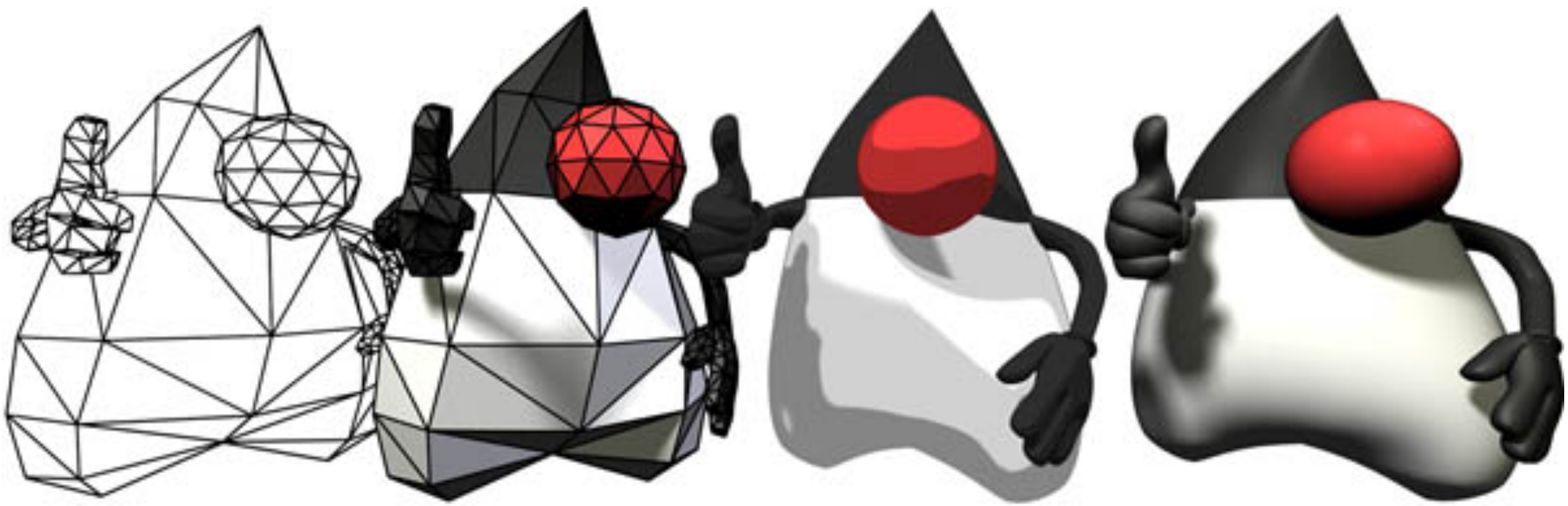
Other Languages on the JVM

Jeremy Singer and Wing Hang Li

University of Glasgow | School of Computing Science

A brief history of **Java**



Why is Java **boring**?

Why is Java **boring?**

1. **it's verbose**
2. **it's uncool**
3. **it's non-functional**

1. it's verbose

```
; Clojure

  (println "hello")
```

1. it's verbose

```
// Java

System.out.println("hello");
```

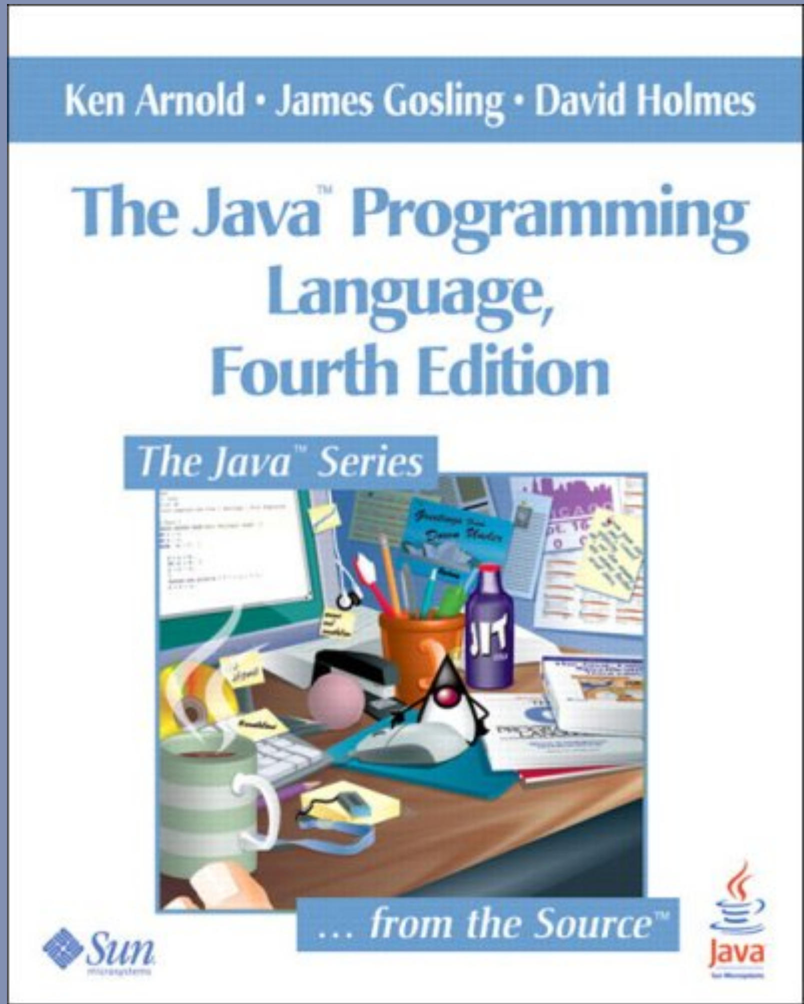
1. it's verbose

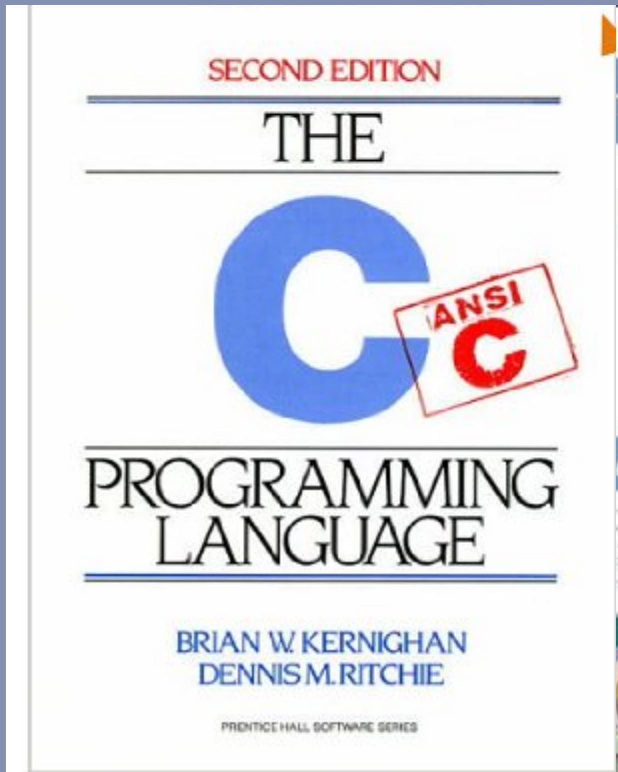
```
// Java

public static void main(String[] args) {
     System.out.println("hello");
}
```

1. it's verbose

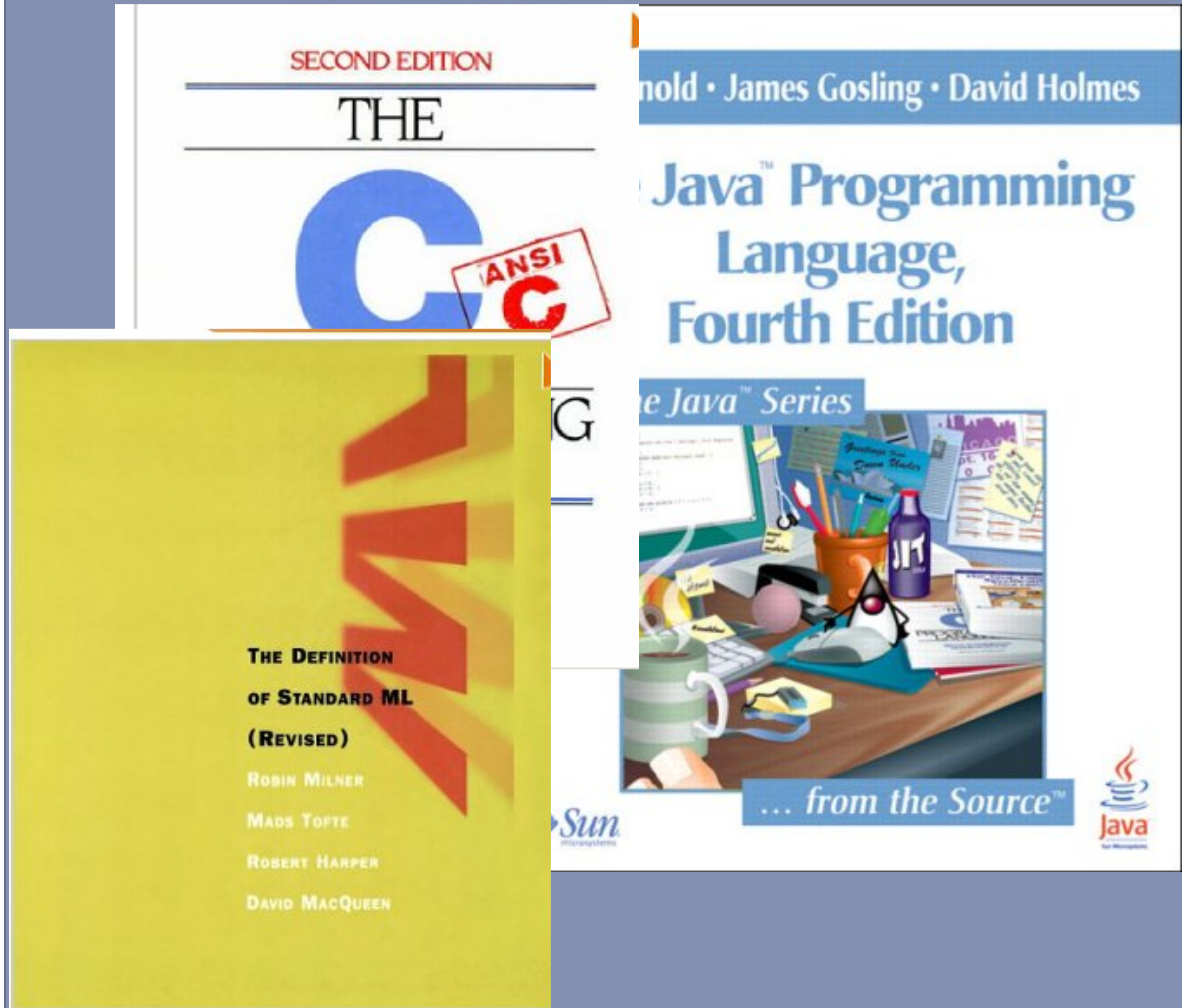
```
  // Java
public class Hello {
public static void main(String[] args) {
    System.out.println("hello");
}
}
```

