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

Wing Hang Li, David R. White & Jeremy Singer

University of Glasgow | School of Computing Science

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

Clojure	youcaneat.at		sonian cloud-powered archiving
JRuby	PONS.eu		Project Kenai beta
Jython	IBM WebSphere		ORACLE Weblogic Server
Scala	twitter	LinkedIn	foursquare

What's the Catch?

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

Java	Scala	Clojure	JRuby
1.92	2.30	4.10	50.23

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 `INVOKEDYNAMIC` instruction in JVM 1.7

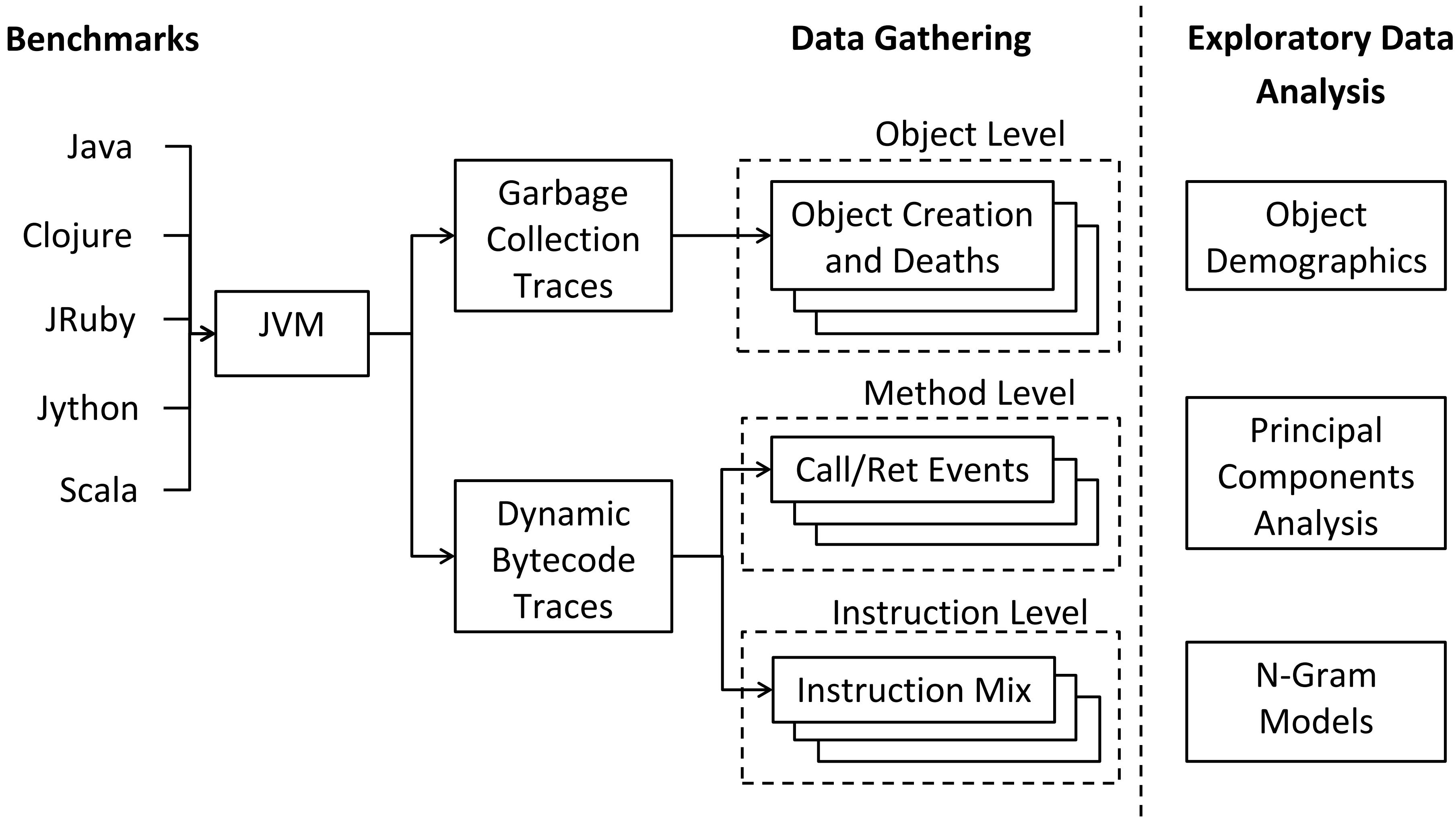
Truffle/Graal Approach

- 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



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/
[2] http://www.cs.tufts.edu/research/redline/elephantTracks/

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/

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

Language	Classes	Methods	Instructions
Jython	68%	86%	96%
JRuby	65%	87%	98%
Clojure	24%	33%	24%
Scala	3%	1%	3%

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[1]
  - 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

Language	Filtered	1-gram	2-gram	3-gram	4-gram
Java	No	192	5772	31864	73033
Clojure	No	177	4002	19474	40165
	Yes	118	1217	3930	7813
JRuby	No	179	4482	26373	64399
	Yes	54	391	1212	2585
Jython	No	178	3427	14887	27852
	Yes	48	422	1055	1964
Scala	No	187	3995	19515	45951
	Yes	163	2624	11979	30164

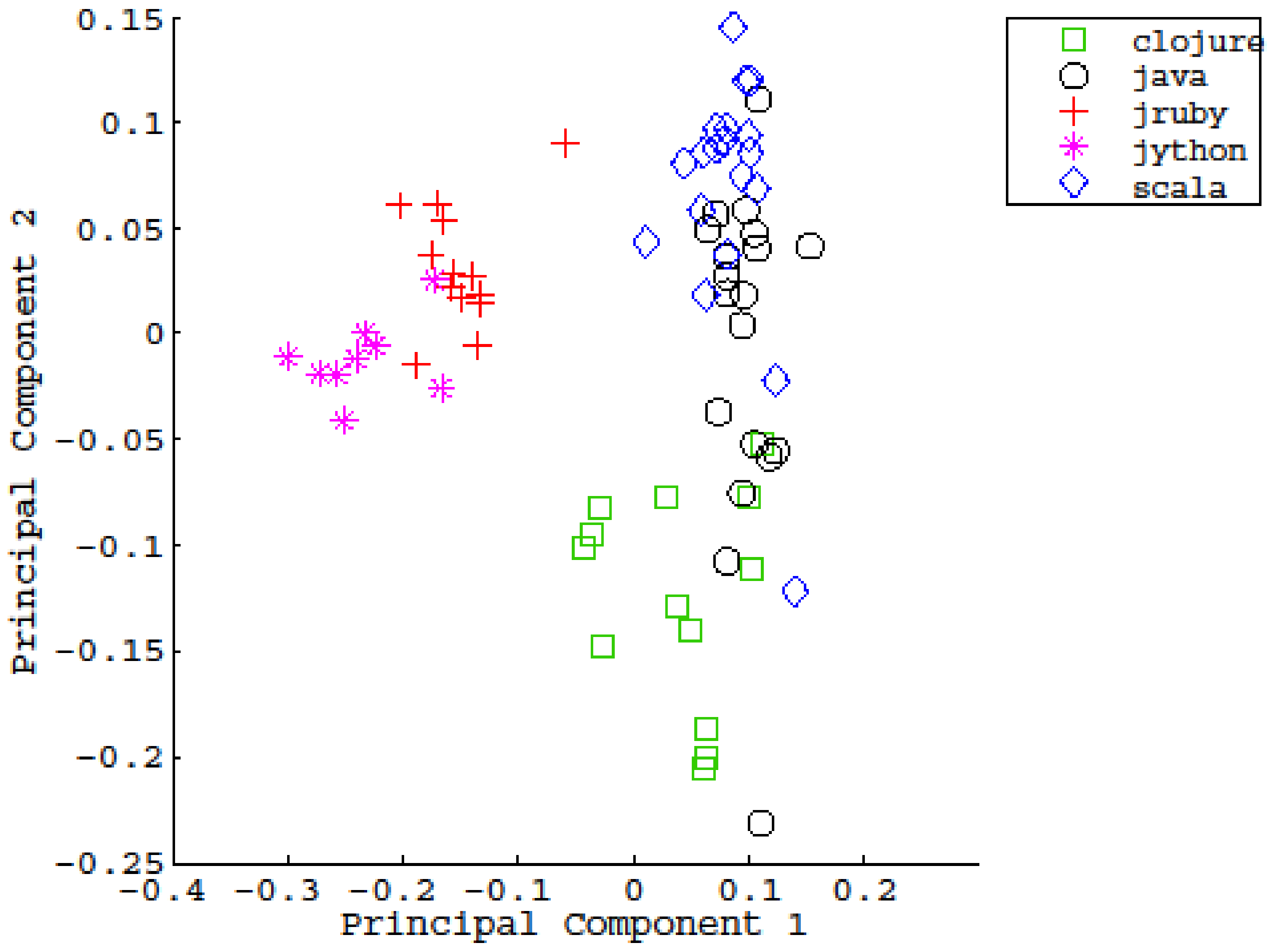
Instruction Level Results

- N-grams not used by Java

Language	Filtered	1-gram	2-gram	3-gram	4-gram
Clojure	No	2	348 (5%)	4578 (23%)	15824 (43%)
	Yes	2	193 (11%)	1957 (46%)	6264 (77%)
JRuby	No	1	512 (1%)	7659 (8%)	30574 (26%)
	Yes	1	44 (2%)	399 (14%)	1681 (42%)
Jython	No	1	161 (1%)	2413 (6%)	8628 (19%)
	Yes	1	38 (7%)	412 (19%)	1491 (56%)
Scala	No	0	335 (2%)	4863 (23%)	21106 (59%)
	Yes	0	288 (3%)	4168 (27%)	18676 (69%)

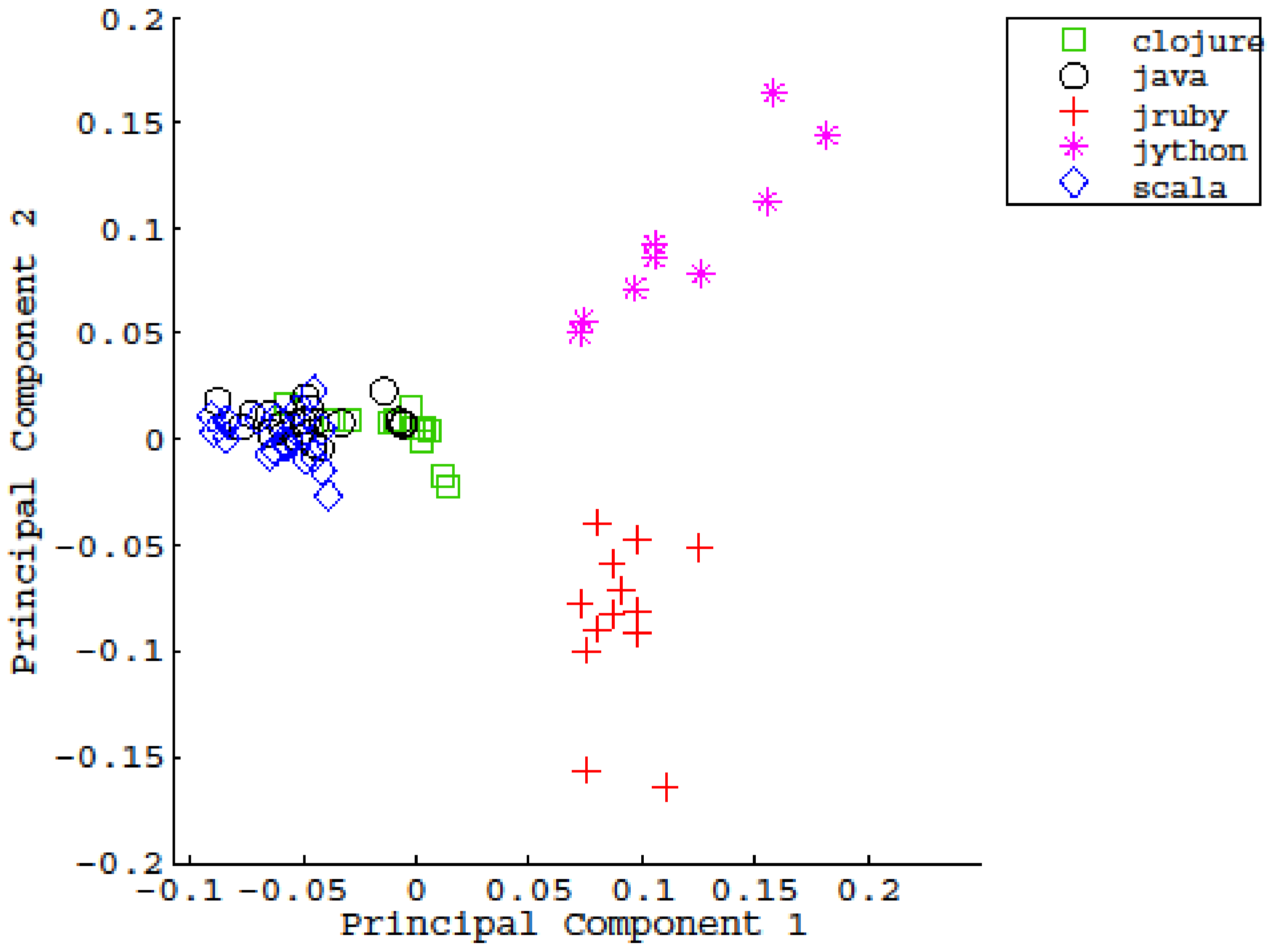
Instruction Level Results

- Principal components analysis (1-gram, filtered)



