just-in-time compiler's inlining heuristic limits the maximum inlining depth (cf. Section 6.4).

2.3.2 Translating First-Class Functions

The Scala compiler converts first-class functions into objects, which are, depending on the function's arity, instances of interfaces `Function0`, `Function1`, etc. One such translation is shown in Listing 2.3, where an anonymous function is passed to the `foreach` method defined in the Scala library. The translation of this function results in a new class `Countdown$$anonfun`, whose `apply` method contains the function's body.⁴ This body refers to variables defined in their enclosing lexical scope; in Listing 2.3, the function captures the variable `xs`, for example. The Scala compiler therefore needs to enclose the captured variable in a heap-allocated box, here of

⁴ This presentation has been simplified considerably: The Scala compiler specializes the `apply` method [Dra10] and thus generates several variants of it, with and without boxing of the function's arguments and return value (an instance of `BoxedUnit` in this case); these variants have been omitted.

```

1 object Countdown {
2   def nums = {
3     var xs = List[Int]()
4     (1 to 10) foreach {
5       x =>
6         xs = x :: xs
7     }
8     xs
9   }
10 }

1 public final class Countdown {
2   public static List nums() {
3     Countdown$.MODULE$.nums();
4   }
5 }
6
7 public final class Countdown$
8   implements ScalaObject {
9   ...
10  public List nums() {
11    ObjectRef xs =
12      new ObjectRef(...Nil$);
13    ...intWrapper(1).to(10).foreach(
14      new Countdown$$anonfun(xs);
15    );
16    return xs.elem;
17  }
18 }
19
20 public final class Countdown$$anonfun
21   implements Function1, ScalaObject {
22  private final ObjectRef xs;
23  Countdown$$anonfun(ObjectRef xs) {
24    this.xs = xs;
25  }
26  public void apply(int x) {
27    xs.elem = xs.elem
28      .$colon$colon(...boxToInteger(x));
29  }
30 }

```

Listing 2.3: Various features of Scala and their translation into Java

type `ObjectRef`, whose contents can be updated from within the function object’s `apply` method.

This translation roughly corresponds to the function object an experienced Java programmer would have written by hand in a similar situation, e.g. when registering a listener or callback. One may safely assume, however, that anonymous functions are far more common in Scala code than their function-object counter-

parts are in Java code. The allocation of both function objects and boxes that contain captured variables may thus significantly influence a Scala program's object demographics (cf. Sections 5.4.8 and 5.4.9).

2.3.3 Translating Singleton Objects and Rich Primitives

In Scala, every value is an object. Moreover, every method is invoked on an instance; there are no **static** methods as there are in Java. However, Scala has built-in support for the Singleton pattern using its **object** keyword, which makes it easy to emulate non-instance methods using the methods of a Singleton instance.

The translation of such a Singleton object produces two classes; in Listing 2.3, the classes `Countdown` and `Countdown$`, for example. The former offers only static methods which forward execution to the sole instance of `Countdown$` kept in the `MODULE$` field. This instance is created on demand, i.e. on first use, by `Countdown`'s static initializer (not shown).

Singleton objects are important in Scala as they often house Factory Methods, in particular if the Singleton happens to be a class's companion object [OSV10, Chapter 4]. Now, each external call of one of the Singleton's methods in theory needs to be dynamically dispatched (**invokevirtual**). In practice, however, these calls can easily be inlined by the JVM, as the type of the `MODULE$` field is precise. Moreover, as the Singleton is an instance of a **final** class, the JVM can be sure that no subclass thereof will be loaded at a later point, which would dilute the previously precise type information; thus, no guards are needed when inlining.

Another feature of Scala, which illustrates the mind-set that every value is an object, are the so-called rich primitives. These are objects which wrap the JVM's primitives (**int**, **long**, etc.) such that one can seemingly invoke methods on them. Under the hood, there exists an implicit conversion from the primitive to its rich wrapper, which is automatically applied by the Scala compiler [OSV10, Chapter 16]; in Listing 2.3, the `intWrapper` method wraps the integer 1 into a `RichInt`, for example. These wrapper objects are typically very short-lived and put unnecessary pressure on the garbage collector (cf. Sections 5.4.8 and 5.4.9) provided that the JVM cannot determine that the rich primitive is only alive in a very limited scope.

To avoid the creation of wrapper objects like `RichInt` altogether, an improvement to the Scala language has been proposed (and accepted)⁵ which removes most wrapping, replacing the dynamically-dispatched calls on the wrapper object with statically-dispatched calls to extension methods. But Scala 2.8, on which my Scala benchmark suite is based, does not implement these value classes yet; they are expected for the (as of this writing) unreleased Scala 2.10.

⁵ SIP-15 (Value Classes). See <http://docs.scala-lang.org/sips/>.

3 Designing a Scala Benchmark Suite

In this chapter, I describe the design of the Scala benchmark suite developed for this thesis. In particular, I will argue that the resulting suite is well-suited for research. I accomplish this goal by carefully choosing both the benchmark harness and the workloads, and by picking a toolchain that ensures build reproducibility.

Parts of this chapter have been published before:

- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048118

3.1 Choosing a Benchmark Harness

Every benchmark suite consists of two components: the actual workloads and an accompanying benchmark harness, which wraps the workloads and allows researchers to iterate them several times and to time the individual iterations. To ease adoption, I based the Scala benchmark suite on the latest release (version 9.12, nicknamed “Bach”) of the DaCapo benchmark suite [BGH⁺06]; the two suites share a harness. By re-using this core component of a benchmark suite that not only strives for “ease of use” [BGH⁺06] but is also immensely popular among JVM researchers,¹ it becomes easy to obtain experimental results for all the benchmarks in the two suites without any change to one’s experimental setup.

3.2 Choosing Representative Workloads

Table 3.1 lists the 12 Scala benchmarks I added to the 14 Java benchmarks from the DaCapo benchmark suite. The Scala benchmark suite² therefore contains almost as many benchmarks as the current release of the DaCapo benchmark suite and one more than its previous release (version 2006-10), despite a more limited set

¹ According to the ACM Digital Library, the paper describing the benchmark suite has gathered 251 citations (as of 9 August 2012).

² See <http://www.scalabench.org/>.

Benchmark	Description Input	References Sizes (#)
actors	Trading sample with Scala and Akka actors Run performance tests (varying number of transactions)	n/a tiny-gargantuan (6)
apparat	Framework to optimize ABC/SWC/SWF files Optimize (strip, inject, and reduce) various SWC files	n/a tiny-gargantuan (6)
factorie	Toolkit for deployable probabilistic modeling Perform Latent Dirichlet Allocation on a variety of data sets	[MSS09] tiny-gargantuan (6)
kiama	Library for language processing Compile programs written in Obr; execute programs written in a extension to Landin's ISWIM language	[Slo08] small-default (2)
scalac	Compiler for the Scala 2 language Compile various parts of the scalap classfile decoder	[Sch05, Dra10] small-large (3)
scaladoc	Scala documentation tool Generate ScalaDoc for various parts of the scalap classfile decoder	n/a small-large (3)
scalap	Scala classfile decoder Disassemble various classfiles compiled with the Scala compiler	n/a small-large (3)
scalariform	Code formatter for Scala Reformat various source code from the Scala library	n/a tiny-huge (5)
scalatest	Testing toolkit for Scala and Java Run various tests of ScalaTest itself	n/a small-huge (4)
scalaxb	XML data-binding tool Compile various XML Schemas	n/a tiny-huge (5)
specs	Behaviour-driven design framework Run various tests and specifications of Specs itself	n/a small-large (3)
tmt	Stanford topic modeling toolbox Learn a topic model (varying number of iterations)	[RRC ⁺ 09] tiny-huge (5)

Table 3.1: The 12 Scala benchmarks selected for inclusion in the benchmark suite, together with their respective inputs.

of well-known Scala programs to choose from. Programs alone, however, do not make workloads; they also require realistic inputs to operate on. All Scala programs therefore come bundled with a least two and up to six inputs of different sizes. This gives rise to 51 unique workloads, i.e. benchmark-input combinations. The DaCapo benchmark suite offers only 44 such workloads, being limited to at most four input sizes: small, default, large, and huge.

Compared to the DaCapo benchmarks, the larger number of inputs per benchmark gives researchers more flexibility. My Scala benchmark suite is therefore better suited for evaluating novel, input-dependent approaches to optimization [TJZS10], although admittedly the number of inputs provided is still rather small for such an approach.³ That being said, a broader range of input sizes is undeniably useful if the researcher's experimental budget is tight; sometimes, the overhead of profiling becomes so high that gathering a profile becomes infeasible even for the benchmark's default input size. This has been an issue, e.g. for the metrics shown in Section 5.4.8. The extra input sizes made it possible to obtain results for smaller input sizes, where doing so for default input sizes would have resulted in completely unwieldy profiles.

3.2.1 Covered Application Domains

The validity of any experimental finding produced with the help of a benchmark suite hinges on that suite's representativeness. I was thus careful to choose not only a large number of programs, but also programs from a range of application domains. Compared to the DaCapo benchmark suite, the Scala benchmark suite only lacks two application domains covered by its Java counterpart: client/server applications (tomcat, tradebeans, and tradesoap) and in-memory databases (h2). In fact, the former domain made its initial appearance only in version 9.12 of the DaCapo benchmark suite. The earlier version 2006-10 does not cover client/server applications but does cover in-memory databases (hsqldb).

The absence of client/server applications from the Scala benchmark suite is explained by the fact that all three such DaCapo benchmarks depend on either a Servlet container (Apache Tomcat) or an application server (Apache Geronimo). As no Servlet container or application server written in Scala exists yet, any Scala benchmark within this category would depend on a Java-based implementation thereof; this would dilute the Scala nature of the benchmark. In fact, I designed a benchmark based on the popular Lift web framework [Sew10] but had to discard it, since the Java-based container dominated its execution profile; the resulting

³ In their study, Tian et al. [TJZS10] used between 9 and 175 inputs per benchmark, with an average of 51.3.

benchmark was not very representative of Scala code. Likewise, the absence of in-memory databases is explained by the fact that, to the best of my knowledge, no such Scala application 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 Chapter 5, I will thus show that the inclusion of several applications from the same domain is indeed justified; in particular, the respective benchmarks each exhibit a distinct instruction mix (cf. Section 5.4.1).

3.2.2 Code Size

While covering a broad range of domains increases the trust in a benchmark suite's representativeness, it is not enough to make it well-suited for JVM research. The constituent benchmarks must also be of considerable size and complexity, as micro-benchmarks often do not reflect the behaviour of larger real-world applications from a given domain. In this section, I will thus argue that the Scala benchmarks are indeed of significant size and complexity.

For the DaCapo benchmark suite on which the suite is based, Blackburn et al. employ the metrics introduced by Chidamber and Kemerer [CK94] to argue that their suite exhibits “much richer code complexity, class structures, and class hierarchies” [BGH⁺06] than the older SPEC JVM98 benchmark suite [Cor98]. But whether the metrics by Chidamber and Kemerer carry over to a hybrid language like Scala, which combines concepts from object-oriented and functional languages, is still an open question. While the metrics still are technically applicable, as the implementation for the Java language⁴ targets Java bytecode rather than source code, the results would be heavily distorted by the translation strategy of the Java compiler; the connection to the original design, as manifested in the Scala source code, is tenuous at best. Also, comparing the Scala benchmarks with older benchmarks using the same language is not necessary, as there are no predecessors, with the exception of a few micro-benchmarks [Hun11, Pha12].⁵

In the following, I will thus focus on basic but universal metrics of code size, in particular on the number of classes loaded and methods called. Figure 3.1 relates these two for both the Java benchmarks from the DaCapo 9.12 benchmark suite and the Scala benchmarks from the new suite. As can be seen, even relatively simple Scala programs like `scalap`, a classfile viewer akin to `javap`, are made up of

⁴ See <http://www.spinellis.gr/sw/ckjm/>.

⁵ See <http://www.scala-lang.org/node/360>.

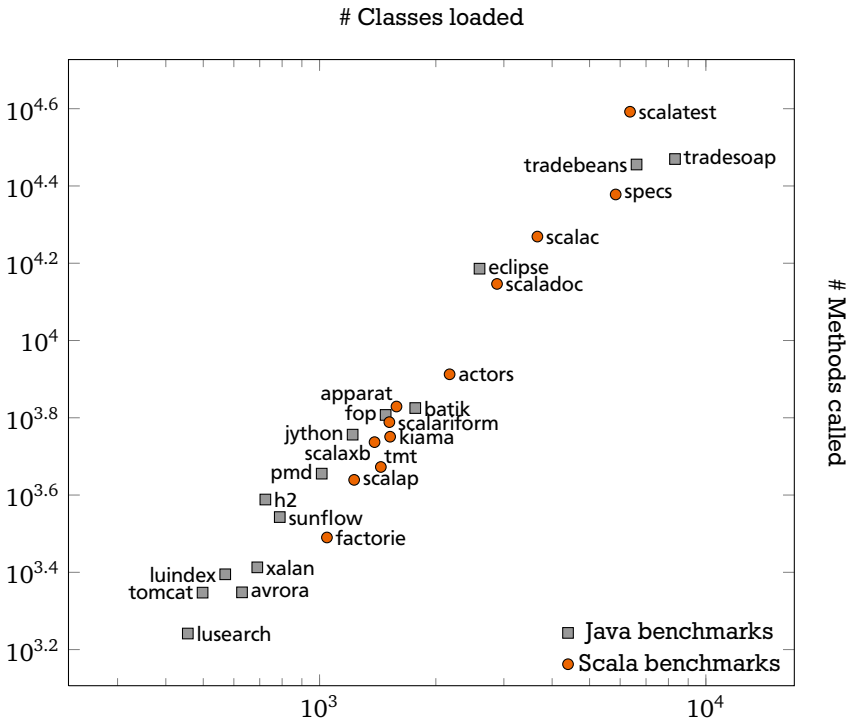


Figure 3.1: Number of classes loaded and methods called at least once by the Java and Scala benchmarks, respectively

thousands of classes and methods: for *scalap*, 1229 classes were loaded and 4357 methods were called at least once. Across all benchmarks, only 4.25 methods per class were, on average, called during the actual benchmark execution. This number is slightly lower for Scala programs (4.14) than for Java programs (4.34). This difference is a consequence of the translation strategy the Scala compiler employs for anonymous functions, which are translated into full-fledged classes containing just a few methods (cf. Section 2.3). This fact may have performance ramifications, as class metadata stored by the JVM can consume a significant amount of memory [OMK⁺10].

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 thusly marked. 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 remaining **final** classes are mostly Singleton classes automatically generated by the Scala compiler (cf. Section 2.3.3).

The methods executed 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.

3.2.3 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. In order to assess to what extent the benchmarks are comprised of Java and Scala code, respectively, all bytecodes loaded by the benchmarks have been categorized 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` source files

Scala Runtime (Scala code). Package `scala`;⁶ `*.scala` source files

Scala application and libraries. Other packages, `*.scala` source files

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

⁶ The package `scala.tools` was excluded; it contains the Scala compiler and the ScalaDoc tool that are used as benchmarks in their own right.

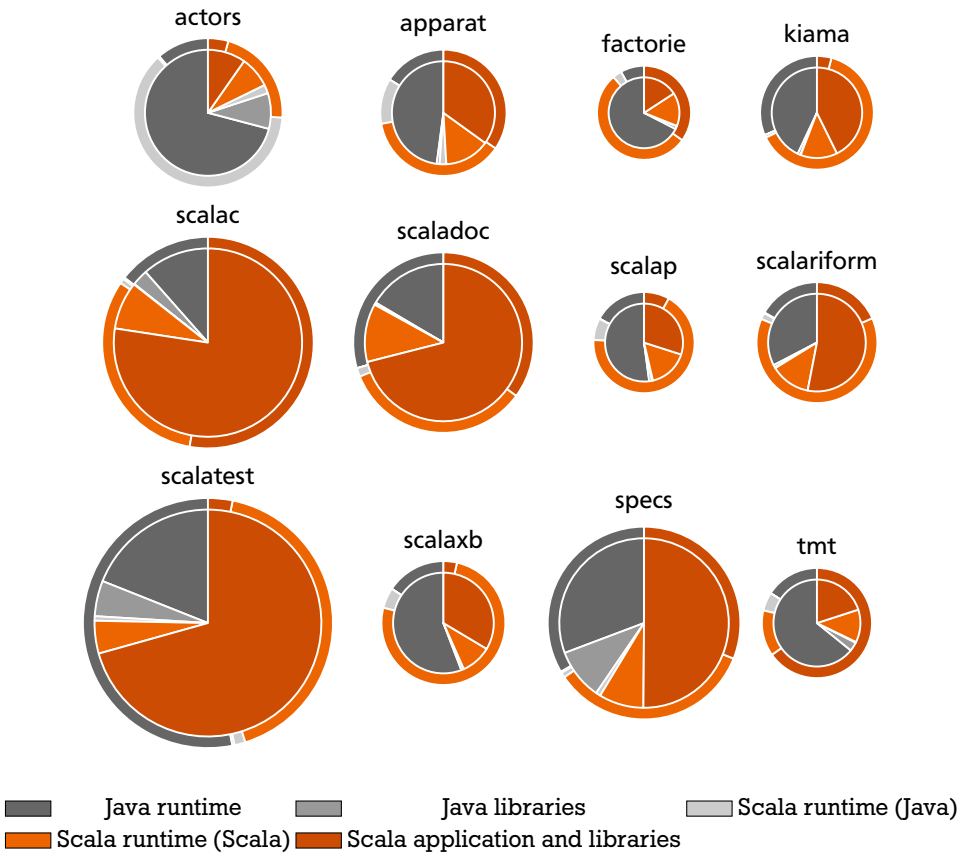


Figure 3.2: Bytecodes loaded and executed by each of the 12 Scala benchmarks (default input size) stemming from the Java runtime, the Java libraries, the part of the Scala runtime written in Java, other parts of the Scala runtime and Scala libraries, or from the Scala application itself

Based on the categorization, the inner circles in Figure 3.2 show how the loaded bytecodes are distributed among the five categories, with the circles' areas indicating 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 indicated 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.

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. Although the volume of test code is high, each test is only executed once and then discarded. This kind of behaviour places the emphasis on the JVM's interpreter or "baseline" just-in-time compiler as well as its class metadata organization. As such, it 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 [KSA09], 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 Chapter 5 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 beyond the scope of Chapter 5, which relies on VM-independent metrics.

3.2.4 The dummy Benchmark

In workload characterization it is often necessary to distinguish the actual workload from any activity occurring during JVM startup, shutdown, or within the benchmark harness. While the harness of the `DaCapo` benchmark suite offers a dedicated callback mechanism which can notify a dynamic analysis of the beginning and end of the actual benchmark iteration, such a callback is sometimes insufficient or at least inconvenient (cf. Section 5.3). The Scala benchmark suite thus ships with an

additional, thirteenth benchmark: dummy. As the name suggests, this benchmark does not perform any work during a benchmark iteration. Consequently, measuring the JVM's behaviour running the dummy benchmark can serve as an indicator of the JVM's activity during JVM startup, shutdown, and within the benchmark harness.

3.3 Choosing a Build Toolchain

The entire benchmark suite is built using Apache Maven,⁷ a build management tool whose basic tenet rings particularly true in the context of a research benchmark suite: build reproducibility. A central artifact repository mirrored many times worldwide contains the (frozen) code of the benchmarked applications. This ensures that the benchmark suite can be built reproducibly in the future, even if some of the applications are later abandoned by their developers.

To ease the development of benchmarks, I have created a dedicated Maven plugin, the `dacapo-benchmark-maven-plugin`. This plugin not only packages a benchmark according to the DaCapo suite's requirements (harness, self-contained dependencies, `.cnf` metadata) but also performs a series of integration tests on the newly-built benchmark. It automatically retrieves but keeps separate all transitive dependencies of the benchmark and its harness. Finally, the plugin automatically generates a report providing summary information about a given benchmark and its inputs. Figure 3.3 shows one such report for the `scalac` benchmark from the Scala benchmark suite. Where necessary, these summary reports are accompanied by further information on the project's website, e.g. on the selection criteria for the inputs used.

Just as the benchmark suite, the Maven plugin is Open Source and freely available for download.⁸

⁷ See <http://maven.apache.org/>.

⁸ See <http://www.plugins.scalabench.org/modules/dacapo-benchmark-maven-plugin/>.

scalac Benchmark

Last Published: 2012-02-16 | Version: 0.1.0-SNAPSHOT | TU Darmstadt | [Home](#) | [Software Technology Group](#) | [Scala Benchmarking Project](#) | [Organizational POM](#) | [Benchmarks](#) | [scalac Benchmark](#) | [Benchmark Configuration](#) | [Dacapo Benchmark Site](#)


Benchmark Configuration

Project Documentation

- Project Information
- Project Details
- Benchmark Configuration**
- Source Xref

Benchmarks

- Dacapo Integration
- Maven Repository
- Maven Plugins

Build by 

Short Description/Name	scalac
Long Description	The compiler for the Scala 2 language
Author	LAMP/EPFL
License	BSD-like
Copyright	Copyright 2002-2010 EPFL, Lausanne
URL	http://www.scala-lang.org/
Version	2.8.1

This benchmark is externally single-threaded.

Input size small

Uses a single thread to drive the workload.

Arguments

`$(SCRA_TCH) scalac src/scalac/foos/scalac/util/StringUtil.scala`

The following is the list of outputs validated by the benchmark harness.

Filename	Test file?	Size	Lines	Digest
<code>scalac/target/scalac/foos/scalac/scala/x/util/StringUtil.class</code>	No	n/a	n/a	8abdc9b443394d42ceeaa80f9d251222e94cd31
<code>scalac/target/scalac/foos/scalac/scala/x/util/StringUtil\$.class</code>	No	n/a	n/a	1d4e6371b0c8bcd7e625504d498159a08f59a5da
<code>slider.log</code>	Yes	n/a	0	n/a

Input size default

Uses a single thread to drive the workload.

Figure 3.3: Report generated by the `dacapo-benchmark-maven-plugin`

4 Rapidly Prototyping Dynamic Analyses

The design of a new benchmark suite suitable for research requires an in-depth analysis of the constituent benchmarks to ensure that they exhibit diverse behaviour (cf. Chapters 3 and 5). A benchmark suite containing only very similar benchmarks not only misrepresents a large fraction of real-world programs that behave dissimilar, but wastes valuable resources; researchers perform their time-consuming experiments on many benchmarks where few would suffice. It is thus of paramount importance to subject the benchmarks to a variety of dynamic analyses to ensure that they cover a wide range of behaviour.

Alas, developing such analyses is costly. This is all the more a problem as the design of a new benchmark suite often proceeds in an iterative, exploratory fashion; a candidate benchmark is selected and subjected to analysis, whose outcome suggests a new candidate or a new metric for which no analysis exists yet. This happened, for example, when I evaluated a candidate benchmark based on the Lift web framework; this benchmark had to be rejected as measurements showed too large an impact of its dependencies, which were written in Java (cf. Section 3.2.3).

In this chapter, I will describe the approaches I have taken to rapidly prototype the necessary dynamic analyses. Section 4.1 discusses these three approaches, all of which lessen the researcher's burden when designing a benchmark suite: re-using dedicated profilers (Section 4.1.1), re-purposing existing tools (Section 4.1.2), and developing tailored profilers in a domain-specific language (Section 4.1.3). Section 4.2 concludes this chapter with a brief discussion of the three case-studies.

Parts of this chapter have been published before:

- Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. Taming reflection (extended version): Static analysis in the presence of reflection and custom class loaders. Technical Report TUD-CS-2010-0066, CASED, 2010
- Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela 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. doi:[10.1145/1985793.1985827](https://doi.org/10.1145/1985793.1985827)
- Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and platform-independent calling context profiling for the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 279(1):61–74, 2011. doi:[10.1016/j.entcs.2011.11.006](https://doi.org/10.1016/j.entcs.2011.11.006)

-
- Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, Martin Schoeberl, and Mira Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java Virtual Machine. In *Proceedings of the 9th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011. doi:10.1145/2093157.2093160
 - Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 2012. doi:10.1016/j.scico.2011.11.003
 - Yudi Zheng, Danilo Ansaloni, Lukas Marek, Andreas Sewe, Walter Binder, Alex Villazón, Petr Tuma, Zhengwei Qi, and Mira Mezini. Turbo DiSL: Partial evaluation for high-level bytecode instrumentation. In Carlo Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, volume 7305 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin / Heidelberg, 2012. doi:10.1007/978-3-642-30561-0_24
 - Danilo Ansaloni, Walter Binder, Christoph Bockisch, Eric Bodden, Kardelen Hatun, Lukáš Marek, Zhengwei Qi, Aibek Sarimbekov, Andreas Sewe, Petr Tůma, and Yudi Zheng. Challenges for refinement and composition of instrumentations: Position paper. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 86–96. Springer Berlin / Heidelberg, 2012. doi:10.1007/978-3-642-30564-1_6

4.1 Approaches

During benchmark-suite design, a multitude of metrics are needed to shed light on features of the benchmarks as diverse as their use of polymorphism (Section 5.4.2), their use of reflective method calls (Section 5.4.6), or their use of object allocations (Section 5.4.8). Most of these metrics are dynamic in nature, i.e. they cannot be computed based on the benchmark’s source- or bytecode alone. But implementing dynamic analyses (often called profilers) to compute the desired metrics is a challenging task. Obviously, accuracy is of high priority, although it sometimes is sufficient to not be completely accurate—if one is aware of the resulting threats to validity and they are well-understood. Often, performance is also a priority; while some overhead incurred by profiling is acceptable during benchmark design, it must not be prohibitively high, i.e. orders of magnitude higher than the actual

benchmark's execution time. Such exorbitant execution times (or memory requirements) would make it impossible to analyze the benchmark in a reasonable time frame.

In this chapter I discuss the three approaches to prototyping dynamic analyses for metric computation used by me during the design of the Scala benchmark suite and contrast it with a fourth one, namely developing profilers from scratch.

4.1.1 Re-using Dedicated Profilers: JP2

Re-using an existing profiler is an attractive choice, as it frees the benchmark suite's designer from developing a profiler from scratch. Often, however, an existing profiler records data that is close to but not exactly what is needed to compute the desired metrics. It is thus paramount that the profiler produces a detailed log from which the desired metrics can be derived, possibly offline, i.e. after the actual profiling run is over.

Case Study

JP2 [SMB⁺11, SSB⁺11, SSB⁺12] is one such profiler, which produces complete and accurate calling-context profiles on any production JVM. In the resulting calling-context profile individual calls are distinguished by the context they occur in, i.e. by the entire call stack that led to the call in question. JP2 thus produces an extremely detailed view of the program's inter-procedural control flow. Moreover and in contrast to its predecessor, the Java Profiler JP [BHMV09], JP2 can keep track of individual call sites [SSB⁺12] and the execution counts of individual basic blocks [SSB⁺11].¹ All this information is kept in a calling-context tree (CCT) [ABL97], an example of which is shown in Figure 4.1.

For this thesis, I modified JP2 by adding a plug-in mechanism that makes it possible to serialize the collected CCT in multiple formats [SSB⁺11]. Moreover, JP2 was modified to store on disk any loaded class files. Both modifications were straight-forward and left the core of JP2 untouched; in particular, its low-level bytecode instrumentation routines did not have to be modified.

Together, the CCT and the class files allowed me to derive various metrics. For example, to derive a benchmark's instruction mix one needs both a basic block's execution count and the list of instructions in that basic block; the former is found in the CCT, whereas the latter is found in the appropriate class file. From this information one can in principle easily derive the benchmark's instruction mix, i.e.

¹ These two extensions [SSB⁺11, SSB⁺12] were developed in a joint effort by Aibek Sarimbekov and the author of this thesis.

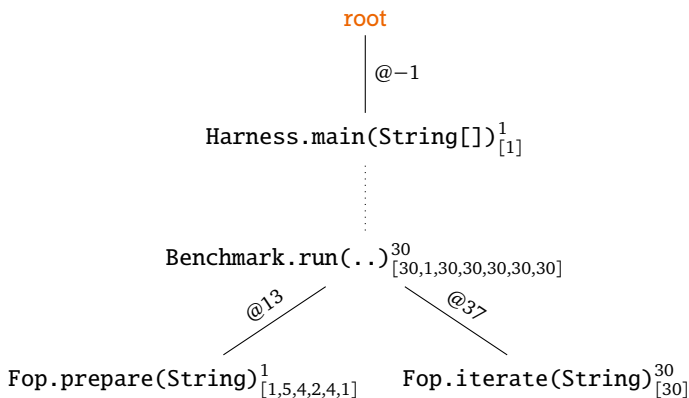


Figure 4.1: Sample calling-context tree, whose nodes keep the methods’ execution count (m), the dynamic execution counts for each basic block ($_{[n_1, n_2, \dots]}$), and the call site ($@i$) at which the corresponding method was invoked in its parent context.

the execution frequencies of all bytecode instructions. But how to do this in practice and with minimal effort?

Ideally, the benchmark designer has a simple, declarative query language at his disposal to describe the desired metrics in. For this case-study, I have chosen XQuery [BCF⁺10], a declarative query language for XML. Of course, this requires that both the CCT and the loaded class files are available in an XML representation. Using an appropriate plug-in, I thus serialized the entire CCT in the simple XML rendition of the calling-context tree exemplified in Figure 4.2. Likewise, I converted all loaded class files into XML by re-using an existing conversion routine from the ASM bytecode engineering library.² While the two resulting representations are verbose, they are easily processable by off-the-shelf XML tools, in particular by an XQuery processor. Metrics can then simply be expressed as declarative queries with respect to these two XML representations.

In the following, I will describe this process in detail for readers unfamiliar with XQuery. Deriving a benchmark’s instruction mix from a JP2 profile will serve as an example. First, the XML representations of the CCT and the loaded class files need to be made available to the XQuery processor. In Listing 4.1, this is done by means of the `$cct` and `$classes` variables. How exactly such external data is passed to

² See <http://asm.ow2.org/>.

```

1 <callingContextTree>
2   ...
3   <method declaringClass="LHarness;"
4     name="main" params="[Ljava/lang/String;" return="V">
5     <executionCount>0</executionCount>
6     <executedInstructions>0</executedInstructions>
7     ...
8     <method declaringClass="Lorg/dacapo/harness/Fop;"
9       name="iterate" params="Ljava/lang/String;" return="V">
10      <executionCount>30</executionCount>
11      <executedInstructions>390</executedInstructions>
12      <callSite instruction="2">
13        ...
14      </callSite>
15      ...
16      <basicBlock start="1" end="13">
17        <executionCount>30</executionCount>
18      </basicBlock>
19    </method>
20    ...
21  </method>
22  ...
23 </callingContextTree>

```

Figure 4.2: Sample output of JP2 using an XML representation of the CCT

the XQuery processor is implementation-defined, but most processors offer a simple command-line interface for this task.

For benchmarking, it is typically desirable to exclude code executed during JVM startup and shutdown from one’s measurements. For this purpose, JP2 offers the option to only profile code in the dynamic extent of a given method. Often, it is sufficient to simply choose the program’s main method. In the case of harnessed benchmarks, however, this would include the harness in the measurements as well, which is undesirable. The DaCapo benchmark harness thus offers a dedicated callback mechanism that makes it possible to exclude not only JVM startup and shutdown but also the benchmark’s harness; only the benchmark itself is measured [SMB⁺11]. While, for technical reasons, code outside the benchmark’s `iterate` method still contributes nodes to the CCT, the nodes’ execution count is

```

1 declare variable $cct-xml external;
2 declare variable $cct := doc($cct-xml)/cct:callingContextTree;
3 declare variable $classes-xml external;
4 declare variable $classes := doc($classes-xml)/classes;
5
6 declare variable $benchmark-nodes :=
7   $cct//cct:method[cct:executionCount > 0];
8
9 declare function asm:method($node) {
10  let $internal-name = asm:internal-name($node/@declaringClass)
11  let $class := $classes/class[@name = $internal-name]
12  let $node-desc :=
13    asm:method-descriptor($node/@params, $node/@return)
14  return
15    $class/method[@name eq $node/@name and @desc eq $node-desc]
16 };
17
18 <instructionMix>{
19  for $instruction in $asm:instruction-list
20  return element {$instruction} {
21    sum(for $node in $benchmark-nodes
22      let $body := $asm:method($node)/asm:instructions(.)
23      for $basic-block in $node/cct:basicBlock
24      let $start := $basic-block/@start
25      let $length :=
26        $basic-block/@end - $basic-block/@start + 1
27      let $instructions := subsequence($body, $start, $length)
28      return count($basic-block/cct:executionCount
29        * $instructions[name() eq $instruction]))
30  }
31 }</instructionMix>

```

Listing 4.1: An XQuery script computing a benchmark’s instruction mix from XML representations of a CCT and the benchmark’s class files

fixed to 0. This property makes it easy to exclude these nodes with XQuery, as the definition of the `$benchmark-nodes` variable in Listing 4.1 shows (Line 6).

Now, for each node in the CCT one needs to identify the corresponding method in the classfiles’ XML representations. Despite the fact that conversions need to be applied to both the declaring class’s name (`asm:internal-name`) and the

method's parameter and return types (`asm:method-descriptor`), the actual mapping is straight-forward, as illustrated by the `asm:method` function (Line 9). Once a node in the CCT, containing a profile of that method's execution, can be mapped to its instructions as stored in the class file, the instruction mix can be easily derived and output in an XML representation, with `<instructionMix>` as root element.

Note how XQuery allows one to specify the desired metric in an entirely declarative fashion: An instruction's execution count is the sum of that instruction's execution counts per basic block, which are in turn the execution counts of the basic blocks themselves multiplied with the number of the occurrences of the instruction in question within a given basic block.

Due to the declarative nature of the query language, the benchmark suite's designer need not be concerned with efficiency of computation at this stage but can focus on clarity instead. For example, it is perfectly natural to view the instruction mix as simply a list of all instructions' execution counts and write the query accordingly. It is then up to the XQuery processor to discover that it need not traverse the CCT multiple times³ but can do with only a single traversal.

Note that the accuracy of this offline-processing step only depends on the accuracy of its inputs, in particular, on the accuracy of the profiles produced by JP2. But as JP2 indeed produces a very detailed profile, this is not an issue; in fact, none of the metrics presented in Section 5.4 required more than two levels of calling context. In other words, a call-graph-based profile would have been sufficient. This is the main drawback of re-using existing profilers: They may record more information than is necessary and thereby incur higher overhead than is feasible during the exploratory stage of benchmark-suite design; profiling runs that take days rather than hours or minutes have the potential to tremendously slow down development. Nevertheless, the ability to use a declarative query language to prototype the metrics makes up for this overhead; while the computational cost increases, development cost decreases.

4.1.2 Re-purposing Existing Tools: TamiFlex

While dedicated profilers come immediately to mind when one wants to perform workload characterization, other tools may offer similar features, despite having been developed for a different purpose. If the output of these tools contains the desired information, the programming effort necessary for re-purposing the tool as a dynamic analysis is low; all that is required is to extract the information from the tool's outputs using the (scripting) language or tool of choice.

³ In this example: 156 times. (ASM distinguishes between 156 different bytecode instructions.)

Case Study

In this section, I briefly describe my experience with one such tool, namely TamiFlex [BSSM10, BSS⁺11]. TamiFlex was originally conceived to increase the soundness of static analyses in the presence of language features like reflection and dynamic class loading. To achieve this goal, TamiFlex follows a very pragmatic approach of logging both reflective calls made and classes loaded dynamically by the running program. This is done by a Java agent called the Play-Out Agent. There also exists a corresponding Play-In Agent, which makes it possible to re-insert offline-transformed classes into another run of the program. Finally, these two online components are complemented by an offline component called the Booster [BSS⁺11], which can transform the logged classfiles to make them usable by any static analysis, even one that is unaware of TamiFlex's logs.

Figure 4.3 shows a simplified version of Tamiflex's overall architecture, excluding the Booster. This figure also highlights those parts of the tool that I re-purposed to collect information about the reflective calls made by the benchmarks under study (cf. Section 5.4.6). As can be seen, only a small part of TamiFlex, the Play-Out Agent, is needed for this purpose. The resulting profiles (`refl.log`) are stored in a simple, textual format, of which Figure 4.4 shows an example.

The log includes events like reflective method invocations and object instantiations, as well as other operations like dynamic class loading or introspection. While it is easy to derive the desired information on, e.g., the reflective calls made from such a log, this example also highlights a limitation of re-purposing existing tools: Some desirable information may be either incomplete or missing completely from the produced profiles. While the log in Figure 4.4 does identify a reflective call's call site (method `org.dacapo.harness.Fop.iterate`, line 41), this information is missing two details: Line numbers do not unambiguously identify a call site and methods are not unambiguously identified using their name alone (lacking their descriptor). Depending on the metric in question, the resulting loss of accuracy may or may not be acceptable. For the metrics I present in Section 5.4.6, this loss of accuracy was indeed minor; reflective call sites are rare in real-world code and multiple reflective call sites sharing a single source line are rarer still. Together with the line number a method's name captures the call site well.

After the online stage of profiling, i.e. running the benchmark together with the Play-Out Agent, the reflection log can be processed using just a few lines of code. As TamiFlex serializes its log in a simple, record-based format, processing it with a scripting language like AWK [AKW88], whose built-in pattern matching capability is tailored to record-oriented data, is straight-forward. The metrics can thus be computed in a semi-declarative fashion; pattern matching the input records is declarative, but aggregating the relevant log entries is not.

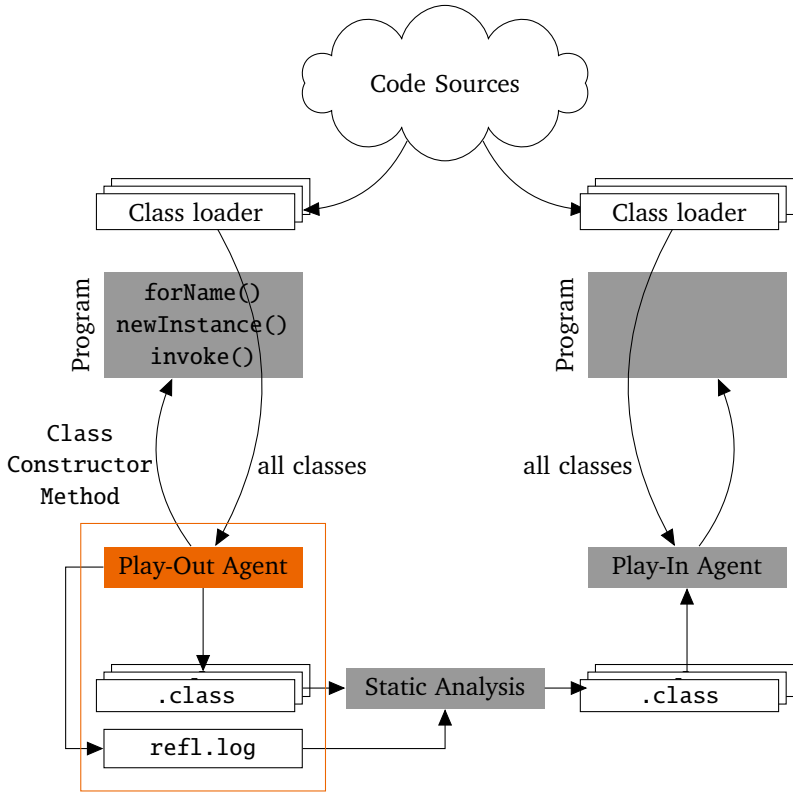


Figure 4.3: The (simplified) architecture of TamiFlex [BSS⁺11] and the parts re-purposed as a dynamic analysis

4.1.3 Developing Tailored Profilers in a DSL: DiSL

Writing a profiler from scratch is often a measure of last resort. Alas, during the exploration stage of benchmark-suite design, this is sometimes necessary when the desired metric requires data not obtainable by re-using existing profilers or by re-purposing existing tools. In such cases, domain-specific languages can alleviate the designer’s burden by both abstracting away from low-level details and by providing a more convenient, expressive syntax to specify the events of interest during the benchmarks’ execution.

```
1 Array.newInstance;¶
2 java.lang.String[];¶
3 java.util.Arrays.copyOf;2245;;
4 Class.forName;¶
5 org.dacapo.harness.Fop;¶
6 org.dacapo.harness.TestHarness.findClass;281;;
7 Constructor.newInstance;¶
8 <org.dacapo.harness.Fop: void <init>(....Config,java.io.File)>;¶
9 org.dacapo.harness.TestHarness.runBenchmark;211
10 Method.getName;¶
11 <org.apache.fop.cli.Main: void main(java.lang.String[])>;¶
12 java.lang.Class.searchMethods;
13 Method.invoke;¶
14 <org.apache.fop.cli.Main: void startFOP(java.lang.String[])>;¶
15 org.dacapo.harness.Fop.iterate;41
```

Figure 4.4: Sample output of TamiFlex (refl.log)

Case Study

In the domain of dynamic program instrumentation, DiSL [MVZ⁺12]⁴ is one such language. I successfully applied DiSL to a whole family of dynamic analyses which record per-object profiles. To do so, such analyses typically maintain a so-called shadow heap; for each “regular” object allocated on the benchmark’s heap, another object is allocated on the shadow heap. Now, correctly maintaining a shadow heap is not trivial: First, all object allocations (including ones of multi-dimensional arrays and reflective ones) need to be instrumented. Second, the shadow heap needs to be properly isolated from the benchmark’s regular heap; shadow objects should only be allocated for regular objects but not for other shadow objects. Moreover, shadow objects should not keep regular objects alive longer than necessary.

To spare other developers this effort, I used DiSL to encapsulate the handling of the shadow heap in a re-usable library.⁵ Essentially, this library consists of two parts: the instrumentation logic and the runtime classes. First, I will discuss the instrumentation logic sketched in Listing 4.2. DiSL allows one to specify instrumentation tasks in (a subset of) plain Java. Each method hereby constitutes a so-called snippet [MVZ⁺12, Section 3.1], which is executed after every successful object al-

⁴ See <http://dag.inf.usi.ch/projects/disl/>.

