JVM-Hosted Languages: They Talk the Talk, but do they Walk the Walk?

Wing Hang Li   David R. White   Jeremy Singer
School of Computing Science, University of Glasgow
w.li.2@research.gla.ac.uk, david.r.white@glasgow.ac.uk, jeremy.singer@glasgow.ac.uk

Abstract

The rapid adoption of non-Java JVM languages is impressive: major international corporations are staking critical parts of their software infrastructure on components built from languages such as Scala and Clojure. However with the possible exception of Scala, there has been little academic consideration and characterization of these languages to date. In this paper, we examine four non-Java JVM languages and use exploratory data analysis techniques to investigate differences in their dynamic behavior compared to Java. We analyse a variety of programs and levels of behavior to draw distinctions between the different programming languages. We briefly discuss the implications of our findings for improving the performance of JIT compilation and garbage collection on the JVM platform.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Performance attributes; D.3.4 [*Programming Languages*]: Processors—Compilers

General Terms Measurement, Performance

Keywords Dynamic analysis; Java virtual machine; JVM bytecode

1. Introduction

Java is one of the most popular programming languages currently in use. Much of its success may be attributed to the Java Virtual Machine (JVM). JVM implementations are available on a wide variety of platforms, allowing developers to "write once, run anywhere". In practice, this means that an application is compiled to bytecode and this bytecode can be executed on any platform with an available JVM. The JVM also provides a sandboxed environment for security, and adaptive optimizations that improve the performance of applications.

An increasing number of programming language toolchains produce bytecode that is executable on the JVM. Developers choose to use these languages for features or programming paradigms that are not available in Java. Many of these JVM languages can interoperate with Java, allowing developers to use existing Java libraries and reducing the risk of adopting the new language. Non-Java languages can take advantage of automatic memory management and adaptive optimizations provided by the JVM. However, JVM implementations expect to execute Java bytecode, therefore the performance of non-Java bytecode may be poor. For instance, Java uses static typing but there are JVM languages that use dynamic typing instead. The JVM introduced support for dynamic types in Java 7; support for other features, such as tails calls, continuations and interface injection, is currently in progress[1].

The adaptive optimization heuristics and garbage collection (GC) algorithms used by the JVM are other possible reasons for the poor performance of non-Java JVM programming languages. Previous studies [26][25] have shown that the characteristic behavior of Scala is different from that of Java. They suggest that it is possible to improve Scala's performance on a JVM based on those differences. Our work examines the behavior of three other JVM programming languages; Clojure, JRuby and Jython. All three languages are dynamically typed and JRuby and Jython in particular are generally used as scripting languages.

Our static and dynamic analyses reveal that significant amounts of Java code are used by non-Java languages. We apply exploratory data analysis methods [12] to data gathered through dynamic profiling of programs written in five JVM languages. Instruction-level analysis shows that bytecode compiled from non-Java languages exhibits instruction sequences never or rarely used by Java. We find differences in method size and stack depths between JVM languages but no difference in method and basic block hotness. Heap profiling reveals that non-Java objects are generally smaller and shorter lived than Java objects. Non-Java languages also make more use of boxed primitive types than Java. We discuss possible changes to JVM optimization heuristics based on the observed behavior of non-Java languages.

1.1 Our Contribution

This empirical investigation makes two key contributions:

- We explore the behavior of benchmarks written in five JVM programming languages (wider scope than earlier studies). To the best of our knowledge, the behavior of Clojure, JRuby and Jython programs has never been previously studied in a systematic fashion.
- We examine the proportion of Java code used by each non-Java programming language through static analysis of each programming language's library and dynamic analysis of a set of benchmarks. With this information, we can estimate the effectiveness of custom optimizations for such JVM-hosted languages.

, .
Copyright © ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/

[1] e.g. `http://openjdk.java.net/projects/mlvm/`

2. Methodology

In this section, we describe the JVM languages in our study, the benchmarks written in those languages, the tools we use to profile them and the analysis techniques we use to identify interesting behaviors. We will provide the source code, experimental scripts and raw data from our empirical study via our project repository[2].

2.1 JVM Languages

We examine five JVM languages:

Java is the original JVM programming language. It is object-oriented and statically typed. We use Java program behavior as the baseline in all our quantitative comparisons.

Clojure [9] is a LISP dialect, with support for advanced concurrency features, including actors and software transactional memory. It is a functional language with dynamic typing.

JRuby [14] is a Java-based implementation of the Ruby programming language. It is a dynamic, object-oriented language.

Jython [17] is an implementation of the Python language for the Java platform. It is a dynamic, object-oriented language.

Scala [15] is a functional and object-oriented programming language, with a static typing discipline. It has advanced facilities for typing and concurrency.

2.2 Program Suite

We searched for suitable examples of applications, representative of each JVM programming language in our study. We used 10 benchmark programs from the Computer Languages Benchmark Game[3] (CLBG) project. The site compares programming language performance by implementing a solution to a set of computing problems in different programming languages. The benchmarks use the same underlying algorithm, therefore they may ignore programming language features that would enable improved performance using a different algorithm. The project organizers acknowledge that the style of each benchmark problem implementation has a significant effect on its relative performance. A recent empirical study of the R scripting language [13] also used the CLBG benchmarks.

The website includes benchmark implementations in four of the five JVM languages (excluding Jython) outlined in Section 2.1. The available Python benchmarks were converted from Python v3 to v2 and minor changes were made to remove multiprocessing and fix output format problems for Jython compatibility. Table 1 outlines the general characteristics of the 10 CLBG benchmark problems we use for comparative analysis and their workloads. Three of the benchmarks use text output from the Fasta benchmark (consisting of 150,000 randomly generated DNA sequences) as their input.

We supplement the CLBG benchmarks with additional benchmarks that are representative examples of non-trivial applications written in the JVM programming languages. The DaCapo Benchmark Suite ver. 9.12 for Java [1] and the Scala Benchmark Suite ver. 0.1.0 [26] contain open source applications chosen from a range of domains. We set the workload size of these benchmark suites to small, to limit the size of the traces produced by the heap profiler.

We could not find any Clojure, JRuby or Jython benchmark suites, and there are few individual open source, real-world applications. We select three examples of applications written in Clojure and JRuby and use them as benchmarks, combined with suitable workloads. The examples were chosen because they were commonly used, executable from a command line and repeatable with the same input.

benchmark	*workload*	*int*	*fp*	*ptr*	*str*
Binary-Trees	tree depth 13			Y	
Fannkuch-Redux	sequence length 9	Y			
Fasta	150000 sequences			Y	
K-Nucleotide	Fasta output			Y	Y
Mandelbrot	500 image size		Y		
Meteor-Contest	2098 solutions	Y			
N-body	100000 steps		Y		
Regex-DNA	Fasta output				Y
Reverse-Complement	Fasta output				Y
Spectral-Norm	160 approximations		Y		

Table 1: Description of the Computer Languages Benchmark Game (CLBG) corpus of programs used in our study, indicating whether a program mostly manipulates integers, floating-point numbers, pointers or strings.

The Clojure applications we profiled were:

Noir - a web framework written in Clojure. We profile a blog site created using Noir 1.2.2 that was provided by the Noir website. We use a Perl script to simulate user interaction with the blog site as input.

Leiningen - a build automation tool that simplifies downloading and building Clojure software. We profile Leiningen 2.1.0 building and packaging the Noir blog site into an executable uberjar.

Incanter - an R-like statistical computing and graphing tool. Many of its modules are based on Java libraries, therefore we only profile the Incanter 1.5.0 core module running its associated unit tests.

The JRuby applications we profiled were:

Ruby on Rails - a popular web framework, using JRuby to run on a JVM. We created a blog site using Ruby on Rails 3.2.13 and the Rails website's tutorial. A Perl script was again used to interact with the blog site while it was being profiled.

Warbler - an application that packages a Ruby or Ruby on Rails application into a Java jar or war file. We profile Warbler 1.3.6 as it builds and packages the Ruby on Rails blog site used previously, into an executable war file.

Lingo - an open-source Ruby application for the automatic indexing of scientific texts. We profile Lingo 1.8.3 as it indexes "Alice in Wonderland" using its standard English dictionary settings.