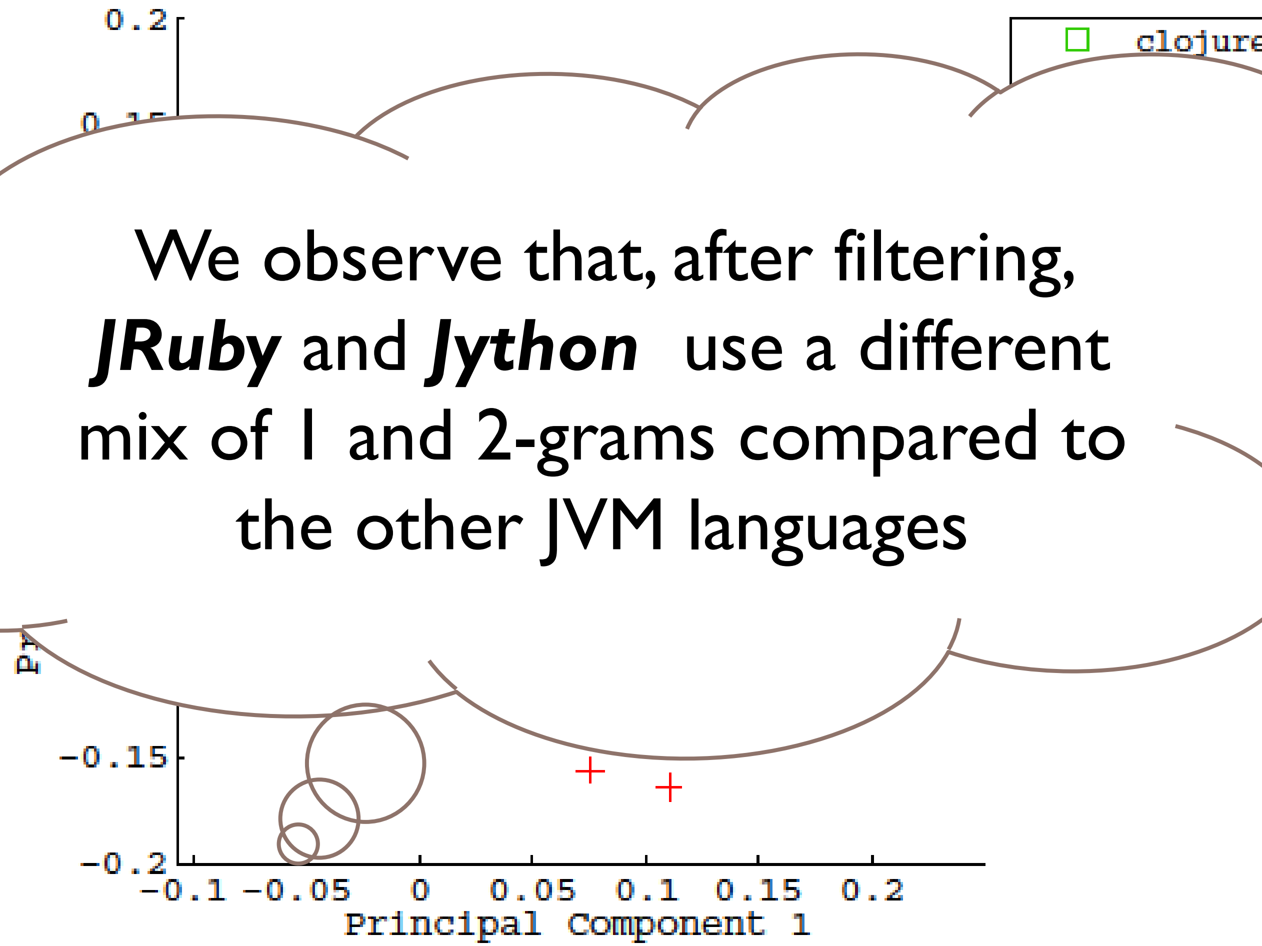
Instruction Level Results

- Principal components analysis (2-gram, filtered)



Instruction Level Results

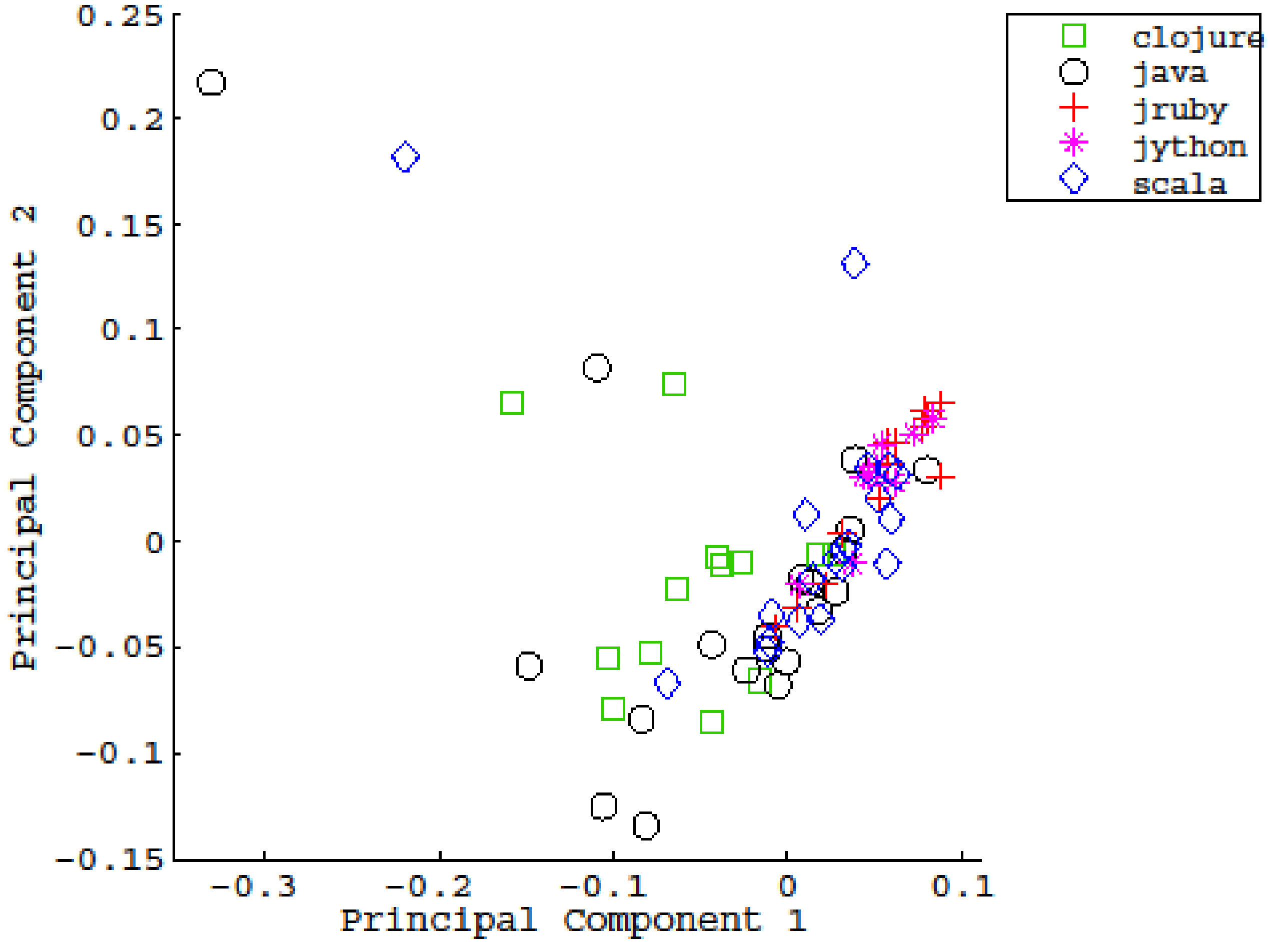
- Principal components analysis (2-gram, filtered)



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

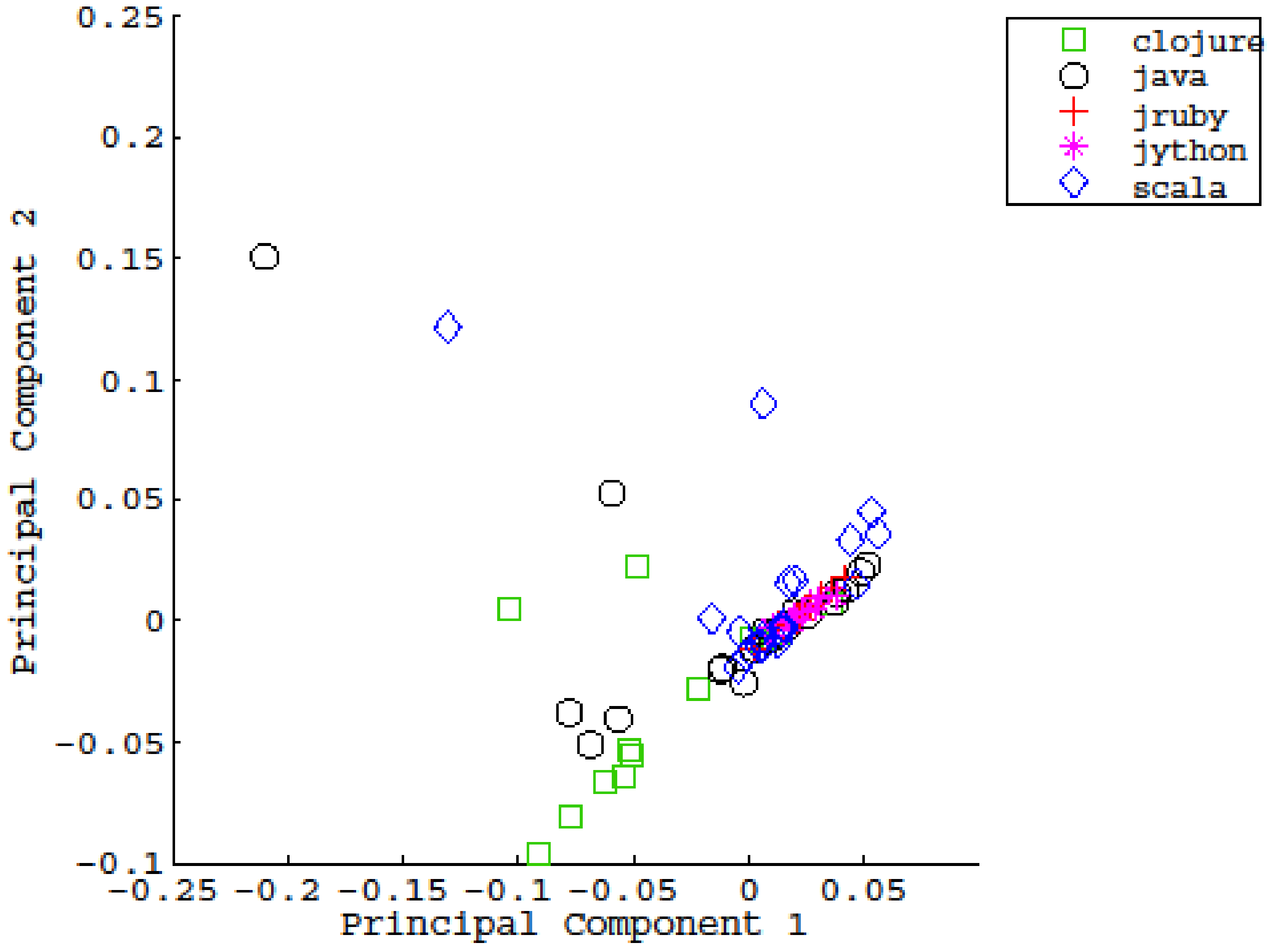
Instruction Level Results

- Principal components analysis (1-gram, unfiltered)