⁵ See <http://www.disl.scalabench.org/modules/shadowheap-disl-analysis/>.

```

1 public class ShadowHeapInstrumentation {
2
3     @AfterReturning(marker=BytecodeMarker.class,
4         args="new,newarray,newarray")
5     public static void objectAllocated(DynamicContext dc,
6         AllocationSiteStaticContext sc) {
7         Object obj = dc.getStackValue(0, Object.class);
8         String site = sc.getAllocationSite();
9         ShadowHeapRuntime.get().objectAllocated(obj, site);
10    }
11
12    @AfterReturning(marker=BytecodeMarker.class,
13        args="multianewarray")
14    public static void multiArrayAllocated(DynamicContext dc,
15        AllocationSiteStaticContext sc) {
16        Object multiarray = dc.getStackValue(0, Object.class);
17        String site = sc.getAllocationSite();
18        ShadowHeapRuntime.get().multiArrayAllocated(multiarray, site);
19    }
20
21    @AfterReturning(marker = BodyMarker.class,
22        scope = "Object Constructor.newInstance(Object[])")
23    public static void objectAllocatedReflectively(DynamicContext dc,
24        AllocationSiteStaticContext sc) {
25        Object obj = dc.getStackValue(0, Object.class);
26        String site = sc.getReflectiveAllocationSite();
27        ShadowHeapRuntime.get().objectAllocated(obj, site);
28    }
29    ...
30 }

```

Listing 4.2: DiSL class instrumenting object allocations

location (`@AfterReturning`). Within the snippet, contexts give access to both the newly allocated object on the operand stack (`DynamicContext`) and to a unique identifier for the allocation site in question (`AllocationSiteStaticContext`). While the dynamic context [MVZ⁺12, Section 3.6] is built into DiSL, the static context [MVZ⁺12, Section 3.5] is tailored to the analysis task at hand. Once the context information is available, control is passed from the snippet to the analysis'

```

1 public abstract class ShadowHeapRuntime<T extends ShadowObject> {
2
3     public abstract T createShadowObject(Object obj, String site);
4
5     public T getShadowObject(Object obj) {
6         return shadowHeap.getUnchecked(obj);
7     }
8
9     public void objectAllocated(Object obj, String site) {
10        T shadowObj = createShadowObject(obj, site);
11        shadowHeap.put(obj, shadowObj);
12    }
13
14    public void multiArrayAllocated(Object array, String site) {
15        if (array.getClass().getComponentType().isArray()) {
16            for (int i = 0; i < Array.getLength(array); i++) {
17                if (Array.get(array, i) != null)
18                    multiArrayAllocated(Array.get(array, i), site);
19            }
20        }
21        objectAllocated(array, allocationSite);
22    }
23    ...
24 }

```

Listing 4.3: Runtime class managing the shadow heap

runtime class (`ShadowHeapRuntime`) or more precisely to a Singleton thereof. I will discuss this class, shown in Listing 4.3, next.

The runtime class's primary responsibility is to maintain a shadow heap that is isolated from the benchmark's regular heap. This can be achieved through the careful use of weak references in an appropriate, map-like data structure (not shown). As the precise nature of the shadow objects varies from analysis to analysis, depending on the profile recorded per object, a Factory Method (`createShadowObject`) is responsible for creating the shadow objects. While interaction with the Singleton pattern is not totally seamless [ABB⁺12], this design allows for re-use of the shadow-heap functionality by other analyses. Moreover, these concrete analyses need not concern themselves with the JVM's rather complex handling of multi-

```

1 public class HashCodeInstrumentation {
2
3     @Before(marker=BytecodeMarker.class, guard=ObjectHashCode.class,
4             args="invokevirtual,invokeinterface")
5     public static void hashCodeCalled(DynamicContext dc) {
6         Object obj = dc.getStackValue(0, Object.class);
7         HashCodeRuntime.get().hashCodeCalled(obj);
8     }
9
10    @Before(marker=BytecodeMarker.class, guard=ObjectHashCode.class,
11            args="invokespecial")
12    public static void superHashCodeCalled(DynamicContext dc,
13        InvocationStaticContext sc, ClassContext cc) {
14        Object obj = dc.getStackValue(0, Object.class);
15        Class superClass = cc.asClass(sc.thisClassSuperName());
16        HashCodeRuntime.get().superHashCodeCalled(obj, superClass);
17    }
18
19    @Before(marker=BytecodeMarker.class,
20            guard=SystemIdentityHashCode.class, args="invokestatic")
21    public static void identityHashCodeCalled(DynamicContext dc) {
22        Object obj = dc.getStackValue(0, Object.class);
23        HashCodeRuntime.get().identityHashCodeCalled(obj);
24    }
25 }

```

Listing 4.4: DiSL class instrumenting hash-code calls

dimensional arrays; each subarray is automatically added to the shadow heap as a shadow object of its own (Lines 14–22).

One concrete analysis that I implemented records how often an object’s hash-code is computed (cf. Section 5.4.15). This of course necessitates that further events are covered by the instrumentation, namely the calls to `Object.hashCode()` and `System.identityHashCode(Object)`. As Listing 4.4 shows, the resulting instrumentation logic is simple and need not concern itself with object allocations at all. Simple guards [MVZ⁺12, Section 3.8] select the desired method calls (`IdentityHashCode`, `ObjectHashCode`). Like in Listing 4.2, the snippets primarily pass context information to the runtime. The only challenge in implementing these snippets lies in correctly handling super-calls; from an **invokespecial**

instruction alone it is impossible⁶ to determine the exact target method that will be invoked at runtime. Thus, information about the super-class of the currently instrumented class needs to be passed on to the runtime, shown in Listing 4.5, as well. The shadow object implementation can then use this information to determine which implementation of `Object.hashCode()` was meant, i.e. at which level in the class hierarchy this implementation resides (`getHashCodeLevel`). At present, this requires the use of reflection. Unfortunately, reflective class introspection may trigger further class loading; thus, the profiler could potentially interfere with the benchmark's execution. I have not, however, observed this in practice.

Now, the shadow object's representation shown in Listing 4.5 becomes quite simple; it is a simple wrapper around a backing array of counters. Other dynamic analyses developed as part of this thesis (cf. Sections 5.4.11–5.4.12), however, keep track of information for each of the shadowed object's fields. As in the case of the abstract runtime class, which leaves the concrete class of shadow objects unspecified through the use of the Factory Method pattern, I also provide an abstract shadow object class, which handles the shadow fields in an abstract manner; again, a Factory Method is responsible for creating the concrete objects, which keep track of accesses to the shadowed field in question.

Based on my positive experience with TamiFlex's record-based log-file format, all tailored profilers I implemented use a similar format (tab-separated values). This makes processing the profiles with AWK [AKW88] straight-forward again.

4.2 Discussion

In this chapter, I have described my experiences with three approaches to rapidly prototype the dynamic analyses required while designing a new benchmark suite. Table 4.1 summarizes these experiences. I now contrast the three case-studies with an approach using profilers written from scratch. While I have not conducted an explicit case study, I here draw from experiences in the development of JP2, TamiFlex, DiSL, and also Elephant Tracks [RGM11] to provide the data for Table 4.1's last column. Note, however, that none of these four tools was developed from scratch and specifically for the purpose of this thesis. In particular, Elephant Tracks played a role similar to JP2; I simply re-used the profiler by applying offline analyses to the profiles produced by it.

To assess the programming effort required by the different approaches, one has to distinguish between customizing the online and offline stages of the profiling

⁶ In pre-Java 1.0 days, an `invokespecial` instruction specified the target method statically; the shown instrumentation does not support this obsolete behaviour (`ACC_SUPER` flag unset).

```

1 public class HashCodeRuntime<HashCodeSO> {
2
3     public HashCodeSO createShadowObject(Object obj, String site) {
4         return new HashCodeShadowObject(obj, site);
5     }
6
7     public void hashCodeCalled(Object obj) {
8         if (obj != null && isSaveToProcessEvent())
9             getShadowObject(obj).hashCodeCalled();
10    }
11
12    public void superHashCodeCalled(Object obj, Class<?> clazz) {
13        if (obj != null && isSaveToProcessEvent())
14            getShadowObject(obj).superHashCodeCalled(clazz);
15    }
16    ...
17 }
18
19 public class HashCodeSO implements ShadowObject {
20
21     private String className, site;
22     private int[] hashCodeCalls;
23
24     public synchronized void hashCodeCalled() {
25         hashCodeCalls[hashCodeCalls.length - 1]++;
26     }
27
28     public synchronized void superHashCodeCalled(Class<?> bound) {
29         hashCodeCalls[getHashCodeLevel(bound)]++;
30     }
31     ...
32 }

```

Listing 4.5: Runtime classes keeping track of per-object hash-code calls

process. For the former stage, the language choice is limited to the implementation language⁷ of the profiler itself, which is typically Java or C++, if such a customization is possible at all. For the latter stage, the language choice is non-

⁷ Of course, using, e.g., Scala to customize a profiler written in Java is perfectly feasible.

	Re-using Profilers	Re-purposing Existing Tools	Developing Profilers in a DSL from scratch	
Case study	JP2	TamiFlex	DiSL	—
LoC (online)	10 to 100	n/a	100 to 1000	1000 to 10 000
Language	Java	n/a	Java + DiSL	Java / C++
Declarativeness	imperative	n/a	semi-decl.	imperative
LoC (offline)	10 to 100	1 to 10	1 to 10	<i>depends</i>
Language	XQuery	AWK	AWK	<i>depends</i>
Declarativeness	declarative	semi-decl.	semi-decl.	<i>depends</i>
Customizabilty	low	low	high	high

Table 4.1: Approaches to prototyping dynamic analyses and their trade-offs

mally unlimited, as all profilers produce profiles in either textual or binary formats, which can then be processed offline. It is thus often possible to use languages which are at least semi- if not fully declarative, thereby reducing the lines of code (LoC) needed to express the desired metrics. Depending on the intrinsic complexity of the recorded data, languages like AWK [AKW88] or XQuery [BCF⁺10] are good choices. AWK excels in handling simple, record-based data, whereas XQuery is needed if the data’s intrinsic structure (e.g. calling-context trees or classfiles) is more complex. This also explains the slightly larger amount of offline-processing code that had to be written in the JP2 case study compared to the TamiFlex and DiSL case studies.

5 A Comparison of Java and Scala Benchmarks Using VM-independent Metrics

In Chapter 3, I have described the Scala benchmark suite devised for this thesis [SMSB11]. In this chapter, I will compare and contrast it with a popular and well-studied Java benchmark suite, namely the DaCapo 9.12 suite [BGH⁺06]. First, Section 5.1 stresses the importance of VM-independent metrics for such an endeavour. This section will also motivate the use of dynamic rather than static metrics. Next, Section 5.2 describes the profilers used to collect these dynamic metrics before Section 5.3 describes any limitations that might threaten the validity of my findings. The main part of this chapter, Section 5.4, then presents the results of the individual analyses. A summary of the findings concludes this chapter.

Parts of this chapter have been published before:

- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048118
- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2012. doi:10.1145/2258996.2259010

5.1 The Argument for VM-independent, Dynamic Metrics

Unlike several other researchers [BGH⁺06, EGDB03], I refrain entirely from using both the JVM's built-in profiling logic and the hardware's performance counters to characterize the benchmark suites' workloads. Instead, I rely exclusively on metrics which are independent of both the specific JVM and architecture, as it has been shown that the JVM used can have a large impact on the results, in particular for short-running benchmarks [EGDB03]. Another, pragmatic reason for using bytecode-based metrics in particular, which are by their very nature VM-independent, is that doing so does not obscure the contribution of the source language and its compiler as much as using metrics based on machine-code would;

the JVM's just-in-time compiler therefore cannot influence the measurements' results. Also, none of the dynamic analyses employed is based on sampling, as is often the case for a JVM's built-in profiling logic [AFG⁺00]; instead, the profilers I use capture the relevant events in their entirety. This avoids any measurement bias introduced by the choice of sampling interval [MDHS10].

All metrics employed in this chapter are dynamic ones, i.e. they were computed from an actual execution of the benchmark in question. Just like the use of VM-independent metrics, this choice is motivated by the desire to eliminate all influences of any particular implementation, be it a JVM or static analysis, on the measurement results. Nevertheless, the metrics have been chosen carefully such that the results are relevant to the optimizations performed by modern JVMs; thus, they can guide implementers towards JVMs that handle both Java and Scala well.

One has to keep in mind, however, that dynamic metrics can hint at optimization opportunities which any concrete JVM may be unable to exploit. If my measurements, e.g., indicate that a call site targets a single method only (cf. Section 5.4.2) and is thus suitable for inlining, it simply means that the call site had just a single target method during the observed benchmark execution; the call site in question may or may not be monomorphic during other executions. Also, a concrete JVM may or may not be able to infer that the call site is de-facto monomorphic due to the inevitable limitations of the static analyses employed by its just-in-time compiler. My results therefore mark the upper bound of what a JVM can infer.

5.2 Profilers

In this section, I will briefly describe the different profilers chosen for gathering the data presented in Section 5.4, in particular with respect to any inaccuracies these choices entail. Not only are the measured metrics VM-independent, but the profilers themselves are also independent from any particular JVM; in particular, they require no modifications to the JVM itself. All measurements can be conducted on any standard Java-6-compatible JVM and thus easily be reproduced by others.

TamiFlex

I have re-purposed the TamiFlex tool [BSSM10, BSS⁺11] (cf. Section 4.1.2) to gather information about the use of reflection by the benchmarks (Section 5.4.6). The profiles produced by TamiFlex itself are entirely accurate in that they include any use of reflection, including reflection used during VM startup and shutdown as well as by the benchmark harness itself. While such use undoubtedly exists,¹ the

¹ In fact, the use of reflection by the benchmark harness of the latest DaCapo release was the primary motivation for the creation of TamiFlex.

number of calls and accesses, respectively, is small and does not dilute the measurements. There is one minor inaccuracy, however, which may affect the metrics presented in Section 5.4.6: Call sites are identified by line number rather than instruction offset; this may exaggerate the number of receiver objects a call site sees, if multiple call sites are accidentally conflated.

JP2

I used the appropriately-modified JP2 calling-context profiler (cf. Section 4.1.1) to collect calling-context trees [ABL97] decorated with the execution frequencies of the respective methods' basic blocks. The rich profiles collected by the call-site aware JP2 make it possible to derive most code-related metrics from a single, consistent set of measurements: instruction mix (Section 5.4.1), call-site polymorphism (Section 5.4.2), argument passing (Section 5.4.4), method and basic block hotness (Section 5.4.5), and the use of boxed types (Section 5.4.7).

A specially-written callback [SMB⁺11] 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's harness. This methodology guarantees that the results are not diluted by startup and shutdown code [DDHV03]. Also, the Scala benchmarks remain undiluted by Java code used only in the harness. Some Scala benchmarks still execute a significant amount of Java code, though (cf. Section 3.2.3).

Elephant Tracks

The existing Elephant Tracks profiler [RGM11]² has been used to collect garbage-collection traces from which many memory-related metrics can be derived: general garbage-collector workload (Section 5.4.8), object churn (Section 5.4.9), and object sizes (Section 5.4.10). Elephant Tracks implements the Merlin algorithm [HBM⁺06] to produce exact garbage collection traces. The profiler itself relies on a combination of bytecode instrumentation and JVMTI (JVM Tool Interface) callbacks to maintain a shadow heap on which the Merlin algorithm then operates. Bytecode instrumentation is done in a separate JVM process; thus, the instrumentation activity does not disturb the execution of the analyzed program.

The traces produced by Elephant Tracks have been analyzed offline using a Garbage Collection simulation framework (GC Simulator). The GC simulator makes it possible to derive the theoretical garbage-collector workload of a particular program run independently of any implementation decisions that might be made in an actual garbage collector. Unfortunately, the use of Elephant Tracks is quite heavy-weight; for some benchmarks, the resulting traces occupy hundreds of

² See <http://www.cs.tufts.edu/research/redline/elephantTracks/>.

gibibytes, even in compressed form. In one case (the `tmt Scala` benchmark) this required me to reduce the benchmark's input size.

Tailored metrics written in DiSL

I complemented Elephant Tracks with several light-weight analyses written in DiSL [MVZ⁺12], a domain-specific language for bytecode instrumentation, to compute metrics not derivable from the traces produced by that profiler: immutability (Section 5.4.11), zero initialization (Section 5.4.12), sharing (Section 5.4.13), synchronization (Section 5.4.14), and the use of identity-hashcodes (Section 5.4.15). While Elephant Tracks is an existing memory profiler, the DiSL-based dynamic analyses were specifically written for the purpose.

DiSL allows rapid development of efficient dynamic program analysis tools with complete code-coverage, i.e. the resulting profilers can instrument all executed bytecode instructions, including those from the Java runtime library. Like Elephant Tracks, DiSLs implementation performs bytecode instrumentation in a separate JVM. Unlike Elephant Tracks, however, the custom analyses I developed maintain their data structures in the JVM process under evaluation. This approach requires to exclude the so-called reference-handler thread, which the JVM uses to process weak references, from the analyses.

5.3 Threats to Validity

Choice of Workloads

The benchmarks from the DaCapo benchmark suite [BGH⁺06], which represent real-world “Java programs,” have been widely accepted by researchers in diverse communities, from the memory-management community to the static analysis community. My novel Scala benchmark suite [SMSB11] is much younger than its Java counterpart, having been publicly released in April 2011. It has therefore not yet gained the same degree of acceptance as the DaCapo benchmark suite.

Since a few benchmarks (`tradebeans`, `tradesoap`, and `actors`) contain hard-coded timeouts, they had to be excluded from any measurements where the profiler incurred a large overhead; the overhead triggers the timeout and causes the benchmark to exit prematurely. Another benchmark (`tomcat`) had to be excluded from several measurements because it exhibits a “controlled” stack overflow³ which unfortunately interferes with the instrumentation-based profilers written in DiSL. Finally, one benchmark (`scalatest`) requires that line number information is correctly preserved during bytecode instrumentation. This again interferes with the profilers

³ See http://sourceforge.net/tracker/?func=detail&atid=861957&aid=2934521&group_id=172498.

written in DiSL; the current version of DiSLs instrumentation agent does not handle line numbers correctly. This is not an permanent restriction, though, but merely a consequence of the current state of DiSLs implementation.

For one benchmark (tmt), the overhead incurred by the dynamic analyses was so high to make it infeasible to use the benchmarks' default input size. In this case, which is marked accordingly (tmt^{small}), I used the small input size.

Choice of Scala Version

There is one further threat to validity concerning the Scala benchmarks: All of them are written in Scala 2.8.x. As both the Scala library and the Scala compiler are subject to much faster evolution than their Java counterparts, it is unclear whether all findings carry over to newer versions of Scala.⁴ This should not be seen as a shortcoming of this thesis, though, but as a fruitful direction for future work (cf. Section 8.1) in which to trace the impact of the Scala language's evolution.

Startup/Shutdown

For performance evaluations it is typically undesirable to include the startup and shutdown activity of both JVM and benchmark harness in the actual measurements. The DaCapo benchmark suite thus offers a callback mechanism which allows one to limit measurements to the benchmark iteration itself. But since the DaCapo group designed this mechanism with performance measurements in mind, it is not always the best solution during workload characterization tasks like mine.

Still, all code-related measurements performed with JP2 make use of the callback mechanism (cf. Section 4.1.1) to exclude both VM startup and shutdown as well as the benchmark's harness. The code-related measurements performed using TamiFlex, however, do include startup, shutdown, and the benchmark harness, as TamiFlex, being a tool re-purposed for profiling, does not offer a way to use the benchmark suites' callback mechanism.

In contrast to the code-related measurements, all memory-related measurements do not make use of the callback mechanism. This is not because of technical limitations but because the callback mechanism it is not very suitable for characterizing the memory behaviour of benchmarks: Objects that are used during the benchmark iteration may have been allocated well in advance, i.e. during startup. The measurements are thus based on the overall execution of the JVM process for a single benchmark run. To compensate for the startup and shutdown activities, albeit in an informal way, I present results for the dummy benchmark (cf. Section 3.2.4), which performs no work of its own. This allows a comparison of the results for a complete benchmark run, i.e. startup, iteration, and shutdown, with a run con-

⁴ As of this writing, Scala 2.9.2 is the latest stable version.

sisting of startup and shutdown only. This methodology is similar to the use of an “empty” program by Dufour et al. [DDHV03].

Dynamic Imprecision

In this chapter, I exclusively resort to dynamic metrics (cf. Section 5.1). On the upside, my results are therefore not subject to *static imprecision* [DRS08], as any static analysis is by nature pessimistic. The results are therefore not disturbed by the intricacies of a particular static analysis but directly mirror each benchmark’s behaviour. On the downside, some metrics may optimistically hint at optimization opportunities which only an omniscient just-in-time compiler or garbage collector would be able to exploit. Any real-world compiler or collector itself needs to resort to static analysis and hence tends towards pessimism rather than optimism, although techniques like dynamic deoptimization may alleviate this by permitting optimizations being carried out speculatively [AFG⁺05]. When a particular metric is subject to *dynamic imprecision* [DRS08], i.e. when the profiler cannot deliver entirely accurate results, this is mentioned for the metric in question.

5.4 Results

In the following, I will describe the results I have obtained using VM-independent, dynamic metrics. These results fall into two broad categories: Metrics that describe the structure and execution properties of the code itself [SMSB11] (Sections 5.4.1–5.4.7) and metrics that describe the memory behaviour this code exhibits [SMSB11] (Sections 5.4.8–5.4.15).

5.4.1 Instruction Mix

The instruction mix of a benchmark, i.e. which instructions are executed during its execution and how often, usually provides a first indication whether the given benchmark is “array intensive” or “floating-point intensive.” Moreover, it may exhibit patterns that allow one to distinguish between Scala and Java code based on the respective benchmarks’ instruction mix alone.

To obtain the instruction mix of the 14 Java and 12 Scala benchmarks I have used the JP2 profiler together with a straight-forward analysis [SSB⁺11] written in XQuery [BCF⁺10]. The measurements resulted in precise frequency counts for all 156 instructions present in the JVM instruction set.⁵

⁵ This number is smaller than the actual number of JVM instructions (201); mere abbreviations like, e.g. `aload_0` and `goto` have been treated as `aload` and `goto_w`, respectively. The `wide` modifier (“instruction” 196) is treated similarly.

While related work on workload characterization either considers all 156 bytecode instructions individually [CMS07, DHPW01] or groups them manually [DHPW01, DDHV03], I applied principal component analysis (PCA) [Pea01] to discern meaningful groupings of instructions. This approach avoids both the extreme low-level view of the former works as well as the inherent subjectivity of the latter. Instead, it automatically offers a higher-level view of the instruction mix which is objective rather than being biased by one’s intuition of what groupings might be meaningful (array operations, floating-point operations, etc.).

Based on the measurements with JP2, one can represent each benchmark by a vector X in a 156-dimensional space. The vector’s components X_i are the relative execution frequencies of the individual instructions, i.e. $\sum_{i=1}^{156} X_i = 1$. As such a high-dimensional vector space is hard to comprehend, I applied PCA to reduce the dimensionality of the data. First, each component X_i is standardized to zero mean, i.e. $Y_i = X_i - \bar{X}_i$. It is not standardized to unit variance, though. In other words, I applied 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.

Those principal components which account for a low variance only, i.e. those which do not discriminate the benchmarks well, were discarded. Just 4 components were thus retained and account for 58.9%, 15.3%, 6.4%, and 5.6%, respectively, of the variance present in the data. Together, these four components explain more than 86.2% of the total variance. In contrast, no single discarded component accounts for more than 2.7% of the total variance. Figure 5.1 depicts the resulting loadings, i.e. the weights $a_{ij} \in [-1, +1]$, of the four retained principal components.

These principal components form the basis of the automatic grouping of instructions. This grouping needs some interpretation, though. For the first component, for example, several instructions exhibit a strong positive correlation ($|a_{1j}| > 0.1$) with the first principal component: reference loads (**aload**), method calls (**invokestatic**, **invokevirtual**, ...), and several kinds of method return (**return**, **areturn**, and **ireturn**). Likewise, one group of three instructions exhibits an equally strong negative correlation: integer variable manipulations (**inc**, **load**, and **istore**). The first component may thus be interpreted as contrasting inter-procedural with intra-procedural 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_cmpge` 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 ac-

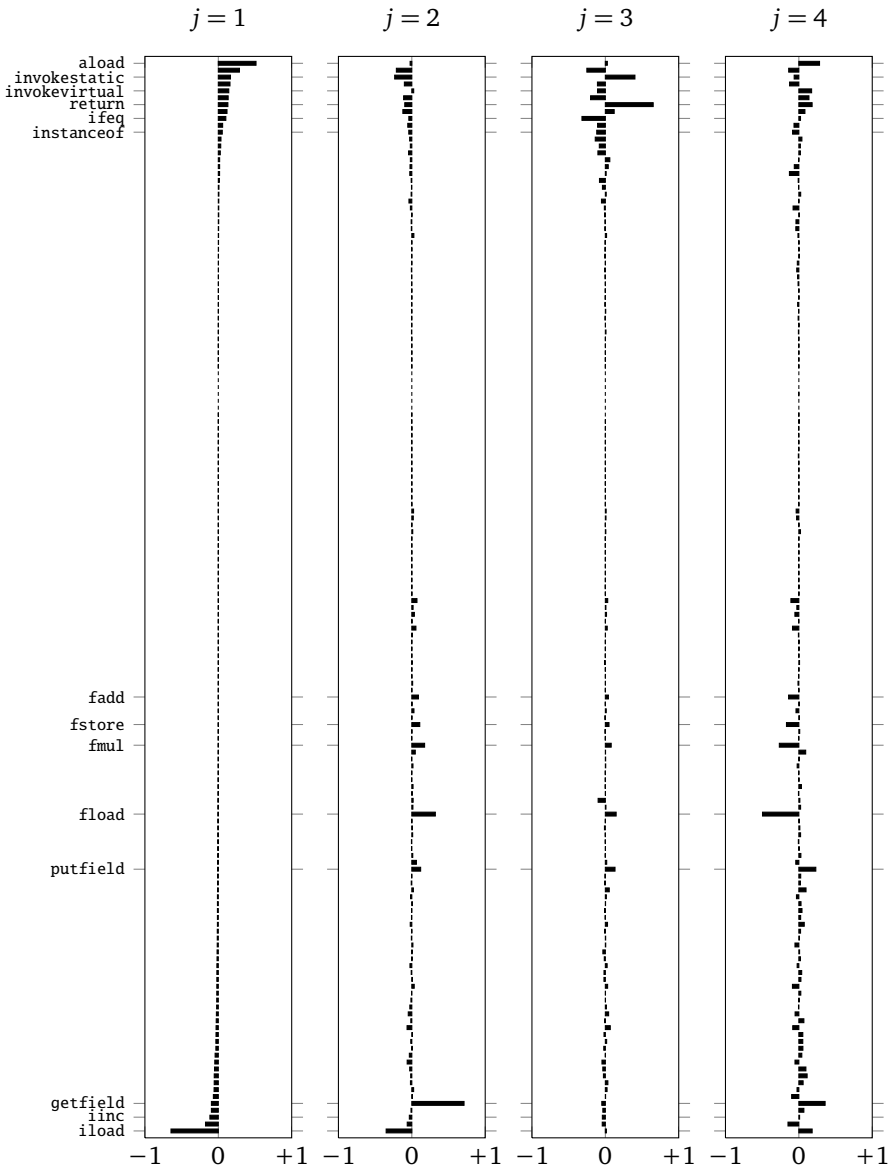


Figure 5.1: The top four principal components that account for 86.2% of variance in the benchmarks' instruction mixes

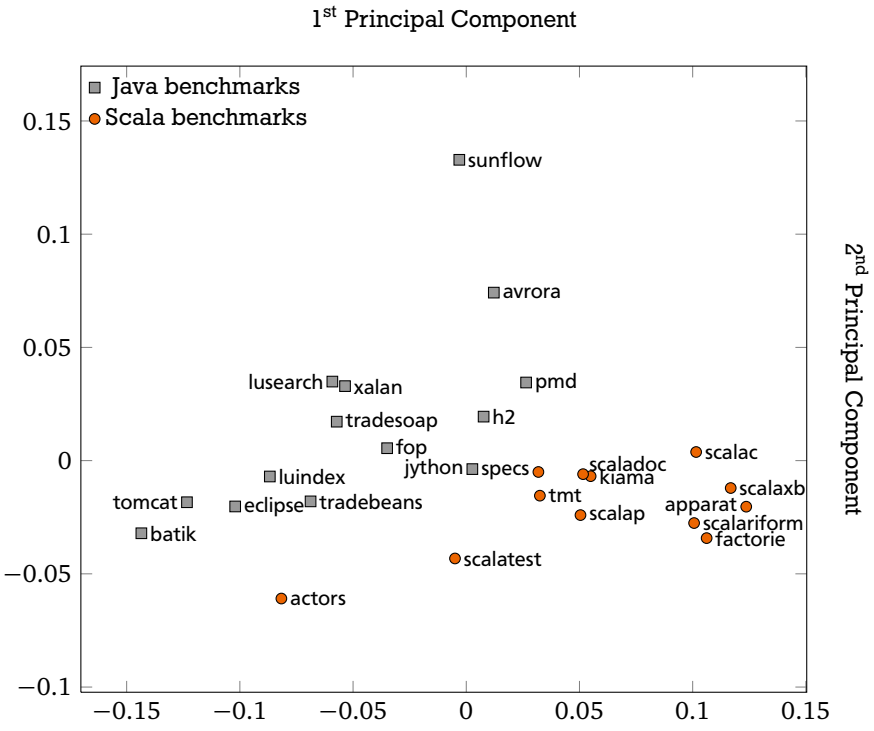


Figure 5.2: The Java and Scala benchmarks with respect to the first and second principal component

cesses (**getfield**, **putfield**), all of which are positively correlated. This may be interpreted as representing both **float**-based numerical computations as well as object-relative field accesses. Note that the **getfield** instruction, e.g., exhibits a small negative correlation with the first component, but a large positive correlation with the second. This is a fundamental property of the groupings produced by PCA; a single instruction may belong, to varying degree, to several groups.

Figure 5.2 shows to what degree both the Scala and Java benchmarks are affected by the first two principal components: All Scala benchmarks, with the exception of **actors**, exhibit high values for the first component, with the **scalatest** benchmark with its heavy usage of the Java runtime (cf. Section 3.2.3) being a borderline case. Likewise, the **actors** benchmark spends a significant portion of execution in those parts of the Scala runtime written in Java. This shows that the Scala

3rd Principal Component

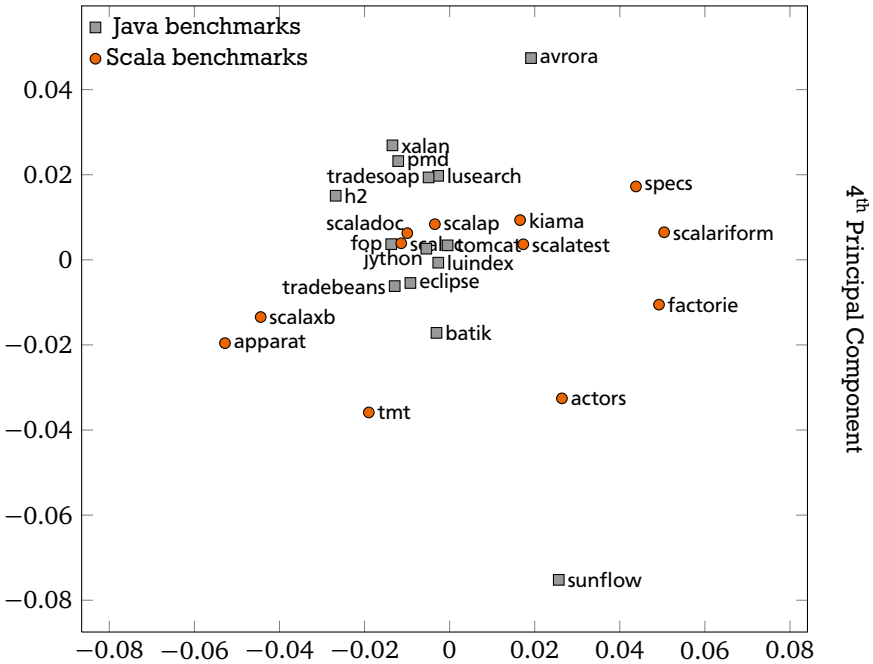


Figure 5.3: The Java and Scala benchmarks with respect to the third and fourth principal component

benchmarks strongly favour inter-procedural over intra-procedural 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 statically dispatched (**invokespecial** and **invokestatic**) and negatively with calls that are dynamically dispatched (**invokevirtual** and **invokeinterface**). Moreover, other forms of dynamic type checks (**checkcast**, **instanceof**) also contribute negatively to this component. These two instructions, together with the **invokevirtual** instruction, are the hallmark of Scala's pattern matching on so-called case classes. The fourth principal component again correlates (negatively) with various floating-point operations, but is otherwise hard to grasp intuitively. This illustrates the prime limitation of a PCA-based approach to analyzing the benchmark's instruction mix; the resulting groups are not always easy to conceptualize. Still, as Figure 5.3

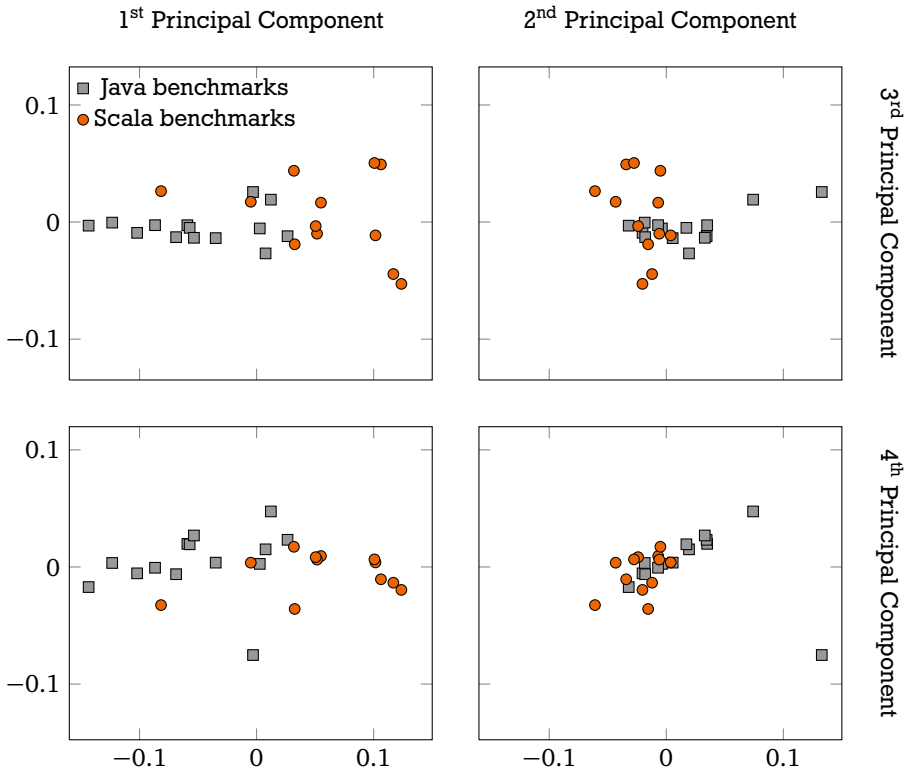


Figure 5.4: The Java and Scala benchmarks with respect to the first four principal components

shows, even such a hard-to-conceptualize component has significant discriminatory power, both with respect to the Java and Scala benchmarks, with the two Java benchmarks *avrora* and *sunflow* marking opposite ends of the spectrum.

What is noteworthy in Figure 5.2 and particularly in Figure 5.3 is that benchmarks like *factorie* and *tmt*, which both stem from the same application domain (here: machine learning) nevertheless exhibit distinctive instruction mixes. This justifies their joint inclusion into the suite.

For the sake of completeness, Figure 5.4 depicts the remaining combinations of components. Note how the plot marks line up on the x-axis for each column and on the y-axis for each row. Also note how the vertical spread of the plot marks

is smaller than the horizontal spread; the third and fourth components (y-axes) exhibit lower variance than the first and second components (x-axes).

5.4.2 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 [DA99a], which is an important optimization (cf. Section 6.4). At the level of Java bytecode, such dynamically-dispatched calls take the form of **invokevirtual** or **invokeinterface** instructions, whereas statically-dispatched calls take the form of **invokespecial** and **invokestatic** instructions. Figures 5.5a and 5.5b contrast the relative occurrences of these instructions in the live part of the Java and Scala 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. There is only a slight shift from **invokevirtual** to **invokeinterface** instructions, which is due to the way the Scala compiler unties inheritance and subtyping [Sch05, Chapter 2] when implementing traits (cf. Section 2.3.1).

But the numbers in these figures provide only a static view. At runtime, the instructions' actual execution frequencies may differ. Figures 5.6a and 5.6b thus show the relative number of actual calls made via the corresponding call sites. Here, the numbers for the Scala and Java benchmarks diverge, with the Scala benchmarks exhibiting a disproportionate amount of calls made by both **invokeinterface** and **invokestatic** instructions. The translation strategy for traits (cf. Section 2.3.1), with its alternating pattern of **invokestatic** and **invokeinterface** instructions, explains much of it. The large number of **invokestatic** calls is additionally explained by the use of Singleton objects in general and by companion objects in particular (cf. Section 2.3.3). 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; Gregg et al. [GPW05] observed only between 0% (Java Grande benchmark suite) and 5.2% (SPEC JVM98) **invokeinterface** calls. Scala code can thus be expected to benefit more than Java code from techniques for the efficient execution of the **invokeinterface** instruction [ACFG01].

Polymorphic calls involve dynamic binding, as the target method may depend on the runtime type of the object receiving the call. Therefore, Dufour et al. [DDHV03] distinguish between the number of target methods and the number of receiver

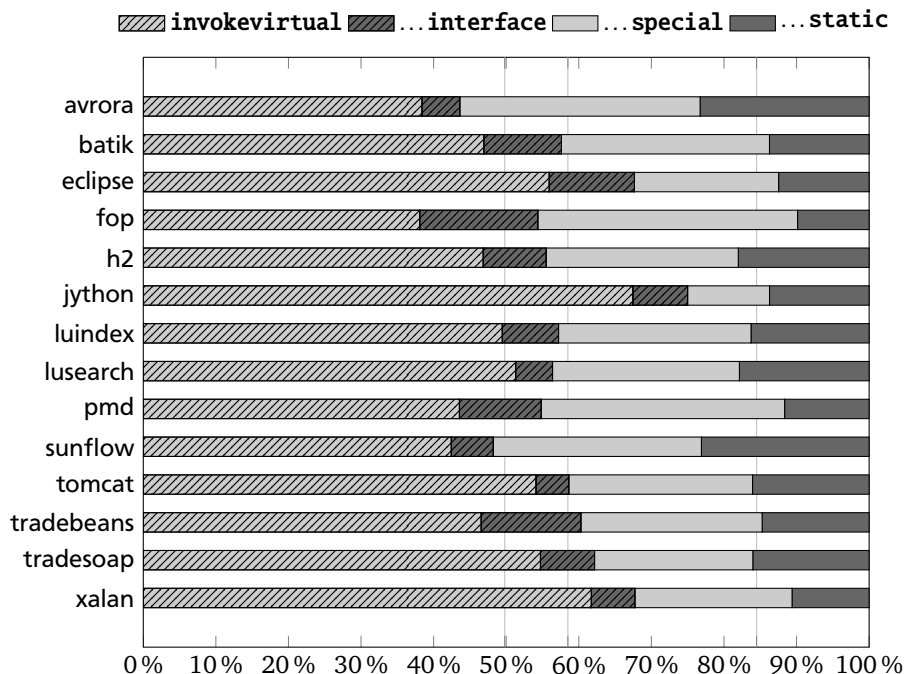


Figure 5.5a: The relative number of call sites using `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokestatic` instructions for the Java benchmarks

types for polymorphic call sites, as there are typically more of the latter than of the former; after all, not all subclasses override all methods.

Both dynamic metrics can be relevant to compiler optimizations [DDHV03]: The number of receiver types for polymorphic call sites is relevant for inline caching [HCU91], an optimization technique commonly used in runtimes for dynamically-typed languages.⁶ In contrast, the number of target methods for polymorphic call sites is relevant for method inlining [DA99b], an extremely effective compiler optimization. As modern JVMs predominantly rely on virtual method tables (vtables) in combination with method inlining to implement polymorphic

⁶ The Scala compiler relies on a similar compilation technique using Java reflection and polymorphic inline caches for structural types [DO09b]. But this technique is rarely applied within the JVM itself.

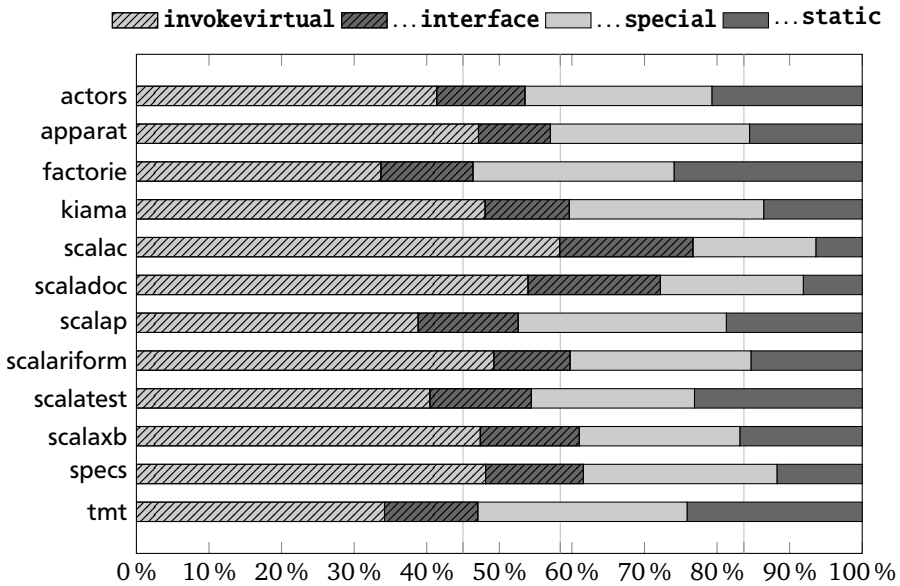


Figure 5.5b: The relative number of call sites using `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokestatic` instructions for the Scala benchmarks

method calls, I did not investigate the number of receiver types in this thesis, but focus on the number of target methods.