1. it's verbose


Ken Arnold · James Gosling · David Holmes
The Java™ Programming Language, Fourth Edition
The Java™ Series
... from the Source™
Sun microsystems
Java


SECOND EDITION
THE
C
ANSI C
PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
PRENTICE HALL SOFTWARE SERIES

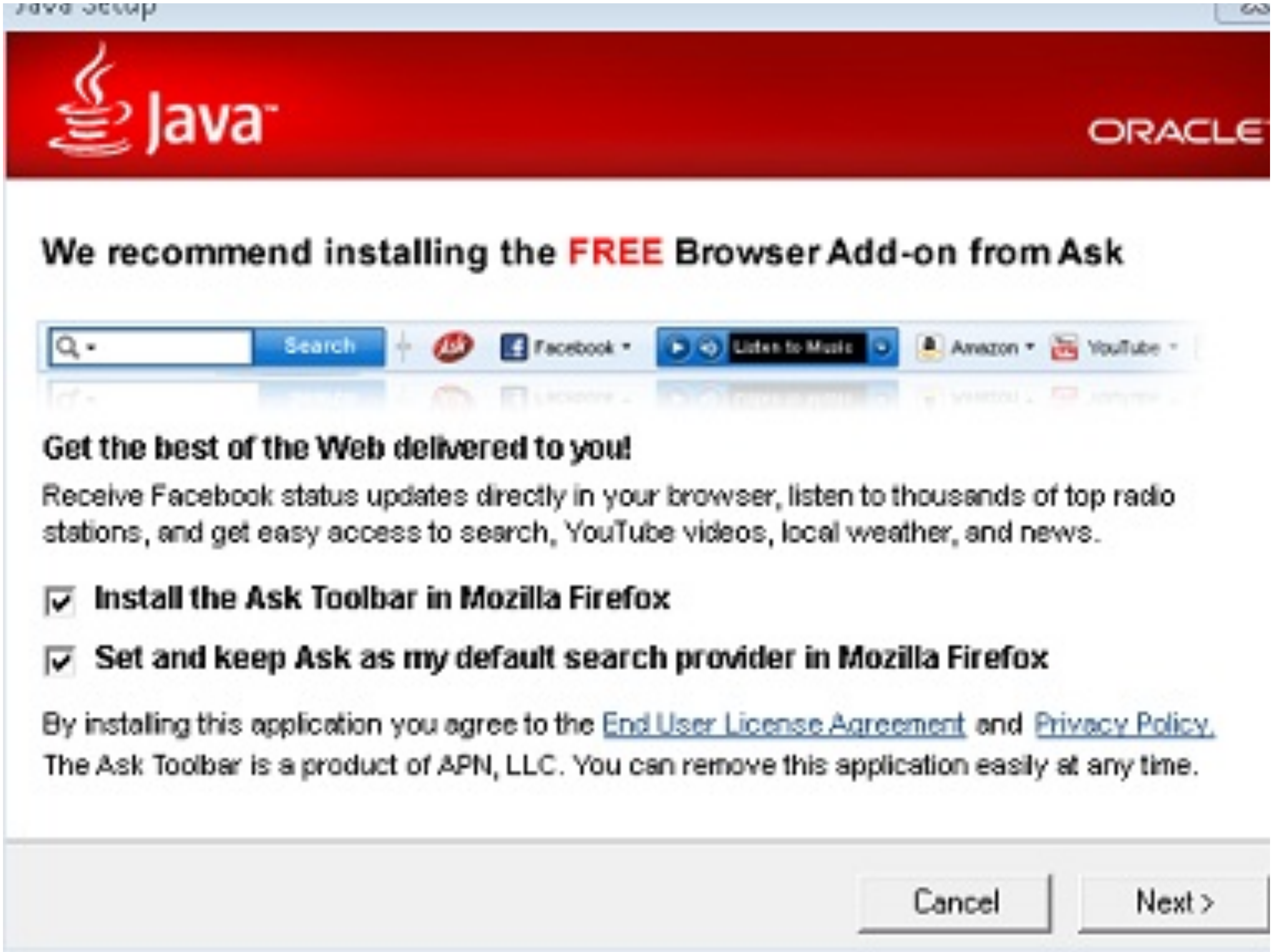

James Gosling · David Holmes
Java™ Programming Language, Fourth Edition
Java™ Series
... from the Source™
Sun
Java


SECOND EDITION
THE
C
ANSI C
G
nold · James Gosling · David Holmes
Java™ Programming
Language,
Fourth Edition
e Java™ Series
... from the Source™
Java
Sun
THE DEFINITION
OF STANDARD ML
(REVISED)
ROBIN MILNER
MADS TOFTE
ROBERT HARPER
DAVID MACQUEEN


SECOND EDITION
THE
C
ANSI C
James Gosling · David Holmes
Java™ Programming
Java™ Series
... from
Sun
THE DEFINITION
OF STANDARD ML
(REVISED)
ROBIN MILNER
MADS TOFTE
ROBERT HARPER
DAVID MACQUEEN
C++11
THE
C++
PROGRAMMING LANGUAGE
FOURTH EDITION
BJARNE STROUSTRUP
THE CREATOR OF C++

Why is Java **boring?**

2. it's uncool


Java Setup
Java
ORACLE
We recommend installing the FREE Browser Add-on from Ask
Search
Facebook
Listen to Music
Amazon
YouTube
Get the best of the Web delivered to you!
Receive Facebook status updates directly in your browser, listen to thousands of top radio stations, and get easy access to search, YouTube videos, local weather, and news.
Install the Ask Toolbar in Mozilla Firefox
Set and keep Ask as my default search provider in Mozilla Firefox
By installing this application you agree to the End User License Agreement and Privacy Policy.
The Ask Toolbar is a product of APN, LLC. You can remove this application easily at any time.
Cancel
Next >

Why is Java **boring?**

3. it's non-functional

Context

This presentation is based on our recent research paper:

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

Available at: http://bit.ly/19JsrKf

Background – the JVM

- One reason for Java's success is the Java Virtual Machine
- The JVM provides:
  - "Write once, run anywhere" capability (**WORA**)
  - Security
  - 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 and JRuby are dynamically typed
  - JRuby and Jython are scripting languages
  - Clojure is 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
- JRuby
- Jython
- Scala

What's the Catch?

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

Fortran Intel	Java	Scala	Clojure	JRuby
1.01	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
- The new `INVOKEDYNAMIC` instruction in Java 7 is one example
- Is it possible to modify a JVM to improve performance for non-Java languages?