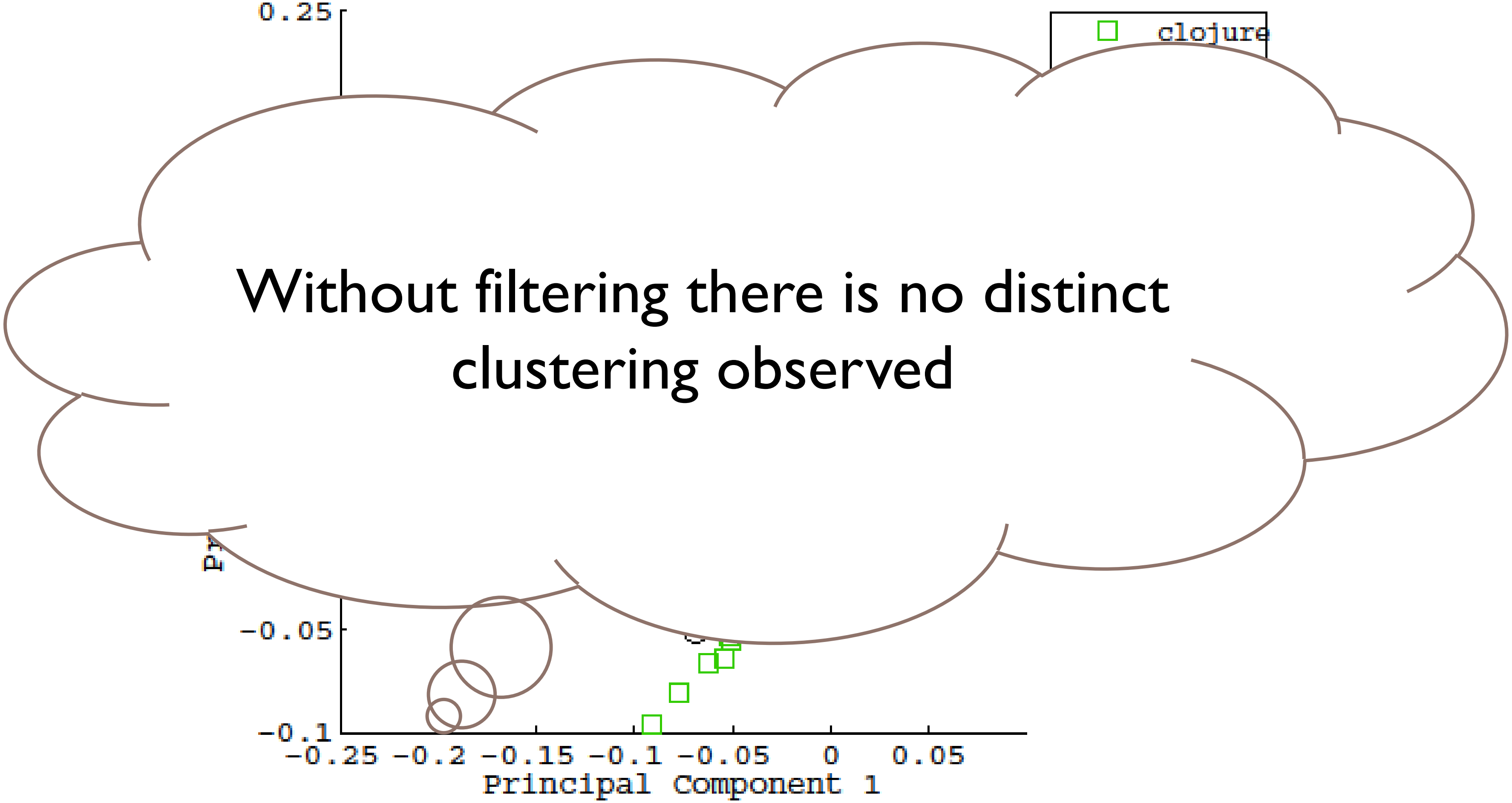
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)



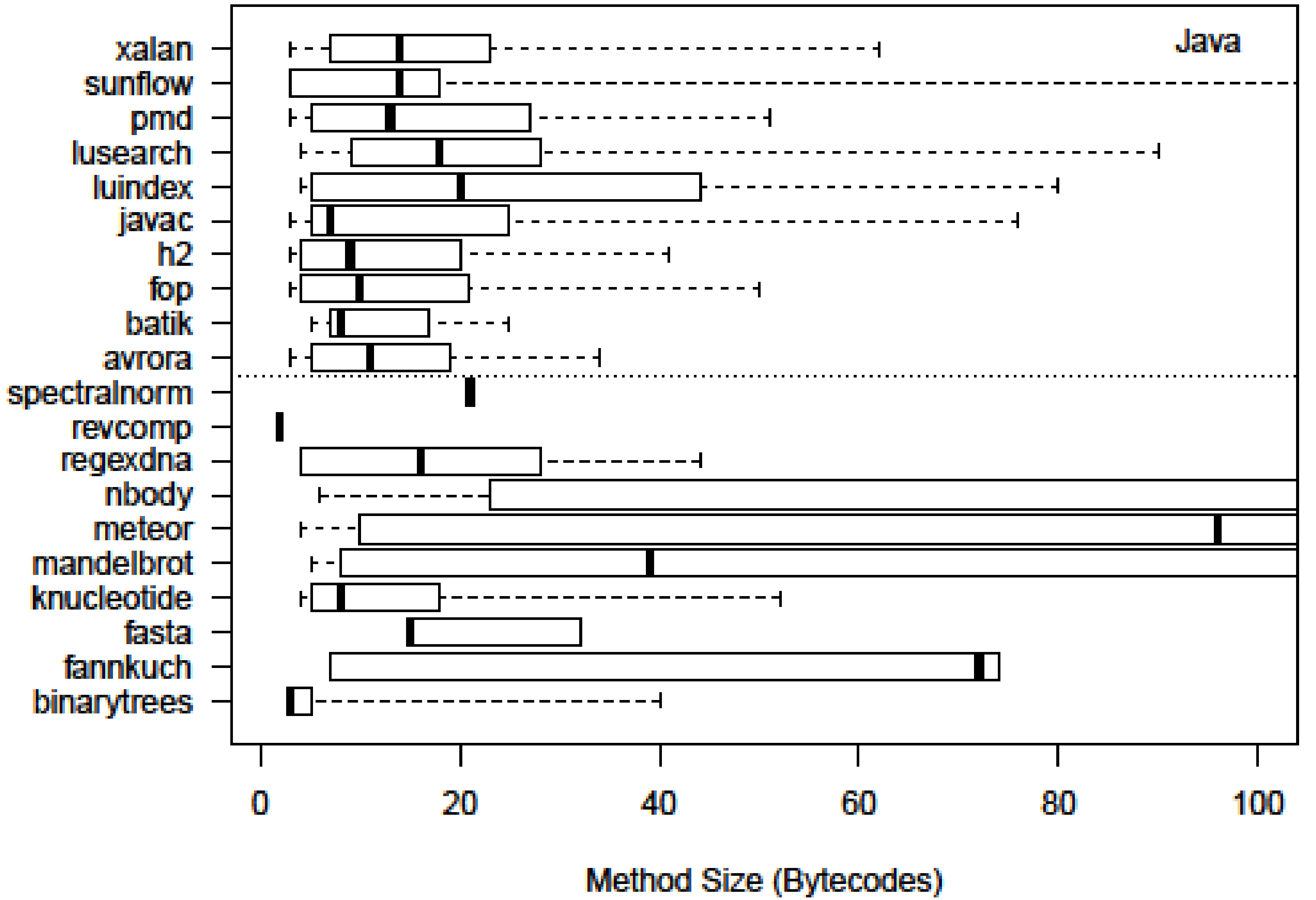
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)



