Workload Characterization of JVM Languages

Doctoral Dissertation submitted to the
Faculty of Informatics of the *Università della Svizzera Italiana*
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Aibek Sarimbekov

under the supervision of
Prof. Walter Binder

May 2014

Dissertation Committee

Prof. Matthias Hauswirth	Università della Svizzera Italiana, Switzerland
Prof. Cesare Pautasso	Università della Svizzera Italiana, Switzerland
Prof. Andreas Krall	Technische Universität Wien, Vienna, Austria
Prof. Petr Tůma	Charles University, Prague, Czech Republic

Dissertation accepted on 9 May 2014

Prof. Walter Binder
Research Advisor
Università della Svizzera Italiana, Switzerland

Prof. Stefan Wolf and Prof. Igor Pivkin
PhD Program Director

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Aibek Sarimbekov
Lugano, 9 May 2014

To my family

Everything should be made as simple
as possible, but not simpler.

Albert Einstein

Abstract

Being developed with a single language in mind, namely Java, the Java Virtual Machine (JVM) nowadays is targeted by numerous programming languages. Automatic memory management, Just-In-Time (JIT) compilation, and adaptive optimizations provided by the JVM make it an attractive target for different language implementations. Even though being targeted by so many languages, the JVM has been tuned with respect to characteristics of Java programs only – different heuristics for the garbage collector or compiler optimizations are focused more on Java programs. In this dissertation, we aim at contributing to the understanding of the workloads imposed on the JVM by both dynamically-typed and statically-typed JVM languages. We introduce a new set of dynamic metrics and an easy-to-use toolchain for collecting the latter. We apply our toolchain to applications written in six JVM languages – Java, Scala, Clojure, Jython, JRuby, and JavaScript. We identify differences and commonalities between the examined languages and discuss their implications. Moreover, we have a close look at one of the most efficient compiler optimizations – method inlining. We present the decision tree of the HotSpot JVM's JIT compiler and analyze how well the JVM performs in inlining the workloads written in different JVM languages.

Acknowledgements

First of all I would like to thank my research adviser Professor Walter Binder who guided me for the past four years through this interesting journey called Ph.D. I am also grateful for my committee members including Professor Andreas Krall, Professor Petr Tůma, Professor Cesare Pautasso, and Professor Matthias Hauswirth, your valuable comments helped me a lot to strengthen this dissertation.

I would like to thank all the past and present members of the Dynamic Analysis Group with whom I had a pleasure to work with during my Ph.D at University of Lugano: Philippe, Danilo, Akira, Alex, Achille, Stephen, Yudi, Lukáš, Lubomír, Andrej, Sebastiano, Andrea – you guys are awesome!

I am thankful to my collaborators and co-authors from different parts of the world: Danilo Ansaloni, Walter Binder, Christoph Bockisch, Eric Bodden, Lubomír Bulej, Samuel Z. Guyer, Kardelen Hatun, Stephen Kell, Lukáš Marek, Mira Mezini, Philippe Moret, Zhengwei Qi, Nathan P. Ricci, Martin Schoeberl, Andreas Sewe, Lukas Stadler, Petr Tůma, Alex Villazón, Yudi Zheng. In particular I am grateful to Andreas Sewe, whose energy and motivation still impresses me. Thank you Andreas for all your help and support throughout my Ph.D.

Elisa, Janine, Danijela and Diana from the decanato office deserve special thanks for their support and help with numerous bookings and registration stuff. I cannot find the words to thank you.

Maksym and Konstantin I will miss a lot our discussions during the lunches and coffee breaks, however, I am more than sure that our friendship will last for good.

And finally I would like to thank my family that keeps supporting me in my every venture. This work is dedicated to every member of my family.

This work was partially funded by Swiss National Science Foundation (project CRSII2_136225).

Contents

Contents **xi**

1 Introduction **1**
1.1 Motivation 1
1.2 Contributions 3
1.2.1 Toolchain for Workload Characterization 3
1.2.2 Rapid Development of Dynamic Program Analysis Tools 3
1.2.3 Workload Characterization of JVM Languages 4
1.3 Dissertation Outline 4
1.4 Publications 5

2 State of the Art **9**
2.1 Overview 9
2.2 Feedback-Directed Optimizations 9
2.3 Workload Characterization 10
2.4 Programming Languages Comparison 13
2.5 Instrumentation 14
2.5.1 Binary Instrumentation 14
2.5.2 Bytecode Instrumentation 15

3 Rapid Development of Dynamic Program Analysis Tools **19**
3.1 Motivation 19
3.2 Background: DiSL Overview 21
3.3 Quantifying the Impact of DiSL 23
3.3.1 Experiment Design 23
3.3.2 Task Design 26
3.3.3 Subjects and Experimental Procedure 26
3.3.4 Variables and Analysis 27
3.3.5 Experimental Results 28
3.3.6 Threats to Validity 29
3.4 DiSL Tools Are Concise and Efficient 31
3.4.1 Overview of Recasted Analysis Tools 31

3.4.2 Instrumentation Conciseness Evaluation 34
3.4.3 Instrumentation Performance Evaluation 36
3.5 Summary 39

4 Toolchain for Workload Characterization 41
4.1 Dynamic Metrics 41
4.1.1 Object access concerns 42
4.1.2 Object allocation concerns 44
4.1.3 Code generation concerns 46
4.2 Toolchain Description 48
4.2.1 Deployment and Use 48
4.2.2 Query-based Metrics 50
4.2.3 Instrumentation 51

5 Workload Characterization of JVM Languages 53
5.1 Experimental Setup 53
5.1.1 Examined Metrics 53
5.1.2 Workloads 54
5.1.3 Measurement Context 55
5.2 Experiment Results 55
5.2.1 Call-site Polymorphism 55
5.2.2 Field, Object, and Class Immutability 64
5.2.3 Identity Hash-code Usage 68
5.2.4 Unnecessary Zeroing 70
5.2.5 Object Lifetimes 72
5.3 Inlining in the JVM 77
5.3.1 Inlining Decision 77
5.3.2 Top Reasons for Inlining Methods 80
5.3.3 Top Reasons for Not Inlining Methods 83
5.3.4 Inlining Depths 85
5.3.5 Fraction of Inlined Methods 89
5.3.6 Fraction of Decompiled Methods 91
5.3.7 Summary 93

6 Conclusions 95
6.1 Summary of Contributions 96
6.2 Future Work 97

Bibliography 99

Chapter 1

Introduction

1.1 Motivation

Modern applications have brought about a paradigm shift in software development. With different application parts calling for different levels of performance and productivity, developers use different languages for the various tasks at hand. Very often, core application parts are written in a statically-typed languages, data management uses a suitable domain-specific language (DSL), and the glue code that binds all the pieces together is written in a dynamically-typed language. This development style is called *polyglot programming*, and promotes using the right language for a particular task [90].

This shift has gained support even in popular managed runtimes such as the .NET Common Language Runtime and the Java Virtual Machine (JVM), which are mainly known for automatic memory management, high performance through Just-In-Time (JIT) compilation and optimization, and a rich class library. In case of the JVM, the introduction of scripting support (JSR-223) and support for dynamically-typed languages (JSR-292) to the Java platform enables scripting in Java programs and simplifies the development of dynamic language runtimes. Consequently, developers of literally hundreds of programming languages target the JVM as the host for their language—both to avoid developing a new runtime from scratch, and to benefit from the JVM's maturity, ubiquity, and performance. Today, programs written in popular dynamic languages such as JavaScript, Ruby, Python, or Clojure (a dialect of Lisp) can be run on the JVM, creating an ecosystem that boosts developer productivity.

Even though being targeted by so many languages, the JVM was originally designed with a single language in mind, namely Java, and therefore was tuned with respect to characteristics of Java programs only. Dynamically-typed languages such as Clojure, Groovy, JRuby, and Jython suffered from performance problems until recently. Some of them have been addressed with introduction of the `invokedynamic` bytecode in Java 7, intended to improve performance of dynamic JVM languages. However, it did not result in significant performance benefits [116]. To gain performance when repurposing

an existing JIT compiler for dynamically-typed languages, compiler developers are encouraged to specialize the generic execution (inherent to such languages) as much as possible, instead of overly relying on the original JIT to gain performance [25].

Making the JVM perform well with various statically- and dynamically-typed languages clearly requires significant effort, not only in optimizing the JVM itself, but also, more importantly, in optimizing the bytecode-emitting language compilers. This in turn requires that developers of both the language compilers and the JVM need to understand the characteristics of the workloads imposed by various languages or compilation strategies on the JVM. Therefore, the research question of this dissertation is:

What are the differences in the workloads imposed on the JVM by different programming languages?

In order to answer stated research question, we require the means of characterizing the full range of workloads on the JVM, including applications written in different JVM languages. Two classes of artifacts are useful for workload characterization: benchmarks and metrics. The former draw representative samples from the space of application code, while the latter identify useful performance dimensions within their behaviour. Whereas benchmarking shows *how well* a system performs at different tasks, metrics show *in what way* these tasks differ from each other, providing essential guidance for optimization effort.

Ideally, metrics should capture the differing properties both of Java and non-Java workloads and should provide useful information for developers of JVM languages and JVM implementers. For example, a developer might hypothesize that a workload performed poorly because of heap pressure generated by increased usage of boxed primitive values, which are used relatively rarely in normal Java code, but frequently in some other JVM languages such as in JRuby. Developers could optimize their bytecode generator, for example, to try harder at using primitives in their unboxed form. A dynamic metric of boxing behaviour would allow these developers to quantify the effects of such optimizations.

Meanwhile, JVM developers may also benefit from the metrics, but in a rather different way. JVM optimizations are dynamic and adaptive. Each optimization decision is guarded by a heuristic decision procedure applied to profile data collected at runtime. For example, the decision whether to inline a callee into a fast path depends on factors such as the hotness of that call site (evaluated by dynamic profiling) and the size of the callee. JVMs can therefore benefit from better heuristics which more accurately match real workloads, including non-Java workloads.

1.2 Contributions

Answering the research question of this dissertation requires versatile research that has to address several needs: (i) the need for tools to perform workload characterization; (ii) the need to rapidly develop those tools; (iii) the actual characterization of different kinds of workloads on the JVM.

1.2.1 Toolchain for Workload Characterization

Dynamic program analysis tools support different software engineering tasks, including profiling [15, 69], debugging [6, 26, 44, 132], and program comprehension [84, 104]. Identifying the intrinsic and differing properties of different JVM workloads can be achieved also by means of dynamic program analysis tools.

There are several approaches serving workload characterization, however, no existing work has defined a comprehensive set of metrics and provided the tools to compute them. Rather, existing approaches are fragmented across different infrastructures: many lack portability by using a modified version of the JVM [38, 74], while others collect only architecture-dependent metrics [114]. In addition, at least one well-known metric suite implementation [41] runs with unnecessarily high performance overhead. Ideally, metrics should be collected within reasonable time, since this enables the use of complex, real-world workloads and shortens the development cycles. Metrics should also be computed based on observation of the whole workload, which not all infrastructures allow. For example, existing metrics collected using AspectJ are suboptimal since they lack coverage of code from the Java class library [19, 27, 94].

We present our approach which bases all metrics on a unified infrastructure which is JVM-portable, offers non-prohibitive runtime overhead with near-complete bytecode coverage, and can compute a full suite of metrics “out of the box”. Among the metrics of interest to be collected by our toolchain are object allocations, method and basic block hotness, the degree of call-site polymorphism, stack usage and recursion, instruction mix, use of immutability and synchronization, amount of unnecessary zeroing, and use of hash codes.

1.2.2 Rapid Development of Dynamic Program Analysis Tools

All the tools from our toolchain rely on bytecode instrumentation, which is usually performed by means of low-level libraries such as BCEL [124], ASM [91], Soot [127], Shrike [66], or Javassist [28]. However, even with those libraries, bytecode instrumentation is an error-prone task and requires advanced expertise from the developers. Due to the low-level nature of the Java bytecode, the resulting code is often verbose, complex, and difficult to maintain or to extend.

The complexity associated with manipulating Java bytecode can be sometimes avoided by using aspect-oriented programming (AOP) [70] to implement the instrumen-

tation. This is possible because AOP provides a high-level abstraction over predefined points in program execution (join points) and allows inserting code (advice) at a declaratively specified set of join points (pointcuts). Tools like the DJProf profiler [94], the RacerAJ data race detector [19], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [104] are examples of successful applications of this approach.

Having many choices for performing bytecode instrumentation, developers face a hard time of choosing the proper instrumentation framework that allows *rapid* development of *efficient* dynamic program analysis tools. To the best of our knowledge, no such quantification is present in the literature concerning instrumentation of Java programs. We address this problem by performing a thorough empirical evaluation: (i) we conduct a controlled experiment to determine which bytecode instrumentation framework increases developer productivity; (ii) we collect different performance and source-code metrics for recasts of ten open-source dynamic program analysis tools in order to find out which framework allows development of efficient dynamic program analysis tools.

1.2.3 Workload Characterization of JVM Languages

Recently, dynamic languages gained a lot of attention due to their flexibility and ease of use. While originally designed for scripting purposes only (e.g., for text processing or for glueing together different components in a large system), they are currently used as general-purpose programming languages along with statically typed ones. After introducing scripting support (JSR-223) and support for dynamically-typed languages (JSR-292), the JVM has become an attractive environment for developers of new dynamic programming languages. However, the JVM appeared two decades ago with a single language in mind and was build specifically for running Java applications; therefore, its heuristics were tuned for efficiently running Java applications. The question whether workloads written in a non-Java language run efficiently on the JVM remains open.

In this dissertation we perform a thorough workload characterization of popular JVM languages such as Java, Clojure, JRuby, Jython, JavaScript, and Scala. We collect numerous bytecode-level metrics and discuss their implications. Moreover, we shed a light on the most efficient compiler optimization, namely method inlining, and identify whether the JVM handles workloads written in different JVM languages equally well as Java workloads.

1.3 Dissertation Outline

This dissertation is structured as follows:

- Chapter 2 discusses the state of the art in the area of workload characterization. It gives an overview of the existing techniques and discusses their limitations. The chapter also discusses bytecode and binary instrumentation techniques that are usually used in the the area of dynamic program analysis. Finally, it gives an overview of studies that compare programming languages.

- Chapter 3 introduces a user study on three popular frameworks for performing bytecode instrumentation. It presents a controlled experiment which aims at identifying a framework that boosts developer productivity. The chapter also presents a second case study, empirical evaluation of the recasts of instrumentation parts of ten existing open-source dynamic program analysis tools.

- Chapter 4 focuses on our toolchain for collecting various dynamic metrics. It gives an overview of the metrics that can be collected by our toolchain, together with a detailed description of the techniques for collecting the metrics.

- Chapter 5 presents the study – workload characterization of six different JVM languages (Clojure, Java, JavaScript, JRuby, Jython, and Scala). The chapter presents the results of applying our toolchain to workloads written in the examined languages and discusses the collected metrics.

- Chapter 6 concludes the dissertation and outlines future research directions opened by this work.

1.4 Publications

This dissertation is based on several published and submitted work. The empirical evaluation of different instrumentation frameworks (Chapter 3) was published in:

- Aibek Sarimbekov, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Lukáš Marek, Walter Binder, Petr Tůma, and Zhengwei Qi. Dynamic Program Analysis – Reconciling Developer Productivity and Tool Performance. *Science of Computer Programming.*, 2014.

- Aibek Sarimbekov, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Lukáš Marek, Walter Binder, Petr Tůma, and Zhengwei Qi. Productive Development of Dynamic Program Analysis Tools with DiSL. *In Proceedings of 22nd Australasian Software Engineering Conference (ASWEC)*, pp. 11–19. Melbourne, Australia, 2013.

The description of the toolchain and new dynamic metrics (Chapter 4) was published in:

- Aibek Sarimbekov, Andreas Sewe, Stephen Kell, Yudi Zheng, Walter Binder, Lubomír Bulej, and Danilo Ansaloni. A comprehensive toolchain for workload

characterization across JVM languages. *In Proceedings of the 11th Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pp. 9–16. Seattle, Washington, 2013.

The work on the calling context profiler JP2 and on its applicability (Chapter 4) was published in:

- 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*.79:146–157, 2014.

- Aibek Sarimbekov, Walter Binder, Philippe Moret, Andreas Sewe, Mira Mezini, and Martin Schoeberl. Portable and Accurate Collection of Calling-Context-Sensitive Bytecode Metrics for the Java Virtual Machine. *In Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, pp. 11–20, Kongens Lyngby, Denmark, 2011.

- Aibek Sarimbekov, Philippe Moret, Walter Binder, Andreas Sewe, and Mira Mezini. Complete and Platform-Independent Calling Context Profiling for the Java Virtual Machine. *In Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, pp. 61–74. Saarbrucken, Germany, 2011.

- Aibek Sarimbekov, Walter Binder, Andreas Sewe, Mira Mezini, and Alex Villazón. JP2 – Collecting Dynamic Bytecode Metrics in JVMs. *In Proceedings of the Conference Companion on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 35–36, Portland, Oregon, 2011.

The work on workload characterization of different JVM languages (Chapter 5) was published in (or is under review in):

- Aibek Sarimbekov, Lukas Stadler, Lubomír Bulej, Andreas Sewe, Andrej Podzimek, Yudi Zheng, and Walter Binder. Workload Characterization of JVM Languages. Currently under review in *Software: Practice and Experience*, 2014.

- Aibek Sarimbekov, Andrej Podzimek, Lubomír Bulej, Yudi Zheng, Nathan Ricci, and Walter Binder. Characteristics of dynamic JVM languages. *In Proceedings of the 7th Workshop on Virtual Machines and Intermediate Languages (VMIL)*, pp. 11–20. Indianapolis, Indiana, 2013.

- Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan P. Ricci, and Samuel Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. *In Proceedings of International Symposium of Memory Managment (ISMM)*, pp. 97–108, Beijing, China, 2012.

- 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 657–676, Portland, Oregon, 2011.

During my Ph.D. studies I was also involved in other projects that resulted in the following publications:

- Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder and Petr Tůma. Introduction to Dynamic Program Analysis with DiSL. *Science of Computer Programming*, 2014.

- Aibek Sarimbekov. Comparison of Instrumentation Techniques for Dynamic Program Analysis. *In Proceedings of 12th International Conference Companion on Aspect-oriented Software Development (AOSD)*, pp. 31–32, Fukuoka, Japan, 2013.

- Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. ShadowVM: Robust and Comprehensive Dynamic Analysis for the Java Platform. *In Proceedings of 12th International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pp. 105–114, Indianapolis, Indiana 2013.

- Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Lubomír Bulej, Aibek Sarimbekov, Walter Binder, and Zhengwei Qi. Introduction to Dynamic Program Analysis with DiSL. *In Proceedings of 4th International Conference on Performance Engineering, (ICPE)*, pp. 429–430, Prague, Czech Republic, 2013.

- Danilo Ansaloni, Walter Binder, Christoph Bockish, 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 Proceedings of 11th International Conference on Software Composition (SC)*, pp. 86–96, Prague, Czech Republic, 2012.

- Lukáš Marek, Yudi Zheng, Danilo Ansaloni, Aibek Sarimbekov, Walter Binder, and Zhengwei Qi. Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis. *In Proceedings of 10th Asian Symposium on Programming Languages and Systems (APLAS)*, pp. 256–263, Kyoto, Japan, 2012.

- Walter Binder, Philippe Moret, Danilo Ansaloni, Aibek Sarimbekov, Akira Yokokawa, and Éric Tanter. Towards a domain-specific aspect language for dynamic program analysis: position paper. In Proceedings of 6th Workshop on Domain-specific Aspect Languages (DSAL), pp. 9–11, Porto de Galinhas, Brasil, 2011.

Chapter 2

State of the Art

2.1 Overview

In this chapter we discuss the state of the art in the area of workload characterization and comparison of programming languages. We give an overview of the existing techniques and discuss their limitations. We first describe what feedback-directed optimizations (FDO) are and discuss their relevance to this dissertation. The chapter also presents one of the main techniques used in FDO, namely program instrumentation.

2.2 Feedback-Directed Optimizations

Feedback-directed optimization is a well-known mechanism used for improving program execution based on its runtime behaviour. There are two types of FDO: off-line and online. In an off-line FDO the programmer first runs the application, gathers some statistics summarizing the behavior of the examined application and after using the collected statistics creates a new version of the application. Therefore, off-line refers to the fact that optimization takes place only after the application run [118], opposed to the online one, where optimizations take place during the application runtime. FDO techniques are heavily used in the area of computer architecture, i.e., in the production of processor chips [49, 103]. Caches, instruction scheduling, register allocations – all these are examples of FDO. FDO is also used in other fields, such as compiler development and software engineering.

Profile-guided compilation (PGC) [35, 64, 117] is a widely known technique used by modern compilers. In PGC the compiler tries to optimize the parts of the program that are frequently executed. Different profiling techniques, such as block [97], edge [45], or path [12] profiling are used to identify frequently executed program parts.

Another approach is offline optimization based on continuous profiling, where the profile collection takes place on the user's machine continuously [3]. After the application terminates, the system will reoptimize the program offline, if necessary.

Opposed to PGC this approach is considered to be more adaptive and to incur less overhead [3, 61, 135].

Techniques for code generation [36, 53] stage the compilation of the program so that the optimization can take place during the program run, instead of performing the optimization during the compile time as in the previously discussed approaches. Partial evaluation is used to identify which parts of the application can benefit from the optimizations based on the information that can be collected during the runtime only. Since the majority of the optimizations anyway happens at compile time, this approach exhibits low overhead.

The most prominent FDO technique in the area of compiler development is online adaptive optimizations [4, 31, 92, 122]. As an example of this technique, one can consider HotSpot's JIT compiler [92] and IBM Jikes RVM's adaptive optimization system [4]. The optimization process in these systems is two staged, first the code is run in interpreted mode and after determining the frequently executed code the system performs additional optimizations. A similar approach is used by the Dynamo system [11].

The work presented in this dissertation uses the techniques that fall in the category of offline FDO. However, our system does not perform any optimizations, we leave this task to the developer who should take responsibilities between different trade-offs that usually happen while making optimizations (e.g., longer startup time vs. faster startup time) [63]. Another reason for not performing optimizations is our primary design goal: we opted to use a standard JVM and do not rely on any modified version of it. An example of such an approach is described in [5], where the authors extended the adaptive optimization system of the Jikes RVM [4]. Our primary goal was to equip JVM developers and JVM language implementers with tools that provide insights of the behaviour imposed by novel JVM languages. Therefore, our system only hints at possible optimizations that can be made.

2.3 Workload Characterization

After a careful study of the related work in the area of workload characterization, we identified several limitations that we try to address with our approach.

Excessive overhead.

Dufour et al. [41] define dynamic platform-independent metrics that serve for categorizing programs according to their dynamic behaviour in five areas: size, data structure, memory use, concurrency, and polymorphism. The tool *J [42] used by the authors relies on the JVMPI[1], a deprecated[2] profiling interface for the JVM. Apart from being

[1] `http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/index.html`

[2] JVMPI was deprecated in Java 5, and JVMTI is now offered instead.

not maintained anymore, it exhibits huge performance overhead making their approach hardly applicable to more complex Java workloads.

Ricci et al. [101] created a tool named ElephantTracks that implements the Merlin algorithm [59] for computing object life times. It produces traces that contain each object allocation and death, method entries and exits, and all pointer update events. Garbage collector (GC) implementers design and evaluate new GC algorithms by simulation based on those traces. However, this tracing tool introduces prohibitive overhead (e.g., it can take several days to run it on a single DaCapo benchmark with small workload size).

Architecture-dependent metrics.

Ammons et al. [2] use hardware performance counters for performing context sensitive and flow sensitive profiling. The authors rely on binary instrumentation using Executable Editing Library in order to record profiling data. Their tool records hardware counters of UltraSPARC processors, such as instructions executed, cycles executed, and cache misses. Calling-context trees (CCT) are used to represent the execution of the application and each CCT is associated with hardware metrics.

Shiv et al. [115] compare the SPECjvm2008[3] and SPECjvm98[4] benchmark suites. The authors present a quantitative evaluation based on different JVM- and architecture-dependent metrics; they look at the effectiveness of Java runtime systems including JIT compilation, dynamic optimizations, synchronization, and object allocation, and report the statistics also for SPECjAppServer2004[5] and SPECjbb2005[6].

These approaches strongly depend on the chosen architecture and cannot be generalized. Moreover, as shown in [62], microarchitecture-dependent metrics hide inherent program behaviour, while microarchitecture-independent metrics are more effective and useful in workload characterization.

Modified JVM.

Daly et al. [38] analyze the Java Grande benchmark suite [24] using JVM-independent metrics. The authors consider static and dynamic instruction mix and use five different Java-to-bytecode compilers in order to identify the impact of the choice of a compiler on the dynamic bytecode frequency. For collecting the metrics of interest, the authors use a modified version of the Kaffe Java Virtual Machine[7].

Other researchers also use a modified version of a virtual machine in order to facilitate workload characterization, thus sacrificing portability. For instance, Gregg et al. [55] use a modified version of the Kaffe Java Virtual Machine for collecting a number of actual invocations to native methods. This approach is also used in [57, 74].

[3] http://www.spec.org/jvm2008/
[4] http://www.spec.org/jvm98/
[5] http://www.spec.org/jAppServer2004
[6] http://www.spec.org/jbb2005
[7] http://www.kaffe.org/

Ha et al. [56] implemented a concurrent analysis framework for multicore hardware. Their implementation is based on a modified version of the Jikes RVM [1]. To assess the framework the authors created 5 different well-known dynamic analyses and applied them to the SPECjvm2008 and the Dacapo 2006 benchmarks.

Limited coverage.

Pearce et al. [94] applied aspect-oriented programming (AOP) to create profiling tools. The authors used AspectJ to profile heap usage and object lifetime. Even though the study shows that AspectJ eases the development of tools for workload characterization, it suffers from a severe limitation. It has limited coverage and does not support profiling classes from the Java class library. The same limitation is true for most AOP-based approaches.

Bodden et al. [19] created RacerAJ that implements the ERASER algorithm [109] for data-race detection. At runtime, RacerAJ monitors all field accesses and lock acquisitions and releases, and reports a potential data race when a field had a write access from multiple threads without synchronizing the accesses. To maintain various per-thread and per-field data structures, RacerAJ uses an AOP-based instrumentation to intercept all field accesses, and all lock acquisitions and releases, both due to synchronized method invocations and due to entering syncrhonized blocks.

Chen et al. [27] developed an aspect-based memory leak detector for Java programs named FindLeaks. The tool identifies loitering objects using the constructor call and field set join points of AspectJ. It reports loitering objects, their construction site and their referencing objects. It also points out where in the source code references to loitering objects were created. However, the tool does not allow to identify a memory leak if no explicit allocation is made in the application code. This limitation can be solved by analyzing also the classes from the Java class library [128, 129].

Other approaches.

Jovic et al. [69] developed the tool LagHunter for detecting performance problems in interactive applications. The authors propose to measure latency instead of the common method hotness metric. Their tool collects so-called landmark methods that contain timing information for measuring lags. These landmark methods represent potential performance issues. In contrast to approaches using complete traces of method calls and returns, LagHunter results in small performance overhead. The authors used their tool for finding performance problems in the Eclipse IDE.

Zaparanuks et al. [134] presented an algorithmic profiling tool. The intention of the tool is to identify to what extent the implementation of an algorithm contributes to the overall performance by automatically producing cost functions. Using these cost functions, developers can understand the complexity of the algorithm and see which inputs cause long execution times.

Binder et al. [14] evaluate the contribution of native methods to Java workloads,

using SPECjvm98 and SPECjbb2005[8] benchmarks on a previous release of the Java JRE 1.6. They conclude that less than 20% of the execution time is spent in native code, thus confirming that platform-independent performance analysis is appropriate for most of the Java workloads.

Ratanaworabhan et al. [100] compare JavaScript benchmarks with real web applications and compute different static and dynamic platform-independent metrics, such as instruction mix, method hotness, number of executed instructions. The authors use an instrumented version of the popular Internet Explorer browser, however the methodology is browser agnostic and can be used with any browser.

Richards et al. [102] apply the same approach to analyze the dynamic behavior of JavaScript, although they do not consider event-driven web applications that happen to be often the case of nowadays popular JavaScript applications as shown in [102].

In contrast to existing work, our approach is focused on obtaining dynamic metrics that spot potential optimizations. Our approach is based on bytecode instrumentation and can be run on any production JVM. It produces JVM-independent metrics that can be obtained in a reasonable amount of time. Our approach supports full bytecode coverage, i.e., it covers every method that has bytecode representation (including dynamically downloaded or generated classes together with the classes from the Java class library).

2.4 Programming Languages Comparison

The first attempts of comparing different languages were done 4 decades ago [126], where Algol 68 and PL/I were compared. Then, several other papers were published on the same topic [81, 123].

In [99] the authors present an empirical evaluation of 7 different programming languages. Performance, source code metrics, and memory consumption of C, C++, Java, Tcl, Ruby, Python, and Rexx applications are considered. As workloads, the authors use different implementations of the same application written in different languages by different programmers, thus avoiding the threat to validity of comparing a single implementation of an application.

In [58] the authors conduct a controlled experiment with subjects in order to compare object-oriented languages with procedural ones in the area of software maintenance. The authors asked the subjects to perform a set of typical software maintenance tasks and assess the difficulty of each task. C and Objective-C are the languages that were compared during the experiment. The authors find that systems developed in object-oriented languages are easier to maintain compared to systems developed in procedural languages.

In [10] the author compares 5 different parallel programming languages based on different paradigms. The author implemented 3 parallel programs in all the 5

[8] http://www.spec.org/jbb2005/

languages and reported on the programming experience, language implementation and performance. Since no platform exists on which all the languages can run, fair performance comparison was not possible, although the author used many different platforms in order to give rough estimations of the observed performance.

In [50] the authors report on a comprehensive comparison of parametric polymorphism in six programming languages: C++, Standard ML, Haskell, Eiffel, Java[9], and Generic C#. The authors wanted to identify what language features were necessary to support generic programming and to provide guidance for the development of language support for generics.

While the topic on comparison of programming languages has always been popular, our intentions are different from previous work in the area. In this dissertation we are not comparing the performance of either programming language, instead we try to identify how well the JVM handles them. Therefore, the methodology used in our dissertation is rather different. The details of the study will follow in Chapter 5.