Truffle/Graal Approach

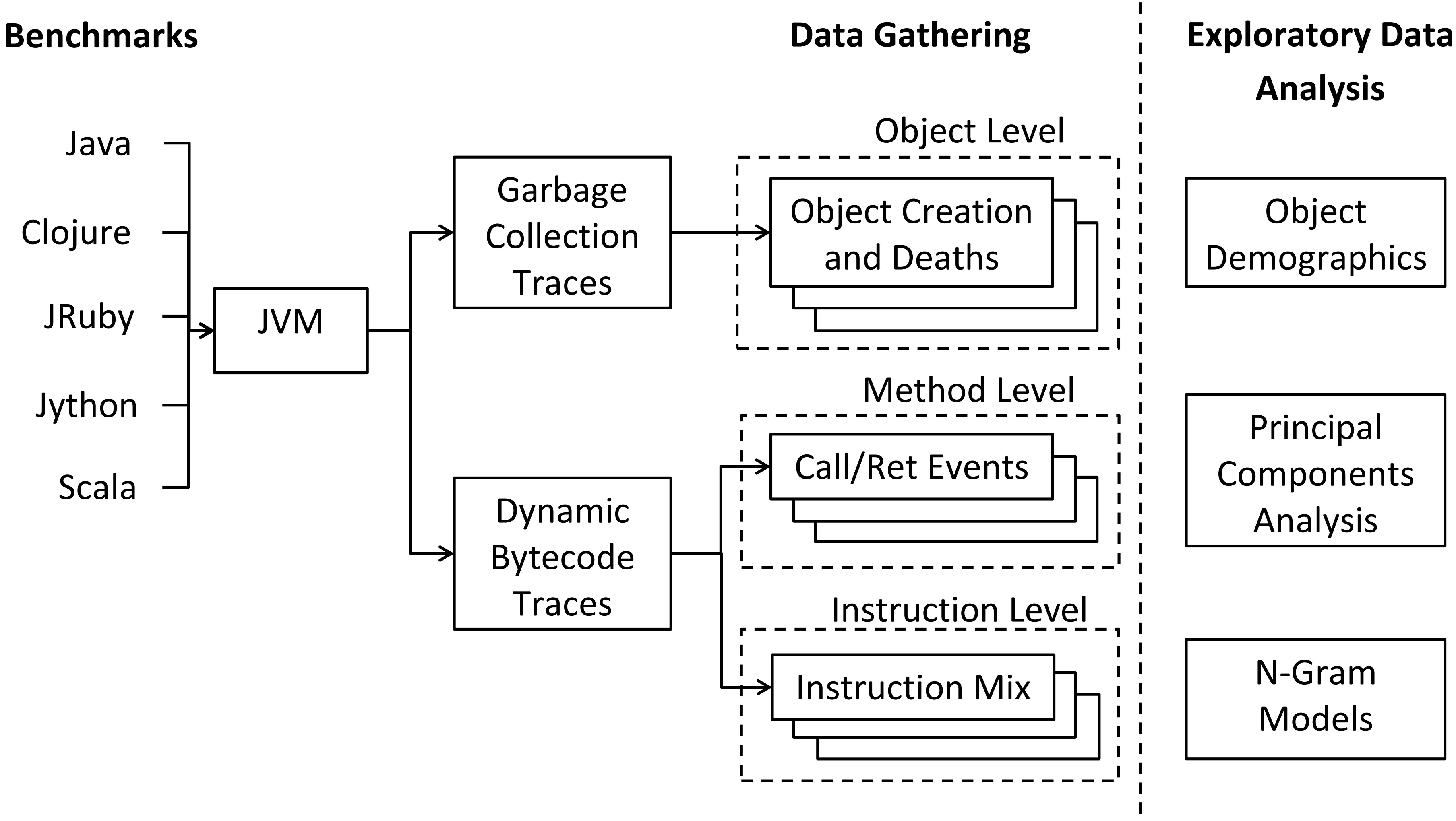
- Oracle Labs

- “One VM to rule them all”

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 a 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
  - N-grams
- 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*
  - 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

*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 Java 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 non-Java code in each non-Java language library

Non-Java Code in JVM Language Libraries

Language	Classes	Methods	Instructions
Scala	97%	99%	97%
Clojure	76%	67%	76%
JRuby	35%	13%	2%
Jython	32%	14%	4%

Analysis tools

- Principal components 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 components analysis and boxplots

[1] Exploratory Data Analysis with MATLAB by W.L. Martinez, A. Martinex 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

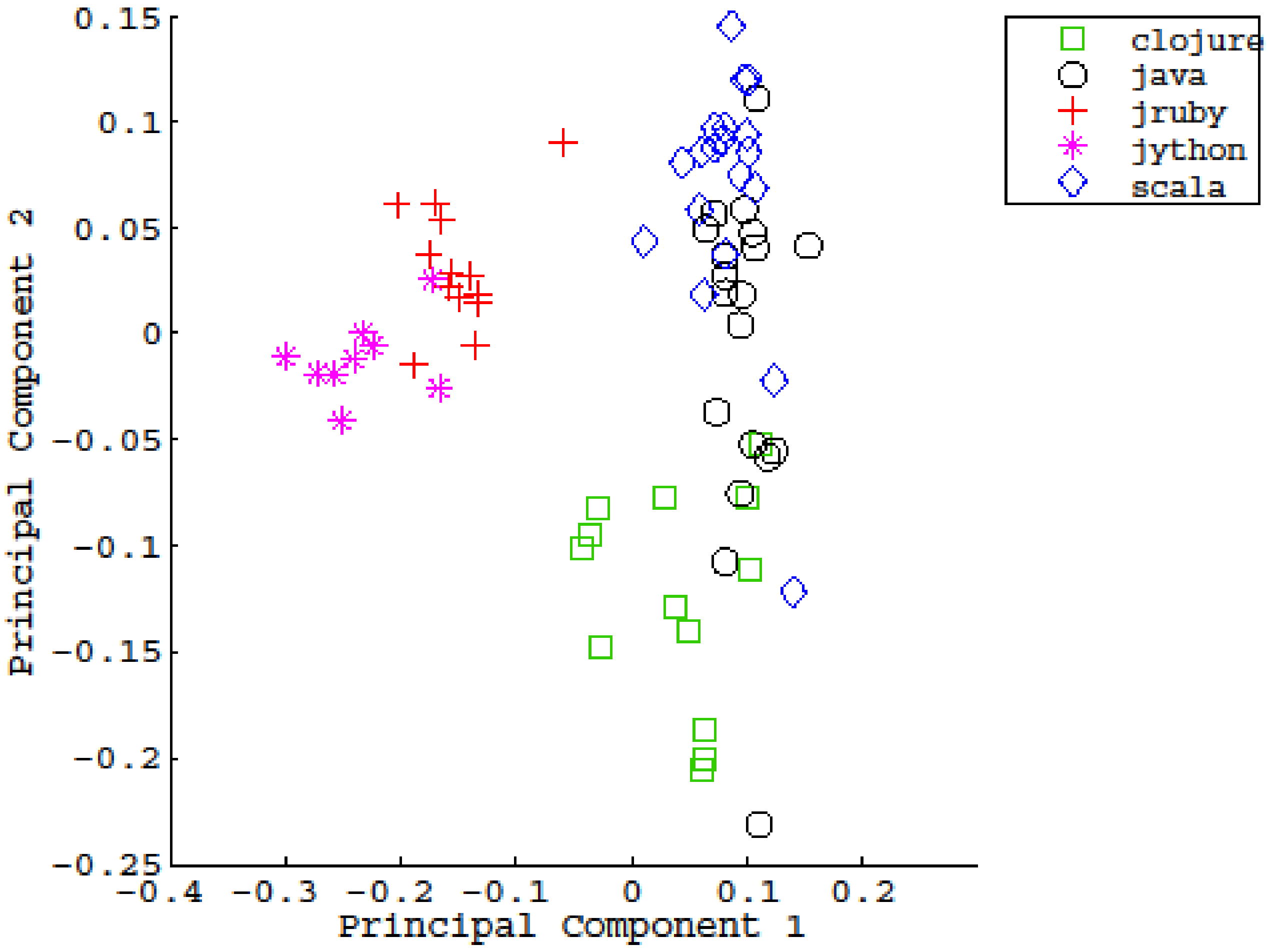
Instruction Level Results

- 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%)

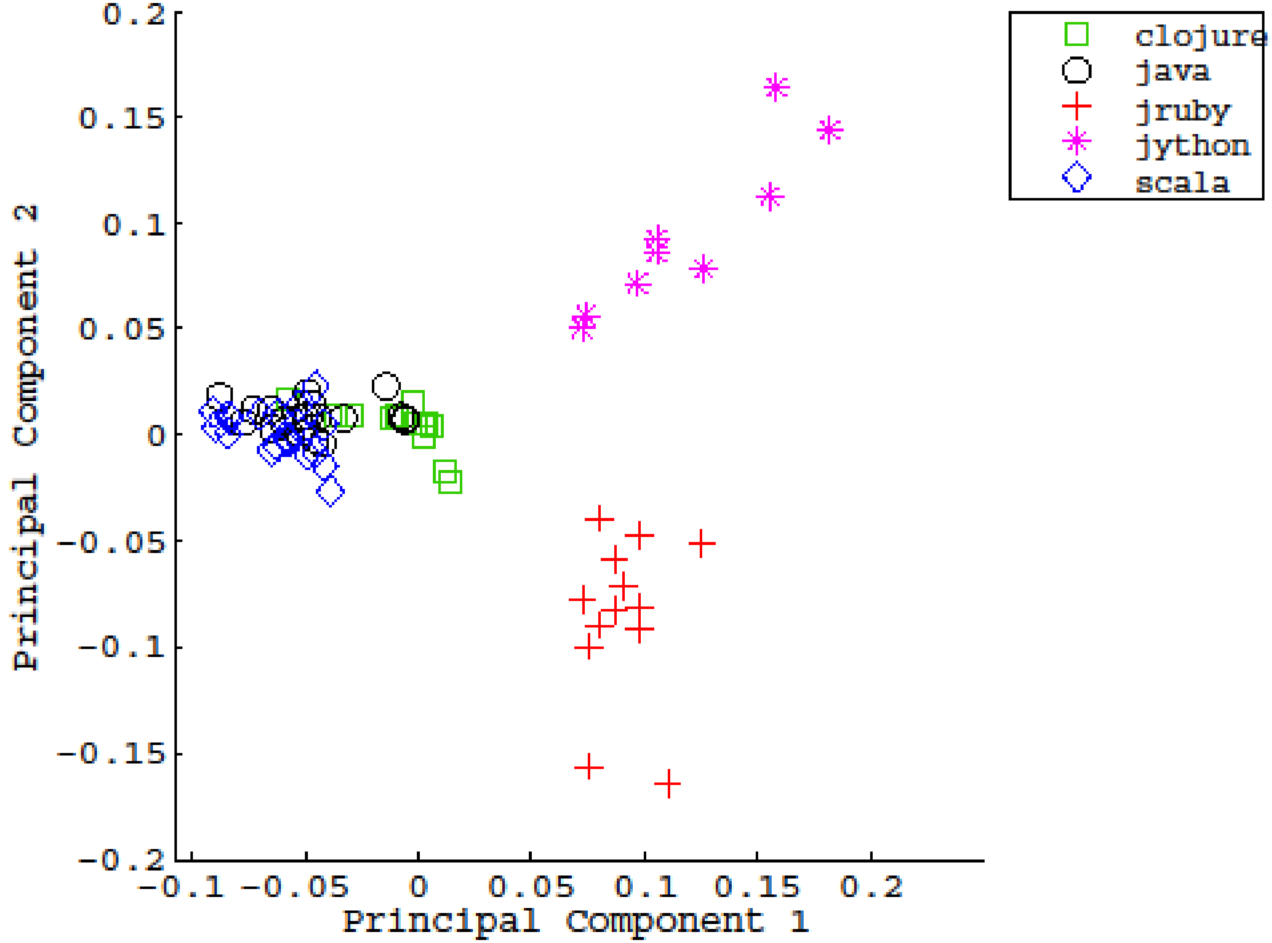
Instruction Level Results

- Principal components analysis (1-gram, filtered)