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 [DA99b]. Only if the number of possible targets grows too large, i.e. if the call site is megamorphic, inlining becomes infeasible. In the following, I will thus investigate the distribution of the number target methods in the different Java and Scala benchmarks.

Figures 5.7a and 5.7b (respectively 5.8a and 5.8b) show histograms of the *potentially* polymorphic call sites (`invokevirtual`, `invokeinterface`), presenting the number of call sites (respectively the number of calls for call sites) with $x \geq 1$ target methods. In Figures 5.7a and 5.7b, the actual number of invocations at a call site is not taken into account; call sites are merely counted. For example, if a call site s 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 num-

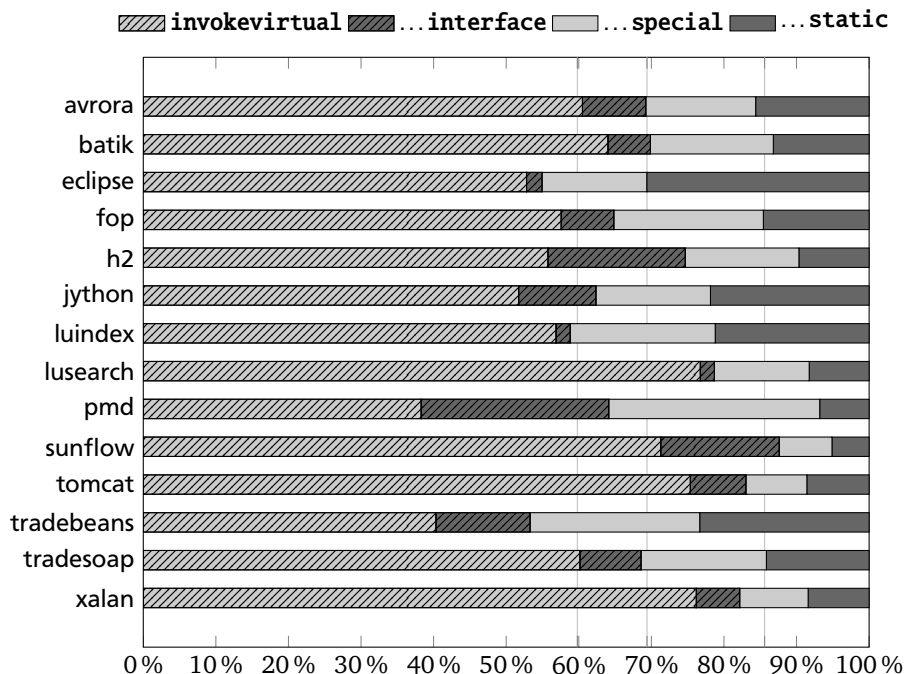


Figure 5.6a: The relative number of calls made using `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokestatic` instructions for the Java benchmarks

bers m_s and n_s of invocations will be ignored. In contrast to Figures 5.7a and 5.7b, Figures 5.8a and 5.8b do take these numbers into account. The call site s from the previous example thus contributes $m_s + n_s$ to the bar at $x = 2$. These latter figures therefore mirror the analysis on the polymorphicity of method calls performed by Gregg et al. [GPW05, Section 3.3] for the Java benchmarks from the Java Grande and SPEC JVM98 benchmark suites.

Figures 5.7a and 5.7b correspond to Figures 5.5a and 5.5b, whereas Figures 5.8a and 5.8b in turn correspond to Figures 5.6a and 5.6b. Whereas the former histograms show the possibility of inlining, the latter show its possible effectiveness. Figures 5.7a–5.7b also state what fraction of call sites proved to be monomorphic in practice. Likewise, Figures 5.8a–5.8b state what fraction of calls was made at call sites that proved to be monomorphic during the actual benchmark run. Put

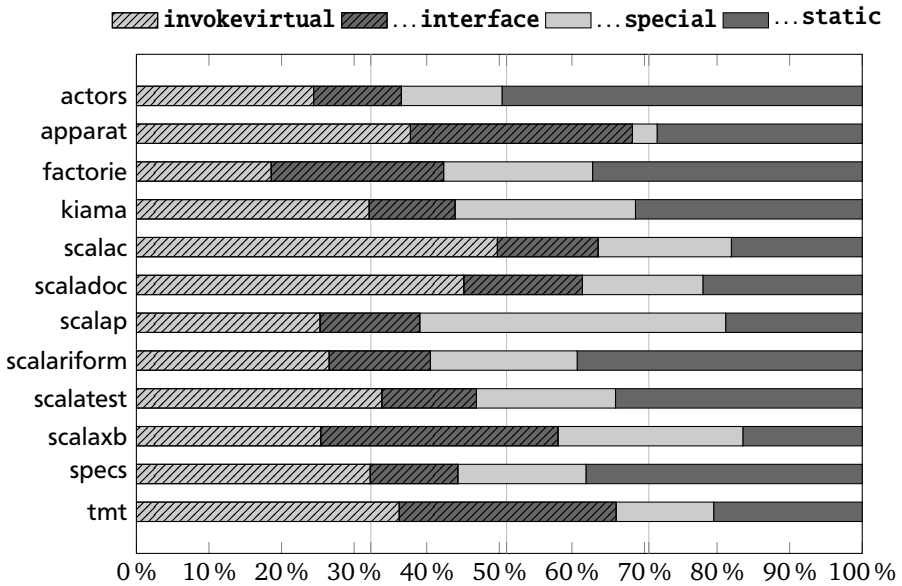


Figure 5.6b: The relative number of calls made using `invokevirtual`, `invokeinterface`, `invokespecial`, and `invokestatic` instructions for the Scala benchmarks

differently, the fractions relate the length of the histogram’s leftmost bar to the combined lengths of all the bars.

As Figures 5.7a–5.8b are concerned with potentially polymorphic call sites, they only consider call sites corresponding to `invokevirtual` or `invokeinterface` bytecodes; call sites corresponding to `invokespecial` and `invokestatic` bytecodes are excluded, as they are trivially monomorphic. Moreover, call sites that are never executed by the workload are excluded as well.

Like the analysis conducted by Gregg et al. for the Java Grande and SPEC JVM98 benchmark suites, my analysis shows that there exist both marked differences between the individual benchmarks and between the two benchmark suites. While on average 96.5% and 95.1% of the potentially polymorphic call sites are de-facto monomorphic for the Java and Scala benchmarks, respectively, on average only 87.5% and 80.2% of calls are made at one of these call sites. Nevertheless, the

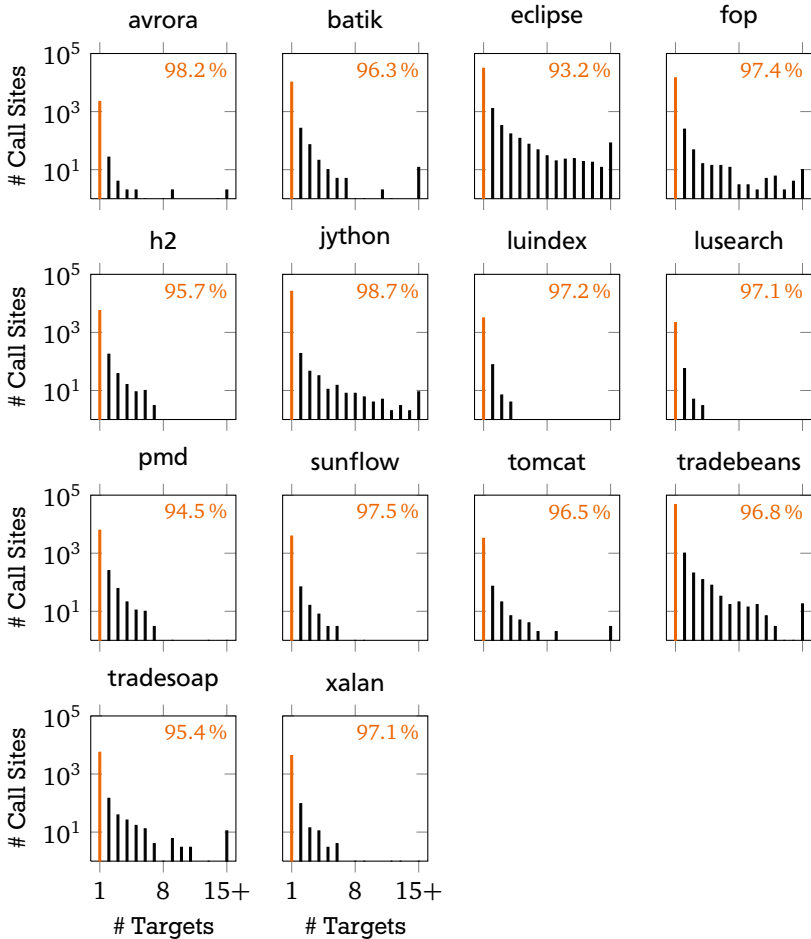


Figure 5.7a: The number of dynamically-dispatched call sites targeting a given number of methods for the Java benchmarks, together with the fraction of monomorphic call sites

latter numbers are still significantly higher than what Gregg et al. reported for the Java Grande (78%) and SPEC JVM98 benchmark suites (45%).⁷

⁷ Gregg et al. only considered dynamically-dispatched calls made using the `invokevirtual` instruction, though; calls made using `invokeinterface` were ignored in their study.

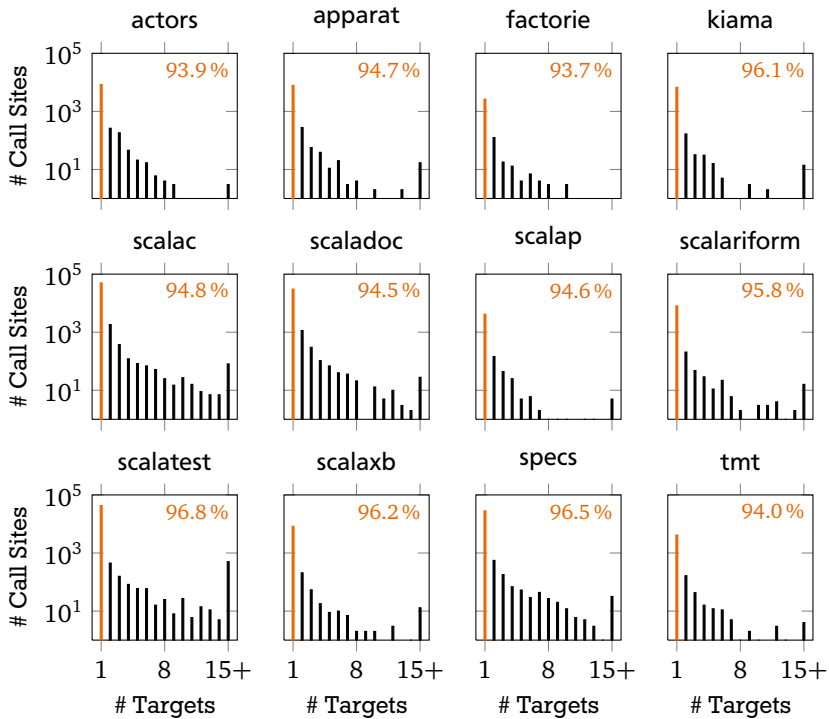


Figure 5.7b: The number of dynamically-dispatched call sites targeting a given number of methods for the Scala benchmarks, together with the fraction of monomorphic call sites

Dynamic dispatch with respect to many different targets per call site is used more frequently by the Scala benchmarks than by the Java benchmarks. That being said, megamorphic call sites, i.e. call sites with 15 or more targets, are exercised by almost all benchmarks in both the Java and Scala benchmark suites, i.e. by 10 out of 14 and 11 out of 12 benchmarks, respectively.

Based on the significantly lower number of calls made at potentially polymorphic yet monomorphic call sites, one may assume that the effectiveness of inlining is significantly reduced for Scala code. But when one considers all call sites, i.e., when one includes the trivially-monomorphic call sites that use **invokespecial** and **invokestatic** instructions, it turns out that the differences between Java and Scala code are far less pronounced: For the Scala benchmarks, on average 97.1%

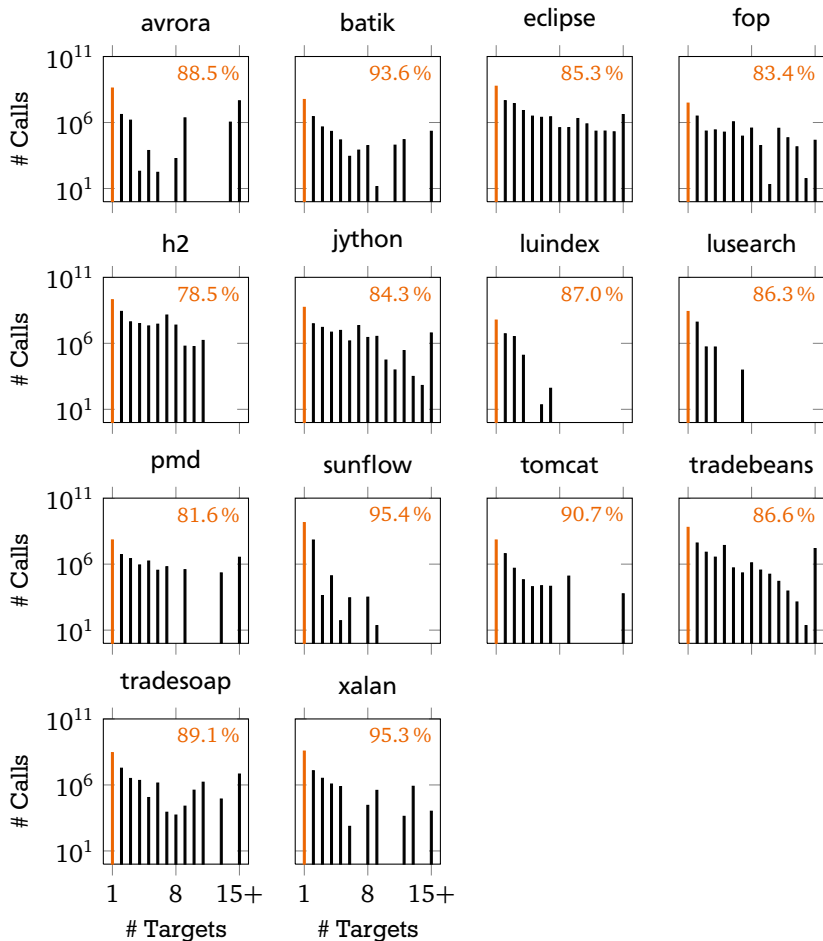


Figure 5.8a: The number of dynamically-dispatched calls made at call sites with a given number of targets for the Java benchmarks, together with the fraction of calls made at monomorphic call sites

of *all* call sites are monomorphic and account for 89.7% of the overall method calls. For the Java benchmarks, on average 97.8% of *all* call sites are monomorphic and account for 91.5% of calls. This is a direct consequence of the observation made in

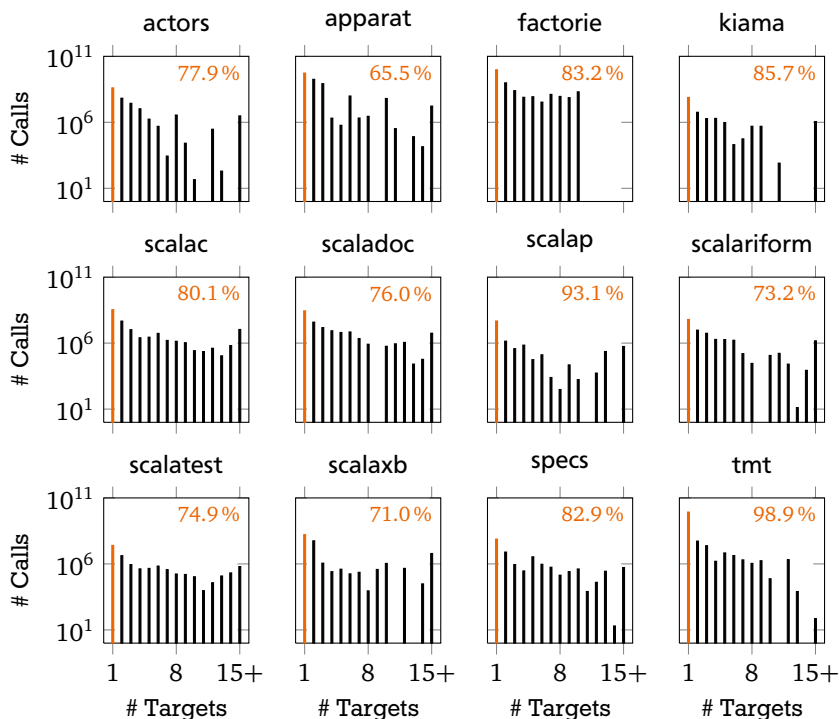


Figure 5.8b: The number of dynamically-dispatched calls made at call sites with a given number of targets for the Scala benchmarks, together with the fraction of calls made at monomorphic call sites

Figures 5.6a and 5.6b: For the Scala benchmarks nearly half of the calls are made using **invokespecial** and **invokestatic** instructions.

The Scala benchmarks, however, also exhibit a higher variance with respect to the number of monomorphic calls than their Java counterparts. With respect to this metric, my Scala benchmark suite is therefore at least as diverse as its role model, the DaCapo benchmark suite.

5.4.3 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 a concrete implemen-

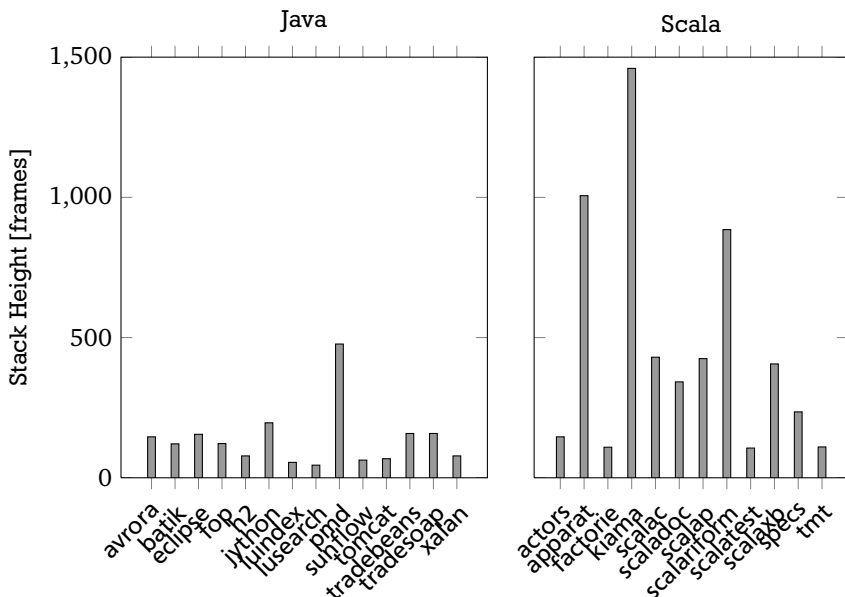


Figure 5.9: The maximum stack height required by the Java and Scala benchmarks

tation of the virtual machine may place some of them in registers instead (cf. Section 5.4.4). As most modern JVMs are unable to re-size a thread’s call stack at runtime,⁸ 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 5.9 shows, the stack usage of the Scala benchmarks is significantly higher than for the Java benchmarks. That being said, for both benchmark suites the required stack size varies widely across benchmarks: For the Scala benchmarks, it ranges from 110 (tmt) to 1460 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 (pm2), with an average of 137.

The question is therefore 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 the use of recursion by application or library code on the other

⁸ Re-sizing call stacks may require moving them in memory; this complicates the implementation.

hand. To assess to what extent the latter contributes to the benchmarks' stack usage, I made 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 often occurs when the Composite pattern is used, where different components implement the same abstract operation.

Figures 5.10a and 5.10b, which depict the distribution of stack heights for each of the benchmarks, therefore use an extended definition of recursion. In this definition I 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 5.10a and 5.10b, 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, for example, 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, namely the abstract syntax tree of the Scala code formatted by `scalariform`. In general, this form of recursion is harder to optimize, as tail-call elimination requires the recursive call's target to be identical to the caller.

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_{rec} from which on all calls are “truly” recursive. This shows that, ultimately, it is the more extensive use of recursion by applications and libraries rather than the infrastructure methods introduced by the Scala compiler that leads to the observed high stack usage.

5.4.4 Argument Passing

A method's stack frame not only stores the 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. Likewise, different kinds of arguments may

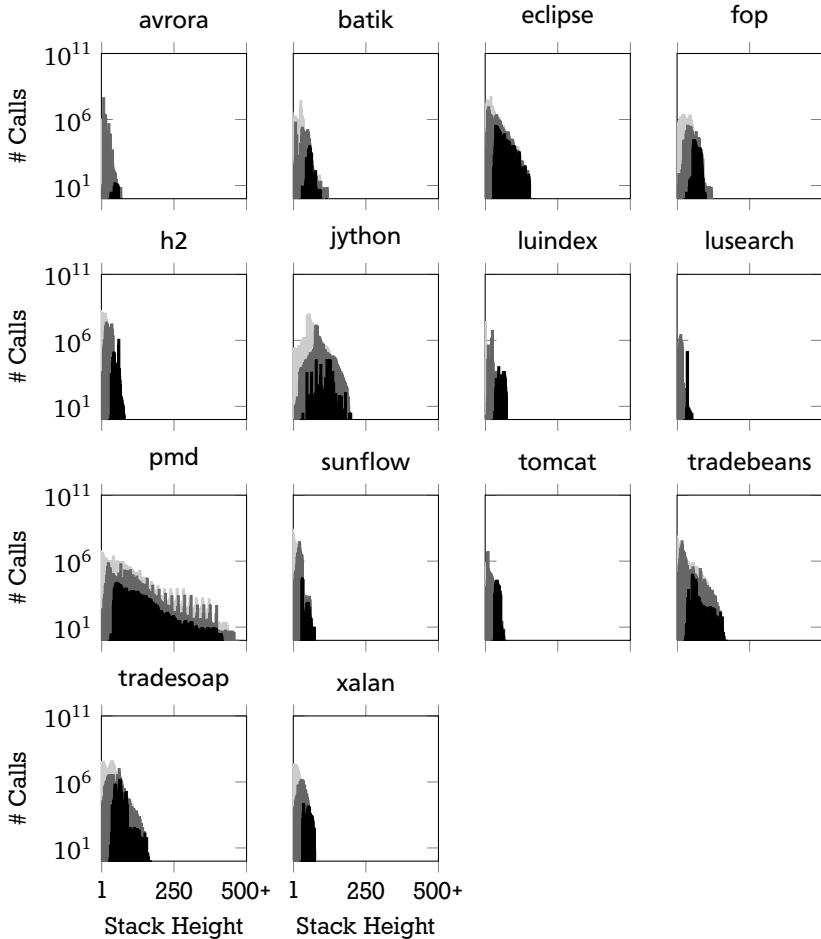


Figure 5.10a: The distribution of stack heights upon a method call for the Java benchmarks: all method calls (■), recursive calls (■), and recursive calls to the same implementation (■)

have to be passed differently; on many architectures floating point numbers occupy a distinct set of registers.

As not all benchmarks in the Java and Scala benchmark suites make much use of floating-point arithmetic, I will first focus on the six benchmarks for which at least

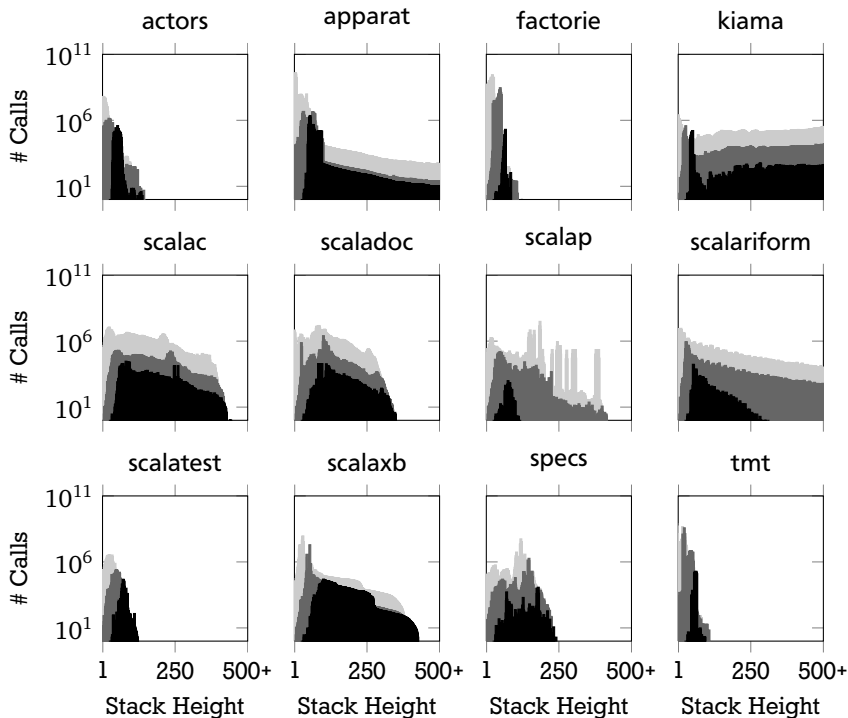


Figure 5.10b: The distribution of stack heights upon a method call for the Scala benchmarks: all method calls (■), recursive calls (■), and recursive calls to the same implementation (■)

1% of method calls carry 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 5.11 depict the number of floating-point arguments passed upon a method call in relation to the overall number of arguments. The bar 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 multiple floating-point arguments: In the case of sunflow, four-argument methods are frequently called with three floating-point arguments (x-, y-, and z-

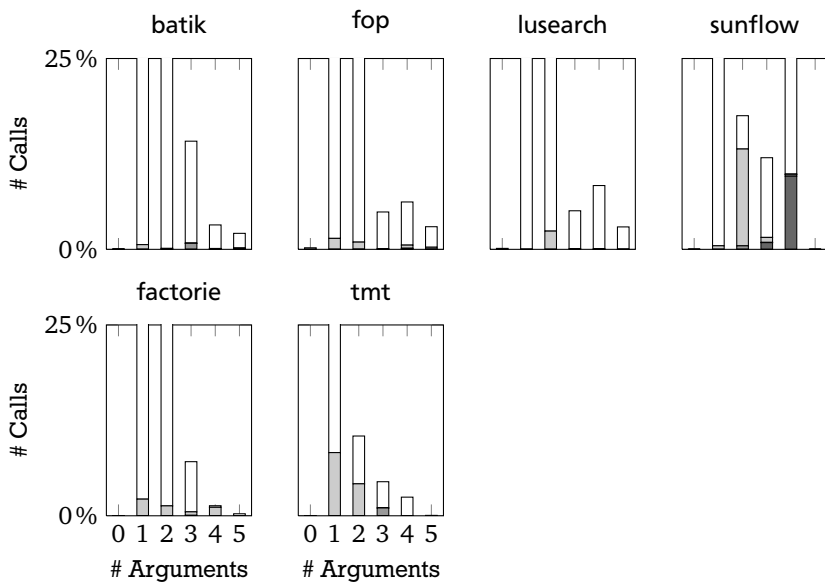


Figure 5.11: Distribution of the number of floating-point arguments passed upon a method call: none (□), 1 (◻), 2 (◼), 3 (◽), 4 (◾), and 5 or more (◿)

coordinate), 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 5.12a and 5.12b 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 bar shadings correspond to the number of reference arguments; the darker the shade, the large the number of reference arguments.

Figures 5.12a and 5.12b distinguish between those calls with an implicit receiver object (**invokevirtual**, **invokeinterface**, and **invokespecial**) and those without one (**invokestatic**). Both figures display the amount of arguments attributed to the former group above the x-axis, whereas the amount of arguments attributed to the latter group 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

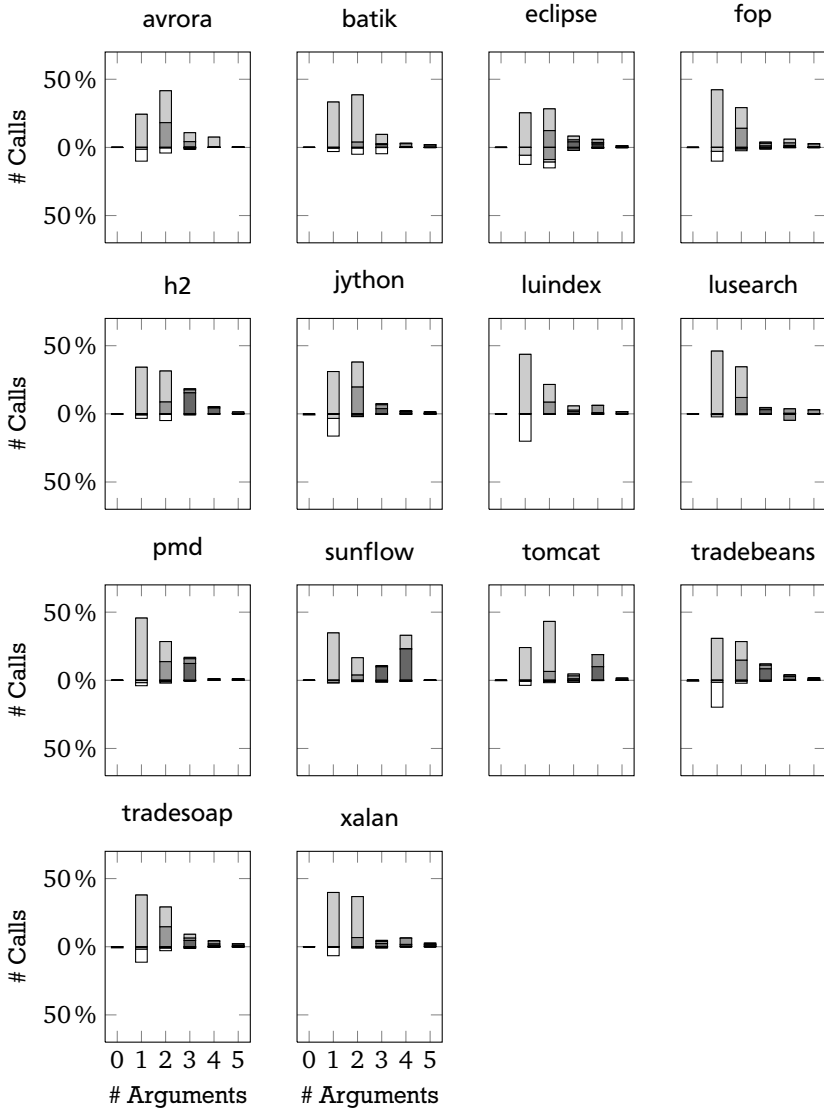


Figure 5.12a: Distribution of the number of reference arguments passed upon a method call by the Java benchmarks: none (\square), 1 (\square), 2 (\square), 3 (\square), 4 (\blacksquare), and 5 or more (\blacksquare)

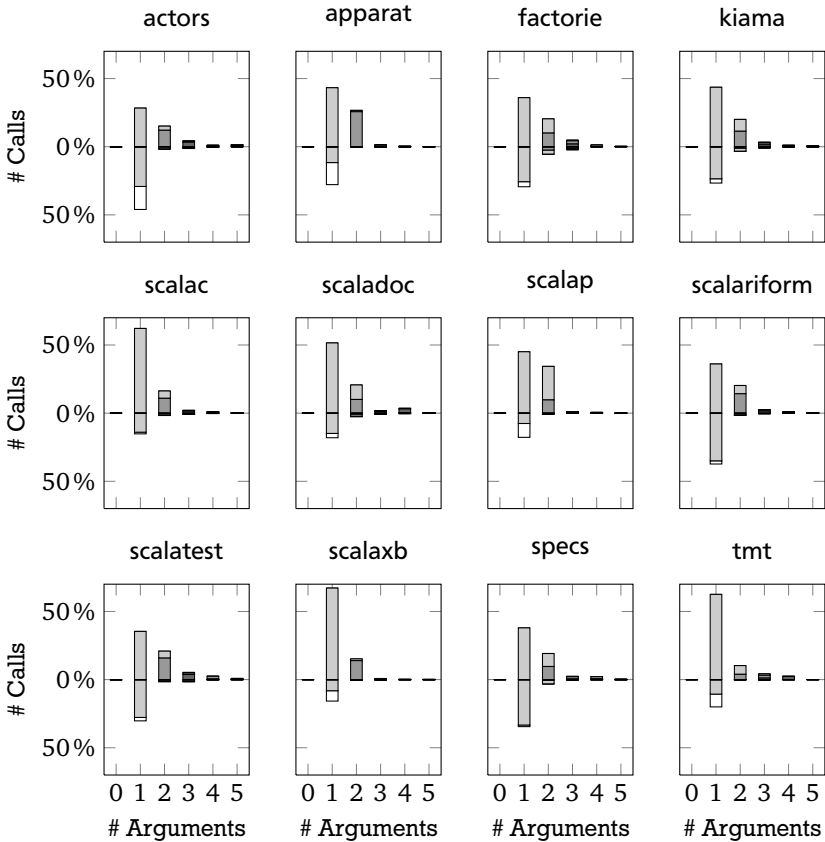


Figure 5.12b: Distribution of the number of reference arguments passed upon a method call by the Scala benchmarks: none (□), 1 (▤), 2 (▥), 3 (▦), 4 (▧), and 5 or more (▨)

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” [LYBB11].

For Scala and Java benchmarks alike, almost all methods have at least one argument, be it explicit or implicit, viz. **this**. This confirms earlier findings by Daly et al. [DHPW01] for Java benchmarks. 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 to 1.47 for Scala code and 1.69 to 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: For a large number of calls, the direct benefit of inlining, i.e. the removal of the actual call, outweighs the possible indirect benefits, i.e. the propagation of information about the arguments’ types and values which, in turn, facilitates constant folding and propagation.

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 more convenient to the programmer. But while methods taking many parameters do exist, they are rarely called. Instead, the parameterless “getters” automatically generated by the Scala compiler for every field dominate.

5.4.5 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 [HHR95, SYN03], or trace-based [BCW⁺10] approach to compilation, pronounced hotspots are fundamental to the effectiveness of adaptive optimization efforts. It is therefore 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” [DDHV03], the metric used in this thesis is continuous. Figures 5.13a and 5.13b 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; again, dormant methods are excluded. Basic blocks, however, 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 by a traditional method-based compiler. 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 basic blocks. To reflect this, Figures 5.13a and 5.13b also report the extent to which the hottest methods are responsible for the execution. The two data series are derived as follows:

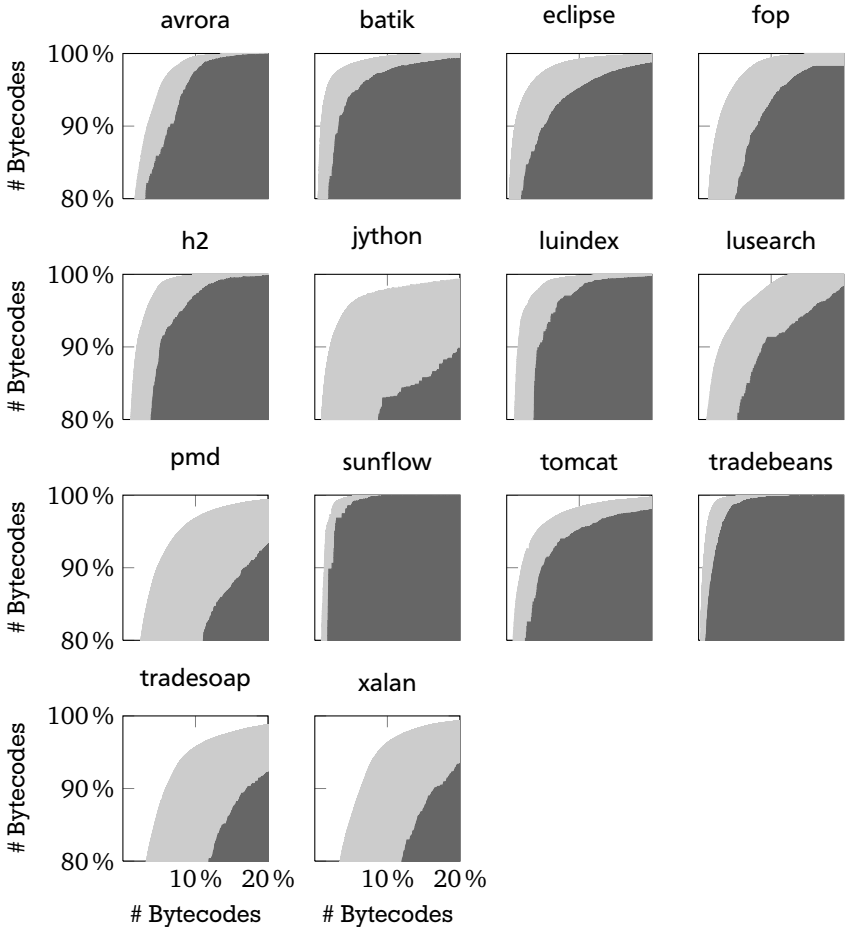


Figure 5.13a: Cumulative number of executed bytecodes for the most frequently executed bytecodes when measured at the granularity of basic blocks (■) or methods (■)

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 $y = \sum_{j=1}^i \text{size}(b_j) \cdot \text{freq}(b_j)$,

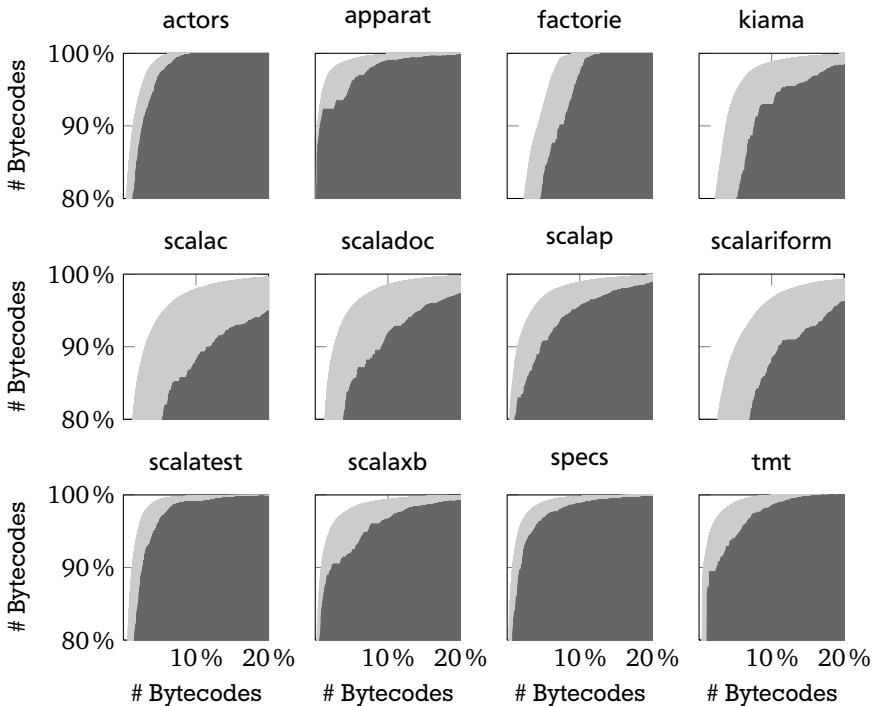


Figure 5.13b: Cumulative number of executed bytecodes for the most frequently executed bytecodes when measured at the granularity of basic blocks (■) or methods (■)

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 denotes the set of basic blocks of method m_j .

For the `actors` and `scalap` benchmarks only 1.4% and 1.5% of all bytecode instructions, respectively, are responsible for 90% of all executed bytecodes. Beyond

that point, however, 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 instructions to cover 90% of all executed instructions, whereas 4.7% of `scalap`'s instructions need to be compiled 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; some blocks are executed frequently and some 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 [HHR95, SYN03] and trace-based compilation [BCW⁺10] can be employed to lessen the effect of such temperature drops within methods.

5.4.6 Use of Reflection

While reflective features that are purely informational, e.g. runtime-type information, do not pose implementation challenges, other introspective features like reflective invocations or instantiations are harder to implement efficiently by a JVM [RZW08]. It is therefore 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 [DO09b]. I have thus extended⁹ TamiFlex [BSS⁺11, BSSM10] 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.).

I 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 [BN99] might avoid the reflective call altogether. For the two benchmark suites, Figure 5.14 depicts 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). They are subject to some minor imprecision, as TamiFlex conflates call sites sharing a source line, thereby potentially exaggerating the number of shared `Method` instances.

⁹ This extension has since become part of TamiFlex's 2.0 release.

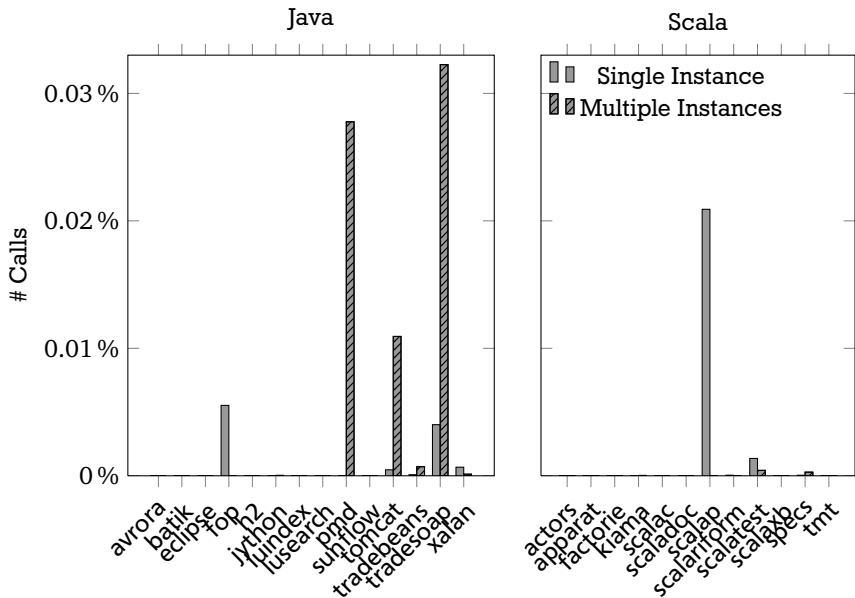


Figure 5.14: Number of methods invoked reflectively with a single or multiple Method instances per call site, normalized to the number of all method invocations

Few benchmarks in either suite perform a significant number of reflective method invocations. Even for those benchmarks that do (*scalap*, *pmd*, *tomcat*, and *tradesoap*), reflective invocations account for at most 0.03% of invocations overall. In the case of *scalap*, these invocations are almost exclusively due to the use of structural types within the benchmark. This also explains why in most cases only a single `Method` instance is involved, which the Scala runtime’s keeps in an inline cache [DO09b]. Should the use of reflection to implement structural types be supplanted by the use of the special `invokedynamic` instruction in future versions of the Scala compiler, these metrics nevertheless remain meaningful, as similar caching techniques can be applied [TR10].

