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.

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 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 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, 2013. This is the author’s version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in PPP'13, September 11–13, http://dx.doi.org/10.1145/2500828.2500838.

Copyright © ACM [to be supplied]…$15.00. http://dx.doi.org/10.1145/1\textsuperscript{c}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.

2.1 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.

Scala [13] 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 (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 gather data from the benchmarks using two profilers, as shown in Figure 1. We use a Perl script to simulate user interaction with the blog site while it was being profiled.

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.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>workload</th>
<th>int</th>
<th>fp</th>
<th>ptr</th>
<th>str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary-Trees</td>
<td>tree depth 13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fannikash-Rex</td>
<td>sequence length 9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fasta</td>
<td>150000 sequences</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>K-Nucleotide</td>
<td>Fasta output</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Mandelbrot</td>
<td>500 image size</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Meteor-Contest</td>
<td>2098 solutions</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>N-body</td>
<td>100000 steps</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Regex-DNA</td>
<td>Fasta output</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Reverse-Complement</td>
<td>Fasta output</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Spectral-Norm</td>
<td>160 approximations</td>
<td></td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

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 profiled 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

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, bytecode, methods and objects implemented in its own language. Neither JP2 or ET provide complete coverage of all the classes used by JVM languages. 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, bytecode, methods and objects implemented in its own language. Neither JP2 or ET provide complete coverage of all the classes used by JVM languages. 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.

Table 5 details whether each benchmark is compiled to bytecode, bytecode, methods and objects implemented in its own language. Review of the source code of each benchmark reveals that JRuby and Jython may not benefit greatly from optimizations based on differences between Java and non-Java programming languages. To verify this, we separate Java and non-Java bytecode, 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 whether each benchmark is compiled to bytecode, bytecode, methods and objects implemented in its own language. Review of the source code of each benchmark reveals that JRuby and Jython may not benefit greatly from optimizations based on differences between Java and non-Java programming languages. To verify this, we separate Java and non-Java bytecode, 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.

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 5 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 languages. To verify this, we separate Java and non-Java bytecode, 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.

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 Leinigen 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.
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 2 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 2 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 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 have 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.
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.
Table 4: Number of N-grams used by non-Java benchmarks that are not used by Java benchmarks. For 2 ≤ N ≤ 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.

<table>
<thead>
<tr>
<th>language</th>
<th>filtered?</th>
<th>1-gram</th>
<th>2-gram</th>
<th>3-gram</th>
<th>4-gram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java</td>
<td>No</td>
<td>192</td>
<td>5772</td>
<td>21864</td>
<td>72032</td>
</tr>
<tr>
<td>Clojure</td>
<td>Yes</td>
<td>118</td>
<td>452</td>
<td>3930</td>
<td>7813</td>
</tr>
<tr>
<td>Clojure</td>
<td>No</td>
<td>177</td>
<td>4002</td>
<td>19474</td>
<td>40165</td>
</tr>
<tr>
<td>JRuby</td>
<td>Yes</td>
<td>54</td>
<td>391</td>
<td>1212</td>
<td>2585</td>
</tr>
<tr>
<td>JRuby</td>
<td>No</td>
<td>54</td>
<td>4482</td>
<td>26373</td>
<td>63499</td>
</tr>
<tr>
<td>Jython</td>
<td>Yes</td>
<td>48</td>
<td>422</td>
<td>1055</td>
<td>1964</td>
</tr>
<tr>
<td>Jython</td>
<td>No</td>
<td>178</td>
<td>3427</td>
<td>14887</td>
<td>27852</td>
</tr>
<tr>
<td>Scala</td>
<td>Yes</td>
<td>163</td>
<td>2624</td>
<td>11979</td>
<td>30164</td>
</tr>
<tr>
<td>Scala</td>
<td>No</td>
<td>187</td>
<td>3995</td>
<td>19515</td>
<td>49591</td>
</tr>
</tbody>
</table>

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, Ruby-
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.
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.
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. [23][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