2.5 Instrumentation

In the following section we give an overview of instrumentation techniques for creating tools for workload characterization. Even though we rely on bytecode instrumentation for building our tools, for the sake of completeness we describe also binary instrumentation techniques.

2.5.1 Binary Instrumentation

Binary instrumentation helps one to analyze native methods that do not have any bytecode representation.

ATOM [119] is a binary instrumentation framework for building customized program analysis tools. ATOM allows selective instrumentation – only specified points in the application are targets for the instrumentation. The user has to provide a file that contains an instrumentation routine, a file with analysis routines and the application that has to be instrumented. ATOM takes care of avoiding interference between the code in the analysis routine and the code in the application itself by having two copies of the method in the executable.

Pin [77] is a tool that follows the ATOM approach. The tool developer needs to analyze an application at the instruction level without the need for detailed knowledge of the underlying architecture. Pin can attach to a running application, instrument it, collect profiles and, finally, detach. Pin has a rich API allowing to write customized instrumentation tools in C/C++.

Javana [78] is a tool that performs binary instrumentation underneath the virtual machine. High-level language constructs such as objects, methods, and threads are

[9]The authors use Java 5.

linked to the low-level native instructions and memory addresses in a vertical map through event handling mechanism. The instrumentation layer communicates with the virtual machine. Javana allows different events to be intercepted, such as class loading, object allocation and relocation, method compilation, Java thread creation and termination. Javana uses a modified version of the Jikes RVM [1].

DIOTA [79] is a tool that allows instrumentation of the running application. DIOTA does not alter the original application, it generates an instrumented version on the fly and keeps it in another memory location. The generated code will be created in such a way, that all data accesses will be taken from the original application, whereas code accesses will be taken from the instrumented version. In such a way, self-modifying code can be handled correctly. DIOTA uses backends that are responsible for what and how the code has to be instrumented.

Valgrind [85] is a dynamic binary instrumentation framework with a distinguishing feature, the support for shadow values. It is used for developing tools that detect memory-management problems, by intercepting all memory reads and writes, memory allocations, and frees. Valgrind's core and the tool to be instrumented are loaded into a single process in order to avoid inter-process communication overhead.

EEL [73] is a tool for editing executables. It provides abstractions that allow one to modify executables without being concerned about the underlying architecture or operating system. Moreover, EEL provides portability across a wide range of systems. EEL provides five abstractions: executable, routine, control flow graph, instruction, and snippet. EEL allows editing control flow graphs by deleting or adding instructions, however it can be potentially dangerous, since instructions that corrupt the state of an application can be inserted.

DynamoRIO [23] is a framework for building customized program analysis tools. It operates with two kinds of code sequences: basic blocks and traces that are represented as linked lists of instructions. DynamoRIO copies basic blocks into a code cache and then executes them natively. A context switch happens at the end of each basic block and the control returns to DynamoRIO for copying the next basic block. The goal of DynamoRIO is to observe and manipulate, if needed, every single instruction prior to execution.

2.5.2 Bytecode Instrumentation

Java bytecode instrumentation is a common technique used in dynamic program analysis tools. It is usually performed by means of low-level libraries that require in-depth knowledge of the bytecode.

BCEL [37] provides a low-level API to analyze, create, and transform Java class files. Java classes are represented as objects that contain all the information of the given class: constant pools, methods, fields and bytecode instructions.

ASM [91] is similar to BCEL and allows generation of stub classes or other proxy classes. It also supports load-time transformation of Java classes. Classes in ASM can

be represented in two ways: an object representation that exposes each class file as a tree of objects and an event-based representation that generates an event each time a specific element of a class file is parsed.

WALA [67] stands for IBM T.J. Watson Libraries for Analysis. WALA contains libraries both for static and dynamic program analysis. While WALA IR is immutable and no code generation is provided, it has limited code transformation abilities through Shrike. Shrike has a patch-based API that atomically applies all modifications to a given method body. For load-time bytecode transformations one has to use Dila [65] which is based on Shrike.

Spoon [93] is a framework for program transformation and static analysis in Java, which reifies the program with respect to a meta-model. This allows for direct access and modification of its structure at compile-time and enables template-based AOP, users can insert code, e.g., before or after a method body. Spoon uses source code-level transformations.

Javassist [28] is a load-time bytecode manipulation library that enables structural reflection, i.e. alter data structures in the program, which are statically fixed at compile time. Javassist provides convenient source-level abstractions, allowing its use without knowledge of Java bytecode. It additionally supports bytecode-level API allowing one to directly edit the class file.

Soot [127] is a bytecode optimization framework. Soot supports multiple bytecode representations in order to simplify the analysis and the transformation of Java bytecode. Soot can be used as a stand alone tool to optimize or inspect Java class files, as well as a framework to develop optimizations or transformations on Java bytecode.

Josh [29] is an AspectJ-like language that allows developers to define domain-specific extensions to the pointcut language. Josh is inspired by the idea of the open-compiler approach [72]. Internal structure and the behaviour of the compiler is represented as understandable abstractions and the open-compiler provides programming interface to customize the compiler through the abstractions. Extensions to the pointcut language can be developed as optional libraries or compiler plug-ins. Josh is built on top of Javassist [28].

Sofya [71] is a framework that allows rapid development of dynamic program analysis tools. It has a layered architecture in which the lower levels act as an abstraction layer on top of BCEL, while the top layers hide low-level details about the bytecode format and offer a publish/subscribe API that promotes composition and reuse of analyses. An Event Description Language (EDL) allows programmers to define custom streams of events, which can be filtered, splitted, and routed to the analyses.

RoadRunner [47] is a framework for composing different small and simple analyses for concurrent programs. Each analysis can stand on its own, but by composing them one can obtain more complex ones. Each dynamic analysis is essentially a filter over event streams, and filters can be chained. Per program run, only one chain of analyses can be specified, thus avoiding the combination of arbitrary analyses in incompatible

way.

Chord[10] is a framework that provides a set of common static and dynamic analyses for Java. Moreover, developers can specify custom analyses, possibly on top of the existing ones. Chord provides a rich and extensible set of low-level events that can be intercepted.

DiSL [80, 136] is a domain-specific language for instrumentation. DiSL enables rapid development of efficient instrumentations for Java-based dynamic program analysis tools. It is built on top of ASM and provides higher abstraction layer at which the instrumentations can be specified.

[10] `http://pag.gatech.edu/chord`

Chapter 3

Rapid Development of Dynamic Program Analysis Tools

3.1 Motivation

With the growing complexity of computer software, dynamic program analysis (DPA) has become an invaluable tool for obtaining information about computer programs that is difficult to ascertain from the source code alone. Existing DPA tools aid in a wide range of tasks, including profiling [15], debugging [6, 44, 132], and program comprehension [84, 104]. DPA tools also serve the task of workload characterization that we aim to conduct in this dissertation.

The implementation of a typical DPA tool usually comprises an analysis part and an instrumentation part. The analysis part implements algorithms and data structures, and determines what points in the execution of the analyzed program must be observed. The instrumentation part is responsible for inserting code into the analyzed program. The inserted code then notifies the analysis part whenever the execution of the analyzed program reaches any of the points that must be observed.

There are many ways to instrument a program, but the focus of this dissertation is on Java bytecode manipulation. Since Java bytecode is similar to machine code, manipulating it is considered difficult and is usually performed using libraries such as BCEL [124], ASM [91], Soot [127], Shrike [66], or Javassist [28]. However, even with those libraries, writing the instrumentation part of a DPA tool is error-prone and requires advanced expertise from the developers. Due to the low-level nature of the Java bytecode, the resulting code is often verbose, complex, and difficult to maintain or to extend.

The complexity associated with manipulating Java bytecode can be sometimes avoided by using aspect-oriented programming (AOP) [70] to implement the instrumentation part of a DPA tool. This is possible because AOP provides a high-level abstraction over predefined points in program execution (join points) and allows inserting code

(advice) at a declaratively specified set of join points (pointcuts). Tools like the DJProf profiler [94], the RacerAJ data race detector [19], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [104], are examples of successful applications of this approach.

AOP, however, is not a general solution to DPA needs—mainly because AOP was not primarily designed for DPA. AspectJ, the de-facto standard AOP language for Java, only provides a limited selection of join point types and thus does not allow inserting code at the boundaries of, e.g., basic blocks, loops, or individual bytecodes. Another important drawback is the lack of support for custom static analysis at instrumentation time, which can be used, e.g., to precompute static information accessible at runtime, or to select join points that need to be captured. An AOP-based DPA tool will usually perform such tasks at runtime, which can significantly increase the overhead of the inserted code. This is further aggravated by the fact that access to certain static and dynamic context information is not very efficient [16].

To leverage the syntactic conciseness of the pointcut-advice mechanism found in AOP without sacrificing the expressiveness and performance attainable by using the low-level bytecode manipulation libraries, the DiSL [80, 136] framework was introduced. DiSL is an open-source framework that enables rapid development of efficient instrumentations for Java-based DPA tools. DiSL achieves this by relying on AOP principles to raise the abstraction level (thus reducing the effort needed to develop an instrumentation), while avoiding the DPA-related shortcomings of AOP languages (thus increasing the expressive power and enabling instrumentations that perform as well as instrumentations developed using low-level bytecode manipulation libraries).

Having many choices for specifying instrumentations, DPA tool developers face a hard time of choosing the right instrumentation framework that fully satisfies their needs. To the best of our knowledge, no such quantification is present in the literature concerning instrumentation of Java programs.

The purpose of this chapter, therefore, is to *quantify* the usefulness of instrumentation frameworks when developing DPA tools. Specifically, we aim to address the following research questions:

RQ1 Which instrumentation framework improves developer productivity in writing instrumentations for DPA?

RQ2 Do instrumentations written in a high-level style perform as fast as their equivalents written using low-level libraries?

To answer the research questions, we conduct a controlled experiment to determine the framework that increases developer productivity. We also perform an extensive evaluation of 10 existing open source DPA tools, in which we reimplement their instrumentation parts using DiSL, which offer a high-level approach for writing instrumentations. We compare reimplemented and the original instrumentation parts of those 10 DPA tools. With respect to RQ1, the controlled experiment provides evidence of

increased developer productivity, supported by the evidence of more concise expression of equivalent instrumentations obtained by comparing the sizes of the original and DiSL-based instrumentations in terms of logical lines of code. Regarding RQ2, we compare the overhead of the evaluated DPA tools on benchmarks from the DaCapo [17] suite using both the original and the DiSL-based instrumentation.

3.2 Background: DiSL Overview

DiSL[1] is a domain-specific language that provides developers of DPA tools with high-level concepts similar to those in AOP, without taking away the expressiveness and performance that can be attained when developing instrumentations using low-level bytecode manipulation libraries.

The key concepts raising the level of abstraction in DiSL instrumentations are *markers* and *snippets*, complemented by *scopes* and *guards*. A marker represents a class of potential instrumentation sites and is similar to a *join point* in AOP. DiSL provides a predefined set of markers at the granularity of methods, basic blocks, loops, exception handlers, and individual bytecodes. Since DiSL follows an open join point model, programmers can implement custom markers to represent the desired instrumentation sites.

Snippets contain the instrumentation code and are similar to *advice* in AOP. Snippets are inlined *before* or *after* an instrumentation site, with the usual semantics found in AOP. The snippet code can access any kind of static context information (e.g., class and method name, basic block index), and may also inspect the dynamic context of the executing method (e.g., stack and method arguments).

Scopes and guards restrict the application of snippets. While scopes are based on method signature matching, guards contain Java code capturing potentially complex conditionals evaluated at instrumentation time. Snippets annotated with markers, scopes, and guards are colocated in a class referred to as DiSL *instrumentation*, which is similar to an *aspect* in AOP.

To illustrate the basic DiSL concepts and their similarity to AOP, Figures 3.1 and 3.2 show the source code of a simple tracing tool implemented using AspectJ and DiSL, respectively. On each method entry and method exit, the tool should output the full name of the method and its signature.

In the AspectJ version, the executionPointcut () construct selects method executions restricted to the desired class, while the before () and after () constructs define the advice code that should be run before and after method execution. Within the advice code, the thisJoinPointStaticPart pseudo-variable is used to access static information, e.g., method name, related to each join-point where the advice is applied.

In the DiSL version, we define two code snippets, represented by the onMethodEntry() and onMethodExit() methods, that print out the method name and signature before

[1] http://disl.ow2.org

```
pointcut executionPointcut () : execution (* HelloWorld.* (..));

before (): executionPointcut () {
  System.out.println ("On "+ thisJoinPointStaticPart.getSignature () +" method entry");
}

after (): executionPointcut () {
  System.out.println ("On "+ thisJoinPointStaticPart.getSignature () +" method exit");
}
```

Figure 3.1. Tracing tool implemented using AspectJ.

```
@Before (marker = BodyMarker.class, scope = "*.HelloWorld.*")
void onMethodEntry (MethodStaticContext msc) {
  System.out.println ("On "+ msc.thisMethodFullName () +" method entry");
}

@After (marker = BodyMarker.class, scope = "*.HelloWorld.*")
void onMethodExit (MethodStaticContext msc) {
  System.out.println ("On "+ msc.thisMethodFullName () +" method exit");
}
```

Figure 3.2. Tracing tool implemented using DiSL.

and after executing a method body. The method name and signature is obtained from a method static context, which is accessed through the msc method argument. To determine when—relative to the desired point in program execution—the snippets should be executed, we use the @Before and @After annotations. The annotation parameters determine where to apply the snippets. The marker parameter selects the whole method body, and the scope parameter restricts the selection only to methods of the HelloWorld class.

To demonstrate the more advanced features of DiSL, Figure 3.3 shows a DiSL-based implementation of the instrumentation part of a field-immutability analysis tool, which identifies fields that were never written to outside the dynamic extent of the constructor [111]. This notion of immutability is dynamic by nature, and while it differs from the classic notion of immutability found in the literature [51, 52], it still provides a developer with valuable insights. The analysis (not shown) driven by the instrumentation tracks all field accesses and object allocations, and keeps a small state machine for each instance field. Every field can be in one of the three states: *virgin* (i.e., not read or not written to), *immutable* (i.e., read or was written to inside the dynamic extent of its owner object's constructor), or *mutable* (otherwise).

To implement the instrumentation for such an analysis in DiSL, we define two snippets, beforeFieldWrite and beforeFieldRead, which intercept the putfield and getfield bytecodes, respectively. Inside the snippets, we extract the reference to the instance

in which the field is being accessed from the operand stack, and pass it along with the field identifier and a queue of objects under construction to the analysis class, ImmutabilityAnalysis[2], using the onFieldWrite and onFieldRead methods, respectively.

To extract values from the operand stack (in this case the object reference), we use the DynamicContext API, which allows obtaining the values from arbitrary (valid) operand stack slots. The type of access to a field is determined by the bytecode instruction to which the snippets are bound, and the instruction in turn determines the operand stack layout we can expect when the corresponding snippet is executed. For field reads we therefore extract the object reference from the top of the operand stack, while for field writes we extract the reference from the second position from the top. The field identifier is obtained through a custom MethodStaticContext.

After each object allocation, we use the afterInitialization snippet to pass the newly allocated object, along with the identification of its allocation site, to the analysis runtime class using the onObjectInitialization method. As in the case of field accesses, the DynamicContext API is used to extract the reference to the newly allocated object from the operand stack.

The ThreadLocal static variable objectsUnderConstruction holds a stack of currently executing constructors, which the analysis uses to determine whether the owner of a field being accessed is under construction. To maintain the stack, the beforeConstructor snippet pushes the object under construction on the stack, whereas the afterConstructor snippet pops the stack. The ConstructorsOnly guard is used at instrumentation time to restrict the application of the two stack-maintenance snippets to constructors only.

3.3 Quantifying the Impact of DiSL

In this section we present the controlled experiment conducted to answer the first research question of this chapter. We first introduce the experiment design, including task and subject descriptions, and then present the results of the experiment followed by a discussion of threats to the validity of the study.

3.3.1 Experiment Design

The purpose of the experiment is to quantitatively evaluate the effectiveness of high-level framework for writing instrumentations for DPA tools compared to the use of a low-level bytecode manipulation library. We claim that using DiSL, developers of DPA tools can improve their productivity and the correctness of the resulting tools. In terms of hypothesis testing, we have formulated the following null hypotheses:

[2]For sake of brevity, we omit the description and the source code of the analysis runtime class, because it is not important in the context of instrumentation—it merely defines an API that the instrumentation will use to notify the analysis about events in the base program.

```
/** INSTRUMENTATION CLASS **/
public class DiSLClass {
  @ThreadLocal
  private static Deque <Object> objectsUnderConstruction;

  /** STACK MAINTENANCE **/
  @Before (marker = BodyMarker.class, guard = ConstructorsOnly.class)
  public static void beforeConstructor (DynamicContext dc) {
    try {
      if (objectsUnderConstruction == null) {
        objectsUnderConstruction = new ArrayDeque <Object> ();
      }

      objectsUnderConstruction.push (dc.getThis ());
    } catch (Throwable t) {
      t.printStackTrace ();
    }
  }

  @After (marker = BodyMarker.class, guard = ConstructorsOnly.class)
  public static void afterConstructor () {
    ImmutabilityAnalysis.instanceOf ().popStackIfNonNull (objectsUnderConstruction);
  }

  /** ALLOCATION SITE **/
  @AfterReturning (marker = BytecodeMarker.class, args = "new")
  public static void afterInitialization (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onObjectInitialization (
      dc.getStackValue (0, Object.class), // the allocated object
      sc.getAllocationSite () // the allocation site
    );
  }

  /** FIELD ACCESSES **/
  @Before (marker = BytecodeMarker.class, args = "putfield")
  public static void beforeFieldWrite (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onFieldWrite (
      dc.getStackValue (1, Object.class), // the accessed object
      sc.getFieldId (), // the field identifier
      objectsUnderConstruction // the stack of constructors
    );
  }

  @Before (marker = BytecodeMarker.class, args = "getfield")
  public static void beforeFieldRead (MyMethodStaticContext sc, DynamicContext dc) {
    ImmutabilityAnalysis.instanceOf ().onFieldRead (
      dc.getStackValue (0, Object.class), // the accessed object
      sc.getFieldId (), // the field identifier
      objectsUnderConstruction // the stack of constructors
    );
  }
}

/** GUARD CLASS **/
class ConstructorsOnly {
  @GuardMethod
  public static boolean isApplicable (MethodStaticContext msc) {
    return msc.thisMethodName ().equals ("<init>");
  }
}
```

Figure 3.3. Field-immutability analysis tool implemented in DiSL.

$H1_0$: Implementing DPA tools with DiSL does not reduce the development time of the tools.

$H2_0$: Implementing DPA tools with DiSL does not improve the correctness of the tools.

We therefore need to determine if there is evidence that would allow us to refute the two null hypotheses in favor of the corresponding alternative hypotheses:

H1: Implementing DPA tools with DiSL reduces the development time of the tools.

H2: Implementing DPA tools with DiSL improves the correctness of the tools.

The rationale behind the first alternative hypothesis is that DiSL provides high-level language constructs that enable users to rapidly specify compact instrumentations that are easy to write and to maintain. The second alternative hypothesis is motivated by the fact that DiSL does not require knowledge of low-level details of the JVM and bytecodes from the developer, although more advanced developers can extend DiSL for special use cases.

To test the hypotheses $H1_0$ and $H2_0$, we define a series of tasks in which the subjects, split between a control and an experimental group, have to implement different instrumentations similar to those commonly found in DPA tools. The subjects in the control group have to solve the tasks using only ASM, while the subjects in the experimental group have to use DiSL.

The choice of ASM as the tool for the control group was driven by several factors. The goal of the experiment was to quantify the impact of the abstractions and the programming model provided by high-level approach on the development of instrumentations for DPA tools. We did a thorough research of existing bytecode manipulation libraries and frameworks, and ASM came out as a clear winner with respect to flexibility and performance, both aspects crucial for development of efficient DPA tools. In addition, ASM is a mature, well-maintained library with an established community. As a result, ASM is often used for instrumentation development (and bytecode manipulation in general) both in academia and industry. We maintain that when a developer is asked to instrument an application by manipulating Java bytecode, ASM will most probably be the library of choice.

DiSL was developed as an abstraction layer on top of ASM precisely because of the above reasons, but with a completely different programming model inspired by AOP, tailored for instrumentation development. Using ASM as the baseline allowed us to *quantify* the impact of the abstraction layer and programming model on the instrumentation development process, compared to a lower-level, but commonly used programming model provided by ASM as the de-facto standard library.

Task	Description
0	a) On method entry, print the method name. b) Count the number of NEW bytecodes in the method. c) On each basic block entry, print its index in the method. d) Before each lock acquisition, invoke a given method that receives the object to be locked as its argument.
1	On method entry, print the number of method arguments.
2	Before array allocation, invoke a given method that receives the array length as its argument.
3	Upon method completion, invoke a given method that receives the dynamic execution count of a particular bytecode instruction as its argument.
4	Before each AASTORE bytecode, invoke a given method that receives the object to be stored in the array together with the corresponding array index as its arguments.
5	On each INVOKEVIRTUAL bytecode, invoke a given method that takes only the receiver of the invoke bytecode as its argument.
6	On each non-static field write access, invoke a given method that receives the object whose field is written to, and the value of the field as its arguments. Invocation shall be made only when writing non-null reference values.

Table 3.1. Description of instrumentation tasks

3.3.2 Task Design

With respect to the instrumentation tasks to be solved during the experiment, we maintain two important criteria: the tasks shall be representative of instrumentations that are used in real-world applications, and they should not be biased towards either ASM or DiSL. Table 3.1 provides descriptions of the instrumentation tasks the subjects have to implement. Those are examples of typical instrumentations that are used in profiling, testing, reverse engineering, and debugging.

To familiarize themselves with all the concepts needed for solving the tasks, the subjects first had to complete a bootstrap task 0.

3.3.3 Subjects and Experimental Procedure

In total, we had 16 subjects—BSc., MSc., and PhD students from Shanghai Jiao Tong University—participate in the experiment on a voluntary basis. Prior to the experiment, all subjects were asked to complete a self-assessment questionnaire regarding their expertise in object-oriented programming (OOP), Java, Eclipse, DPA, Java bytecode, ASM, and AOP. The subjects rated themselves on a scale from 0 (no experience) to 4 (expert), and on average achieved a level of 2.6 for OOP, 2.5 for Java, 2.5 for Eclipse, 0.75 for DPA, 0.6 for JVM bytecode, 0 for ASM, and 0.12 for AOP. Our subjects can be thus considered average (from knowledgeable to advanced) Java developers who had experience with Eclipse and with writing very simple instrumentation tasks, but

with little knowledge of DPA and JVM in general, and with no expertise in ASM and AOP. Based on the self-assessment results, the subjects were assigned to the control and experimental groups so as to maintain approximately equal distribution of expertise.

The subjects in both groups were given a thorough tutorial on DPA, JVM internals, and ASM. The ASM tutorial focused on the tree API, which is considered easier to understand and use. In addition, the subjects in the experimental group were given a tutorial on DiSL. Since the DiSL programming model is conceptually closer to AOP and significantly differs from the programming model provided by low-level bytecode manipulation libraries, including ASM, we saw no benefit in giving the tutorial on DiSL also to the subjects in the control group. The tutorial was based on the experience of the author of this dissertation with dynamic program analysis and was given in form of an informal 3-hour lecture. The subjects were free to ask clarification questions. The experiment was performed in a single session in order to minimize the experimental bias (e.g., by giving different tutorials on the same topic) that could affect the experimental results. The session was supervised, allowing the subjects to ask clarification questions and preventing them from cheating. The subjects were not familiar with the goal of the experiment and the hypotheses.

We provided the subjects with disk images for VirtualBox,[3] which was the only piece of software that had to be installed. Each disk image contained all the software necessary to complete the tasks: Eclipse IDE, Apache Ant, and ASM installed on an Ubuntu 10.4 operating system. In addition, the disk images for the experimental group also contained an installation of DiSL. All subjects received the task descriptions and a debriefing questionnaire, which required the subjects to rate the perceived time pressure and task difficulty. The tasks had to be completed in 180 minutes, giving a 30 minutes time slot for each task.

3.3.4 Variables and Analysis

The only independent variable in our experiment is the availability of DiSL during the tasks.

The first dependent variable is the time the subjects spend to implement the instrumentations, measured by having the subjects write down the current time when starting and finishing a task. Since the session is supervised and going back to the previous task is not allowed, there is no chance for the subjects to cheat.

The second dependent variable is the correctness of the implemented solutions. This is assessed by code reviews and by manual verification. A fully correct solution is awarded 4 points, no solution 0 points, partially correct solutions can get from 1 to 3 points.

For hypothesis testing, we use the parametric one-tailed Student's t-test, after validating the assumptions of normality and equal variance using the Kolmogorov-

[3] `http://www.virtualbox.org/`

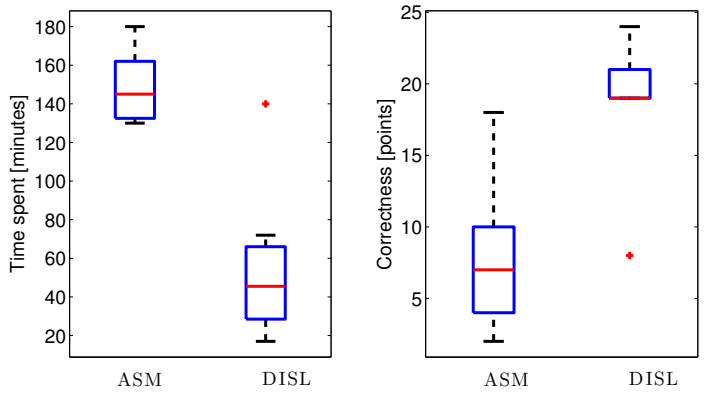

Figure 3.4. Box plots for development time spent (left) and correctness of the tools (right). The red dot represents an outlier.

Smirnov and Levene's tests. We maintain the typical confidence level of 99% ($\alpha = 0.01$) for all tests. All calculations were made using the SPSS statistical package.

3.3.5 Experimental Results

Development Time

On average, the DiSL group spent 63% less time writing the instrumentations. The time spent by the two groups to complete the tasks is visualized as a box plot in Figure 3.4.

To assess whether the positive impact on the development time observed with DiSL has a statistical significance, we test the null hypothesis $H1_0$, which says that DiSL does not reduce the development time. Neither Kolmogorov-Smirnov nor Levene's tests indicate a violation of the Student's t-test assumptions. The application of the latter gives a p-value[4] of 0.001, which is one order of magnitude less than $\alpha = 0.01$ (see Table 3.2). We therefore reject the null hypothesis in favor of the alternative hypothesis, which means that the time spent is statistically significantly reduced by the availability of DiSL.

We attribute the substantial time difference to the fact that instrumentations written using ASM are very verbose and low-level, whereas DiSL allows one to write instrumentations at a higher level of abstraction.

[4]In statistical testing, the p-value is the probability of obtaining a result at least as extreme as the one that was observed, provided that the null hypothesis is true. Generally, if the p-value is less than the chosen significance level (e.g., 0.01), then obtaining such a result with the null hypothesis being true is highly unlikely, and the null hypothesis is therefore rejected.

	Time [minutes]		Correctness [points]	
	ASM	DiSL	ASM	DiSL
Summary statistics				
mean	148.62	54.62	8.75	18.75
difference	-63.2%		+46.6%	
min	130	17	2	8
max	180	140	18	24
median	145	45.5	7	19
stdev.	18.73	38.96	5.92	4.68
Assumption checks				
Kolmogorov-Smirnov Z	0.267	0.203	0.241	0.396
Levene F	1.291		1.939	
One-tailed Student's t-test				
df	14		14	
t	6.150		-3.746	
p-value	<0.001		0.002	

Table 3.2. Descriptive statistics of the experimental results

Instrumentation Correctness

As reported in Table 3.2, the average amount of points scored by a subject in the experimental (DiSL) group is 46.6% higher than in the control (ASM) group. A box plot for the results is shown in Figure 3.4.

To test the null hypothesis $H2_0$, which says that there is no impact of using DiSL on instrumentation correctness, we again apply the Student's t-test, since neither Kolmogorov-Smirnov nor Levene's tests indicate a violation of the normality assumptions. The t-test gives a p-value of 0.002 (see Table 3.2) which is again an order of magnitude less than $\alpha = 0.01$. We therefore reject the null hypothesis in favor of the alternative hypothesis $H2$, which means that the correctness of instrumentations is statistically significantly improved by the availability of DiSL.

3.3.6 Threats to Validity

Internal Validity

There is a chance that the subjects participating in the experiment may not have been competent enough. To minimize the impact of this uncertainty, we ensured that the subjects had expertise at least at the level of average Java developers by using a preliminary self-assessment questionnaire. Moreover, we ensured that the subjects were equally distributed among the control group and the experimental group according to their expertise. Also both groups were given a thorough tutorial on DPA and had to complete task 0 before starting to solve the evaluated tasks.

The instrumentation tasks were designed by the author of this dissertation (who

is also a contributor to the DiSL framework), and therefore could be biased towards DiSL. To avoid this threat, we created instrumentations that are representative and used in some real-world scenarios. We asked other faculty members, who are not involved in the DiSL project but are familiar with DPA, to provide their comments on the tasks. Additionally, the tasks may have been too difficult and the time slot of 30 minutes may have been insufficient. To address this threat, we conducted a pilot study at Charles University in Prague and collected feedback about the perceived task difficulty and time pressure, which allowed us to adjust the difficulty of the instrumentation tasks. Moreover, the pilot study allowed us to adjust the tutorial and refine the documentation of DiSL.

External Validity

One can question the generalization of our results given the limited representativeness of the subjects and tasks. Even though the literature [40] suggests to avoid using only students in a controlled experiment, we could not attract other subjects to participate in our experiment.

Another threat to external validity is the fact that we compare high-level (DiSL) and low-level (ASM) approaches for instrumentation development. This choice is a necessary consequence of considering DPA tools to be the primary targets for DiSL-based instrumentations—high-level bytecode manipulation frameworks typically have limitations with respect to flexibility of instrumentation and control over inserted code, both of which are crucial for development of efficient DPA tools.

While low-level libraries can be typically used for a wide range of tasks, it is the focus on a specific purpose that allows high-level libraries to hide the low-level details common to that particular purpose. In this sense, DiSL was specifically designed for instrumentation development, while other high-level frameworks often target general code manipulation and transformation tasks. Our study quantifies the impact of introducing a high-level, AOP-inspired API on the developer productivity compared to the common practice. An additional user study involving DiSL and other high-level bytecode manipulation frameworks could explore the suitability of various high-level interfaces for instrumentation development, but that would not invalidate our study.