I next consider reflective object instantiations, i.e. calls to `Class.newInstance`, `Constructor.invoke`, as well as `Array.newInstance`. Again, I distinguish between call sites referring to a single meta-object and call sites referring to several.¹⁰

¹⁰ The `Class` instance passed to `Array.newInstance` are treated as the meta-object here.

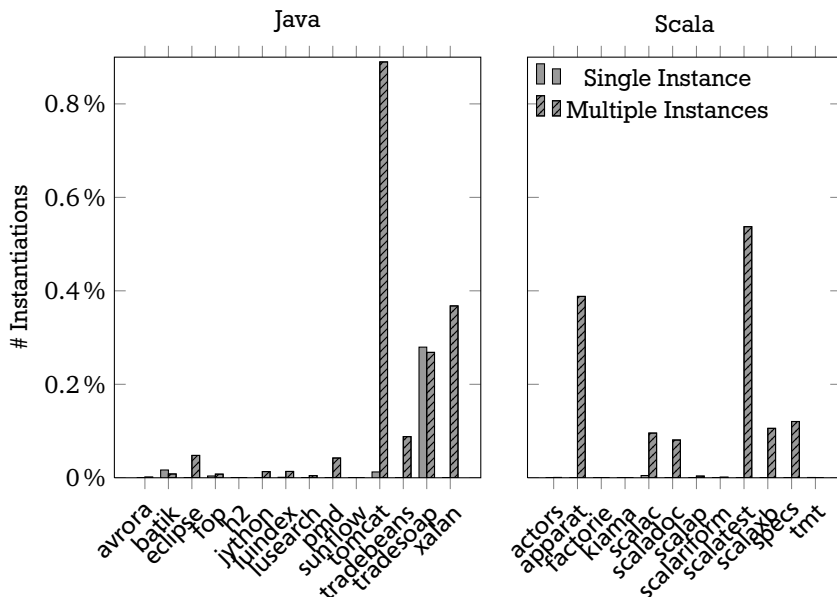


Figure 5.15: Number of objects instantiated reflectively with a single or multiple Class instances per call site, normalized to the number of all object instantiations (**new**)

Figure 5.15 depicts the number of reflective instantiations for the Java and Scala benchmark suites, respectively. The numbers have been normalized with respect to the number of overall allocations (**new** bytecode instructions). Again, the numbers are subject to some minor imprecision, as TamiFlex conflates reflective allocation sites sharing a source line.

The large number of reflective instantiations by several Scala benchmarks can be traced back 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 (cf. Section 2.1); the **newarray**, **anewarray**, and **multianewarray** all require the array’s type to be hard-coded in the instruction stream.

Unlike reflective invocations and instantiations, reflective field accesses are almost absent from both Scala and Java benchmarks. The only notable exception is

eclipse, which reflectively writes to a few hundred fields to perform dependency injection.

5.4.7 Use of Boxed Types

Unlike Java, which distinguishes between primitive and reference types, Scala maintains the illusion that “every value is an object,” even though it tries to use primitive types internally for reasons of efficiency. It is sometimes necessary, however, for the Scala compiler to wrap a primitive value in an object box to maintain the illusion throughout.

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. I have therefore measured to which extent the Java and Scala benchmarks create boxed values. To this end, I 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 of commonly-used values but may impede JIT compiler optimizations [Chi07].

Figure 5.16 shows 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 I have not counted boxed values created from strings instead of unboxed values, as the intent here is often rather different, namely to parse a sequence of characters. While for the Java benchmarks, boxing accounts only for very few object creations, this is not true for many of the Scala benchmarks; for example, almost all of the objects created by 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 that these two benchmarks operate on floating-point numbers, for which 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 [DO09a] 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 4547 call sites of a boxing constructor.

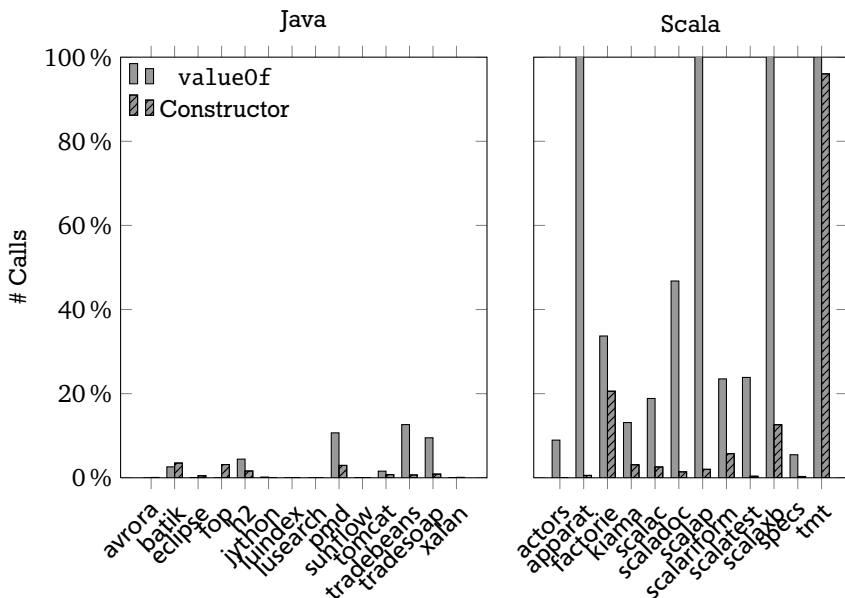


Figure 5.16: Boxed instances requested (`valueOf`) and created (`Constructor`), normalized to the number of all object allocations (**new**)

5.4.8 Garbage-Collector Workload

So far I have focused on code-related metrics. But how does the garbage-collector workload differ between Java and Scala programs? This question has been answered using Elephant Tracks [RGM11], which produces an exact trace containing object allocations and deaths as well as field updates. The resulting traces were run through a GC simulator, which was configured to simulate a generational collection scheme: New objects are allocated in a 4 MiB nursery, a nursery size that was also used by Blackburn et al. in their analysis of the initial version of the Da-Capo benchmarks [BGH⁺06]. When full, the nursery is collected, and any survivors are promoted to the older generation. The size of this older generation was set to 4 GiB. When this older generation is filled, a full-heap collection is performed. While this setup is far simpler than the generational collectors found in production JVMs, it nevertheless gives a good initial intuition of the garbage-collector workload posed by the different benchmarks. Moreover, the setup is similar enough to

Benchmark	Cons	Marks	Mark/Cons	Survival
avro	2 075 466	59 684	0.03	2.88 %
batik	1 088 785	327 505	0.30	30.08 %
eclipse	66 569 509	171 766 557	2.58	30.82 %
fop	2 982 888	486 253	0.16	16.30 %
h2	100 265 924	1 948 804 662	19.44	46.00 %
jython	43 752 983	10 654 369	0.24	24.35 %
luindex	404 186	50 616	0.13	12.52 %
lusearch	13 323 025	29 544 022	2.22	19.86 %
pmd	9 110 278	2 671 559	0.29	29.32 %
sunflow	61 883 982	3 577 022	0.06	5.78 %
xalan	10 250 705	747 468	0.07	7.29 %
apparat ^{small}	8 953 723	338 633	0.04	3.78 %
factorie	1 505 398 186	6 475 518 895	4.30	1.24 %
kiana	12 891 237	205 002	0.02	1.59 %
scalac	19 875 421	433 046	0.02	2.18 %
scaladoc	18 077 250	395 418	0.02	2.19 %
scalap	1 948 053	64 735	0.03	3.32 %
scalariform	10 077 808	325 769	0.03	3.23 %
scalaxb	4 343 541	42 037	0.01	0.97 %
specs	12 684 716	248 027	0.02	1.96 %
tmt ^{small}	395 247 346	59 516 039	0.15	0.01 %

Table 5.1: Garbage collection marks and cons (object allocations) used, together with the survival rates in a 4 MiB nursery. (To keep the traces produced by Elephant Tracks manageable, two benchmarks had to be run with a reduced input size.)

that of Blackburn et al. [BGH⁺06] to allow for comparisons with the older version of the DaCapo benchmark suite.

Table 5.1 shows the simulation’s results. In this table, “cons” denotes the total number of objects allocated by the benchmark, “marks” refers to the total number of times those objects are marked as live, and the nursery survival rate denotes the fraction of allocated objects which survive a minor garbage collection, i.e. which are promoted to the older generation.

Note that the nursery survival rate of the Scala programs in the benchmark suite is considerably lower than that of the Java programs from the DaCapo 9.12 suite:

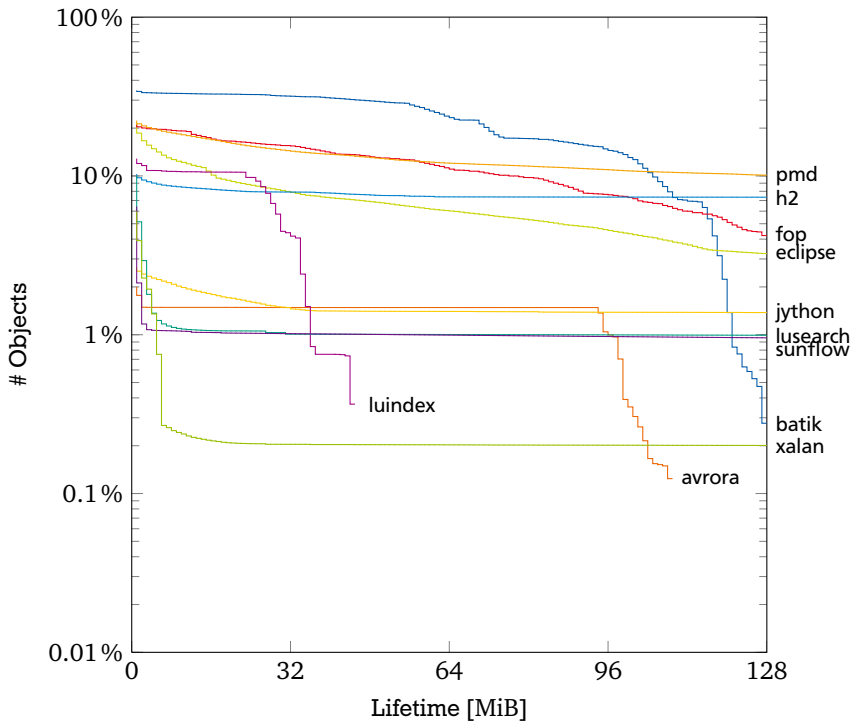


Figure 5.17a: Fraction of objects surviving more than a given amount of allocation for the Java benchmarks

Aside from *avrora*, the entirety of the DaCapo suite has a higher nursery survival rate than the Scala benchmark suite, whose *apparat* benchmark sports the highest nursery survival rate therein with a mere 3.78%. The low nursery survival rate observed in the simulation suggests that, at least for most Scala benchmarks, objects die younger than for the Java benchmarks. Figures 5.17a and 5.17b confirm this: For half of the Java benchmarks at least 10% of objects survive for 20 MiB of allocation, whereas few objects allocated by the Scala benchmarks survive for more than a few MiB of allocation.

The question arises whether the sharp drop of the Scala benchmarks' survival rates can be explained by the fact that the Scala compiler generates many short-lived objects under the hood, e.g. to represent closures (cf. Section 2.3.2). For the purpose of this study, I have identified four categories of such

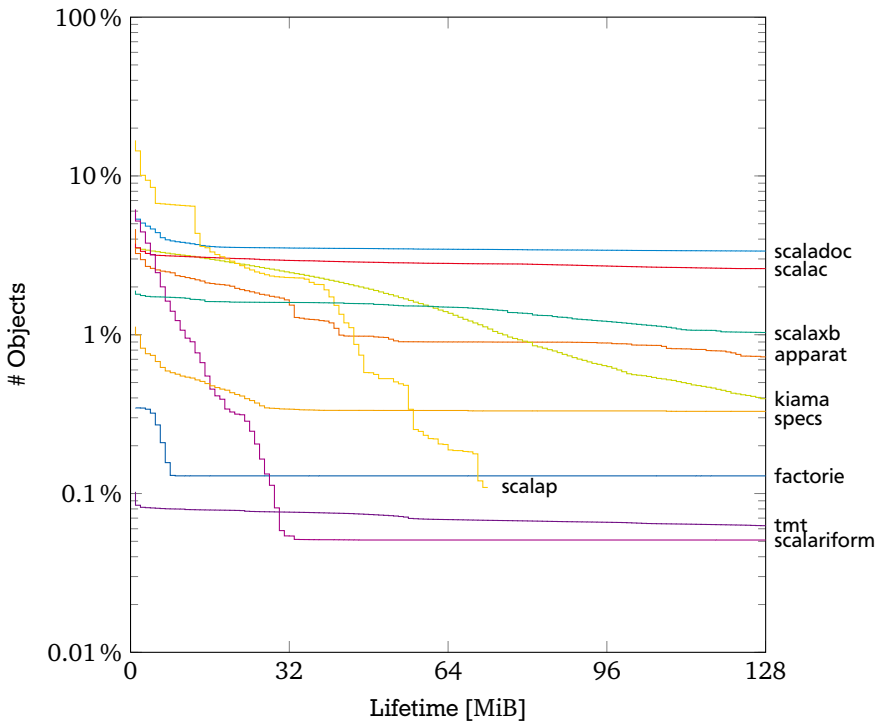


Figure 5.17b: Fraction of objects surviving more than a given amount of allocation for the Scala benchmarks

“under-the-hood” objects: objects representing closures, objects representing variables captured by a closure (e.g. `scala.runtime.IntRef`), boxed primitives (e.g. `java.lang.Integer`), and rich primitives (e.g. `scala.runtime.RichInt`). Listing 5.1 illustrates these categories using the example from Section 2.3.3 again.

To answer the aforementioned question, Table 5.2 tabulates both the likelihood of a new object belonging to one of the four categories of “under-the-hood” objects or to the category of other objects together with the likelihood of these objects surviving just 1 MiB of allocation. As can be seen, for all Scala benchmarks a significant portion of allocations is due to what I termed “under-the-hood” objects; only 68.62% of allocations are due to “regular” objects. Closures in particular contribute significantly to overall allocations. 19.19% of objects allocated by the `scalac` benchmark, e.g., represent closures. But only 0.10% of those objects exhibit a

```

1 object Countdown {
2   def nums = {
3     var xs = List[Int]() // Captured Variables
4     (1 to 10) foreach { // Rich Primitives
5       x \underline{=>} // Closures
6         xs = x :: xs // Boxed Primitives
7     }
8     xs
9   }
10 }

```

Listing 5.1: The four categories of “under-the-hood” objects (cf. Listing 2.3)

lifetime of 1 MiB or more; for this benchmark, most closures are extremely short-lived. What Table 5.2 also shows is that boxed primitives play a noticeable role for almost all benchmarks, with only `apparat` and `specs` spending less than 1% of their respective allocations on boxed primitives. As noted elsewhere [SMSB11], the `tmt` benchmark is an outlier in this respect; almost all objects allocated by this numerical-intensive benchmark are boxed floating-point numbers, which happen to be extremely short-lived.

While Table 5.2 shows that short-lived “under-the-hood” objects indeed significantly contribute to the low survival rates observed for the Scala benchmarks (cf. Figure 5.17b), on average they account for only one third of allocations. The remaining two thirds are regular objects allocated directly on behalf of the program. But they too exhibit low survival rates.

Lifetimes and survival rates are one important component of a garbage collector’s workload, but the sheer amount of allocations and pointer mutations, if some write barrier is employed, are another. Figure 5.18 thus visualizes the size of the different benchmarks, both in terms of allocation volume and pointer mutations. This figure not only gives a good impression of the size of traces the GC simulator had to process, but also shows that the Java benchmarks from the DaCapo benchmark suite are more likely to favour mutation over allocation than their counterparts from the Scala benchmark suite, with the sole exception of `apparat`. Also, the two Scala benchmarks `factorie` and `tmt` are far more allocation-intensive than any other benchmarks in either suite—in particular, since Figure 5.18 shows only the small input size for the latter benchmark. (The profiles for `tmt`’s default input size grew too large to process.)

	Closures	Captured Variables	Boxed Primitives	Rich Primitives	Other Objects	
apparat	3.45 %	1.35 %	0.41 %	0.64 %	94.15 %	Cons
	1.31 %	0.07 %	21.28 %	0.00 %	5.82 %	Survival
factorie	20.48 %	1.22 %	19.15 %	0.00 %	59.15 %	Cons
	0.00 %	0.00 %	0.76 %	0.00 %	0.34 %	Survival
kiama	8.53 %	1.14 %	2.34 %	4.48 %	83.51 %	Cons
	0.44 %	0.20 %	0.13 %	0.00 %	4.21 %	Survival
scalac	19.19 %	4.40 %	2.25 %	4.52 %	69.64 %	Cons
	0.10 %	0.04 %	1.49 %	0.00 %	5.28 %	Survival
scaladoc	12.40 %	3.16 %	1.14 %	17.35 %	65.95 %	Cons
	0.73 %	0.01 %	1.08 %	0.00 %	8.54 %	Survival
scalap	29.67 %	1.84 %	1.78 %	0.13 %	66.58 %	Cons
	19.78 %	60.53 %	32.13 %	0.00 %	13.74 %	Survival
scalariform	11.89 %	1.41 %	4.90 %	0.20 %	81.61 %	Cons
	0.21 %	0.17 %	0.13 %	0.00 %	7.49 %	Survival
scalaxb	22.79 %	0.95 %	10.67 %	0.92 %	64.66 %	Cons
	0.00 %	0.01 %	0.14 %	0.00 %	2.88 %	Survival
specs	3.16 %	0.20 %	0.49 %	0.32 %	95.83 %	Cons
	1.44 %	0.05 %	1.56 %	0.00 %	1.09 %	Survival
tmt ^{small}	1.38 %	0.00 %	92.99 %	0.47 %	5.16 %	Cons
	0.16 %	0.04 %	0.04 %	0.00 %	1.29 %	Survival
Mean	13.29 %	1.57 %	13.61 %	2.90 %	68.62 %	Cons
	2.42 %	6.11 %	5.87 %	0.00 %	5.07 %	Survival

Table 5.2: Distribution of allocations for the Scala benchmarks, together with the 1 MiB survival rate for each of the five categories

5.4.9 Object Churn

Object churn, i.e. the creation of many temporary objects, is an important source of overhead which badly hurts framework-intensive Java programs [DRS08]. But as I have shown in the previous section, Scala programs exhibit at least as many short-lived objects as Java programs. The fact that temporary objects are often not

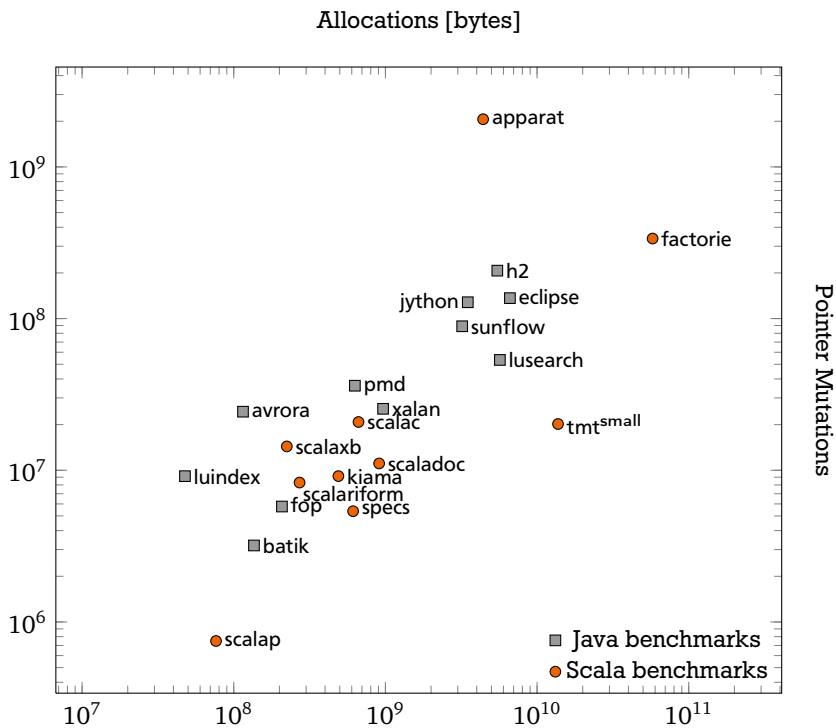


Figure 5.18: Allocations and pointer mutations performed by the benchmarks

only used within a single method but either passed on or returned to other methods makes intra-procedural escape analysis ineffective in identifying such temporaries. Their identification therefore requires either an expensive inter-procedural analysis [DRS08] or careful use of method inlining [SAB08] to expose multiple methods to an inexpensive intra-procedural analysis. In the latter case in particular, it is not so much of interest how long an object lives but how closely it is captured by a calling context. Ideally, the object dies in the calling context it was allocated in.

To study object churn, I defined the novel metric of dynamic churn distance, illustrated by Figure 5.19. For each object, one determines both its allocation and death context. From these two, one derives the closest capturing context; intermediate contexts are ignored. The dynamic churn distance is then either the distance from the capturing context to the object’s allocation context or the distance from the capturing context to the object’s death context, whichever is larger. Note that

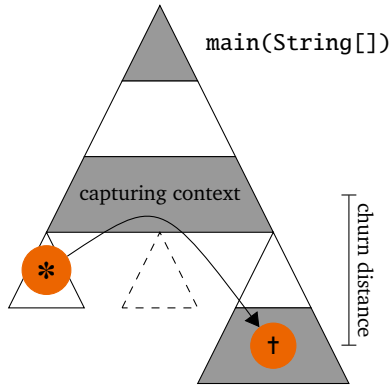


Figure 5.19: The churn distance of an object is computed as the largest distance between its allocation (*) respectively death (†) context and its closest capturing context.

the object’s death context is defined with respect to the object’s allocating thread, which may or may not have been the thread which relinquished the last reference to the object in question. This simple definition is sufficient since objects that escape their allocating thread are not only hard to optimize away but typically also quite rare.

Figures 5.20a and 5.20b depict the distribution of churn distances for the Java and Scala benchmarks, respectively, as derived from the traces produced by Elephant Tracks [RGM11]. The distribution is remarkably benchmark-dependent, with the most peaked histograms belonging to number-crunching benchmarks: *avrora* (processor simulation), *sunflow* (raytracing), and *tmt* (machine learning). Here, a small kernel dominates the birth and death patterns of objects. Furthermore remarkable is that for most benchmarks a churn distance of zero is very rare, with a maximum of 12.83 % (*sunflow*) and an average of only 2.64 % and 1.27 % for the Java and Scala benchmarks, respectively (excluding dummy); few objects die in the same method that allocated them. Nevertheless, large churn distances are also relatively uncommon; the median churn distance is never larger than 4, with that for the Scala programs generally being higher than for their Java counterparts.

Re-using the categorization of Table 5.2, Table 5.3 tabulates the median churn distances observed for different categories of objects. What is noteworthy is that rich primitives not only exhibit a median churn distance of 1 for all benchmarks, but that their churn distance is *always* equal to 1. This is a direct consequence of their typical usage pattern, which creates a rich primitive in an **implicit** conversion

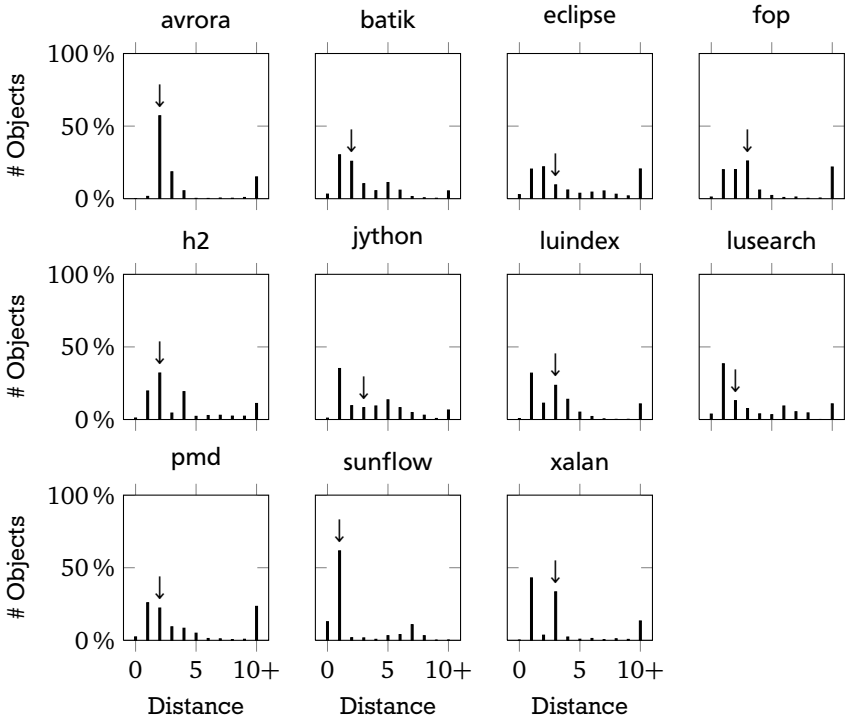


Figure 5.20a: The distribution of churn distances with median marked (↓) for the Java benchmarks

method, invokes a method on the returned object, and then discards it. In contrast to rich primitives, boxed primitives predominately exhibit median churn distances higher than the overall median. This indicates that these objects are kept for longer, e.g. to be passed around in a collection. Unlike for rich and boxed primitives, the churn distances of “under-the-hood” objects that represent closures and their captured variables exhibit no such pattern; their churn distances vary widely from benchmark to benchmark.

5.4.10 Object Sizes

If a program allocates not only many, but many small objects the ratio of payload to header overhead gets worse. It is thus of interest to examine the distribution

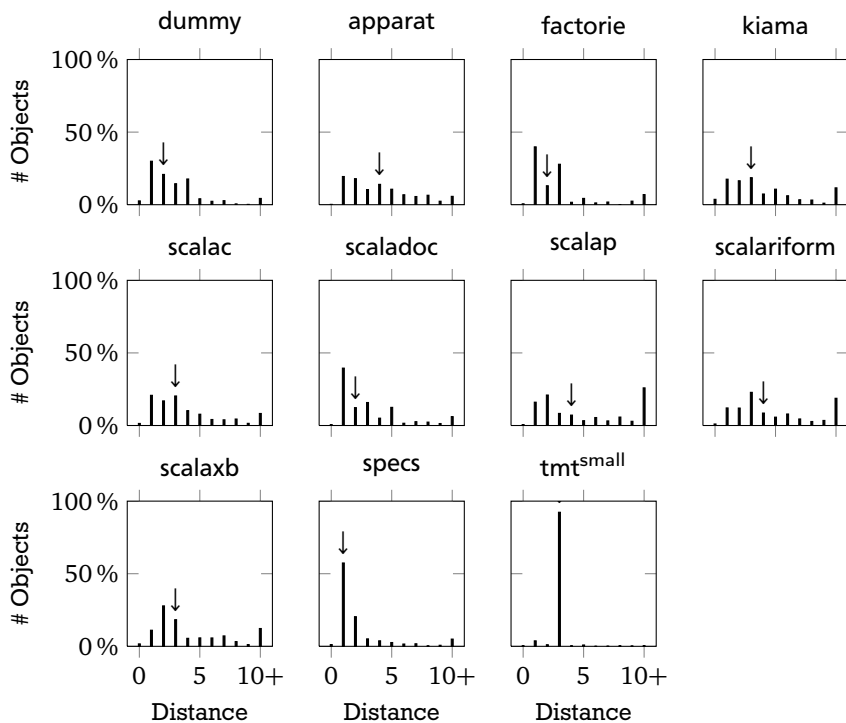


Figure 5.20b: The distribution of churn distances with median marked (\downarrow) for the Scala benchmarks

of object sizes as depicted in Figures 5.21a and 5.21b for the different Java and Scala benchmarks. This figure focuses on objects of small size (less than 88 bytes) and shows that, on average, Scala programs allocated significantly smaller objects than Java programs: For most Scala benchmarks the median is either just 8 bytes or 16 bytes, the size of one or two pointers, respectively. Thus, the overhead introduced by the object header, on whose properties (locks, identity hash-codes) I will focus later (cf. Sections 5.4.14 and 5.4.15), becomes more pronounced.

5.4.11 Immutability

While favouring immutable data structures is considered a best practice in Java [Blo08, GPB⁺06], it is even more so in Scala [OSV10]; in particular, Scala's

Benchmark	Closures	Captured Variables	Boxed Primitives	Rich	Other Objects
apparat	3↓	1↓	13↑	1↓	4
factorie	2	4↑	3↑	1↓	1↓
kiama	4↑	5↑	3	1↓	3
scalac	3	1↓	3	1↓	3
scaladoc	3↑	1↓	6↑	1↓	3↑
scalap	6↑	26↑	18↑	1↓	4
scalariform	3↓	0↓	3↓	1↓	4
scalaxb	3	1↓	4↑	1↓	3
specs	7↑	1	7↑	1	1
tmt	2↓	0↓	3	1↓	1↓

Table 5.3: Median churn distances for the Scala benchmarks for each of the five classes, together with an indication whether the category’s median churn distance is lower (↓) or higher (↑) than the overall median

collection library offers a large selection of basic, immutable data structures in the `scala.collection.immutable` package. But immutable data does not only make it easier for the programmer to reason about a program, it also allows for various optimizations [PS05].

I thus assessed to what extent Java and Scala programs make use of immutable data structures. In the analysis, I distinguish between class and object immutability [HP09] as well as between per-class and per-object field immutability: A class is considered immutable if all of its instances are immutable objects.¹¹ Likewise, an object is considered immutable if all of its instance fields are immutable. If a field proves to be immutable not just for a single instance (per-object immutable), but for all objects of a class, I consider it to be per-class immutable.

While the above definitions are straight-forward, the question when exactly a field is considered immutable is a tricky one, as even otherwise immutable fields are commonly initialized to some value. I therefore adopt the following definition: An object’s field is immutable if it is never written to outside of the dynamic extent of that object’s constructor. Note, however, that this definition is only an approximation; not all initialization happens inside the constructor. In particular, cyclic data structure or Java beans are frequently initialized outside the constructor [HP09]. Also, arrays, by their very nature, lack a constructor; thus, arrays were

¹¹ Static fields are not considered in this analysis. Compared to the number of field instances, the number of static fields is insignificant.

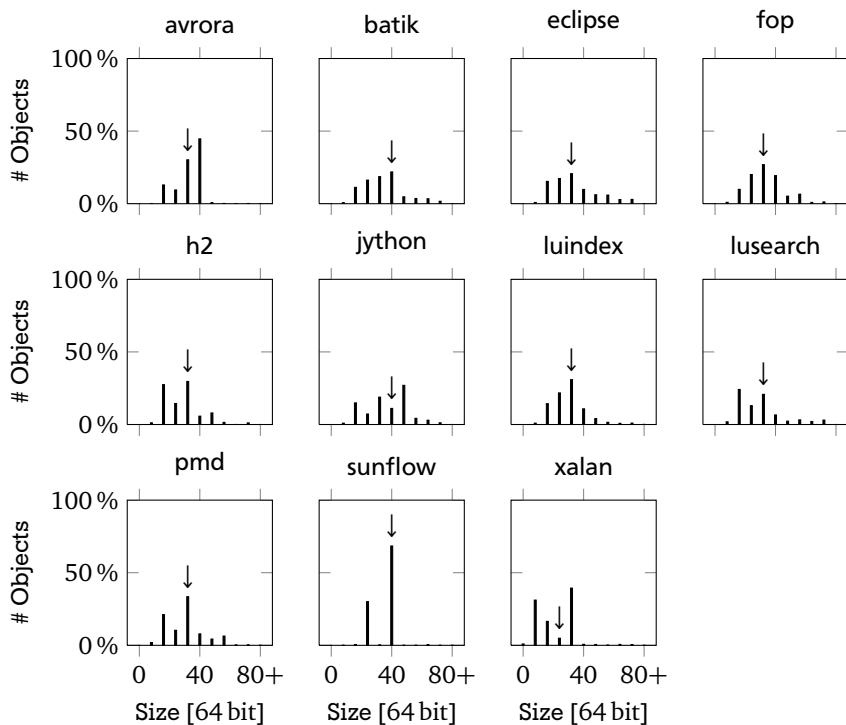


Figure 5.21a: The distribution of object sizes (excluding the header) with median marked (↓) for the Java benchmarks. Each bin is 8 bytes wide, the size of a pointer.

excluded from this analysis. Note furthermore that the above definition differs from some definitions of immutability found elsewhere [GJS⁺11, GPB⁺06]. In particular, a field may be immutable in a particular program run only. Static analyses for identifying immutable classes and objects [PBKM00] may thus be unable to detect this fact. But the tailored dynamic analysis I use sidesteps these limitations; unlike a static analysis it need not be overly conservative.

I implemented a tailored analysis¹² using DiSL to measure immutability. Figure 5.22a depicts the fraction of instance fields that are never mutated during the course of the respective benchmark, except, of course, during construction.

¹² See <http://www.disl.scalabench.org/modules/immutability-disl-analysis/>.

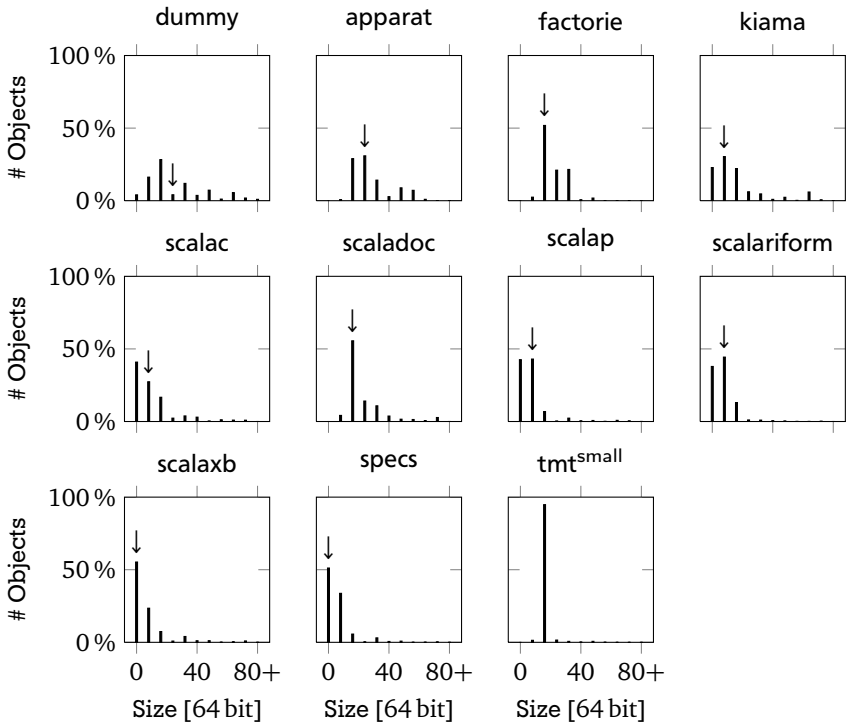


Figure 5.21b: The distribution of object sizes (excluding the header) with median marked (\downarrow) for the Scala benchmarks. Each bin is 8 bytes wide, the size of a pointer.

Figure 5.22b contrasts this with the fraction of fields that are per-class immutable. In other words, Figure 5.22b provides a static view of the program, whereas the view of Figure 5.22a is a dynamic one. As can be seen, these two views differ significantly. That being said, the Scala benchmarks in general exhibit a higher fraction of immutable fields than their Java counterparts—both per-object and per-class.

Figures 5.22a and 5.22b considered each field individually. However, an object may contain both mutable and immutable fields, rendering the entire object mutable if it contains just a single mutable field. Figures 5.23a and 5.23b thus consider object and class immutability, respectively. The Scala benchmarks consistently exhibit a larger fraction of immutable classes than the Java benchmarks: 79.67% and 52.06%, respectively. What is furthermore interesting to observe is that the

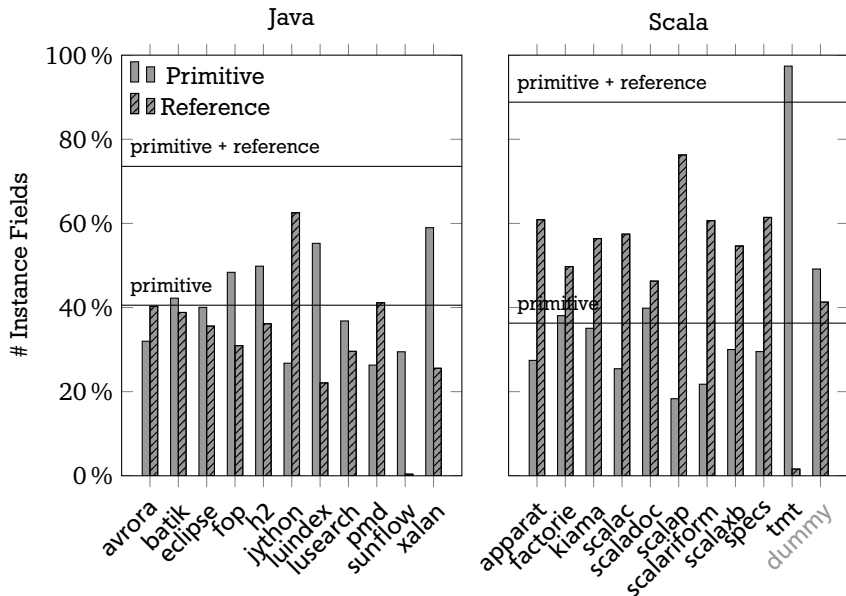


Figure 5.22a: Fraction of primitive and reference *instance* fields that are per-object immutable (including averages without dummy)

numbers from Figure 5.23b (immutable classes) for the Scala benchmarks (excluding dummy) almost exactly mirror those from Figure 5.22b (per-class immutable fields); a mixture of mutable and immutable fields within the same class is extremely rare in the Scala programs—but not in Java programs. This is at least partly explained by the smaller size of Scala objects (cf. Section 5.4.10); for objects with just a single field immutability is an all-or-nothing proposition.

5.4.12 Zero Initialization

Related to object allocation is zero initialization, which is mandated by the JVM specification [LYBB11]; depending on its type, every field is guaranteed to be initialized to a “zero value” of `0`, `false`, or `null`, respectively. This seemingly simple operation has a surprisingly large impact on performance [YBF⁺11]—and is not always strictly necessary: If an object’s constructor initializes a field explicitly by

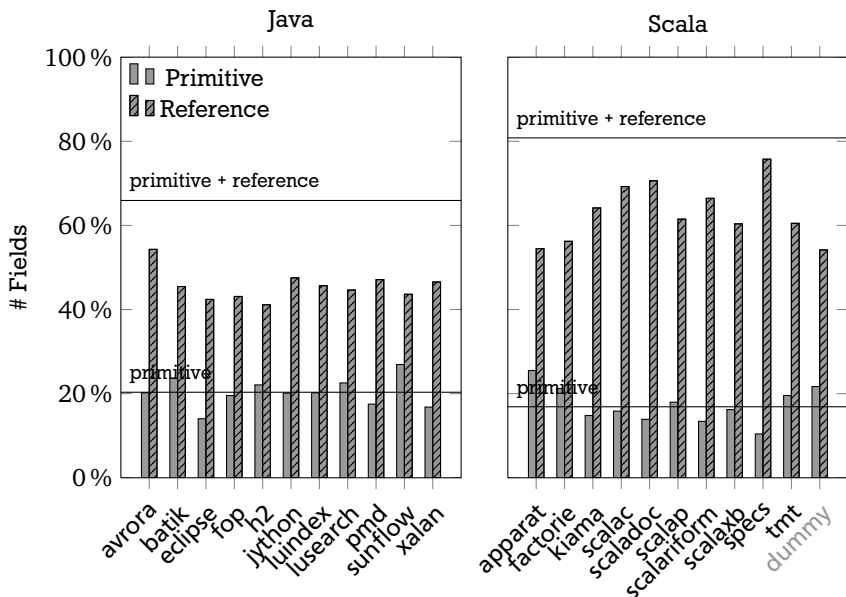


Figure 5.22b: Fraction of primitive and reference fields that are per-class immutable (including averages without dummy)

assigning it a value (including a zero value), the field’s implicit initialization to a zero value by the JVM’s allocation logic was unnecessary.

I used a tailored dynamic analysis¹³ written in DiSL¹⁴ to measure to what extent such unnecessary zeroing occurs in practice, as it hints at an optimization opportunity. This analysis considers zeroing of an instance field unnecessary if the following condition is met: The field is assigned in the dynamic extent of the constructor without being read prior to the explicit assignment. In particular, zeroing of a field that is neither read nor written to in the dynamic extent of the constructor is *not* considered unnecessary.

Note that the above condition does not take into account the fact that the JVM, in order to elide unnecessary zeroing, has not only to ensure that the program does not observe the uninitialized field, but also that the garbage collector remains un-

¹³ See <http://www.disl.scalabench.org/modules/immortality-disl-analysis/>.

¹⁴ See <http://www.disl.scalabench.org/modules/immortality-disl-analysis/>.