Jython is mainly used as a scripting language, therefore we could not find any suitable large applications written in this language. We had difficulty porting Python applications since the latest Python language specification and libraries are not fully implemented by Jython and therefore we only profile the CLBG Python (Jython) benchmarks.

2.3 Data Generation

We gather data from the benchmarks using two profilers, as shown in Figure 1. The first profiler we use is a modified version of JP2 2.1 [23] to collect information about the bytecode instructions and methods used by each benchmark. The original version of JP2 records method and basic block execution counts; our version also records the bytecode instructions within each basic block. Only 198 of the 202 bytecodes are recorded distinctly since bytecodes similar to `LDC`, `JSR` and `GOTO` are aggregated. The data produced by JP2 is non-deterministic. However, using 3 sets of traces, we found that the figures recorded for bytecodes and methods executed display a

[2] `http://www.dcs.gla.ac.uk/~wingli/jvm_language_study`

[3] `http://shootout.alioth.debian.org/`

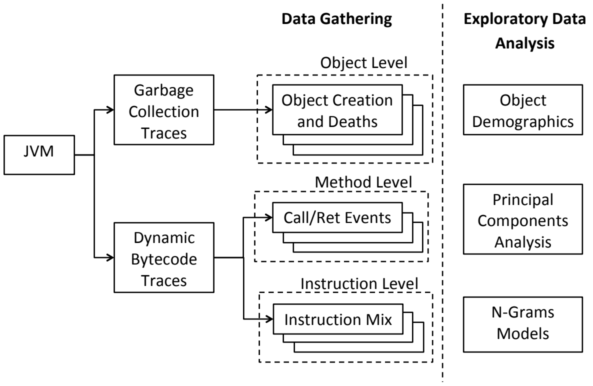


Figure 1: Schematic diagram of profiling data generation and analysis techniques.

standard deviation of less than 0.2%.

The second profiler we use is Elephant Tracks (ET) 0.6 [21, 22], a heap profiler that provides information about object allocations and deaths. Some benchmarks failed to complete using ET; possibly due to the high overhead or instrumentation used by the profiler. Neither JP2 or ET provide complete coverage of all the classes used by a JVM language. JP2 instruments each method used, therefore there is no record of classes that are only accessed by their fields. ET has a known bug where objects in the constant pool have no allocation record. However, these problems occur infrequently.

We start by compiling the source code of each benchmark to JVM bytecode using the canonical compiler for its language. The exception is Jython, which uses high-level language interpretation. Table 5 details whether each benchmark is compiled to bytecode ahead-of-time (AOT), just-in-time (JIT) or interpreted. The JVM language toolchain versions we use are Java 1.6_30, Clojure 1.3.0, JRuby 1.6.7, Jython 2.5.3 and Scala 2.9.2. We execute the compiled bytecode (or run the Jython interpreter) using the Java HotSpot 64-bit Server VM 1.6.0_30, running on Linux kernel v3.2.0-48, x86_64 build. The traces produced by JP2 and ET are parsed to extract the desired metrics.

2.4 Analysis Techniques

Figure 1 shows the range of *exploratory data analysis* techniques we apply to gain an understanding of the dynamic behavior of the various programs. We endeavor to follow appropriate past practice in the community in presenting our results, as indicated in the rest of this section.

N-gram models: An N-gram is a sequence of N consecutive JVM bytecode instructions within a single basic block. We consider the coverage of observed N-grams in relation to the theoretical maximum– if there are 198 observable bytecode instructions, then there can be 198^N N-gram sequences.

O'Donoghue et al. [16] introduce the concept of N-gram analysis for Java execution traces. They measure dynamic 2-gram frequencies for a set of Java benchmarks, to identify candidates for bytecode super-instructions.

Principal Component Analysis: (PCA) [27] is a frequently used technique for dimension reduction, to aid visualization and comprehension. PCA works by choosing and combining dimensions that contain the greatest variance. For each individual benchmark program, we measure the relative frequency of each JVM bytecode instruction to produce a 198-vector of values in range $[0, 1]$ or a 39204-vector of values for 2-grams. We apply PCA to reduce the number of dimensions from 198 or 39204 to 4.

Sewe et al. [26] use this approach to characterize Scala and Java programs. Blackburn et al. [1] apply PCA to dynamic processor-level metrics to visualize Java benchmark diversity.

Boxplots: We use boxplots to summarize distributions of data for measurements on methods and objects. The following conventions are used for all boxplots in this paper. Each box denotes the interquartile range, with the median marked by a thick black stroke. The boxplot whiskers mark the 9th and 91st percentiles; values outside this range are considered outliers. The absolute value reported to the right of each box is the maximum recorded observation for that program. The use of boxplots to summarize static and dynamic characteristics of Java programs is becoming common practice, e.g. [7, 34].

Heat maps: We use heat maps to compare object lifetimes between JVM languages. Darker shadings indicate a higher proportion of objects within a lifetime range. The left of the heat map represents objects with a short lifetime while the right represents longer lived objects.

Dufour et al. [6] use relative frequencies to compare dynamic metrics values across Java benchmarks.

3. Results

3.1 Java vs. Non-Java Code

All non-Java JVM languages use Java classes to a certain extent. This may be exhibited (a) *statically*, in terms of the proportion of shipped bytecode classes that are compiled from Java source code, and (b) *dynamically*, in terms of the proportion of executed methods that are compiled from Java source code. The proportion of Java used by each non-Java programming language will have an impact on the effectiveness of any language-specific optimization that we might propose.

A static analysis of the language libraries distributed with each non-Java JVM language reveals the proportion of the library code that is implemented in Java. Each class in the library is classified using the extension of the corresponding source file. If this source file information is not stored in the class metadata, we locate and classify the source file manually via source code inspection.

Table 2 shows that the proportion of Java code within the Clojure and Scala libraries is exceeded by non-Java code. However, the proportion of Java code within the JRuby and Jython libraries exceeds the amount of non-Java code by a significant factor. This suggests that JRuby and Jython may not benefit greatly from optimizations based on differences between Java and non-Java programming language behavior. To verify whether this is the case, we separate Java and non-Java bytecodes, methods and objects by *filtering* the dynamic traces obtained from the profiling tools, using the same method we used to classify the classes in the language libraries. This allows us to explore the behavior of the non-Java language methods and objects in isolation.

Table 5 details run-time statistics for the profile traces of all our benchmark programs. The table shows that some benchmarks (e.g. Clojure and Scala Regex-Dna) never execute a high percentage of non-Java bytecode due to their reliance on Java library classes.

The data in Table 5 shows that the proportion of Java code executed at run-time is similar to the results of the static analysis. JRuby and Jython, as expected, use a small proportion of instructions, methods and objects produced from JRuby or Jython source code. The Clojure CLBG benchmarks use less Java than the three application benchmarks, indicating that Clojure applications are more likely to depend on Java components. For instance, the Noir blog uses Jetty, a web server written in Java, while the Clojure compiler used by Leiningen is implemented in Java. Scala is the only non-Java JVM language that uses a significant proportion of instructions, methods and objects implemented in its own language.

	Java			Non-Java		
Language	**Classes**	**Methods**	**Instructions**	**Classes**	**Methods**	**Instructions**
Clojure 1.3.0	696 (24%)	3887 (33%)	146333 (24%)	2171 (76%)	7813 (67%)	455567 (76%)
JRuby 1.6.7	5167 (65%)	37877 (87%)	1987564 (98%)	2767 (35%)	5874 (13%)	47130 (2%)
Jython 2.5.3	4002 (68%)	45813 (86%)	1592392 (96%)	1852 (32%)	7641 (14%)	63015 (4%)
Scala 2.9.2	140 (3%)	1195 (1%)	25840 (3%)	5230 (97%)	100616 (99%)	866451 (97%)

Table 2: The number (respectively, proportion) of classes, methods and instructions from Java and non-Java source files within non-Java programming language libraries.