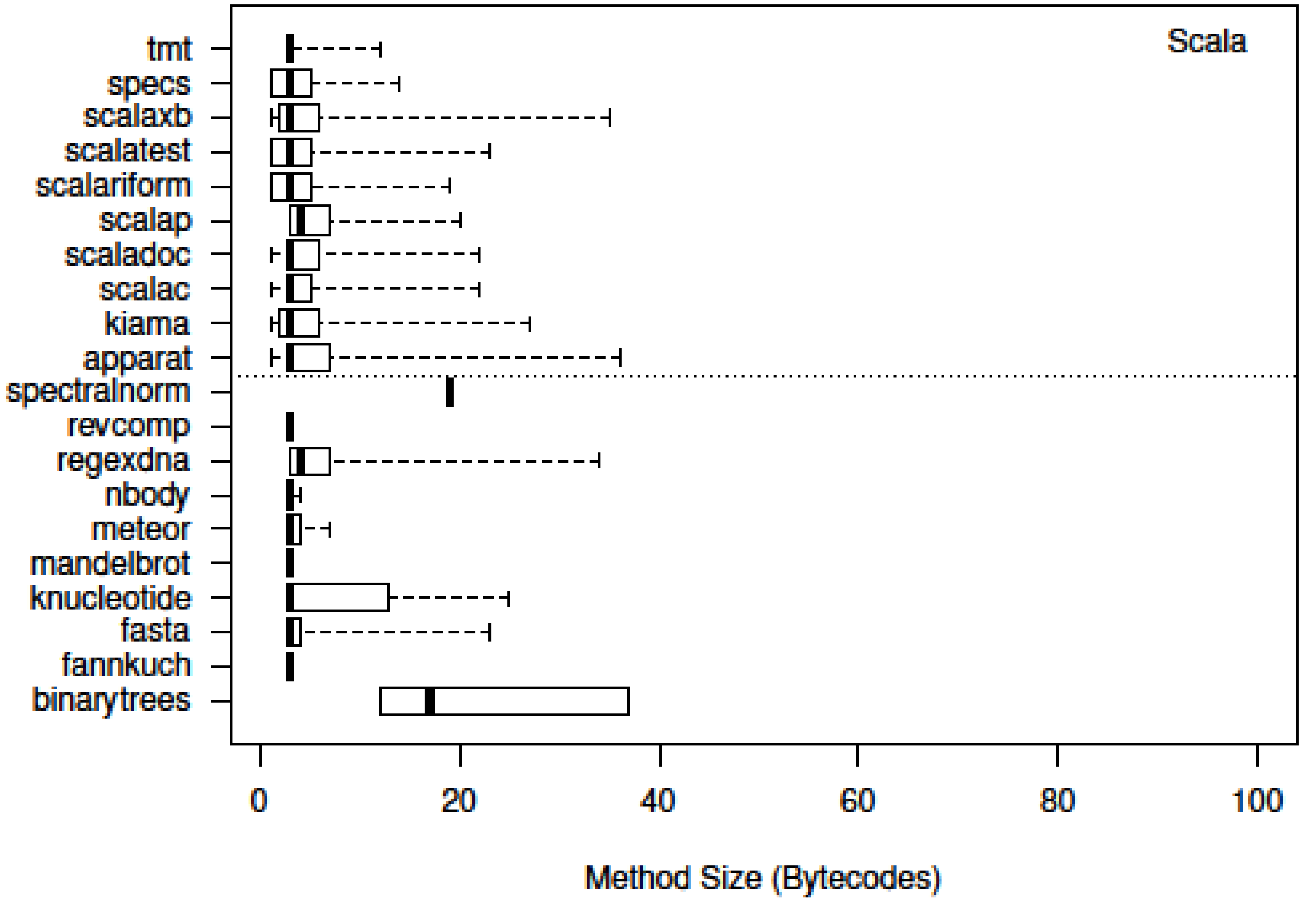
Method Level Results - Java

- Results for the distribution of method sizes



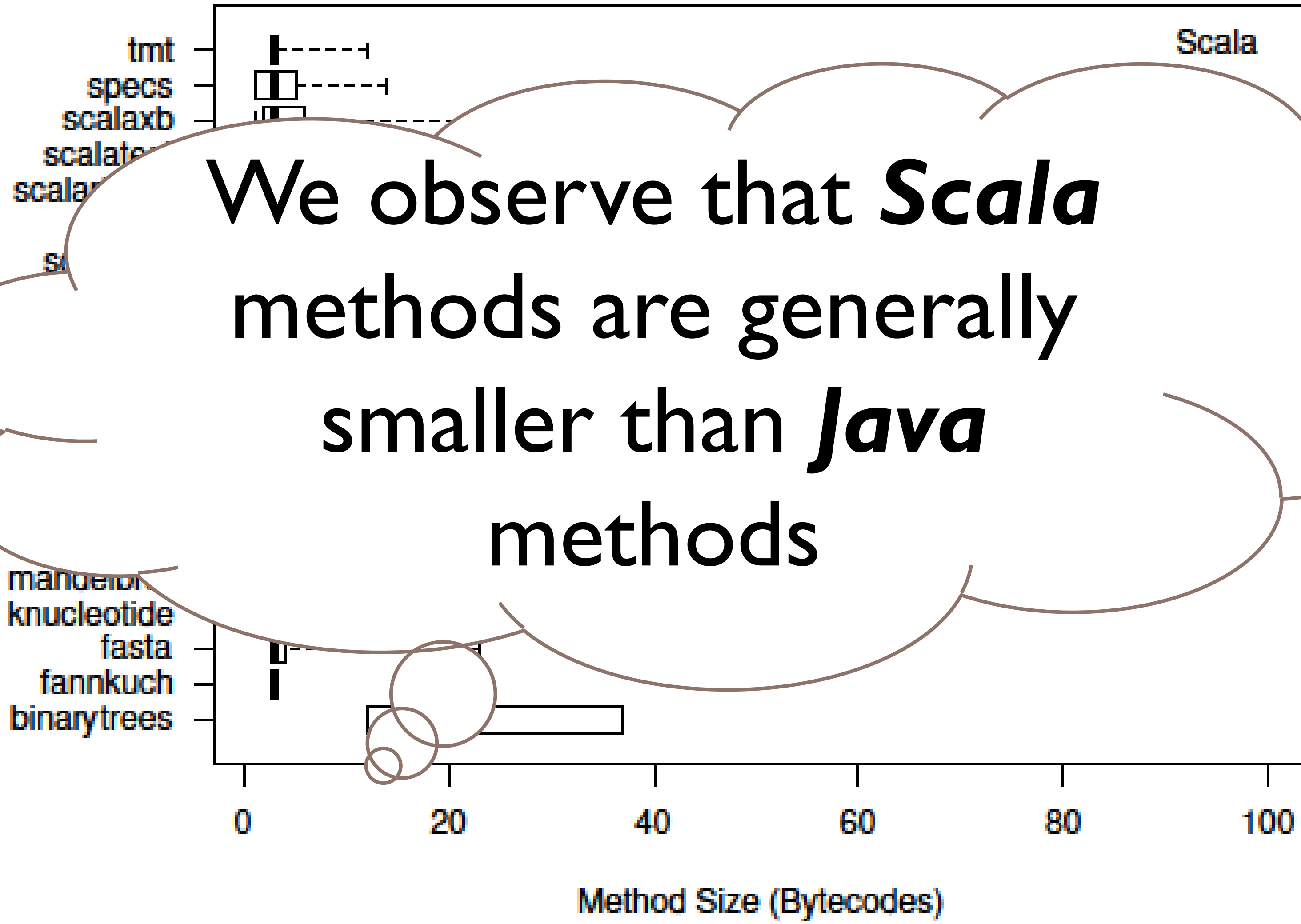
Method Level Results - Scala

- Results for the distribution of method sizes (filtered)



Method Level Results - Scala

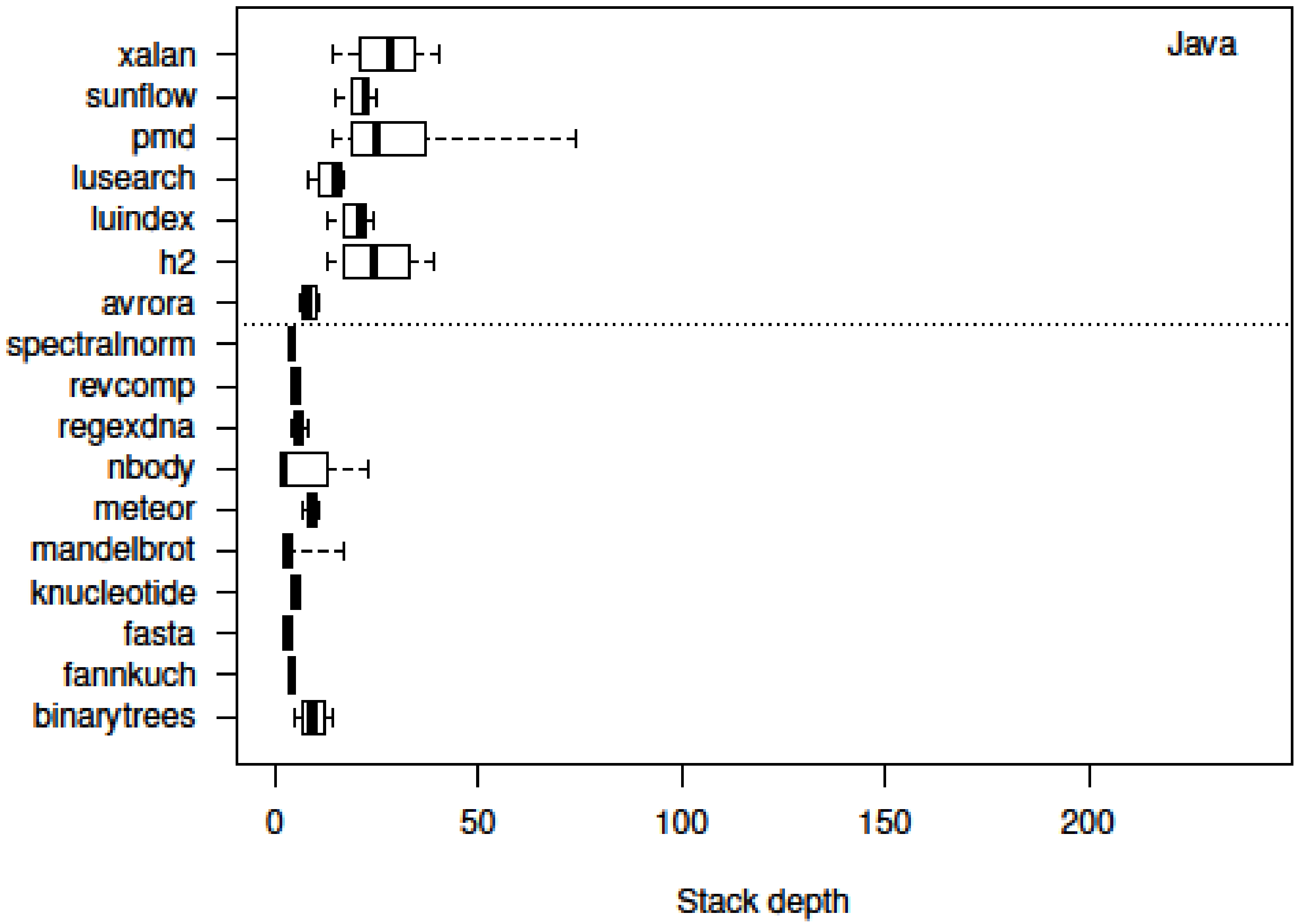
- Results for the distribution of method sizes (filtered)



We observe that ***Scala*** methods are generally smaller than ***Java*** methods