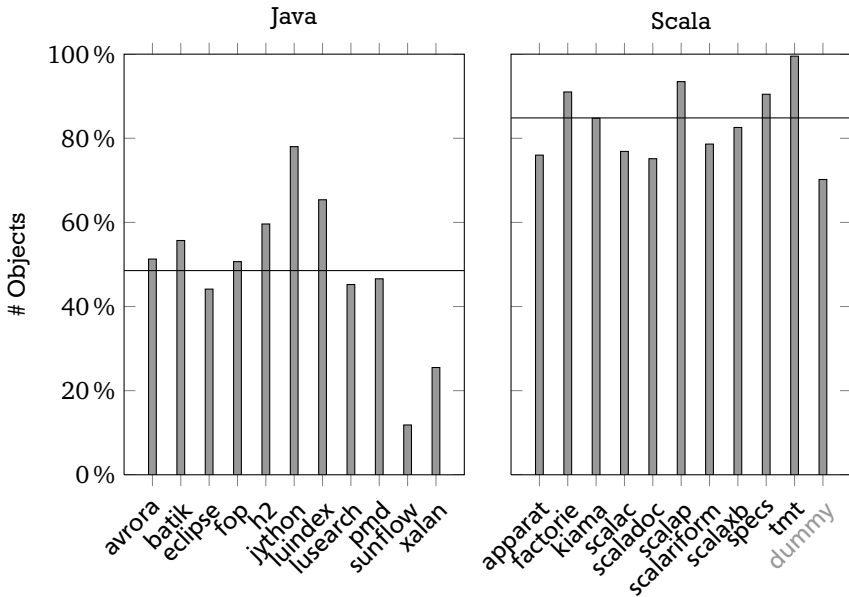


Figure 5.23a: Fraction of immutable objects (including averages without dummy)

aware of the field’s uninitialized state.¹⁵ But this source of imprecision is inevitable if the metric should be VM-independent.

Figures 5.24a and 5.24b depict the extent to which zeroing of instance fields is unnecessary, distinguishing between fields of primitive (`int`, `double`, etc.) and reference type. In general, zeroing of reference fields is necessary less often than zeroing of primitive fields; in other words, the initial value is `null` less often than `0` or `false`. Furthermore, reference fields play a larger role for the Scala benchmarks (57.6%) than for the Java benchmarks (40.0%). The sole exception is the `tmt` benchmark, which suffers from excessive boxing of primitive values [SMSB11]; almost all `Double` instances (accounting for 97.87% of the objects allocated) have their primitive value field explicitly set in the constructor.

¹⁵ This significantly complicates optimizing away unnecessary zeroing in the Oracle HotSpot VM, which implements this intricate optimization [YBF⁺11].

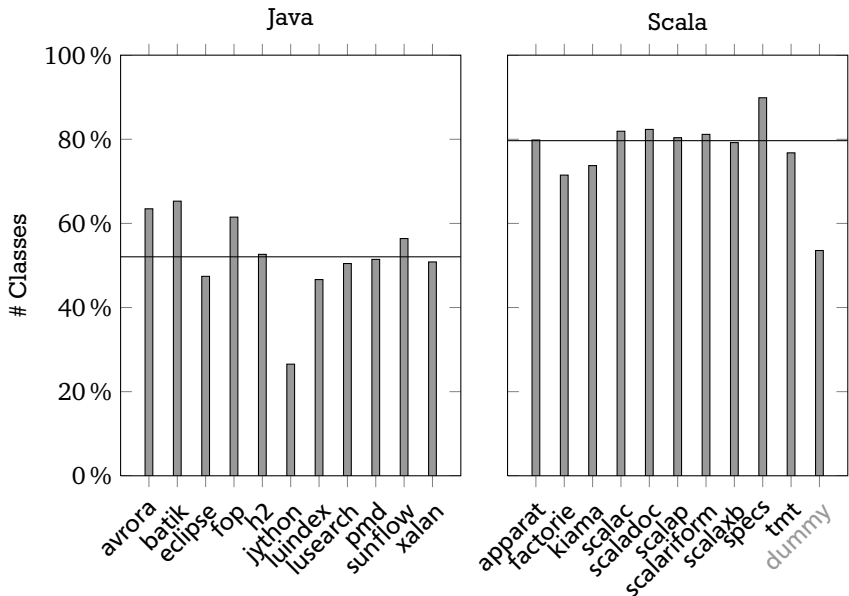


Figure 5.23b: Fraction of immutable classes (including averages without dummy)

5.4.13 Sharing

Many of the Java benchmarks and some of the Scala benchmarks are multi-threaded. It is thus of interest to what extent objects are shared between different threads. The metric employed here is “sharing by actual usage” as used by Mole et al. [MJK12]; an object is considered shared only if at least one of its fields is accessed by a thread other than the allocating thread. Unlike Mole et al. but consistent with my custom immutability analysis, I track sharing at the level of individual fields. This makes it possible to recognize situations where only parts of an object are accessed by different threads; this partial sharing contrasts with full sharing, where all fields of an object are read or written to by another thread.

I used a tailored dynamic analysis¹⁶ written in DiSL to track read and write accesses by the different threads, distinguishing between the thread that allocated the object in question and all other threads. For the purpose of this analysis, arrays are treated as objects with two pseudo-fields: one keeping the array’s length and

¹⁶ See <http://www.disl.scalabench.org/modules/sharing-disl-analysis/>.

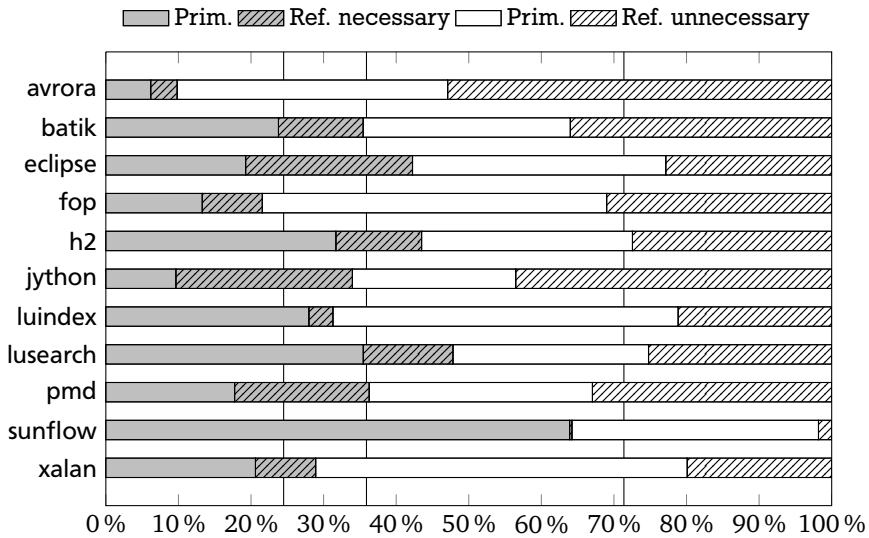


Figure 5.24a: Necessary and unnecessary zeroing of primitive and reference instance fields, respectively, for the Java benchmarks (including averages)

one keeping its components (treated as a single field). Static fields, which others have found to be frequently shared [KMJV12], are not taken into account. Even if not shared, they are, as globally accessible resources, hard to optimize for.

One minor source of imprecision is that the DiSL instrumentation is not yet active while the JVM is bootstrapping. For the few objects allocated during that time, the dynamic analysis is obviously unable to determine the allocating thread; the object and its fields may be incorrectly flagged as shared.

The results of the analysis are depicted in Figures 5.25a, 5.25b and 5.25c. Only a small fraction of objects allocated during a benchmark invocation is shared among threads, the big exceptions being *avroa* and *lusearch*, two Java benchmarks from the DaCapo suite.¹⁷ These two benchmarks are again quite different as there is just one type predominantly shared among threads in *avroa* (`RippleSynchronizer.WaitLink`) whereas there are several in *lusearch* (`FSIndexInput`, `CSIndexInput`, `char[]`, `Token`, etc.). Figures 5.25a and

¹⁷ The dummy benchmark also exhibits a high fraction of shared objects, but allocates very few objects overall, namely 7300.

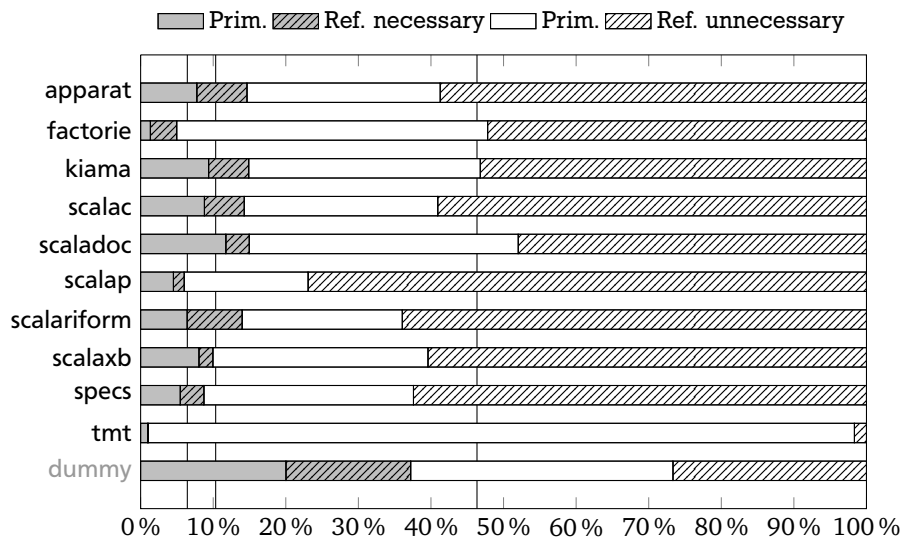


Figure 5.24b: Necessary and unnecessary zeroing of primitive and reference instance fields, respectively, for the Scala benchmarks (including averages without dummy)

5.25b illustrate the fraction of individual instances shared among threads, which on average is quite low. But as Figure 5.25c illustrates, the number of classes for which at least one instance is shared is surprisingly high: 28.94% (Java) and 11.56% (Scala); thus, all these classes potentially require some synchronization.

Such synchronization, however, is superfluous if the object in question is of an immutable class (cf. Section 5.4.11). Of those (non-array) objects whose fields are read but not written to by multiple threads, more than 53.57% belong to a (potentially) immutable class in the case of the Java programs. In the case of the Scala programs, this fraction is even higher; more than 87.06% of objects belong to a class whose fields are not mutated after construction.

5.4.14 Synchronization

Conceptually, on the Java virtual machine every object has an associated monitor. A thread acquires such a monitor either explicitly by entering a **synchronized** block (executing a **monitorenter** instruction) or implicitly by entering a

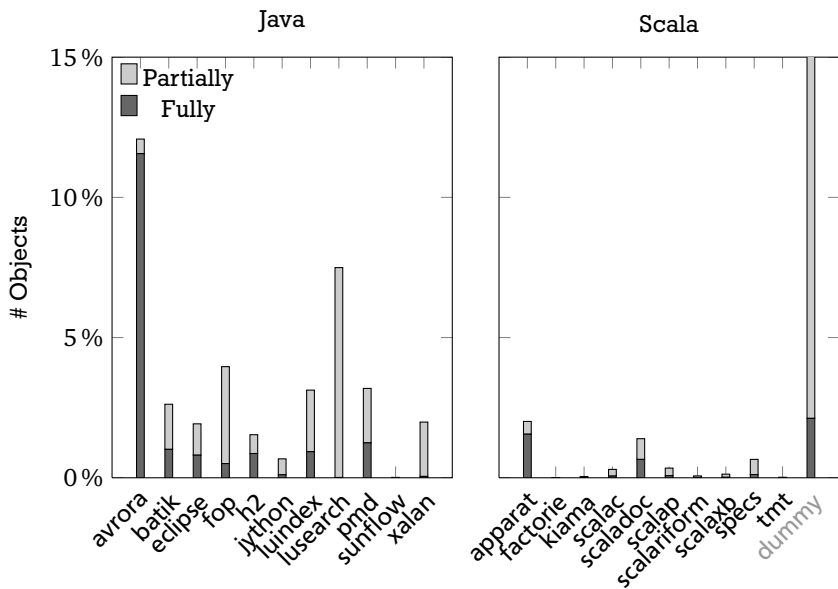


Figure 5.25a: Partially and fully shared objects with respect to read accesses, i.e., with fields read by multiple threads

synchronized method and releases it again after exiting the block (**monitorexit**) or method, respectively. This locking facility, alas, comes at a cost.

To avoid both the runtime and memory cost of multi-word “fat locks” kept in a data structure separate from their associated objects, researchers have developed “thin locks” [BKMS98, ADG⁺99], which require only a handful of bits in the header of the object itself. These lock compression techniques exploit the fact that most locks are never subject to contention by multiple threads. If they are, however, the affected locks must be decompressed again. Biased locks [RD06, PFH11] go one step further than thin locks by exploiting the fact that most locks are not only never contended for, but are also only ever owned by one thread. The resulting lock further improves the runtime cost of lock acquisition and release, if not the lock’s memory cost.

To be effective, both thin locks and biased locks rely on two assumptions, which I have validated using a tailored dynamic analysis written in DiSL: First, most locks are only acquired by a single thread. Second, nested locking is shallow. The former assumption is fundamental to biased locking whereas the latter affects all com-

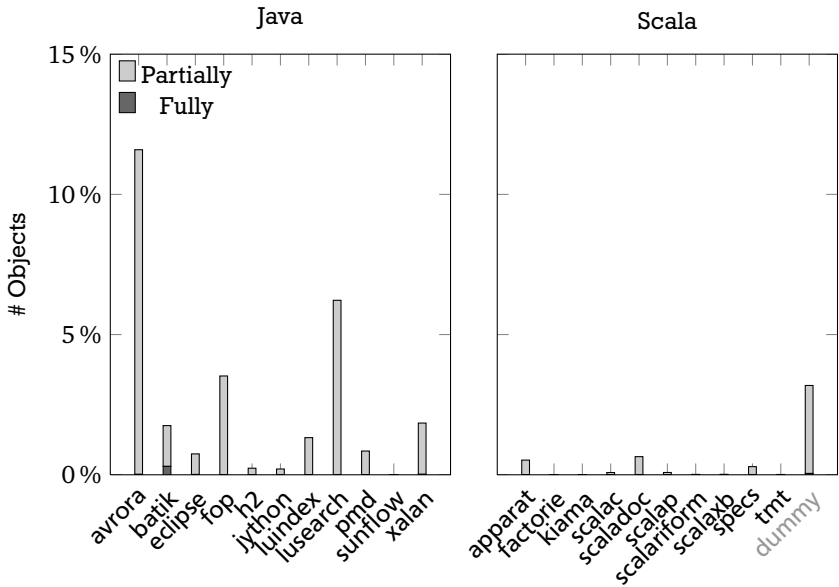


Figure 5.25b: Partially and fully shared objects with respect to write accesses, i.e., with fields written to by multiple threads

pression techniques that reserve only a few bits in the object header for the lock’s recursion count.¹⁸ Especially in light of the deeply recursive calls common in Scala programs (cf. Section 5.4.3), the latter metric is of interest. I used a tailored dynamic analysis¹⁹ to obtain the necessary data. Like Bacon et al. [BKMS98], I also record the total number of objects, total number of synchronized objects, and total number of synchronization operations. The results are depicted in Figure 5.26.²⁰

The vast majority of objects are only ever synchronized on by a single thread; on average only 0.49 % (Java) respectively 1.75 % (Scala) of all locks are owned by more than one thread. This makes thin locks in general and biased locks in particular effective for both Java and Scala programs. Nevertheless, in both benchmark suites there exist notable outliers: 3.37 % (sunflow, Java) and 13.68 % (tmt, Scala), respectively. This is countered by the very small fraction of objects these two benchmarks synchronize on at all; it is virtually zero in both cases.

¹⁸ Techniques exist to store the recursion count outside the thin lock [RD06].

¹⁹ See <http://www.disl.scalabench.org/modules/monitoring-disl-analysis/>.

²⁰ Explicit locks from `java.util.concurrent.locks` were not considered.

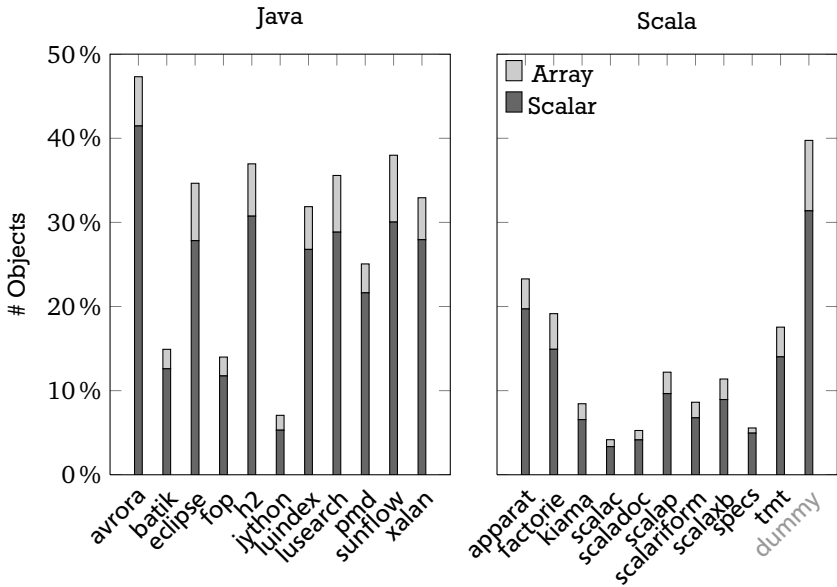


Figure 5.25c: Fraction of scalar and array types that are shared, i.e., for which at least one instance is shared

Figures 5.27a and 5.27b confirm the findings of Bacon et al. [BKMS98] that nested locking does not prevent the use of thin locks for both the Java and Scala benchmarks; all locking operations are relatively shallow. In fact, only four benchmarks (*avrora*, *eclipse*, *h2*, and *xalan*) exhibit more than ten levels of recursive synchronization on the same object. That these are all Java benchmarks also shows that Scala programs, despite their tendency towards deeply recursive calls (cf. Section 5.4.3), only exhibit shallow locking.

I finally assess to what extent code from different sources (cf. Section 3.2.3) employs synchronization. Of the lock operations performed by the Java benchmarks, on average 68.8% of operations target objects from the Java runtime and 31.2% of operations target objects from the (Java) application and its libraries. For the Scala benchmarks, an even higher fraction of operations targets Java runtime objects (77.4%). Naturally, Java libraries play only a small role (1.1%), whereas objects from both the Scala runtime (8.5%) and the Scala application (13.0%) are targeted by a fair number of lock operations. The small part of the Scala runtime written in Java (cf. Section 3.2.3) plays no role at all (0.0%).

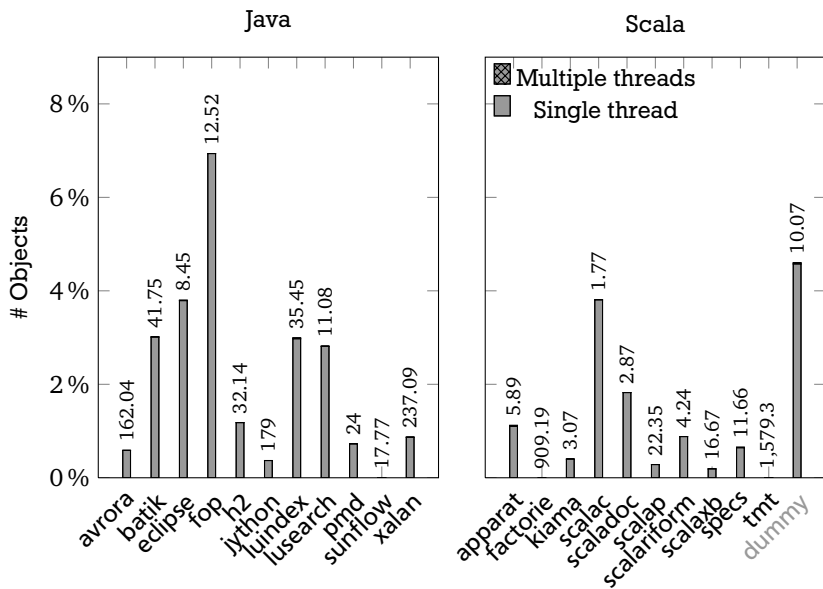


Figure 5.26: Fraction of objects synchronized on by one or more threads, together with the average number of lock operations per object

5.4.15 Use of Identity Hash-Codes

Not only has every object on the JVM an associated monitor but also an associated hash-code, which the programmer may compute by invoking the object's `hashCode` method. Not every class declares its own `hashCode` method, however. It is therefore the virtual machine's responsibility to provide some implementation of `Object.hashCode()`.²¹ As the return value of this method, the so-called identity hash-code, must not change across invocations, it is tempting to simply store it in an extra slot in the object's header. This, however, wastes space, since the identity hash-codes of many objects are never queried.

This observation led to research on header compression techniques that do not store the identity hash-code but rather turn the object's address into its hash-code [BFG06]. This address-based hashing scheme eliminates the space overhead

²¹ Even if a class overrides `Object.hashCode()`, its implementation is always accessible through the `static System.identityHashCode(Object)` method.

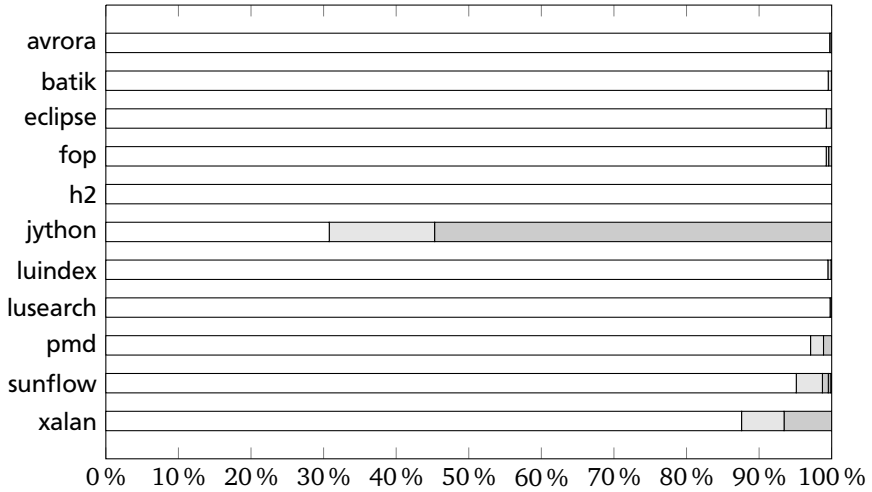


Figure 5.27a: The maximum nesting depth reached per lock for the Java benchmarks, ranging from 0 (□) to 10 or more (■)

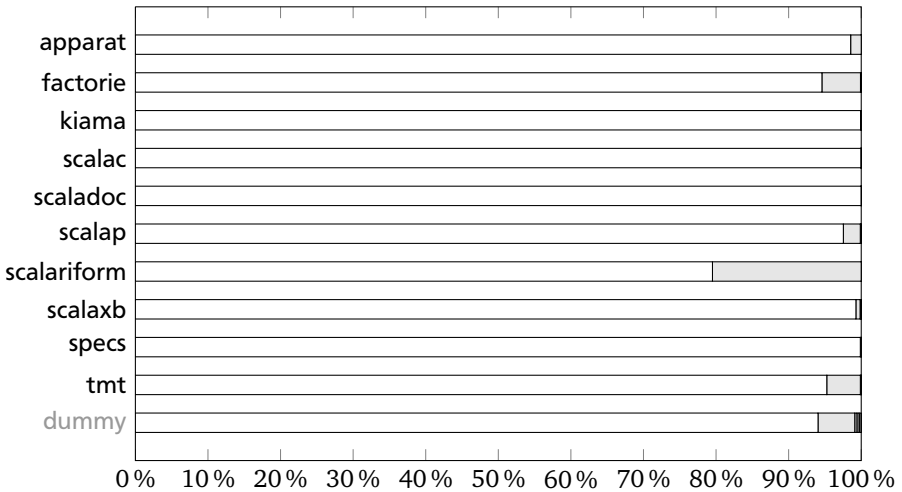


Figure 5.27b: The maximum nesting depth reached per lock for the Scala benchmarks, ranging from 0 (□) to 10 or more (■)

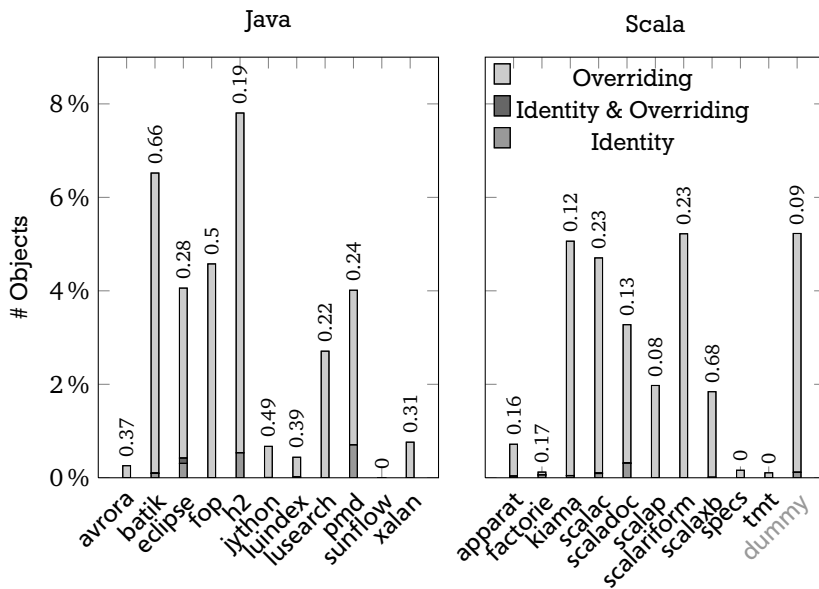


Figure 5.28: Fraction of objects hashed using the identity hashCode method, an overriding implementation thereof, or both together with the average number of hash operations per object

completely, but does not work as-is for copying collectors, a frequently used collector design. Such collectors must add an extra header slot whenever they copy an object whose identity hash-code has already been computed to a new address.

To assess the usage of hash-codes I used a tailored dynamic analysis²² written in DiSL. But as DiSL relies on bytecode instrumentation, only calls to `System.identityHashCode(Object)` and `Object.hashCode()` made from methods with a bytecode representation are covered; calls made from native code are not. I believe this to be a negligible source of dynamic imprecision.

The results of the analysis are depicted in Figure 5.28 for the Java and Scala benchmarks, respectively. I distinguish between objects whose identity hash-code was taken and objects whose hash-code was provided by a custom, overriding implementation of `hashCode()`. Only the former objects require, if hashed and then moved by the garbage collector [BFG06], additional space in the object header to

²² See <http://www.disl.scalabench.org/modules/hashcode-disl-analysis/>.

permanently keep their identity hash-code. Note that virtually no objects have both their identity and custom hash-code taken. But if this is the case, this is almost exclusively due to a overriding `hashCode()` implementation performing a super-call to `Object.hashCode()`. The only notable exception to this rule are `String` objects, where the same object is kept in two hashtables that use `Object.hashCode()` and `System.identityHashCode(Object)`, respectively, as hash function.

That being said, at most 0.70% of objects (pmd) have their identity hash-code taken, with an average of 0.16% and 0.06% for the Java and Scala benchmarks, respectively. Address-based hashing is thus expected to work as good if not better for Scala programs than it does for Java programs.

5.5 Summary

In this chapter, I have compared Scala and Java programs taken from two benchmarks suites. This led to the following findings about the behaviour of Scala programs on the Java Virtual Machine:

Instruction Mix. Scala and Java programs differ significantly in their instruction mix. In particular, most Scala programs favour inter-procedural over intra-procedural control-flow, which is reflected in the instruction mix.

Call-Site Polymorphism. Although polymorphism plays a larger role for Scala than for Java code, the overwhelming number of call sites is effectively monomorphic and accounts for the majority of calls. Inlining is thus expected to be as effective for Scala code as it is for Java code.²³

Stack Usage and Recursion. Scala programs 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 argument are rarely called. This may negatively affect the economics of method inlining, as the optimization propagates less information into the inlined method.

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

²³ As Chapter 6 shows, inlining is even more effective for Scala code; in fact, it becomes a necessity.

Use of 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 benefit from the **invokedynamic** instruction than dynamically-typed languages like Clojure, Groovy, Ruby, or Python.

Use of 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.

Garbage-Collector Workload. Objects in Scala programs are even more likely to die “young” than Java objects. Closures and boxed primitives contribute significantly to this. Captured variables and rich primitives are only a minor contribution, however.

Object Churn. Despite the above, objects in Scala programs do not necessarily die “close” to their allocation site.

Object Sizes. Very small objects play a significantly larger role in Scala than in Java.

Immutability. Immutable fields and objects play an even larger role in Scala programs than in Java programs.

Zero Initialization. The implicit zeroing of fields during allocation is mostly unnecessary, in particular for Scala programs.

Synchronization. Scala code does not negatively affect common lock implementations (thin, biased locking) optimized for Java code.

Use of Identity Hash-Codes. The identity hash-code associated with objects is rarely used by Scala and Java programs.

Taken together, these findings provide the first comprehensive overview about both code-related [SMSB11] and memory-related [SMS⁺12] behaviour of Scala programs on the Java Virtual Machine. They thus sharpen the JVM implementers’ intuition as to what the differences are between Java and Scala code when viewed from the JVM’s perspective.



6 An Analysis of the Impact of Scala Code on High-Performance JVMs

In the previous chapter, I have used VM-independent metrics to shed light on some of the key differences between Java and Scala code. While such independence from any particular VM is crucial during the inception of a benchmark suite, as it helps to avoid a benchmark selection which favours just a single VM, VM-independent metrics can only point VM implementers in the right direction; by their very nature, they cannot pinpoint a particular piece of performance-critical implementation that needs to be adapted to better accommodate Scala code. In this chapter, I thus go one step further and use my Scala benchmark suite to compare and contrast the ability of several modern high-performance JVMs to cope with Java and Scala code, respectively.

First, Section 6.1 describes my experimental setup. Next, Section 6.2 presents an initial investigation into the Java and Scala performance of the selected VMs. Thereafter, Sections 6.3 and 6.4 explore two hypotheses to explain the observed performance differences. Finally, a discussion of the findings concludes this chapter.

The research presented in this chapter has not been published before.

6.1 Experimental Setup

To get a realistic impression of how Scala code impacts different JVMs, it is crucial to consider a broad selection of modern virtual machines. For this experiment I have thus chosen five Java virtual machine configurations overall, the first four being based on well-known production JVMs and the fifth being based on the premier research JVM [DFD10]:

OpenJDK 6 The OpenJDK 64-bit Server VM (build 19.0-b09, mixed mode) with the IcedTea 1.10.6 class library

OpenJDK 7u The OpenJDK 64-bit Server VM (build 23.0-b15, mixed mode) with the generic OpenJDK class library,¹ built from Mercurial tag `jdk7u4-b11`

JRockit The Oracle JRockit® VM (build R28.1.3-11-141760-1.6.0_24-20110301-1432 linux-x86_64, compiled mode)

¹ Using IcedTea 2.0.1 was not an option, due to a build failure under Debian GNU/Linux.

J9 The IBM J9 VM (build 2.4, JRE 1.6.0 IBM J9 2.4 Linux amd64-64 jvms6460sr9-20110203_74623)

Jikes RVM The Jikes Research VM [AAB⁺05] in its production configuration, built from Mercurial tag 3.1.2,² with the GNU Classpath 0.97.2 class library

The three Open Source VMs (both OpenJDKs and Jikes RVM) were built from scratch, but without manual tuning. For example, for Jikes RVM I refrained from both building a profiled boot-image and tuning the VM’s compiler DNA (cf. Section 6.3). For the IBM J9 VM I have disabled ahead-of-time compilation (`-Xnoaot` command-line option); disallowing precompiled code avoids favouring J9 when measuring startup performance. Aside from this J9-specific configuration and unless otherwise noted command-line options were only used to set the VM’s heap size (cf. Section 6.1.1). The validity of the following findings thus depends on the assumption that the VMs’ default configuration is already well-tuned for a variety of (Java) workloads. This assumption is reasonable, as most users of both production and research VMs use them without manual tuning, thereby creating a strong incentive for VM implementers to tune their respective VM for a broad range of workloads.

The Java and Scala workloads stem from the DaCapo 9.12 benchmark suite [BGH⁺06] and from the Scala benchmark suite developed for this thesis, respectively. Each benchmark’s default input size was used.

All measurements were conducted on a 4-core/8-thread Intel Core i7-870 processor clocked at 2.93 GHz with 4×32 KiB L1 data and instruction caches, 4×256 KiB L2 cache, 8 MiB L3 cache, and 4096 MiB RAM running a 64-bit version of GNU/Linux (Debian “Squeeze” 6.0.4, kernel 2.6.32). This machine serves as a good example of a modern multi-core commonly found in desktop and server machines.

6.1.1 Choosing Heap Sizes

As I am primarily interested in the JVMs’ ability to cope with Java and Scala *code*, the performance impact of garbage collection needs to be minimized. But simply fixing a large heap size, say 1 GiB, is counterproductive, as locality effects can cause performance to deteriorate at excessive heap sizes. In a single-JVM setting, one therefore traditionally chooses some small multiple³ of the minimum heap size in

² Jikes RVM was built with changeset `f47765eca54f` backed out, due to a build failure under 64-bit Debian GNU/Linux. See <http://jira.codehaus.org/browse/RVM-942>.

³ Two to six times the minimum heap size is common [BGH⁺06, BMG⁺08, YBF⁺11].

Benchmark	OpenJDK 6	OpenJDK 7u	JRockit	J9	Jikes RVM
avrora	4	4	16	4	24
batik	24	—	20	24	—
eclipse	100	104	68	84	—
fop	28	32	24	40	—
h2	284	280	212	384	—
jython	24	24	20	40	64
luindex	4	4	16	8	24
lusearch	4	4	16	4	28
pmd	32	36	20	32	52
sunflow	4	4	16	12	28
tomcat	12	12	16	16	—
tradebeans	20	16	40	28	—
tradesoap	20	24	32	28	—
xalan	4	4	16	8	40
actors	4	4	16	4	—
apparat	20	16	16	24	—
factorie	104	104	84	148	—
kiamia	4	4	16	12	36
scalac	28	36	24	44	80
scaladoc	32	36	28	44	80
scalap	4	4	16	8	32
scalariform	4	4	16	8	32
scalatest	12	16	16	12	—
scalaxb	20	16	16	16	40
specs	4	4	16	8	—
tmt	32	32	32	36	—

Table 6.1: The minimum heap size (in MiB) required to successfully run a single iteration of the respective benchmark (measured at 4 MiB granularity)

which a given benchmark runs successfully without raising an `OutOfMemoryError`; this multiple is then used as heap size during the actual measurement runs. This approach, however, is not applicable when comparing multiple JVMs, because different JVMs sometimes require significantly different heap sizes, as set by the `-Xms/-Xmx` command-line options, to pass a benchmark run. Table 6.1 illustrates these discrepancies for the five JVM configurations considered. Of the five JVMs, Jikes RVM is the only JVM which consistently fails to execute several benchmarks (7

Java benchmarks; 6 Scala benchmarks); this is mostly due to the GNU Classpath class library lacking classes required by the benchmarks in question.

The different heap and object layouts used by the five JVMs only explain part of the discrepancies in heap consumption. What also contributes to the observed differences is the simple fact that different JVMs interpret the `-Xms/-Xmx` options differently: Due to its meta-circular nature, Jikes RVM considers all machine code generated by its just-in-time compilers to be part of the application's heap, whereas OpenJDK considers generated machine code to be part of a dedicated code-cache area which is separate from the heap itself. These differences in interpretation explain why Jikes RVM requires a larger minimum heap size than its four competitors; for example, on the kiama benchmark it requires a heap size nine times larger than that required by OpenJDK, namely 36 MiB rather than 4 MiB.

These differences in interpretation unfortunately prevent the traditional approach of heap sizing from being applied. If one were to use a different multiple of the minimum heap size for each JVM/workload combination, then the fact that Jikes RVM considers generated machine code to be part of the heap would cause correspondingly more memory to be awarded to Jikes RVM than to JVMs with a different interpretation of `-Xms/-Xmx`—even though the amount of generated machine code stays more or less constant. Conversely, if one were to use the same multiple of the single smallest minimum heap size across *all* JVMs for a particular benchmark, then one would severely disadvantage a JVM like Jikes RVM. In this case, Jikes RVM would likely not even be able to run benchmarks like kiama, since it requires a minimum heap size nine times larger than the smallest minimum heap size, namely that of OpenJDK.

I therefore chose a different approach:⁴ For each JVM/workload combination, I empirically determined the optimal heap size, i.e. the heap size for which the given JVM delivered the fastest startup performance. For this purpose, I measured the benchmark's first iteration across a range of different heap sizes, from twice the minimum heap size required (cf. Table 6.1) up to eight times that heap size or 128 MiB, whichever is smaller. Again, measurements were done at the granularity of 4 MiB. Table 6.2 summarizes the results.

This approach to heap sizing may be susceptible to jitter; small changes in the heap size may result in large changes in execution time. Table 6.2 therefore also states the coefficients of variation for the latter, as derived from a five-measurement interval around the optimal heap size (-8 MiB, -4 MiB, ± 0 MiB, $+8$ MiB, $+8$ MiB). For 69 out of 112 JVM/workload combinations, the coefficient does not exceed 2.0%, indicating that performance has reached a steady state (cf. Section 6.1.2)

⁴ The members of the `jikesrvm-researchers` mailing list in general and Steve Blackburn in particular provided valuable feedback on this approach.

Benchmark	OpenJDK 6		OpenJDK 7u		JRockit		J9		Jikes RVM	
avro	128	2.74	68	1.25	20	2.22	72	2.12	176	1.27
batik	140	0.45	—	—	160	0.35	136	8.62	—	—
eclipse	680	1.52	732	2.49	268	1.42	548	3.12	—	—
fop	212	0.26	116	6.11	180	0.31	284	0.90	—	—
h2	1072	2.59	468	2.16	1380	3.11	2476	2.71	—	—
jython	184	0.78	156	2.34	152	0.69	284	2.43	468	1.17
luindex	36	1.80	40	3.44	128	1.29	48	3.52	136	1.89
lusearch	60	7.99	124	3.41	128	1.46	120	2.10	220	2.75
pmd	192	1.92	276	1.26	132	1.72	244	3.05	300	1.95
sunflow	100	1.27	120	2.21	96	2.60	128	2.39	168	0.88
tomcat	116	0.85	124	3.58	104	0.47	104	0.76	—	—
tradebeans	160	2.30	104	4.18	260	5.84	212	1.16	—	—
tradesoap	120	3.56	192	1.48	256	2.95	192	3.50	—	—
xalan	80	4.00	100	6.07	116	0.84	92	5.96	244	2.16
actors	92	1.14	112	0.87	68	1.30	116	1.70	—	—
apparat	60	2.89	88	7.09	72	1.00	32	7.05	—	—
factorie	500	4.37	280	6.72	644	2.38	1124	3.72	—	—
kiama	64	1.00	104	2.71	96	0.34	124	0.88	204	1.46
scalac	224	1.49	264	4.42	176	0.18	328	2.19	468	1.79
scaladoc	92	2.55	276	2.68	172	0.37	308	1.46	328	1.03
scalap	72	0.85	112	2.18	100	0.34	76	4.07	208	1.02
scalariform	128	0.61	120	2.81	124	0.37	100	1.16	244	1.06
scalatest	96	0.92	96	1.14	92	0.67	92	0.80	—	—
scalaxb	92	0.56	72	1.99	40	0.64	96	0.94	220	5.16
specs	116	0.54	104	1.17	120	0.36	88	0.71	—	—
tmt	256	2.49	252	1.26	248	0.25	256	0.63	—	—

Table 6.2: The optimal heap size (in MiB) to run a single iteration of the respective benchmark (measured at 4 MiB granularity in the range of 1 x to 8 x the minimum heap size) together with the coefficient of variation (%) for a five-measurement interval around that heap size

also with respect to minor heap-size adjustments. For 95 combinations, the coefficient stays below 3.0%.

The validity of my findings might also be threatened by the fact that heap sizes were chosen based on optimal startup performance, i.e. based on a single benchmark iteration. Unlike its competitors, Jikes RVM accrues more and more machine

code on the application's heap from iteration to iteration, as its optimizing compiler re-compiles methods. The effect is minor, though, and does not threaten validity; under normal circumstances the extra machine code rarely exceeds 1 MiB to 2 MiB. Inlining much more aggressively than the default, however, does have an effect: While it does not generate much extra machine code (less than 10 MiB), Jikes RVM's optimizing compiler consumes vastly more memory. In some cases, this even triggers an `OutOfMemoryError`, as the garbage collection workload temporarily becomes too high (cf. Section 6.4).

6.1.2 Statistically Rigorous Methodology

Throughout this chapter, I use a statistically rigorous performance evaluation methodology [GBE07] to compute both startup and steady-state performance of the benchmarks under investigation: Overall startup performance is averaged across 30 VM invocations, each invocation running a single benchmark iteration [GBE07, Section 4.1], whose execution time is taken. The large number of invocations allows me to assume that the (independent) execution times follow a Gaussian distribution [GBE07, Section 3.2.1]. Overall steady-state performance is averaged across 15 VM invocations, whose steady-state performance is defined to be the arithmetic mean of the execution time of the first five consecutive iterations whose coefficient of variation is below 2.0% [GBE07, Section 4.2] or, if the benchmark fails to converge,⁵ the last 5 of 30 iterations overall. Due to the smaller number of invocations, I had to assume that the steady-state execution time average follows the Student's *t*-distribution (with 15 – 1 degrees of freedom) rather than a Gaussian distribution [GBE07, Section 3.2.2]. Where used, the confidence intervals around the quotient of two means were computed using Fieller's theorem, i.e. assuming that the means were sampled from Gaussian distributions even if a relatively small number of samples was taken.

Prior to the measured invocations, the benchmark was run once without contributing to the measurements. This extra invocation ensures that disk I/O does not adversely affect the first measured invocation; the extra invocation primes any disk caches and thereby a priori avoids an outlier in the measurements.

⁵ The following benchmarks failed to converge on multiple invocations: batik (OpenJDK 6), pmd (Jikes RVM), scalaxb (OpenJDK 7u), sunflow (Jikes RVM), and tradesoap (JRockit).