The results indicate that non-Java JVM languages are heavily dependent on Java, therefore any efforts to tune the adaptive optimizations or garbage collection algorithms used by a JVM must take care not to degrade Java performance. The results also indicate that observed differences in the behavior between non-Java JVM languages and Java will only be of use to Scala and, to a lesser extent, Clojure. Java is a necessary part of non-Java JVM languages and therefore we also look at unfiltered data in our instruction, method and object level results. This will reveal if the behavior of non-Java applications, including the Java code that they use, differs from standard Java applications.

3.2 Instruction-level Results

In this section, we look for differences in the bytecode executed by the JVM for the 5 JVM languages.

3.2.1 N-gram Coverage

This investigation considers the dynamic N-gram vocabulary exhibited by each language. A *use* of N-gram g for language L means at least one of the benchmark implementations in L executes a basic block containing g at least once.

Table 3 shows that the Java N-gram vocabulary is more diverse than for non-Java languages. However, it should be noted that there are 20 Java benchmarks in this study, while there are 13 or fewer benchmarks for Clojure, JRuby and Jython. Scala has 20 benchmarks, but it still displays a smaller N-gram vocabulary than Java. Table 4 shows that, despite Java using diverse N-grams, each non-Java JVM language still uses N-grams not used by the Java benchmarks. Moreover, these N-grams are executed frequently; for instance, 58.5% of 4-grams executed by Scala are not found in bytecode executed by Java. This indicates that a significant amount of the N-grams used by non-Java JVM languages are, at the very least, uncommonly used by Java.

3.2.2 Principal Component Analysis

We apply PCA to the bytecode instruction frequencies and 2-gram frequencies, producing the scatter plots in Figure 2. The unfiltered 1 and 2-gram PCA graphs show that benchmarks belonging to different JVM languages display a low amount of divergence; forming a single cluster with few outliers. This is likely due to the amount of Java used by non-Java JVM languages, resulting in similar instructions being used. The filtered 1 and 2-gram PCA graphs show JRuby and Jython benchmarks forming distinct clusters away from Java, Clojure and Scala. The clustering is most apparent in the filtered 2-gram PCA graph.

The PCA graphs indicate that N-grams produced by the JRuby compiler and the Jython interpreter are distinct from other JVM languages in the frequency and type of 1 and 2-grams they use. Conversely, Java, Clojure and Scala use similar 1 and 2-grams, even when Java code has been filtered from the Clojure and Scala N-grams. The PCA graphs indicate that for shorter N-grams, only the filtered JRuby and Jython show different behavior compared to Java.

3.3 Method-level Results

This section compares the method sizes, stack depths, method hotness and basic block hotness between the five JVM languages.

3.3.1 Method Sizes

Method inlining heuristics are generally parameterized on caller/ callee static method size [29], therefore we examine the method sizes commonly used by different JVM languages. The boxplots in Figure 3 show the distribution of static method sizes for each benchmark, measured in terms of bytecode instructions, weighted by dynamic invocation frequency. The median method size for CLBG benchmarks written in Java varies considerably compared to the DaCapo benchmarks. This is not true for the non-Java JVM languages, whose unfiltered median method sizes show less variation. The filtered median method sizes for Clojure, JRuby and Jython show much more variability. However, the results of the dynamic analysis in Table 5 has shown that these methods represent a small proportion of the executed methods. Scala shows the most interesting behavior; the median method size is small, typically three instructions only, for most of its benchmarks.

3.3.2 Stack Depths

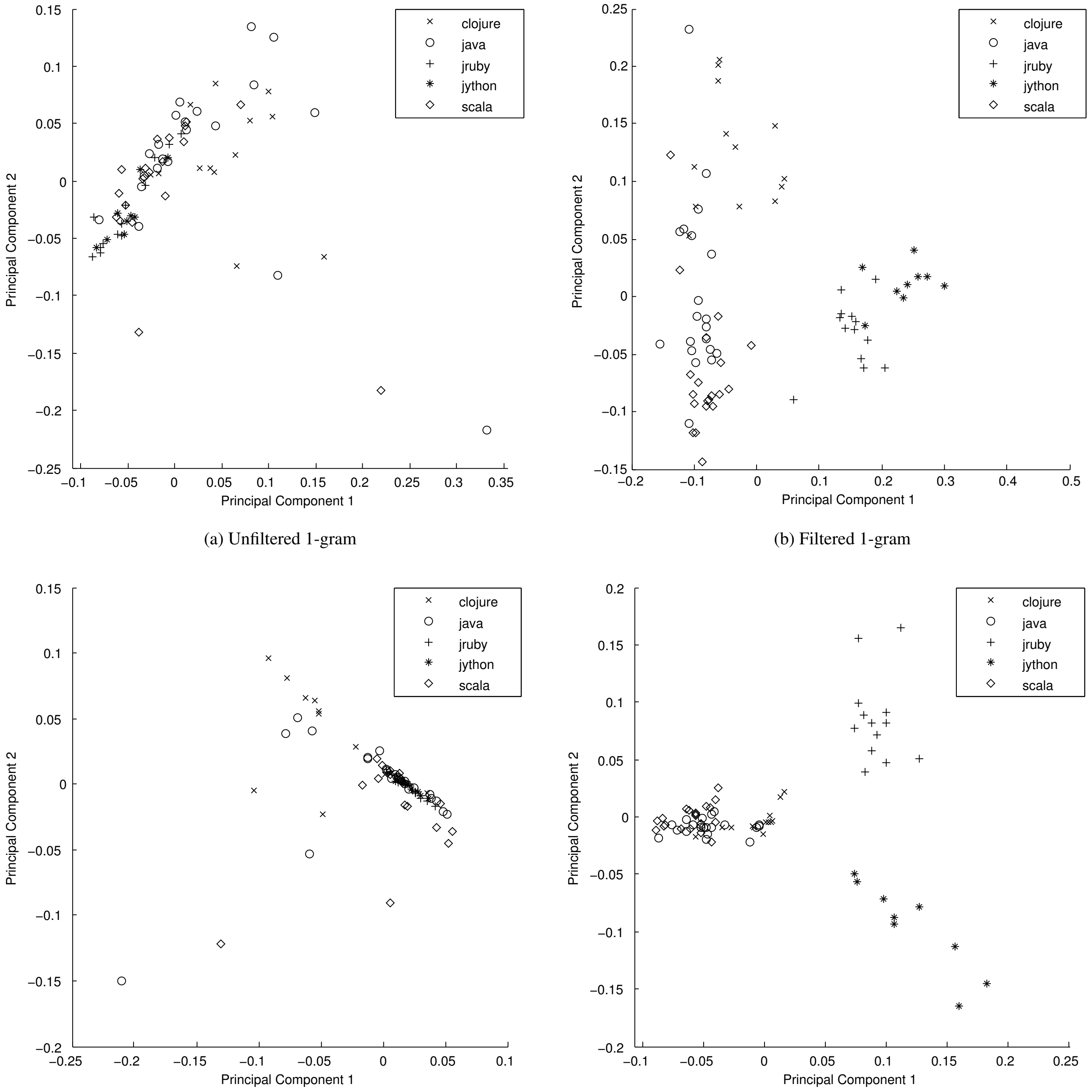
Each thread maintains its own stack; we record the thread's current stack depth whenever a method entry or exit occurs in an ET trace. We report the distribution of recorded stack depths for each benchmark. However, exceptional method exits are not recorded by ET and method stacks can switch between threads while sleeping, leading to a small amount of inaccuracy in our results. Figure 4 shows that the median stack depth for the CLBG benchmarks is less than 50 for all of the JVM languages. However, the median stack depth is higher for larger applications. In particular, Clojure and Scala applications display larger stack depths, possibly due to recursion used by these functional languages.

3.3.3 Method and Basic Block Coverage

We measure method and basic block hotness as per [24] by determining the most frequently executed 20% of methods and basic blocks and measuring the amount of executed bytecodes that they cover. The results in Table 5 show that in most cases, the most frequent 20% of methods will cover almost all of the executed bytecode. One anomaly is the `revcomp` benchmark for Clojure, Java, JRuby and Scala, for which there are relatively many frequently executed methods that are small in size. However, the most frequent 20% of basic blocks covers more than 98% of bytecodes executed for all benchmarks. There is little difference in the method and basic block hotness between Java and non-Java benchmarks.