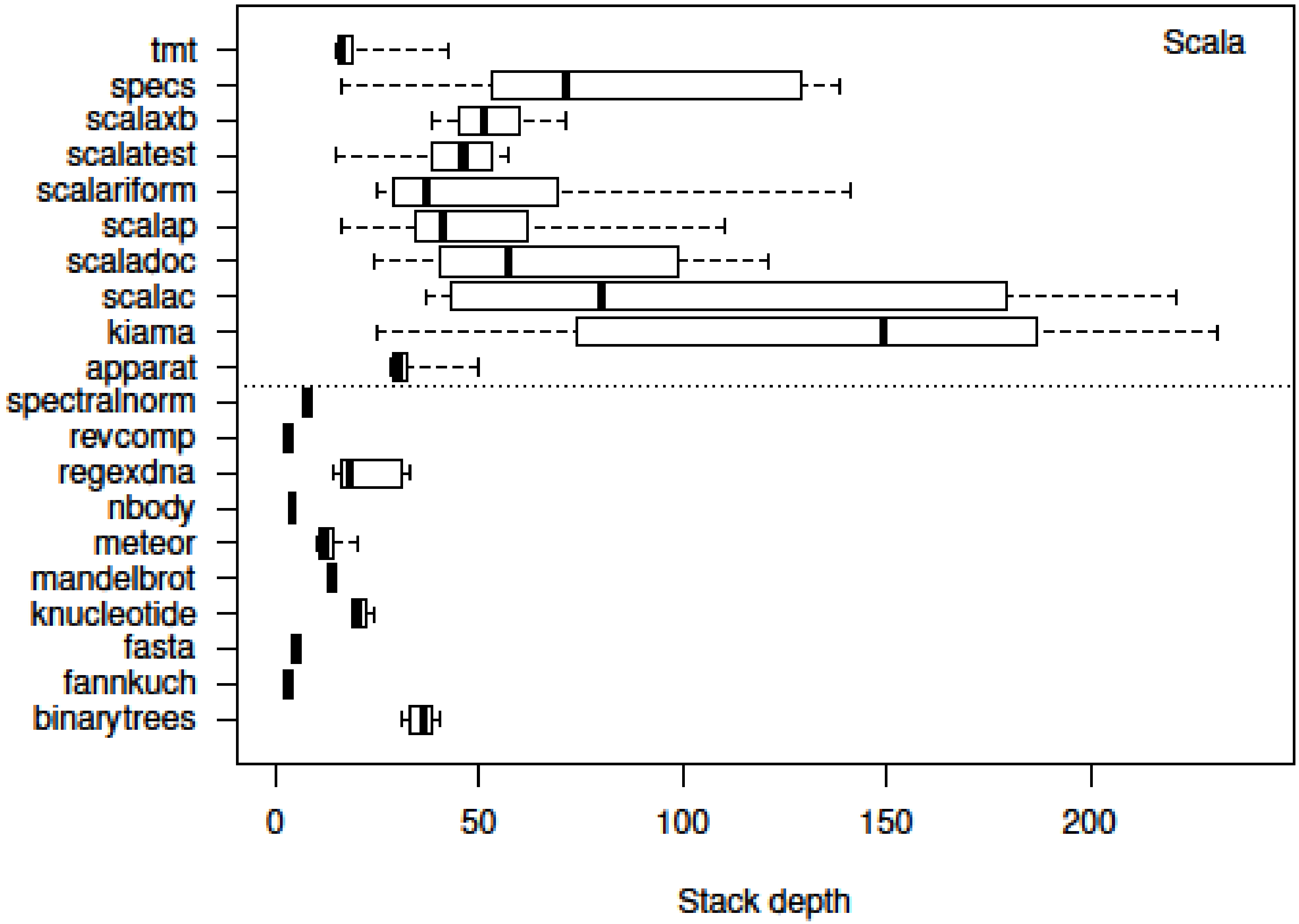
Method Level Results - Java

- Results for the distribution of method stack depths



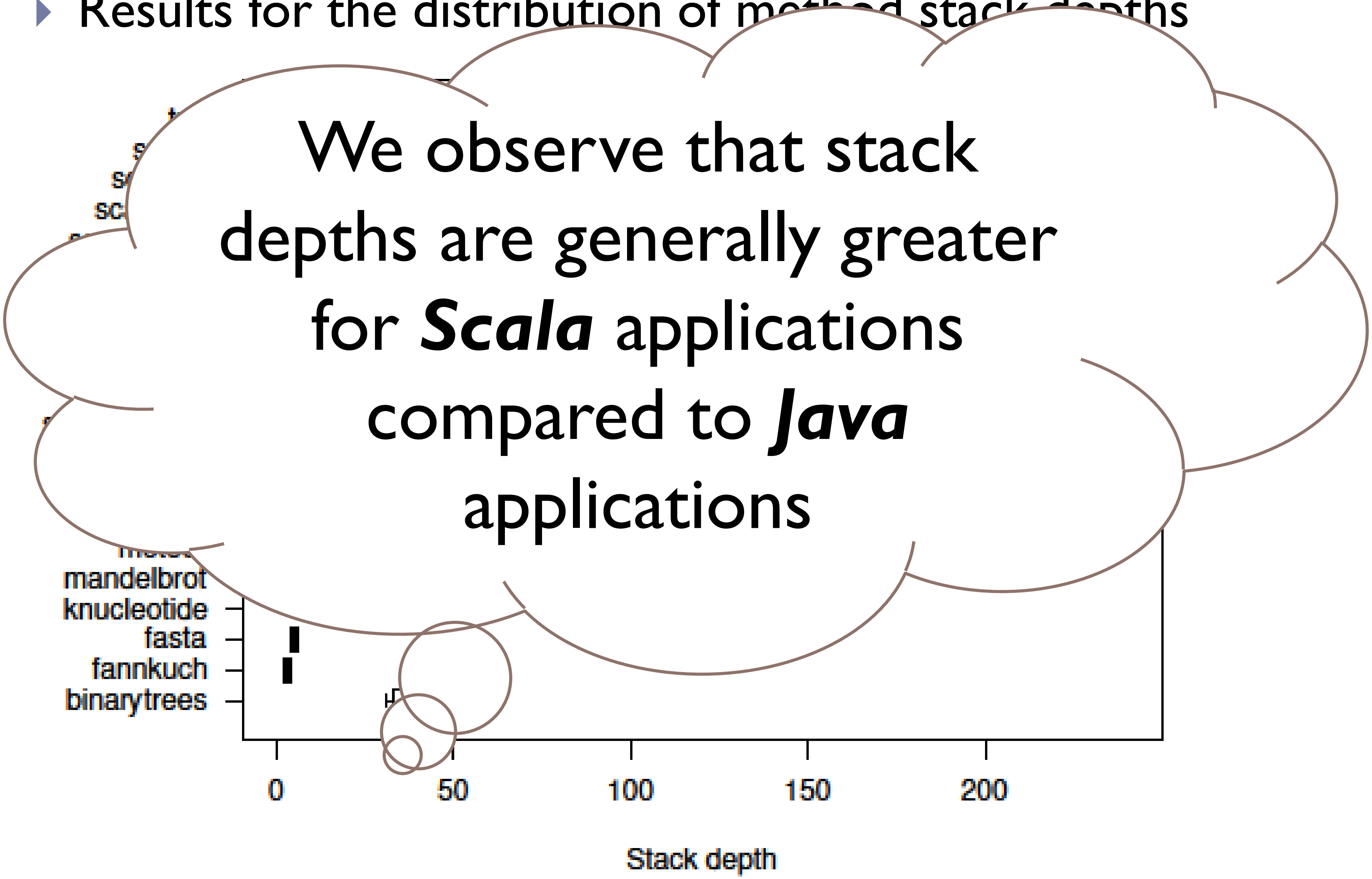
Method Level Results - Scala

- Results for the distribution of method stack depths



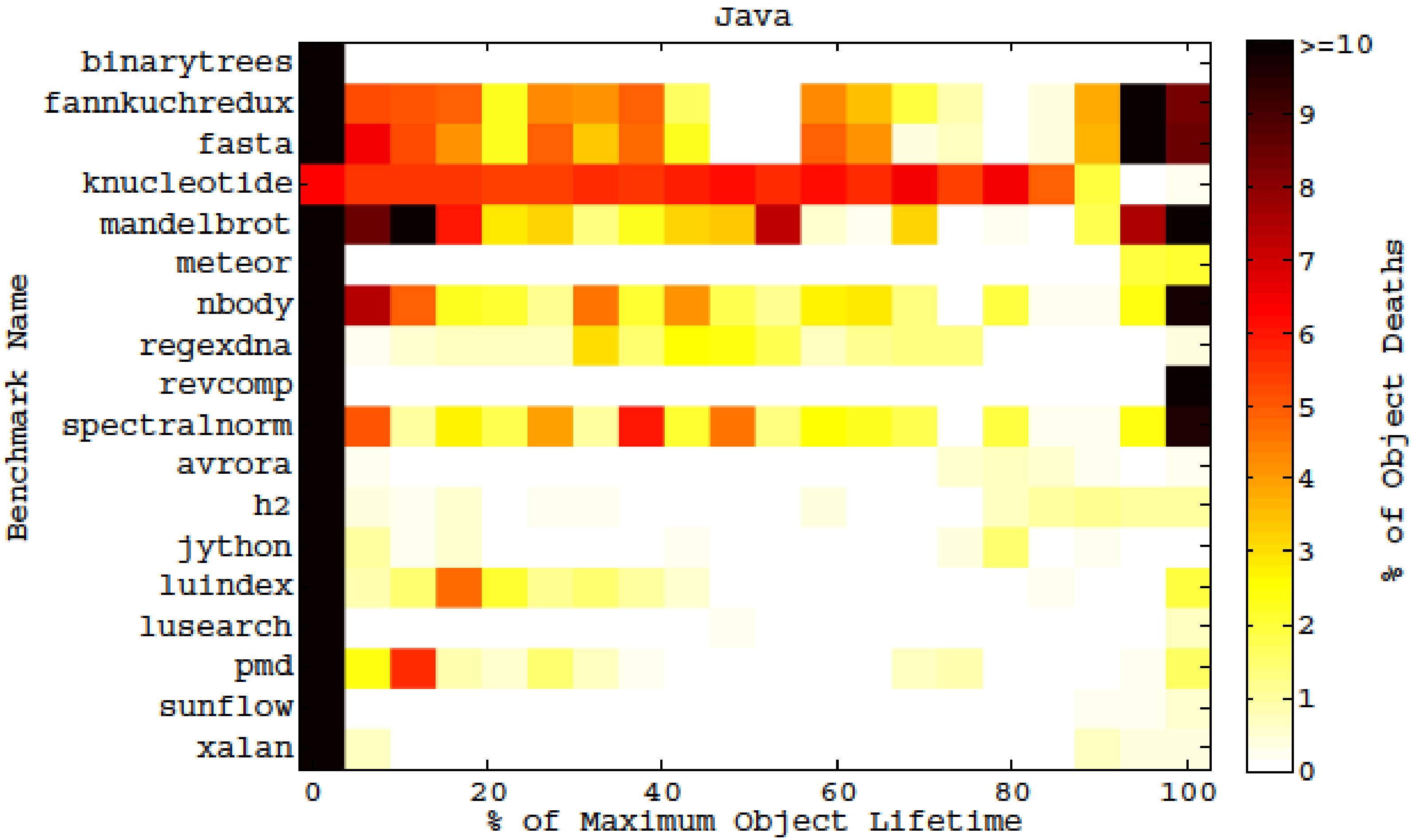
Method Level Results - Scala

- Results for the distribution of method stack depths



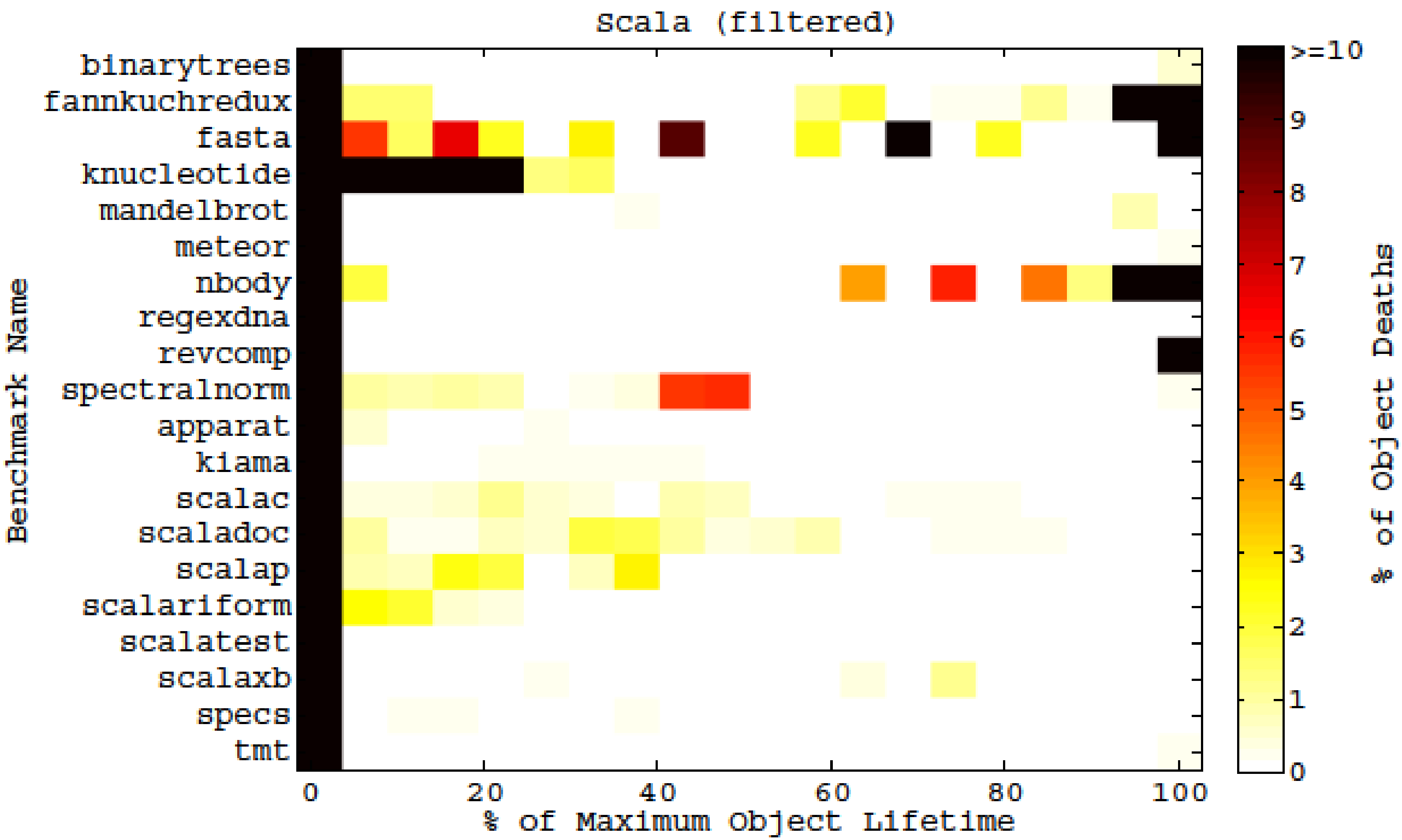
Object Level Results - Java

- Object lifetime



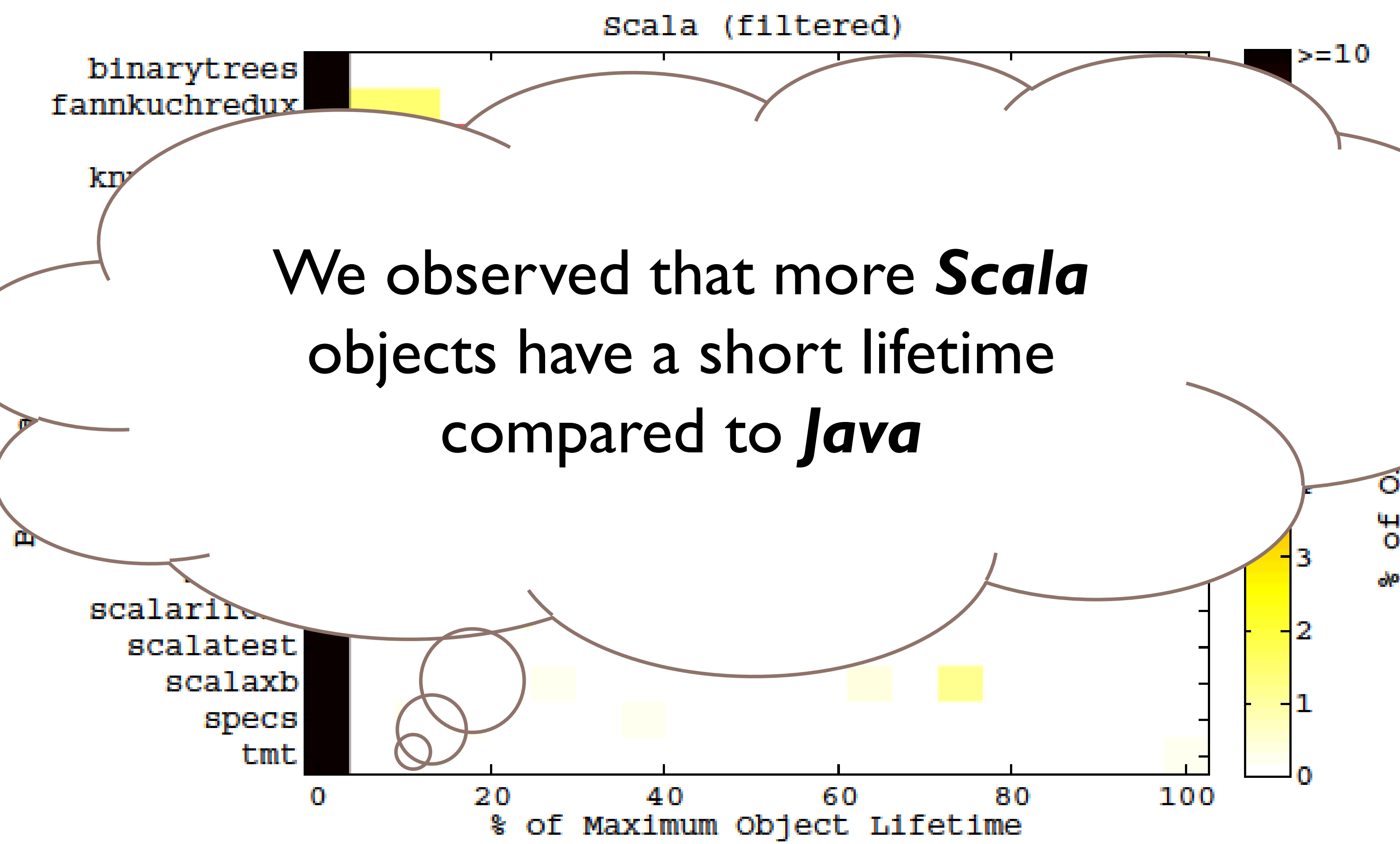
Object Level Results - Scala

- Object lifetime (filtered)