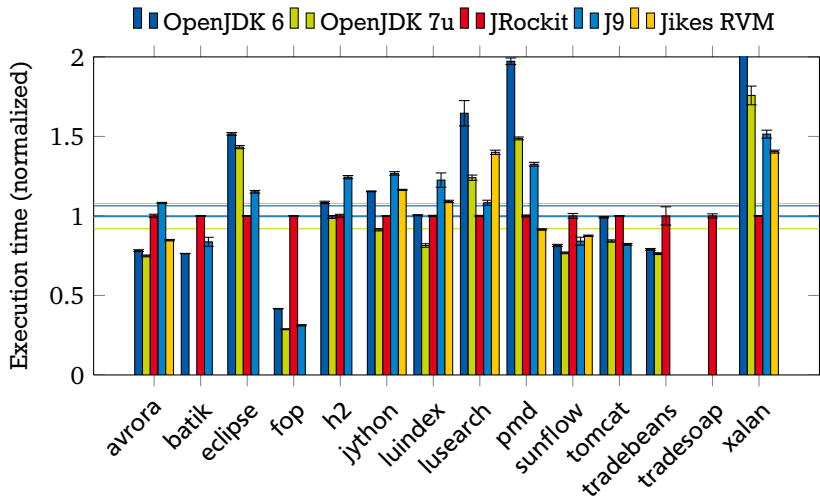


Figure 6.1a: Startup execution time of the Java benchmarks normalized to that of JRockit (Arithmetic mean \pm 95% confidence interval), including the geometric mean for each JVM

6.2 Startup and Steady-State Performance

In this section, I will give a first intuition of how well the five selected JVMs cope with Java and Scala code, respectively. I thus measured their startup and steady-state performance using the methodology described in Section 6.1.2.

Startup Performance

Figures 6.1a and 6.1b depict the startup execution time of the Java and Scala benchmarks, respectively, when run on one of the five JVMs under consideration. All execution times have been normalized to that of the respective benchmark run on the JRockit VM, which is the only JVM which consistently passes the benchmark harness's output verification for all the workloads: During startup, OpenJDK 7u fails on batik, both OpenJDKs and J9 occasionally fail on the twin benchmarks tradebeans and tradesoap, and Jikes RVM fails several benchmarks, including half of the Scala ones.

Aside from the occasional failures, several of the JVMs exhibit some performance pathologies during startup. OpenJDK 6, for example, performs badly on xalan. Although at least in this particular case, the pathology could be mitigated by disabling

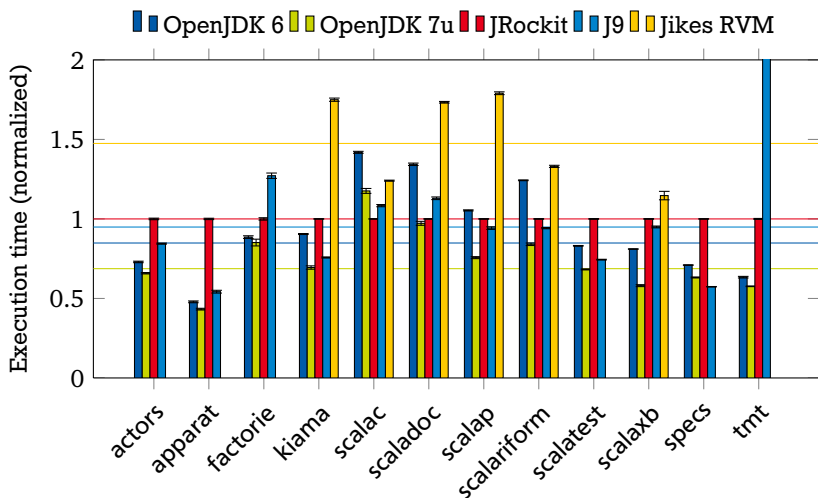


Figure 6.1b: Startup execution time of the Scala benchmarks normalized to that of JRockit (Arithmetic mean \pm 95 % confidence interval), including the geometric mean for each JVM

background compilation through a command-line option (-Xbatch), which causes OpenJDK 6 to achieve the same performance as the newer OpenJDK 7u, I refrained from any JVM-specific tuning to keep a level playing field. This is all the more justified as both OpenJDK 6 and 7u perform extremely well once reaching the steady-state for xalan (cf. Figure 6.2a); the “performance pathology” simply reflects a trade-off between startup and steady-state performance made by the implementers of OpenJDK.

That being said, Figures 6.1a and 6.1b already allow for some interesting observations: On the one hand, despite the aforementioned performance pathologies for individual benchmarks, the five JVMs exhibit remarkably similar average performance for the various Java benchmarks. On the other hand, the JVMs exhibit wildly different average performance for the Scala benchmarks. This indicates that the common case implementers of all five JVMs strive to optimize for is typical Java code, as typified by the benchmarks from the DaCapo benchmark suite, rather than Scala code, as typified by my novel Scala benchmark suite.

Also note that at least for Java code the Jikes Research VM delivers performance competitive with most production JVMs; it is only 1.4% slower than OpenJDK 6 and 17.2% slower than OpenJDK 7u, which are the slowest and fastest of the

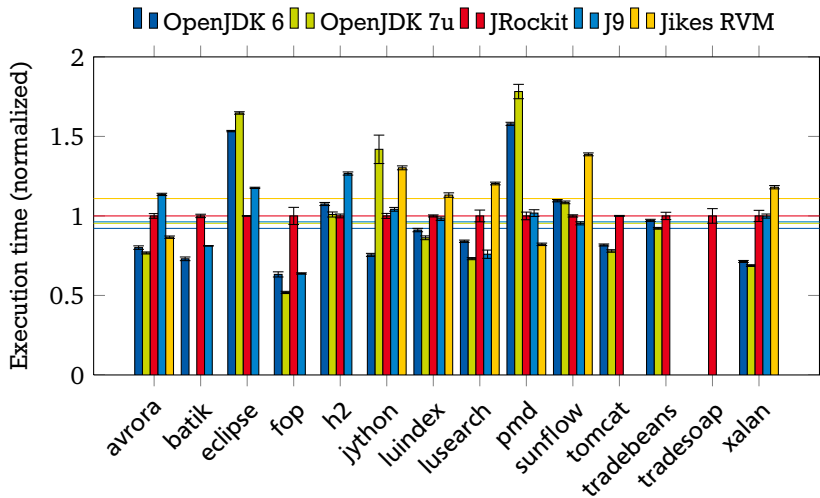


Figure 6.2a: Steady-state execution time of the Java benchmarks normalized to that of JRockit (Arithmetic mean \pm 95% confidence interval), including the geometric mean for each JVM

four production JVMs, respectively. The means have to be interpreted with care, however, as Jikes RVM suffers more failures on the benchmarks of the DaCapo 9.12 benchmark suite than the production JVMs'; this may bias the (geometric) mean.

Steady-state Performance

Figures 6.2a and 6.2b depict the steady-state performance of the five JVMs considered in this chapter. Again, the performance has been normalized to that of the JRockit VM, which is the only JVM that successfully passes output verification for all benchmarks. In particular, OpenJDK, J9, and Jikes RVM fail on some benchmarks before reaching their steady-state performance, even though they successfully passed output verification after a single iteration of the same benchmarks.

For the steady state, the observations made earlier about the startup performance still hold; in fact, they become even more pronounced: The JVMs' performance becomes even more similar for the Java benchmarks (means ranging from 0.92 to 1.11) and even more dissimilar for the Scala benchmarks (0.62 to 1.89).

OpenJDK 6 performs much better on the Scala benchmarks than on the Java benchmarks and is only beaten by its own successor, OpenJDK 7u. In contrast, Jikes RVM performs a lot worse than the other four JVMs on the Scala benchmarks,

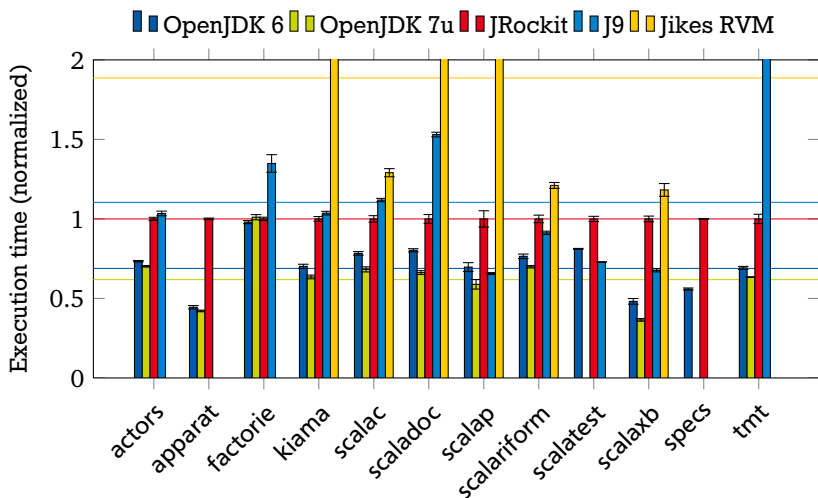


Figure 6.2b: Steady-state execution time of the Scala benchmarks normalized to that of JRockit (Arithmetic mean \pm 95% confidence interval), including the geometric mean for each JVM

even though the research VM delivers Java performance competitive to that of the production JVMs. While Jikes RVM is only 10.9% slower than the slowest (JRockit) and 20.4% slower than the fastest (OpenJDK 6) of the four production JVMs on the Java benchmarks, performance is abysmal on the Scala benchmarks, with Jikes RVM being 70.8% slower than slowest (J9) and even 200.5% slower than the fastest competitor (OpenJDK 7u). For `scaladoc` at least the abysmal performance can be explained by the fact that Jikes RVM spends about one third of the benchmark's execution time in the operating system's kernel, executing just a handful of native methods,⁶ whereas the other JVMs spend only a small fraction of their execution in kernel mode. For the other five Scala benchmarks that Jikes RVM runs successfully, however, no such simple explanation presents itself.

Of course, the above comparison needs to be taken with a grain of salt, as all JVMs except for JRockit fail on one or more benchmarks and not always on the same subset; thus, strictly speaking the geometric means are not comparable. Nevertheless, Figures 6.1a to 6.2b already give a good indication that some JVMs are

⁶ The native methods executed are `VMFile.exists`, `VMFile.isDirectory`, `VMFile.isFile`, and `VMFile.toCanonicalForm`, all of which ultimately perform a `stat/lstat` syscall.

able to cope with Scala code better than others. In particular, one may speculate that the stellar Scala performance of both OpenJDKs is due to the fact that, as of this writing, the developers of Scala rely exclusively on OpenJDK 6 to detect performance regressions in the Scala compiler.⁷ This bias towards OpenJDK 6 is only reinforced by the improvements made by its implementers to OpenJDK 7u.

Regardless of the postulated bias towards OpenJDK, Figures 6.1a to 6.2b prove that Scala code poses particular challenges to modern JVMs, which some of them handle much better than others. Jikes RVM in particular seems to have difficulty producing well-optimized code, which raises the question what factors influence a JVM's performance when running Scala code.

In the following I will explore two hypotheses to shed light onto the observed performance differences in general and onto the dismal Scala performance of Jikes RVM in particular:

Different Cost-Benefit Trade-Offs The just-in-time compilation of Scala code exhibits different costs and benefits than the compilation of Java code. Thus, compiler optimizations are applied less often than they should be.

Method Inlining Scala code negatively effects the just-in-time compiler's inlining heuristic. Thus, failure to inline causes performance degradation.

In Section 6.3 I will explore the former hypothesis, whereas in Section 6.4 I will explore the latter.

6.3 The Effect of Scala Code on Just-in-Time Compilers

In this section as well as in the remainder of this chapter, I will limit myself to comparing both OpenJDKs and Jikes RVM, as these JVMs sit at opposite ends of the performance spectrum. Moreover, these three JVMs are Open Source, which allows for a more in-depth analysis of optimization decisions than is possible for their closed-source counterparts, JRockit and J9. Unfortunately, this choice also means limiting myself to just a subset of benchmarks, as the OpenJDKs and Jikes RVM in particular fail to consistently execute several benchmarks. While Jikes RVM fails several benchmarks outright, mostly because of classes missing from GNU Classpath, the OpenJDKs exhibit more spurious failures.

First, I analyzed to what extent the three JVMs utilize their respective optimizing compilers, since code can only be well-optimized if it is optimized at all. I thus recorded which methods have been optimized and at which optimization level once

⁷ See <http://www.scala-lang.org/node/360>.

the JVM has reached its steady state, i.e. when the coefficient of variation of five consecutive benchmark iterations dropped below 2% (cf. Section 6.1.2). Considering the steady-state performance rather than the JVM's startup performance helps to reduce the inevitable fluctuation in optimization decisions across different invocations [GBE07], of which I performed 15 per benchmark and JVM.

For OpenJDK 6, I report the number of methods compiled with its single compiler, the so-called C2 or server compiler [PVC01]. For OpenJDK 7u, which uses a tiered compilation approach with multiple optimization levels, I report the number of methods compiled at each of the four levels (levels 1, 2, 3 use the C1 compiler [KWM⁺08], level 4 uses the C2 compiler [PVC01]). For Jikes RVM, which follows a tiered, compile-only approach, I report both the number of methods compiled with its “baseline” compiler as well as the number of methods compiled with its optimizing compiler [Wha99] at each of the three available optimization levels (levels O0, O1, O2). As Jikes RVM is a meta-circular VM itself written in Java, I took care to exclude the handful of methods from the runtime itself (package `org.jikesrvm`) that are (re)compiled during benchmark execution. For all three JVMs, I also excluded methods that are only part of the benchmark harness (packages `org.dacapo.harness`, `org.apache.commons.cli`, and `org.scalabench`). Any native methods are excluded as well, even though OpenJDK and Jikes RVM do compile a small method stub for them. If a method is compiled several times at different optimization levels, I attributed it to the last level it was compiled at. This accounts both for the effects of adaptive optimization, where frequently executed methods are re-compiled at higher optimization levels, and the effects of dynamic de-optimization, where an initial, speculative optimization decision proved overly optimistic and is later reverted by the just-in-time compiler [AFG⁺05].

Figures 6.3a to 6.3c depict the number of methods optimized by the three JVMs at their different optimization levels. The three figures clearly show that Jikes RVM subjects significantly less methods to its optimizing just-in-time compiler (levels O0, O1, and O2) than do either OpenJDK 6 or 7u. Overall, however, Jikes RVM compiles many more methods than OpenJDK—namely all of them. But the vast majority of methods is compiled with the Jikes RVM's baseline compiler only, which does not perform any optimizations.

Of course, the number of optimized methods is only one simple indicator of steady-state performance, as it treats all methods alike, regardless of their size. Moreover, all modern optimizing compilers perform method inlining, which means that a single method may be compiled multiple times in different contexts. For

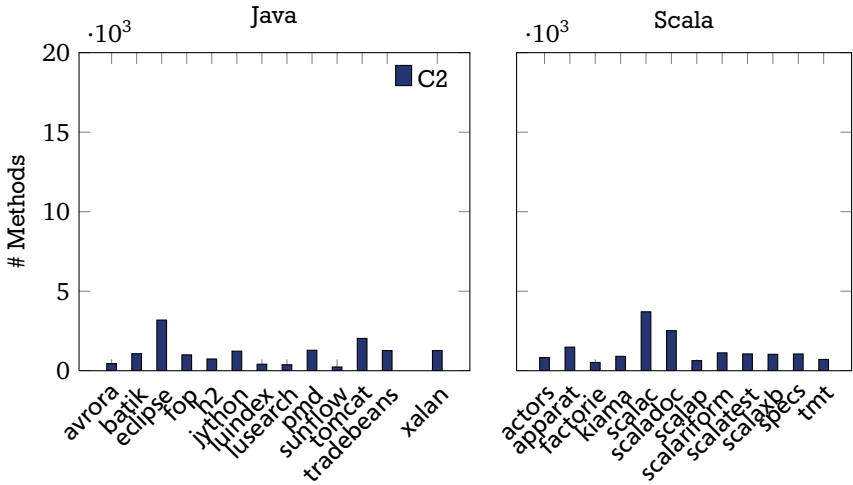


Figure 6.3a: Number of methods compiled by OpenJDK 6 (Arithmetic mean across 15 invocations)

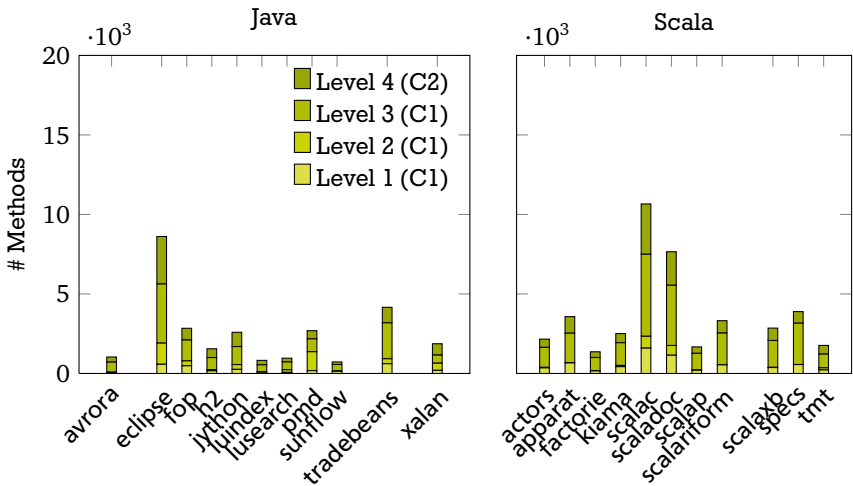


Figure 6.3b: Number of methods compiled by OpenJDK 7u at different optimization levels (Arithmetic mean across 15 invocations)

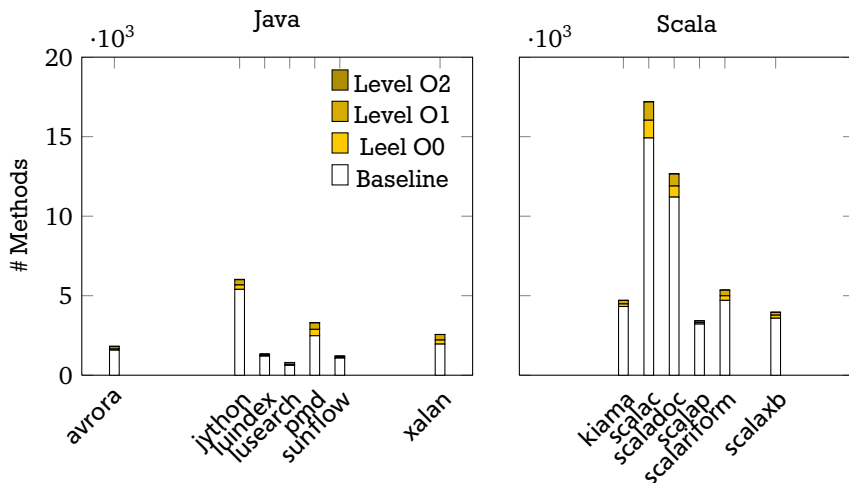


Figure 6.3c: Number of methods compiled by Jikes RVM at different optimization levels (Arithmetic mean across 15 invocations)

the purpose of this section, however, the number of compiled methods suffices as metric,⁸ as this section focuses on in large-scale optimization decisions.

In particular, Figures 6.3a to 6.3c already highlight a crucial difference between Jikes RVM and OpenJDK. Jikes RVM, which follows a compile-only approach, deems extensive optimizations unprofitable for the overwhelming majority of methods; such methods are subjected to the baseline compiler only. This reasoning, as both Figures 6.1a and 6.2a attest, serves Jikes RVM well for the Java benchmarks from the DaCapo benchmark suite; it delivers performance close to that of production VMs. But for the Scala benchmarks, one may hypothesize that Jikes RVM’s adaptive optimization system (AOS) [AFG⁺00] is far off the mark and does not compile enough methods with the RVM’s optimizing compiler.

In the following, I will thus briefly review how OpenJDK and Jikes RVM arrive at their respective optimization decisions. OpenJDK uses a straight-forward threshold-based approach to trigger the compilation and hence optimization of a method. Conceptually, in both OpenJDK 6 and 7u the decision is governed by two thresholds: an invocation threshold (`-XX:CompileThreshold`) and a back-edge threshold (`-XX:BackEdgeThreshold`). When the number of interpreted

⁸ For more detailed metrics, the reader is referred to Section 6.4, which explicitly discusses the effects of method inlining.

method executions exceeds the invocation threshold, that method is compiled. Likewise, when the number of backward branches taken within a loop exceeds the back-edge threshold, the containing method is compiled and installed right away using on-stack replacement (OSR) [FQ03]. OSR thereby ensures that a single, long-running loop cannot deteriorate performance. Unlike its predecessor, OpenJDK 7u uses a tiered compilation approach; thus, a method can successively be re-compiled at four different optimization levels. But the basic approach remains the same: For each of the four levels there exist both invocation and back-edge thresholds to trigger compilation at that level (`-XX:Tier2CompileThreshold`, `-XX:Tier2BackEdgeThreshold`, etc.), along with several other thresholds and triggers for fine-tuning the overhead incurred by maintaining invocation and back-edge counters in optimized code.

Aside from these complications, OpenJDK's approach is rather simple. In contrast, Jikes RVM's approach is much more involved [AFG⁺00, Section 4.3]. For each method, Jikes RVM uses sampling to estimate the time spent executing that method at the current optimization level. Once a method is considered *hot*, i.e. when a lot of time is spent executing that method, the AOS performs a cost-benefit analysis of re-compiling it at the same⁹ or a higher optimization level. It estimates the costs of optimizing the method based on the method's size and the compiler's compilation rate for the optimization level considered. The benefits of optimization are estimated based on the expected speedup and the predicted future execution time of the overall application and hence also of the method in question.¹⁰ Together, compilation rate and expected speedup form the so-called compiler DNA.¹¹ For each of the optimization levels, the factors have been determined in dedicated training runs, in which Jikes RVM compiles all methods at the level in question.

These training runs were performed using a set of Java benchmarks (SPEC JVM98); thus, one may hypothesize that they are inherently biased towards the execution of Java programs. To test this hypothesis, I re-computed the compiler DNA from the steady-state performance exhibited on the benchmarks from both the DaCapo and the Scala benchmark suite. Table 6.3 shows the resulting compiler DNA together with the built-in DNA that was derived by the implementers of Jikes RVM by training on the SPEC JVM98 suite. Unfortunately, several entries in Table 6.3 are missing, as forcing the optimizing compiler to compile a benchmark in its entirety exposes several bugs both in Jikes RVM¹² and in the DaCapo bench-

⁹ The availability of new profiling data may make re-optimizing at the same level attractive.

¹⁰ As the AOS obviously cannot predict the future, it simply assumes that the program will execute for twice the current duration [AFG⁺00].

¹¹ See <http://jikesrvm.org/Compiler+DNA>.

¹² See <http://jira.codehaus.org/browse/RVM-957> and <http://jira.codehaus.org/browse/RVM-958>.

Benchmark	Base	O0		O1		O2	
	Rate	Speedup	Rate	Speedup	Rate	Speedup	Rate
avrora	1328	1.82	59.63	2.11	30.17	2.10	28.76
ython	824	—	—	—	—	—	—
luindex	1534	2.27	50.32	2.62	27.30	2.62	26.11
lusearch	945	2.08	49.44	2.31	25.75	2.33	24.72
pmd	771	1.51	45.51	—	—	—	—
sunflow	1225	5.09	54.79	5.57	28.42	5.57	27.26
xalan	1056	2.04	52.64	2.33	28.48	2.32	27.36
geo. mean	1068	2.26	51.87	2.78	27.99	2.78	26.81
kiama	848	1.99	37.68	2.44	22.53	2.44	21.71
scalac	943	2.71	41.96	—	—	—	—
scaladoc	932	2.00	41.50	—	—	—	—
scalap	851	1.88	41.50	—	—	—	—
scalarifform	882	2.45	38.06	3.45	22.44	3.51	21.62
scalaxb	924	2.62	41.83	3.34	23.25	3.34	22.36
geo. mean	896	2.25	40.38	3.04	22.74	3.05	21.89
Built-in	909	4.03	39.53	5.88	18.48	5.93	17.28

Table 6.3: Compiler DNA for Jikes RVM, i.e., the compilation rates (in KiB/ms) at the various optimization levels and the speedup over baseline-compiled code (Arithmetic mean across 15 invocations)

marks.¹³ Nevertheless, the results allow for some interesting observations: First, the default compiler DNA underestimates the compilation rates achieved by the optimizing compiler but overestimates the resulting speedup. Second, optimizing Scala code results in marginally higher speedups over the baseline than optimizing Java code. Third, for either compiler, the compilation rates are significantly lower for Scala code than they are for Java code.

The question is thus whether the low compilation rates for Scala code can be explained by a bottleneck in the optimizing compiler, e.g., by a particularly expensive optimization which is applied more often to Scala code than to Java code. But as Table 6.4 shows, the lower compilation rates observed cannot easily be attributed to a single cause, i.e. to a single phase of the RVM's optimizing compiler. What is noteworthy, however, is that the two initial lowering transformations from

¹³ See http://sourceforge.net/tracker/?func=detail&atid=861957&aid=3049886&group_id=172498.

	O0		O1		O2	
	Java	Scala	Java	Scala	Java	Scala
Bytecodes → HIR	6.1	6.2	8.4	8.6	8.4	8.8
CFG Transformations	—		1.7	1.7	1.7	1.7
CFG Structural Analysis	1.1	1.2	0.7	0.7	0.7	0.7
Flow-insensitive Opt.	—		1.2	1.3	1.2	1.3
Escape Transformations	—		1.7	1.6	1.7	1.6
Branch Optimizations	—		0.5	0.5	0.5	0.5
Copy Propagation	0.9	1.0	0.2	0.2	0.2	0.2
Constant Propagation	0.3	0.3	0.1	0.2	0.1	0.2
Common Subexpr. Elimination	1.2	0.9	0.7	0.5	0.7	0.5
Field Analysis	0.1	0.2	0.1	0.1	0.1	0.1
HIR → LIR	20.7	22.7	24.9	27.8	24.9	27.8
Copy Propagation	0.6	0.8	0.4	0.5	0.4	0.5
Constant Propagation	0.3	0.4	0.6	0.8	0.6	0.8
Local CSE	1.3	1.1	1.1	0.9	1.1	0.9
Flow-insensitive Opt.	1.8	2.1	1.5	1.6	1.5	1.6
Basic Block Freq. Estimation	1.6	2.0	1.6	2.0	1.6	2.0
Code Reordering	0.1	0.2	1.6	1.9	1.6	2.0
Branch Optimizations	1.1	1.2	1.6	1.6	1.6	1.7
LIR → MIR	38.6	30.3	26.6	20.9	26.6	20.3
Register Mapping	20.6	25.2	20.7	22.4	20.7	22.7
Branch Optimizations	—		0.9	1.0	0.9	1.0
Machine Code Generation	3.4	4.1	3.2	3.2	3.2	3.2

Table 6.4: Percentage of time spent by Jikes RVM in the phases of its optimizing compiler when running Java and Scala benchmarks, respectively (Arithmetic mean across 15 invocations)

Java bytecode to the compiler’s high-level intermediate representation (HIR) and from the high-level to the low-level intermediate representation (LIR) are more expensive for the Scala benchmarks, whereas the final lowering to machine-level intermediate representation (MIR) is less expensive.

The question is, however, whether the aforementioned differences are large enough to have an impact on the resulting steady-state performance. Figures 6.4a and 6.4b answer this question: Tuning the compiler DNA for a specific language can result in a small although not always (statistically) significant speedup. Likewise,

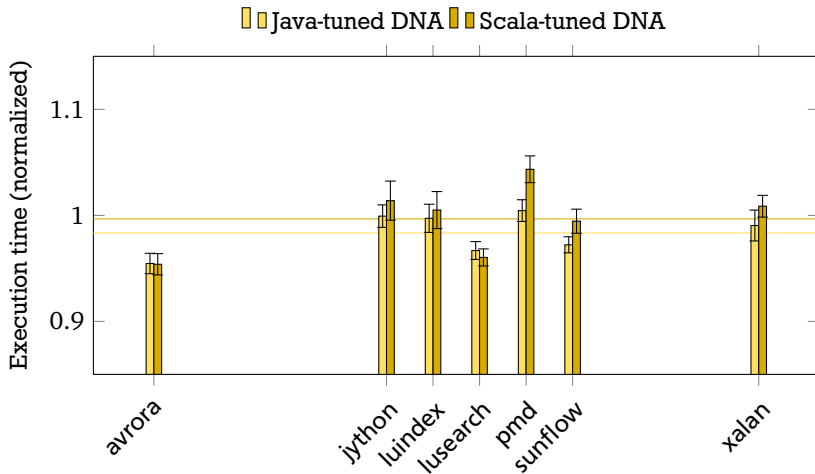


Figure 6.4a: Steady-state execution time of the Scala benchmarks on Jikes RVM with tuned compiler DNA, normalized to the default DNA (Arithmetic mean \pm 95% confidence interval)

tuning for the “wrong” language can result in a slowdown. On the Java benchmarks the DNA tuned for Java is superior to the DNA tuned for Scala, while on the Scala benchmarks it is the other way around. Nevertheless, the observed speedups and slowdowns are not nearly large enough to explain the dismal performance of Scala programs running on Jikes RVM (cf. Section 6.2). In the next section, I will therefore investigate whether Scala code as such negatively affects an optimization routinely performed by high-performance JVMs: method inlining. This hypothesis is supported by the VM-independent analysis in Sections 5.4.2 and 5.4.4; Scala code does indeed exhibit different patterns of call-site polymorphism and argument usage than Java code.

6.4 The Effect of Method Inlining on the Performance of Scala Code

Method inlining [DA99a], i.e. replacing a call site in the caller with the callee’s body, is one of the most effective optimizations available to compilers; all modern, high-performance JVMs perform this optimization. The effectiveness of inlining, however, depends crucially on carefully-tuned heuristics—and these heuristics have

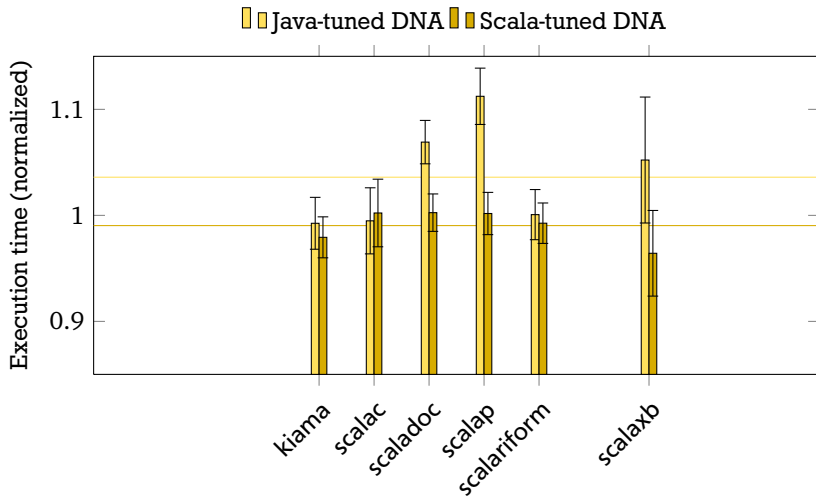


Figure 6.4b: Steady-state execution time of the Java benchmarks on Jikes RVM with tuned compiler DNA, normalized to the default DNA (Arithmetic mean \pm 95% confidence interval)

been tuned with Java programs in mind. In this section, I will investigate how these heuristics fare when confronted with Scala code.

Inline Expansion

I first revisit a metric from Section 6.3, namely the number of methods compiled by the different JVMs at different optimization levels (Figures 6.3a to 6.3c). While the number of compiled methods is a relatively coarse-grained metric, the number of compiled bytecodes is much more fine-grained; in particular, it accounts for methods of different sizes.

Now, due to inlining, the bytecode of a single method may be compiled several times in different contexts. Moreover, some methods may never be compiled as a free-standing method; they are always inlined into some other method by an optimizing compiler. Small “getter” and “setter” methods are the prime example of this. All these effects distort one’s view of a JVM’s optimization decisions when one is restricted to the number of compiled methods as a metric.

Figures 6.5a to 6.5c thus depict the amount of bytecode compiled at the different optimization levels. In particular, the figures distinguish between bytecodes that stem from the (root) method being compiled and bytecodes that stem from other

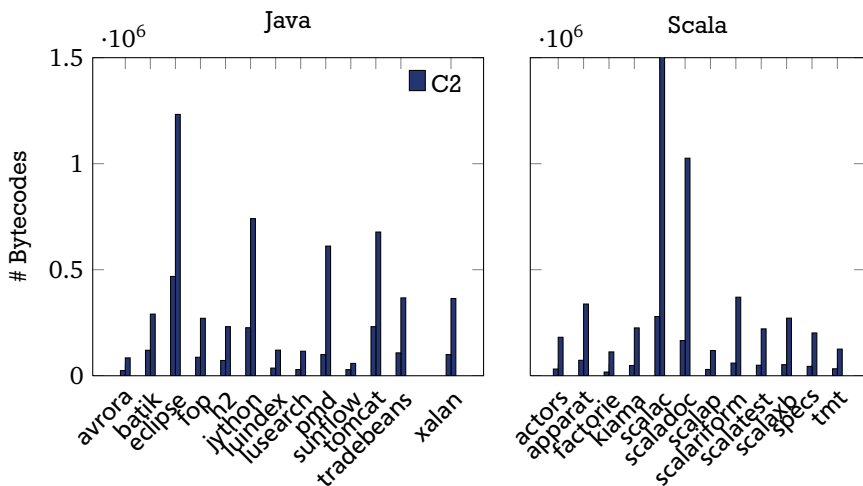


Figure 6.5a: Number of bytecodes compiled by OpenJDK 6 (Arithmetic mean across 15 invocations), excluding (left) or including (right) inlined methods

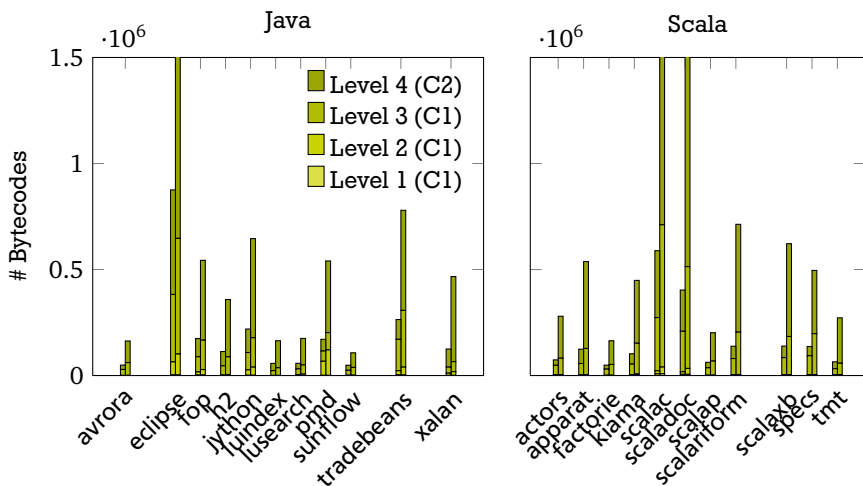


Figure 6.5b: Number of bytecodes compiled by OpenJDK 7u at different optimization levels (Arithmetic mean across 15 invocations), excluding (left) or including (right) inlined methods

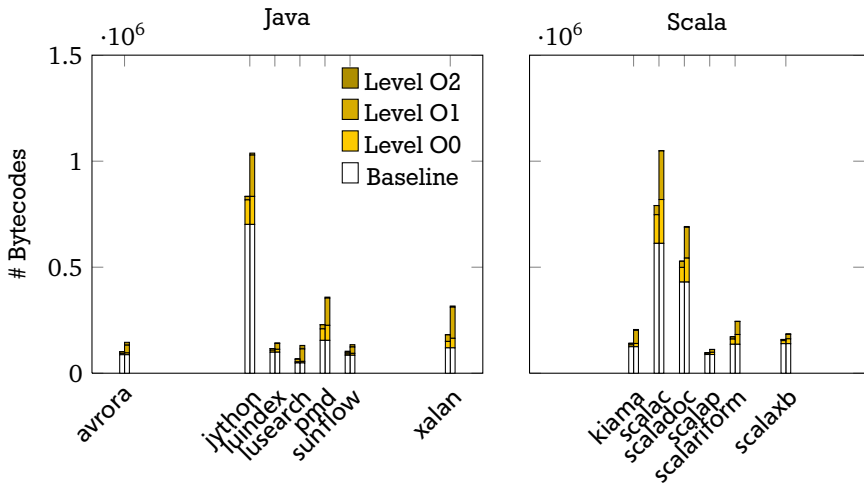


Figure 6.5c: Number of bytecodes compiled by Jikes RVM at different optimization levels (Arithmetic mean across 15 invocations), excluding (left) or including (right) inlined methods

methods that have been inlined into the root method. The left-hand-side bars depict the number of bytecodes of just the methods compiled directly, whereas the right-hand-side bars depict the number of bytecodes of methods compiled both directly and indirectly, i.e. through inlining.

As Figures 6.5a to 6.5c show, even a small number of methods compiled at one of the higher optimization levels can amount to a large number of compiled bytecodes; this is a direct consequence of other methods being inlined into the root method. Conversely, both the C1 compiler of OpenJDK 7u, at its lowest optimization level, and the baseline compiler of Jikes RVM do not inline at all. This optimization is the responsibility of Jikes RVM's optimizing compiler, which does compile a large fraction of bytecodes at one of its three optimization levels (Figure 6.5c): Including bytecodes that stem from inlined methods, it ranges from 30.7% (luindex) to 63.0% (lusearch) and from 21.5% (scalap) to 44.3% (scalariform) for the Java and Scala benchmarks, respectively. On average, 46.1% respectively 34.9% of bytecodes are compiled by the optimizing compiler, even though only a small percentage of methods is optimized by it. Figure 6.6 further illustrates this fact by depicting the number of bytecodes compiled by Jikes RVM over time, i.e. from one benchmark iteration to the next until the steady-state

is reached (coefficient of variation is below 2.0 % for 5 consecutive iterations). As can be seen, the fraction of baseline-compiled bytecode amongst all compiled bytecode steadily decreases over time as more and more methods are re-compiled with the optimizing compiler. Nevertheless, the overall amount of bytecodes compiled increases over time.

Now, inlining contributes significantly to this observed code expansion; even if only a few (root) methods are optimized, copies of other methods may be inlined into the optimized methods and are subsequently optimized as well. Since the indirect benefits of inlining, e.g. the propagation of data-flow information from caller to callee [SJM11], vary depending on the optimizations performed in later compiler phases,¹⁴ the compiler's inlining heuristic also varies in its aggressiveness. At higher optimization levels, inlining is performed more aggressively, as this enables further optimizations later on. Figures 6.7a to 6.7c illustrate this connection: The higher the optimization level, the more aggressive the inlining heuristic, i.e. the larger the amount of inline expansion.

Conversely, as Figures 6.7b and 6.7c clearly show, inlining heuristics are not very aggressive at lower optimization levels. In fact, OpenJDK 7u does not inline at all at its lowest optimization level. Also, Jikes RVM performs little inline expansion at its lowest optimization level, O0; only for levels O1 and up does inlining increase the root method's size significantly.

But the amount of inline expansion does not only correlate with the optimization level chosen but also with the programming language used for the benchmark programs. For example, OpenJDK's C2 compiler, which is used exclusively by OpenJDK 6 and powers the highest optimization level of OpenJDK 7u, inlines much more aggressively when compiling Scala code than when compiling Java code; inlining expands Scala code on average by a factor of 5.1 and 6.9 for OpenJDK 6 and 7u, respectively, whereas it expands Java code by just a factor of 3.2 and 4.3, respectively. While inline expansion at the lower optimization levels of OpenJDK 7u, which are powered by the C1 compiler, is also higher for Scala code than it is for Java code, this difference is less pronounced than at the highest optimization level. That being said, the differences are still more pronounced than those found for Jikes RVM's inlining heuristic; there, inline expansion of Java and Scala code is much more similar, with Scala code facing between 8.1 % (level O1) and 20.1 % (O2) more code expansion than Java code.