3.4 Object-level Results

In this section, we compare object lifetimes, object sizes and the amount of boxed primitives used by the five JVM languages in our study.



(a) Unfiltered 1-gram

(b) Filtered 1-gram

(c) Unfiltered 2-gram

(d) Filtered 2-gram

Figure 2: PCA scatter plots illustrating variation in bytecode instruction mix across different JVM languages.

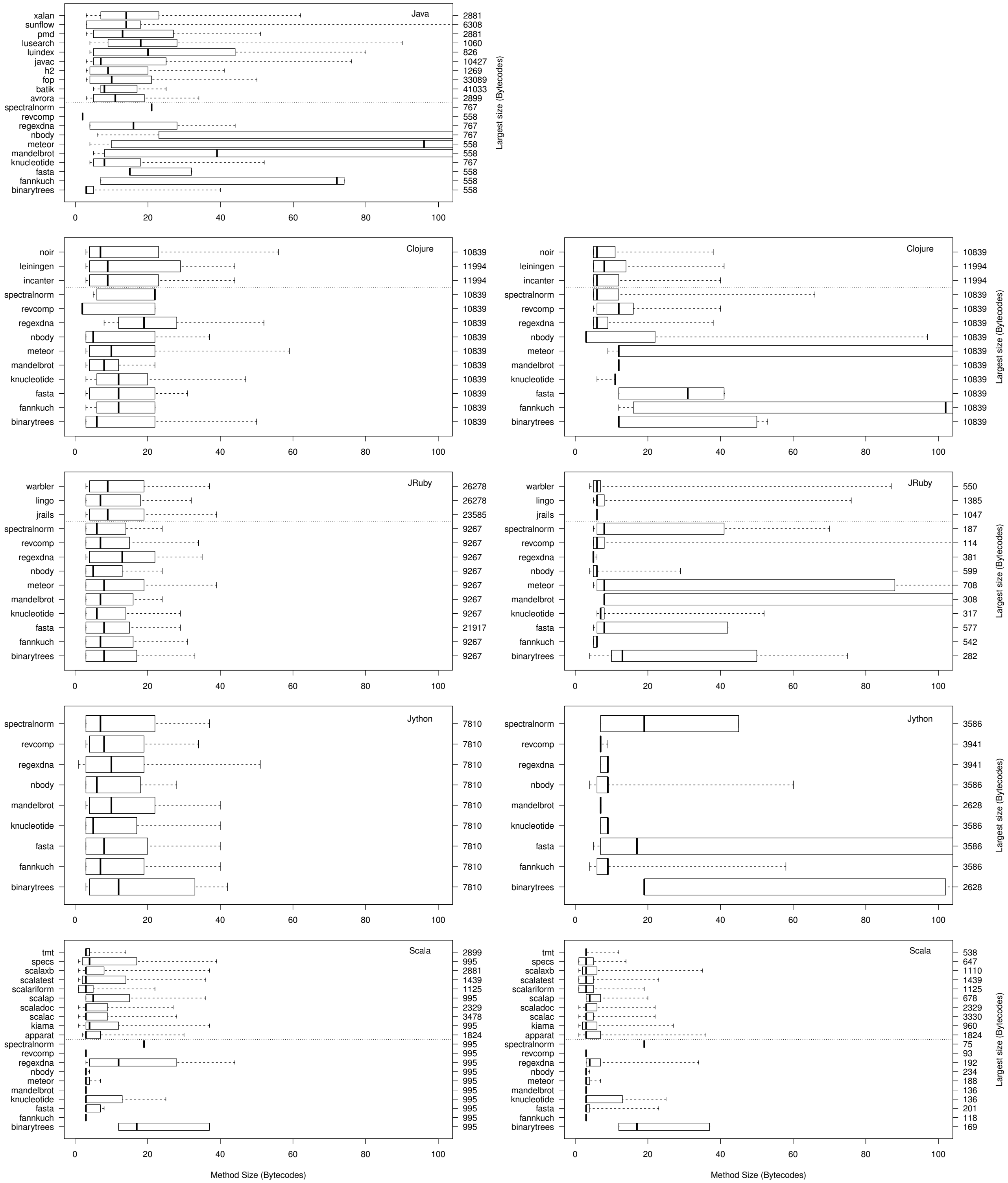


Figure 3: The distribution of method sizes for each benchmark, grouped by language. Boxplots in the right column are based on filtered (i.e. non-Java) methods only. Methods of size greater than 100 instructions are off the scale.

language	*filtered?*	*1-gram*	*2-gram*	*3-gram*	*4-gram*
Java	No	192	5772	31864	73033
Clojure	Yes	118 (0.61)	1217 (0.21)	3930 (0.12)	7813 (0.10)
Clojure	No	177 (0.92)	4002 (0.68)	19474 (0.58)	40165 (0.51)
JRuby	Yes	54 (0.28)	391 (0.07)	1212 (0.04)	2585 (0.03)
JRuby	No	179 (0.93)	4482(0.76)	26373 (0.79)	64399 (0.81)
Jython	Yes	48 (0.25)	422 (0.07)	1055 (0.03)	1964 (0.02)
Jython	No	178 (0.92)	3427 (0.58)	14887 (0.44)	27852 (0.35)
Scala	Yes	163 (0.84)	2624 (0.45)	11979 (0.36)	30164 (0.38)
Scala	No	187 (0.97)	3995 (0.68)	19515 (0.58)	45951 (0.58)

Table 3: N-gram coverage for various JVM languages. Number in brackets shows the value relative to Java

language	*filtered?*	*1-gram*	*2-gram*	*3-gram*	*4-gram*
Clojure	Yes	2	193 (0.11)	1957 (0.46)	6264 (0.77)
Clojure	No	2	348 (0.05)	4578 (0.23)	15824 (0.43)
JRuby	Yes	1	44 (0.02)	399 (0.14)	1681 (0.42)
JRuby	No	1	512 (0.01)	7659 (0.08)	30574 (0.26)
Jython	Yes	1	38 (0.07)	412 (0.19)	1491 (0.56)
Jython	No	1	161 (0.01)	2413 (0.06)	8628 (0.19)
Scala	Yes	0	288 (0.03)	4168 (0.27)	18676 (0.69)
Scala	No	0	335 (0.02)	4863 (0.23)	21106 (0.59)

Table 4: Number of N-grams used by non-Java benchmarks that are not used by Java benchmarks. For $2 \leq N \leq 4$ the number in brackets represents the value relative to total number of filtered or unfiltered N-grams covered by these non-Java N-grams.

3.4.1 Object Lifetimes

The heat maps in Figure 5 show the percentage of objects allocated within 5% intervals of the total execution time. A black colored interval indicates that 10% or more of the total objects allocated are within the lifetime range. We observe that most objects die young for the DaCapo and Scala benchmark suites and the three Clojure application benchmarks, following the weak generational hypothesis [32]. However the lifetimes of objects allocated by the Java, Scala and Clojure CLBG benchmarks are more varied. More objects belonging to the Scala benchmark suite die young compared to the DaCapo benchmarks, confirming previous results by Sewe et. al [25]. However, more Clojure objects live longer compared to Java. Unfiltered JRuby and Jython display interesting behavior; their objects either live for less than 5% of the total execution time or for the duration of program execution.

3.4.2 Object Sizes

We examine the distribution of object sizes, weighted by their dynamic allocation frequency. The size of `java.lang.Object` is 16 bytes for the JVM we use. The boxplots in Figure 6 show that the object size for most non-Java JVM languages is dominated by only one or two sizes. This can be seen from the median object size in the unfiltered JRuby and Jython boxplots and the filtered Clojure and Scala boxplots. However, the median object size for Java varies between 24 to 48 bytes. By comparing the unfiltered and filtered boxplots, we see that Clojure and Scala use smaller objects more frequently than Java.