Results Summary and Future Work

In summary, the experiment confirms that DiSL improves developer productivity compared to ASM, the library of choice for bytecode manipulation in DPA tools. In terms of development time and instrumentation correctness, the improvement is both practically and statistically significant.

However, our study can be possibly improved in several ways. The controlled experiment presented in this chapter has a “between” subject design. Conducting a similar study with a “with-in” subject design, where subjects from the control and

the experimental groups swap the tasks after the first experiment might provide new insights and strengthen the results. Moreover, comparing DiSL with other high-level approaches for performing instrumentations (e.g., AspectJ) would be an interesting continuation of this work.

3.4 DiSL Tools Are Concise and Efficient

In this section we present another experiment—an extensive evaluation of high-level approach in the context of 10 existing open-source DPA tools. We have identified and recasted the instrumentation part of each tool in DiSL, without touching the analysis part. We then compared the amount of code required to implement the DiSL-based instrumentation, as well as its performance, to the original instrumentation. In the following text, we will refer to the unmodified version of a tool as *original*, and to the version using an equivalent DiSL-based instrumentation as *recasted*.

3.4.1 Overview of Recasted Analysis Tools

To establish a common context for both parts of the evaluation, we first present a short overview of each tool. To improve clarity, all descriptions adhere to a common template: we start with a high-level overview of the tool, then we describe the instrumentation logic used to trigger the analysis actions, and finally we point out the DiSL features used to reimplement the instrumentation.

The original instrumentations were implemented mostly using ASM, or AspectJ, with C used in one case. Most of the ASM-based and AOP-based tools rely on a Java agent, which is part of the java.lang.instrument API, and perform load-time instrumentation of all (loaded) classes.

Cobertura[5] is a tool for Java code coverage analysis. At runtime, Cobertura collects coverage information for every line of source code and for every branch.

Cobertura uses an ASM-based instrumentation to intercept branch instructions and blocks of bytecode corresponding to lines of code, and to invoke the method corresponding to the intercepted event on the analysis class to update the coverage information.

The recasted instrumentation uses three custom markers to capture the same join points as Cobertura, and a synthetic local variable to indicate whether a branch occurred.

EMMA[6] is a tool for Java code coverage analysis. During instrumentation, EMMA analyzes the classes and collects various static information, including the number of methods in a class, and the number of basic blocks in a method. At runtime, EMMA collects coverage information for every basic block of every method in every class.

[5] `http://cobertura.sourceforge.net/`

[6] `http://emma.sourceforge.net`

EMMA uses ASM for both static analysis and instrumentation to intercept every method entry, where it associates a two-dimensional array with each class, and every basic block exit, where it updates the array to mark a basic block visited.

The recasted instrumentation uses the DiSL method body marker to intercept method entry, where it registers the two-dimensional array with EMMA, and a basic block marker to intercept basic block entry, where it triggers the update of coverage information. The array is stored in a per-class synthetic static field, which can be shared among snippets executed in different methods. A guard is used to filter out interface classes.

HPROF [87] is a heap and CPU profiler for Java distributed with the HotSpot JVM. Since HPROF is a native JVM agent implemented in C, we have reimplemented one of its features in Java to enable comparison with a DiSL-based tool. Our tool only provides the heap allocation statistics feature of HPROF, and uses a DiSL-based instrumentation to collect data. We therefore use $HPROF^*$ as a designation for "HPROF with only the heap allocation statistics feature" in the following text, and all comparisons against the original HPROF only concern that single feature.

To keep track of allocated objects, the $HPROF^*$ agent uses the Java Virtual Machine Tool Interface (JVMTI) [89] to intercept object allocation and death events, and collects type, size, and allocation site for each object.

The recasted instrumentation uses the DiSL bytecode marker to intercept object and array allocations, and a dynamic context API to obtain the references to newly allocated objects from the operand stack.

JCarder[7] is a tool for finding potential deadlocks in multi-threaded Java applications. At runtime, JCarder constructs a dependency graph for threads and locks, and if the graph contains a cycle, JCarder reports a potential deadlock.

To maintain the dependency graph, JCarder uses an ASM-based instrumentation to intercept acquisition and release of locks. To simplify the instrumentation, synchronized methods are converted to normal methods with explicit locking.

The recasted instrumentation uses the DiSL bytecode marker to intercept the lock acquisition/release bytecodes, and a method body marker with a guard to intercept synchronized method entry and exit. A custom static context is used to precompute the static method description required by the analysis class, and the dynamic context API is used to extract the lock reference from the stack.

JP2 [107] is a calling-context profiler for Java. For each method, JP2 collects various static metrics (i.e., method names, number and sizes of basic blocks) and dynamic metrics (i.e., method invocations, basic block executions, and number of executed bytecodes), and associates them with a corresponding node in a calling-context tree (CCT) [2], grouped by the position of the method call-site in the caller.

JP2 uses an ASM-based instrumentation to intercept method entry and exit, basic block entry, and execution of bytecodes that may trigger method invocation or execu-

[7] `http://www.jcarder.org/`

tion of class initializers upon loading a class. Static information is collected during instrumentation.

The recasted instrumentation uses default DiSL markers (method body, basic block, and bytecode) to update the CCT upon method entry and exit. Thread-local variables are used to access the CCT instance and call-site position in the bytecode, while synthetic local variables are used to cache and share CCT nodes and call-site position between snippets. Static information is collected at instrumentation time using the method body, basic block, and bytecode static contexts.

JRat[8] is a call graph profiler for Java. For each method, JRat collects the execution time of each invocation, grouped by the caller. The data is used to produce a call graph with execution time statistics, which allows to attribute the time spent in a particular method to individual callers.

To measure method execution time and to determine the caller, JRat uses an ASM-based instrumentation to intercept each method entry and exit.

The recasted instrumentation uses the DiSL method body marker to intercept method invocations, a synthetic local variable to share time stamps between snippets, and a per-method synthetic static field to store the instance of an analysis class. A guard is used to avoid instrumentation of constructors and static class initializers.

RacerAJ [18] is a tool for finding potential data races in multi-threaded Java applications. At runtime, RacerAJ monitors all field accesses and lock acquisitions/releases, and reports a potential data race when a field is accessed from multiple threads without holding a lock that synchronizes the accesses.

To maintain various per-thread and per-field data structures, RacerAJ uses an AOP-based instrumentation to intercept all field accesses, and all lock acquisitions/releases, both explicit and implicit due to synchronized methods.

The recasted instrumentation uses the DiSL bytecode marker to intercept the lock acquisition/release and field access bytecodes, and a method body marker together with a guard to intercept synchronized method entry and exit. A custom static context is used to obtain a field identifier and the name of the class owning the field that is being accessed. A class static context is used to identify a field access site, while the dynamic context API is used to extract the lock reference from the stack.

ReCrash [6] is a tool for reproducing software failures. During program execution, ReCrash snapshots method arguments leading to failures and uses them to generate unit tests that reproduce the failure.

ReCrash uses an ASM-based instrumentation to intercept method entry, where it snapshots the method arguments, normal method exit, where it discards the snapshot, and abnormal method exit (only in the main method), where it uses the snapshot to generate a test case.

The recasted instrumentation uses the DiSL method body marker to intercept method entry and normal/abnormal method exit. A per-method synthetic static field

[8] `http://jrat.sourceforge.net/`

is used to cache method static information, while a synthetic local variable is used to share the index of the argument snapshot between snippets on method entry and exit. The dynamic context API is used to take a snapshot of method arguments, and a guard filters out private methods, methods without arguments, empty methods, static class initializers, and constructors.

Senseo [104] is a tool for profiling and code comprehension. For each method invocation, Senseo collects calling-context specific information, which a plugin[9] then makes available to the user via enriched code views in the Eclipse IDE.

Senseo uses an AspectJ-based instrumentation, and intercepts each method entry and exit to count method invocations and uses join point API to collect statistics on method arguments and return types. Within methods, it also intercepts object allocations to count the number of allocated objects.

The recasted instrumentation uses the DiSL method body marker to intercept method invocation, and a bytecode marker to intercept object allocations. Method arguments and newly allocated objects are accessed via the dynamic context API. Guards are used to differentiate between methods that only accept and return primitive types and methods that work with reference types.

TamiFlex [20] is a tool that helps other (static analysis) tools deal with reflection and dynamically generated classes in Java. TamiFlex logs all reflective method calls and dumps all loaded classes, including those that are dynamically generated, to the disk. The collected information can be used to perform static analysis either with a TamiFlex-aware tool or, after transforming all reflective calls to actual method calls, with any other static analysis tool.

TamiFlex uses an ASM-based instrumentation in its *Play-out* agent to intercept method invocations on the instances of the Class, Constructor, and Method reflection classes.

The recasted instrumentation of the *Play-out* agent uses the DiSL method body marker restricted by a scope to intercept exits from methods in the aforementioned reflection classes. A corresponding snippet is used for each transformation from the *Play-out* agent. Support for dumping both the original and instrumented classes is present in DiSL and has been used.

3.4.2 Instrumentation Conciseness Evaluation

Based on the experience with recasting the instrumentation parts of the tools in this evaluation, we usually expect the instrumentations implemented using DiSL to require less logical source lines of code (SLOC) than their ASM-based equivalents. Even though "less code" does not generally mean "better code", we assume that in the same context and for the same task, a shorter implementation can be considered more concise, and thus easier to write, understand, and maintain, if it also enables increased developer

[9] `http://scg.unibe.ch/research/senseo`

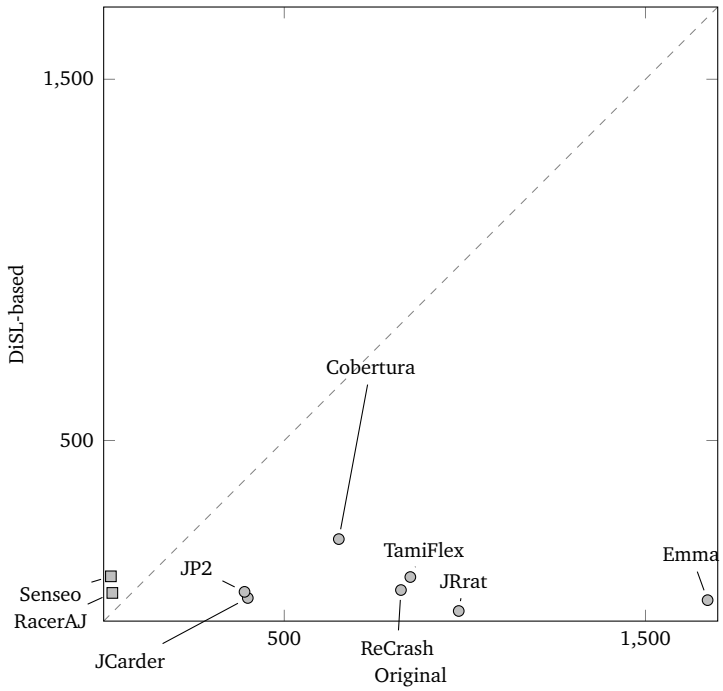

Figure 3.5. Logical source lines of code (SLOC) counts for original and DiSL-based instrumentations.
We use ○ to denote instrumentations originally based on ASM, and □ to denote those based on AspectJ.

productivity.

Since the study in Section 3.3 shows that DiSL indeed positively impacts productivity compared to ASM, we use the SLOC count as a metric to compare different implementations of equivalent instrumentations in DPA tools. As a result, we provide quantitative evidence that DiSL-based instrumentations are more concise than their ASM-based equivalents, but not as concise as the AOP-based variants.

The plot in Figure 3.5 shows the SLOC counts[10] of both the DiSL-based and the original instrumentations for each tool. Each data point in the plot corresponds to a single tool, with the SLOC count of the original instrumentation on the x-axis, and the SLOC count of the DiSL-based instrumentation on the y-axis.

Figure 3.5 indicates that DiSL-based instrumentations generally require less code than their ASM-based counterparts, because bytecode manipulation, even when using ASM, results in more verbose code. The extreme savings in the case of EMMA are due to EMMA having to implement its own static analysis, whereas the DiSL-based instrumentation can use the static context information provided by DiSL.

[10]Calculated using Unified CodeCount by CSSE USC, rel. 2011.10, `http://sunset.usc.edu/research/CODECOUNT`.

In line with our expectations, the DiSL-based instrumentations require more code than their AOP-based equivalents. This can be partially attributed to DiSL being an embedded language hosted in Java, whereas AOP has the advantage of being a separate language. Moreover, DiSL instrumentations also include code that is evaluated at instrumentation time, which increases the code size, but provides significant performance benefits at runtime. However, in the context of instrumentations for DPA, DiSL is more flexible and expressive than AOP without impairing the performance of the resulting tools.

The results for HPROF were intentionally omitted from the plot, because we were unable to isolate the instrumentation code for $HPROF^*$ from the rest of the application. In total, HPROF consists of more than 9000 SLOC written in C, whereas our version of $HPROF^*$ written in Java consists of 168 SLOC, of which 39 is the DiSL-based instrumentation.

3.4.3 Instrumentation Performance Evaluation

In this section, we conduct a series of experiments to provide answer to RQ2, i.e., whether DiSL-based instrumentations perform as fast as the equivalent instrumentations written using low-level bytecode manipulation libraries.

To evaluate the instrumentation performance, we compare the execution time of the original and the recasted tools on benchmarks from the DaCapo [17] suite (release 9.12). Of the fourteen benchmarks present in the suite, we excluded tradesoap, tradebeans and tomcat due to well known issues[11] unrelated to DiSL. All experiments were run on a multicore platform[12] with all non-essential system services disabled.

We present results for startup and steady-state performance in Figure 3.6 and Figure 3.7, respectively. Both figures contain a separate plot for each of the evaluated tools, displaying the overhead factor of a particular tool during the execution of each individual DaCapo benchmark. The data points marked in gray represent the execution of a single DaCapo benchmark, with the overhead factor of the original and the recasted tool on the x-axis and the y-axis, respectively. The single black data point in each plot represents the geometric mean of overhead factors from all benchmarks. The diagonal line serves to indicate the data points for which the overhead factor of the original and the recasted tool is the same. To determine the startup overhead, we executed 3 runs of a single iteration of each benchmark and measured the time from the start of the process till the end of the iteration to capture the instrumentation overhead. We relied on the filesystem cache to mitigate the influence of I/O operations during startup. To determine the steady-state overhead, we made a single run with 10 iterations of

[11] See bug ID 2955469 (hardcoded timeout in tradesoap and tradebeans) and bug ID 2934521 (StackOverflowError in tomcat) in the DaCapo bug tracker at `http://sourceforge.net/tracker/?group_id=172498&atid=861957`.

[12] Four quad-core Intel Xeon CPUs E7340, 2.4 GHz, 16 GB RAM, Ubuntu GNU/Linux 11.04 64-bit with kernel 2.6.38, Oracle Java HotSpot 64-bit Server VM 1.6.0_29.

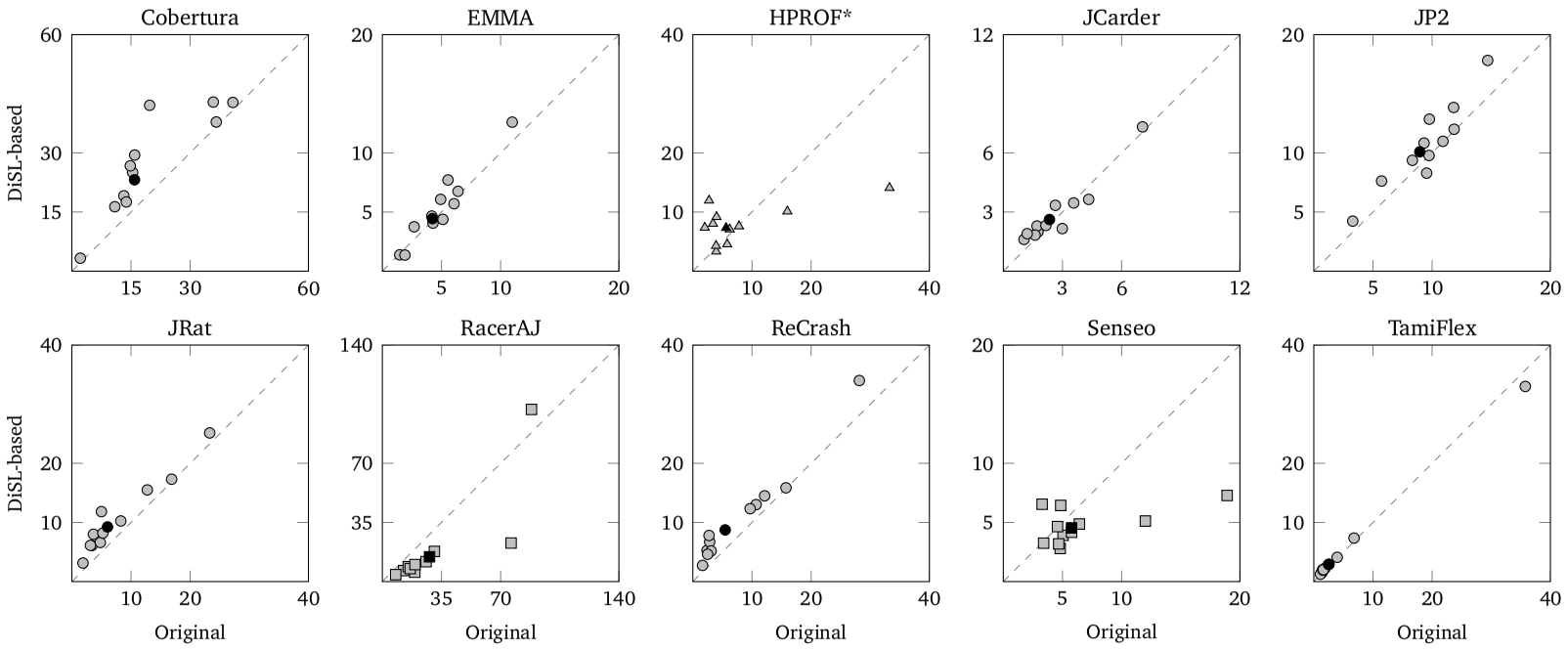

Figure 3.6. Startup phase – Overhead factor of original and DiSL-based applications compared to the baseline. Data points refer to a selection of 11 DaCapo 9.12 benchmarks and to the geometric mean of the overhead factors. We use ○ to indicate applications originally based on ASM, □ for those based on AspectJ, and △ for those based on C.

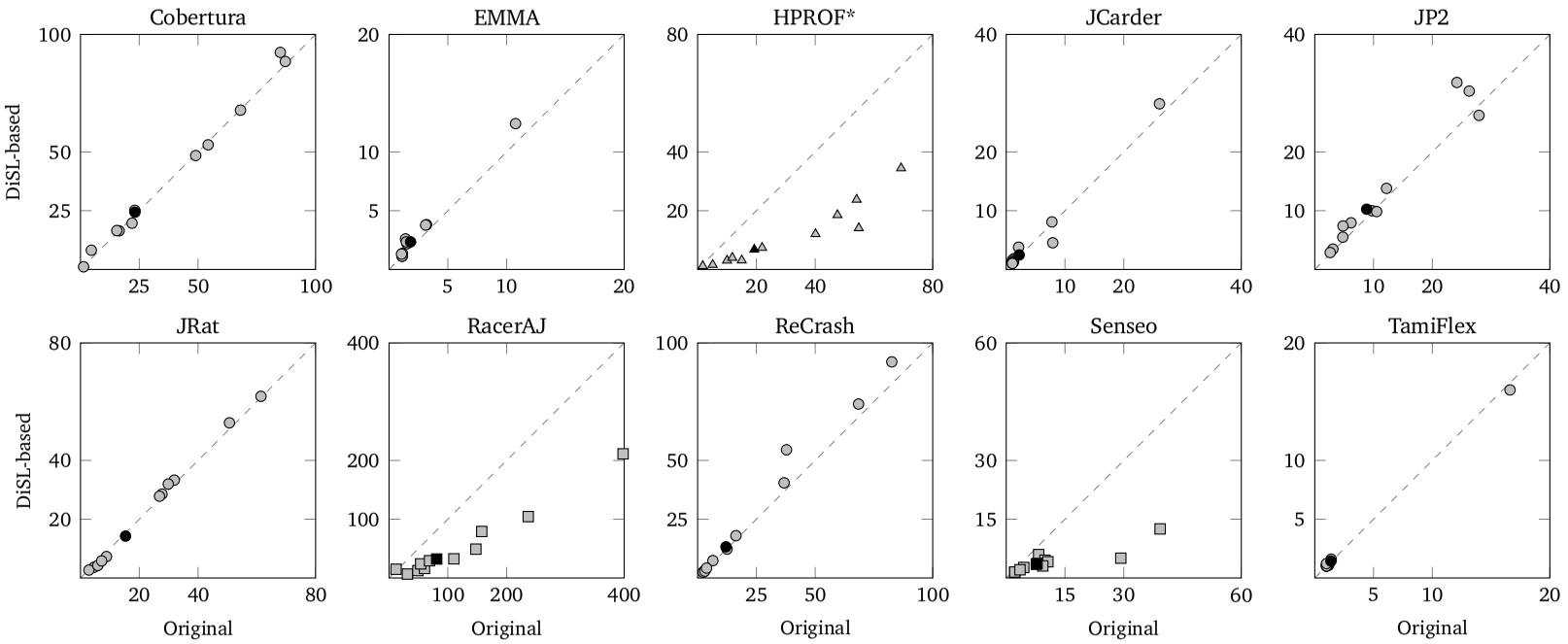

Figure 3.7. Steady-state phase – Overhead factor of original and DiSL-based applications compared to the baseline. Data points refer to a selection of 11 DaCapo 9.12 benchmarks and to the geometric mean of the overhead factors. We use ○ to indicate applications originally based on ASM, □ for those based on AspectJ, and △ for those based on C.

each benchmark and excluded the first 5 iterations to minimize the influence of startup transients and interpreted code. The number of iterations to exclude was determined by visual inspection of the data from the benchmarks.

Concerning the startup overhead, the results in Figure 3.6 indicate that DiSL often slows down the startup phase of a benchmark. This can be partially attributed to DiSL using ASM to first create a tree representation of every class and only applying the exclusion filters at the level of methods. In this case, DiSL could be improved to decide early whether a class needs to be instrumented at all and thus avoid processing classes that need not be touched. Another source of overhead is evaluating guards and computing static context information for snippets that require it. In this case, the higher overhead at instrumentation time is traded for a lower overhead at runtime, as discussed below. The startup phase overhead is only important for applications where the amount of class loading relative to other code execution is high. We believe these cases to be rare.

Concerning the steady-state overhead, the results in Figure 3.7 show that the recasted tools are typically about as fast as their original counterparts, sometimes much faster, but never much slower. Performance improvements can be observed in the case of AOP-based tools (RacerAJ and Senseo both use AspectJ for instrumentation), and in the case of HPROF. The improved performance can be attributed mainly to the fact that DiSL allows to use static information at instrumentation time to precisely control where to insert snippet code, hence avoiding costly checks and static information computation (often comprising string concatenations) at runtime. The need for runtime checks is extremely pronounced with HPROF, which needs to filter the events related to program execution emitted by the JVM. Additional performance gains can be attributed to the ability of DiSL snippets to efficiently access the constant pool and the JVM operand stack, which is particularly relevant in comparisons with AOP-based tools.

3.5 Summary

In this chapter we aimed in identifying an instrumentation framework that allows rapid development of dynamic program analysis tools, as we needed one to base our tools on. In this regard we performed a thorough evaluation and assessment of DiSL [80, 136], a new abstraction layer on top of the well-known bytecode manipulation library ASM [91]. DiSL is a domain-specific aspect language especially designed for instrumentation-based dynamic program analysis. The design of DiSL aims at reconciling (i) a convenient high-level programming model to reduce tool development time, (ii) high expressiveness to enable the implementation of any instrumentation-based dynamic analysis tool, and (iii) efficiency of the generated code to ensure good tool performance.

First, we conducted a controlled experiment to compare DiSL with ASM, measuring development time and correctness of the developed tools for 6 common dynamic analysis tasks. We showed that the use of DiSL reduced development time and improved

tool correctness with both practical and statistical significance. Second, we recasted 10 open-source software engineering tools with DiSL, showing quantitative evidence that DiSL-based tools are (i) considerably more concise than equivalent tools based on ASM and (ii) only slightly more verbose than equivalent tools implemented in AspectJ. Regarding performance, DiSL-based tools incur higher startup overhead than ASM-based tools but yield comparable steady-state performance; DiSL-based tools significantly outperform tools implemented in AspectJ, both in terms of startup and steady-state performance. We conclude that DiSL is a valuable abstraction layer on top of ASM which indeed succeeds in boosting the productivity of tool developers. In contrast to AspectJ, DiSL neither limits expressiveness nor impairs performance of the resulting tools.

Based on the results of these experiments, we chose DiSL as an instrumentation framework for developing our workload characterization toolchain. Details on the toolchain will follow in the Chapter 4.

Chapter 4

Toolchain for Workload Characterization

4.1 Dynamic Metrics

Similar to performance evaluation (benchmarking), workload characterization employs benchmarks (representing samples from the domain of applications) to induce workload on the observed system while collecting metrics that characterize different aspects of the behavior of the system. However, unlike benchmarking, which aims to determine *how well* a system performs at different tasks, workload characterization aims to determine *in what way* these tasks differ from each other, providing essential guidance, e.g., for optimization effort. Ideally, the metrics characterizing JVM workloads should capture the differences between Java and non-Java workloads and—when correlated with JVM performance on a particular workload—they should provide developers of both JVM languages and the JVM itself with useful insights.

For example, a developer might hypothesize that a workload performed poorly because of heap pressure generated by increased usage of boxed primitive values, which are used relatively rarely in normal Java code, but frequently in some other JVM languages such as in JRuby. JVM language developers could optimize their bytecode generator, for example, to try harder at using primitives in their unboxed form. A dynamic metric capturing the boxing behavior of a particular workload would allow these developers to quantify the effects of such optimizations. Meanwhile, JVM developers may benefit from the metrics in a different way. Because JVM optimizations are dynamic and adaptive, each optimization decision is guarded by a heuristic decision procedure applied to profile data collected at runtime. For example, the decision whether to inline a callee into a fast path depends on factors such as the hotness of that call site (evaluated by dynamic profiling) and the size of the callee. JVMs can therefore benefit from better heuristics which more accurately match real workloads, including non-Java workloads.

For maximum benefit, there must be an easy way for developers to compute these metrics over workloads of their choosing. However, no existing work has defined a comprehensive set of such metrics and provided the tools to compute them. Rather, existing approaches are fragmented across different infrastructures: many lack portability due to using a modified version of the JVM [38, 74], while others collect only architecture-dependent metrics [114]. In addition, at least one well-known metric suite implementation [41] runs with unnecessarily high performance overhead. Ideally, metrics should be collected within reasonable time, since this enables the use of complex, real-world workloads and shortens the development cycles. Metrics should also be computed based on observation of the whole workload, which not all infrastructures allow. For example, existing metrics collected using AspectJ are suboptimal since they lack coverage of code from the Java class library [19, 27, 94].

Our approach bases all metrics on a unified infrastructure which is JVM-portable, offers non-prohibitive runtime overhead with near-complete bytecode coverage, and can compute a full suite of metrics "out of the box". All the metrics are *dynamic*, meaning that they can be evaluated only by running a program with some input. The significance of dynamic metrics—in contrast to static metrics such as code size, or instruction distribution—has been motivated elsewhere by Dufour et al. [41], who defined a list of sixty metrics considered useful for guiding optimization of Java programs. Our infrastructure can compute all of these metrics.

However, the workloads produced by the various JVM languages exhibit properties that vary significantly between Java and non-Java workloads, and require additional metrics for proper characterization. In this section we present such metrics, summarized in Table 4.1. Like those of Dufour et al., the new metrics are defined at the bytecode level, making them JVM-independent and allowing portable implementation. We believe that the new metrics are *comprehensive* with respect to the current selection of JVM languages, in that they cover the differences arising from these languages' distinct semantics. In turn, these differences imply that different optimizations will be required on the part of JVM and language (front-end) developers. We have therefore grouped the metrics according to the language-level concerns which motivate them: *object access*, *object allocation*, and *code generation*. We now review each group in turn.

4.1.1 Object access concerns

Object access concerns affect optimizations related to sharing of objects among threads. Accessing shared objects requires locking, unless immutable data structures are used. Our metrics thefore focus on identifying effectively immutable objects, the kind of locks used, and the kind of access to shared objects.

Immutability. In recent years, functional languages have gained much attention. In general, functional programs tend to use immutable data structures to avoid side effects, which makes such programs amenable to parallelization. Finding objects

Metric family	Description of metrics
Argument passing	distribution of floating point arguments over all dynamic invocations (see text) distribution of reference arguments over all dynamic invocations (see text)
Basic block hotness	contribution of the top 20% of basic blocks to the dynamic total number of basic block executions
Call-site polymorphism	distribution of target method count over all dynamically-dispatched calls number of dynamically-dispatched call sites targeting a given number of methods number call sites using each of the four invoke instructions number of calls made using each of the four invoke instructions.
Instruction mix	execution counts for each distinct bytecode instruction (opcode)
Method hotness	contribution of the top 20% of methods to the dynamic total number of method executions
Stack usage and recursion depth	distribution of stack heights upon recursive calls
Use of boxed types	number of boxed primitives allocated number of boxed primitives requested (using valueOf; see text)
Field sharing	number of objects partially read-shared between different threads number of objects partially write-shared between different threads number of objects fully read-shared between different threads number of objects fully write-shared between different threads
Field synchronization	number of objects synchronized on the average number of locking operations per object the maximum nesting depth reached per lock
Field immutability	number of fields immutable, counted once per containing object number of fields immutable (per class) number of objects immutable (all fields immutable). number of classes immutable (all fields immutable for all objects)
Implicit zeroing	number of primitive fields unnecessarily zeroed number of reference fields unnecessarily zeroed
Use of identity hashcodes	execution counts of overridden hashCode methods. execution counts of System.identityHashCode methods. execution counts of default Object.hashCode method.
Object churn distance	distribution of object churn distances (see text)
Object lifetimes	distribution of object survival times (see text)
Object sizes	distribution of object sizes (see text)

Table 4.1. Metrics that can be computed by our toolchain.

that are effectively immutable can help a developer to identify code locations where using immutable types could simplify parallelization. In addition, popular compiler optimization techniques benefit from immutable objects and data structures [96]. One example of such an optimization is load elimination, which replaces repeated memory accesses to the immutable objects with access to a compiler-generated temporary (likely to be stored in a register). However, this optimization is defeated in the presence of method calls or synchronization. Immutable objects avoid this problem, since they are known not to change across method calls.