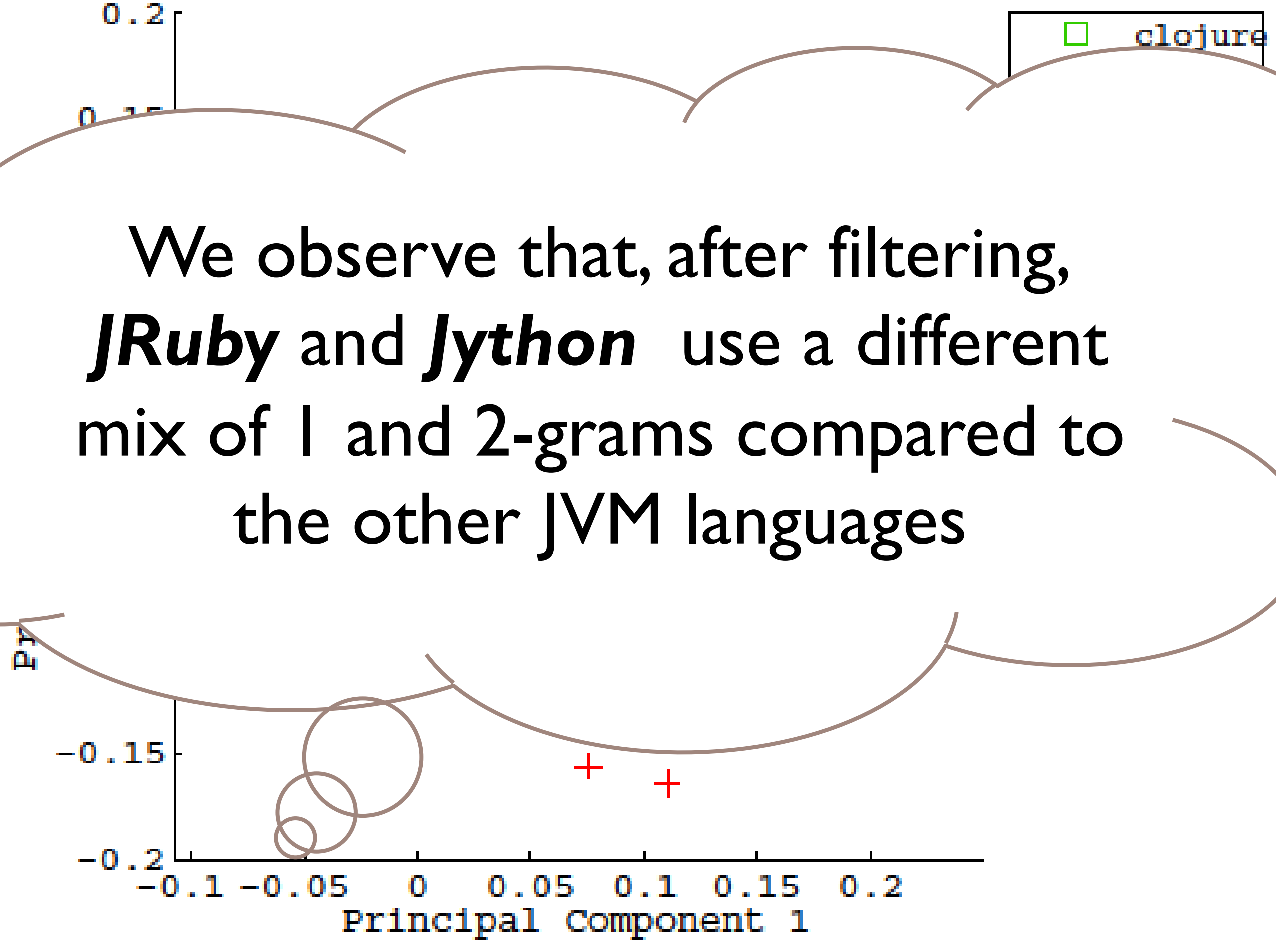
Instruction Level Results

- Principal components analysis (2-gram, filtered)



Instruction Level Results

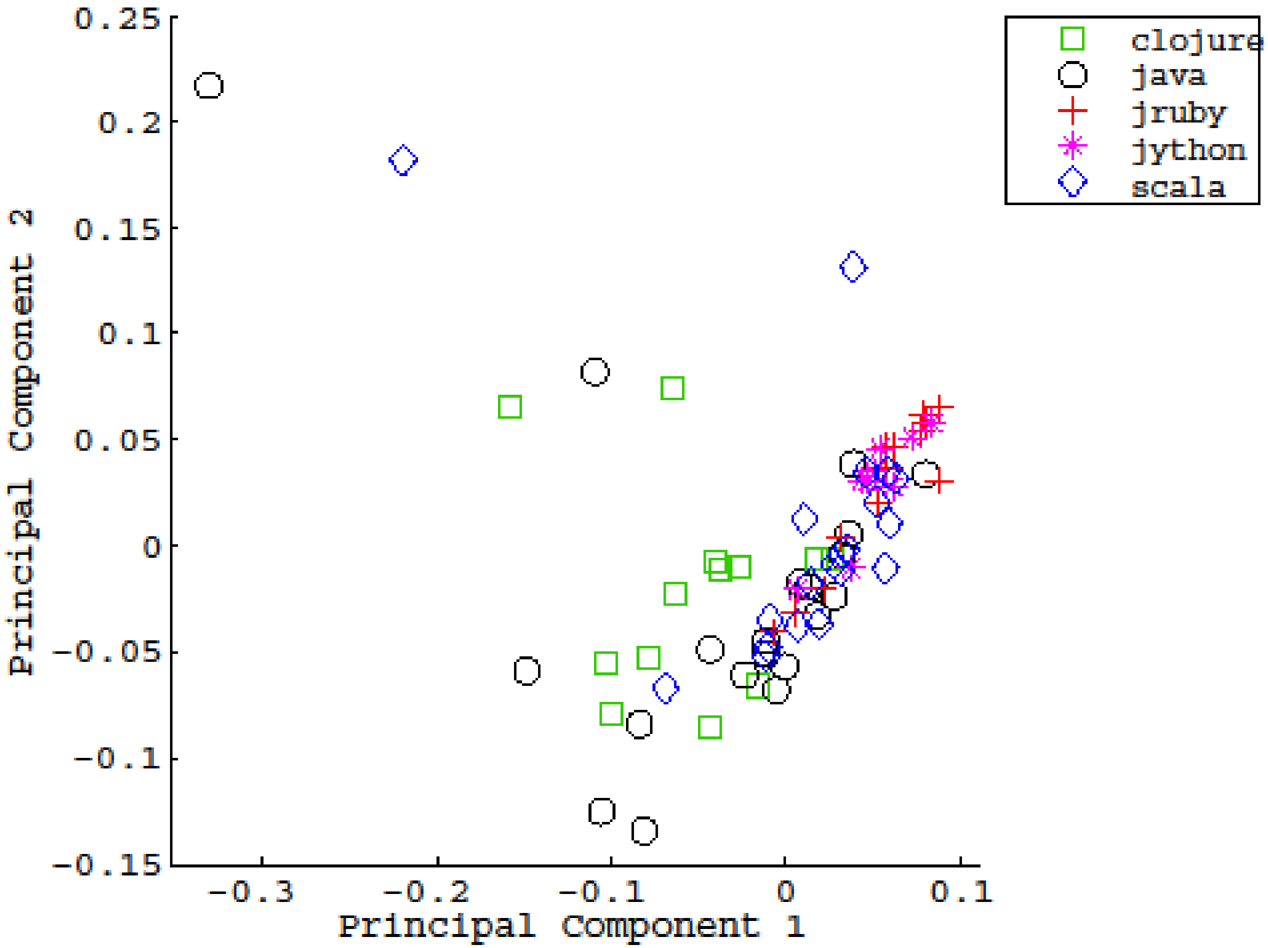
- Principal components analysis (2-gram, filtered)