Object Level Results - Scala

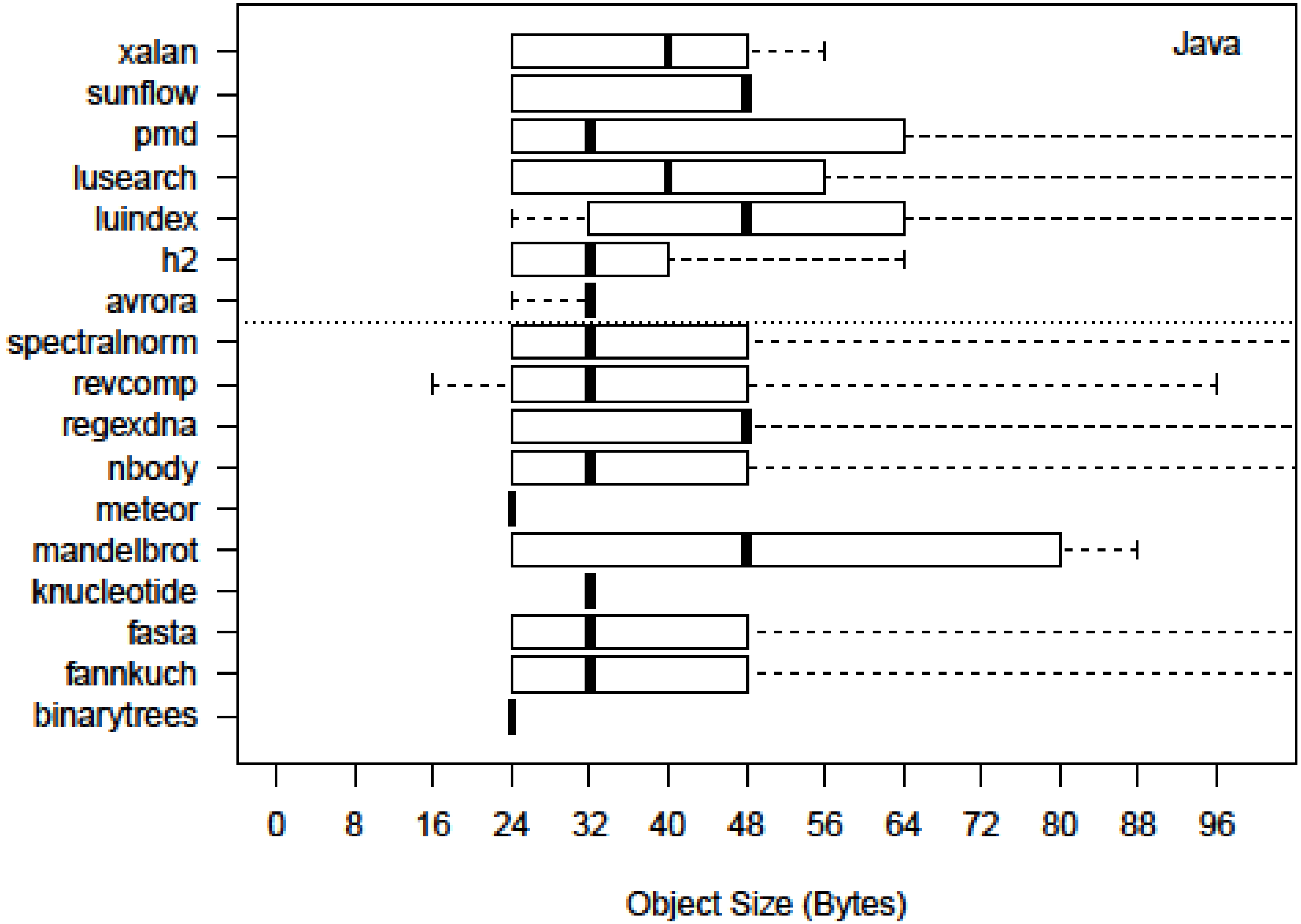
- Object lifetime (filtered)



We observed that more ***Scala*** objects have a short lifetime compared to ***Java***

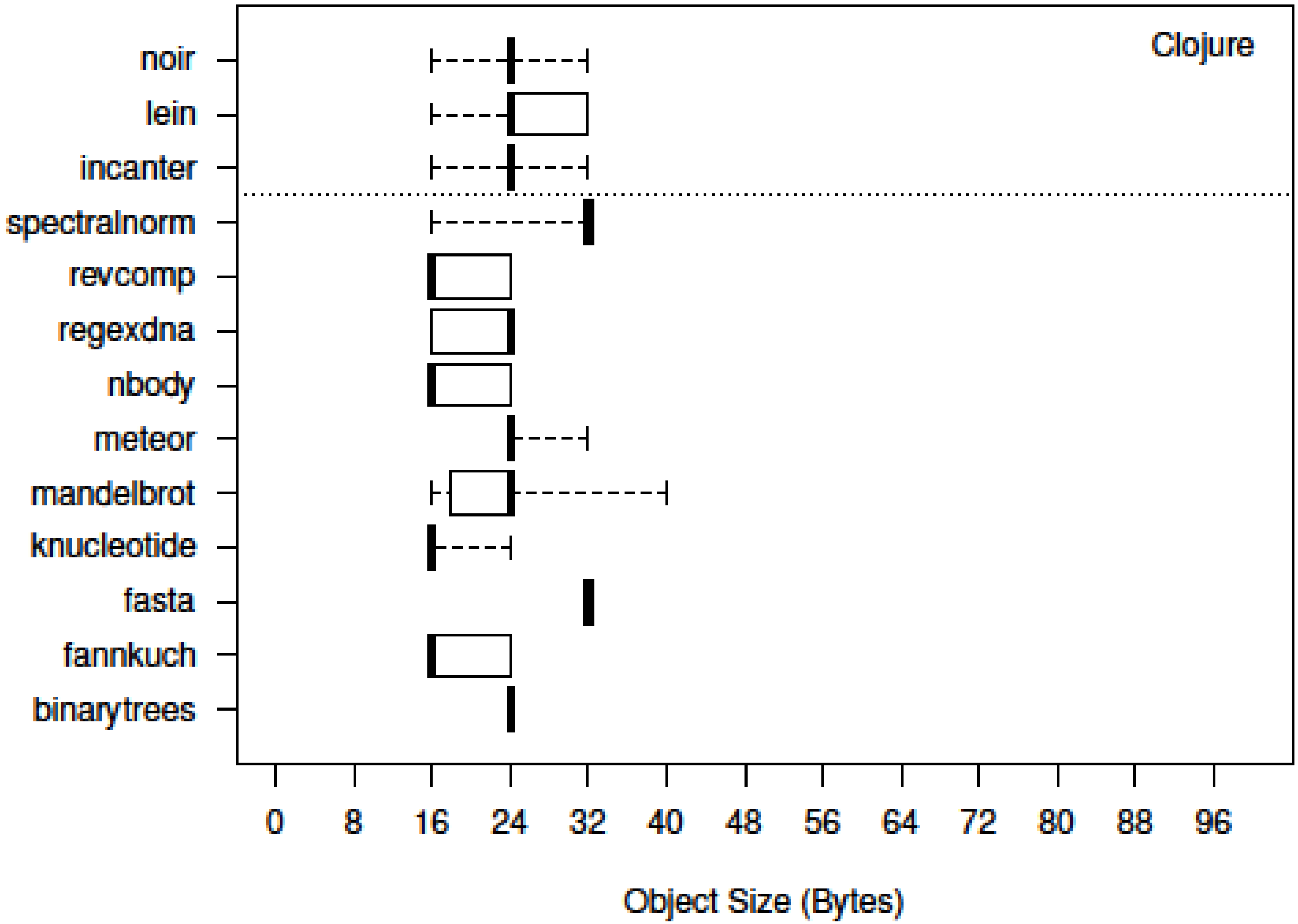
Object Sizes - Java

- Results for the distribution of object sizes (filtered)



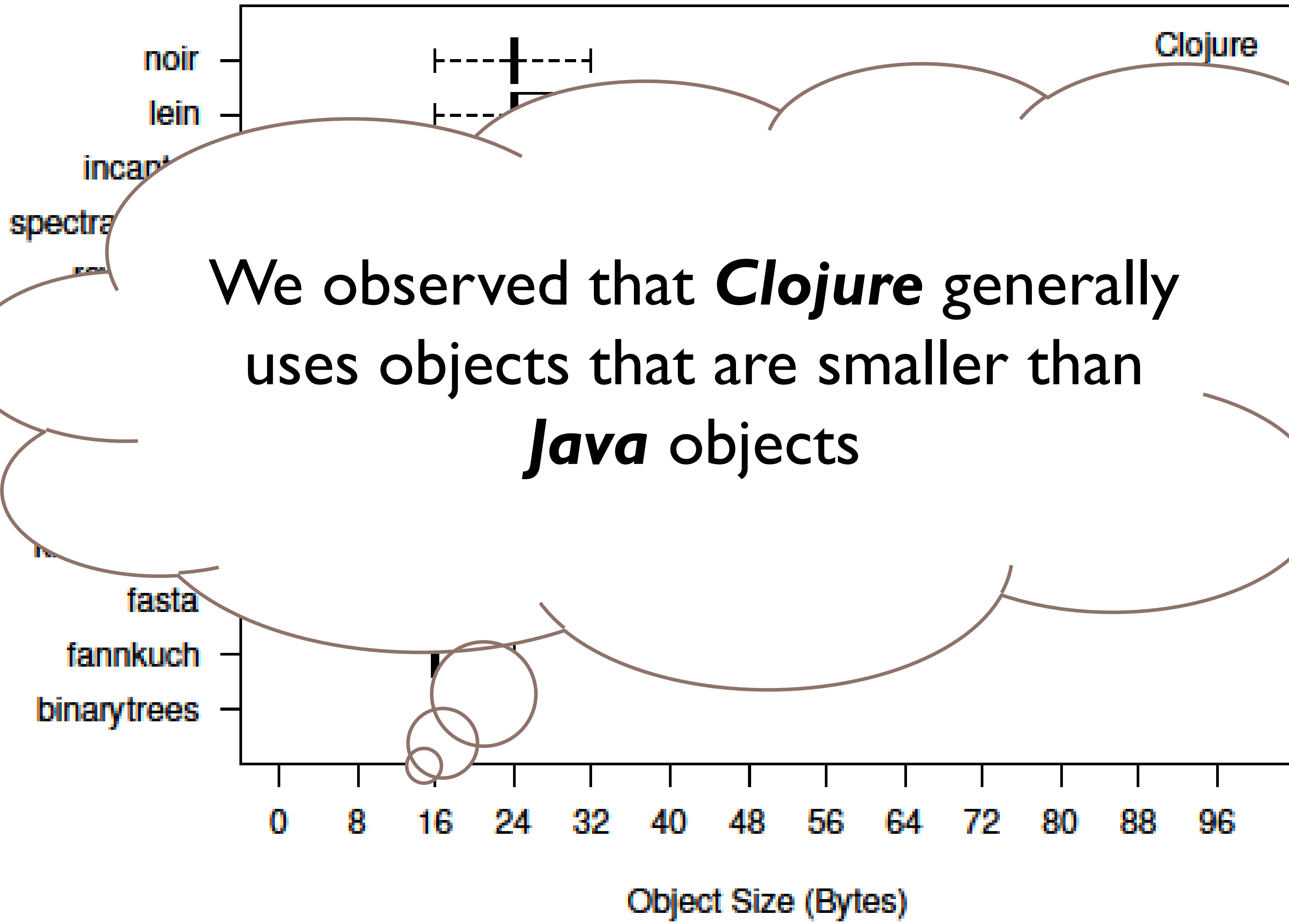
Object Sizes - Clojure

- Results for the distribution of object sizes (filtered)