3.5 Boxed Primitives

The final metric in our study measures boxing, in terms of the proportion of allocated objects that are used to wrap primitive types. For Java, Clojure and Scala, we count the number of Java `Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long` and `Short` objects allocated, as well as `BigDecimal` and `BigInteger`. JRuby and Jython use Java boxed classes and implement their own boxing classes. For JRuby, we count `RubyBoolean`, `RubyComplex`, `RubyBigDecimal`, `RubyBigNum`, `RubyFloat`, `RubyFixNum`, `RubyNumeric` and `RubyRational` objects allocated. For Jython, we count `PyBoolean`, `PyComplex`, `PyComplexDerived`, `PyFloat`, `PyFloatDerived`, `PyInteger`, `PyIntegerDerived`, `PyLong` and `PyLongDerived` objects allocated.

The results in Table 5 shows that Java benchmarks use very little boxing. However, non-Java benchmarks are more likely to use a high proportion of boxed primitives. The proportion of boxed primitives used by Clojure is unexpectedly high, even though primitives can be used in Clojure. Conversely, all primitives must be boxed in Scala, yet the level of boxing used is lower than for Clojure.

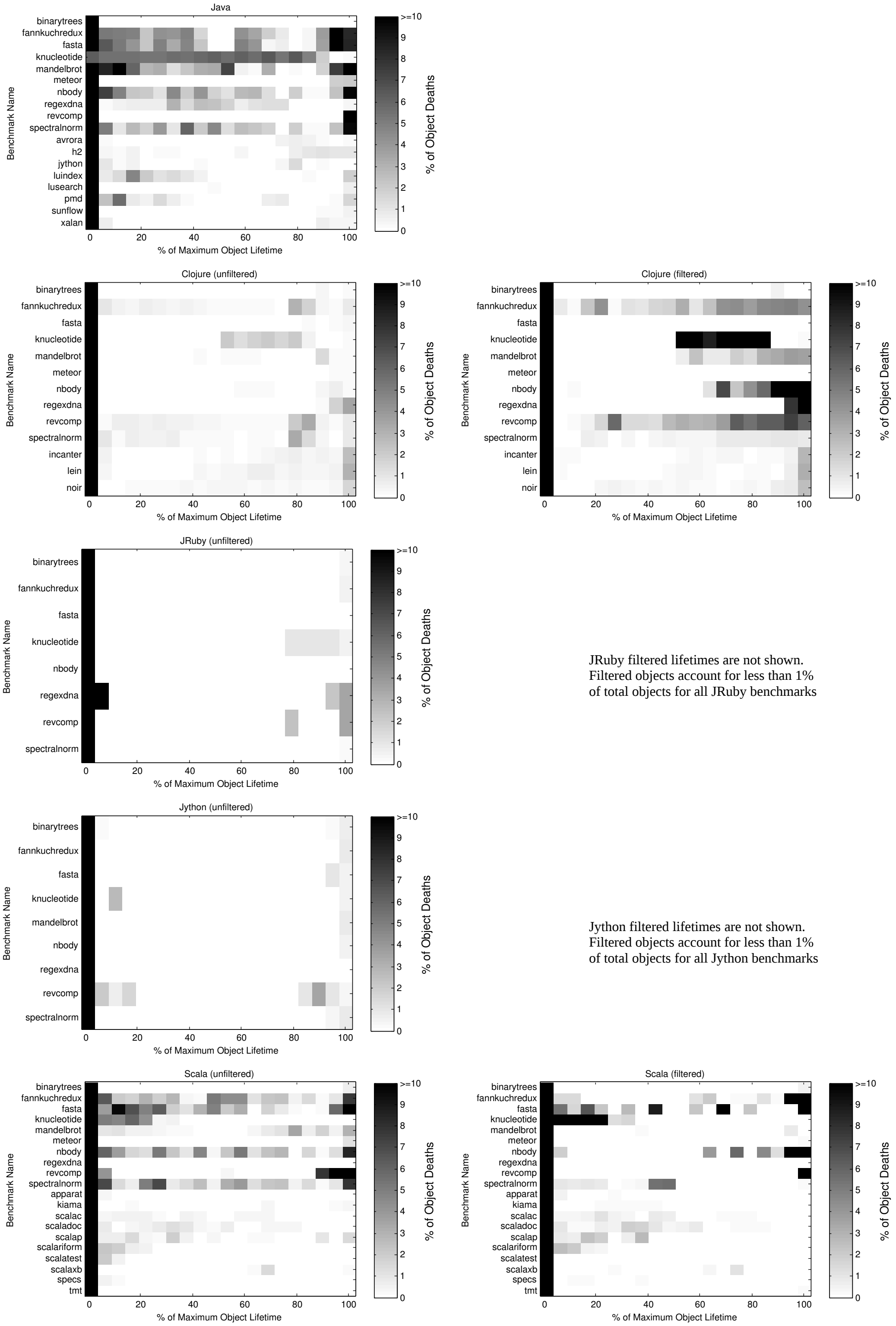
4. Related Work

Many researchers have sought to characterize Java programs. However to date, large-scale studies based on corpora such as *Qualitas* [30] have been restricted entirely to *static* analysis, e.g. [3, 31].

Dynamic studies of Java programs have generally focused on one particular aspect such as instruction mix [19] or object demographics [5, 10]. Stephenson and Holst [28] present an analysis of multicodes, which are a generalization of bigrams [16] to variable length dynamic bytecode sequences.

Dufour et al. [6] present a broader dynamic analysis study. They introduce a range of new metrics to characterize various aspects of Java program behaviour including polymorphism, memory use and concurrency. The explicit design rationale for their metrics is to enable quantitative characterization of a Java program with relevance to compiler and runtime developers. Sarimbekov et al. [24] motivate the need for workload characterization across JVM languages. They propose a suite of dynamic metrics and develop a toolchain to collect these metrics. Our work relies in part on earlier incarnations of their tools (e.g. JP2 [23]) and has the same motivation of JVM-based cross-language comparison.

Radhakrishnan et al. [20] study the instruction-level and method-level dynamic properties of a limited set of Java benchmarks in order to make proposals about architectural performance implications of JVM execution. They discuss processor cache configuration and instruction-level parallelism. Blackburn et al. [1] use various dynamic analysis techniques including PCA on architecture-level


Java

Clojure (unfiltered)

Clojure (filtered)

JRuby (unfiltered)

JRuby filtered lifetimes are not shown.
Filtered objects account for less than 1%
of total objects for all JRuby benchmarks