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

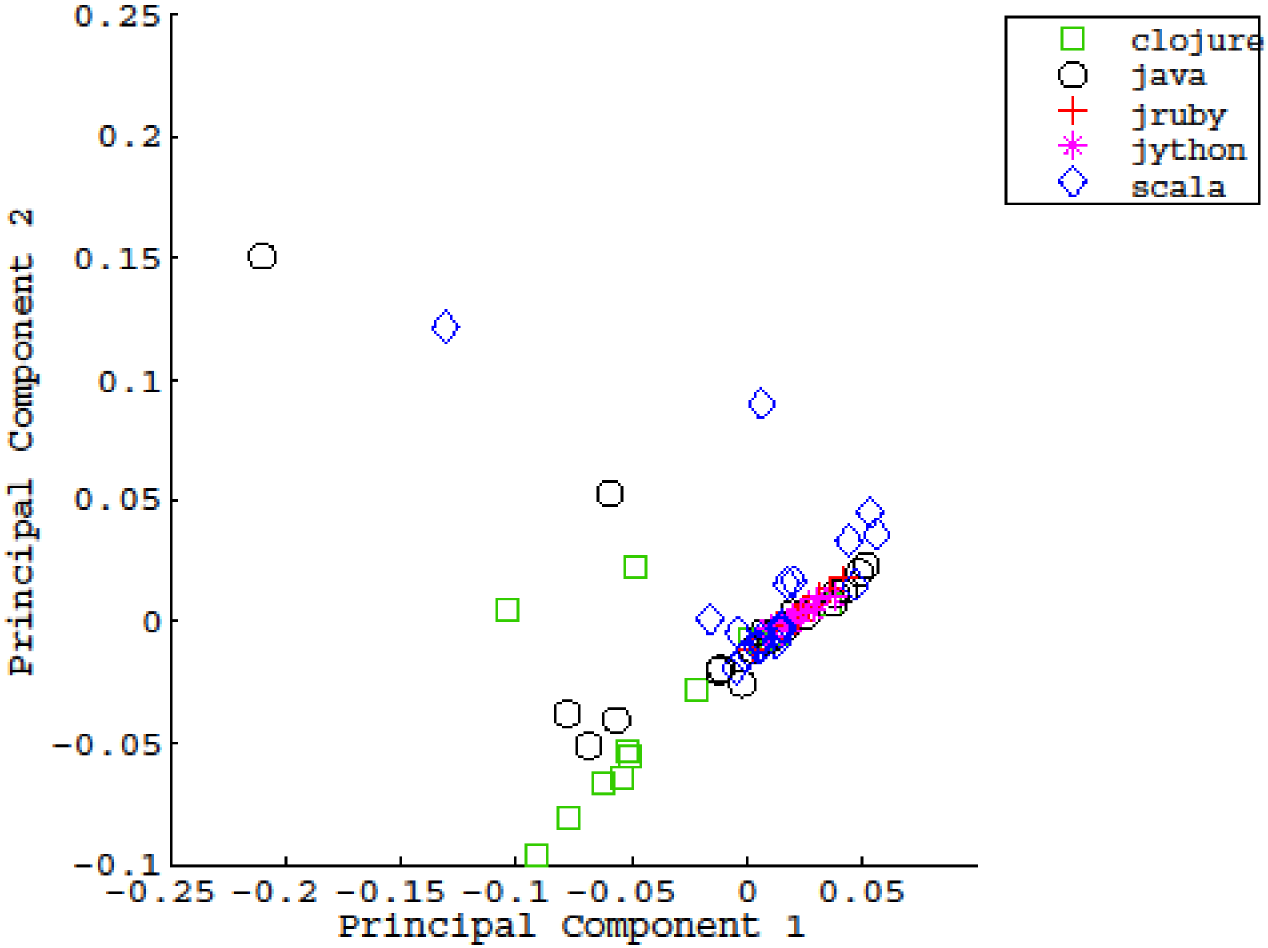
Instruction Level Results

- Principal components analysis (1-gram, unfiltered)



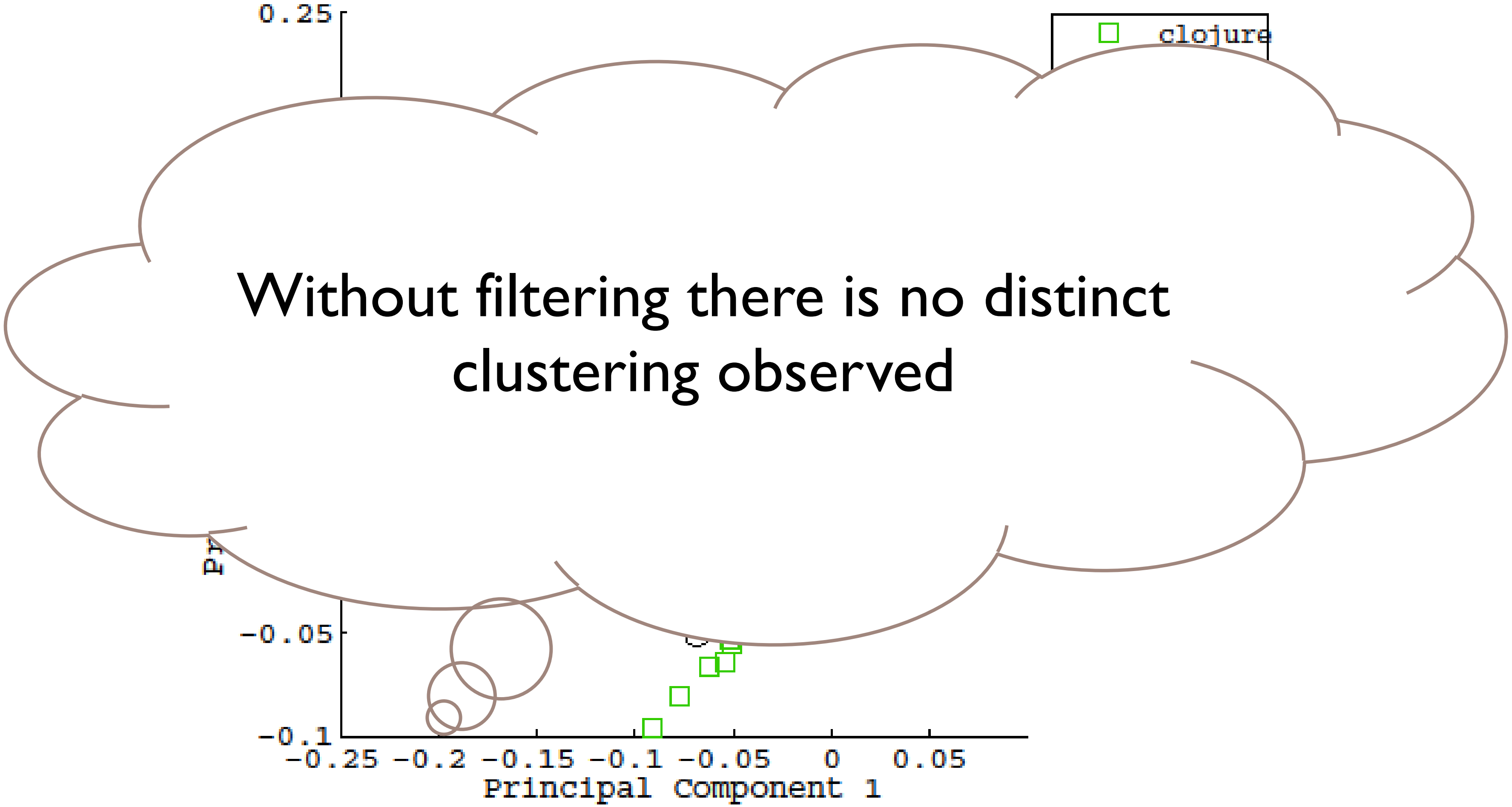
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)



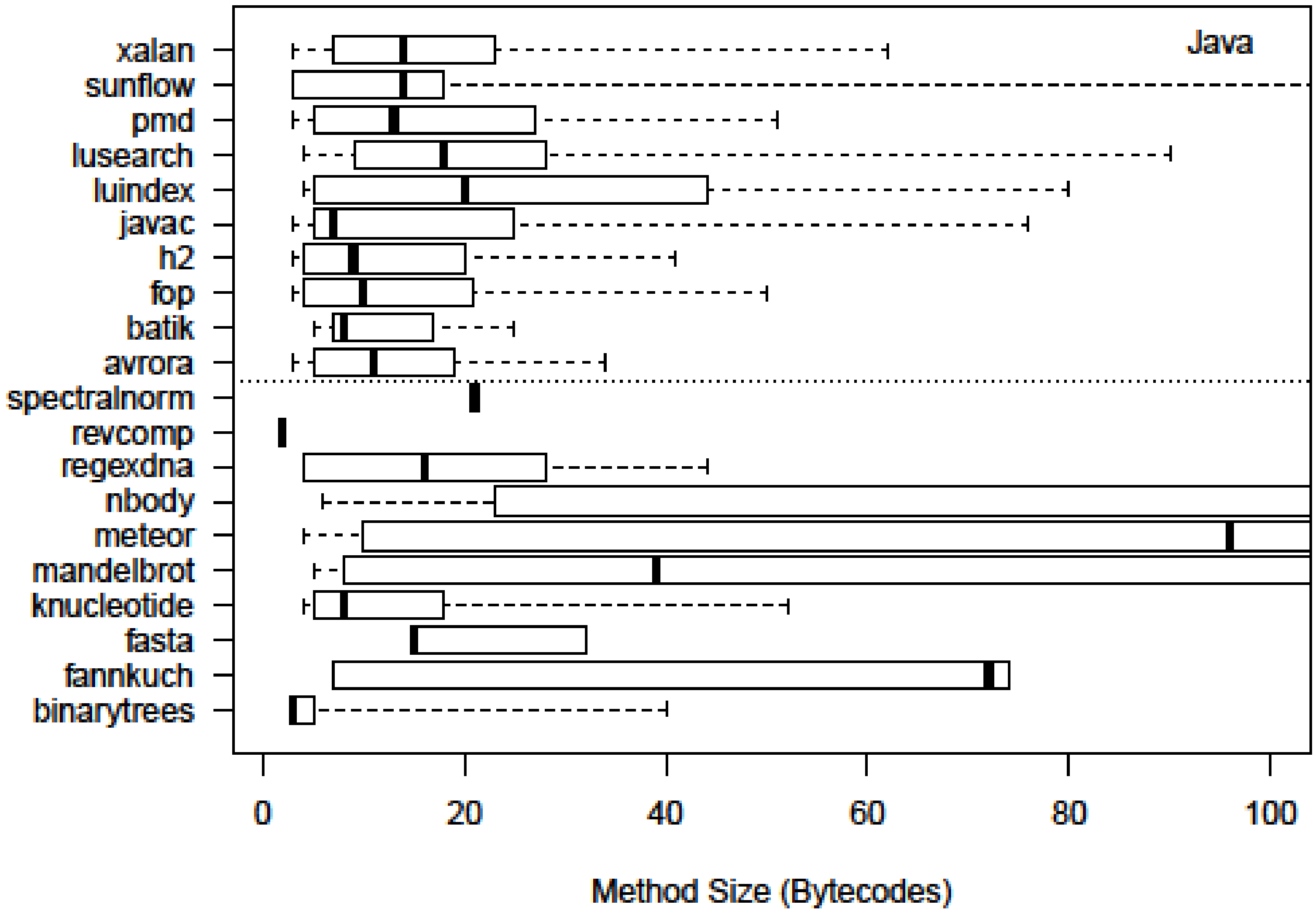
Instruction Level Results

- Principal components analysis (2-gram, unfiltered)