To characterize immutability, we define four metrics, distinguishing between class and object immutability: number of instance fields that are per-object immutable, number of objects that are immutable (i.e., all fields immutable), number of fields that are immutable in all allocated objects of the defining class, and number of classes for

which all allocated instances are immutable.[1]

Lock usage and sharing. Since locking operations come at a cost, researchers have developed thin locks [9] and biased locks [98] to minimize the runtime overhead and memory cost. Thin locks are used in the situation where most locks are never subject to contention by multiple threads. Moreover, if most of the locks are only acquired by a single thread, biased locks are used. To apply synchronization optimizations one has to identify the common-case nature of locking operations in the application. We count the number of objects synchronized on, and the average number of locking operations per object, and the maximum nesting depth reached per (recursive) lock.

Unnecessary synchronization. Immutability and sharing analyses can be used in combination to aid in removal of unnecessary synchronization [21]. Ordinarily, objects shared among different threads potentially require some synchronization. However, the synchronization is redundant if we find that the object is immutable. The metrics capture the number of objects shared between different threads, with separate counts for read-only sharing (two or more readers; exactly one writer, i.e. the allocating thread) and write-sharing (two or more writers; any number of readers). As in the case of the immutability analysis, we distinguish between fully and partially shared objects, yielding four distinct metrics in total.

4.1.2 Object allocation concerns

In a managed runtime, developers rely on a garbage collector (GC) to reclaim unused memory. While such a programming model greatly simplifies development, abusing it may result in undesirable pressure on the GC, causing significant performance degradation. This is of paramount importance for developers of JVM language compilers, because their decisions regarding minute details in the implementation of various language constructs may greatly influence the character of the workload imposed on the JVM.

Use of boxed types. Different languages make differing use of boxed primitives. For example, all primitive values in JRuby are boxed. However, boxing is expensive because it creates additional heap pressure and can defeat optimization passes usually applied to stack- and register-allocated primitive values. Different optimization techniques can be used to reduce performance overhead incurred by boxed values. Therefore, a metric characterizing the extent of boxing in the workload is very useful. Our two metrics here are the counts of boxed primitives allocated and boxed primitives requested (by calls to `valueOf()` methods on `Integer`, `Byte` and so on).

Object churn. Creation of many temporary objects (i.e., object churn), which may

[1] Many of our metrics are collected as raw numbers, but could be more usefully represented as fractions. Although we do not state this explicitly from hereon, in all such cases the relevant total is available for use as a divisor. As such, both fractions and raw numbers are available.

or not be boxed types, is detrimental to performance, since it comes at a cost of very frequent garbage collection and inhibits parallelization if temporary objects require synchronization [113]. Dufour et al. [43] showed that object churn is the main source of performance overhead in framework-intensive Java applications. Identifying places where object churn happens leverages performance understanding and is the basis of escape analysis [30, 83].

Object churn distance is a metric defined recently by Sewe et al. [111], and which has been shown to exhibit variation between Java and Scala workloads.

We illustrate the concepts behind the metric in Figure 4.1. For each object, we keep track of the calling contexts where an object has been allocated and where it died, i.e., stopped being referenced. The dynamic churn distance is then the distance between the allocation and the death calling contexts via the closest capturing (common parent) context. This metric is of particular importance in dynamic languages where primitive types are boxed—these workloads exhibit lower average churn distances. We group objects by their churn distances and count the frequency for each group.

Impact of zeroing. According to the JVM specification [76], every primitive and reference type has to be initialized to a zero value—0 in case of primitive types, `false` in case of a boolean type, and `null` in case of a reference type. Yang et al. [133] have shown that zeroing has a large impact on performance. However, different languages have different rules concerning the initialization of fields, and different programming styles lead to greater or lesser extents of explicit initialization. For example, more declarative languages are less likely to rely on constructor-based piecewise imperative initialization of objects than conventional Java code.

A zero initialization analysis can help compiler developers to see whether implicit zeroing is actually necessary—fields that are written before they are first read do not need to be explicitly zeroed. Our zeroing analysis records occurrences of this pattern. The metric is a count of unnecessary zeroing of primitive and reference fields.

Identity hash codes. The JVM requires that every object has a hash code. If the object does not override the `hashCode()` method, then `identityHashCode()` is used instead. Implementation of the latter varies between JVM implementations, but commonly, a computed identity hash code is stored in the header of each object. This incurs costs in memory and cache usage. The overhead can be eliminated by using header compression techniques that define the default hash code of an object to be its address [8]. The hashcode is explicitly stored only if an object has been moved by the GC *and* its identity hash code has been exposed—in such case, an extra header slot is lazily added to the object.

In workloads where the identity hash code is rarely used, this extra slot will rarely be allocated, yielding lower memory consumption with little runtime cost. In other workloads, eagerly allocating the header space for the hash code will yield better performance. Consequently, some heuristic is necessary to decide between the two approaches.

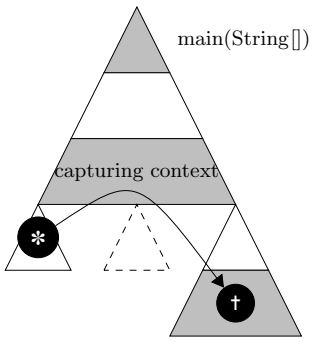

Figure 4.1. The churn distance of an object is computed as the distance between its allocation (✻) and death (†) calling contexts via their closest capturing context.

We define three metrics over binned invocation counts: frequency of objects receiving overridden `hashCode` invocations; frequency of objects receiving `System.identityHashCode` invocations; frequency of objects receiving the default `Object.hashCode` invocation (either by lack of override, or use of `super`).

Object lifetimes and sizes. Some languages allocate more, smaller and/or shorter-lived objects than others. Object lifetime analysis is of particular importance for GC developers. New GC algorithms are designed and evaluated by simulation based on object lifetime traces. An example of such an algorithm is lifetime-aware GC [68], in which the allocator lays out objects based on their death-time predictions. At each collection only objects that are expected to die are scavenged. An object's lifetime together with its size provide an estimate of the GC cost, since larger objects that live longer incur greater overhead than small, short-lived ones.

Our lifetime metric counts objects binned according to their survival time measured in cumulative bytes allocated ($\leq$1MiB, sim. 2MiB, 4MiB and 100MiB, and a separate bin for objects not surviving beyond nursery collection). The size metric collects a binned distribution of object sizes (including the header).

4.1.3 Code generation concerns

To take advantage of the infrastructure provided by a JVM, i.e., its just-in-time (JIT) compiler and the GC, a JVM language should be compiled into Java bytecode—while interpreted JVM languages exist, the optimization performed by the JVM only apply to the interpreter and not to the code it is executing. For compiled JVM languages, the resulting bytecode executed by the JVM plays a major role in the resulting performance. To aid in compiler construction, the last set of metrics characterizes properties that affect dynamic optimizations, which in turn depend on the use of virtual dispatch, the density of procedural abstraction, argument passing behaviors, and the overall instruction mix.

Instruction mix. An instruction mix metric can describe the nature of the application—

whether it is floating-point intensive or pointer intensive. This is relevant because, for example, some languages are more commonly used for numerical computations. This metric can be used for checking the diversity of the benchmarks in a benchmark suite, thus verifying that the benchmark suite indeed covers different application domains. Some interpreters such as in CACAO VM use super instructions [46, 54]. Instruction mix metric can identify right bytecode instruction sequences that can be used as super superinstructions. Moreover, this metric can lead to possible dynamic optimizations. For instance, array bounds check removal for array intensive applications can help further optimizations like code motion and loop transformations.

To classify applications based on the instruction mix they execute, the bytecodes executed by the JVM are split into groups that are specific to particular application types. In contrast to Dufour et al. [41], who grouped over 200 individual bytecodes manually, we use principal component analysis (PCA) [95] to reduce the dimensionality of the data, and to obtain a high-level view of the instruction mix in which the groupings of bytecode instructions are tailored to the workload.

Stack usage and recursion depth. This is an important metric for the developers of dynamic languages supporting the functional programming paradigm, e.g., Clojure. Functional languages often leverage recursion to perform iterations. Therefore it is very important for compiler developers of those languages to perform tail call elimination, such that an executed method will not allocate any new stack frames and perform recursive calls in constant space.

Our metric collects the distribution of stack heights upon each of three cases of method calls: all method calls, "potentially recursive" calls (virtual calls which *can* dispatch to the same method), and "true recursive" calls (which actually *do* dispatch in this way, whether virtually or by `final`).

Argument passing. Information on parameters passed to methods can be used by JVM developers to choose an optimal calling convention in JIT-compiled code, making use of the registers available on the target architecture. Some architectures require particular types of arguments to be passed differently, for example, using special floating-point registers. We partition arguments into three kinds—integer primitive values, references and floating-point values—and count each separately for each call. Our metrics bin all method invocations by their total argument count, then for each bin, compute a 5-vector counting the number of those arguments that are floating-point (zero to four and $\geq$5; elements beyond the total argument count are always zero), and similarly for reference arguments.

Basic-block hotness. Hotness metrics are fundamental, since any JVM with a just-in-time compiler optimizes the code based on its hotness (i.e., the code that is most frequently executed). While hotness is traditionally identified at the granularity of methods, modern dynamic compilers instead use trace-based approaches which rely on identifying sequences of hot basic blocks (possibly crossing method boundaries).

These are particularly popular among contemporary dynamic language implementations such as PyPy [22] or Mozilla's TraceMonkey Javascript implementation.[2] Therefore, a finer-grained hotness metric is useful.

Having both method and basic block hotness data can indicate the relative gains from different compiler optimizations (say, inlining versus loop unrolling). Our metrics report to which extent the most executed 20% of all (distinct) methods in the code contribute to overall dynamic bytecode execution, and likewise for basic blocks.

4.2 Toolchain Description

In the following we describe the toolchain for collecting different kinds of metrics discussed in the previous section. Our toolchain consists of several distinct tools with a common infrastructure which is designed for ease of use and extension.

4.2.1 Deployment and Use

The primary goal of our infrastructure is to avoid imposing unnecessary overheads on developers wanting to make use of dynamic metrics. These include learning and setting up multiple new runtime environments and/or instrumentation tools. To avoid such overheads, all our tools are implemented using DiSL, a domain specific language for instrumentation. DiSL provides full bytecode coverage, meaning execution within the Java class library is covered. This is essential for the accuracy of our metrics. Each metric can be computed for a given workload application using a single script invocation. Execution produces a trace, whose contents vary according to the metric being computed. A separate post-processor script uses the trace to calculate the metric's value. This separation is useful because in some cases multiple metrics can be computed from the same trace; several of our metrics exploit this, as we explain shortly (§4.2.2).

Since all instrumentation is done using the same high-level domain-specific language (DiSL), our implementations are amenable to customization with relatively low familiarization overhead. We envisage they can usefully be tweaked and extended for specific needs, such as dumping the trace in a different format or adding a custom online analysis. A subset of our metrics are query-based, and these offer an additional level of customizability, since custom queries can be written in the high-level XQuery language.

The tools in our toolchain exhibit acceptable runtime overhead. Among the most heavyweight of our tools is JP2, which produces calling context trees; this incurs an overhead factor of roughly 100 [108]. However, this cost is amortized in that many different metrics are computed (as queries) over its output. Object lifetime analysis also relies on heavyweight instrumentation. However, other tools instrument considerably

[2] `https://developer.mozilla.org/en-US/docs/SpiderMonkey/Internals/Tracing_JIT`

```
public class FieldState {
  private State currentState = State.VIRGIN;
  private enum State { VIRGIN, IMMUTABLE, MUTABLE };
  private boolean defaultInit = false;
  public synchronized void onRead() {
    switch (currentState) {
      case VIRGIN:
        defaultInit = true;
        currentState = State.IMMUTABLE;
        break;
  }   }
  public synchronized void onWrite(boolean isInConstructor) {
    switch(currentState) {
      case VIRGIN:
      case IMMUTABLE:
        currentState = isInConstructor ? State.IMMUTABLE : State.MUTABLE;
        break;
    }
  }
  /* ... */
}
```

Figure 4.2. The field immutability state machine.

fewer events—for example, hashcode analysis instruments only a few method entries—and incur correspondingly less overhead. We note that our instrumentation-based approach generally outperforms like-for-like metric implementations using the older JVMPI interface, including those described by Dufour et al. [41].

Metrics such as field immutability, zeroing, field sharing, and use of identity hashcodes are collected via custom tools that use DiSL to perform bytecode instrumentation. In each case, the instrumentation maintains shadow state for each object. Depending on the analysis different events are intercepted and different information is stored in a shadow state. For example, to measure immutability, our shadow object keeps track of all field accesses to the underlying object, according to a state machine. Each shadow object records the class name, object allocation site and an array of field states, each of which is a state machine with states virgin (i.e., not read or not written to), immutable (i.e., read or was written to inside the dynamic extent of its owner object's constructor), or mutable (otherwise). Figure 4.2 depicts the corresponding FieldState class.

A suitably modified version of this shadow object approach is used in field sharing, field synchronization and hash code analysis (storing counters for thread accesses, counters for monitor ownership, and counters for executions of Object.hashCode() and System.identityHashCode() methods, respectively).

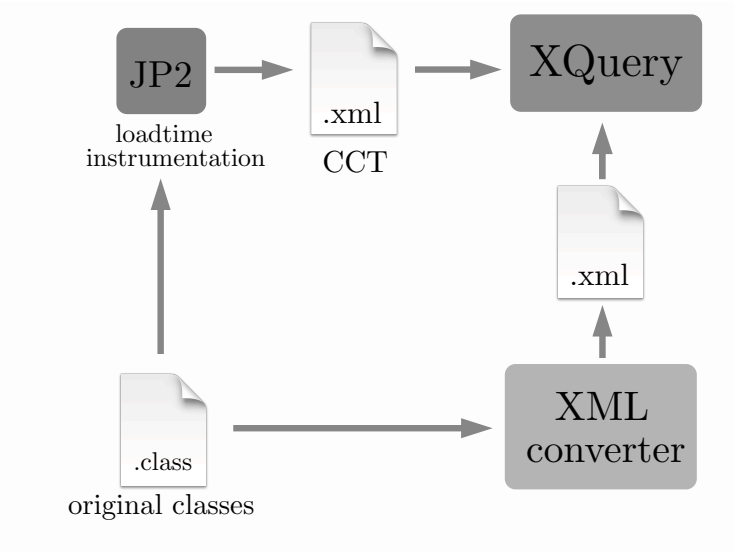

Figure 4.3. Query-based metrics are implemented on top of JP2 [108].

4.2.2 Query-based Metrics

Many of our metrics are defined as queries over trace data. Specifically, these are metrics concerning instruction mix, call-site polymorphism, stack usage and recursion depth, argument passing, method and basic block hotness, and use of boxed types. All these metrics are obtained using JP2 [107, 108], which has been reimplemented using the DiSL instrumentation framework to fit well into our framework. JP2 is a calling-context profiler which produces execution traces in the form of an annotated calling-context tree (CCT). Each node in a CCT corresponds to a particular callchain and keeps the dynamic metrics, such as number of method invocations and number of executed bytecodes. JP2 is call-site aware, meaning different call sites in the same method are distinguished even if their target method is the same. Unlike many other profilers, JP2 performs both inter- and intra-procedural analysis and reports dynamic execution counts for each basic-block of code in methods.

JP2 provides complete dumps of an entire execution, including coverage within the Java class library and some coverage of native code. Although native methods do not have any bytecode representation, JP2 uses JVMTI's native method prefixing feature to insert bytecode wrappers for each native method. Control flow within native methods is covered only from points where these call back into Java code or other prefixed natives.

Figure 4.3 depicts a three-step process of computing dynamic metrics with JP2. First, the application is instrumented for profiling; second, the collected profile is dumped in an XML-based format for later offline analysis; finally, the desired metrics are computed offline. Dumping in XML format allows using off-the-shelf tools for metrics computation. We use XQuery for formulating metrics as queries.

```
for $method in functx:distinct-nodes(
  for $bb in $methods/dcg:basicBlock
    order by $bb/dcg:executionCount/xs:
             long(.) descending
    return $bb/..)
```

Figure 4.4. Example of a query for identifying methods with hottest basic blocks. `dcg` refers to "dynamic call graph", referring to the calling context tree. Other identifiers are self-explanatory.

4.2.3 Instrumentation

Some of the information needed for our metrics' computation is not stored in a CCT, but depends on static properties of class files. For this, we use another facility of JP2, which can dump a list of all classes loaded during execution. These classes are converted to an XML representation to allow querying alongside the CCT data [108]. Many of our queries make use of the ability to cross-reference between CCT and class data.

Figure 4.4 shows an example of a query for identifying methods with hottest basic blocks. It can be useful for finding methods with rich intra-procedural control flow, but with low method execution counts that cannot be spotted with typical profilers. The algorithm is straightforward: return the methods of the application, sorted in decreasing order of the total execution counts over all their contained basic blocks.

A key benefit of the query-based design is that custom queries can be used to formulate previously unanticipated metrics. For example, dumped CCTs contain enough information to recover a k-calling context forest, which offers an alternative (k-bounded) level of context sensitivity offering advantages in certain scenarios [7].

The separation between dumps and queries avoids potential problems with nondeterminism. Multiple different metrics can be computed without the need for repeated application runs, hence avoiding any risk of divergent behaviour across such runs.

Chapter 5

Workload Characterization of JVM Languages

5.1 Experimental Setup

To obtain information necessary to answer the research question of this dissertation, we used our toolchain to collect the presented dynamic metrics. We analyzed the collected data, looking for significant differences between the pure Java workloads and the workloads induced by JVM languages, which may hint at optimization opportunities for programs executing on the JVM.

5.1.1 Examined Metrics

In Section 5.2 we present details and discuss results for the following metrics:

Call-site Polymorphism. Hints at opportunities for optimizations at polymorphic call-sites, e.g. inline caching [60] (based on the number of receiver types), or method inlining [39] (based on the number of target methods).

Field, Object, and Class Immutability. Enables load elimination [13] (replacing repeated accesses to immutable objects with an access to a compiler-generated temporary stored in a register), and identifies objects and side-effect-free data structures amenable to parallelization.

Object Lifetimes. Determines garbage-collector (GC) workload, and aids in design and evaluation of new GC algorithms, e.g. the lifetime-aware GC [68].

Unnecessary Zeroing. Hints at opportunities for eliminating unnecessary zeroing of memory for newly allocated objects, which comes with a performance penalty [133].

Identity Hash-code Usage. Hints at opportunities for reducing header size for objects that never need to store their identity hash code (often derived from their memory location upon first request [8]).

While we have collected the full range of dynamic metrics for different workloads, in this dissertation we only discuss metrics that show striking difference between static and dynamic languages. Thanks to our toolchain, we were able to collect metrics that cover both the application (and the coressponding language runtime) and the Java Class Library (including any proprietary JVM vendor-specific classes) on a standard JVM.

5.1.2 Workloads

The lack of an established benchmark suite—something akin to the DaCapo [17] or SPECjvm2008 [125] suites, but for JVM languages other than Java—is a widely recognized issue. A new benchmark suite has recently been proposed for Scala [112], but there is no such suite for most dynamic languages, including those we consider here.

The closest to a benchmark suite for dynamic languages is the Computer Language Benchmarks Game (CLBG) project [32], which collects and compares performance results for various benchmarks implemented in many different programming languages, including Java, Clojure, Python, Scala, JavaScript, and Ruby, which we chose for the comparison. Each benchmark describes an algorithm, and there is an idiomatic implementation of that algorithm in each language. Considering their size and focus on algorithms, the CLBG benchmarks fall into the category of micro-benchmarks, and only represent a certain aspect of real-world applications. The authors of the CLBG project are well aware of this, and explicitly warn against jumping to conclusions[1]. Despite some controversies, and for the lack of any better multi-language benchmark suite, the CLBG project remains a popular source of rough estimates of raw performance achievable by many programming languages.

Benchmarks from the CLBG project have been used in the recent study on performance differences between Python compilers [25]. Li et al. [75] published an exploratory study characterizing workloads for five JVM languages using CLBG benchmarks. To provide complementary results for comparable workloads, we have decided to adopt the approach of Li et al., and based our study (mostly, but not completely) on 9 CLBG benchmarks, listed in Table 5.1 along with a brief description and inputs used.

Still, to avoid relying solely on micro-benchmarks, we complemented our workload selection with 3 real-world application benchmarks for each examined JVM language, listed in Table 5.2. These unfortunately lack the nice property of being idiomatic implementations of the same task. The avrora, eclipse, fop and jython benchmarks come from the DaCapo suite [17], while apparat, factorie, and scalac come from the

[1] `http://benchmarksgame.alioth.debian.org/dont-jump-to-conclusions.php`

Benchmark	Description	Input
binarytrees	Allocate and deallocate many binary trees	16
fannkuch-redux	Repeatedly access a tiny integer-sequence	10
fasta	Generate and write random DNA sequences	150,000
k-nucleotide	Repeatedly update hashtables and k-nucleotide strings	fasta output
mandelbrot	Generate a Mandelbrot set and write a portable bitmap	1,000
nbody	Perform an N-body simulation of the Jovian planets	500,000
regexdna	Match DNA 8-mers and substitute nucleotides for IUB code	fasta output
revcomp	Read DNA sequences and write their reverse-complement	fasta output
spectral-norm	Calculate an eigenvalue using the power method	500

Table 5.1. Benchmarks from the CLBG project (implemented in Java, Clojure, Ruby, Python, Scala, and JavaScript) selected for workload characterization.

Scala Benchmarking suite [112]. deltablue, raytrace, and richards come from the Octane benchmarking suite [86]. clj-pdf [33], frak [48], minilight [82], scavis [110], rubyflux [105], voodoo [130], opal [88], and clojure-script [34] are open-source projects from GitHub.

5.1.3 Measurement Context

All metrics were collected with Java 1.6, Clojure 1.5.1, JRuby 1.7.3, Rhino 1.7, Scala 2.10.3, and Jython 2.7 runtimes, yielding a total of 72 different language-benchmark combinations, all executed using the OpenJDK 1.6.0_27 JRE running on Ubuntu Linux 12.0.4.2. Due to the high number of combinations and extensive duration of the experiments, we have not included different language runtimes among independent variables. Similarly, we have not varied the execution platform, because the metrics are defined at the bytecode level, and can be considered largely JVM[2] and platform independent.

5.2 Experiment Results

5.2.1 Call-site Polymorphism

Hot polymorphic call-sites are good candidates for optimizations such as inline caching [60] and method inlining [39, 121], which specialize code paths to frequent receiver types or target methods. In the case of dynamic languages targeting the JVM, specialization is considered to be one of the most beneficial performance optimizations [25]. In our study, we collected metrics that are indicative specifically for method inlining, which

[2]Different vendors may provide different implementation of the Java Class Library as well as other proprietary classes in their JVM.

	Application	Description	Input
Clojure	clojure-script	A compiler for Clojure that targets JavaScript	twitterbuzz source
	clj-pdf	A library for generating PDFs from Clojure	default example
	frak	A library for transforming strings into regular expressions	default example
Java	avrora	A simulator of programs running on a grid of AVR microcontrollers	DaCapo default
	eclipse	An integrated development envinronent (IDE)	DaCapo default
	fop	An output-independent print formatter	DaCapo default
JavaScript	deltablue	A one-way constraint solver	Octane default
	raytracer	A ray tracer benchmark	Octane default
	richards	An OS kernel simulation benchmark	Octane default
JRuby	opal	A Ruby to JavaScript compiler	meteor source
	rubyflux	A Ruby to Java compiler	meteor source
	voodoo	Image manipulation library for Ruby	simple image
Jython	jython	An interpreter of Python running on the JVM	DaCapo default
	minilight	A minimal global illumination renderer	default example
	scavis	A scientific computation environment	Matrix multiplication
Scala	apparat	Framework to optimize ABC, SWC, and SWF files	Scalabench default
	factorie	Toolkit for deployable probabilistic modeling	Scalabench default
	scalac	Compiler for the Scala 2 language	Scalabench default

Table 5.2. Real-world applications selected as benchmarks to complement the CLBG benchmarks for workload characterization.

removes costly method invocations and increases the effective scope of subsequent optimizations.

The results consist of two sets of histograms for each language, derived from the number of target methods and the number of calls made at each polymorphic call-site during the execution of each workload. The plots in Figures 5.2–5.6 show the number of call sites binned according to the number of targeted methods (x-axis), with an extra bin for call-sites targeting 15 or more methods. The plots in Figures 5.8–5.12 then show the actual number of invocations performed at those call sites.

We observe that polymorphic invocations in the CLBG benchmarks do not target more than 6 methods for Java, and more than 11 methods for Scala (with the majority below 6). This is not surprising, given the microbenchmark nature of the CLBG workloads. The situation is vastly different—and more realistic—with the real-world applications. Still, on average, 98.2% of the call sites (accounting for 90.8% of all method calls) only had a single target in the case of Java and 97% of the call sites (accounting for 80% of all method calls) target a single method in the case of Scala.

The microbenchmark nature of the CLBG workloads is much less pronounced (compared to the real application workloads) in the case of dynamic JVM languages. This suggests that even the CLBG workloads do exhibit some of the traits representative of a particular dynamic language.

The results for the Clojure workloads show that polymorphic invocations target 1 to 10 methods, with an average of 99.3% of the call sites (accounting for 91.2% of method

calls) actually targeting a single method. The results for the Jython workloads show that polymorphic invocations mostly target 1 to 10 methods, with a small number of sites targeting 15 or more methods. Invocations at such sites are surprisingly frequent, but still 98.7% of the call sites (accounting for 91.7% of method calls) target a single method.

Interestingly, JavaScript stands out from the rest of the dynamic languages, and the results look similar to those of Scala—the number of targets in both languages is similar, but JavaScript workloads perform more calls at callsites with lower number of targets. Also, the diference between microbenchmarks and real-world applications is not as pronounced as in the case of Scala. On average, 98% of the call sites (accounting for 86% of the method calls) target a single method.

Finally, the results for JRuby show little difference between the CLBG benchmarks and the real-world application, and consistently show a significant number of call-sites with 15 or more targets. Interestingly, the number of calls made at those sites is surprisingly high—comparable with the other sites. However, on average, 98.4% of the call sites (accounting for 88% of method calls) target a single method.

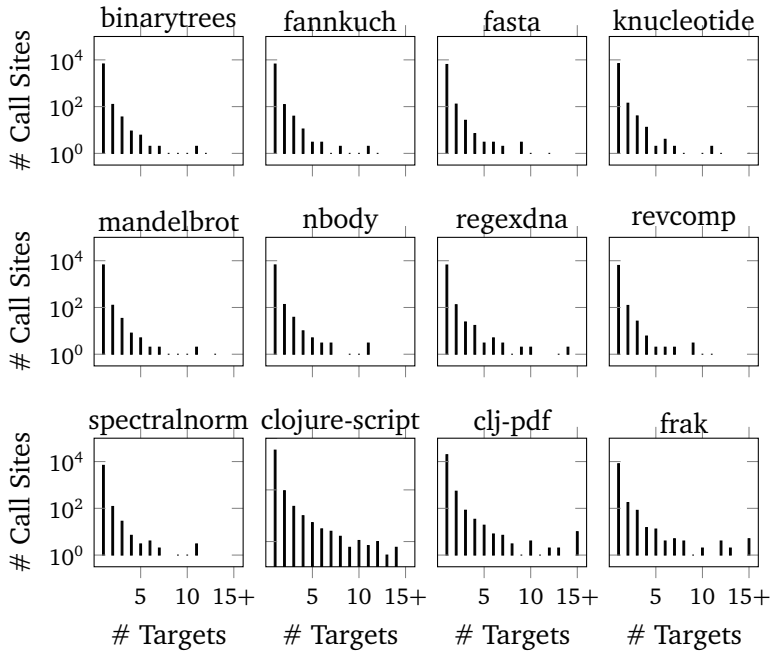

Figure 5.1. The number of dynamically-dispatched call sites targeting a given number of methods for the Clojure benchmarks.

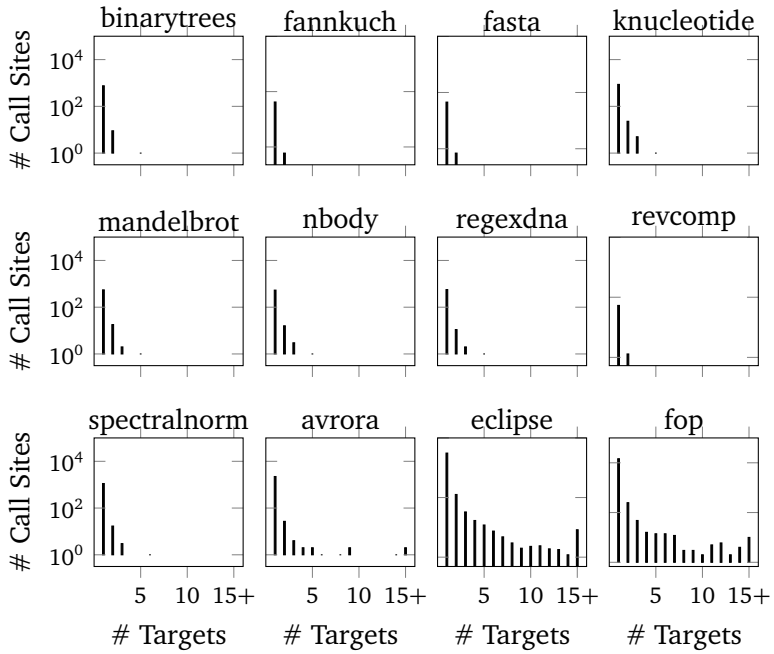

Figure 5.2. The number of dynamically-dispatched call sites targeting a given number of methods for the Java benchmarks.