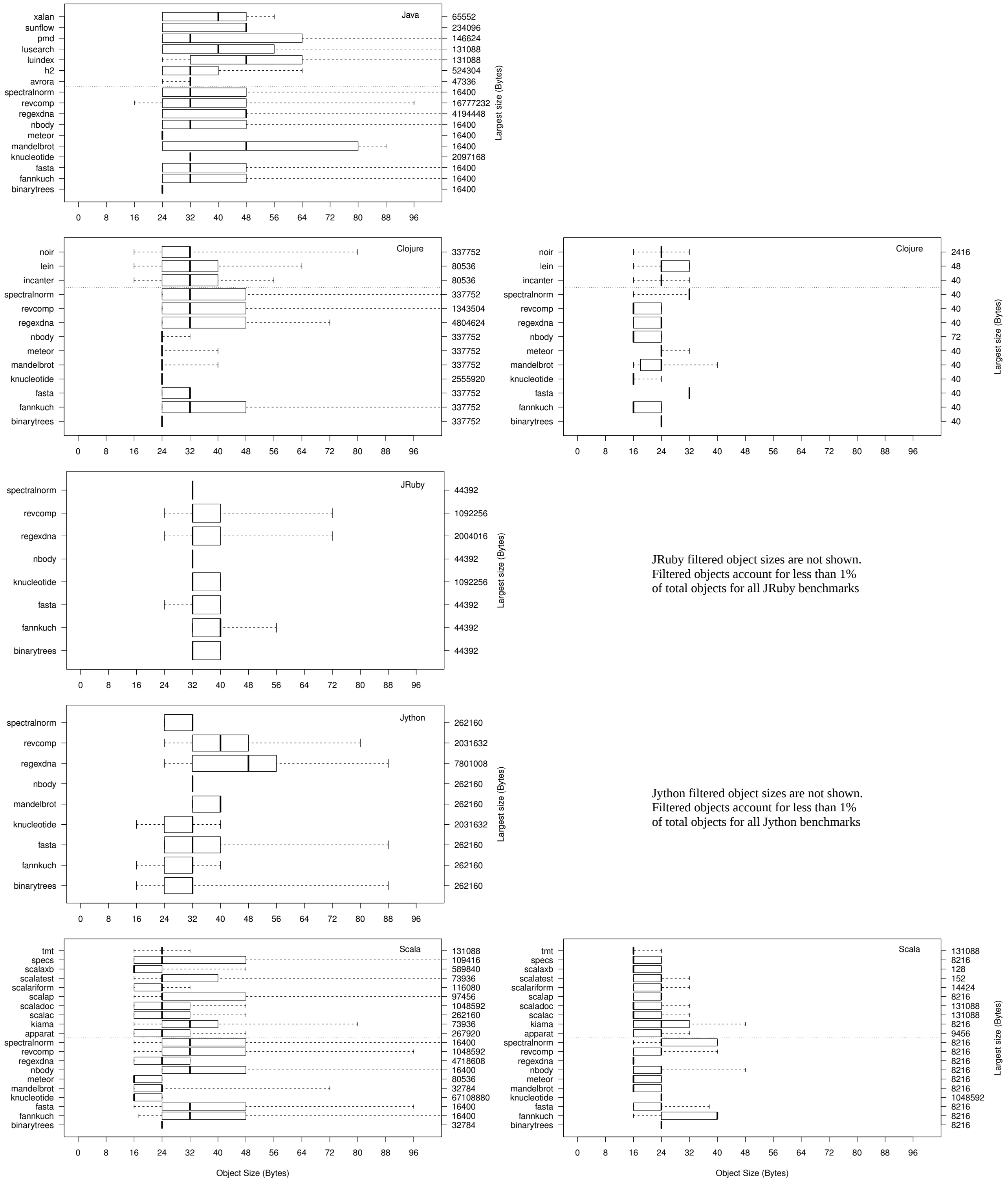
Jython (unfiltered)

Jython filtered lifetimes are not shown.
Filtered objects account for less than 1%
of total objects for all Jython benchmarks

Scala (unfiltered)

Scala (filtered)

Figure 5: Heat maps illustrating distribution of relative object lifetimes for each benchmark, grouped by language. Shorter lifetimes are on the left, longer lifetimes on the right of each graph. Darker regions denote higher proportions of objects with this lifetime. Heat maps in the right column are based on filtered (i.e. non-Java) objects only. Heat maps for filtered JRuby and Jython data are not shown due to the small proportion of allocated objects they represent.



JRuby filtered object sizes are not shown.
Filtered objects account for less than 1%
of total objects for all JRuby benchmarks

Jython filtered object sizes are not shown.
Filtered objects account for less than 1%
of total objects for all Jython benchmarks

Figure 6: The distribution of object sizes for each benchmark, grouped by language. Boxplots in the right column are based on filtered (i.e. non-Java) objects only. Boxplots for filtered JRuby and Jython data are not shown due to the small proportion of allocated objects they represent.

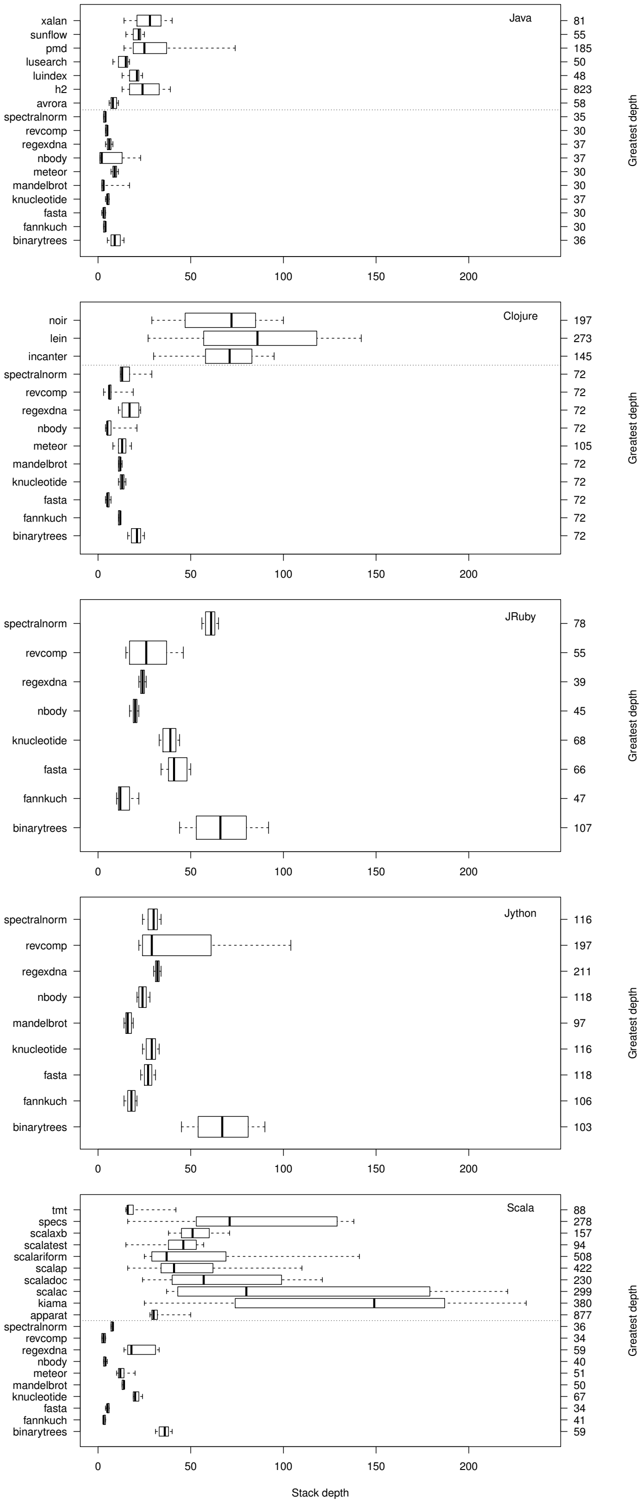


Figure 4: The distribution of method stack depths for each benchmark, grouped by language.

metrics to demonstrate quantitative differences between two Java benchmark suites. Their work follows on from earlier PCA-based workload characterization studies for Java: Chow et al. [2] consider architecture-level metrics, whereas Yoo et al. [33] consider OS-level metrics. In contrast, we restrict attention to VM-level metrics in our PCA characterization.

Sewe et al. [25, 26] use a range of static and dynamic analyses to characterize the Java DaCapo benchmark suite in relation to their own Scala benchmark suite. Their study does not compare the performance of the two languages, they state "Such a study would require two sets of equivalent yet idiomatic applications written in both languages." Instead, their intent was to study the characteristics of a number of real-world Java and Scala applications. To the best of our knowledge, there are no similar academic studies of other JVM languages like Clojure, JRuby and Jython. We are aware of the intrinsic difficulties in performing objective comparative studies of programming languages [18]. However in our study we have attempted to present data in a fair and objective way: sometimes this means we have presented more data (e.g. filtered and unfiltered) to avoid bias.

Unlike the JVM platform, the .NET common language runtime (CLR) was originally designed to be an appropriate target architecture for multiple high-level programming languages. Thus it has a slightly richer bytecode instruction set. Gough [8] evaluates the merits of CLR for various high-level languages. Dick et al. [4] present a simple analysis of the dynamic behavior of .NET bytecode for a small-scale benchmark suite. Based on their limited analysis, they discuss the similarities of .NET and JVM bytecode.

Knuth is the pioneer of dynamic characterization of programs. His empirical study of Fortran [11] was restricted by the available programs. His code acquisition techniques include 'recovering' punch cards from waste bins and 'borrowing' user's programs to duplicate them while they waited in the compilation queue. While we acknowledge the limited size of our program corpus, we are confident (like Knuth) that larger corpora will be easier to collect and curate in the future.