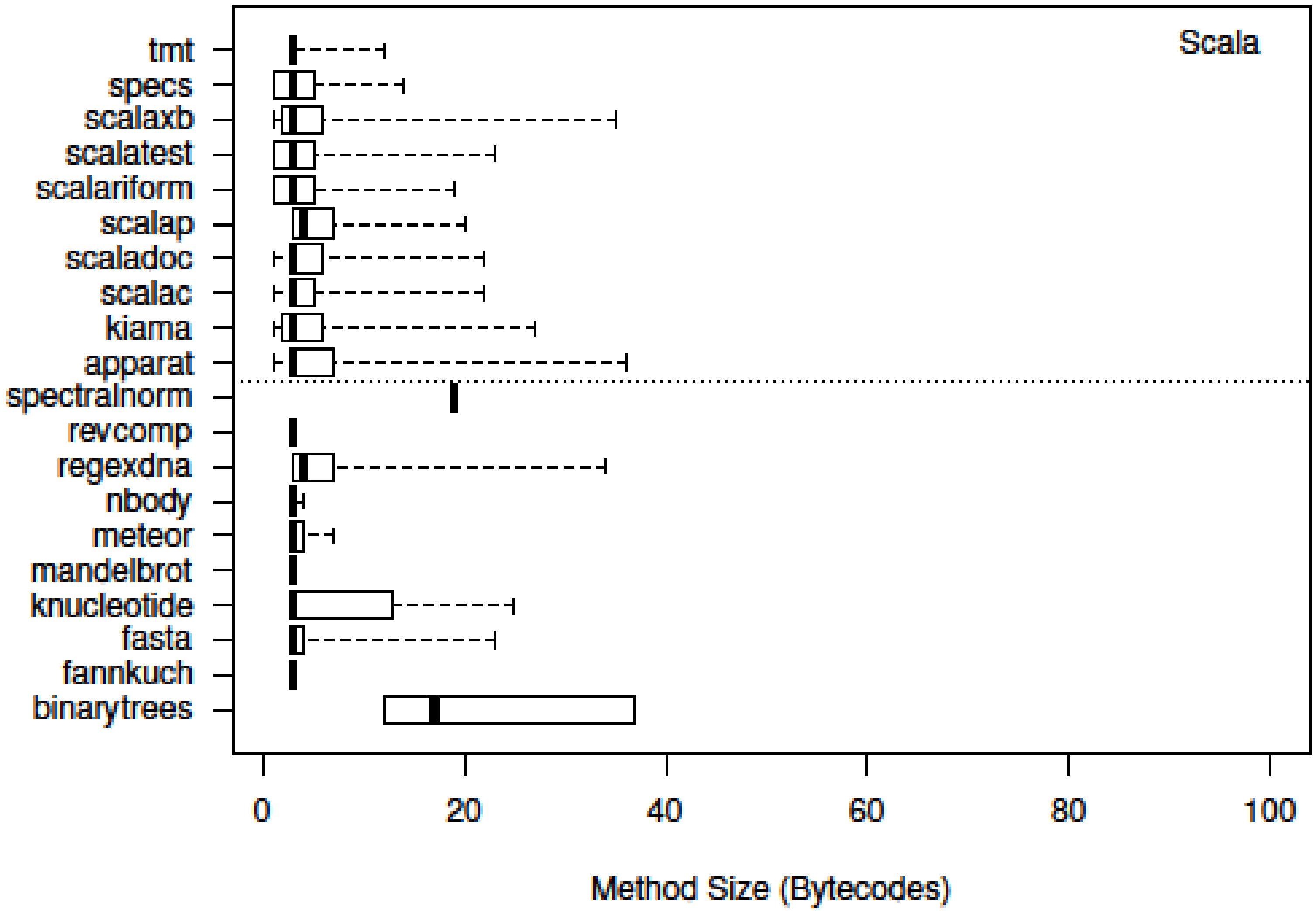
Method Level Results

- Results for the distribution of method sizes



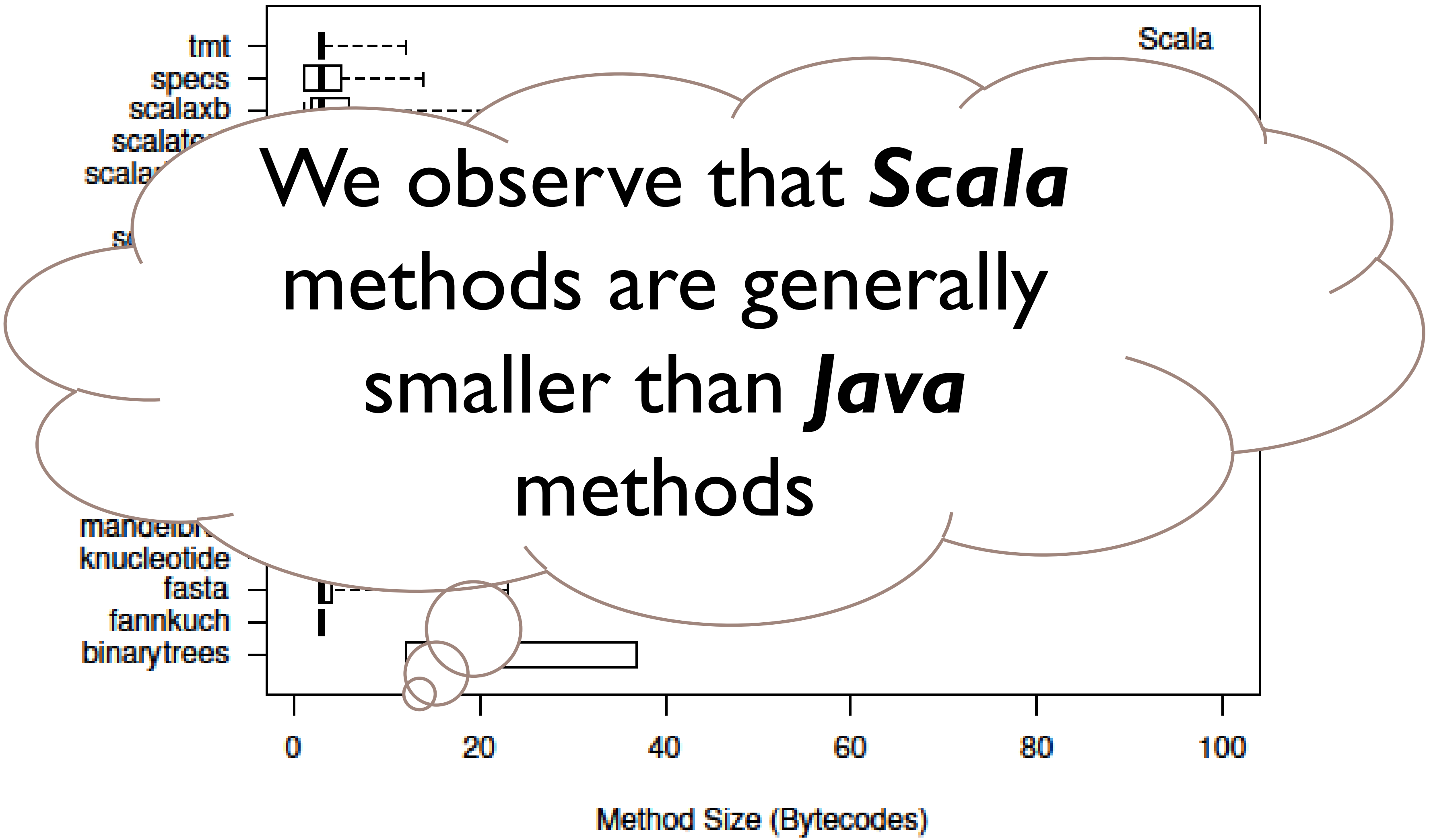
Method Level Results

- Results for the distribution of method sizes (filtered)



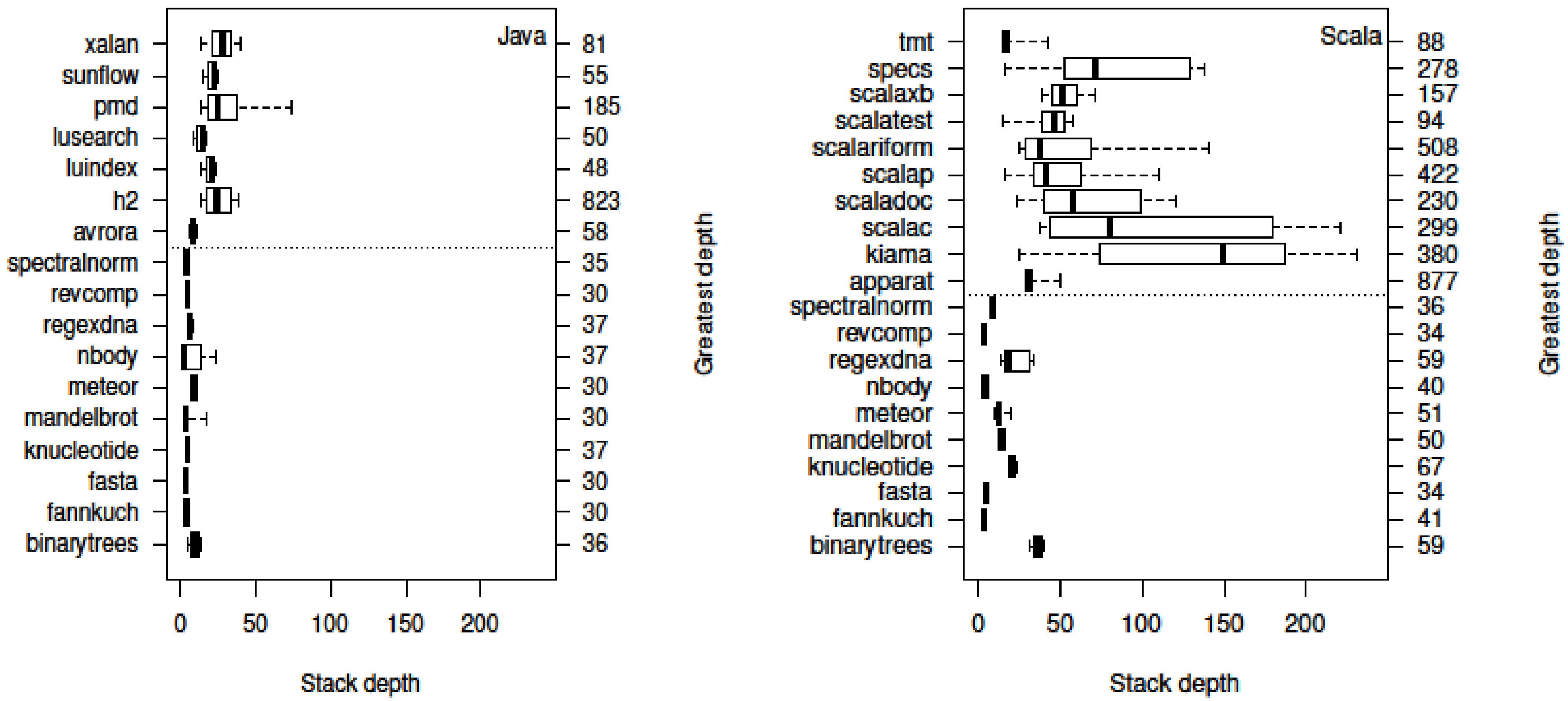
Method Level Results

- Results for the distribution of method sizes (filtered)