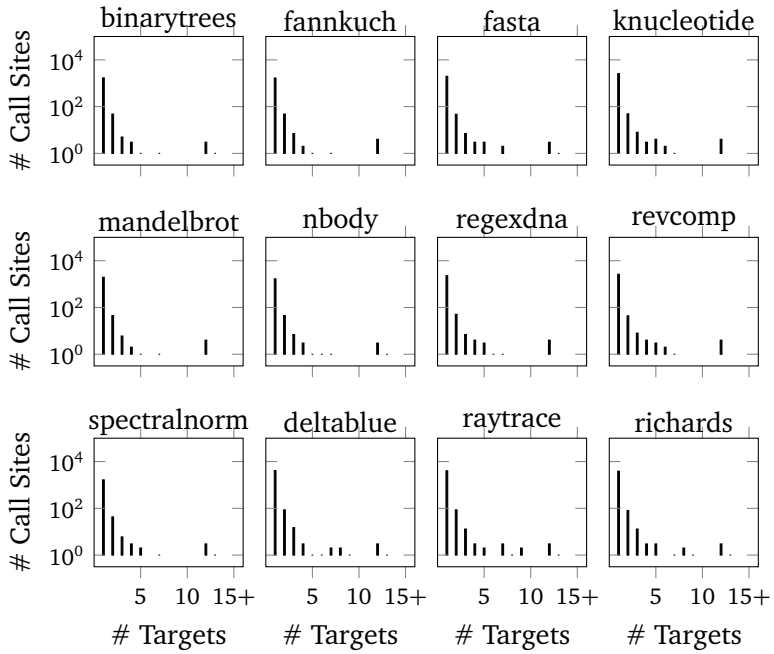

Figure 5.3. The number of dynamically-dispatched call sites targeting a given number of methods for the JavaScript benchmarks.

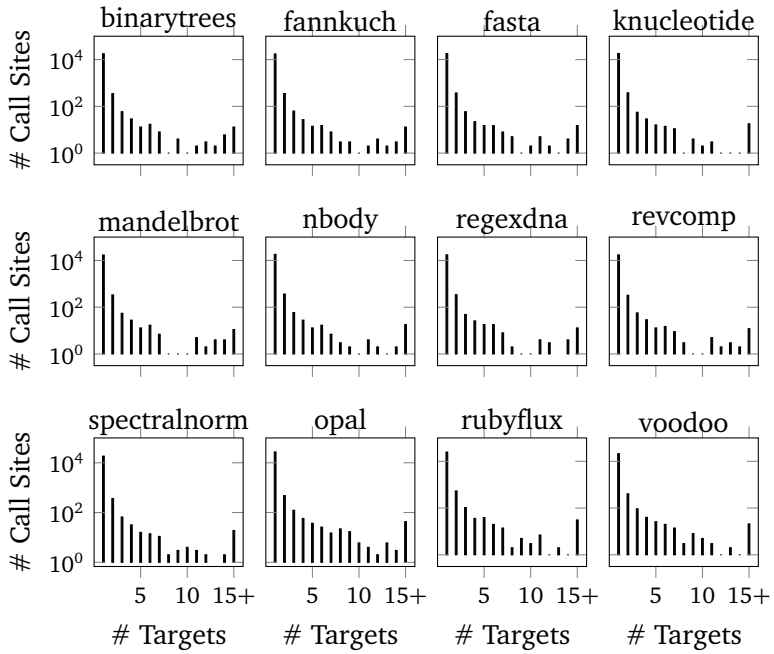

Figure 5.4. The number of dynamically-dispatched call sites targeting a given number of methods for the JRuby benchmarks.

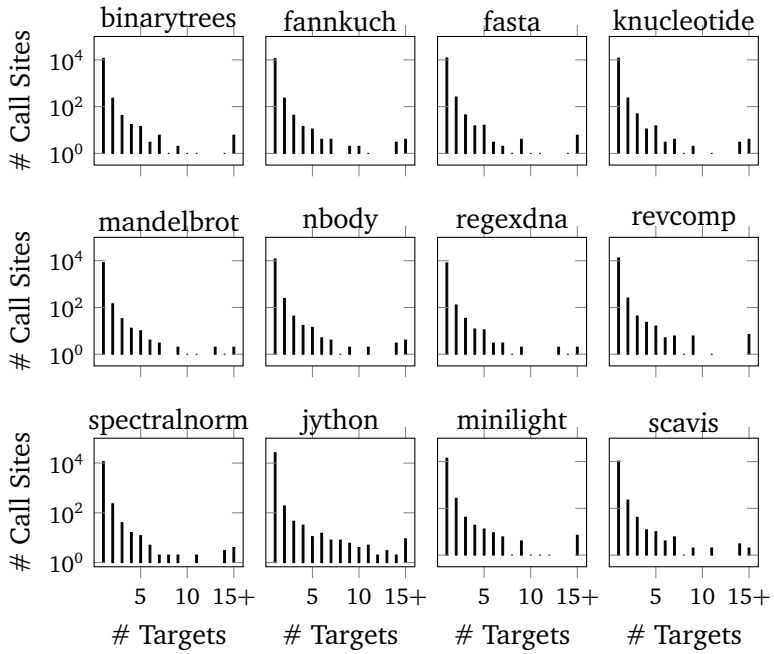

Figure 5.5. The number of dynamically-dispatched call sites targeting a given number of methods for the Jython benchmarks.

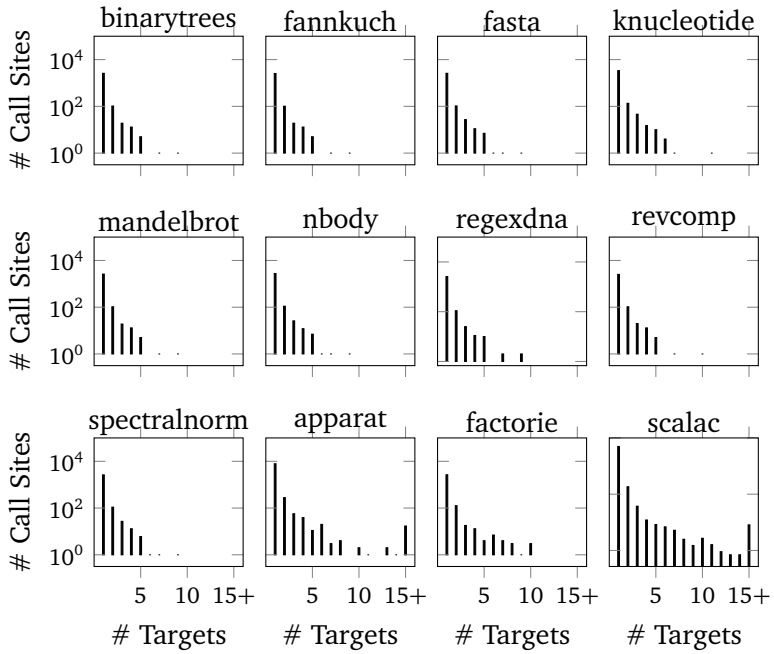

Figure 5.6. The number of dynamically-dispatched call sites targeting a given number of methods for the Scala benchmarks.

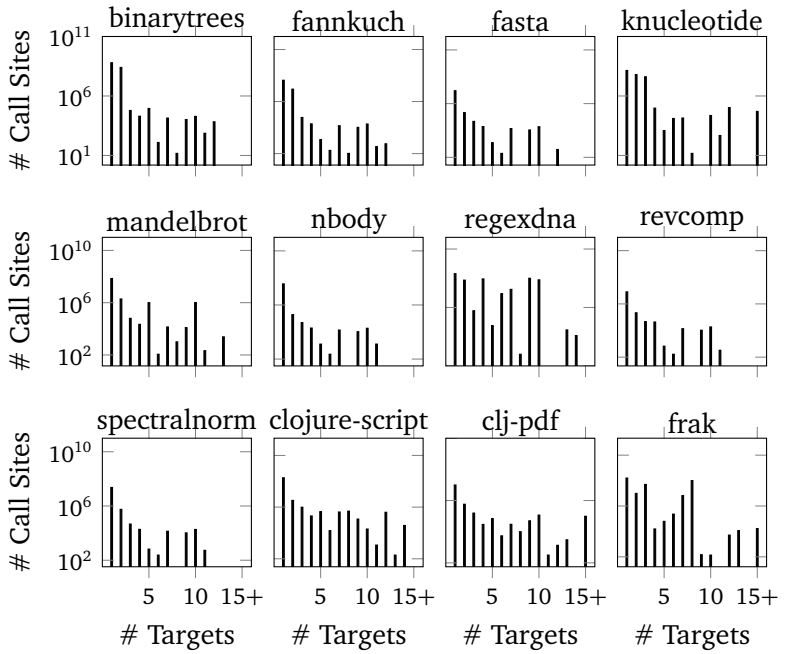

Figure 5.7. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Clojure benchmarks.

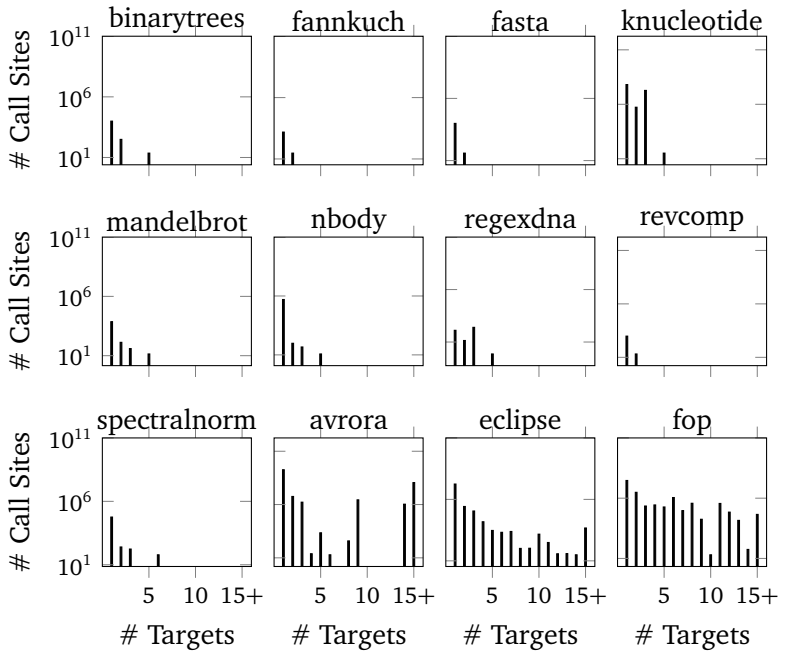

Figure 5.8. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Java benchmarks.

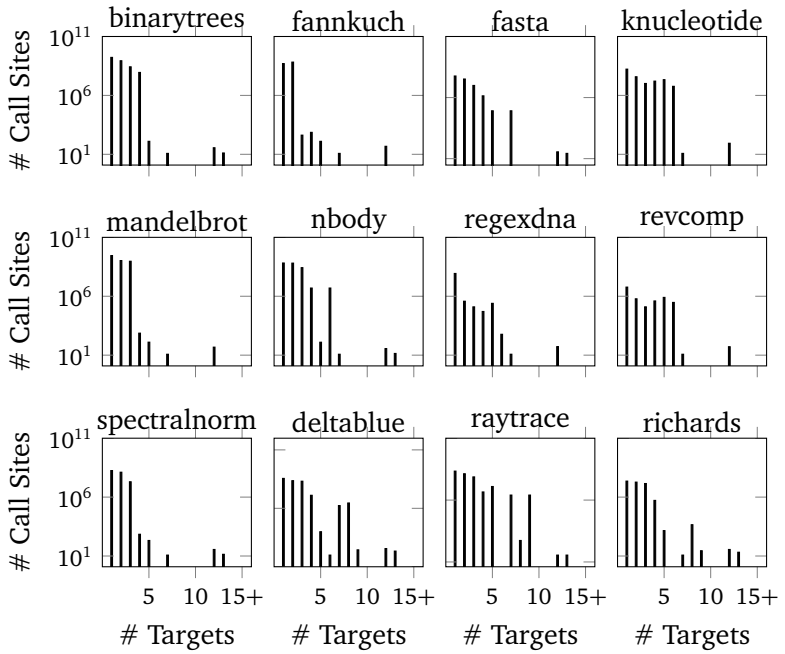

Figure 5.9. The number of dynamically-dispatched calls made at call sites with a given number of targets for the JavaScript benchmarks.

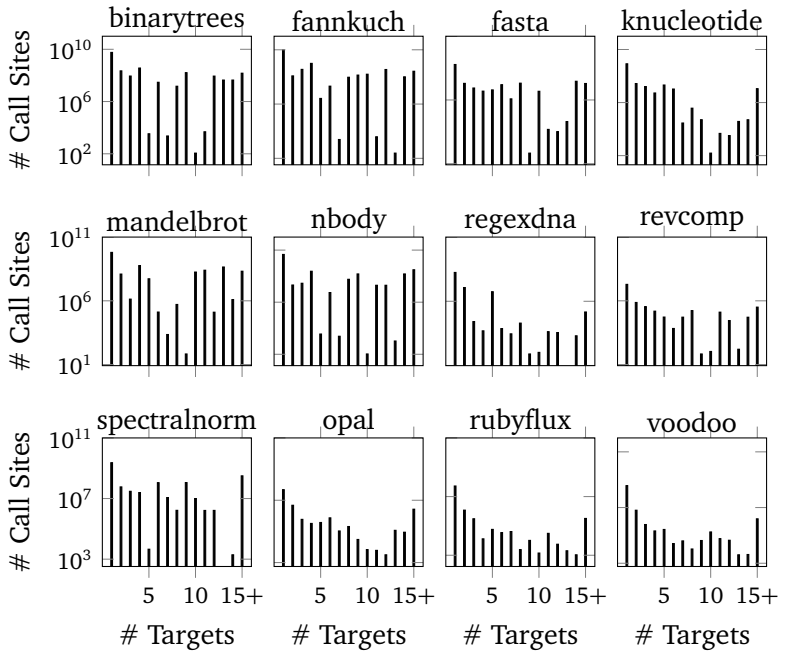

Figure 5.10. The number of dynamically-dispatched calls made at call sites with a given number of targets for the JRuby benchmarks.

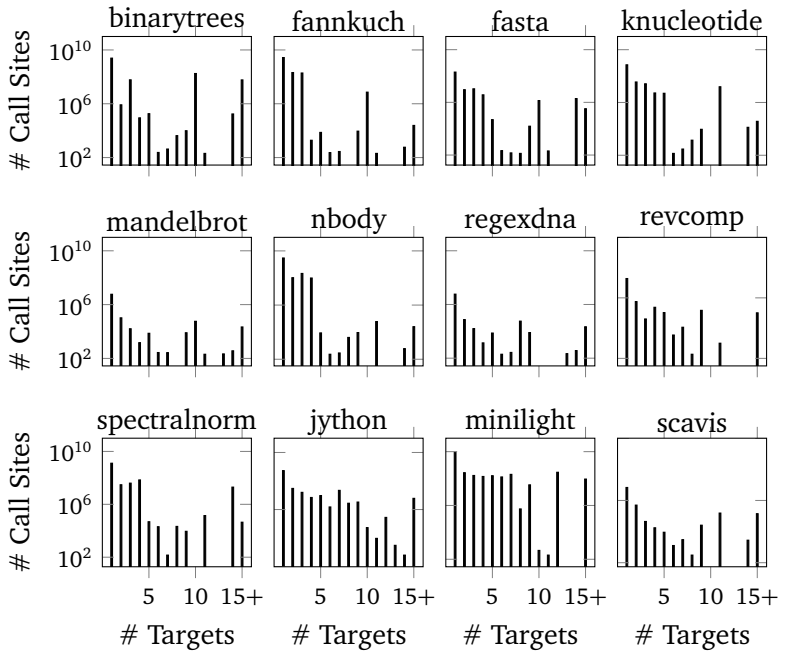

Figure 5.11. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Jython benchmarks.

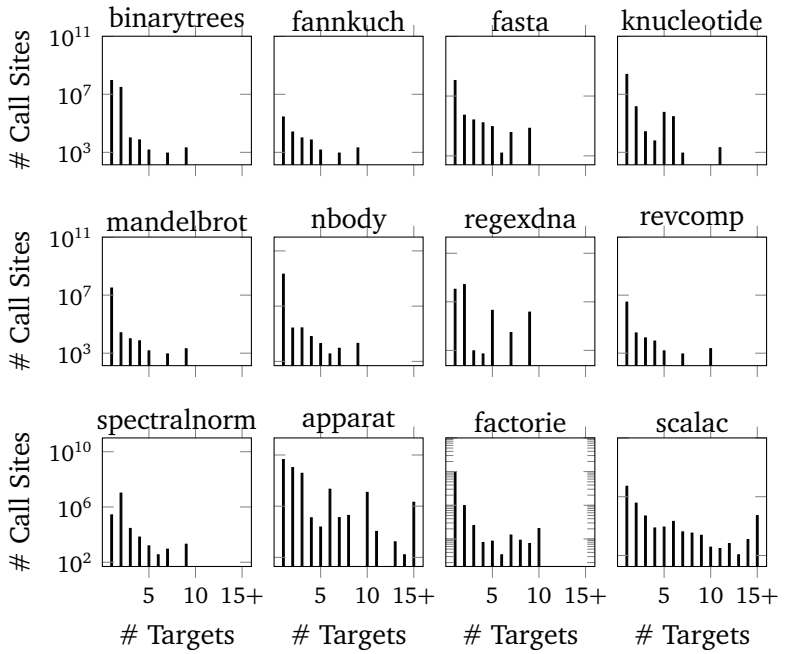

Figure 5.12. The number of dynamically-dispatched calls made at call sites with a given number of targets for the Scala benchmarks.

5.2.2 Field, Object, and Class Immutability

In our study of the dynamic behavior of JVM workloads, we use an extended notion of immutability instead of the "classic" definition: an object field is considered immutable if it is never written to outside the dynamic extent of that object's constructor. This notion is dynamic in the sense that it holds only for a particular program execution or for a specific program input [106, 111].

Extending this notion to objects and classes, we distinguish (1) per-object *immutable fields*, assigned at most once during the entire program execution, (2) *immutable objects*, consisting only of immutable fields, and (3) *immutable classes*, for which only immutable objects were observed. The results shown in Figure 5.13 indicate that there is a significant fraction of immutable fields (as per our definition) in most of the studied workloads, without significant differences between the CLBG and real-world benchmarks. Except in the Java binarytrees CLBG workload, we observed more than 50% of immutable fields in all benchmarks, with Clojure having the highest average number of immutable fields. Another interesting observation is that JavaScript benchmarks have very little fraction of reference instance fields.

At the granularity of objects, the results in Figure 5.14 show varying immutability ratios across different workloads. The ratios are consistently high, especially for the dynamic languages (mostly over 50%), with Clojure, JavaScript, and JRuby scoring almost 100% on five workloads (with four workloads common to Clojure and JRuby). This can be attributed to the large amount of boxing and auxiliary objects created by the language runtimes [75].

Finally, at the granularity of classes, the results in Figure 5.15 show consistent fractions of immutable classes across different workloads (except for Jython), with significant differences between the languages. These systematic differences can be attributed both to different coding styles typical for the respective languages, and to the number of helper classes produced by a particular dynamic language runtime environment.

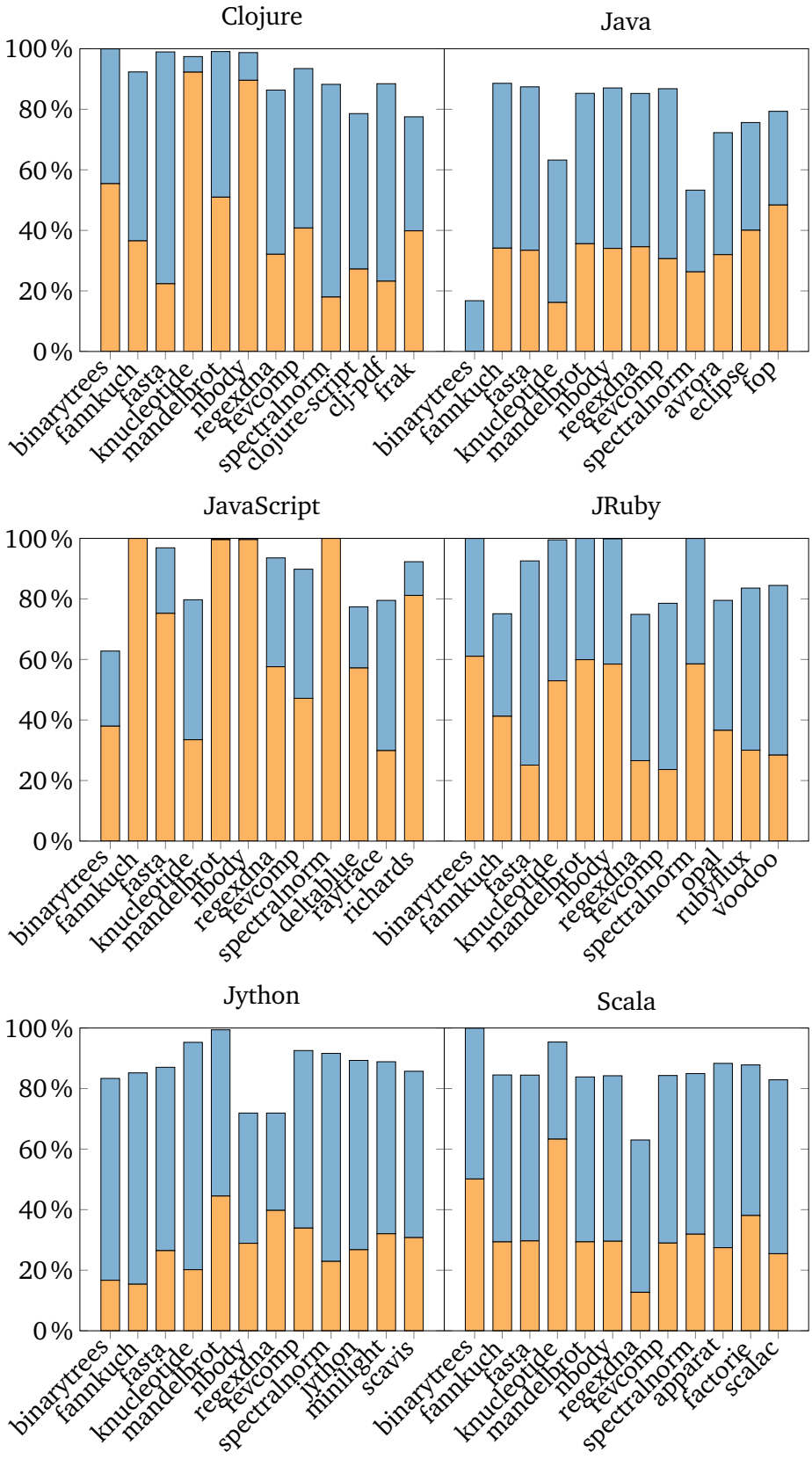

Figure 5.13. Fraction of primitive (■) and reference (■) *instance* fields that are per-object immutable

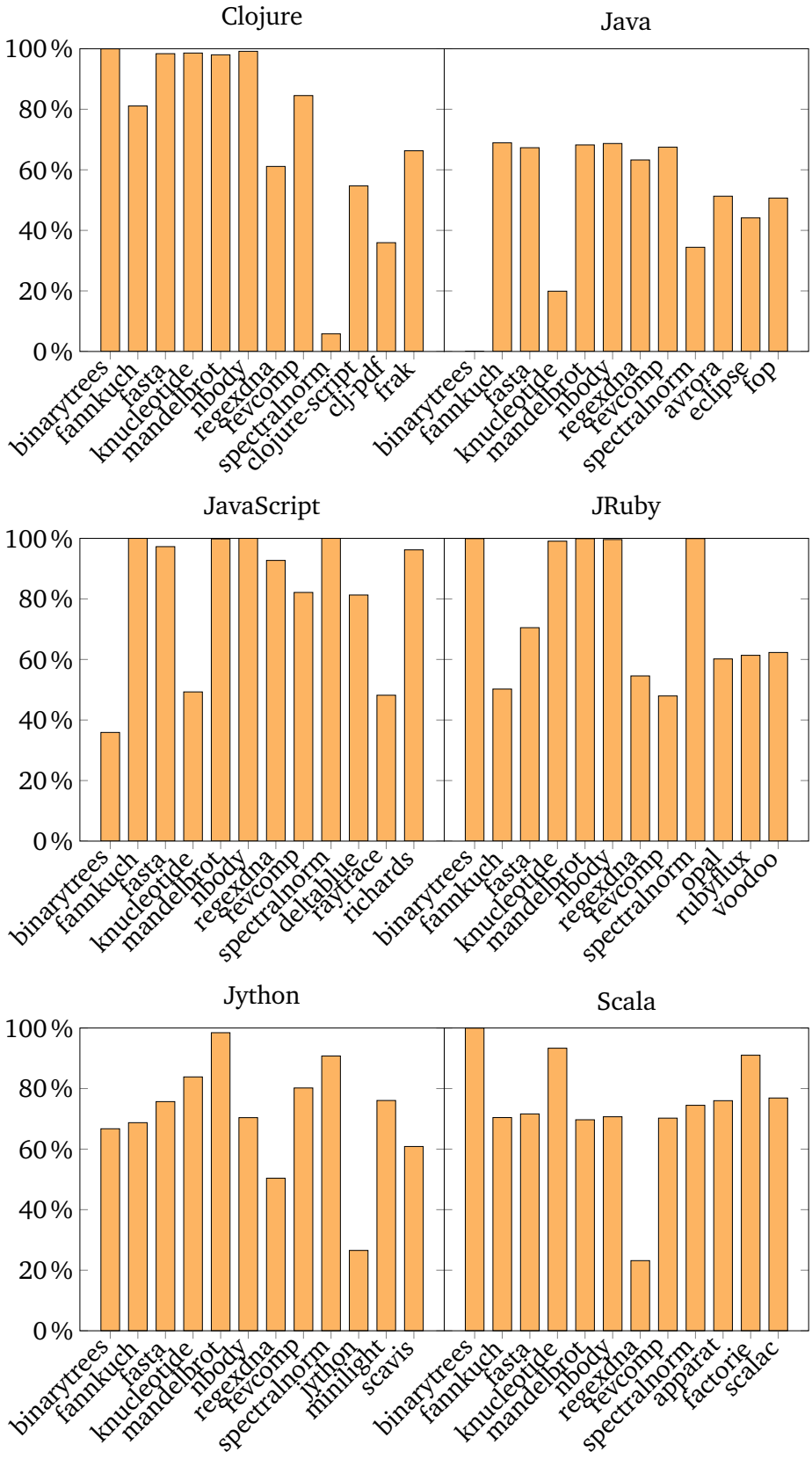

Figure 5.14. Fraction of immutable objects

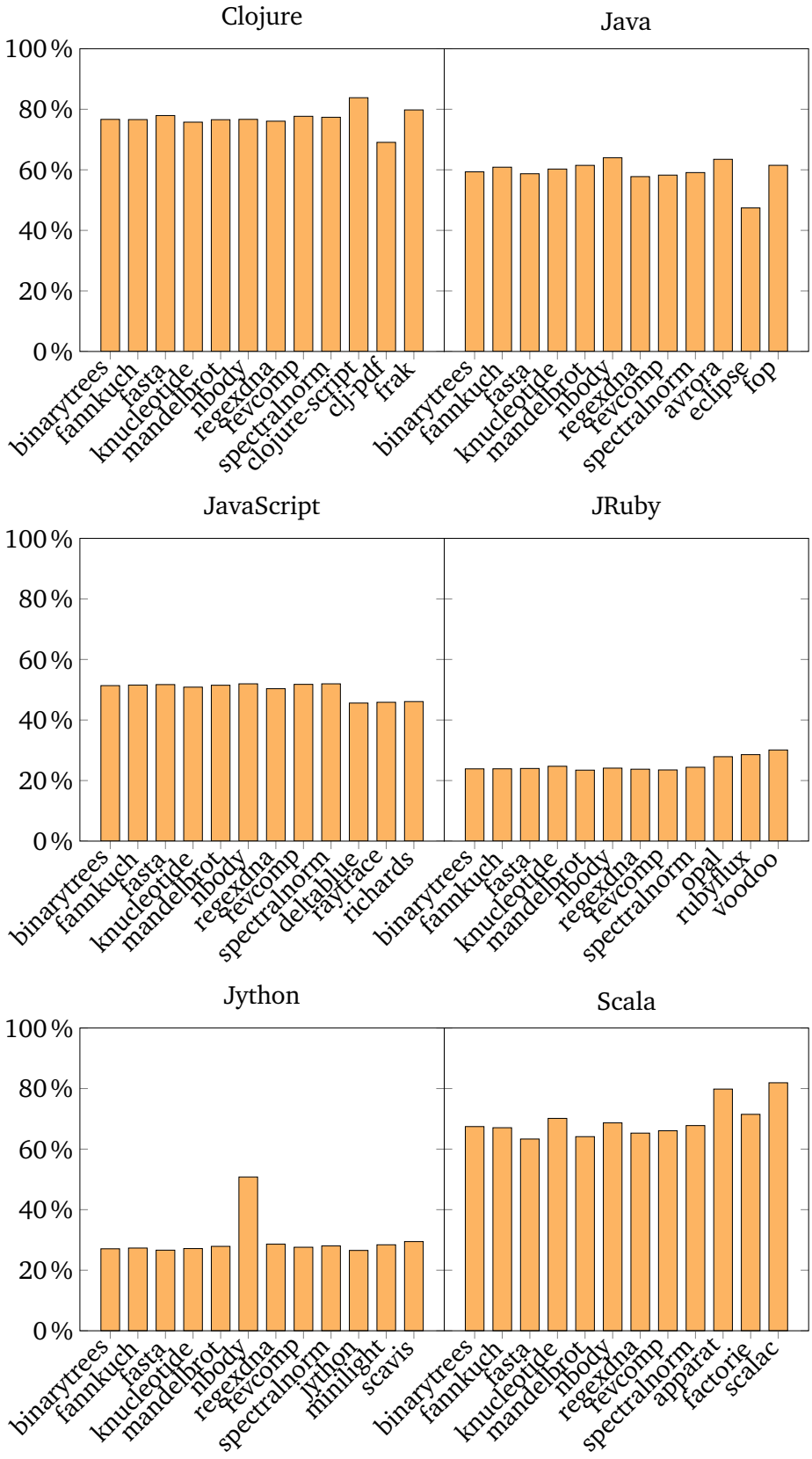

Figure 5.15. Fraction of immutable classes

5.2.3 Identity Hash-code Usage

The JVM requires every object to have a hash code. The default implementation of the hashCode method in the Object class uses the System.identityHashCode method to ensure that every object satisfies the JVM requirement. The computed hash code is usually stored in the object header, which increases memory and cache usage—JVMs therefore tend to use an object's address as its implicit identity hash code, and store it explicitly only upon first request (to make it persistent in presence of a copying GC).

