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

Wing Hang Li, David R. White & Jeremy Singer
Background – the JVM

- One reason for Java’s success is the Java Virtual Machine
- The JVM provides:
  - “Write once, run anywhere” capability (WORA)
  - Sandboxed execution environment
  - Automatic memory management
  - Adaptive optimisation

- New Trend – WOIALRA
- write once in any language run anywhere
Other JVM Programming Languages

- Clojure, JRuby, Jython and Scala are popular JVM languages

Language features:
- Clojure, JRuby and Jython are dynamically typed
- JRuby and Jython are scripting languages
- Clojure is a functional language
- Scala is multi-paradigm
Growing Popularity of JVM languages

Top reasons are:

- Access new features
- Interoperability allows existing Java libraries to be used
- Use existing frameworks on the JVM (JRuby on Rails for instance)

Twitter uses Scala:

- Flexibility
- Concurrency
## JVM Languages in the Real World

<table>
<thead>
<tr>
<th>Language</th>
<th>Company 1</th>
<th>Company 2</th>
<th>Company 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clojure</td>
<td>youcaneat.at</td>
<td>sonian</td>
<td></td>
</tr>
<tr>
<td>JRuby</td>
<td>PONS.eu</td>
<td>Project Kenai</td>
<td></td>
</tr>
<tr>
<td>Jython</td>
<td>IBM</td>
<td></td>
<td>ORACLE</td>
</tr>
<tr>
<td></td>
<td>WebSphere</td>
<td></td>
<td>Weblogic Server</td>
</tr>
<tr>
<td>Scala</td>
<td>twitter</td>
<td>LinkedIn</td>
<td>foursquare</td>
</tr>
</tbody>
</table>
What’s the Catch?

- The JVM was designed to run Java code
- Other JVM languages have:
  - Poor performance
  - Use more memory

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Scala</th>
<th>Clojure</th>
<th>JRuby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>1.92</td>
<td>2.30</td>
<td>4.10</td>
<td>50.23</td>
</tr>
</tbody>
</table>

How much slower each language performs compared to the fastest time.

Figures from the Computer Languages Benchmark Game
Why are Non-Java Languages Slower?

- What are the differences between Java and the other JVM languages?
- Work on improving performance has usually been on the programming language side
- New `INVOKE_DYNAMIC` instruction in JVM 1.7
Truffle/Graal Approach

- Oracle Labs
- Graal - plugin your own intermediate representations and optimisations
- Truffle – produce an abstract syntax tree from source code and run it using an interpreter
- “One VM to rule them all”
- Our approach is different because we examine the JVM language behavior
Aim of our Study

- This study is the first stage of a project to improve the performance of non-Java JVM languages.
- We do this by profiling benchmarks written in Java, Clojure, JRuby, Jython and Scala.
- We found differences in their characteristics that may be exploitable for optimisations.
Data Gathering and Analysis

**Benchmarks**
- Java
- Clojure
- JRuby
- Jython
- Scala

**Data Gathering**
- JVM
  - Garbage Collection Traces
  - Dynamic Bytecode Traces
  - Method Level
    - Call/Ret Events
  - Instruction Level
    - Instruction Mix
  - Object Level
    - Object Creation and Deaths

**Exploratory Data Analysis**
- Object Demographics
- Principal Components Analysis
- N-Gram Models
Profiling Tools

- **JP2\(^1\) profiler:**
  - Proportion of Java and non-Java bytecode
  - Frequency of different instructions
  - Method and basic block frequencies and sizes
  - Produce N-grams from JP2 output

- **Elephant Tracks\(^2\) heap profiler:**
  - Object allocations and deaths
  - Object size
  - Pointer updates
  - Stack depth at method entry and exit for each thread

---

\(^1\) [http://code.google.com/p/jp2/](http://code.google.com/p/jp2/)

Benchmarks

- Obtained from the Computer Languages Benchmarks Game\(^1\)
  - The same algorithm is implemented in each programming language
  - Well known problems like N-body, Mandelbrot and Meteor puzzle
  - Benchmarks available in Java, Clojure, JRuby, Python and Scala

\(^1\) [http://shootout.alioth.debian.org/](http://shootout.alioth.debian.org/)
Benchmarks

- **Java**
  - DaCapo benchmark suite

- **Clojure**
  - Noir – web application framework
  - Leiningen – project automation
  - Incanter – R like statistical calculation and graphs

- **JRuby**
  - Ruby on Rails – web application framework
  - Warbler – converts Ruby applications into a jar or war
  - Lingo – automatic indexing of scientific texts

- **Scala**
  - Scala Benchmark Suite
Problems Encountered

- Non-Java programming languages use Java
  - Java library
  - JRuby and Jython are implemented in Java
- Can be mitigated by filtering out methods and objects using source file metadata
- We examine the amount of Java code in each non-Java language library
Non-Java Code in JVM Language Libraries

- Static analysis of each non-Java language library

<table>
<thead>
<tr>
<th>Language</th>
<th>Classes</th>
<th>Methods</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jython</td>
<td>68%</td>
<td>86%</td>
<td>96%</td>
</tr>
<tr>
<td>JRuby</td>
<td>65%</td>
<td>87%</td>
<td>98%</td>
</tr>
<tr>
<td>Clojure</td>
<td>24%</td>
<td>33%</td>
<td>24%</td>
</tr>
<tr>
<td>Scala</td>
<td>3%</td>
<td>1%</td>
<td>3%</td>
</tr>
</tbody>
</table>
Analysis tools

- Principal Component Analysis using MATLAB
  - Can be used for dimension reduction
  - Spot patterns or features when projected to fewer dimensions