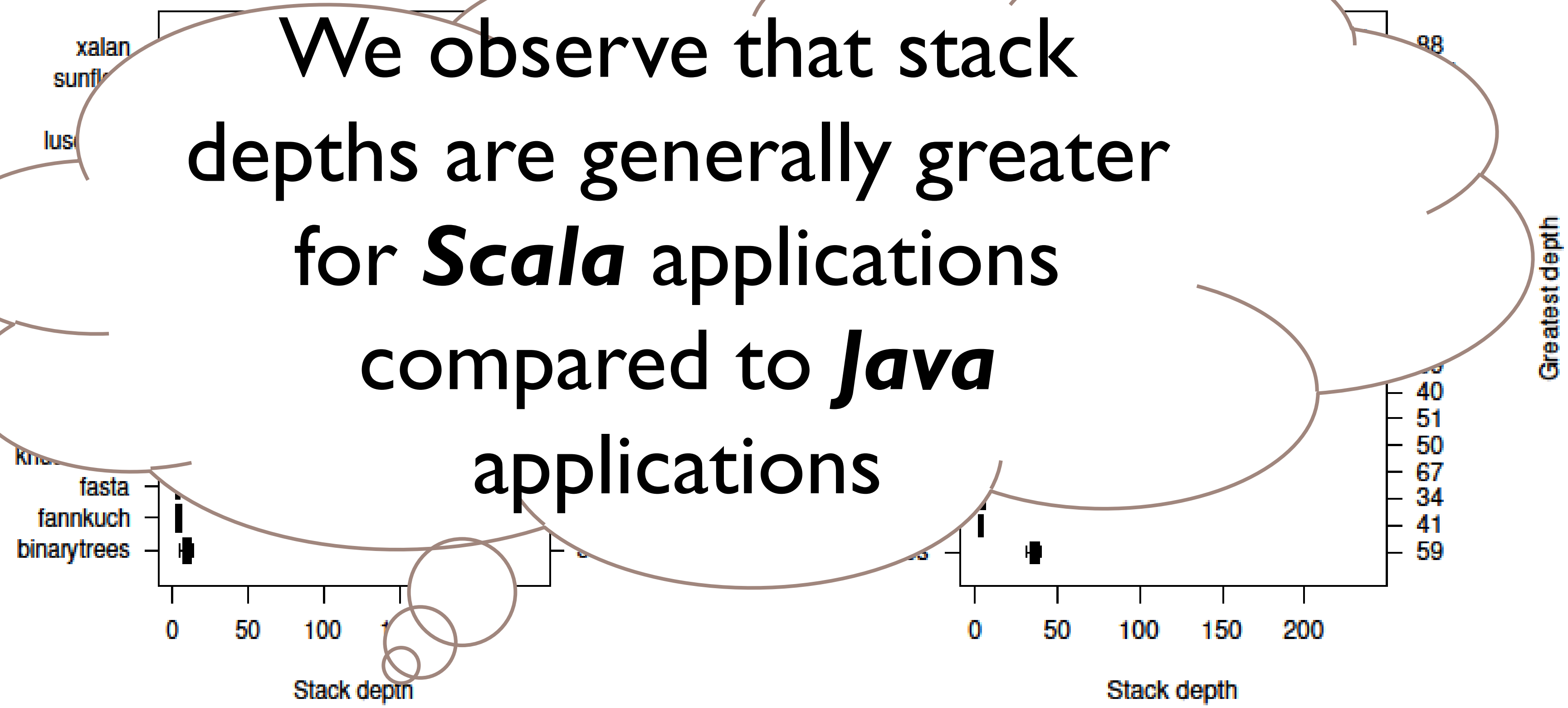
Method Level Results

- Results for the distribution of method stack depths



Method Level Results

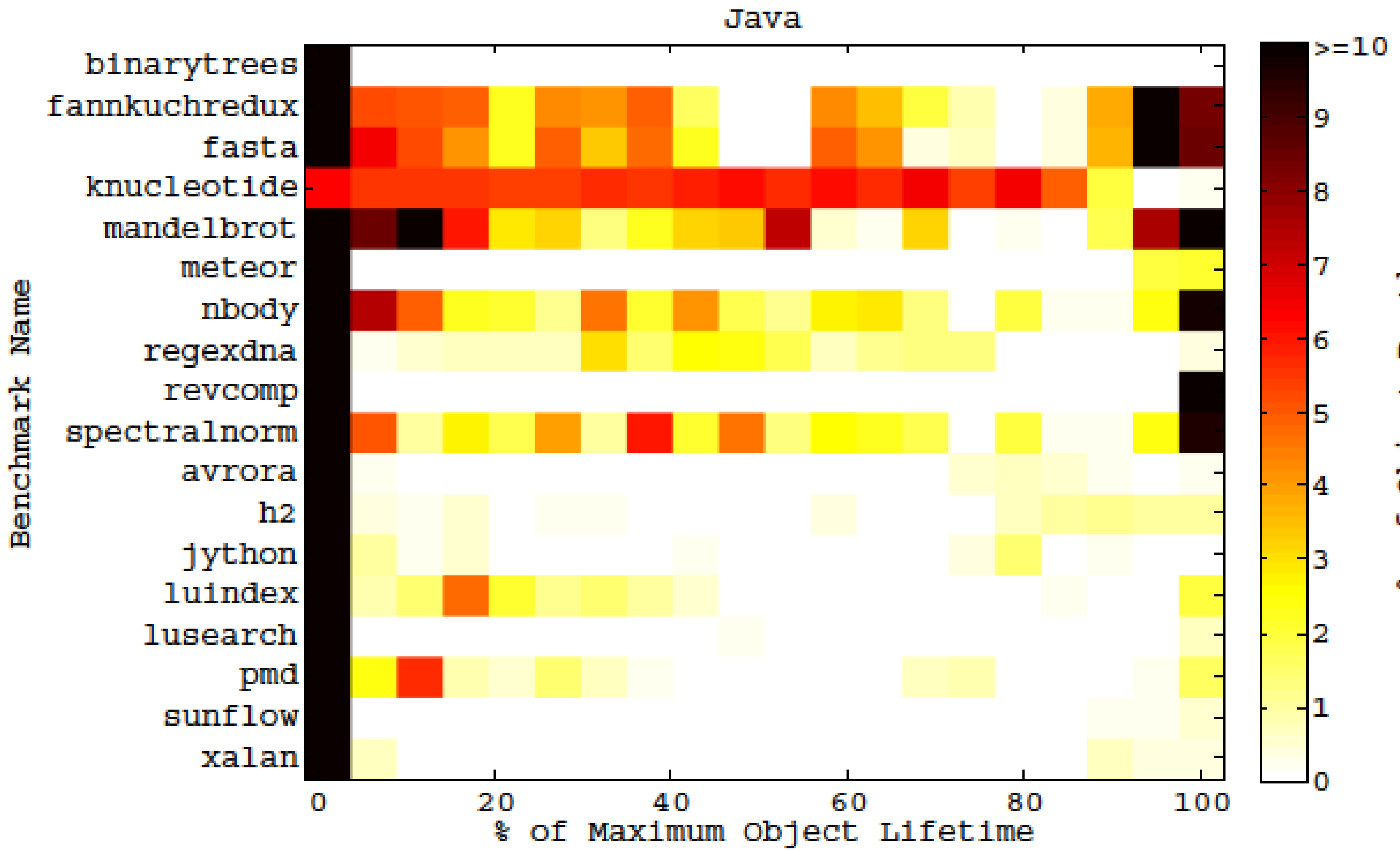
- Results for the distribution of method stack depths



We observe that stack depths are generally greater for ***Scala*** applications compared to ***Java*** applications