5. Conclusions

5.1 Summary

The JVM platform now plays host to a variety of programming languages, but its optimizations are implicitly tuned to the characteristics of Java programs. This paper investigates the behavior of a sample of programs written in Java, Clojure, JRuby, Jython and Scala, to determine whether Java and non-Java JVM languages behave differently. We perform static analysis on the language libraries and dynamic analysis on a set of 75 benchmarks written in those languages. Exploratory data analysis techniques were used to visually identify interesting behaviors in the data. Static and dynamic analysis shows that non-Java JVM languages rely heavily on Java code (a) for implementing significant parts of their language runtimes and libraries, and (b) as clients of the Java standard library classes.

At the instruction-level, non-Java benchmarks produce N-grams not found within the Java benchmarks, suggesting they do not share precisely the same instruction-level vocabulary as Java. Java method sizes are more varied than non-Java method sizes; Scala in particular has much smaller methods than the other JVM languages. Clojure and Scala applications exhibit deeper stack depths than other JVM language benchmarks. However there is no difference in method and basic block hotness between JVM languages. The object lifetimes of non-Java JVM languages are generally shorter than for Java. Filtered Clojure and Scala object sizes are smaller than they are for Java. Finally, we observe that non-Java languages use more boxed primitive types than Java.

Our research demonstrates that there are some noticeable differences in certain behavior between Java and non-Java JVM languages. We acknowledge that these observed behaviors may change as JVM languages mature and as new features are added to the JVM platform.

5.2 Future Work

The primary goal of this study is not a search for optimization opportunities. However, certain results do indicate some interesting behaviors that differ between Java and non-Java JVM languages. We plan to investigate whether existing JVM optimizations already cover these behaviors, and whether deployed JVM heuristics require re-tuning.

N-gram analysis has shown that Java and non-Java languages use different N-grams. JIT compilers perform peephole optimization, a technique that replaces sequences of bytecode instructions with shorter or more efficient sequences. We plan to examine existing JVM peephole optimizations to determine whether they already cover these diverse N-grams.

The median method sizes for the non-Java JVM languages are generally smaller and show less variation between benchmarks than they do for Java. It is therefore worthwhile investigating if the JVM's method inlining heuristic requires re-tuning to take advantage of smaller method sizes, especially for Scala.

The unfiltered JRuby and Jython heat maps (Figure 5) show their object lifetimes are either very short or near immortal. Garbage collection optimizations might include aggressive nursery region resizing and predictive object pretenuring.

References

[1] 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, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proc. OOPSLA*, pages 169–190, 2006.

[2] K. Chow, A. Wright, and K. Lai. Characterization of Java workloads by principal components analysis and indirect branches. In *Proc. Workshop on Workload Characterization*, pages 11–19, 1998.

[3] C. Collberg, G. Myles, and M. Stepp. An Empirical Study of Java Bytecode Programs. *Software: Practice and Experience*, 37(6):581–641, 2007.

[4] J. R. Dick, K. B. Kent, and J. C. Libby. A quantitative analysis of the .NET common language runtime. *Journal of Systems Architecture*, 54(7):679–696, 2008.

[5] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proc. ECOOP*, pages 92–115, 1999.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic Metrics for Java. In *Proc. OOPSLA*, pages 149–168, 2003.

[7] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level Phase Behavior in Java Workloads. In *Proc. OOPSLA*, pages 270–287, 2004.

[8] J. Gough. *Compiling for the .NET Common Language Runtime (CLR)*. Prentice Hall, 2002.

[9] R. Hickey. The Clojure Programming Language. In *Proc. Symposium on Dynamic languages*, pages 1:1–1:1, 2008.

[10] R. E. Jones and C. Ryder. A Study of Java Object Demographics. In *Proc. ISMM*, pages 121–130, 2008.

[11] D. E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[12] W. L. Martinez, A. Martinez, and J. Solka. *Exploratory Data Analysis with MATLAB*. Chapman and Hall/CRC, 2004.

[13] F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the design of the R language. In *Proc. ECOOP*, pages 104–131, 2012.

[14] C. O. Nutter, T. Enebo, N. Sieger, O. Bini, and I. Dees. *Using JRuby: Bringing Ruby to Java*. Pragmatic Bookshelf, 2011.

[15] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004.

[16] D. O'Donoghue, A. Leddy, J. Power, and J. Waldron. Bigram Analysis of Java Bytecode Sequences. In *Proc. PPPJ/IRE*, pages 187–192, 2002.

[17] S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly, 2002.

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

[19] R. Radhakrishnan, J. Rubio, and L. John. Characterization of Java Applications at Bytecode and Ultra-SPARC Machine Code Levels. In *Proc. ICCD*, pages 281–284, 1999.

[20] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Trans. on Computers*, 50(2):131–146, 2001.

[21] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant Tracks: Generating Program Traces with Object Death Records. In *Proc. PPPJ*, pages 139–142, 2011.

[22] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: portable production of complete and precise GC traces. In *Proc. ISMM*, pages 109–118, 2013.

[23] 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 *Proc. PPPJ*, pages 11–20, 2011.

[24] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, and D. Ansaloni. A comprehensive toolchain for workload characterization across JVM languages. In *Proc. PASTE*, pages 9–16, 2013.

[25] 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 *Proc. ISMM*, pages 97–108, 2012.

[26] 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 *Proc. OOPSLA*, pages 657–676, 2011.

[27] L. I. Smith. A Tutorial on Principal Components Analysis. Technical report, Cornell University, USA, 2002.

[28] B. Stephenson and W. Holst. A quantitative analysis of Java bytecode sequences. In *Proc. PPPJ*, pages 15–20, 2004.

[29] T. Suganuma, T. Yasue, and T. Nakatani. An Empirical Study of Method Inlining for a Java Just-in-Time Compiler. In *Proc. Java Virtual Machine Research and Technology Symposium*, pages 91–104, 2002.

[30] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The Qualitas Corpus: A curated collection of Java code for empirical studies. In *Proc. 17th Asia Pacific Software Engineering Conference*, pages 336–345, 2010.

[31] E. Tempero, J. Noble, and H. Melton. How do Java Programs use Inheritance? an Empirical Study of Inheritance in Java Software. In *Proc. ECOOP*, pages 667–691, 2008.

[32] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGSOFT Software Engineering Notes*, 9(3):157–167, 1984.

[33] R. Yoo, H.-H. Lee, H. Lee, and K. Chow. Hierarchical means: Single number benchmarking with workload cluster analysis. In *Proc. International Symposium on Workload Characterization*, pages 204–213, 2007.

[34] D. Zaparanuks and M. Hauswirth. Characterizing the Design and Performance of Interactive Java Applications. In *International Symposium on Performance Analysis of Systems and Software*, pages 23–32, 2010.