<table>
<thead>
<tr>
<th>language/benchmark</th>
<th>compiled or interpreted</th>
<th>bytecode executed</th>
<th>non-Java bytecode executed %</th>
<th>methods executed</th>
<th>non-Java methods executed %</th>
<th>method hotness %</th>
<th>basic block hotness %</th>
<th>objects allocated</th>
<th>non-Java objects allocated %</th>
<th>boxed primitives use %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java/binarytrees</td>
<td>AOT</td>
<td>313895496</td>
<td>12.59</td>
<td>61446679</td>
<td>20.82</td>
<td>99.92</td>
<td>99.98</td>
<td>99.98</td>
<td>1260790</td>
<td>0.00</td>
</tr>
<tr>
<td>Java/fannkuchredux</td>
<td>AOT</td>
<td>4681486488</td>
<td>13.34</td>
<td>51438417</td>
<td>1.66</td>
<td>96.90</td>
<td>96.90</td>
<td>96.90</td>
<td>532753</td>
<td>0.04</td>
</tr>
<tr>
<td>Java/fasta</td>
<td>AOT</td>
<td>5654994350</td>
<td>6.21</td>
<td>617823701</td>
<td>2.41</td>
<td>99.97</td>
<td>99.97</td>
<td>99.97</td>
<td>48889898</td>
<td>0.01</td>
</tr>
<tr>
<td>Java/knucleotide</td>
<td>AOT</td>
<td>1138452118</td>
<td>3.63</td>
<td>108179843</td>
<td>2.46</td>
<td>99.15</td>
<td>99.15</td>
<td>99.15</td>
<td>18740991</td>
<td>0.00</td>
</tr>
<tr>
<td>Java/mandelbrot</td>
<td>AOT</td>
<td>2097005655</td>
<td>6.26</td>
<td>2455403535</td>
<td>2.72</td>
<td>99.99</td>
<td>99.99</td>
<td>99.99</td>
<td>31283900</td>
<td>0.01</td>
</tr>
<tr>
<td>Java/nbody</td>
<td>AOT</td>
<td>7054115048</td>
<td>8.78</td>
<td>904506170</td>
<td>0.81</td>
<td>99.97</td>
<td>99.97</td>
<td>99.97</td>
<td>30638260</td>
<td>0.00</td>
</tr>
<tr>
<td>Java/regexprx</td>
<td>AOT</td>
<td>4898013910</td>
<td>0.54</td>
<td>226505597</td>
<td>0.02</td>
<td>94.49</td>
<td>94.49</td>
<td>94.49</td>
<td>5073490</td>
<td>0.43</td>
</tr>
<tr>
<td>Java/spectralnorm</td>
<td>AOT</td>
<td>1720305146</td>
<td>34.47</td>
<td>18146460</td>
<td>0.55</td>
<td>95.92</td>
<td>95.92</td>
<td>95.92</td>
<td>2338522</td>
<td>0.57</td>
</tr>
<tr>
<td>Java/scalaxb</td>
<td>AOT</td>
<td>237284318</td>
<td>1.52</td>
<td>18632693</td>
<td>1.48</td>
<td>76.62</td>
<td>76.62</td>
<td>76.62</td>
<td>333311</td>
<td>0.41</td>
</tr>
<tr>
<td>Java/scalatest</td>
<td>AOT</td>
<td>2132208626</td>
<td>2.36</td>
<td>1130404020</td>
<td>1.73</td>
<td>98.88</td>
<td>98.88</td>
<td>98.88</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
<tr>
<td>Java/scalap</td>
<td>AOT</td>
<td>796227729</td>
<td>1.32</td>
<td>272266633</td>
<td>3.32</td>
<td>99.96</td>
<td>99.96</td>
<td>99.96</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
<tr>
<td>Java/spectralnorm</td>
<td>JIT</td>
<td>6116790296</td>
<td>7.77</td>
<td>481948180</td>
<td>2.21</td>
<td>99.84</td>
<td>99.84</td>
<td>99.84</td>
<td>27183655</td>
<td>0.19</td>
</tr>
<tr>
<td>Java/xalan</td>
<td>AOT</td>
<td>887655117</td>
<td>0.00</td>
<td>84854720</td>
<td>0.00</td>
<td>97.35</td>
<td>97.35</td>
<td>97.35</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
<tr>
<td>Java/regexdna</td>
<td>AOT</td>
<td>748736484</td>
<td>0.00</td>
<td>51306184</td>
<td>0.00</td>
<td>99.96</td>
<td>99.96</td>
<td>99.96</td>
<td>478508</td>
<td>0.13</td>
</tr>
<tr>
<td>Java/nbody</td>
<td>AOT</td>
<td>640750929</td>
<td>0.00</td>
<td>40670592</td>
<td>0.00</td>
<td>97.42</td>
<td>97.42</td>
<td>97.42</td>
<td>312859</td>
<td>0.00</td>
</tr>
<tr>
<td>Java/regexprx</td>
<td>AOT</td>
<td>4898013910</td>
<td>0.54</td>
<td>226505597</td>
<td>0.02</td>
<td>94.49</td>
<td>94.49</td>
<td>94.49</td>
<td>5073490</td>
<td>0.43</td>
</tr>
<tr>
<td>Java/spectralnorm</td>
<td>AOT</td>
<td>1720305146</td>
<td>34.47</td>
<td>18146460</td>
<td>0.55</td>
<td>95.92</td>
<td>95.92</td>
<td>95.92</td>
<td>2338522</td>
<td>0.57</td>
</tr>
<tr>
<td>Java/scalaxb</td>
<td>AOT</td>
<td>237284318</td>
<td>1.52</td>
<td>18632693</td>
<td>1.48</td>
<td>76.62</td>
<td>76.62</td>
<td>76.62</td>
<td>333311</td>
<td>0.41</td>
</tr>
<tr>
<td>Java/scalatest</td>
<td>AOT</td>
<td>2132208626</td>
<td>2.36</td>
<td>1130404020</td>
<td>1.73</td>
<td>98.88</td>
<td>98.88</td>
<td>98.88</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
<tr>
<td>Java/scalap</td>
<td>AOT</td>
<td>796227729</td>
<td>1.32</td>
<td>272266633</td>
<td>3.32</td>
<td>99.96</td>
<td>99.96</td>
<td>99.96</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
<tr>
<td>Java/spectralnorm</td>
<td>JIT</td>
<td>6116790296</td>
<td>7.77</td>
<td>481948180</td>
<td>2.21</td>
<td>99.84</td>
<td>99.84</td>
<td>99.84</td>
<td>27183655</td>
<td>0.19</td>
</tr>
<tr>
<td>Java/xalan</td>
<td>AOT</td>
<td>887655117</td>
<td>0.00</td>
<td>84854720</td>
<td>0.00</td>
<td>97.35</td>
<td>97.35</td>
<td>97.35</td>
<td>ET failed to complete trace</td>
<td></td>
</tr>
</tbody>
</table>

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