I have shown in Chapter 5 that the Scala benchmarks favor inter-procedural over intra-procedural control-flow, i.e. that method calls are more prominent than loops. The fact that Jikes RVM does not react to this with inlining much more aggressively

¹⁴ To completely exploit the indirect benefits, Jikes RVM performs inlining in the very first compiler phase: Bytecodes → HIR (cf. Table 6.4)

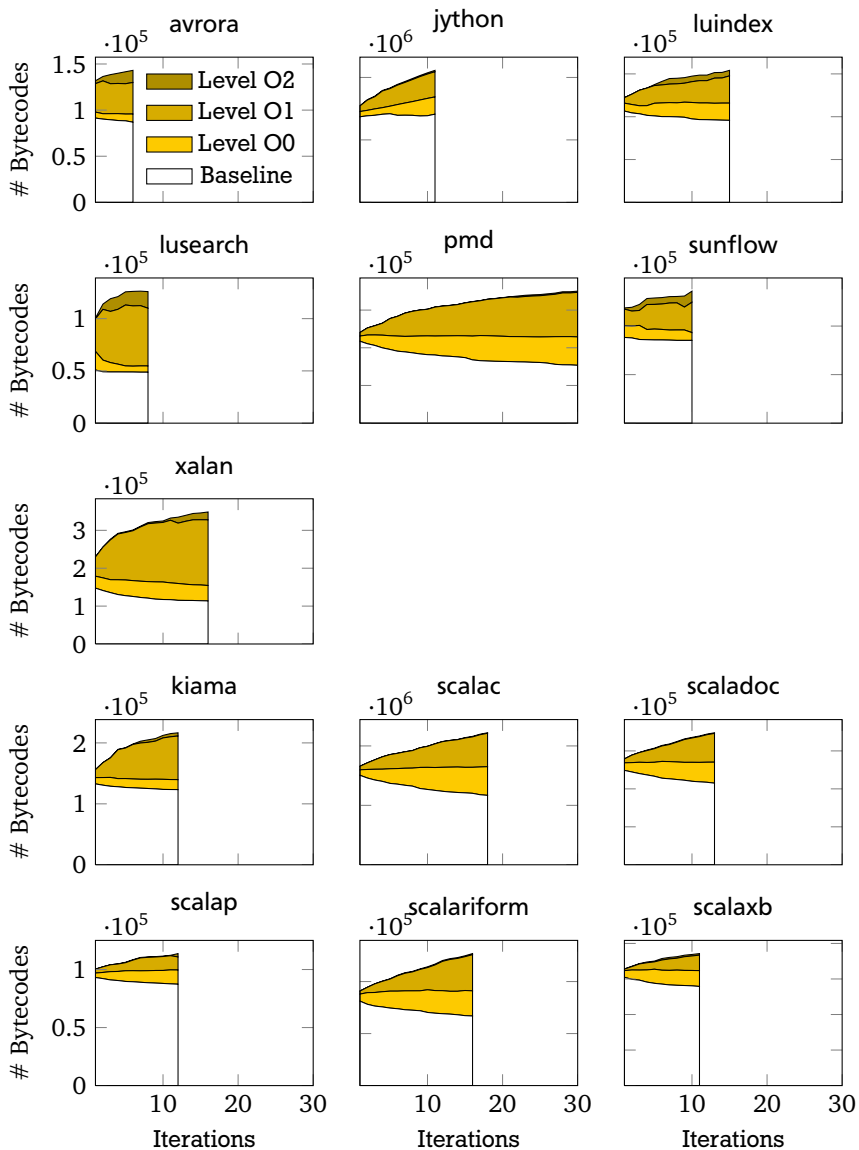


Figure 6.6: The number of bytecodes compiled by Jikes RVM at different optimization levels (best-performing invocation) until the steady-state is reached

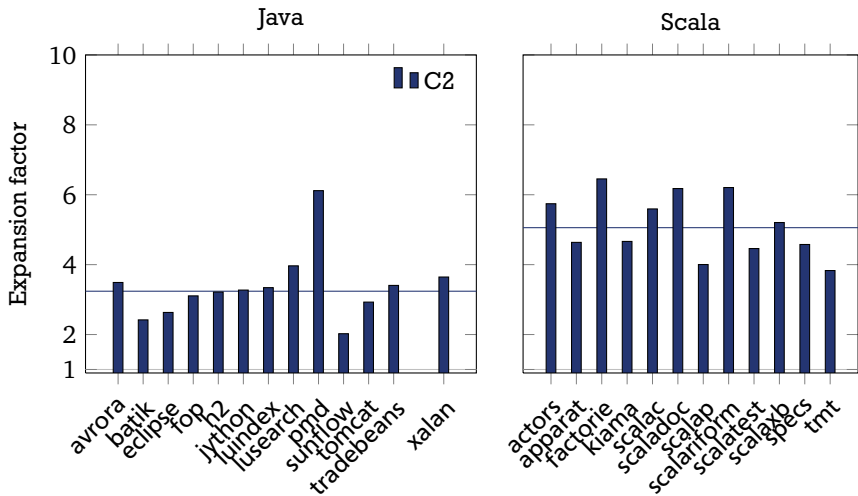


Figure 6.7a: Amount of inline expansion in OpenJDK 6

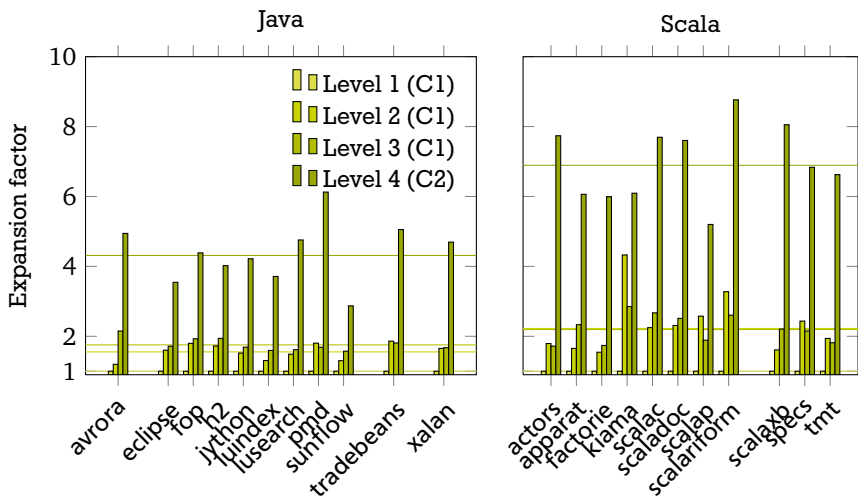


Figure 6.7b: Amount of inline expansion in OpenJDK 7u

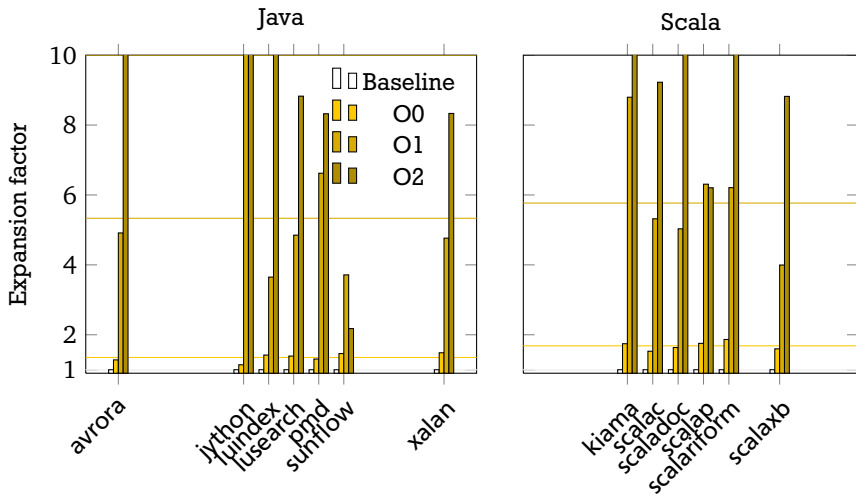


Figure 6.7c: Amount of inline expansion in Jikes RVM

than it already does for the Java benchmarks indicates that the inlining heuristic may be responsible for Jikes RVM’s poor performance on the Scala benchmark suite.

Tuning Jikes RVM’s Inlining Heuristic

I have therefore investigated whether Jikes RVM’s inlining heuristic can be tuned such that the optimizing compiler performs better on Scala code. The aggressiveness of the compiler’s inlining heuristic is kept in check by four parameters: an upper bound on the inlining depth (`-X:opt:inline_max_inline_depth`; defaults to 5), an upper bound on the target’s size (`-X:opt:inline_max_target_size`; 23), an upper bound on the root method’s size (`-X:opt:inline_massive_method_size`; 2048), and a lower bound below which the target method is always inlined (`-X:opt:inline_max_always_inline_target_size`; 11). Of these four parameters, the first two have the largest impact on the VM’s performance [Yan06].

Other parameters help to further fine-tune the cost-benefit analysis performed by the inlining heuristic by reducing the target’s perceived size if inlining would propagate additional information into the target method, e.g. when an argument’s value or exact type is known [SJM11]. While it is possible to automatically tune more than a dozen parameters [CO05], I have chosen to perform a more focused, manual investigation and consider the two most important parameters only, namely `-X:opt:inline_max_inline_depth` and `-X:opt:inline_max_target_size`.

Figures 6.8a and 6.8b show the impact that varying these parameters has on Jikes RVM's steady-state performance on the Java and Scala benchmarks, respectively. During the measurements, the upper bound on the inlining depth assumed one of five values from 5 (the default) to 9. Likewise, the upper bound on the target's size assumed one of nine values from 23 (the default) to 115, i.e. five times that size. This gives rise to 45 combinations of the two parameters' values.

The results confirm the findings of Yang [Yan06] that changing the maximum target method size has a significant effect on performance, whereas changing the maximum inlining depths has a lesser, but still noticeable effect. Moreover, Figure 6.8a shows that inlining more aggressively than the current default hurts steady-state performance on most Java benchmarks; only `ijthon` and `sunflow` benefit from more aggressive inlining. In the case of `ijthon`, however, moderately increasing the upper bound on the target method size to more than twice the parameter's default causes the benchmark to fail. Such failures are indicated by the lines cut short in Figures 6.8a and 6.8b. While even for very aggressive inlining the amount of inlining-induced code bloat is small compared to the overall heap size (cf. Table 6.2), the optimizing compiler nevertheless allocates large amounts of temporary objects which drive the garbage collection workload above a built-in threshold; this causes Jikes RVM to raise an `OutOfMemoryException` and the benchmark to fail.

In general, Figure 6.8a proves that Jikes RVM's inlining heuristic is already well-tuned for Java workloads. Inlining more aggressively does not improve performance for 5 out of 7 benchmarks; instead, the resulting code bloat causes severe performance degradations on benchmarks like `luindex`, `lusearch`, `pmd`, and `xalan`.

In contrast, Figure 6.8b shows that the Scala benchmarks indeed benefit from inlining more aggressively—up to a point. Beyond that point, however, steady-state performance starts to degrade again, as can be observed for the `scala` and `scalaxb` benchmarks. Moreover, inlining too aggressively can cause `OutOfMemoryError`-induced failures (`scalac`, `scaladoc`), as the optimizing compiler allocates a large number of objects; this phenomenon is also observable for the `ijthon` DaCapo benchmark. It is noteworthy that this benchmark from the DaCapo suite, which behaves similar to `scaladoc`, is also not a genuine Java benchmark; the Java bytecode produced from Python source reacts to inlining quite similar to Scala code.

As 6.8b clearly shows, the Scala benchmarks indeed benefit from a more aggressive choice of parameters for Jikes RVM's inlining heuristic than is the case for the Java benchmarks. Consequently, tuning the heuristic with respect to Scala code can indeed yield significant performance improvements, albeit at the detriment of Java performance. That being said, the observed speedups of up to 14.9% are not large enough to explain the observed performance differences between Jikes RVM

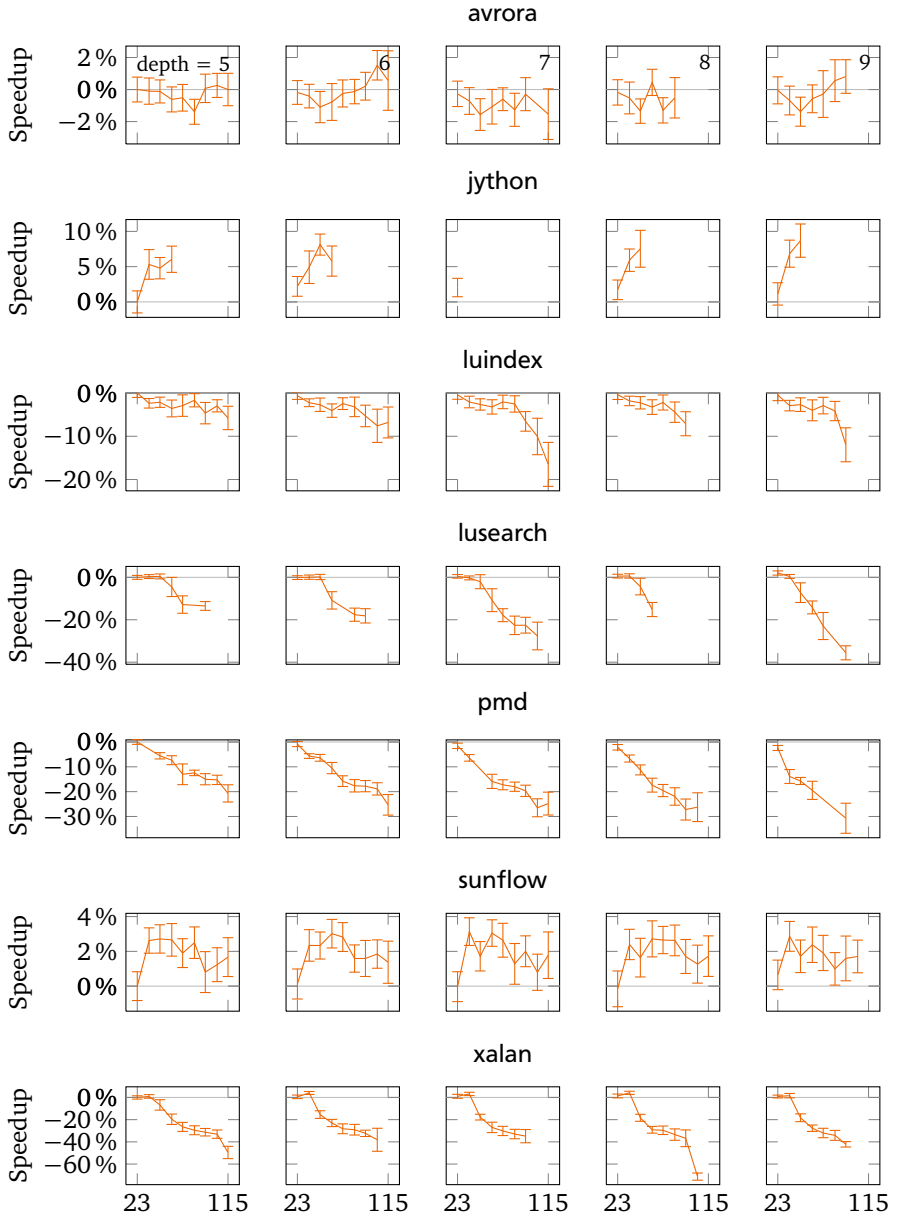


Figure 6.8a: Speedup achieved by tuning the inlining heuristic (Java; cf. Figure 6.8b)

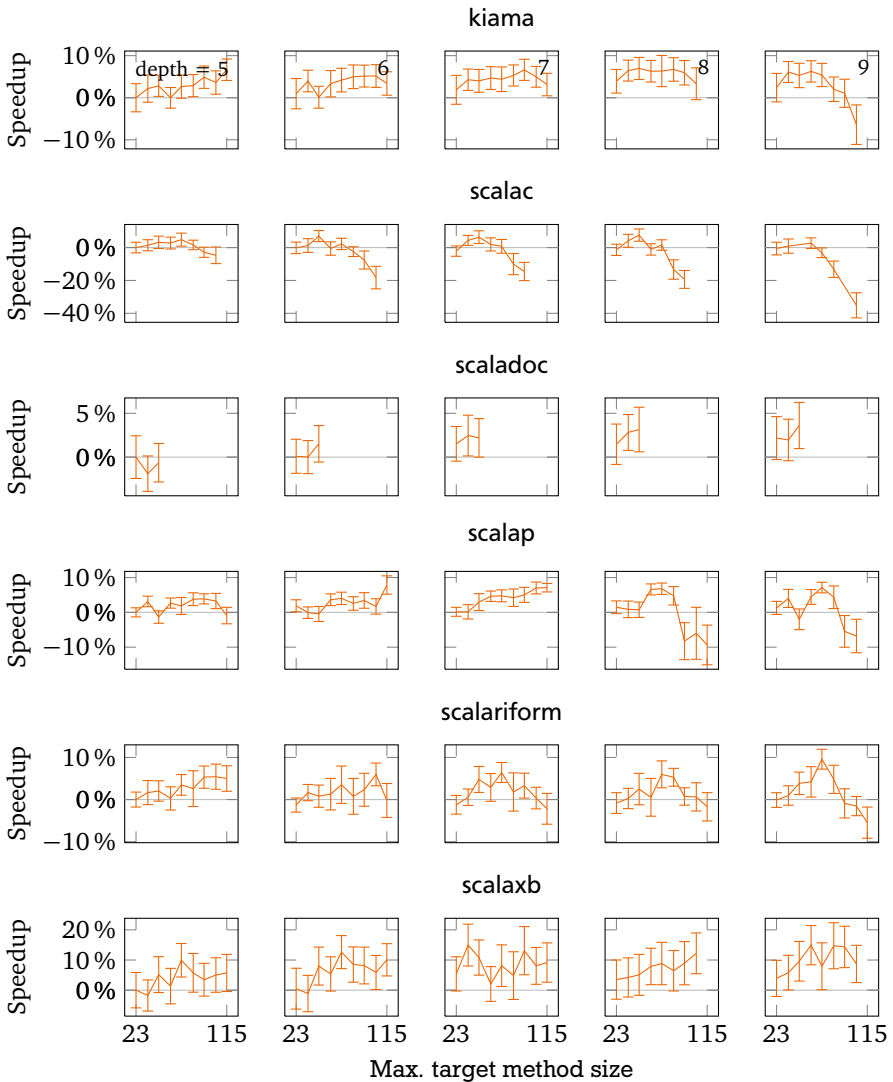


Figure 6.8b: Speedup achieved by tuning the inlining heuristic with respect to maximum inlining depth and maximum target method size over the Scala benchmarks' steady-state performance (Steady-state; arithmetic mean \pm 95% confidence interval)

on the one hand and OpenJDK 6 and 7u on the other hand solely in terms of a less well-tuned inlining heuristic. In their steady-state, the OpenJDKs are about three times as fast as Jikes RVM with its default heuristic.

But failure to inline need not be the result of an out-of-tune heuristic. Instead, it is conceivable that Jikes VM fails to recognize some call sites as inlineable altogether. This is a different interpretation of my hypothesis that “failure to inline causes performance degradation” than the one addressed in this section, which was focused on tuning the Jikes RVM’s inlining heuristic. When Jikes RVM completely misses opportunities for inlining, no amount of tuning will help. Spotting such missed opportunities is extremely hard, as thousands of method calls are compiled per benchmark. I thus consider a modified hypothesis first: Jikes RVM performs no worse than OpenJDK *except* for its failure to inline in certain situations.

Performance without Inlining

I therefore proceed to compare the three JVMs in a setting where inlining cannot impact the JVMs’ performance: For this, I again measure the startup and steady-state performance of both OpenJDKs and Jikes RVM, but this time with inlining disabled (`-XX:-Inline` and `-X:opt:inline=false`, respectively). Note, though, that the precise interpretation of these command-line options varies by JVM. For example, Jikes RVM refuses to inline even VM-internal methods explicitly annotated with `@Inline` when inlining is disabled. I have ascertained, however, that this has minimal impact on performance, as the command-line option does affect neither code in the bootimage, i.e. the RVM itself, nor crucial entrypoint methods into the RVM’s runtime, which are always inlined during the compiler’s lowering phase (HIR → LIR; cf. Table 6.4).

Figures 6.9a and 6.9b depict the change in startup performance due to *enabling* the inlining optimization. As can be seen, at least during startup, inlining does not always result in a speedup. OpenJDK 6 in particular suffers from slowdowns on several benchmarks. This is explained by the fact that I used both OpenJDK VMs in their server configuration (`-server`, the default). In the case of OpenJDK 6, this configuration exclusively uses the C2 compiler [PVC01]. At the price of slower startup, this compiler applies more costly optimizations than its C1 counterpart [KWM⁺08]. In the case of OpenJDK 7, which uses a tiered compilation approach, both compilers are used. This explains why OpenJDK 7u fares better than its predecessor: Its tiered compilation approach makes use of the faster C1 compiler for a large number of methods (cf. Section 6.3).

That being said, Figures 6.9a and 6.9b clearly show that method inlining is already effective during startup of the Scala benchmarks, the only counterpoint being the `scalatest` benchmark. This benchmark is insofar special as it executes various

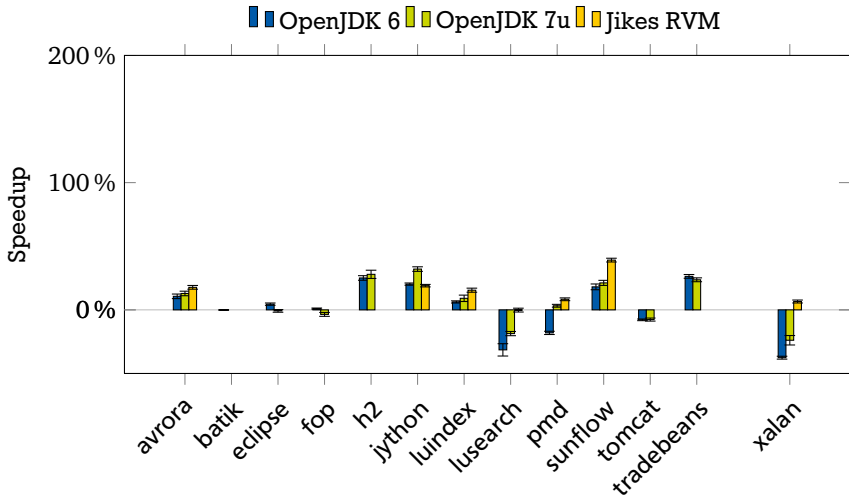


Figure 6.9a: Speedup achieved by inlining over the Java benchmarks' startup performance (Arithmetic mean \pm 95% confidence interval)

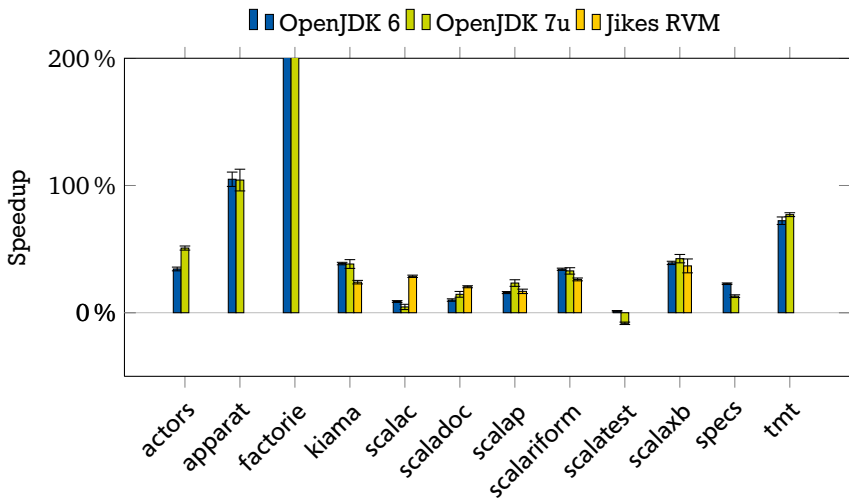


Figure 6.9b: Speedup achieved by inlining over the Scala benchmarks' startup performance (Arithmetic mean \pm 95% confidence interval)

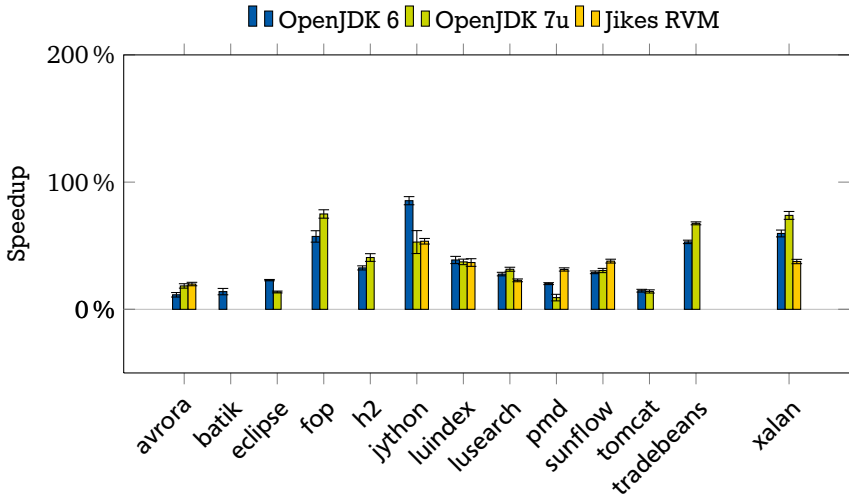


Figure 6.10a: Speedup achieved by inlining over the Java benchmarks’ steady-state performance (Arithmetic mean \pm 95 % confidence interval)

unit tests (cf. Section 3.2.1), all of which are executed only once—by the JVM’s interpreter; its just-in-time compiler is little used. The specs benchmark, however, which also stems from the domain of automated testing does benefit from inlining, even during startup. The scalatest benchmark is therefore clearly an outlier. So, while inlining is a mixed blessing for short-running Java programs, it almost always pays off for short-running Scala programs.

I now focus on the effect of method inlining on long-running programs, i.e. on the benchmark’s steady-state. Figures 6.10a and 6.10b depict the speedup achieved by the inlining optimization in this situation. What immediately catches the eye is that inlining, on all three JVMs, is a much more effective optimization for the Scala benchmarks than it is for the Java benchmarks. In the latter case, the observed speedups are at most 85.4 % (OpenJDK 6/jython), whereas in the former case they can be as high as 518.6 % (OpenJDK 6/factorie). What Figure 6.10b also shows is that enabling inlining on Jikes RVM results in less pronounced speedups than on OpenJDK. While a 116.1 % speedup (Jikes RVM/scalariform) is still impressive, it falls short of the high speedups seen for both OpenJDK 6 and 7u.

The question is whether the fact that both OpenJDKs benefit much more from inlining than Jikes RVM means that without this single optimization the optimizing compilers of all three JVMs would play in the same league, i.e. implement equally

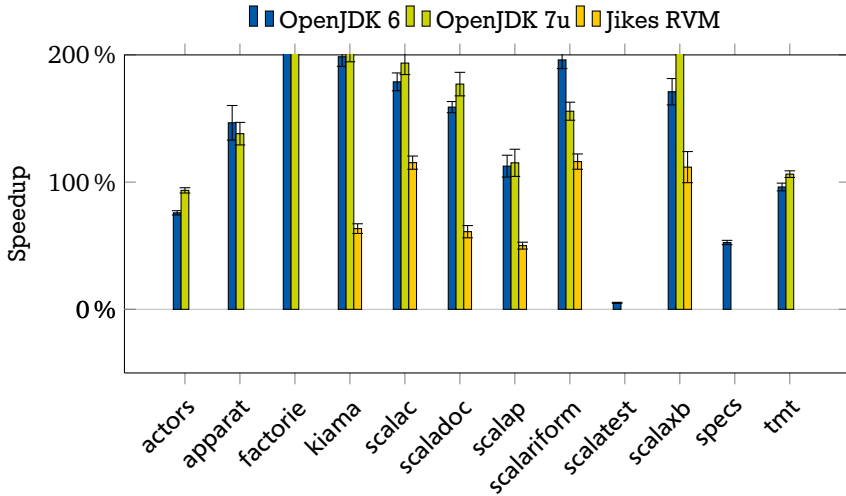


Figure 6.10b: Speedup achieved by inlining over the Scala benchmarks' steady-state performance (Arithmetic mean \pm 95 % confidence interval)

powerful optimizations (besides inlining, of course). Figures 6.11a and 6.11b thus contrast the steady-state execution times achieved with (dark) and without inlining (light shade). Even without inlining, Jikes RVM performs worse on every single benchmark except for `pmd`.¹⁵ Thus, one can conclude that the optimizations (other than method inlining) performed by Jikes RVM's optimizing compiler are significantly less powerful than those performed by OpenJDK's compilers in general and the C2 compiler in particular. It is therefore safe to conclude that this is what hurts Scala performance on Jikes RVM most; other factors like a biased compiler DNA or the less aggressive inlining heuristic contribute their share, but the primary cause is found in the optimizing compiler's later phases.

6.5 Discussion

In this section, I have compared the performance of several high-performance JVMs. Upon finding that one of them, namely the Jikes Research VM, performed

¹⁵ The measurement for `pmd` is questionable, as this benchmark's harness performs very lax output verification, thereby possibly masking a VM failure. See http://sourceforge.net/tracker/?func=detail&atid=861957&aid=3529087&group_id=172498 for further information.

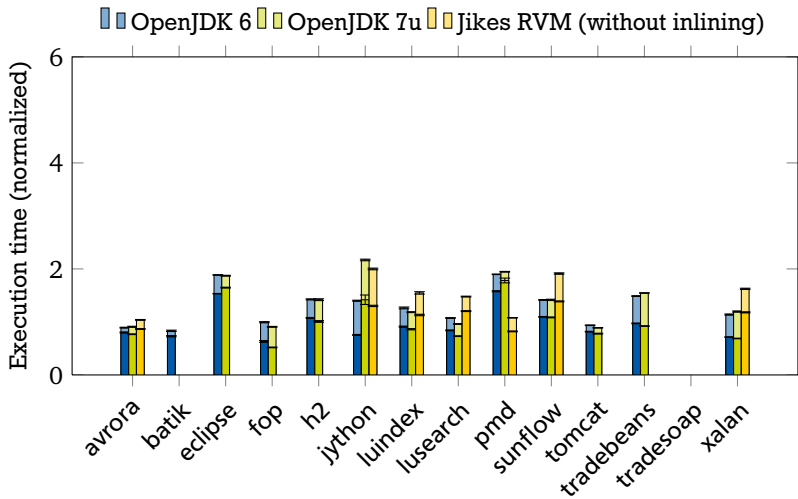


Figure 6.11a: Steady-state execution time of the Java benchmarks with (dark) and without (light shade) inlining, normalized to that of JRockit (Arithmetic mean \pm 95 % confidence interval)

particularly poorly on Scala code despite offering competitive performance on Java code, I investigated the cause and was able to discard two plausible hypotheses as to the primary cause of the observed performance:

Different Cost-Benefit Trade-Offs While JIT-compiling Scala code indeed exhibits costs and benefits different from Java code, the bias towards Java is small.

Method Inlining Scala code indeed benefits much more from method inlining than Java code. But aggressive inlining is only necessary for good performance on Scala code; it is not sufficient.

This investigation allows one to conclude that Jikes RVM’s optimizing compiler as a whole is responsible for that VMs poor performance on the Scala benchmarks; neither the adaptive optimization system nor the inlining heuristic can solely be blamed for the optimizing compiler’s shortcomings. That being said, my investigation has also shown that tuning both the adaptive optimization system and inlining heuristic for a language other than Java can indeed improve performance; tuning the inlining heuristic in particular improved steady-state performance on Scala

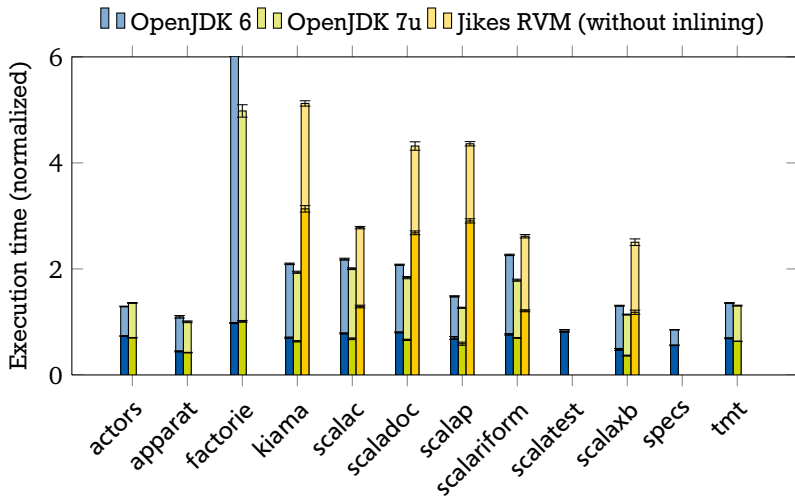


Figure 6.11b: Steady-state execution time of the Scala benchmarks with (dark) and without (light shade) inlining, normalized to that of JRockit (Arithmetic mean \pm 95% confidence interval)

code by up to 14.9%. There is the very real danger, though, that doing so adversely impacts performance on Java code.

Note that I was led to these conclusions only because of the availability of my novel Scala benchmark suite. From measurements obtained using a Java benchmark suite alone one cannot safely generalize to other languages; even if the performance on Java code is competitive, this need not be true for Scala code. This highlights the importance of having a wide variety of benchmarks available, in particular as the popularity of non-Java languages on the Java VM is growing, and stresses the importance of this thesis's contribution: a Scala benchmark suite for the Java Virtual Machine.

7 Related Work

This chapter contains a brief discussion of work that bears relation to mine: First, Section 7.1 discusses efforts to design and establish benchmark suites for the Java Virtual Machine. Next, Section 7.2 describes efforts to workload characterization, focusing on but not limited to languages targeting the JVM. Finally, Section 7.3 reviews research efforts that aim to improve the performance of the Scala language.

Parts of this chapter have been published before:

- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048118
- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2012. doi:10.1145/2258996.2259010

7.1 Benchmark Suites

Shortly after its inception, the first benchmarks for the Java Virtual Machine have sprung up [Bel97, Rou98]. These early benchmarks, however, were simple micro-benchmarks exercising basic operations like arithmetic, field accesses, or method calls in tight loops. Due to their obvious shortcomings, they were largely ignored by the scientific community. But since 1998, various benchmark suites have emerged that subsequently gained wide-spread adoption by JVM researchers. Figure 7.1 shows their lineage.

Among the pioneering benchmark suites for research use are the SPEC JVM98 benchmarks [Cor98] and the Java Grande benchmark suite [BSW⁺00]. Of these, the Java Grande benchmark suite, which is available in both single-threaded and multi-threaded flavours, specifically aimed at so-called “Grande” applications, i.e. large-scale applications with high demands on memory and computational resources. Nevertheless, all these early benchmarks mostly relied on small kernels,

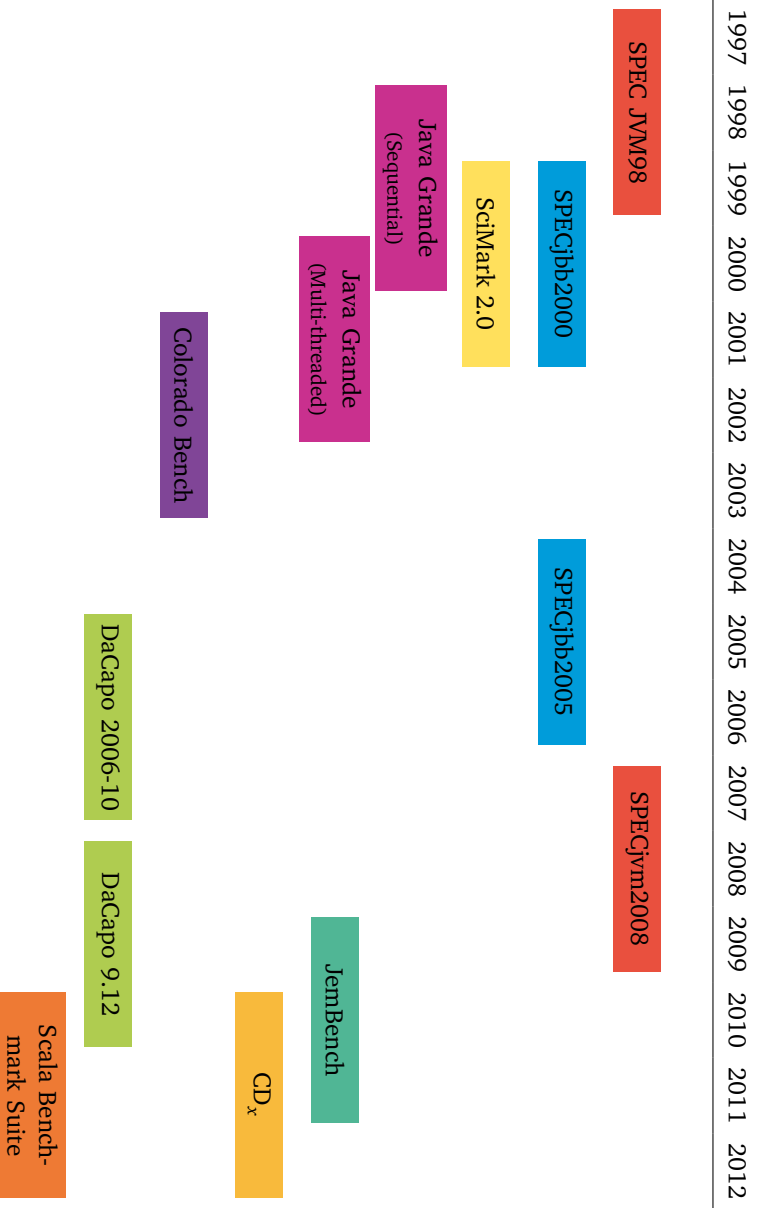


Figure 7.1: The principal benchmark suites being used for Java Virtual Machine research.

which repeatedly performed the same computation. Moreover, they were predominantly numerical in nature, although the SPEC JVM98 suite contained some notable exceptions like the `db` and `javac` benchmarks.

Also part of the SPEC effort are the SPECjbb2000 and SPECjbb2005 “Java business benchmarks.” Both consist of just a single benchmark that emulates a three-tier client/server system. In a research setting, one shortcoming of both benchmarks is that the benchmarks run for a fixed amount of time rather than with a fixed workload. This has led to the development of the so-called `pseudojbb` variant,¹ which keeps the workload fixed by processing a pre-defined number of transactions. There also exists a real-time Java variant of the benchmark called SPECjbbRT.

The SciMark 2.0 benchmark suite [PM00] consists of several numerical kernels that perform, e.g., a fast Fourier transform or Monte Carlo integration. Again, the benchmarks are micro-benchmarks only. But years later they have been integrated with the SPECjvm2008 benchmark suite to jointly form the larger `scimark` benchmark.

In 2002, the Colorado Bench benchmark suite² marked a first attempt to turn a diverse set of four large real-world applications (XSLT processor, persistent XML database, webserver, chatserver) into a benchmark suite. While the benchmark suite has not been as widely used for JVM research as most other suites presented in this section, it nevertheless had its influence on future developments; one of its constituent applications (the XSLT processor `xa1an`) was later turned into one of the original DaCapo benchmarks.

In 2006, Blackburn et al. [BGH⁺06] created the DaCapo benchmark suite to improve upon the state-of-the-art of Java benchmark suites at that time, which the authors criticize heavily in an article submitted to the Communications of the ACM [BMG⁺08]. The eleven benchmarks included in the DaCapo 2006-10 suite cover a broad range of application domains, none of which is primarily concerned with numerical computations. The second release of the DaCapo benchmark suite in December 2009, called version 9.12, rectified this by adding a workload focused on numerical computations, namely the Raytracer `sunflow`. Moreover, the 9.12 release also marks the inclusion of several workloads derived from client/server applications: `tomcat`, `tradebeans`, and `tradesoap`. As their names suggest, the latter two benchmarks share a common core but differ in their use of client/server communication protocols.

In 2008, ten years after the inception of the SPEC JVM98 benchmark suite, the Standard Performance Evaluation Corporation (SPEC) released the SPECjvm2008

¹ See <http://cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005>.

² See http://www-plan.cs.colorado.edu/henkel/projects/colorado_bench/.

suite [Cor08]. By default and like SPECjbb2005, SPECjvm2008 focuses on throughput, i.e. it reports performance in operations per second. But as this is often undesirable for research use [BMG⁺08], it supports a second mode (`--lagom` command-line option) which keeps the workload fixed. It also includes a new startup pseudo-benchmark, which runs all the other constituent benchmarks except derby, which exhibits a high setup cost, for a single operation; this is meant to measure a JVM's startup performance.

Schoeberl et al. [SPU10] developed the JemBench benchmark suite which is specifically aimed at embedded Java applications. It consists of six core benchmarks, bundled together with several micro-benchmarks that cover, e.g., basic integer arithmetic. All benchmarks require only the CLDC API, the lowest denominator among J2ME configurations.

Recently, Kalibera et al. [KHM⁺11] presented the CD_x benchmark for real-time Java. While CD_x is, on the surface, only a single benchmark which exercises an idealized aircraft collision detection algorithm, it is in fact a whole family of benchmarks, each of which targets a specific real-time JVM or API (e.g. RTSJ or Safety Critical Java). Moreover, CD_x comes with several workloads and a noise generator, which can incur additional computational or allocation noise. Interestingly, the CD_x benchmark also comes with an equivalent implementation of the benchmark in the C programming language commonly used for real-time programming.

Other benchmarks which target the Java platform are SPECjms2007, SPECjEnterprise2010, and SPECpower_ssj2008, all of which are published by the Standard Performance Evaluation Corporation. The former two target complex enterprise setups using message-oriented middleware or Java EE application servers whereas the latter focusses on a server's power consumption. All these benchmarks have not yet been picked up by the community of JVM researchers, as other systems (middleware, databases) besides the JVM itself substantially contribute to overall performance.

All of the aforementioned benchmarks not only target the Java platform, but also the Java language, possibly in one of its dialects for embedded or real-time programming. To the best of my knowledge, little effort has been invested so far into benchmark suites for other languages targeting this platform. Dufour et al. [DGH⁺04] made an initial attempt to establish some benchmarks for the AspectJ language. Since then, this initial benchmark suite has been expanded slightly,³ but it has not been picked up by the research community.

Also, the DaCapo benchmark suite, in both its 2006-10 and 9.12 incarnations contains a single benchmark for the Python language, namely `jython`. But while this benchmark, like the entire suite, has been picked up by researchers, little evi-

³ See <http://www.sable.mcgill.ca/abc/benchmarks>.

dence exists that results obtained for this single benchmark can be generalized to all Python workloads on the Java Virtual Machine.

7.2 Workload Characterization

Java

As part of the early Java Grande benchmarking effort, Daly et al. [DHPW01] conducted a VM-independent analysis of its constituent benchmarks. In their work, the authors focus on the benchmarks' instruction mix with both static and dynamic metrics. They consider the 201 instructions of Java bytecode both individually and by manual assignment to one of 22 groups. In Section 5.4.1, I instead apply principal component analysis to automatically extract groupings, which avoids the subjectivity inherent to any manual assignment. Moreover, I use this tool not only to show that there are differences between individual benchmarks, but also between the languages the benchmarks are written in: Java and Scala.

Dufour et al. [DDHV03] analyze a selection of Java workloads using a broad collection of VM-independent metrics. They do so in an effort to assess the metrics' ability to discriminate their workloads; thus, the authors' primary objects of study are the metrics rather than the workloads. This distinguishes the work of Dufour et al. from the one presented in Chapter 5 of this thesis. Nevertheless, Dufour et al. not only describe but also measure various metrics in five different categories: program size and structure, data structures, polymorphism, memory use, and concurrency and synchronization. In particular with respect to concurrency, the metrics I present in Chapter 5 differ significantly; I am interested in the sharing of data among threads rather than in the number of threads running overall. Moreover, I refrain from measuring fickle properties like lock contention and instead measure more stable properties (bias and nesting) that impact the effectiveness of current lock implementations.