language/benchmark	*compiled*	*bytecodes*	*non-Java*	*methods*	*non-Java*	*method*	*basic block*	*objects*	*non-Java*	*boxed*
	or	*executed*	*bytecodes*	*executed*	*methods*	*hotness*	*hotness*	*allocated*	*objects*	*primitive*
	interpreted		*%*		*%*	*%*	*%*		*%*	*use %*
Java/binarytrees	AOT	170105156	0.00	25176723	0.00	99.81	99.99	2651267	0.00	0.00
Java/fannkuchredux	AOT	141676329	0.00	1003221	0.00	99.96	99.99	2293	0.00	0.26
Java/fasta	AOT	100060247	0.00	2456162	0.00	76.76	99.99	2343	0.00	0.26
Java/knucleotide	AOT	1202353259	0.00	49678282	0.00	50.24	99.99	572950	0.00	0.00
Java/mandelbrot	AOT	262768004	0.00	78893	0.00	99.97	100.00	2878	0.00	0.21
Java/meteor	AOT	330191630	0.00	5431167	0.00	99.48	99.99	262824	0.00	0.00
Java/nbody	AOT	171061794	0.00	152240	0.00	99.47	99.98	4698	0.00	0.17
Java/regexdna	AOT	3947367657	0.00	226454549	0.00	99.68	100.00	312589	0.00	0.09
Java/revcomp	AOT	54864024	0.00	1548046	0.00	6.69	99.97	2868	0.00	9.24
Java/spectralnorm	AOT	38870722	0.00	1079088	0.00	99.76	99.94	4785	0.00	0.13
Java/avrora	AOT	4043803001	0.00	322511916	0.00	99.32	99.96	991319	0.00	0.12
Java/batik	AOT	812845536	0.00	40670592	0.00	74.52	99.75	ET failed to complete trace		
Java/fop	AOT	149718163	0.00	9541593	0.00	90.68	98.88	ET failed to complete trace		
Java/h2	AOT	9653613964	0.00	769997845	0.00	93.64	99.98	25548956	0.00	1.92
Java/javac/sunflow	AOT	16635753858	0.00	1077651364	0.00	88.01	99.50	ET failed to complete trace		
Java/luindex	AOT	123562365	0.00	4524912	0.00	92.91	99.86	112724	0.00	0.01
Java/lusearch	AOT	1140089272	0.00	50417227	0.00	99.17	99.96	1272318	0.00	0.00
Java/pmd	AOT	54097593	0.00	2366460	0.00	91.24	99.15	133466	0.00	0.26
Java/sunflow	AOT	1987247487	0.00	64365743	0.00	99.09	99.98	2422198	0.00	0.03
Java/xalan	AOT	887655171	0.00	48845720	0.00	97.35	99.41	1117739	0.00	0.09
Clojure/binarytrees	AOT	458013959	50.18	46066179	23.11	99.81	99.93	5519929	48.01	47.82
Clojure/fannkuchredux	AOT	949916910	32.01	77452181	1.68	98.21	99.96	258837	0.67	0.79
Clojure/fasta	AOT	430594595	40.53	37624799	9.69	90.69	99.93	2641156	45.50	45.64
Clojure/knucleotide	AOT	1741954505	23.02	131910496	4.43	80.60	99.97	7165252	4.66	85.27
Clojure/mandelbrot	AOT	697889749	62.96	42874651	15.85	99.82	99.94	821435	0.48	70.20
Clojure/meteor	AOT	4097931808	28.21	389333271	3.91	99.02	99.98	10199180	9.40	25.69
Clojure/nbody	AOT	292196152	66.32	17028481	39.02	99.18	99.90	1239830	0.13	80.85
Clojure/regexdna	AOT	4689013910	0.54	226505597	0.02	99.43	99.99	507349	0.43	0.40
Clojure/revcomp	AOT	224945319	44.47	18146640	0.55	55.92	99.86	233852	0.57	1.29
Clojure/spectralnorm	AOT	152301546	38.49	7716311	0.80	99.38	99.78	259541	3.25	3.23
Clojure/incanter	JIT	273986921	4.79	17850648	4.48	96.72	99.03	1635693	1.78	1.33
Clojure/leiningen	AOT	1006923598	2.18	57874646	2.32	97.64	99.49	1278616	1.94	1.68
Clojure/noir/blog	AOT	1323734731	1.37	21635558	5.04	99.81	99.90	1909561	1.09	1.11
JRuby/binarytrees	AOT	3172344964	12.57	354573991	6.27	99.98	99.99	12667277	0.01	0.00
JRuby/fannkuchredux	AOT	4684146884	13.34	537414857	3.14	89.65	100.00	3272573	0.04	0.01
JRuby/fasta	AOT	5564934530	6.21	617823701	2.41	99.97	99.99	14888989	0.01	32.24
JRuby/knucleotide	AOT	11381452118	3.63	1081798483	2.46	99.15	100.00	18740991	0.01	0.00
JRuby/mandelbrot	AOT	13483710728	8.53	1654536903	0.80	99.97	99.99	ET failed to complete trace		
JRuby/meteor	AOT	25990259655	6.26	2455403535	2.72	99.99	100.00	ET failed to complete trace		
JRuby/nbody	AOT	7054115048	8.78	950461750	0.81	99.97	100.00	30638260	0.00	94.66
JRuby/regexdna	AOT	10090042993	0.00	628235221	0.00	99.87	100.00	336937	0.41	0.10
JRuby/revcomp	AOT	277284831	1.52	18636289	1.48	76.62	99.92	333311	0.41	0.11
JRuby/spectralnorm	AOT	2183551569	9.14	272266633	3.32	99.96	99.99	8515852	0.02	96.21
JRuby/jrails	AOT	5197249613	0.47	313574609	1.08	98.24	99.77	ET failed to complete trace		
JRuby/lingo	JIT	12322206926	2.36	1110460420	1.73	98.88	99.88	ET failed to complete trace		
JRuby/warbler	JIT	7943627729	1.33	526990999	1.02	99.22	99.84	ET failed to complete trace		
Jython/binarytrees	int.	6116792936	7.77	481948160	2.21	99.84	99.99	27183665	0.04	19.42
Jython/fannkuchredux	int.	4429512501	4.99	452006371	0.00	94.81	99.98	20308641	0.01	1.72
Jython/fasta	int.	7361689841	6.92	758276792	0.98	99.21	99.98	11091087	0.19	44.00
Jython/knucleotide	int.	10899263124	3.79	1142232229	0.00	95.75	99.99	40776401	0.03	32.48
Jython/mandelbrot	int.	6260601597	5.58	645681679	0.01	94.31	99.99	20477645	0.01	95.81
Jython/nbody	int.	6656716672	7.91	899402775	0.00	91.93	99.99	33011071	0.00	82.28
Jython/regexdna	int.	7144141615	0.03	432717385	0.02	86.18	99.97	136423507	0.02	0.19
Jython/revcomp	int.	1164263231	0.20	78797100	0.04	92.18	99.81	3090414	0.05	3.64
Jython/spectralnorm	int.	2061855998	5.75	225838375	1.37	96.72	99.96	16063707	0.05	57.64
Scala/binarytrees	AOT	116374550	98.60	8209028	98.56	99.67	99.92	2666944	99.37	0.40
Scala/fannkuchredux	AOT	292713859	99.59	27164782	99.67	99.96	99.99	6612	5.05	15.62
Scala/fasta	AOT	247153292	81.63	29521359	66.34	79.92	99.99	3621	4.97	14.50
Scala/knucleotide	AOT	2743126868	92.52	279800678	91.54	96.91	99.99	4329096	32.82	65.83
Scala/mandelbrot	AOT	281828727	99.48	12620894	99.27	99.93	99.98	19299	57.46	5.72
Scala/meteor	AOT	38201494343	97.60	4688797729	95.62	99.99	100.00	2340559	98.48	0.88
Scala/nbody	AOT	307725221	99.67	41360373	99.86	99.60	99.99	5920	2.57	6.05
Scala/regexdna	AOT	3681464412	14.54	246469635	36.78	99.93	100.00	6555923	26.30	0.01
Scala/revcomp	AOT	69607945	98.85	3079180	98.42	14.09	99.97	3181	5.85	7.17
Scala/spectralnorm	AOT	44336176	97.61	1110874	94.41	99.75	99.92	6546	9.21	5.44
Scala/apparat	AOT	5473129320	69.78	782465549	59.90	99.38	99.89	8828083	69.63	0.45
Scala/kiama	AOT	96747090	44.44	8744873	72.05	95.47	99.41	534871	60.45	1.64
Scala/scalac	AOT	841758028	50.77	86270143	77.48	97.11	99.82	4384353	59.82	1.09
Scala/scaladoc	AOT	1009919955	56.19	111072697	76.91	97.16	99.82	4283738	62.97	0.71
Scala/scalap	AOT	50006991	23.40	3387070	53.31	94.23	99.45	166594	43.44	1.52
Scala/scalariform	AOT	665331715	76.76	102056949	88.80	94.47	99.25	5121304	82.18	4.53
Scala/scalatest	AOT	867200843	30.23	61964606	67.16	99.90	99.96	3503330	46.58	0.13
Scala/scalaxb	AOT	371552561	47.38	35445090	72.17	94.64	99.54	1432529	52.82	21.09
Scala/specs	AOT	721279181	20.13	49903727	55.97	99.76	99.90	3872916	41.33	0.90
Scala/tmt	AOT	44096213851	79.66	4757318965	80.30	97.88	100.00	110104114	9.14	74.64

Table 5: Summary of dynamic metrics obtained, each benchmark occupies a single row in the table.