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

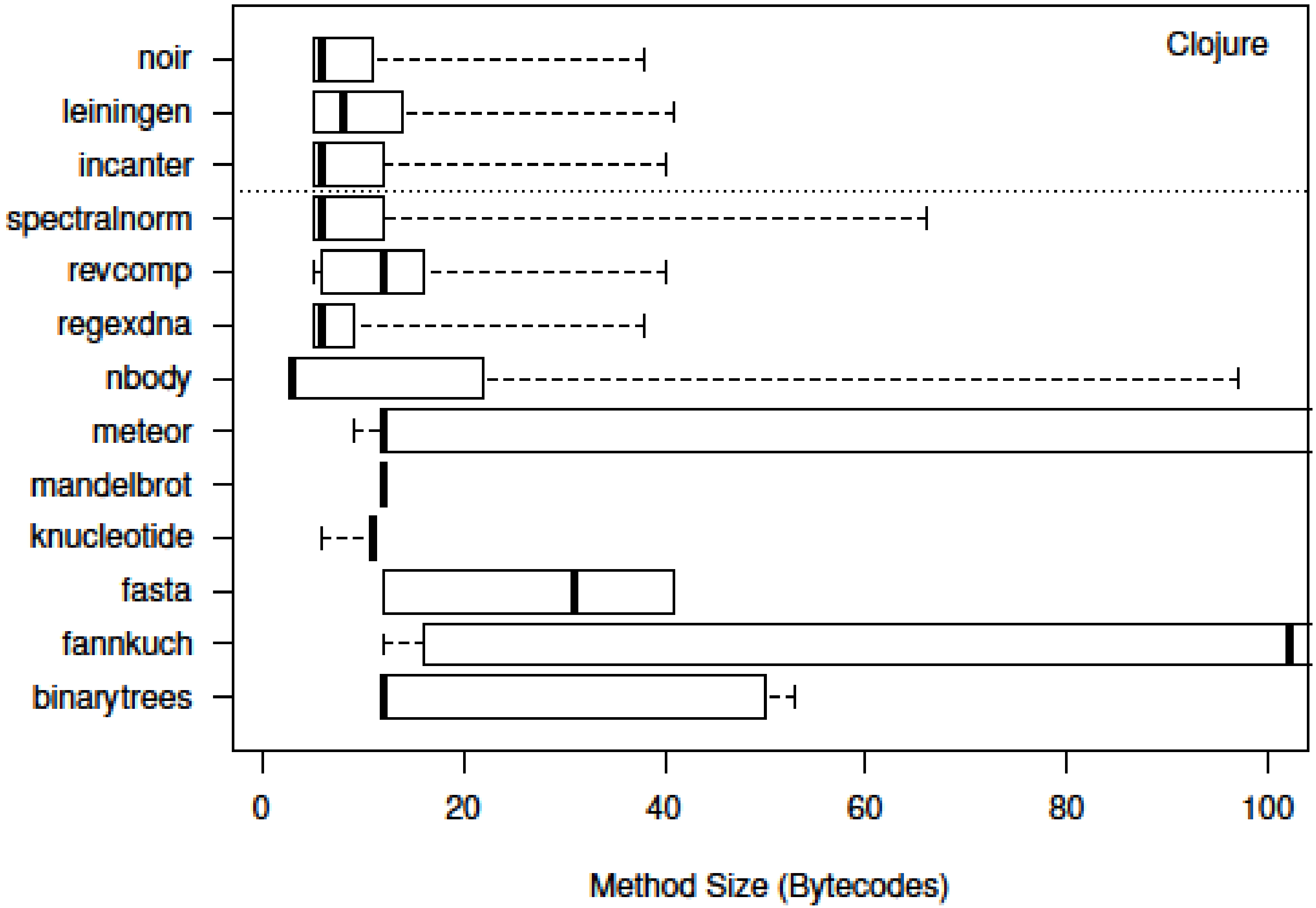
- Aim of study is to investigate the reasons for the poor performance of JVM languages
- Benchmarks in 5 JVM languages were profiled
- JVM languages do have distinctive characteristics related to their features
- Next step is to optimise performance using the observed characteristics

Our research paper, experimental scripts and results are available at: http://bit.ly/19JsrKf

Questions?

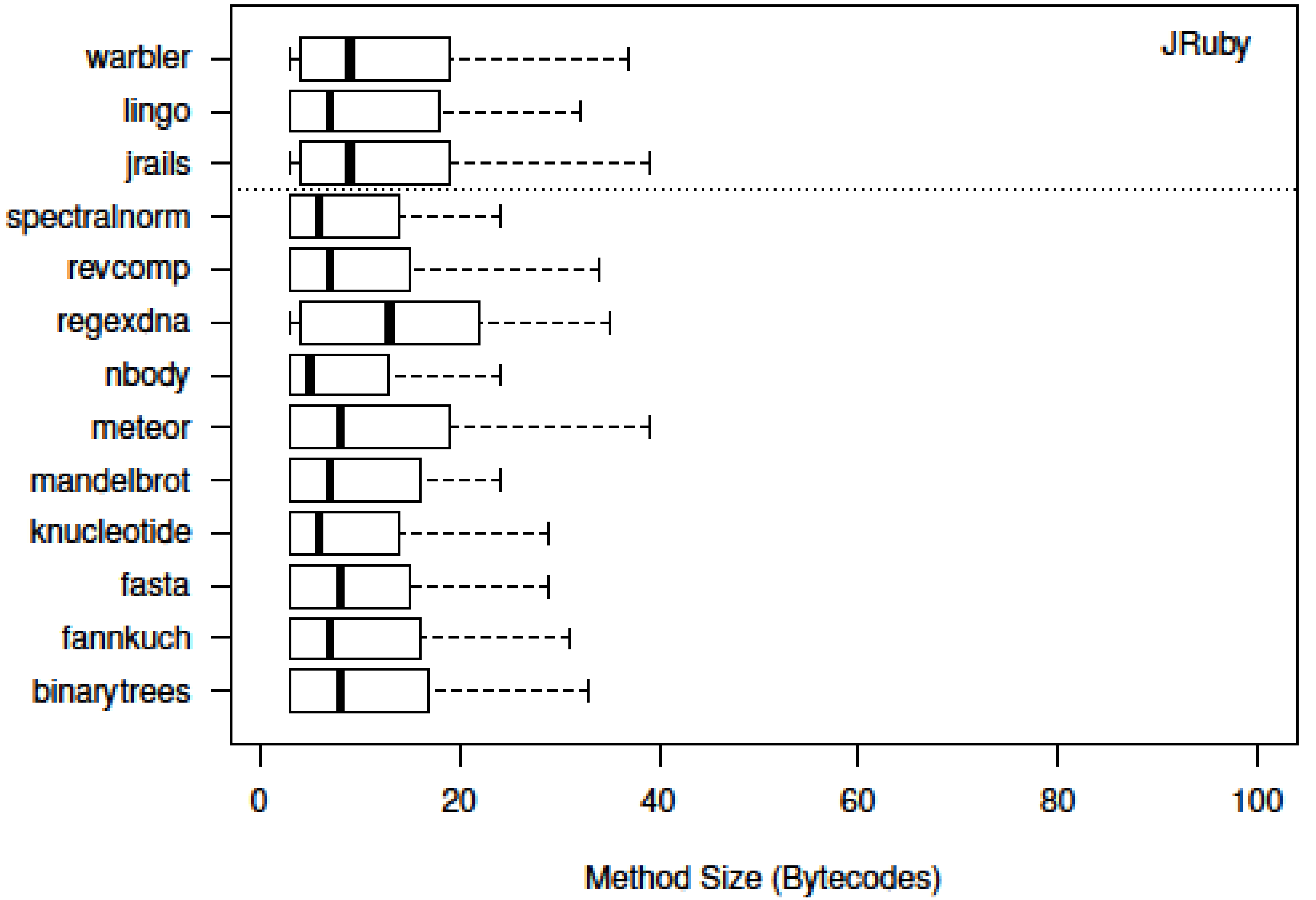
More Method Size Graphs - Clojure

- Results for the distribution of method sizes (filtered)

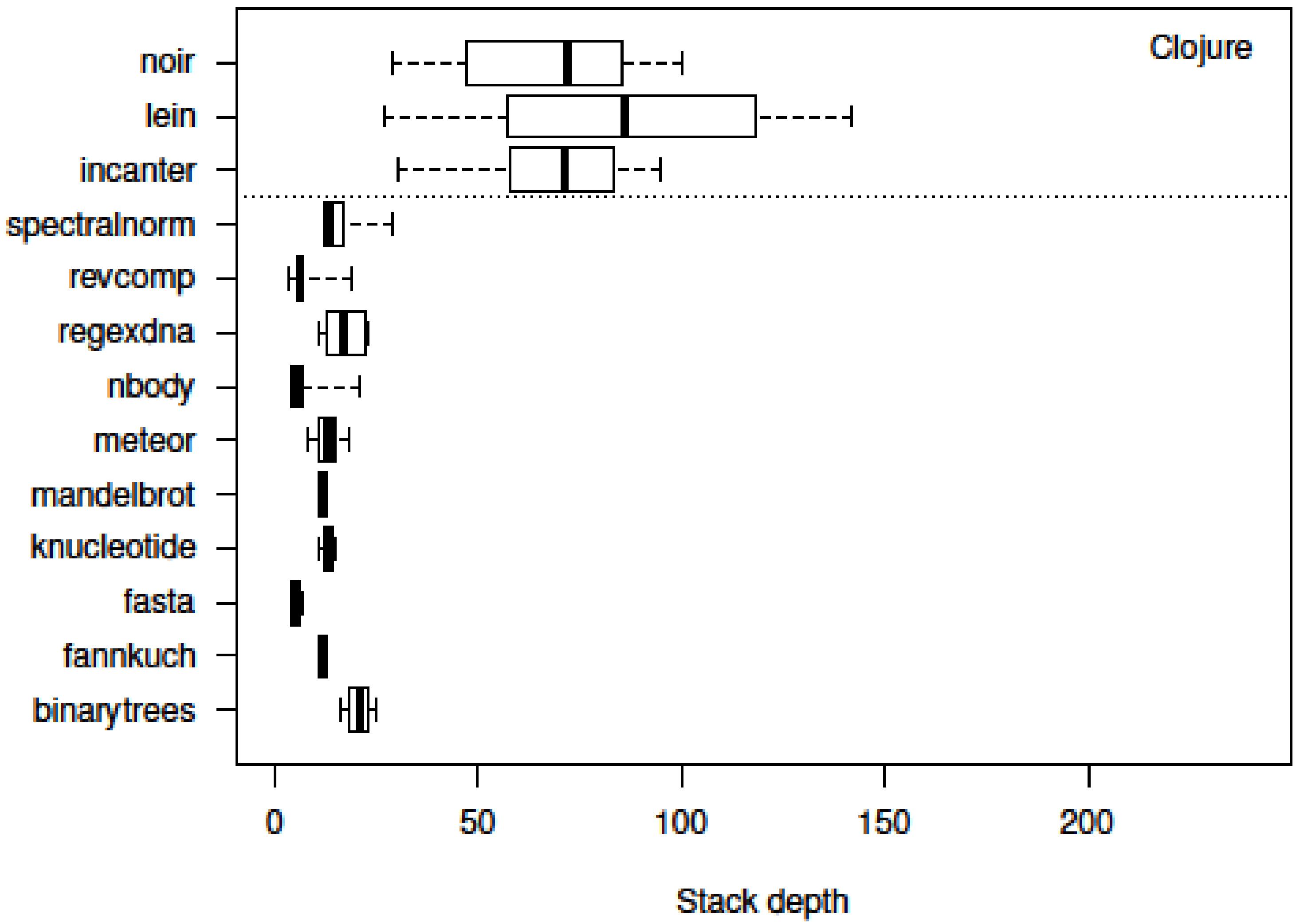


More Method Size Graphs - JRuby

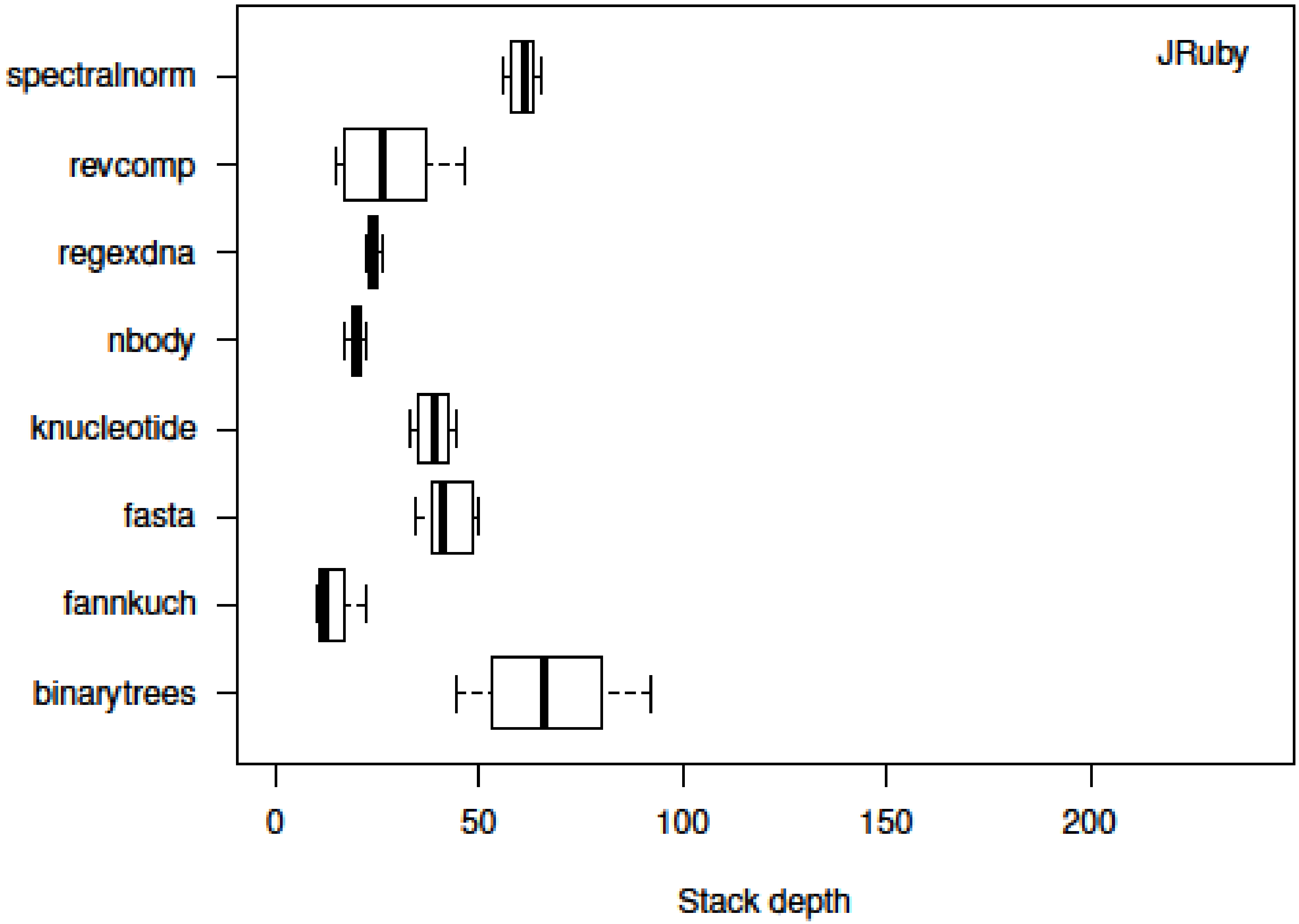
- Results for the distribution of method sizes (unfiltered)