That said, performance may be improved by allocating the extra header slot either eagerly or lazily, depending on the usage of identity hash codes in a workload. Since systematic variations in hash code usage were identified between Java and Scala workloads [111], we also analyzed hash code usage of the workloads in our study.

The results shown in Figure 5.16 suggest that the identity hash code is never requested for a vast majority of objects. Despite a comparatively frequent use of hash code in the Java and Scala workloads, the use of identity hash code remains well below 2% for most workloads. The dynamic languages appear to be using hash code very little, the exception being the knucleotide benchmark in the cases of Jython and JavaScript. Clojure shows the highest use of hash code among dynamic languages, in contrast to JRuby and JavaScript. The highest amount of per-object hash code operations can be found in JavaScript. Among the real-world benchmarks, only the eclipse and clojurescript benchmarks had objects on which both an overriden and the identity hash code methods were invoked.

Because dynamic languages appear to produce many short-lived objects (c.f. Section 5.2.5), the results suggest that object header compression with lazy identity hash code slot allocation is an adequate heuristic for the dynamic language runtimes, especially for JRuby and JavaScript.

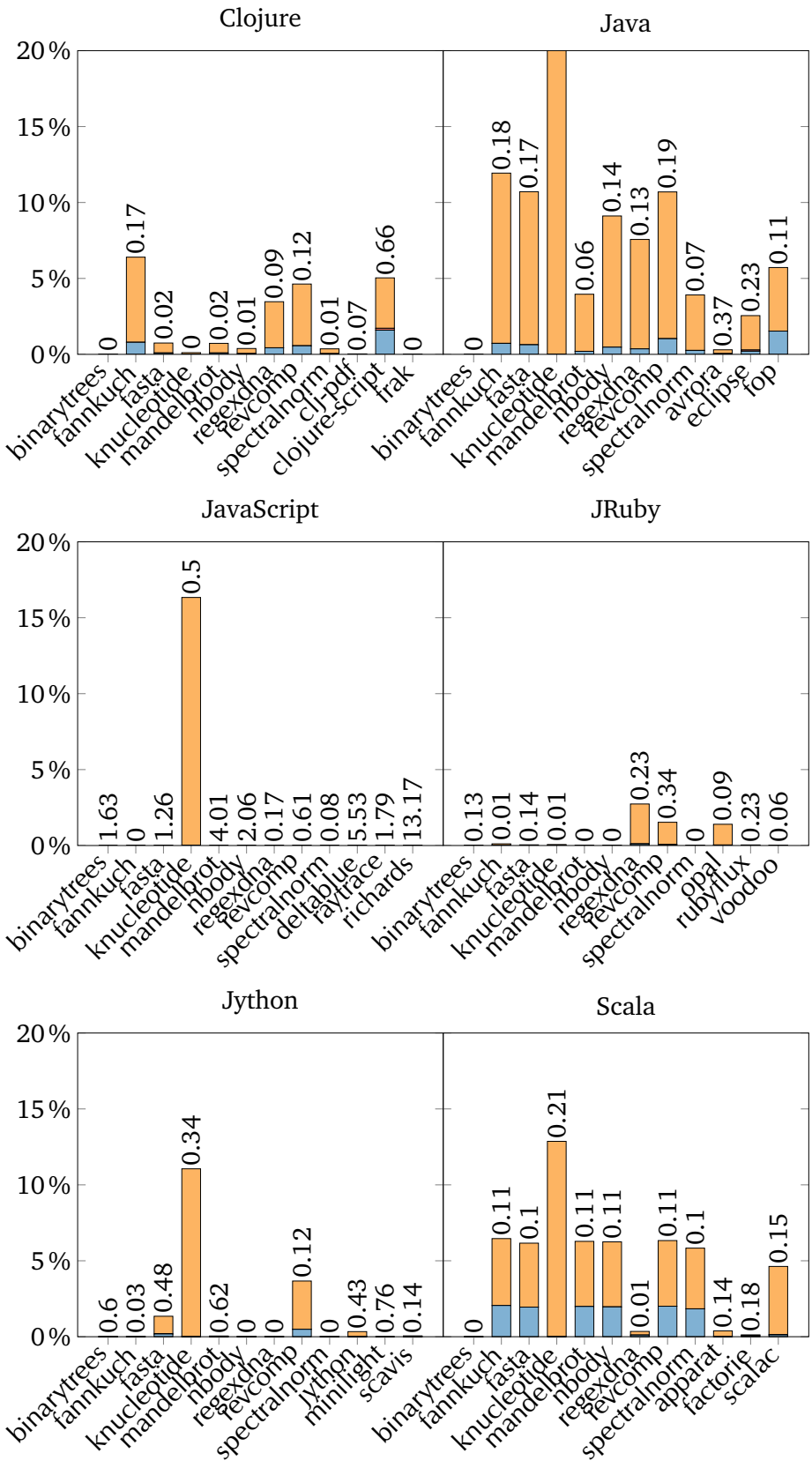

Figure 5.16. Fraction of objects hashed using an overriden hashCode method (■), the identityHashCode method (■), or both (■), along with an average number of hash operations per object.

5.2.4 Unnecessary Zeroing

The Java language specification requires that all fields have a default value of `null`, `false` or `0`, unless explicitly initialized. The explicit zeroing may cause unnecessary overhead [133], especially in the case of fields assigned (without being first read) within the dynamic extent of a constructor Our analysis detects and reports such fields.

A JVM could potentially try optimize away some of the initialization overhead, e.g., by not zeroing unused fields where appropriate, but it would have to ensure that the (uninitialized) field values are never exposed to the garbage collector. While such an optimization can make the results of our analysis less accurate, trying to detect and correctly handle the effect of this optimization in our analysis would be difficult, and possibly require making it JVM-specific, which is what we want to avoid. We therefore do not take these potential optimizations into account, and consider zeroing of an unused field to be mandatory.

Figure 5.17 shows the amount of unnecessary zeroing (according to our metric) performed by the workloads in our study. For the CLBG benchmarks, Clojure exhibits the highest average percentage of unnecessary zeroing (85.9%), followed by JavaScript (77.8%), Scala (71.74%), Jython (66.4%), JRuby (46.04%) and Java (49.03%). Interestingly, this language ordering appears to partially correlate with the ordering imposed by the percentage of immutable instance fields (shown in Figure 5.13) with average values of 91.5% for Clojure, 90.8% for JavaScript, 89.2% for JRuby, 86.8% for Jython, 83.9% for Scala, and 73.4% for Java. Our results therefore suggest that the more immutable instance fields exist, the more unnecessary zeroing takes place. In other words: fields are assigned once in the constructor, not through setters.

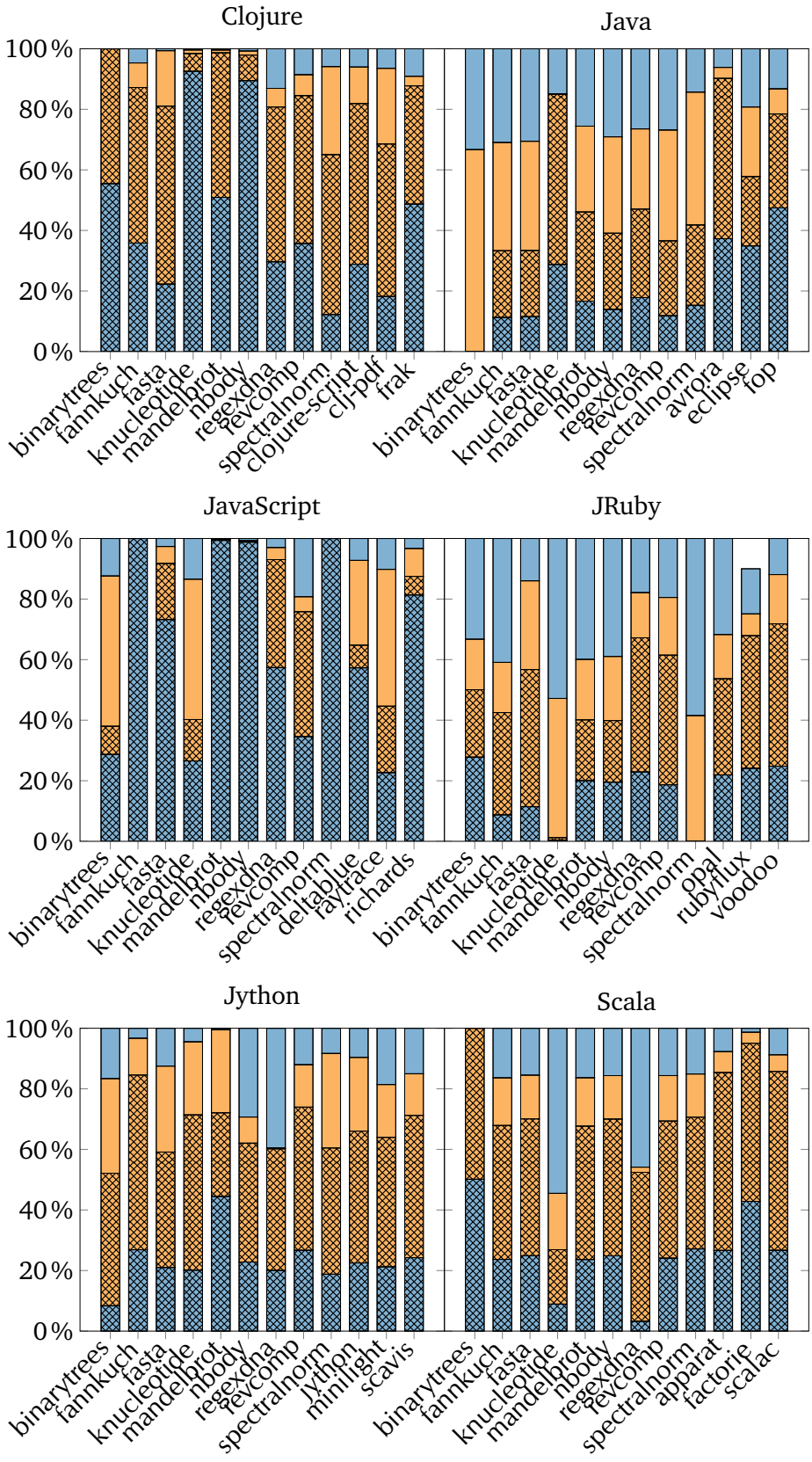

Figure 5.17. Unnecessary (▩,▩) and necessary (□,□) zeroing of primitive (□, ▩) and reference (□,▩) instance fields.

5.2.5 Object Lifetimes

Object sizes and lifetimes characterize a program's memory management "habits" and largely determine the GC workload due to the program's execution. To approximate and analyze the GC behavior, we used ElephantTracks [101] to collect object allocation, field update, and object death traces, and run them through a GC simulator configured for a generational collection scheme with a 4 MiB nursery, and 4 GiB old generation. None of the microbenchmarks allocated enough memory to trigger a full heap (old generation) collection.

The results are summarized in Table 5.3. The *mark* column contains the number of times the GC marked an object live, the *cons* column contains the number of allocated objects, and the *nursery survival* column contains the fraction of allocated objects that survive a nursery collection.

The most striking difference is the number of objects allocated by the dynamic language CLBG benchmarks compared to their Java and Scala counterparts—in all of them, the Java and Scala workloads allocate at least one order or magnitude less objects, and in some cases even several orders less. The results for the statically typed languages, such as Java and Scala are not too surprising—given the microbenchmark nature of the CLBG workloads—but they indicate how inherently costly the dynamic language features are in terms of increased GC workload.

The plots in Figures 5.19-5.23, show the evolution of the object survival rate plotted against logical time expressed as cumulative memory allocated by a benchmark. The Java fannkuch-redux, fasta, mandlebrot, nbody, and spectralnorm benchmarks are not shown, because they allocate less than 1 MiB.

The results show that while the workloads written in the dynamic languages allocate much more objects than their counterparts written in the static languages, most of the objects die young, suggesting that they are mainly temporaries resulting from features specific to dynamic languages.

Table 5.3. Garbage collector workload

Benchmark	Mark	Cons	Mark / Cons	Nursery Survival	Benchmark	Mark	Cons	Mark / Cons	Nursery Survival
Clojure					**JRuby**				
binarytrees	4 021 096	147 910 977	0.03	2.72 %	binarytrees	30 303 133	118 527 001	0.26	7.40 %
fannkuchredux	38 900	317 062	0.12	12.27 %	fannkuchredux	263 722	10 041 379	0.03	2.63 %
fasta	122 560	2 721 195	0.05	4.50 %	fasta	513 005	27 557 549	0.02	1.86 %
knucleotide	1 158 380	18 436 145	0.06	6.28 %	knucleotide	1 788 684	19 762 437	0.09	9.05 %
mandelbrot	134 617	2 822 088	0.05	4.77 %	mandelbrot	81 977 235	200 409 954	0.41	1.81 %
nbody	180 333	5 320 075	0.03	3.39 %	nbody	44 632 312	154 314 308	0.29	1.90 %
regexdna	197 257	586 176	0.34	33.65 %	regexdna	148 158	399 044	0.37	37.13 %
revcomp	60 604	437 639	0.14	13.85 %	revcomp	109 951	635 720	0.17	17.30 %
spectralnorm	563 338	5 592 427	0.10	10.07 %	spectralnorm	9 262 955	113 332 054	0.08	1.93 %
clojure-script	8 658 091	30 656 512	0.28	28.24 %	opal	382 540	2 632 867	0.15	14.53 %
clj-pdf	582 276	1 715 457	0.33	33 %	voodoo	138 121	2 413 103	0.05	5.7 %
frak					rubyflux	234 742	1 183 666	0.19	19.8 %
Java					**Jython**				
binarytrees	1 136 479	29 581 095	0.04	3.84 %	binarytrees	73 870 086	297 905 683	0.25	3.22 %
fannkuchredux	0	2226	0.00	0.00 %	fannkuchredux	44 953 062	225 126 759	0.19	3.6 %
fasta	0	2329	0.00	0.00 %	fasta	340 799	3 111 487	0.11	10.95 %
knucleotide	962 320	967 492	0.99	99.47 %	knucleotide	12 538 292	40 411 395	0.31	31.03 %
mandelbrot	0	3699	0.00	0.00 %	mandelbrot	2 301 641	80 129 997	0.03	2.87 %
nbody	0	2726	0.00	0.00 %	nbody	13 095 284	161 984 760	0.08	2.83 %
regexdna	1295	2698	0.48	48.00 %	regexdna	58 047 534	227 296 897	0.26	3.78 %
revcomp	1158	2848	0.41	40.66 %	revcomp	183 874	1 012 778	0.18	18.16 %
spectralnorm	0	6263	0.00	0.00 %	spectralnorm	6 131 554	140 908 487	0.04	3.29 %
avrora	59 684	2 075 466	0.03	2.88 %	jython	10 654 369	43 752 983	0.24	24.35 %
eclipse	171 766 557	66 569 509	2.58	30.82 %	scavis	59 698	891 738	0.06	6.6 %
fop	486 253	2 982 888	0.16	16.30 %	minilight				
JavaScript					**Scala**				
binarytrees	10 529 660	273 311 395	0.03	2.3 %	binarytrees	84 257	29 607 558	0.002	0.28 %
fannkuchredux	288 337	132 237 781	0.002	0.2 %	fannkuchredux	0	28 209	0	0 %
fasta	231 614	9 603 313	0.02	2.4,%	fasta	0	28 847	0	0 %
knucleotide	2 829 086	26 899 084	0.10	10.5 %	knucleotide	1 205 165	4 340 095	0.27	27 %
mandelbrot	1 049 053	201 508 680	0.005	0.2 %	mandelbrot	0	28 777	0	0 %
nbody	339 056	127 015 472	0.002	0.2 %	nbody	0	29 246	0	0 %
regexdna	90 620	1 805 161	0.05	5.0 %	regexdna	277 584	14 829	0.05	5.3 %
revcomp	53 710	601 282	0.08	8.9 %	revcomp	3150	28 628	0.11	11 %
spectralnorm	295 685	120 057 851	0.002	0.2 %	spectralnorm	0	30 596	0	0 %
deltablue	50 358	3 251 569	0.015	1.5 %	apparat	338 633	8 953 723	0.04	3.78 %
raytrace	162 579	38 186 985	0.004	0.4 %	factorie	6 475 518 895	1 505 398 186	4.30	1.24 %
richards	7933	968 004	0.008	0.8 %	scalac	433 046	19 875 421	0.02105	1.59 %

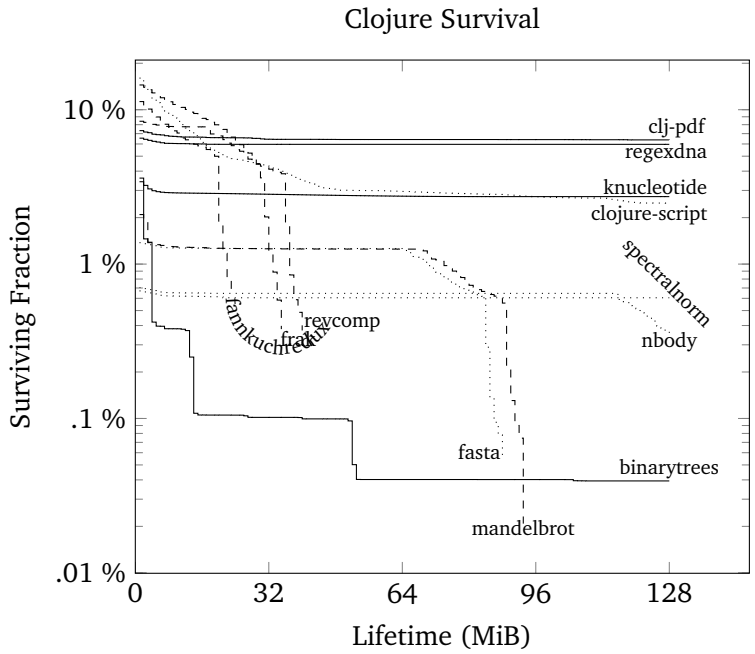

Figure 5.18. Fraction of objects surviving more than a given amount of allocation in the Clojure benchmarks

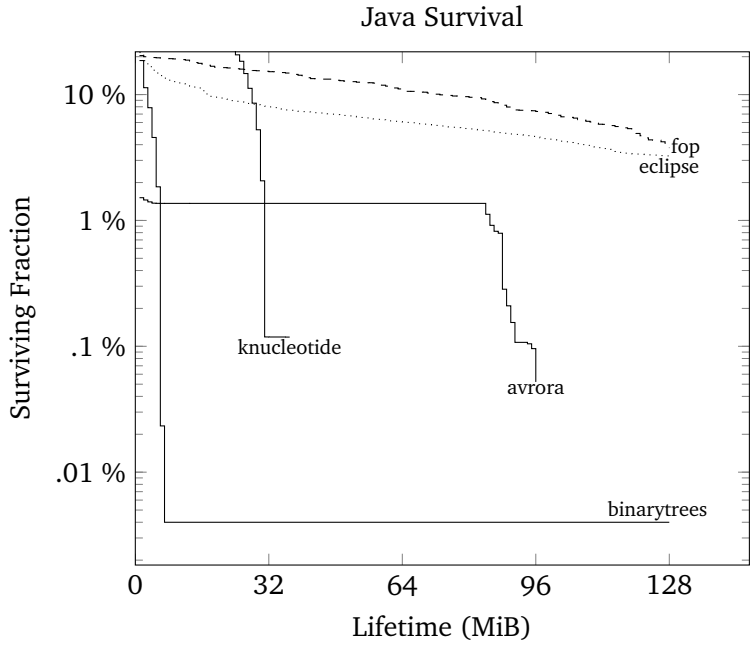

Figure 5.19. Fraction of objects surviving more than a given amount of allocation in the Java benchmarks

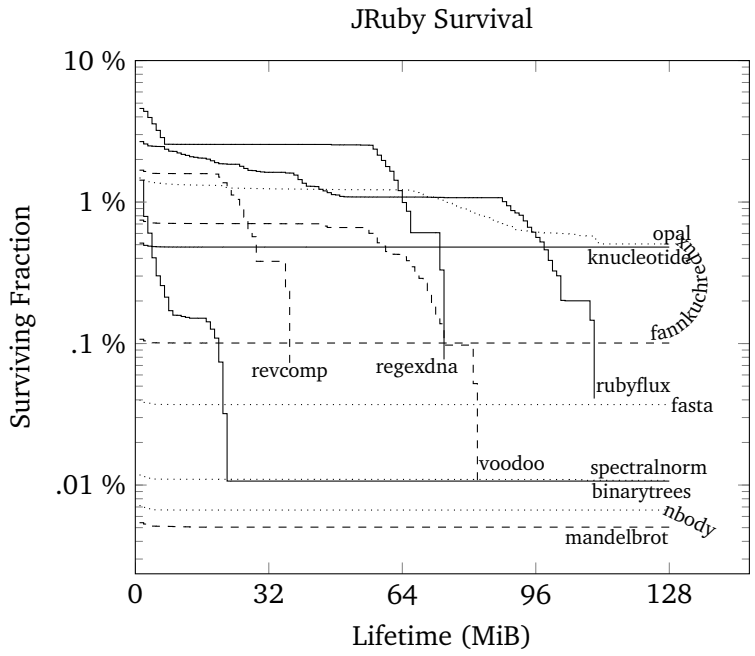

Figure 5.20. Fraction of objects surviving more than a given amount of allocation in the JRuby benchmarks

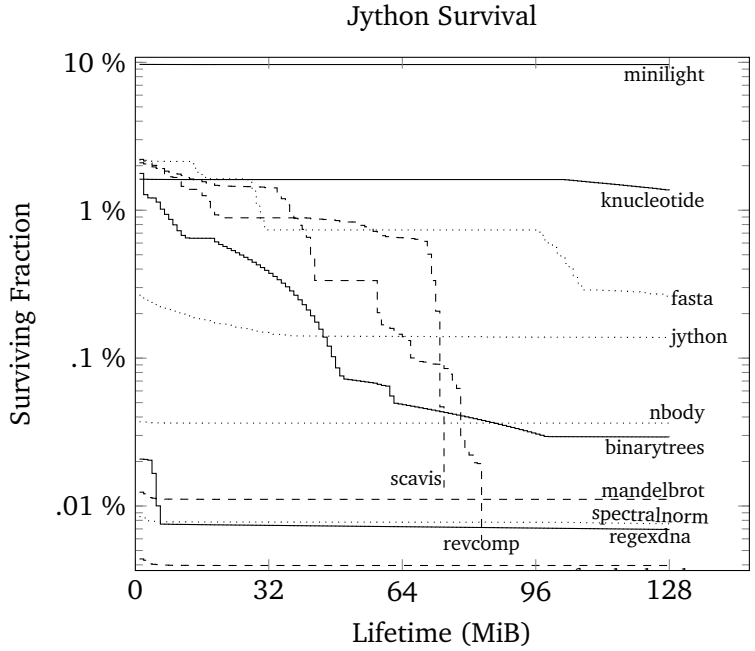

Figure 5.21. Fraction of objects surviving more than a given amount of allocation in the Jython benchmarks

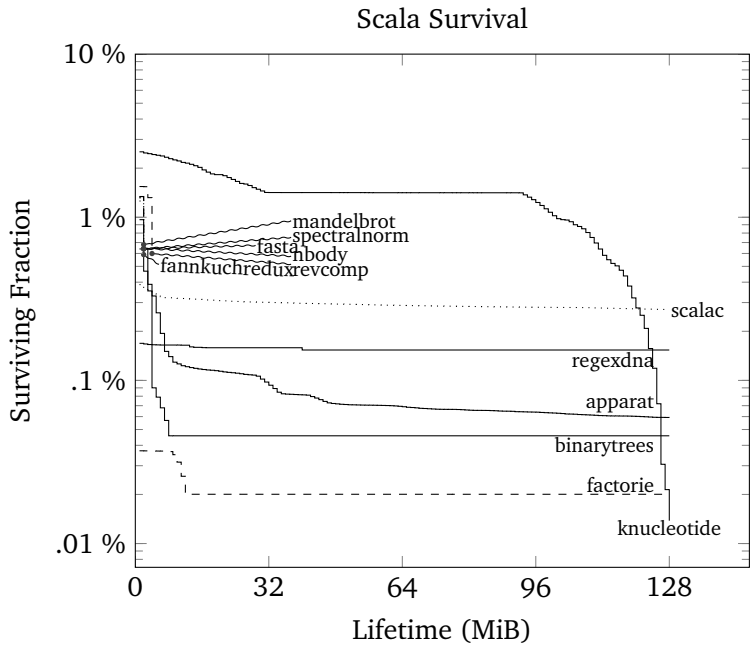

Figure 5.22. Fraction of objects surviving more than a given amount of allocation in the Scala benchmarks

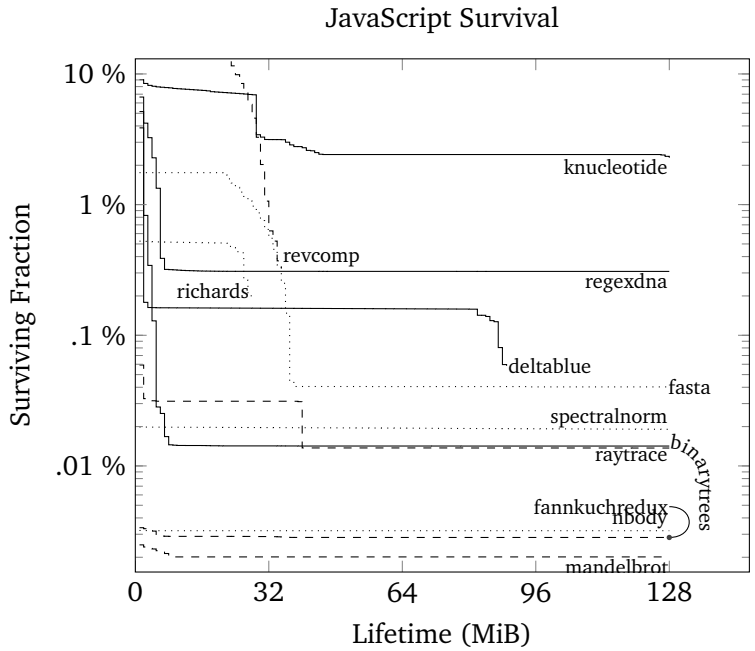

Figure 5.23. Fraction of objects surviving more than a given amount of allocation in the JavaScript benchmarks

5.3 Inlining in the JVM

Method inlining improves performance of programs by removing the overhead of method calls. By expanding the scope for analysis and optimizations in the caller, inlining increases the potential for intra-procedural optimizations and specialization of the code being inlined. In a JVM, inlining is performed by the JIT compiler. Given the importance of inlining, we investigate the ability of the HotSpot JVM to perform inlining with bytecode originating from different JVM languages.

In this section, we first present an overview of the inlining decisions made by the compiler—a result of studying the source code of the server JIT compiler in the HotSpot JVM. Then, having modified the compiler to collect traces of actual inlining decisions while the JVM was executing the workloads from our study, we present metrics calculated using these traces. The metrics provide insights into differences between JVM languages that affect inlining, and include the fraction of inlined and not inlined methods, the fraction of decompiled methods, the distribution of inlining levels, and the distribution of top ten reasons for which inlining did and did not happen.

5.3.1 Inlining Decision

While inlining can facilitate dramatic performance increases, its main effect, code duplication within methods, comes at a certain cost:

- The size of the compiled methods will increase significantly, so that more memory is required to store it. This leads to a larger memory usage and, subsequently, more frequent evictions of compiled methods because of memory pressure.

- Code caches within the CPU will be polluted with multiple versions of the same method, so that the caches also suffer from more memory pressure and more frequent evictions.

- Compilation time increases because of the larger compilation scope. If optimizations within the compiler are not designed carefully, they may exhibit a non-linear growth in complexity.

A JIT compiler therefore needs to constantly make decisions whether to inline, or to not inline a particular method, balancing the costs and benefits of inlining at each call site. Because it is not possible to determine the optimal set of inlined call sites, compilers employ complex heuristics to make the inlining decisions. The heuristic used by the HotSpot server compiler, extracted from the compiler source code in form of a decision tree, is shown in Figure 5.24. The decision tree is basically an expert system applied to static and dynamic information about the code being executed, which tells the compiler whether to *inline* or to *not inline* a method. When considering a specific call

site, the compiler traverses[3] the tree from the root until it reaches a leaf representing a decision.

The compiler first checks whether the target method is a *compiler intrinsic*—a special method that is always inlined. The server compiler differentiates between system intrinsics and method handle intrinsics. In the next step, the compiler checks the usage of strict floating point arithmetics. If either the caller or the callee requires it but the other does not, the call cannot be inlined.

If the call site is polymorphic and the call needs dispatching, the compiler consults the dynamic profiling information to determine whether there is a prominent target for that call site, i.e., whether the call at this particular call site has exactly one or two receivers, or whether there is a major (more frequent) receiver when there are more than two potential targets. If there is no prominent target for this particular call site, the call cannot be inlined and has to undergo virtual method dispatch. Otherwise, the call site is considered for inlining, but is subjected to additional checks to avoid inlining in situations in which the potential performance benefits may not materialize or may be negated by the adverse effects of code duplication.

If the call site was not encountered often enough during profiling, i.e., the call site is considered cold, or if the size of the target method exceeds a certain limit, the decision will be to not inline. These two checks will be skipped if the call site is forcibly inlined or if the call site received many thrown exceptions. Afterwards, simple accessor methods will be always inlined, while for other methods, all of the following conditions must hold for them to be inlined:

- The current method's intermediate representation needs to be below a certain limit.
- The call site needs to have been reached during profiling.
- Inlining must not be disabled.
- The current number of nested inlining scopes needs to be below a certain limit.
- The current number of nested recursive inlining scopes needs to be below a certain limit.
- The sum of the bytecode counts of all inlined methods needs to stay below a certain limit.