The VM-independent evaluation in Chapter 5 of this thesis for the most part ignores native methods. This is for two reasons: First, the Scala compiler emits Java bytecode, so any metric targeting native code rather than bytecode says little about the Scala compiler and its translation strategy. Second, native methods are only used from within the Java runtime library but not from within the Scala runtime library. In the broader context of workload characterization native methods have nevertheless been subjected to some study. Gregg et al. [GPW05] observe the number of native calls using an instrumented version of the Kaffe virtual machine.⁴ Unfortunately, this instrumentation approach renders their profiler inherently non-portable. Moreover, the number of calls provides only a coarse-grained view of a

⁴ See <http://www.kaffe.org/>.

workload's hotspots. While other researchers [HC99, LS03] provide a more detailed breakdown of where CPU time is spent in Java workloads, their respective profilers also rely on non-portable instrumentation approaches.

The aforementioned study by Gregg et al. [GPW05] is interesting, however, as its main motivation is to compare two benchmark suites, namely Java Grande and SPEC JVM98, in order to contrast scientific with regular, object-oriented Java programs. But unlike my study, theirs is not doing a cross-language comparison; both benchmark suites contain only Java workloads. Nevertheless, their method-level metrics are very similar to mine: the calls and call sites using different instructions (**invokevirtual**–**invokeinterface**) and the number of implementations targeted by them (cf. Section 5.4.2).

Blackburn et al. [BGH⁺06] provide a detailed analysis of their DaCapo 2006-10 benchmark suite. Their selection of metrics puts a strong emphasis on the benchmarks' memory behaviour (allocations, accesses), providing a wealth of data to the developers of garbage collectors. In contrast to Blackburn et al., Chapter 5 of this thesis in equal parts considers code-related and memory-related metrics. While the former metrics are directly motivated by my goal to discern the difference between Scala and Java code, the latter were also selected such that they measure behaviour that may be caused by the Scala compiler's translation strategy. The code-related metrics reported by Blackburn et al. are either JVM-independent but static or dynamic but JVM-dependent. But static metrics like the number of loaded classes, their cohesiveness, and the coupling between them are, despite being independent of the used JVM, mostly not independent of the used language; the design metrics of Chidamer and Kemerer in particular [CK94] depend on the source language. This makes these metrics, in their current form, unsuitable to compare two benchmarks written in different source languages. The dynamic metrics reported by Blackburn et al., e.g., the instruction mix (cf. Section 5.4.1) or method hotness (cf. Section 5.4.5), have been measured in a JVM-dependent fashion, which makes it harder than necessary to carry the results over to different JVMs. Like both myself and Hoste and Eeckhout below [HE07], Blackburn et al. use principal component analysis to demonstrate the benchmarks' diversity.

Hoste and Eeckhout [HE07] show that workload characterization is best done independently of a specific, real-world micro-architecture; instead, metrics should be defined with respect to an idealized micro-architecture. This is exactly the approach I take in Chapter 5, with Java bytecode being the natural choice for such an idealized micro-architecture.

Shiv et al. [SCWP09] characterize the SPECjvm2008 benchmark suite both qualitatively and quantitatively. For their quantitative analysis, however, the authors rely primarily on metrics that are both JVM- and architecture-dependent. This is

again in contrast to my analysis in Chapter 5. Shiv et al. also briefly compare the SPECjvm2008 suite, which is their primary object of study, with its predecessor, the SPEC JVM 98 suite, both of which are pure Java benchmark suites.

To motivate their work on thin locks, Bacon et al. [BKMS98] performed a study on synchronization for a large albeit ad-hoc set of Java benchmarks. While my analysis in Section 5.4.14 is similar in nature and scope, it uses and compares two state-of-the-art benchmarks suites for Java and Scala, respectively.

As they did in their earlier work, Bacon et al. [BFG06] motivated their work on header-compression techniques by means of a study on the use of identity hash-codes by a set of Java benchmarks (SPECjvm98, SPECjbb2000); Java programs compute the identity hash-code of just a small fraction of objects (1.3%). The analysis in Section 5.4.15 replicates the measurements of Bacon et al. for both a more recent Java benchmark suite and the novel Scala benchmark suite described in this thesis.

To my knowledge, Dieckmann and Hölzle [DH99] performed the first extensive study on the memory behaviour of Java programs. Similar to the use of Elephant Tracks (cf. Section 5.2), their experimental setup uses traces together with a heap simulator to measure heap composition and the age distribution of objects. In their experiments, Dieckmann and Hölzle found that both arrays and non-reference fields contribute to a large extent to a Java program’s memory consumption. Furthermore, by comparison to other studies the authors found that Java objects are less likely to die at a young age than in ML or Lisp programs. With respect to the analysis presented in Chapter 5, this puts Scala’s memory behaviour firmly in the camp of functional languages like ML or Lisp.

Jones and Ryder [JR08] performed a detailed study of Java object demographics with a focus on lifetime classification. In their study, the authors found that Java objects commonly required lifetime classifications more complex than “short-lived,” “long-lived,” and “immortal,” even though only a few lifetime distribution patterns dominate the observed demographics. Their study, which specifically focuses on object lifetimes, is considerably more detailed than what I present in Section 5.4.8; nevertheless, Sections 5.4.8 and 5.4.9 already provide strong indication that the objects created “under-the-hood” by the Scala compiler to represent, e.g. closures, exhibit very distinct lifetime distribution patterns.

In a very recent observational study, Kalibera et al. [KMJV12] investigated to what extent the workloads from the DaCapo benchmark suite (releases 2006-10 and 9.12) exhibit parallelism. Like I did in Chapter 5, the authors solely rely on VM-independent metrics to ensure that their findings are not restricted to a particular JVM implementation or hardware platform. But unlike my analysis of sharing of objects amongst threads in Section 5.4.13, Kalibera et al. define their metrics

to be time-sensitive. This allows them to detect patterns like a producer/consumer relationship among threads or rapidly changing object ownership. The authors also distinguish between the notions of “shared use” and “shared reachable” and analyze the use of **volatile** fields and concurrent APIs like `java.util.concurrent`. Where my analysis in Chapter 5 aims at giving a broad overview, theirs investigates one issue, namely parallelism, in-depth. Applying their metrics, techniques, and tools to the Scala benchmarks from my benchmark suite might thus be a fruitful direction for future work.

AspectJ

Dufour et al. [DGH⁺04] characterized the execution behaviour of programs written in AspectJ, an aspect-oriented extension to Java. To this end, the authors tagged individual bytecodes as corresponding to the Java and AspectJ parts of the source program, respectively, or as corresponding to low-level infrastructure code inserted by the AspectJ compiler for various purposes. The work by Dufour et al. is similar to mine in that it relies predominately on bytecode-based metrics, but dissimilar in that I compare two benchmark suites, one for Java and one for Scala, whereas their benchmarks *eo ipso* are written in two languages, the base-language Java and the aspect-language AspectJ. That being said, the goals of my work and theirs are similar as well: to shed light on the execution behaviour of a novel language targeting the JVM.

PHP

In recent work, Jibaja et al. [JBHM11] performed a comparison of the memory behaviour of two managed languages, namely Java and PHP. Using the SPECjvm98 and PHP benchmarks suites, they found significant similarities in behaviour and concluded that the implementers of PHP, who chose to implement a very primitive memory manager (reference counting with backup tracing to detect cyclic garbage), should focus on the tried and tested garbage collector designs (generational tracing collectors) found in modern JVMs. There exists new evidence, though, that even reference counting with backup tracing can deliver good performance when carefully tuned [SBF12].

JavaScript

Recently, both Richards et al. [RLBV10] and Ratanaworabhan et al. [RLZ10] set out to study the dynamic behaviour of JavaScript. Despite the difference in the languages studied—Scala in my case and JavaScript in theirs—they follow an approach similar to the one I describe in Chapter 5, using VM-independent metrics like instruction mix, call-site polymorphism, and method hotness. For this thesis,

however, I went one step further in that I developed a state-of-the-art benchmark suite for other researchers to use. Moreover, I also compared the results obtained for the newly developed suite with an already established (Java) benchmark suite. Of the aforementioned two studies, only the one by Ratanaworabhan et al. includes a comparison with other, established JavaScript benchmarks suites.

Scala and Other Languages

Recently, Robert Hundt [Hun11] conducted a small-scale study in which he compared four implementations of a single algorithm (loop recognition) implemented in C++, Java, Go, and Scala, respectively, along several dimensions: “language features, code complexity, compilers and compile time, binary sizes, run-times, and memory footprint.” The author readily admits that his study is an “anecdotal comparison” only. Nevertheless, the study does point out some issues with the Scala compiler’s translation strategy of Scala source- to Java bytecode, in particular with respect to for-comprehensions. Moreover, Hundt found that for both his Java and Scala benchmark, tuning the garbage collector had an disproportionate effect on the benchmarks’ runtime performance.

Totoo et al. [TDL12] conducted a similar study, comparing the support for parallel programming in three languages: Haskell, F#, and Scala. Like Hundt, the authors compared several implementations of a single algorithm (here: Barnes–Hut simulation). With respect to Scala, Totoo et al. found that whether or not the Scala compiler performs tail recursion elimination has a significant impact on their benchmark’s performance. They also note that, while object allocation on the JVM⁵ is very light-weight, one has to take care of not performing overly expensive object initialization; thus, constructors should finish their work as quickly as possible.

7.3 Scala Performance

While one strand of research on the Scala programming language is firmly focussed on issues of language design [OZ05, MPO08], another strand focusses on language implementation in general and on performance issues in particular.

In his thesis, Schinz [Sch05, Chapter 6] outlines the Scala compiler’s general translation strategy. In particular, he discusses several strategies to translate runtime types and measured their impact on Scala’s performance using earlier versions (circa 2005) of two Scala programs part of my benchmark suite: `scalac` and `scalap`. But the goal of Schinz, being one of the developers of the Scala compiler,

⁵ Unfortunately, the authors do not clearly identify the JVM used (“JVM 1.7”); presumably, they used a recent release of Oracle’s HotSpot JVM.

differs from mine: While his work evaluates different translation strategies for runtime types, mine evaluates the impact that the one strategy finally chosen, together with all the other choices made by the Scala compiler’s developers, has on different Java Virtual Machines.

In his thesis, Dragos [Dra10] describes two key optimizations applied by the Scala compiler: inlining [Dra08] and specialization [DO09a]. The former optimization [Dra10, Section 3] is automatically applied by the Scala compiler⁶ and acts as enabling optimization for closure and dead-code elimination. In contrast, the latter optimization [Dra10, Section 4] requires the programmer to annotate the generic methods to be specialized. Both optimizations have the potential to remove the need for boxing (cf. Section 5.4.7). In his thesis, the author evaluates both optimizations separately on two sets of small-scale benchmarks,⁷ both with respect to the increase in code size and the resulting startup and steady-state performance. The observed speedups for the inlining optimization are highly benchmark-dependent, whereas Dragos observed a consistent twofold speedup for specialization except for one notable outlier in a benchmark whose performance was governed, prior to specialization, almost exclusively by boxing; here, specialization resulted in a 35-fold speedup. All these results were, however, obtained using small-scale benchmarks only, where the failure to eliminate one instance closure or boxing operation alone can ruin performance. Despite the limitations of his benchmarking setup, Dragos arrived at the conclusion “that optimizations [performed by the Scala compiler] can remove more than just the boxing cost, and that the JVM does not perform the same level of inlining and cross-method optimizations.” [Dra10]

Dubochet and Odersky [DO09b] compare reflective and generative approaches to compiling structural types on the JVM. As representatives of the former approach, the authors chose two variants of the Scala compiler which translate structural method calls to reflective calls using monomorphic and polymorphic inline caches, respectively. As representative of the latter approach, the authors chose the Whiteoak compiler, which translates structural method calls to calls to a proxy object’s methods. The authors conclude that Whiteoak’s generative approach potentially outperforms reflective approaches, at least once the higher startup cost of on-the-fly proxy generation are amortized. As the reflective approach is easier to implement, however, the Scala compiler currently implements structural types using reflection, assisted by a polymorphic inline cache. This implementation decision

⁶ Provided that the respective compiler option has been used; by default optimizations are disabled (cf. Section 8.1).

⁷ Most benchmarks are based either on numerical kernels (matrix multiplication, fast Fourier transformation) or on the repeated application of higher-order functions (map, fold).

has been reached solely based on micro-benchmarks. But as my own experiments show, structural types are rarely used in larger benchmarks (cf. Section 5.4.6), so they do not (yet) warrant much optimization effort. Nevertheless, from the experiments one can observe the inline cache's effectiveness; most reflective call sites indeed target just a single target method.

Rytz and Odersky [RO10] provide a detailed description of how the Scala language handles named and default arguments, two features introduced with Scala 2.8, the version of Scala that my benchmark suite targets. While the authors describe the implementation of these features [RO10, Section 3], they do not comment on its performance implications, not even using micro-benchmarks. This is unfortunate, as at least default arguments have the potential to affect performance both positively and negatively:⁸ The positive effect ensues if the default arguments are, as is often the case, simple constants, which can be propagated into the called method; the negative effect ensues because each default argument translates into a virtual method invocation of its own. But these further methods are invoked on the same object as is the called method, so that if inlining is possible for the latter, it is also possible for the former.

Rompf et al. [RMO09] describe how the Scala compiler implements delimited continuations by transforming parts of the program to continuation-passing style. The authors compare Scala's performance on two benchmarks (actors, generators) with that of Kilim, another language with continuations which also targets the JVM. Moreover, they compare an implementation of a third benchmark (same-fringe) using Scala's delimited continuations to alternative Scala implementations. But all three performance comparisons differ in intention very much from mine. In Chapter 5, I compare and contrast the execution characteristics of two languages, without necessarily making a statement about the languages' overall performance. In Chapter 6, I do make statements about performance, but rather about the differences between JVMs rather than between languages. This is because statements of the latter kind are hard to justify and even harder to generalize; what is required is the same benchmark program written in multiple languages, e.g. in both idiomatic Scala and idiomatic Kilim. For the Scala benchmark suite and its Java counterpart, no such one-to-one correspondence between benchmarks exists. In fact, it would be extremely time-consuming to produce a dozen Scala duplicates of large, real-world Java benchmarks, but written in idiomatic Scala.

Recently, Pham [Pha12] developed a tool, incidentally also called Scala Benchmarking Suite (SBS), designed to help writing Scala micro-benchmarks.⁹ The re-

⁸ Named arguments are compiled away entirely.

⁹ Using real-world programs for benchmarks is also supported through so-called snippet benchmarks, but no such benchmark suite has yet been assembled.

sulting micro-benchmarks can be used both to assess the JVM's performance in a statistically rigorous fashion [GBE07], but also to compute a limited selection of dynamic metrics: classes loaded, methods called, use of boxed types, memory consumption, number of GC cycles. In this latter respect, SBS is similar to but more limited than the selection of dynamic metrics presented in Chapter 5. This is a direct consequence of SBS's reliance on the Java Debug Interface, which is less flexible than many of the tools presented in Chapter 4. That being said, the focus of Pham is different than mine; while he wants to assist the developers of the Scala compiler in tuning their translation strategy (cf. Section 2.3), my main audience are the developers of JVMs, who I assist in spotting performance problems (cf. Chapter 6).

8 Conclusions and Future Directions

With the growing popularity of modern programming languages such as Scala, Groovy, or Clojure in recent years, the Java Virtual Machine has become a *Joint Virtual Machines*. Alas, the research community has not caught up yet with this development. While micro-benchmark suites abound on the Internet for nearly every conceivable JVM language, few of them satisfy the requirements of JVM researchers with respect to representativeness and rigorous design.

In this thesis I have addressed this shortcoming for the Scala programming language by designing a full-fledged benchmark suite comparable to and compatible with the well-established DaCapo 9.12 benchmark suite. It is freely available to other researchers, complete with source code for the benchmark harnesses and the build toolchain from the project's website (Figure 8.1). This Scala benchmark suite therefore makes up a valuable piece of research infrastructure, which I myself have already used to good effect [Sew10, SMSB11, SMS⁺12].

In Chapter 5, I was able to answer my initial question of “Scala $\stackrel{?}{\equiv}$ Java mod JVM” [Sew10]: When viewed from the JVM's perspective, Scala programs indeed exhibit patterns, e.g. in the code's instruction mix (cf. Section 5.4.1) or the objects' lifetimes (cf. Section 5.4.8), which are different from Java programs. However, with respect to other metrics, the structure, execution characteristics, and memory behaviour of Scala code are remarkably similar. Therefore, depending on the metric and thus the area of JVM implementation of interest, Scala code is similar or dissimilar to Java code.

But, as I have shown in Chapter 6, the dissimilarities are pronounced enough that some modern JVM implementations perform much worse than the competition on Scala benchmarks—despite delivering competitive performance for Java benchmarks.

8.1 Directions for Future Work

Empirical Analysis

On the Java Virtual Machine, it has traditionally been the responsibility of the just-in-time compiler to perform optimizations; while `javac`, the compiler that compiles Java source- to bytecode, offers a `-optimise` command-line option, the option nowadays has become a no-op. For Scala, however, the situation is different; recently, `scalac`, which compiles Scala source- to Java bytecode, has acquired the

Scala Benchmarking Project

Last Published: 2012-09-21 | Version: 1.5M4SHQ1 | [TU Darmstadt](#) | [Informatic](#) | [Software Technology Group](#) | [Scala Benchmarking Project](#) | [Organizational Profile](#) | [Scala Benchmarking Project](#) | [Introduction](#)

[Dacapo Benchmark Suite](#)

Overview

[Introduction](#)
[OpenPGP Keys](#)
[FAQ](#)

Project Documentation

[Project Overview](#)
[About](#)
[Project License](#)
[Making Lists](#)
[Maven Archetypes](#)
[Source Repository](#)
[Project Team](#)
[Project Summary](#)

Modules

[Organizational POM](#)
[Cross-site Configuration](#)
[Core Projects](#)

[Benchmarks](#)
[Dacapo Integration](#)
[DSL Analyser](#)
[Maven Plugins](#)

[Built by](#)
[maven](#)

Introduction

While originally conceived as a target platform of the Java language only, the Java Virtual Machine has since become an attractive target for a multitude of programming languages, one of which is Scala. But although the Scala compiler emits plain Java bytecode, the execution characteristics of Scala programs are not necessarily similar to those of Java programs. In a nutshell, the Scala Benchmarking Project thus attempts to answer the following research question: *Scala = Java (mod JVM)?*

Also, the benchmark suites so often used in JVM research do not yet reflect the growing popularity of non-Java languages on the JVM; all prevalent suites are still firmly Java-focused. The first major contribution of the Scala Benchmarking Project has thus been to complement a popular Java benchmark suite with a large set of benchmarks based on real-world Scala applications, thereby allowing JVM researchers to finally compare and contrast the performance characteristics of Java and Scala programs beyond the level of micro-benchmarks.

Core Projects

At the moment, the Scala Benchmarking Project consists of the following core projects.

Project	Description
Benchmarks	A collection of Dacapo-based Scala Benchmarks
Dacapo Integration	A collection of projects useful when integrating with the Dacapo benchmark suite
DSL Analyser	A collection of dynamic analysers written in DSL
Maven Archetypes	A collection of archetypes for Apache Maven
Maven Plugins	A collection of plugins for Apache Maven that assist in performance characterization

Download

You will find all development snapshots in our Maven repository. In particular, the Scala benchmark suite is available for download.

Please cite the benchmark suite's exact version number whenever using the suite for your research. We also kindly ask you to cite our OOPSLA '11 publication.

Acknowledgments

This project was supported by the [Center for Advanced Security Research Darmstadt](#) and the [Swiss National Science Foundation](#). It was furthermore partly supported by the [National Science Foundation](#).

Figure 8.1: The website of the Scala Benchmark Project

capability to perform a set of optimizations (`-Yinline`, `-Yclosure-elim` options; `@specialize` annotation) on its own [Dra10]. Not only is the implementation of these optimizations rather intricate [DO09a, Dra08], it also seems redundant: method inlining and escape analysis (needed for closure elimination) are stock optimizations performed by modern JVMs.

One fruitful direction for future work is thus to assess the effectiveness of these optimizations when performed by the Scala compiler rather than the JVM. This would require variants of all the Scala benchmarks, built with different compiler options or even different versions of the Scala compiler. Such a study might also benefit from some compiler support for tagging “under-the-hood” objects with the cause of their allocation. Similar tagging support has been used by Dufour et al. in measuring the dynamic behaviour of AspectJ programs [DGH⁺04].

Benchmarks from my Scala benchmark suite might also be useful in assessing the performance impact of language changes like the value classes proposed in SIP-15.¹ This change in particular has the potential to prevent the creation of most implicit wrapper objects for so-called rich primitives (cf. Section 5.4.8), which at least for some benchmarks (*kiama*, *scalac*, and *scaladoc*) contribute significantly to the benchmarks’ allocations. In contrast to method inlining and closure elimination, value classes were not primarily designed as an optimization. Nevertheless, they may have a positive effect on performance—which needs a full-fledged benchmark suite to quantify exactly.

While this thesis (cf. Chapter 6) assessed the impact of Scala code on the just-in-time compilers of modern Java Virtual Machines, additional work is needed to assess its effect on the JVMs’ garbage collectors. Temporary objects like the ones for closures and implicit wrappers, in particular, may be either stack-allocated or optimized away entirely by modern just-in-time compilers. But whether the complex translation from Scala source- to Java bytecode produces something that is feasible for the JVM to analyze is an open question.

The Scala distribution also supports the Microsoft .NET platform,² although as of this writing the primary focus of the Scala compiler’s developers seems to be the JVM backend. For practical reasons, the analyses performed as part of this thesis have been restricted to the JVM platform, though, as the benchmark suite I designed does not yet exist in a .NET variant. But given such a variant, future work could compare and contrast the two platforms in the same fashion as I compared two benchmark suites in Chapter 5. Also, it could be used to evaluate the performance of competing virtual machines, e.g. Microsoft’s implementation of the CLR with the Mono project’s, like it did for Java virtual machines in Chapter 6.

¹ SIP-15 (Value Classes). See <http://docs.scala-lang.org/sips/>.

² See <http://www.scala-lang.org/node/168>.

Optimizations and Tuning

This directly leads to another fruitful direction for future work: the development of optimizations which are, if not language-specific,³ then at least geared towards a specific language which exhibits different characteristics than Java itself. Given the important role that method inlining plays for Scala performance (cf. Section 6.4), making inlining heuristics more aware of some of the source language's usage patterns, e.g. mixin inheritance or the frequent use of first-class functions, seems worthwhile.⁴ A very first step in this direction would be honouring the `@inline` annotation available to Scala programmers; that way, the JVM would guarantee that certain methods are inlined, relieving the Scala compiler of the burden to implement inlining in its backend [Dra10].

Tuning the optimizing compiler in general and its inlining heuristic in particular also seems to offer some potential for performance improvements. For example, even a simple optimization like tuning the Jikes RVM's compiler DNA already resulted in statistically significant speedups for Scala (cf. Section 6.3). Fully automated approaches in particular [CO05, HGE10] promise to make this not only feasible, but to also give rise to further insights into the factors that influence Scala performance on the Java Virtual Machine.

Maintenance of the Scala Benchmark Suite

No benchmark suite stays relevant forever. Like Blackburn et al. have done for the DaCapo benchmark suite [BGH⁺06], I plan to maintain the benchmark suite, incorporate community feedback, and extend the suite to cover further application domains once suitable Scala applications emerge. In the one and a half years after the suite's first public appearance in April 2011, the number of publicly available, large-scale Scala applications has steadily increased, so it seems likely that the suite's next incarnation will be even broader in scope.

I will also maintain and extend the toolchain used to build the Scala benchmark suite (cf. Section 3.3), so that it becomes easier for other researchers to build their own benchmarks, possibly for further languages beyond Java or Scala.

The DaCapo Benchmark Suite, Release 13.x

The author of this thesis has since been invited to become a committer for the next major overhaul of the DaCapo benchmark suite, tentatively called release 13.x. In addition to the usual maintenance tasks required to keep a large benchmark suite up-to-date (updating benchmarks, retiring old ones) I plan to

³ Any optimization on the JVM is ipso facto applicable to all languages which target the JVM.

⁴ While still conducted in a Java environment, preliminary research by the author on making the inlining heuristic aware of function-object-like behavior is encouraging [SJM11].

adapt the lessons learned from the Scala benchmark suite's build toolchain (cf. Section 3.3) to the DaCapo suite. Moreover, I believe that the tools and techniques honed while analyzing the Scala benchmark suite will prove valuable when evaluating new benchmarks for inclusion into release 13.x. Finally, my experience with benchmarks based on testing frameworks (cf. Section 3.2.3) suggests the inclusion of one such benchmark, presumably built around JUnit, to account for the emergence of test-driven development in that suite as well.



Bibliography

- [AAB⁺05] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria Butrico, Antony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn S. McKinley, MarkMergen, J. Eliot B. Moss, Ton Ngo, and Vivek Sarkar. The Jikes virtual machine research project: Building an open-source research community. *IBM Systems Journal*, 44:399–417, 2005. doi : 10.1147/sj.442.0399.
- [ABB⁺12] Danilo Ansaloni, Walter Binder, Christoph Bockisch, Eric Bodden, Kardelen Hatun, Lukáš Marek, Zhengwei Qi, Aibek Sarimbekov, Andreas Sewe, Petr Tůma, and Yudi Zheng. Challenges for refinement and composition of instrumentations: Position paper. In Thomas Gschwind, Flavio De Paoli, Volker Gruhn, and Matthias Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 86–96. Springer Berlin / Heidelberg, 2012. doi : 10.1007/978-3-642-30564-1_6.
- [ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1997. doi : 10.1145/258915.258924.
- [ACFG01] Bowen Alpern, Anthony Cocchi, Stephen Fink, and David 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. doi : 10.1145/504282.504291.
- [ADG⁺99] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. Technical report, Sun Microsystems, Inc., 1999.
- [AFG⁺00] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000. doi : 10.1145/353171.353175.

-
- [AFG⁺05] M. Arnold, S.J. Fink, D. Grove, M. Hind, and P.F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93(2):449–466, 2005. doi:10.1109/JPROC.2004.840305.
- [AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.
- [Ayc03] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003. doi:10.1145/857076.857077.
- [BCF⁺10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon, editors. *XQuery 1.0: An XML Query Language*. World Wide Web Consortium, 2nd edition, 2010.
- [BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/Performance)*, 2004. doi:10.1145/1005686.1005693.
- [BCW⁺10] Michael Bebenita, Mason Chang, Gregor Wagner, Andreas Gal, Christian Wimmer, and Michael Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2010. doi:10.1145/1852761.1852771.
- [Bel97] Doug Bell. Make Java fast: Optimize! *JavaWorld*, 2(4), 1997. URL: <http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html>.
- [BFG06] David Bacon, Stephen Fink, and David Grove. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, 2006. doi:10.1007/3-540-47993-7_5.
- [BGH⁺06] 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, 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 (OOPSLA), 2006. doi : 10.1145/1167473.1167488.

- [BHMV09] Walter Binder, Jarle Hulaas, Philippe Moret, and Alex Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009. doi : 10.1002/spe.890.
- [BKMS98] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1998. doi : 10.1145/277650.277734.
- [Blo08] Joshua Bloch. *Effective Java*. Sun Microsystems, Inc., 2nd edition, 2008.
- [BMG⁺08] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiederemann. Wake up and smell the coffee: evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008. doi : 10.1145/1378704.1378723.
- [BN99] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in Java. In *Proceedings of the Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, 1999. doi : 10.1145/328690.328693.
- [BSS⁺11] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela 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. doi : 10.1145/1985793.1985827.
- [BSSM10] Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. Taming reflection (extended version): Static analysis in the presence of reflection and custom class loaders. Technical Report TUD-CS-2010-0066, CASED, 2010.

-
- [BSW⁺00] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A benchmark suite for high performance Java. *Concurrency: Practice and Experience*, 12(6):375–388, 2000. doi:10.1002/1096-9128(200005)12:6<375::AID-CPE480>3.0.CO;2-M.
- [Chi07] Yuji Chiba. Redundant boxing elimination by a dynamic compiler for Java. In *Proceedings of the 5th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2007. doi:10.1145/1294325.1294355.
- [CK94] S.R. Chidamber and C.F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994. doi:10.1109/32.295895.
- [CMS07] Christian Collberg, Ginger Myles, and Michael Stepp. An empirical study of Java bytecode programs. *Software: Practice and Experience*, 37(6):581–641, 2007. doi:10.1002/spe.v37:6.
- [CO05] John Cavazos and Michael F. P. O’Boyle. Automatic tuning of inlining heuristics. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC)*, 2005. doi:10.1109/SC.2005.14.
- [Cor98] Standard Performance Evaluation Corporation. SPEC JVM98 benchmarks, 1998. URL: <http://www.spec.org/jvm98/>.
- [Cor08] Standard Performance Evaluation Corporation. SPECjvm2008, 2008. URL: <http://www.spec.org/jvm2008/>.
- [DA99a] David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP)*, 1999. doi:10.1007/3-540-48743-3_12.
- [DA99b] David Detlefs and Ole Agesen. Inlining of virtual methods. In *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP)*, 1999. doi:10.1007/3-540-48743-3_12.
- [DDHV03] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003. doi:10.1145/949305.949320.

-
- [DFD10] Vinicius H. S. Durelli, Katia R. Felizardo, and Marcio E. Delamaro. Systematic mapping study on high-level language virtual machines. In *Proceedings of the 4th Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2010. doi:10.1145/1941054.1941058.
- [DGH⁺04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004. doi:10.1145/1028976.1028990.
- [DH99] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP)*, 1999. doi:10.1007/3-540-48743-3_5.
- [DHPW01] Charles Daly, Jane Horgan, James Power, and John 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. doi:10.1145/376656.376826.
- [DO09a] Iulian Dragos and Martin 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. doi:10.1145/1565824.1565830.
- [DO09b] Gilles Dubochet and Martin 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. doi:10.1145/1565824.1565829.
- [Doe03] Osvaldo Doederlein. The tale of Java performance. *Journal of Object Technology (JOT)*, 2(5):17–40, 2003.
- [Dra08] Iulian Dragos. Optimizing higher-order functions in Scala. In *Proceedings of the 3rd Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, 2008.

-
- [Dra10] Iulian Dragos. *Compiling Scala for Performance*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2010. doi:10.5075/epfl-thesis-4820.
- [DRS08] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings of the 16th Symposium on Foundations of Software Engineering (FSE)*, 2008. doi:10.1145/1453101.1453111.
- [ECM10] ECMA. *Common Language Infrastructure (CLI): Partitions I to VI*. ECMA International, 5th edition, 2010.
- [EGDB03] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003. doi:10.1145/949305.949321.
- [ES11] Michael Eichberg and Andreas Sewe. Encoding the Java Virtual Machine's instruction set. *Electronic Notes in Theoretical Computer Science*, 264(4):35–50, 2011. doi:10.1016/j.entcs.2011.02.004.
- [FQ03] S.J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Proceedings of the Symposium on Code Generation and Optimization (CGO)*, 2003. doi:10.1109/CGO.2003.1191549.
- [GBE07] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2007. doi:10.1145/1297027.1297033.
- [GJS⁺11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java™ Language Specification*. Oracle America, Inc., Java 7 SE edition, 2011.
- [GPB⁺06] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Longman Publishing Co., Inc., 2006.

-
- [GPW05] David Gregg, James Power, and John Waldron. A method-level comparison of the Java Grande and SPEC JVM98 benchmark suites. *Concurrency and Computation: Practice and Experience*, 17(7-8):757–773, 2005. doi:10.1002/cpe.v17:7/8.
- [HBM⁺06] Matthew Hertz, Stephen M. Blackburn, J. Eliot B. Moss, Kathryn S. McKinley, and Darko Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(3):476–516, 2006. doi:10.1145/1133651.1133654.
- [HC99] Nathan M. Hanish and William E. Cohen. Hardware support for profiling Java programs. In *Proceedings of the Workshop on Hardware Support for Objects and Microarchitectures for Java (WHSO)*, 1999.
- [HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP)*, 1991. doi:10.1007/BFb0057013.
- [HE07] Kenneth Hoste and Lieven Eeckhout. Microarchitecture-independent workload characterization. *IEEE Micro*, 27(3):63–72, 2007. doi:10.1109/MM.2007.56.
- [HGE10] Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated just-in-time compiler tuning. In *Proceedings of the 8th Symposium on Code Generation and Optimization (CGO)*, 2010. doi:10.1145/1772954.1772965.
- [HHR95] Richard E. Hank, Wen-Mei W. Hwu, and B. Ramakrishna Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th Symposium on Microarchitecture (MICRO)*, 1995.
- [HP09] Christian Haack and Erik Poll. Type-based object immutability with flexible initialization. In *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP)*, 2009. doi:10.1007/978-3-642-03013-0_24.
- [Hun11] Robert Hundt. Loop recognition in C++/Java/Go/Scala. In *Proceedings of the 2nd Scala Workshop (ScalaDays)*, 2011.

-
- [JBHM11] Ivan Jibaja, Stephen M. Blackburn, Mohammad Haghghat, and Kathryn 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. doi:10.1145/1988915.1988930.
- [JR08] Richard E. Jones and Chris Ryder. A study of Java object demographics. In *Proceedings of the 7th International Symposium on Memory Management (ISMM)*, 2008. doi:10.1145/1375634.1375652.
- [KHM⁺11] Tomas Kalibera, Jeff Hagelberg, Petr Maj, Filip Pizlo, Ben Titzer, and Jan Vitek. A family of real-time Java benchmarks. *Concurrency and Computation: Practice and Experience*, 23(14):1679–1700, 2011. doi:10.1002/cpe.1677.
- [KMJV12] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek. A black-box approach to understanding concurrency in DaCapo. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012. (to appear).
- [KSA09] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2009. doi:10.1145/1596655.1596658.
- [KWM⁺08] Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ client compiler for Java 6. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(1):7:1–7:32, 2008. doi:10.1145/1369396.1370017.
- [LS03] Ghulam Lashari and Suresh Srinivas. Characterizing Java™ application performance. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS)*, 2003. doi:10.1109/IPDPS.2003.1213265.
- [LYBB11] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *Java Virtual Machine Specification*. Oracle America, Inc., Java 7 SE edition, 2011.
- [MDHS10] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Evaluating the accuracy of Java profilers. In *Proceedings*

of the Conference on Programming Language Design and Implementation (PLDI), 2010. doi : 10.1145/1806596.1806618.

- [MJK12] Matthew Mole, Richard Jones, and Tomas Kalibera. A study of sharing definitions in thread-local heaps (position paper). In *Proceedings of the 7th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS)*, 2012.
- [MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008. doi : 10.1145/1449764.1449798.
- [MSS09] Andrew McCallum, Karl Schultz, and Sameer Singh. FACTORIE: Probabilistic programming via imperatively defined factor graphs. *Advances on Neural Information Processing Systems*, 2009.
- [MVZ⁺12] Lukas Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Conference on Aspect-Oriented Software Development (AOSD)*, 2012.
- [OMK⁺10] Kazunori Ogata, Dai Mikurube, Kiyokuni Kawachiya, Scott Trent, and Tamiya 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. doi : 10.1145/1869459.1869477.
- [OSV10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Inc., 2nd edition, 2010.
- [OZ05] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005. doi : 10.1145/1094811.1094815.
- [PBKM00] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2000.

-
- [Pea01] K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(6):559–572, 1901.
- [PFH11] Filip Pizlo, Daniel Frampton, and Antony L. Hosking. Fine-grained adaptive biased locking. In *Proceedings of the 9th Conference on Principles and Practice of Programming in Java (PPPJ)*, 2011. doi:10.1145/2093157.2093184.
- [Pha12] Ngoc Duy Pham. *Scala Benchmarking Suite: Scala performance regression pinpointing*. Bachelor thesis, Vietnam National University, 2012.
- [PM00] Roldan Pozo and Bruce Miller. SciMark 2.0 benchmarks, 2000. URL: <http://math.nist.gov/scimark2/>.
- [PS05] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience*, 17(5–6):639–662, 2005. doi:10.1002/cpe.853.
- [PVC01] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot™ server compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01)*, 2001.
- [RBC⁺11] John Rose, Ola Bini, William R. Cook, Rémi Forax, Samuele Pedroni, and Jochen Theodorou. *JSR-292: Supporting Dynamically Typed Languages on the Java Platform*, 2011. URL: <http://jcp.org/en/jsr/detail?id=292>.
- [RD06] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006. doi:10.1145/1167473.1167496.
- [RGM11] Nathan P Ricci, Samuel Z. Guyer, and J. Eliot B. Moss. Elephant Tracks: generating program traces with object death records. In *Proceedings of the 9th Conference on Principles and Practice of Programming in Java (PPPJ)*, 2011. doi:10.1145/2093157.2093178.
- [RLBV10] Gregor Richards, Sylvain Lebesne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2010. doi:10.1145/1806596.1806598.

-
- [RLZ10] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. JS-Meter: Comparing the behavior of JavaScript benchmarks with real Web applications. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2010.
- [RMO09] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *Proceedings of the 14th International Conference on Functional Programming (ICFP)*, 2009. doi:10.1145/1596550.1596596.
- [RO10] Lukas Rytz and Martin Odersky. Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the scala language. In *Proceedings of the Symposium on Applied Computing (SAC)*, 2010. doi:10.1145/1774088.1774529.
- [Ros09] John R. Rose. Bytecodes meet combinators: invokedynamic on the JVM. In *Proceedings of the 3rd Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2009. doi:10.1145/1711506.1711508.
- [Rou98] Mark Roulo. Accelerate your Java apps! *JavaWorld*, 3(9), 1998. URL: <http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html>.
- [RRC⁺09] Daniel Ramage, Evan Rosen, Jason Chuang, Christopher D. Manning, and Daniel A. McFarland. Topic modeling for the social sciences. In *Proceedings of the NIPS Workshop on Applications for Topic Models: Text and Beyond*, 2009.
- [RZW08] Ian Rogers, Jisheng Zhao, and Ian 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.
- [SAB08] Ajeet Shankar, Matthew Arnold, and Rastislav Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the 23rd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2008. doi:10.1145/1449955.1449775.
- [SBF12] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. Down for the count? Getting reference counting back in the ring. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2012. doi:10.1145/2258996.2259008.

-
- [Sch05] Michel Schinz. *Compiling Scala for the Java Virtual Machine*. PhD thesis, École Polytechnique Fédérale de Lausanne, 2005. doi:10.5075/epfl-thesis-3302.
- [SCWP09] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. SPECjvm2008 performance characterization. In David Kaeli and Kai Sachs, editors, *Computer Performance Evaluation and Benchmarking*, volume 5419 of *Lecture Notes in Computer Science*, pages 17–35. Springer Berlin / Heidelberg, 2009. doi:10.1007/978-3-540-93799-9_2.
- [Sew10] Andreas Sewe. Scala $\stackrel{?}{\cong}$ Java mod JVM. In *Proceedings of the Work-in-Progress Session at the 8th Conference on the Principles and Practice of Programming in Java (PPPJ)*, volume 692 of *CEUR Workshop Proceedings*, 2010.
- [Sin03] Jeremy Singer. JVM versus CLR: a comparative study. In *Proceedings of the 2nd Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2003.
- [SJM11] Andreas Sewe, Jannik Jochem, and Mira Mezini. Next in line, please! exploiting the indirect benefits of inlining by accurately predicting further inlining. In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2011. doi:10.1145/2095050.2095102.
- [Slo08] A. M. Sloane. Experiences with domain-specific language embedding in Scala. In *Proceedings of the 2nd Workshop on Domain-Specific Program Development (DSPD)*, 2008.
- [SMB⁺11] Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and platform-independent calling context profiling for the Java Virtual Machine. *Electronic Notes in Theoretical Computer Science*, 279(1):61–74, 2011. doi:10.1016/j.entcs.2011.11.006.
- [SMS⁺12] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs. In *Proceedings of the International Symposium on Memory Management (ISMM)*, 2012. doi:10.1145/2258996.2259010.

-
- [SMSB11] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java Virtual Machine. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048118.
- [SPU10] Martin Schoeberl, Thomas B. Preusser, and Sascha Uhrig. The embedded Java benchmark suite JemBench. In *Proceedings of the 8th Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, 2010. doi:10.1145/1850771.1850789.
- [SSB⁺11] Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, Martin Schoeberl, and Mira Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java Virtual Machine. In *Proceedings of the 9th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011. doi:10.1145/2093157.2093160.
- [SSB⁺12] Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. JP2: Call-site aware calling context profiling for the Java Virtual Machine. *Science of Computer Programming*, 2012. doi:10.1016/j.scico.2011.11.003.
- [SYN03] Toshio Sukanuma, Toshiaki Yasue, and Toshio Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2003. doi:10.1145/781131.781166.
- [TDL12] Prabhat Tootoo, Pantazis Deligiannis, and Hans-Wolfgang Loidl. Haskell vs. F# vs. Scala: A high-level language features and parallelism support comparison. In *Proceedings of the Workshop on Functional High-Performance Computing (FHPC)*, 2012.
- [TJZS10] Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the 25th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2010. doi:10.1145/1869459.1869471.
- [TR10] Christian Thalinger and John Rose. Optimizing invokedynamic. In *Proceedings of the 8th Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2010. doi:10.1145/1852761.1852763.

-
- [Wha99] John Whaley. Dynamic optimization through the use of automatic runtime specialization. Master's thesis, Massachusetts Institute of Technology, 1999.
- [Yan06] Jing Yang. Statistical analysis of inlining heuristics in Jikes RVM. Report, University of Virginia, 2006. URL: <http://www.cs.virginia.edu/~jy8y/publications/stat51306.pdf>.
- [YBF⁺11] Xi Yang, Stephen M. Blackburn, Daniel Frampton, Jennifer B. Sartor, and Kathryn S. McKinley. Why nothing matters: the impact of zeroing. In *Proceedings of the 26th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011. doi:10.1145/2048066.2048092.
- [ZAM⁺12] Yudi Zheng, Danilo Ansaloni, Lukas Marek, Andreas Sewe, Walter Binder, Alex Villazón, Petr Tuma, Zhengwei Qi, and Mira Mezini. Turbo DiSL: Partial evaluation for high-level bytecode instrumentation. In Carlo Furia and Sebastian Nanz, editors, *Objects, Models, Components, Patterns*, volume 7305 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin / Heidelberg, 2012. doi:10.1007/978-3-642-30561-0_24.