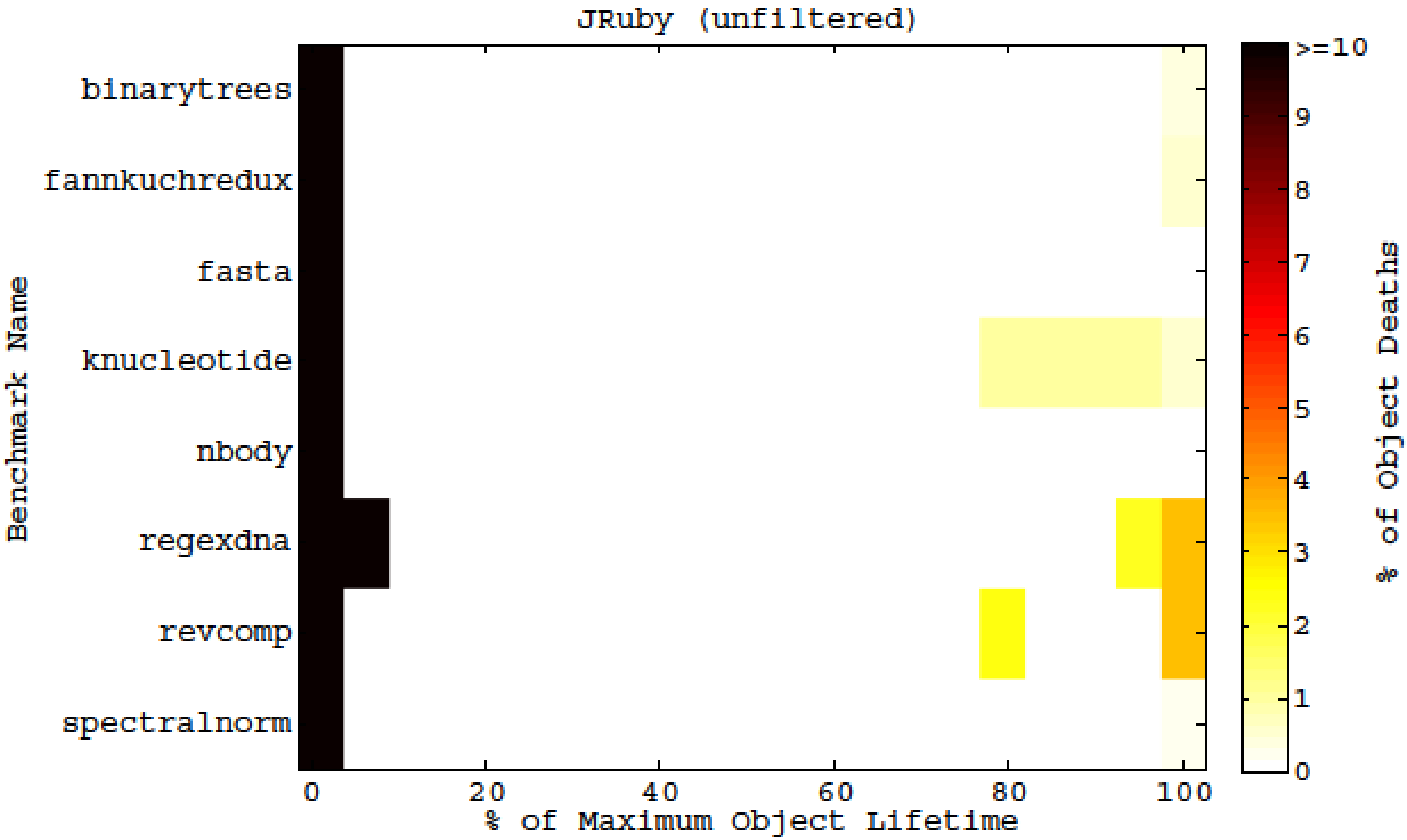
More Method Stack Depth Results - Clojure



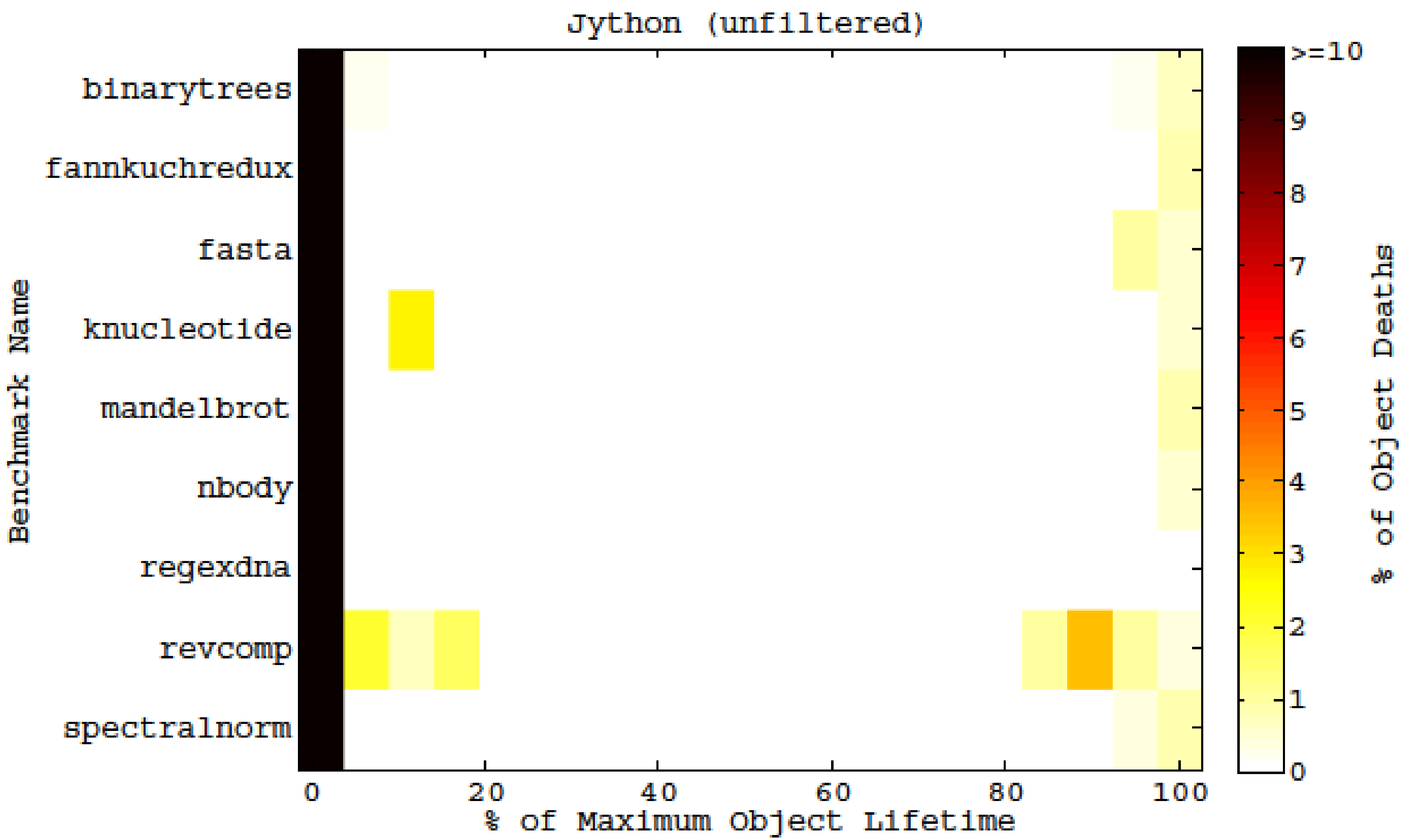
More Method Stack Depth Results - JRuby



More Object Lifetime Graphs - JRuby

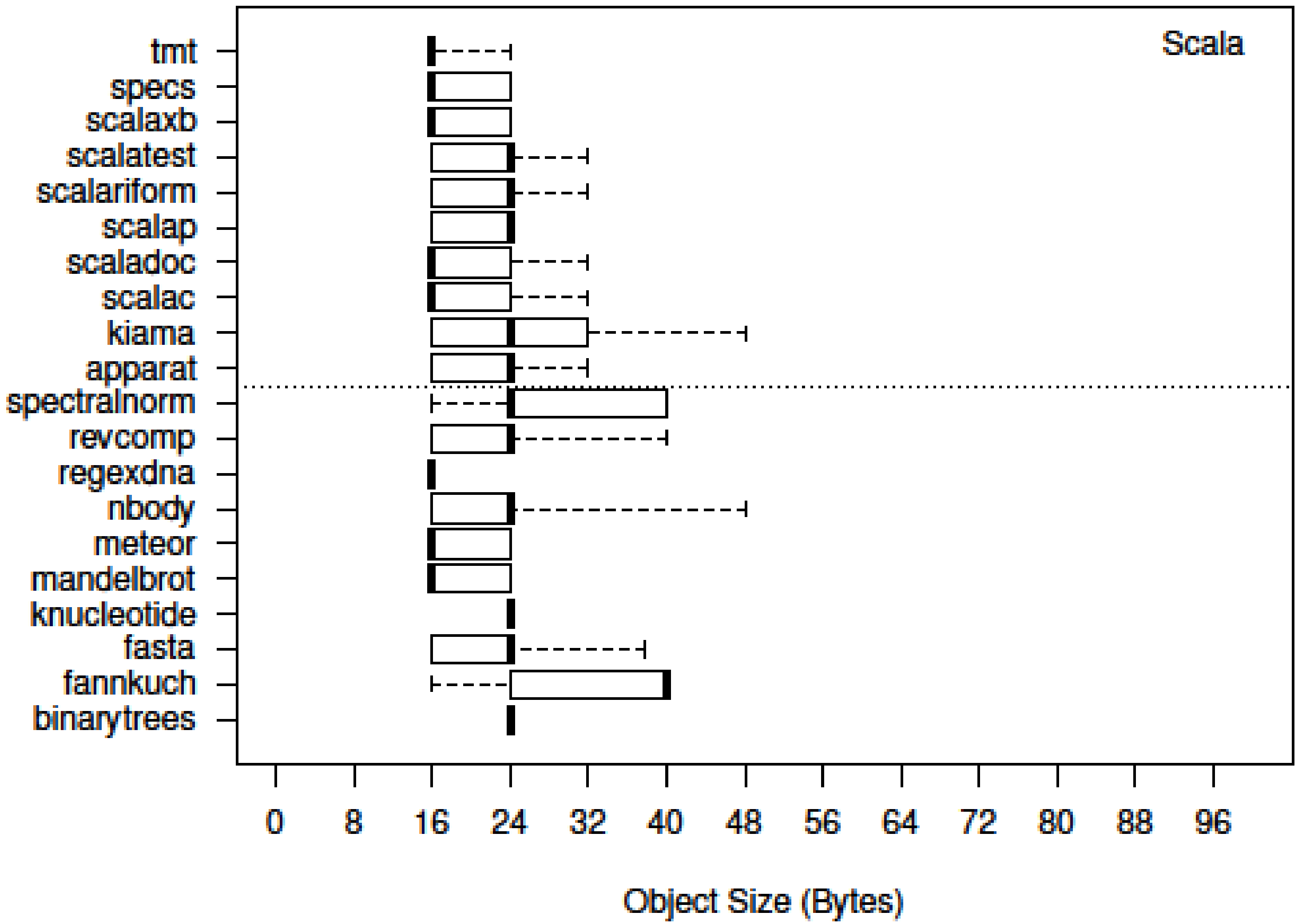


More Object Lifetime Graphs - Jython



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)