To capture the inlining decisions taken during execution, we modified the compiler to collect the inlining decisions for each call site. In the following two sections, we summarize the main reasons for inlining and not inlining methods for each JVM language

[3]The decision tree is actually represented as conditional code in the compiler source code, not an actual data structure. By traversing the tree we mean executing conditional statement in order implied by the decision tree.

Figure 5.24. Decision tree of the HotSpot JVM's JIT compiler.

and workload in our study. Each decision basically represents a path from the root of the decision tree to one of its leaves. To explain why a particular decision was made, we encode this path using a sequence of mnemonic codes that capture the outcome of the conditionals along the path. Each mnemonic code starts with a capital letter so that multiple codes in a sequence can be easily distinguished. Table 5.4 summarizes the individual mnemonic codes, and needs to be consulted when interpreting the results presenting inlining decisions. For example, a combination of mnemonic codes reading "SSmFTsA" encodes a positive inlining decision due to the inlinee being an accessor method (A), in addition to being a synthetic (S), static monomorphic (Sm), and frequently called (F) method of trivial size (Ts).

5.3.2 Top Reasons for Inlining Methods

We first consider the positive inlining decisions. Figure 5.25 shows the distribution of reasons that account for 90% of methods that were inlined. Each color denotes a distinct decision, comprising a combination of reasons. For the sake of readability we filtered out reasons that account for less than 2%, which turned out to be especially pronounced in the case of Clojure. To interpret the results, we cross-reference the legend in Figure 5.25 with Table 5.4.

We observe that in the case of Java, the results for the CLBG workloads are so varied and irregular, that it is impossible to come up with a meaningful interpretation. This is because the CLBG benchmarks are too small for JVM-native language. The situation improves significantly with the real-world workloads, where most inlining applies to frequently called (hot) static methods (both normal and trivial-sized).

In the case of Scala, also a statically typed language, the results for the CLBG workloads appear to suffer from the same problem as in Java—the benchmarks being too small to really exercise the language runtime. Still, they are partially consistent with the results for the real-world workloads. In contrast to Java, we note much more trivial-sized static methods being inlined.

In the case of Clojure, we observe inlining of a large amount of static accessor-style methods and system intrinsics. The CLBG workloads appear similar to the real-world workload, except for higher amount of inlined system intrinsic methods.

In the case of JavaScript, the results for the CLBG and the real-world workloads are consistent, which suggests that a significant amount of JavaScript runtime code gets executed even with microbenchmarks. Most of the inlined methods are static (both normal and trivial-sized), with inlining performed surprisingly even for significant amount of cold call sites and methods. A small amount (but more than with the other languages) of inlined methods was actually virtual, but profiled monomorphic or bimorphic.

In the case of JRuby, we can again observe a certain similarity between the CLBG and the real-world workloads, indicating that significant amount of framework code gets executed even with the microbenchmarks. Interestingly, the CLBG workloads

Code	Description	Code	Description
Contextual Information		Reasons for non-inlining	
Cc	Cold callsite	Ctb	Inlinee already compiled to big method
Pb	Profiled bimorphic	Ctm	Inlinee already compiled to medium method
Pm	Profiled monomophic	Ie	Inlinee was executed infrequently
S	Synthetic method	Mrl	Maximum recursive inlining level reached
Sm	Static monomorphic	Ml	Maximum inlining level reached
Ts	Trivial size	N	Inlinee is native method
Reasons for Inlining		Nmm	No major megamorphic receiver
A	Inlinee is accessor method	Ne	Inlinee was never executed
F	Inlining frequently called method	Tbc	Inlinee too big for cold callsite
Mhi	Inlinee is method handle intrinsic	Tbh	Inlinee to big for hot callsite
Nf	Inlining non-frequent method	Uc	Unreached callsite
Si	Inlinee is system intrinsic		

Table 5.4. Decision codes for inlining and non-inlining reasons.

exhibit inlining of a significant number of synthetic method handle intrinsics compared to the real-world workloads, which we attribute to the relative smaller size of the CLBG workloads. The real-world workloads exhibit a striking similarity among themselves, despite representing rather different tasks (compiler vs. image manipulation).

The results for Jython are similar to JavaScript, which is interesting, because unlike JavaScript, Jython is an interpreter. In contrast to JavaScript, there are more inlined system intrinsic and static accessor methods. As in the case of JavaScript, the results for the CLBG and the real-world workloads are rather similar, indicating that significant amount of framework code gets executed even with microbenchmarks.

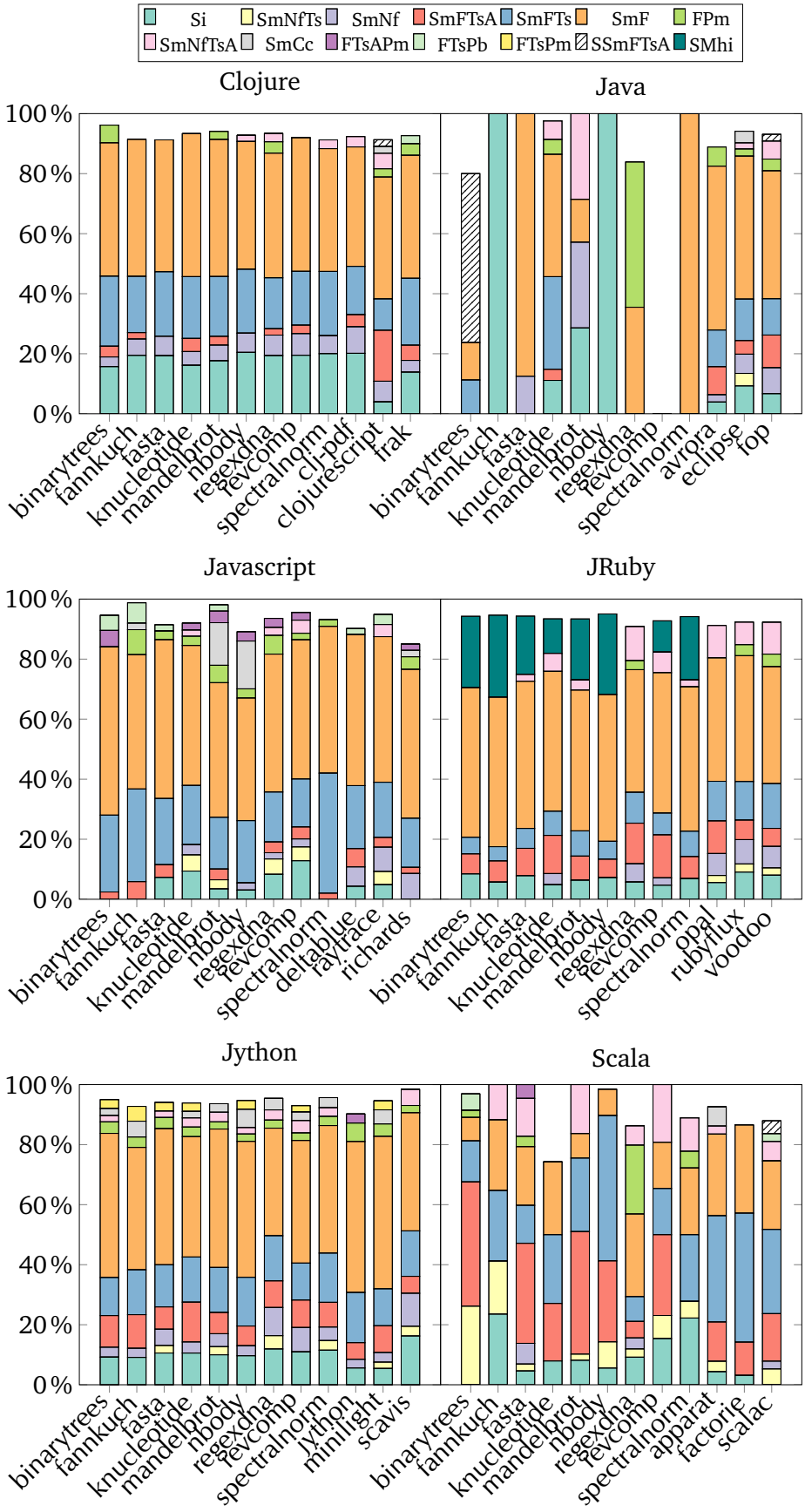

Figure 5.25. Distribution of top 10 reasons for inlining that account for 90% of all inlined methods.

5.3.3 Top Reasons for Not Inlining Methods

We now consider the negative inlining decisions. Figure 5.26 shows the distribution of reasons that account for 90% of methods that were not inlined. To interpret the results, we again cross-reference the legend in Figure 5.26 with Table 5.4.

The results for Java confirm the fact that the CLBG benchmarks are too small for a JVM-native language. However, in the case of the real-world workloads, many methods were not inlined because they have already been compiled to medium-sized or big methods.

The results for Scala again exhibit certain similarity to Java, due to the results for the CLBG workloads significantly deviating from the results for the real-world workloads. The prevailing reason for not inlining methods in the CLBG workloads was that the methods were too big for a cold call site. The reasons get more diverse but somewhat more consistent for the real-world workloads. In addition to methods being too big to inline, we can observe the appearance of polymorphic call sites without a major receiver.

In the case of Clojure, many methods were too big or cold, in addition to being considered at cold call sites. In most workloads, there were polymorphic call sites without a major receiver as well as a significant amount of unreached call sites. The latter is caused by the compiler generating code to handle special cases that never occurred at runtime. There are no striking differences between the CLBG and real-world workloads.

In the case of JavaScript, methods were not inlined mainly because they were too big, and considered at cold or even unreached call sites. Overall, the results appear qualitatively similar to Clojure, except for the different proportions. In contrast to Clojure, there were not much unreached call sites at which hot methods were considered for inlining. The results for the CLBG workloads do not deviate significantly from the results for the real-world workloads.

The results for JRuby are very consistent across all workloads. The prominent reason for not inlining was again the methods being too big for a cold call site, and a significant amount of unreached call sites.

The results for Jython are qualitatively similar to JRuby, again with different proportions between variants of similar reasons (methods being too big to inline), but still quite consistent across all workloads. Compared to JRuby, there is an increased proportion of hot methods considered at unreached call sites.

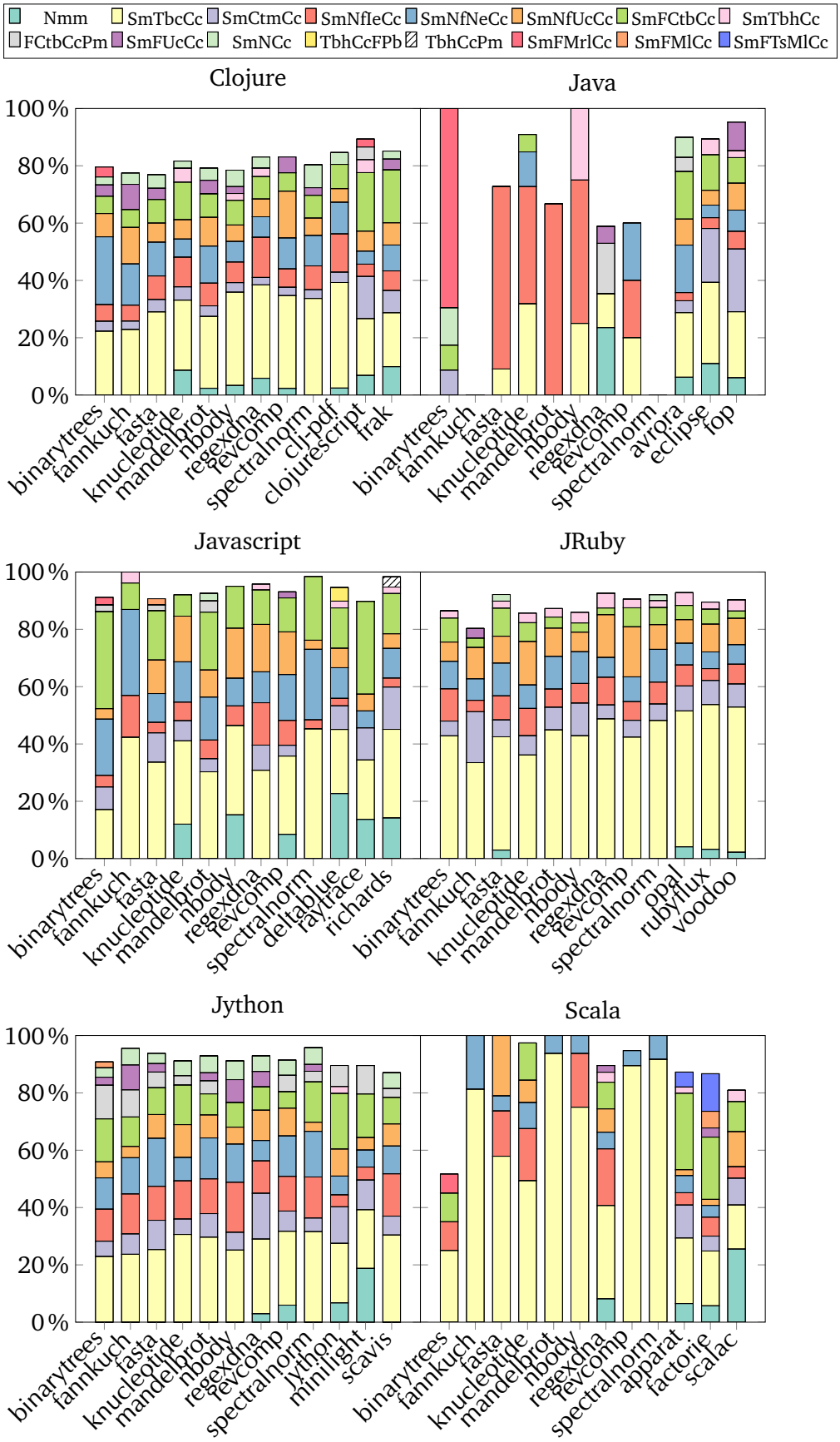

Figure 5.26. Distribution of top 10 reasons for non-inlining that account for 90% of all non-inlined methods.

5.3.4 Inlining Depths

The inlining depth metric represents the number of methods inlined at a given level. Overall, it should provide an approximation of the additional levels of abstractions introduced by language implementations and the VM facilities they use (e.g., `invokedynamic`).

The results of the inlining depth analysis for the CLBG workloads are shown in Figures 5.27-5.32. We observe that JRuby has the deepest inlining structure—this is a symptom of JRuby using `invokedynamic` and the specific `invokedynamic` implementation that adds additional inlining levels. A single call from one JRuby method to another regularly incurs 5-10 inlining levels.

Jython and Clojure use deeply nested static helper functions throughout the generated code, which leads to a large amount of deeply nested inlining. JavaScript shows a smaller amount of inlining, albeit still reaching 10 levels for many of the CLBG workloads.

Scala only adds one or two levels of indirection when compared to the Java code. Scala was designed from the start with execution on top of a JVM in mind, and its language structure can be represented efficiently in Java bytecode. In general, Scala introduces additional layers of abstraction when compared to Java, so that inlining should have a larger influence on the performance of Scala code than it has on Java code. This assumption is supported by the observation made in [120].

The results for Java only show at most four levels of inlining. Since Java does not need abstractions to run on a JVM, this is an approximation of the level of inlining naturally present in the algorithm implemented by a particular CLBG benchmark.

The situation is vastly different in the case of real-world workloads, as shown in Figures 5.27-5.32. In the cases of Java and (especially) Scala, these workloads appear to be much more complex than the CLBG workloads. Interestingly, in the case of JRuby, the real-world workloads appear to be actually less complex than the CLBG workloads. This suggests that the inlining depths reported for the JRuby in the figure are actually more indicative of the workload character than the language implementation.

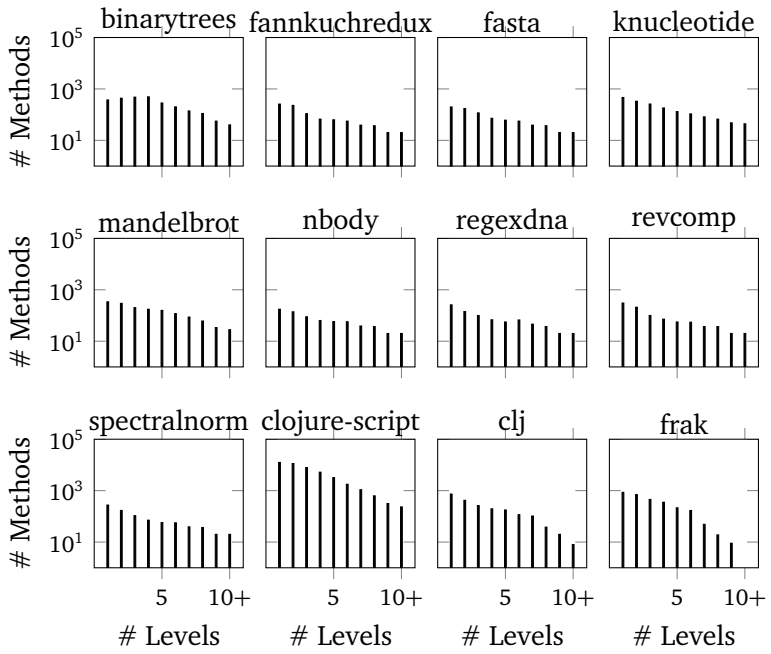

Figure 5.27. The number of methods inlined at a given level for the Clojure benchmarks.

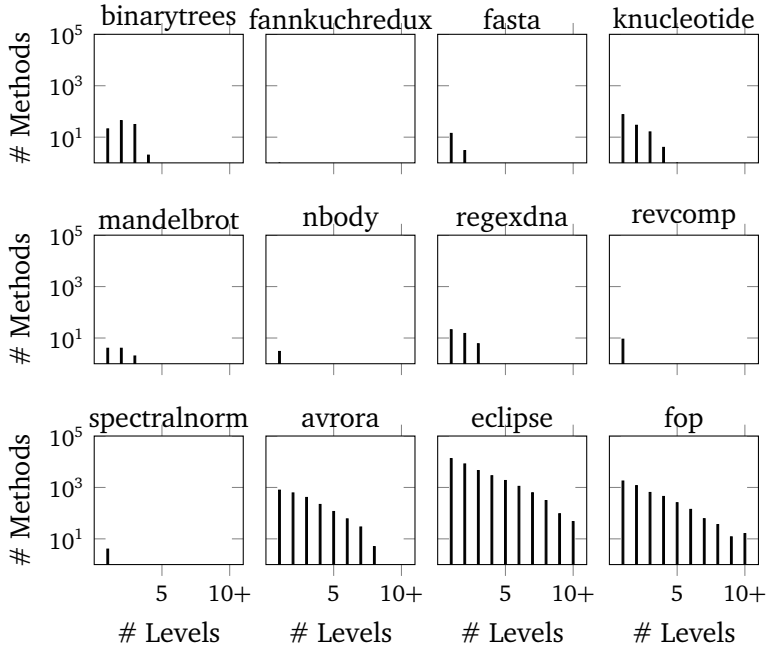

Figure 5.28. The number of methods inlined at a given level for the Java benchmarks.

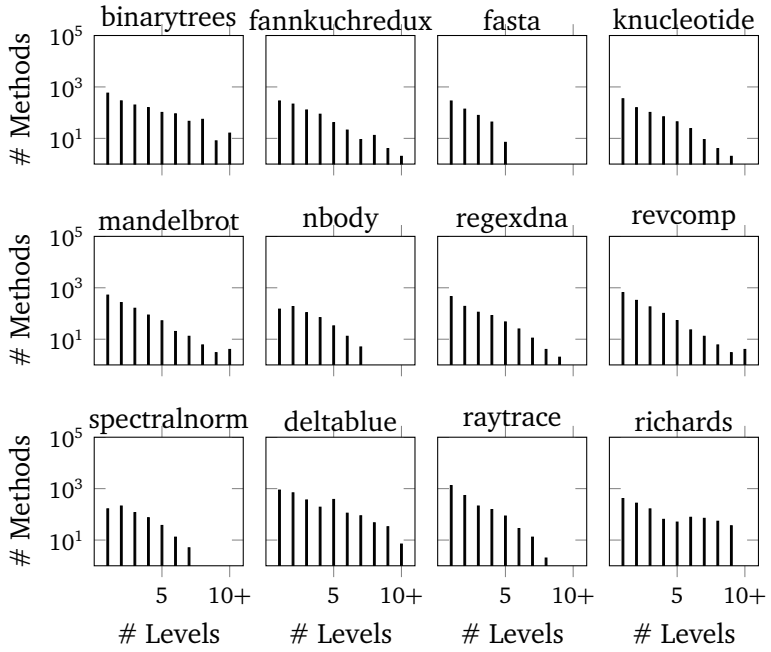

Figure 5.29. The number of methods inlined at a given level for the JavaScript benchmarks.

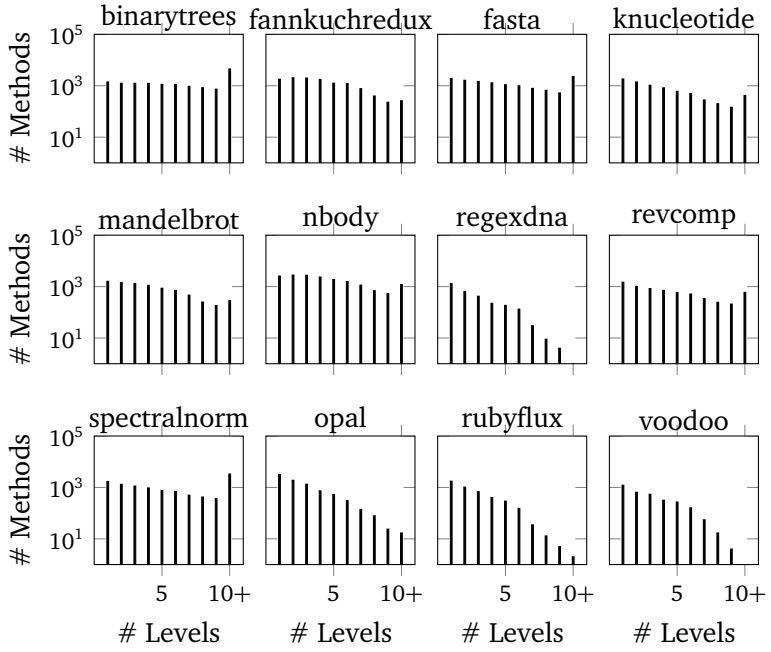

Figure 5.30. The number of methods inlined at a given level for the JRuby benchmarks.

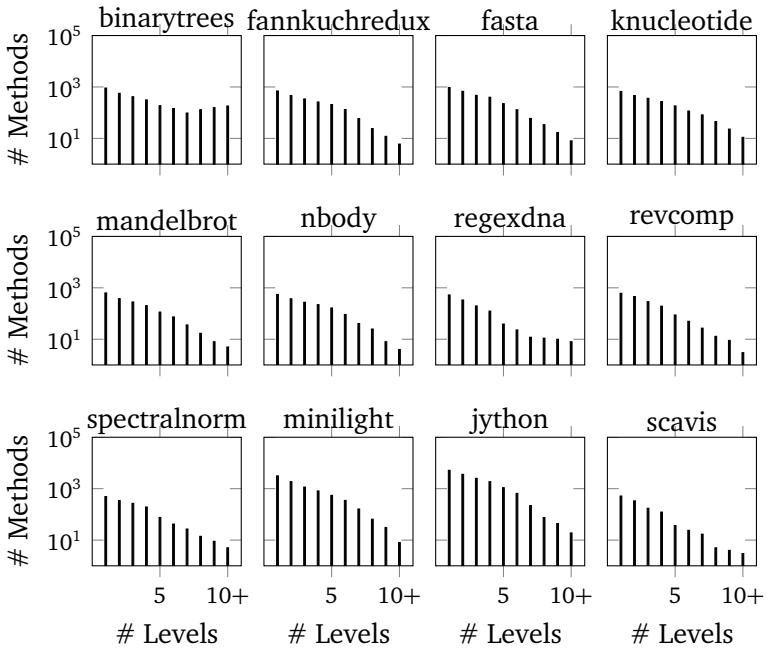

Figure 5.31. The number of methods inlined at a given level for the Jython benchmarks.

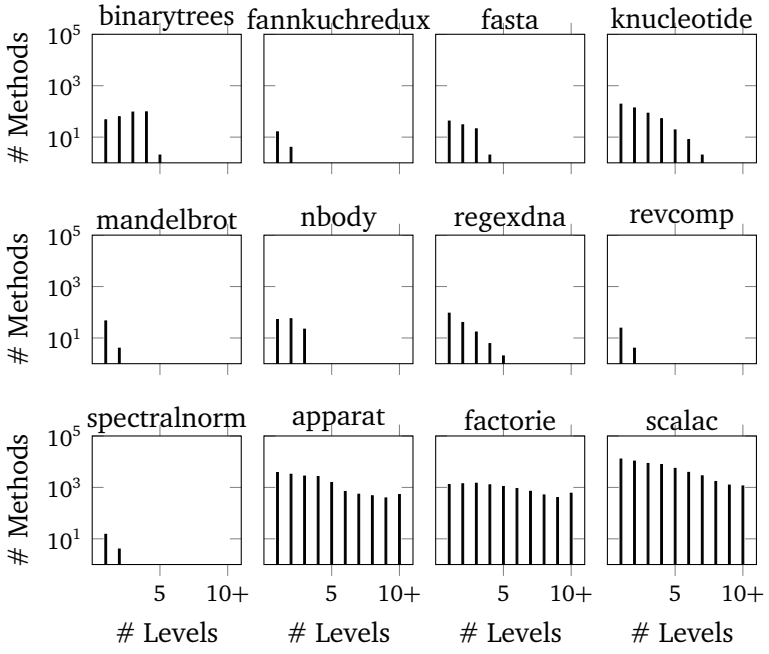

Figure 5.32. The number of methods inlined at a given level for the Scala benchmarks.

5.3.5 Fraction of Inlined Methods

The results in Figure 5.33 show the fraction of inlined methods. Again, we can observe that the results for Java and (to a certain extent) Scala show significant variance, suggesting that the CLBG workloads for these two languages are rather small and not representative enough. The results for Java are more indicative of the workload algorithm structure, while the results for other languages show primarily the inlining behavior of the language runtime.

The use of the `invokedynamic` bytecode instruction in JRuby leads to many successful inlinings, which correlates with the high inlining depths presented earlier. The results for JavaScript show a rather high proportion of unsuccessful inlining attempts, which hints at the generated code calling large helper methods—this is supported by the breakdown of reasons for not inlining methods in Section 5.3.3. The results for the real-world workloads in Scala show the largest fraction of successful inlinings, which again correlates with the high inlining depths observed earlier.

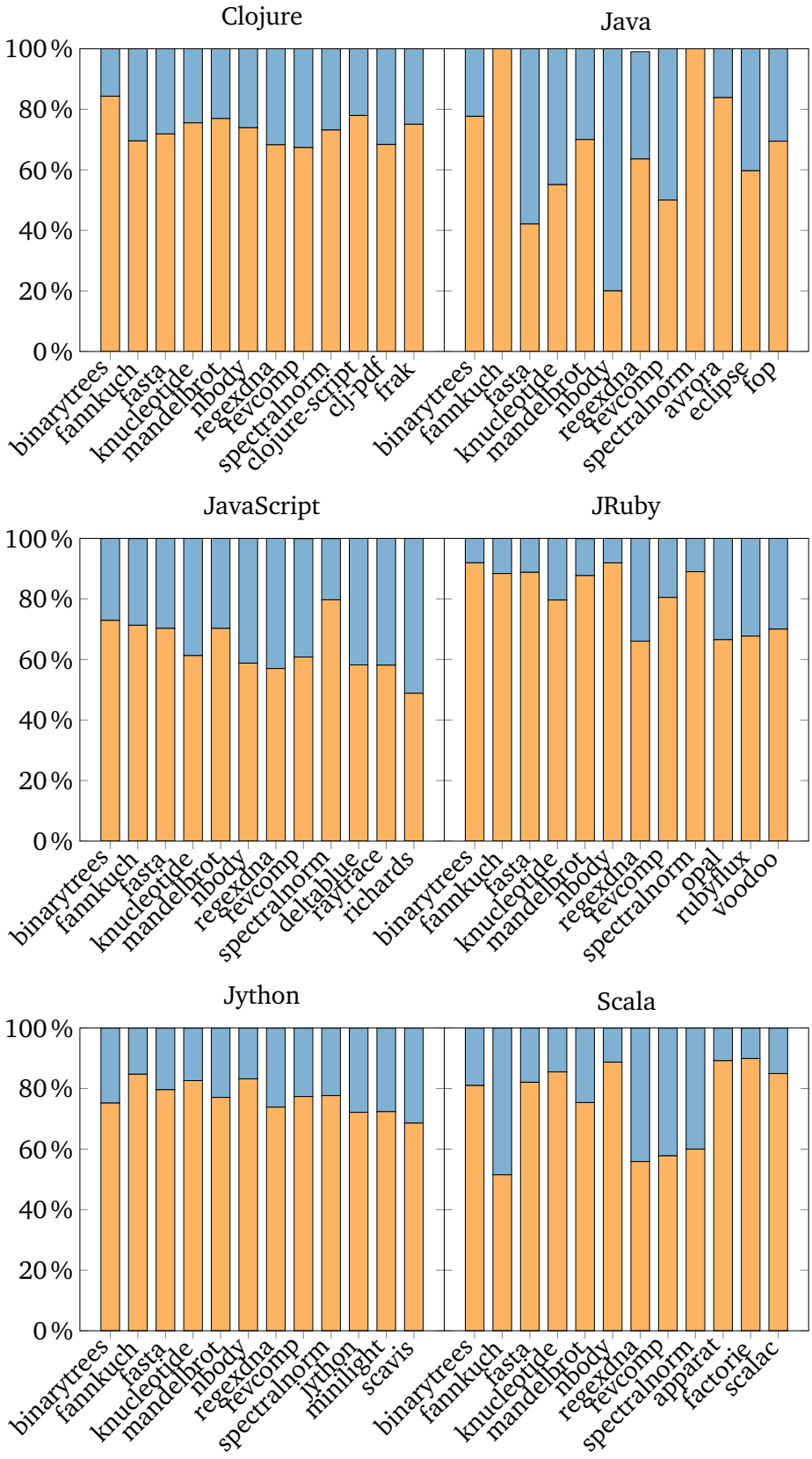

Figure 5.33. Fraction of inlined (■) and not inlined (■) methods.