- Object Demographics
  - Memory behaviour of objects
  - Size and lifetime of objects

- Exploratory Data Analysis
  - Spot patterns or features using various graphical techniques
  - Principal component analysis and boxplots

---

1 Exploratory Data Analysis with MATLAB by W.L. Martinez, A. Martinez and J. Solka.
Instruction Level Results

- Variety of n-grams used

<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>31864</td>
<td>73033</td>
</tr>
<tr>
<td>Clojure</td>
<td>No</td>
<td>177</td>
<td>4002</td>
<td>19474</td>
<td>40165</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>118</td>
<td>1217</td>
<td>3930</td>
<td>7813</td>
</tr>
<tr>
<td>JRuby</td>
<td>No</td>
<td>179</td>
<td>4482</td>
<td>26373</td>
<td>64399</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>54</td>
<td>391</td>
<td>1212</td>
<td>2585</td>
</tr>
<tr>
<td>Jython</td>
<td>No</td>
<td>178</td>
<td>3427</td>
<td>14887</td>
<td>27852</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>48</td>
<td>422</td>
<td>1055</td>
<td>1964</td>
</tr>
<tr>
<td>Scala</td>
<td>No</td>
<td>187</td>
<td>3995</td>
<td>19515</td>
<td>45951</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>163</td>
<td>2624</td>
<td>11979</td>
<td>30164</td>
</tr>
</tbody>
</table>
### Instruction Level Results

- **N-grams not used by Java**

<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>Clojure</td>
<td>No</td>
<td>2</td>
<td>348 (5%)</td>
<td>4578 (23%)</td>
<td>15824 (43%)</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>2</td>
<td>193 (11%)</td>
<td>1957 (46%)</td>
<td>6264 (77%)</td>
</tr>
<tr>
<td>JRuby</td>
<td>No</td>
<td>1</td>
<td>512 (1%)</td>
<td>7659 (8%)</td>
<td>30574 (26%)</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>1</td>
<td>44 (2%)</td>
<td>399 (14%)</td>
<td>1681 (42%)</td>
</tr>
<tr>
<td>Jython</td>
<td>No</td>
<td>1</td>
<td>161 (1%)</td>
<td>2413 (6%)</td>
<td>8628 (19%)</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>1</td>
<td>38 (7%)</td>
<td>412 (19%)</td>
<td>1491 (56%)</td>
</tr>
<tr>
<td>Scala</td>
<td>No</td>
<td>0</td>
<td>335 (2%)</td>
<td>4863 (23%)</td>
<td>21106 (59%)</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>0</td>
<td>288 (3%)</td>
<td>4168 (27%)</td>
<td>18676 (69%)</td>
</tr>
</tbody>
</table>
Instruction Level Results

- Principal components analysis (1-gram, filtered)
Instruction Level Results

- Principal components analysis (2-gram, filtered)
Instruction Level Results

- Principal components analysis (2-gram, filtered)

We observe that, after filtering, **JRuby** and **Jython** use a different mix of 1 and 2-grams compared to the other JVM languages.
Instruction Level Results

- Principal components analysis (1-gram, unfiltered)
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)

Without filtering there is no distinct clustering observed
Method Level Results - Java

- Results for the distribution of method sizes
Method Level Results - Scala

- Results for the distribution of method sizes (filtered)
Method Level Results - Scala

- Results for the distribution of method sizes (filtered)

We observe that Scala methods are generally smaller than Java methods.
Method Level Results - Java

- Results for the distribution of method stack depths
Method Level Results - Scala

- Results for the distribution of method stack depths
Method Level Results - Scala

- Results for the distribution of method stack depths

We observe that stack depths are generally greater for Scala applications compared to Java applications
Object Level Results - Java

- Object lifetime

![Heatmap of Object Lifetime](image)
Object Level Results - Scala

- Object lifetime (filtered)
We observed that more Scala objects have a short lifetime compared to Java.
Object Sizes - Java

- Results for the distribution of object sizes (filtered)
Object Sizes - Clojure

- Results for the distribution of object sizes (filtered)
We observed that Clojure generally uses objects that are smaller than Java objects
Other Results

- All benchmarks showed a high level of method and basic block hotness. There were no significant differences between JVM-hosted languages.

- Non-Java JVM languages are more likely to use boxed primitives.
Future Work

- Examine the programming language characteristics to find opportunities for:
  - Tuning existing optimisations
  - Proposing new optimisations
- Implement these in a JVM to see if performance has improved
Conclusions

- Aim of study is to investigate the reasons for the poor performance of JVM languages
- Benchmarks in 5 JVM languages were profiled
- JVM languages do have distinctive characteristics related to their features
- Next step is to optimise performance using the observed characteristics

Our research paper, experimental scripts and results are available at: http://bit.ly/19JsrKf
Questions?
More Method Size Graphs - Clojure

- Results for the distribution of method sizes (filtered)
More Method Size Graphs - JRuby

- Results for the distribution of method sizes (unfiltered)
More Method Stack Depth Results - Clojure
More Method Stack Depth Results - JRuby
More Object Lifetime Graphs - JRuby

The diagram shows the object lifetime graphs for JRuby (unfiltered). The x-axis represents the percentage of the maximum object lifetime, while the y-axis lists benchmark names. The color scale on the right indicates the percentage of object deaths.
More Object Lifetime Graphs - Jython

![Graph showing object lifetime statistics for Jython with benchmark names and percentage of maximum object lifetime on the x-axis, and percentage of object deaths on the y-axis.](image-url)
More Object Size Graphs - Scala

- Results for the distribution of object sizes (filtered)
More Object Size Graphs - JRuby

- Results for the distribution of object sizes (unfiltered)