5.3.6 Fraction of Decompiled Methods

To achieve high performance, the JVM sometimes performs aggressive optimizations that are based on rather optimistic assumptions [131]. This includes static assumptions about the system's state (e.g., the hierarchy of loaded classes), and dynamic assumptions about the behavior of the application (e.g., unreached branches within the application code). When the assumptions that a JIT compiler made when compiling a method do not or cease to hold, the method will be invalidated and its compiled version discarded. The fraction of decompiled methods metric shows how good the JVM is in making assumptions about the code it executes.

The results of this analysis are shown in Figure 5.34. The significant variations in the results for Java and Scala again suggest that the CLBG workloads fail to stress the language runtime of these languages. The results for JRuby, Jython and Clojure are very consistent. With the CLBG workloads, Clojure exhibits an increased amount of decompiled methods compared to the other two languages, but the difference is less pronounced with the real-world workloads. In the case of JavaScript, there are benchmarks with no decompilation whatsoever, but there are also many cases where a large fraction of methods gets decompiled. This suggests that assumptions made for the JavaScript workloads often do not hold during the lifetime of the application.

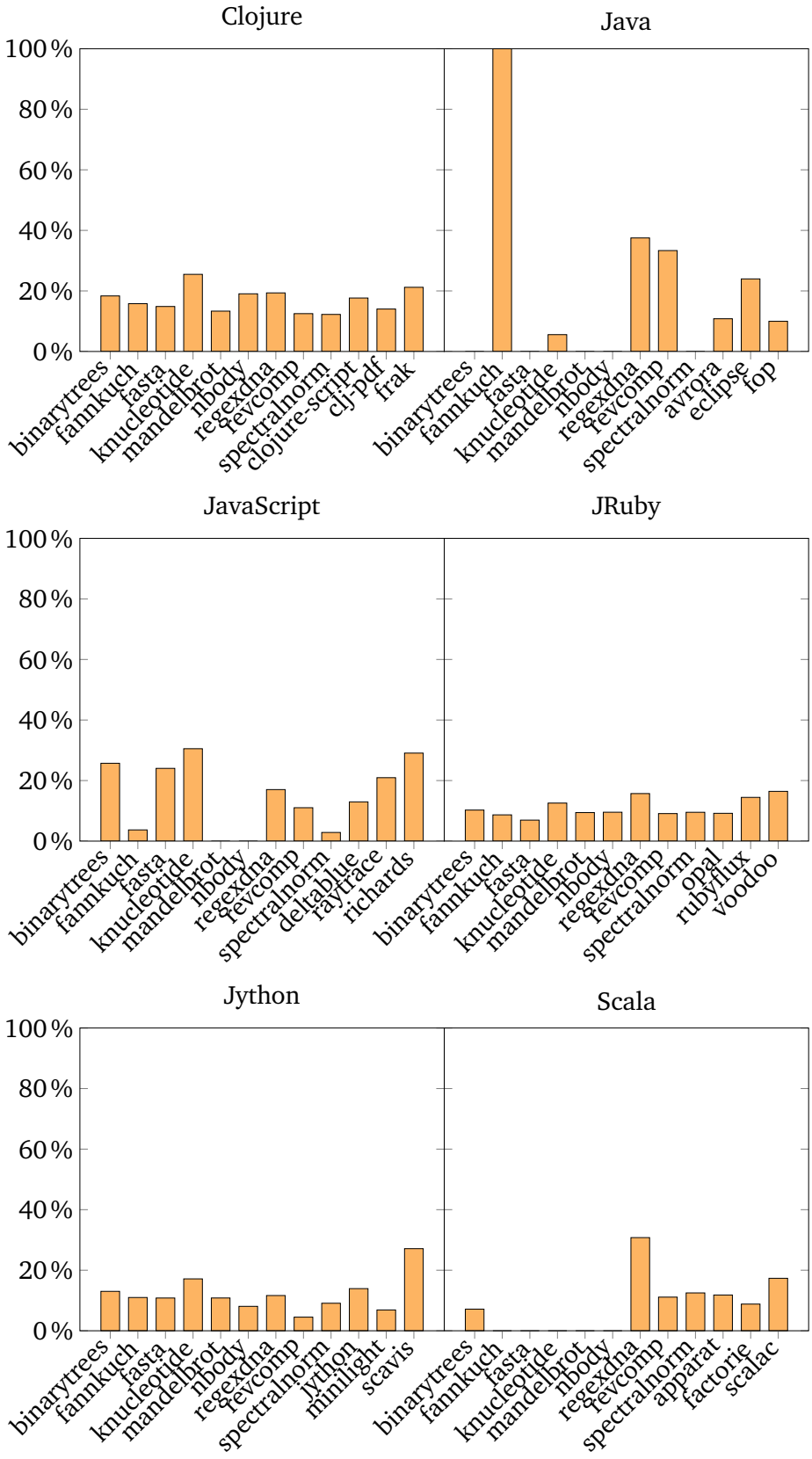

Figure 5.34. Fraction of decompiled methods.

5.3.7 Summary

There is a noticable trend in the results for dynamic languages: the microbenchmarks from CLBG apparently manage to exercise a significant amount of the language runtime code, resulting in workloads that do not differ much for real-world workloads. However, the situation is vastly different with the statically typed languages, i.e., Java and Scala. The microbenchmark nature of the CLBG workloads is clearly evident in all the metrics, especially when compared to the results for the real-world workloads. This means that conclusions concerning statically typed JVM languages can be misleading when based solely on observations of microbenchmark behavior. This is consistent with general benchmarking practices. The interesting result is that this is not necessarily the case with dynamic JVM languages, which include a significant amount of language runtime and framework code in their execution.

The primary reasons for inlining methods vary between the languages, but remain rather consistent for workloads in the same language. The majority of inlined methods are either static, where the inlining decision boils down to method size and other properties, or intrinsic. Inlining of virtual methods is much less common, and was more visible only in the case of JavaScript, where the JVM managed to inline some methods profiled monomorphic, bimorphic, and megamorphic with a major receiver.

The primary reasons for not inlining methods were mostly method sizes in conjunction with cold or even unreached call sites generated by dynamic language compilers. Qualitatively, the results for many dynamic languages were similar, with different proportions between various decisions. Each language exhibited a specific set of negative inlining decisions, but these tended to be in minority compared to negative decisions related to method sizes.

The results of the inlining depth analysis suggest the dynamic languages employ additional level of abstractions to execute the workload code, with JRuby having the deepest inlining structure. Compared to other dynamic languages, JavaScript appears to be somewhat inlining-unfriendly and unpredictable for the JIT compiler, as evidenced by the cases with high ratio of decompiled methods.

Chapter 6

Conclusions

The introduction of scripting support and support for dynamically-typed languages to the Java platform made the JVM and its runtime library an attractive target for the developers of new dynamic programming languages. Those are typically more expressive—trading raw performance of the statically-typed languages for increased developer productivity—and by targeting the JVM, their authors hope to gain the performance and maturity of the Java platform, while enjoying the benefits of the dynamic languages. However, the optimizations found in JVM implementations have been mostly tuned with Java in mind, therefore the sought-after benefits do not automatically come just from running on the JVM.

In this dissertation, we performed workload characterization for 72 different workloads produced by benchmarks and applications written in Java, Scala, Clojure, Jython, JavaScript, and JRuby. Using our workload characterization suite [106], we collected and analyzed terabytes of data resulting from weeks of running experiments with the aim to contribute to the understanding of the characteristics of workloads produced by static and dynamic languages executing on the JVM. Due to the lack of a proper benchmarking suite for dynamic languages, we opted, like Li et al. [75] before us, to use the benchmarks from the CLBG project augmented with several real-world applications as the workloads for our study. Moreover, we had a close look at one of the most effective compiler optimizations, namely method inlining. We have shown the decision tree of the HotSpot JVM's JIT compiler and gathered statistics about the ability of the JVM to inline for different JVM workloads. Here we summarize the findings of our study:

Call-site Polymorphism. Despite high number of polymorphic call-sites targeting multiple methods, a very high percentage of method invocations actually happens at sites that only target a single method.

Field, Object, and Class Immutability. The dynamic languages use a significant amount of immutable classes and objects.

Object Lifetimes. Compared to static languages, the dynamic language workloads allocate significantly more objects, but most of them do not live for long, which suggests that many temporaries are used (often resulting from unnecessary boxing and unboxing of primitive types).

Unnecessary Zeroing. The dynamic languages (especially Clojure and Jython) exhibit a significant amount of unnecessary zeroing. This correlates with the significant amount of short-lived immutable objects allocated by the respective dynamic language workloads.

Identity Hash-code Usage. All the workloads use the identity hash code very scarcely, suggesting that object header compression with lazy handling of identity hash code storage is an appropriate heuristic for reducing object memory and cache footprint.

Inlining. Java and Scala behave very different from the dynamic languages in every metric. The runtime system that dynamic languages have to employ heavily influences the code the JIT sees. In particular, the JVM is bad in handling the JavaScript workloads.

6.1 Summary of Contributions

In the following we summarize the contributions of this dissertation.

Rapid Development of Dynamic Program Analysis Tools. In this dissertation we performed a thorough evaluation of bytecode instrumentation frameworks for developing dynamic program analysis tools. In this regard, we conducted a controlled experiment to compare DiSL with ASM, measuring development time and correctness of the developed tools. Secondly, we recasted 10 open-source software engineering tools with DiSL, showing quantitative evidence that DiSL-based tools are (i) considerably more concise than equivalent tools based on ASM and (ii) only slightly more verbose than equivalent tools implemented in AspectJ. Moreover, DiSL aims at reconciling (iii) efficiency of the generated code to ensure good tool performance.

Toolchain for Workload Characterization. We introduced a new set of dynamic platform-independent metrics that capture differing properties of workloads running on the JVM. We developed a toolchain based on a unified infrastructure which is JVM-portable, offers non-prohibitive runtime overhead with near-complete bytecode coverage, and can compute a full suite of metrics “out of the box”. Among the metrics of interest to be collected by our toolchain are object allocations, method and basic block hotness, the degree of call-site polymorphism, stack usage

and recursion depth, instruction mix, use of immutability and synchronization, amount of unnecessary zeroing, and use of hash code.

Workload Characterization of JVM Languages. We performed a thorough evaluation of six JVM languages on micro-benchmarks and real-world applications. We collected a full range of dynamic metrics with our toolchain and identified differing properties imposed by Java and non-Java workloads. We elaborated on our findings and spotted cases for potential optimization.

Inlining in the JVM. We investigated the ability of the JVM to perform a fundamental optimization—inlining—on workloads written in different JVM languages. We revealed the HotSpot JVM server compiler's inlining heuristics. We presented an analysis of JVM inlining behavior and performed the first in-depth investigation for six JVM languages.

6.2 Future Work

The work presented in this dissertation opens several future work directions. In the following we give an overview of future research plans:

Empirical evaluation of a broad range of bytecode instrumentation frameworks. We believe that more comprehensive evaluation of bytecode instrumentation frameworks will be valuable for the community. In this dissertation we considered only three tools, ASM, AspectJ, and DiSL. Possible future work would be a study involving more instrumentation frameworks both offering high-level and low-level approaches for performing instrumentations, such as Chord, Soot, Javassist, Josh and others.

Introduction of new metrics. Dynamic metrics that we presented in this disseration serve a specific problem, namely tuning a multi-language infrastructure. Therefore, all the metrics were collected at the bytecode level. However, it would be interesting to extend our toolchain for collecting metrics specific to certain non-Java languages, such as JavaScript. Among the metrics of interest could be prototype-based object creation, field additions, and so on.

In-depth analysis of optimization opportunities. In this dissertation we performed a first step towards understanding the differing properties of JVM workloads. We characterized a full range of JVM workloads, however we did not perform any optimization. We left the optimization decision making as a task for JVM implementers and developers of JVM language compilers. Possible future direction would be to incorporate our findings and perform the optimizations either at the language compiler level or at the JVM level.

Workload characterization of languages that run on .NET. The study presented in this dissertation was focused on the JVM, however the methodology can be generalized to the case of .NET (CLR and DLR), although, the instrumentation would be slightly different. Therefore, workload characterization of languages that run on .NET will shed a light on how different the platforms are. Such a study would be of great importance to the developers of JVM language compilers for specializing their compilers to the specific back-end (i.e., JVM or .NET).

Bibliography

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, Jan. 2000.

[2] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl. Continuous profiling: Where have all the cycles gone? In *ACM Transactions on Computer Systems*, pages 1–14, 1997.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM.

[5] M. Arnold, M. Hind, and B. G. Ryder. Online feedback-directed optimization of java. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 111–129, New York, NY, USA, 2002. ACM.

[6] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings European Conference on Object-Oriented Programming*, volume 5142 of *LNCS*, pages 542–565. Springer-Verlag, 2008.

[7] G. Ausiello, C. Demetrescu, I. Finocchi, and D. Firmani. k-calling context profiling. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '12, pages 867–878, New York, NY, USA, 2012. ACM.

[8] D. F. Bacon, S. J. Fink, and D. Grove. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 111–132, London, UK, 2002. Springer-Verlag.

[9] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '98, pages 258–268, New York, NY, USA, 1998. ACM.

[10] H. E. Bal. A comparative study of five parallel programming languages. In *Distributed Open Systems*, pages 209–228. IEEE, 1991.

[11] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM.

[12] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings Symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society.

[13] R. Barik and V. Sarkar. Interprocedural load elimination for dynamic optimization of parallel programs. In *Proceedings Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 41–52, Washington, DC, USA, 2009. IEEE Computer Society.

[14] W. Binder, J. Hulaas, and P. Moret. A quantitative evaluation of the contribution of native code to Java workloads. In *Proceedings Symposium on Workload Characterization*, pages 201–209, 2006.

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

[16] W. Binder, A. Villazón, D. Ansaloni, and P. Moret. @J - Towards rapid development of dynamic analysis tools for the Java Virtual Machine. In *Proceedings Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 1–9. ACM, 2009.

[17] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, 2006.

[18] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *Proceedings International Symposium on Software Testing and Analysis*, ISSTA '08, pages 155–165. ACM, 2008.

[19] E. Bodden and K. Havelund. Aspect-oriented race detection in Java. *IEEE Transactions on Software Engineering*, 36(4):509–527, July 2010.

[20] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings Conference on Software Engineering*, ICSE '11, pages 241–250. ACM, 2011.

[21] J. Bogda and U. Hölzle. Removing unnecessary synchronization in Java. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 35–46, New York, NY, USA, 1999. ACM.

[22] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, 2009.

[23] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society.

[24] J. M. Bull, L. A. Smith, M. D. Westhead, D. S. Henty, and R. A. Davey. A methodology for benchmarking Java Grande applications. In *Proceedings Conference on Java Grande*, pages 81–88. ACM Press, 1999.

[25] J. Castanos, D. Edelsohn, K. Ishizaki, P. Nagpurkar, T. Nakatani, T. Ogasawara, and P. Wu. On the benefits and pitfalls of extending a statically typed language JIT compiler for dynamic scripting languages. In *Proceedings Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 195–212, 2012.

[26] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *Proceedings Conference on Software Engineering*, ICSE '08, pages 221–230. ACM, 2008.

[27] K. Chen and J.-B. Chen. Aspect-based instrumentation for locating memory leaks in Java programs. In *Proceedings Conference on Computer Software and Applications*, COMPSAC '07, pages 23–28, Washington, DC, USA, 2007. IEEE Computer Society.

[28] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[29] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *Proceedings Conference on Aspect-Oriented Software Development*, AOSD '04, pages 102–111. ACM, 2004.

[30] J.-D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Stack allocation and synchronization optimizations for Java using escape analysis. *ACM Transactions on Programing Languages and Systems*, 25(6):876–910, Nov. 2003.

[31] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '00, pages 13–26, New York, NY, USA, 2000. ACM.

[32] CLBG. The Computer Language Benchmarks Game. Web pages at `http://benchmarksgame.alioth.debian.org/`, 2014.

[33] clj-pdf. A library for generating PDFs from Clojure. Web pages at `https://github.com/yogthos/clj-pdf`, 2014.

[34] clojure-script. Clojure to JavaScript compiler. Web pages at `https://github.com/clojure/clojurescript`, 2014.

[35] R. Cohn and P. G. Lowney. Feedback directed optimization in compaq's compilation tools for alpha. In *Proceedings Workshop on Feedback Directed Optimization*, pages 3–12, 1999.

[36] C. Consel, L. Hornof, R. Marlet, G. Muller, S. Thibault, E. n. Volanschi, J. Lawall, and J. Noye. Tempo: Specializing systems applications and beyond. *ACM Computing Surveys, Symposium on Partial Evaluation*, 30, 1998.

[37] M. Dahm. Byte code engineering. In *Java-Information-Tage 1999 (JIT'99)*, Sept. 1999. `http://jakarta.apache.org/bcel/`.

[38] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic Java virtual machine analysis: the Java Grande forum benchmark suite. In *Proceedings Conference on Java Grande*, pages 106–115. ACM Press, 2001.

[39] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings European Conference on Object-oriented Programming*, ECOOP '99, pages 258–278, 1999.

[40] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Proceedings Conference on Program Comprehension*, ICPC '07, pages 281–285. IEEE CS, 2007.

[41] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '03, pages 149–168, 2003.

[42] B. Dufour, L. Hendren, and C. Verbrugge. *J: a tool for dynamic analysis of Java programs. In *Proceedings Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '03, pages 306–307, New York, NY, USA, 2003. ACM.

[43] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *Proceedings Symposium on Foundations of Software Engineering*, FSE '08, pages 59–70, New York, NY, USA, 2008. ACM.

[44] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Proceedings Conference on Software Composition*, volume 4829 of *LNCS*, pages 200–215. Springer-Verlag, 2007.

[45] A. Eichenberger and S. Lobo. Efficient edge profiling for ILP-processors. In *Proceedings Conference on Parallel Architectures and Compilation Techniques*, pages 294–303, Oct 1998.

[46] M. A. Ertl, C. Thalinger, and A. Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4(1):25–32, 2006.

[47] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, 2010.

[48] frak. Regular expressions generator in Clojure. Web pages at `https://github.com/noprompt/frak`, 2014.

[49] D. H. Friendly, S. J. Patel, and Y. N. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proceedings Symposium on Microarchitecture*, MICRO 31, pages 173–181, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[50] R. Garcia, J. Jarvi, A. Lumsdaine, J. G. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proceedings Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '03, pages 115–134, New York, NY, USA, 2003. ACM.

[51] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Longman Publishing Co., Inc., 2006.

[52] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2005.

[53] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Annotation-directed run-time specialization in c. In *Proceedings Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '97, pages 163–178, New York, NY, USA, 1997. ACM.

[54] D. Gregg, M. A. Ertl, and A. Krall. Implementing an efficient Java interpreter. In V. Getov and G. K. Thiruvathukal, editors, *HPCN'01, Java in High Performance Computing*, LNCS 2110, pages 613–620, Amsterdam, June 2001. Springer.

[55] D. Gregg, J. Power, and J. Waldron. A method-level comparison of the Java grande and SPECjvm98 benchmark suites: Research articles. *Concurrency and Computation: Practice and Experience*, 17:757–773, June 2005.

[56] J. Ha, M. Arnold, S. M. Blackburn, and K. S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings Conference on Object Oriented Programming, Systems Languages and Applications*, OOPSLA '09, pages 155–174, New York, NY, USA, 2009. ACM.

[57] N. M. Hanish and W. Cohen. Hardware support for profiling Java programs. In *Workshop on Hardware Support for Objects And Microarchitectures for Java*, 1999.

[58] S. M. Henry and M. C. Humphrey. Comparison of an object-oriented programming language to a procedural programming language for effectiveness in program maintenance. Technical report, Virginia Polytechnic Institute, Blacksburg, VA, USA, 1988.

[59] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Error-free garbage collection traces: how to cheat and not get caught. In *Proceedings Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '02, pages 140–151, New York, NY, USA, 2002. ACM.

[60] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *Proceedings European Conference on Object-Oriented Programming*, ECOOP '91, pages 21–38. Springer-Verlag, 1991.

[61] R. J. Hookway and M. A. Herdeg. Digital fx!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, Jan. 1997.

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

[63] K. Hoste and L. Eeckhout. Cole: Compiler optimization level exploration. In *Proceedings Symposium on Code Generation and Optimization*, CGO '08, pages 165–174, New York, NY, USA, 2008. ACM.

[64] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1-2):229–248, May 1993.

[65] IBM. Dynamic Load-time Instrumentation Library for Java (Dila). Web pages at `http://wala.sourceforge.net/wiki/index.php/GettingStarted:wala.dila`.

[66] IBM. Shrike Bytecode Instrumentation Library. Web pages at `http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview`.

[67] IBM. Watson Libraries for Analysis (WALA). Web pages at `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[68] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 182–196, 2006.

[69] M. Jovic, A. Adamoli, and M. Hauswirth. Catch me if you can: performance bug detection in the wild. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '11, pages 155–170, New York, NY, USA, 2011. ACM.

[70] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, June 1997.

[71] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting rapid development of dynamic program analyses for Java. In *Companion Proceedings Conference on Software Engineering*, ICSE '07, pages 51–52. IEEE CS, 2007.

[72] J. Lamping, G. Kiczales, L. H. Rodriguez, Jr., and E. Ruf. An architecture for an open compiler. In *Proceedings Workshop on Reflection and Meta-level Architectures*, pages 95–106. Morgan Kauffman, 1992.

[73] J. R. Larus and E. Schnarr. EEL: machine-independent executable editing. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '95, pages 291–300, New York, NY, USA, 1995. ACM.

[74] G. Lashari and S. Srinivas. Characterizing Java application performance. In *Proceedings Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 22–26, Washington, DC, USA, 2003. IEEE Computer Society.

[75] W. H. Li, J. Singer, and D. White. JVM-Hosted Languages: They talk the talk, but do they walk the walk? In *Proceedings Conference on Principles and Practices of Programming on the Java platform: Virtual Machines, Languages, and Tools.*, PPPJ '13, pages 101–112. ACM, 2013.

[76] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 1999.

[77] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[78] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. Javana: a system for building customized Java program analysis tools. In *Proceedings Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 153–168, New York, NY, USA, 2006. ACM.

[79] J. Maebe, M. Ronsse, and K. D. Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings Workshop on Binary Translation*, WBT '02, 2002.

[80] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proceedings Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, New York, NY, USA, 2012. ACM.

[81] S. I. W. MASS. *Evaluation of ALGOL 68, Jovial J3B, PASCAL, SIMULA 67, and TACPOL versus TINMAN requirements for a common high order programming language*. Defense Technical Information Center, 1976.

[82] minilight. Minimal global illumination renderer. Web pages at `https://github.com/brunonery/minilight-python`, 2014.

[83] P. Molnar, A. Krall, and F. Brandner. Stack allocation of objects in the CACAO virtual machine. In B. Stephenson and C. W. Probst, editors, *International Conference on Principles and Practice of Programming in Java*, pages 153–161, Calgary, Canada, August 2009. ACM.

[84] NetBeans. The NetBeans Profiler Project. Web pages at `http://profiler.netbeans.org/`, 2013.

[85] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[86] octane-benchmark. The JavaScript Benchmark Suite for the Modern Web. Web pages at `https://developers.google.com/octane/`, 2014.

[87] K. O'Hair. HPROF: A Heap/CPU Profiling Tool in J2SE 5.0, 2004.

[88] opal. Ruby to JavaScript compiler. Web pages at `https://github.com/opal/opal`, 2014.

[89] Oracle Corp. JVM Tool Interface (JVMTI), version 1.2, 2007.

[90] J. K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, Mar. 1998.

[91] OW2 Consortium. ASM – A Java bytecode engineering library. Web pages at `http://asm.ow2.org/`.

[92] M. Paleczny, C. Vick, and C. Click. The Java Hotspot Server Compiler. In *Proceedings Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 1–1, Berkeley, CA, USA, 2001. USENIX Association.

[93] R. Pawlak. Spoon: Compile-time annotation processing for middleware. *IEEE Distributed Systems Online*, 7(11):1, 2006.

[94] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007.

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

[96] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. *Concurrency and Computation: Practice and Experience*, 17(5–6):639–662, April/May 2005. Special Issue: Java Grande/ISCOPE 2002.

[97] K. Pettis, R. C. Hansen, and J. W. Davidson. Profile guided code positioning. *SIGPLAN Notices*, 39(4):398–411, Apr. 2004.

[98] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *Proceedings Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 171–181, New York, NY, USA, 2011. ACM.

[99] L. Prechelt. An empirical comparison of seven programming languages. *Computer*, 33(10):23 –29, oct 2000.

[100] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMether: comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings Conference on Web Application Development*, WebApps '10, pages 27–38, Berkeley, CA, USA, 2010. USENIX.

[101] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: generating program traces with object death records. In *Proceedings Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 139–142, New York, NY, USA, 2011. ACM.

[102] G. Richards, S. Lebresne, B. Burg, and J. Vitek. An analysis of the dynamic behavior of JavaScript programs. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM.

[103] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith. Trace processors. In *Proceedings Symposium on Microarchitecture*, pages 138–148, Dec 1997.

[104] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazon, and O. Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering*, 38:579–591, 2012.

[105] rubyflux. Ruby to Java compiler. Web pages at `https://github.com/headius/rubyflux`, 2014.

[106] A. Sarimbekov, S. Kell, L. Bulej, A. Sewe, Y. Zheng, D. Ansaloni, and W. Binder. A comprehensive toolchain for workload characterization across JVM languages. In *Proceedings Workshop on Program Analysis for Software Tools and Engineering*, PASTE '13, pages 9–16, June 2013.

[107] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site aware calling context profiling for the Java virtual machine. *Science of Computer Programming*, 79(0):146–157, 2014.

[108] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, M. Schoeberl, and M. Mezini. Portable and accurate collection of calling-context-sensitive bytecode metrics for the Java virtual machine. In *Proceedings Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 11–20, New York, NY, USA, 2011. ACM.

[109] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[110] Scavis. Scientific Computation and Visualization Environment. Web pages at `http://jwork.org/scavis/overview`, 2014.

[111] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proceedings International Symposium on Memory Management*, ISMM '12, pages 97–108, 2012.

[112] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. In *Proceedings Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 657–676, 2011.

[113] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '08, pages 127–142, New York, NY, USA, 2008. ACM.

[114] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[115] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko. SPECjvm2008 performance characterization. In *Proceedings SPEC Workshop on Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer-Verlag.

[116] J. Siek, S. Bharadwaj, and J. Baker. `invokedynamic` and Jython. Presentation at 2011 JVM Language Summit, available `http://wiki.jvmlangsummit.com/images/8/8d/Indy_and_Jython-Shashank_Bharadwaj.pdf`, retrieved 2013.

[117] M. D. Smith. Extending suif for machine-dependent optimizations. In *In Proceedings of the First SUIF Compiler Workshop*, pages 14–25, 1996.

[118] M. D. Smith. Overcoming the challenges to feedback-directed optimization (keynote talk). In *Proceedings Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 1–11, New York, NY, USA, 2000. ACM.

[119] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '94, pages 196–205, New York, NY, USA, 1994. ACM.

[120] L. Stadler, G. Duboscq, H. Mössenböck, T. Würthinger, and D. Simon. An experimental study of the influence of dynamic compiler optimizations on scala performance. In *Proceedings Workshop on Scala*, SCALA '13, pages 1–8, New York, NY, USA, 2013. ACM.

[121] E. Steiner, A. Krall, and C. Thalinger. Adaptive inlining and on-stack replacement in the CACAO virtual machine. In *International Conference on Principles and Practice of Programming in Java*, pages 221–226, Monte de Caparica/Lisbon, Portugal, September 2007.

[122] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 180–195, New York, NY, USA, 2001. ACM.

[123] A. S. Tanenbaum. A comparison of PASCAL and ALGOL 68. *The Computer Journal*, 21(4):316–323, 1978.

[124] The Apache Jakarta Project. The Byte Code Engineering Library (BCEL). Web pages at `http://jakarta.apache.org/bcel/`, 2014.

[125] The Standard Performance Evaluation Corporation. SPECjvm2008 benchmarks. Web pages at `http://www.spec.org/jvm2008/`, 2008.

[126] S. H. Valentine. Comparative notes on ALGOL 68 and PL/I. *The Computer Journal*, 17(4):325–331, 1974.

[127] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings Conference on Compiler Construction*, CC '00, pages 18–34, 2000.

[128] A. Villazón, W. Binder, and P. Moret. Flexible calling context reification for aspect-oriented programming. In *Proceedings Conference on Aspect-oriented Software Development*, AOSD '09, pages 63–74, New York, NY, USA, 2009. ACM.

[129] A. Villazón, W. Binder, P. Moret, and D. Ansaloni. Comprehensive aspect weaving for Java. *Science of Computer Programming*, 76(11):1015 – 1036, 2011.

[130] voodoo. Image manipulation library for JRuby. Web pages at `https://github.com/jruby/image_voodoo`, 2014.

[131] C. Wimmer and T. Würthinger. Truffle: A self-optimizing runtime system. In *Proceedings Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, pages 13–14, New York, NY, USA, 2012. ACM.

[132] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *Proceedings Conference on Software Engineering*, ICSE '08, pages 151–160. ACM, 2008.

[133] X. Yang, S. M. Blackburn, D. Frampton, J. B. Sartor, and K. S. McKinley. Why nothing matters: the impact of zeroing. In *Proceedings Conference on Object*

Oriented Programming, Systems Languages and Applications, OOPSLA '11, pages 307–324, 2011.

[134] D. Zaparanuks and M. Hauswirth. Algorithmic profiling. In *Proceedings Conference on Programming Language Design and Implementation*, PLDI '12, pages 67–76, New York, NY, USA, 2012. ACM.

[135] X. Zhang, Z. Wang, N. Gloy, J. B. Chen, and M. D. Smith. System support for automatic profiling and optimization. In *Proceedings Symposium on Operating Systems Principles*, SOSP '97, pages 15–26, New York, NY, USA, 1997. ACM.

[136] Y. Zheng, D. Ansaloni, L. Marek, A. Sewe, W. Binder, A. Villazón, P. Tuma, Z. Qi, and M. Mezini. Turbo DiSL: partial evaluation for high-level bytecode instrumentation. In *Proceedings Conference on Objects, Models, Components, Patterns*, TOOLS '12, pages 353–368, Berlin, Heidelberg, 2012. Springer-Verlag.