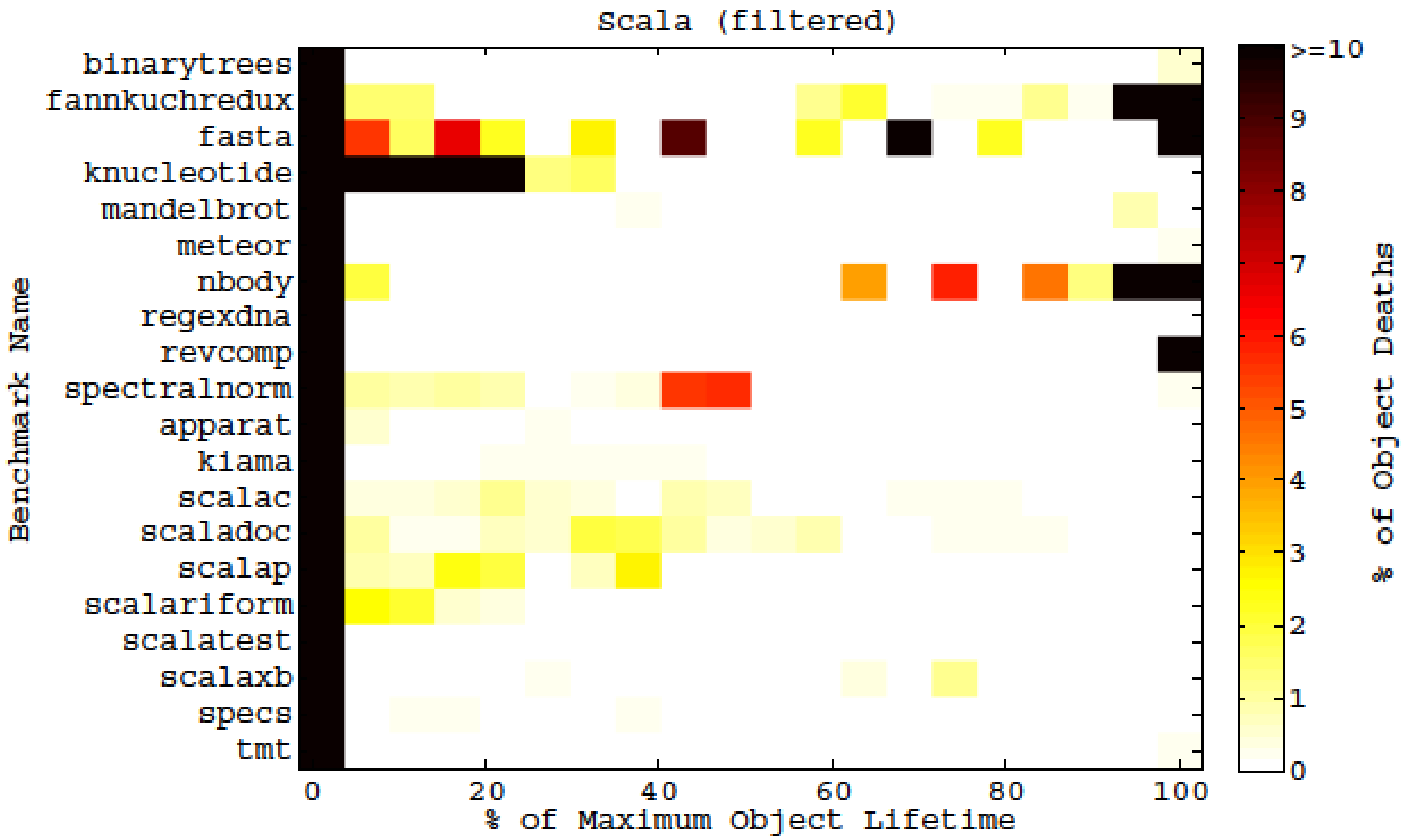
Object Level Results

- Object lifetime



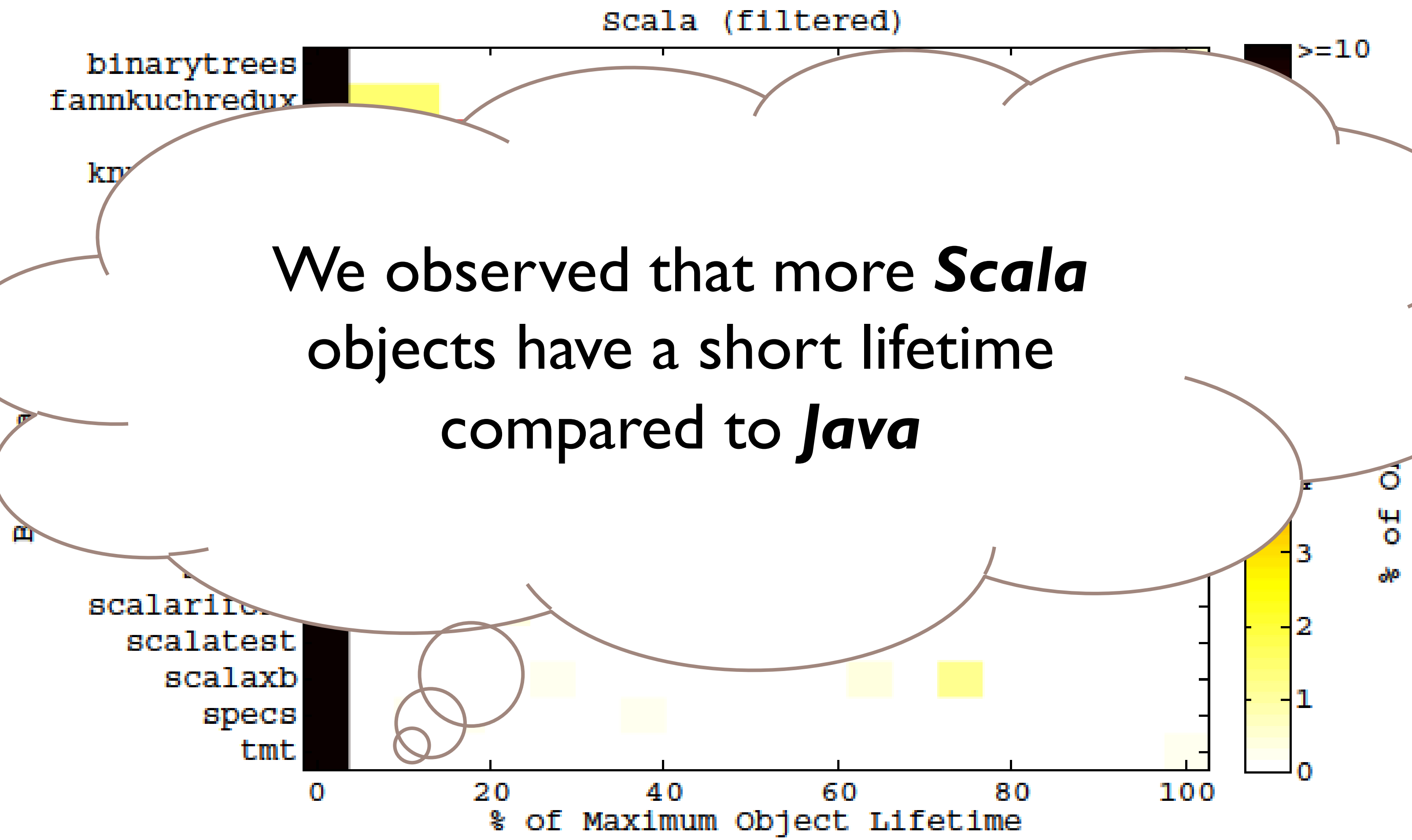
Object Level Results

- Object lifetime (filtered)



Object Level Results

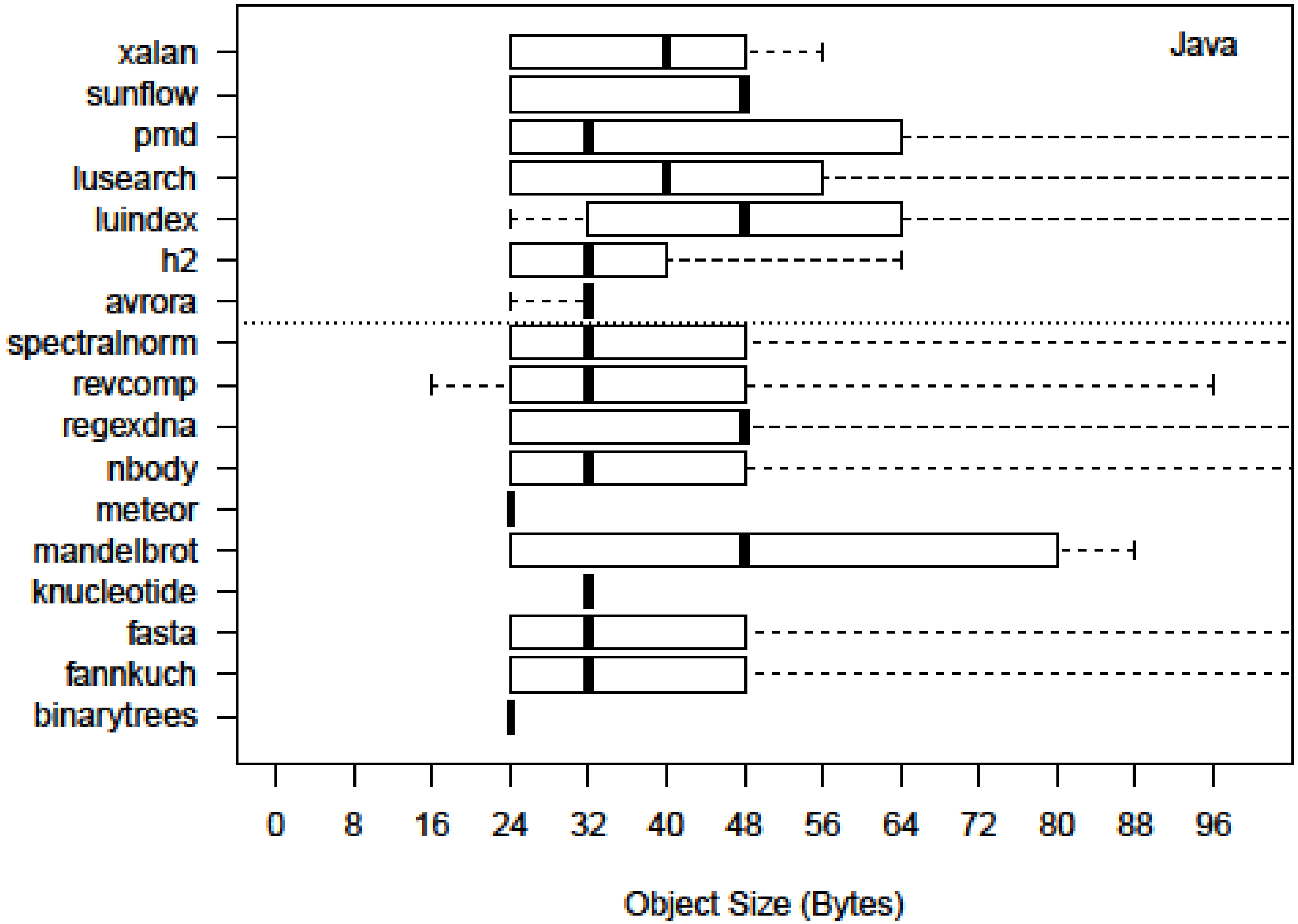
- Object lifetime (filtered)



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

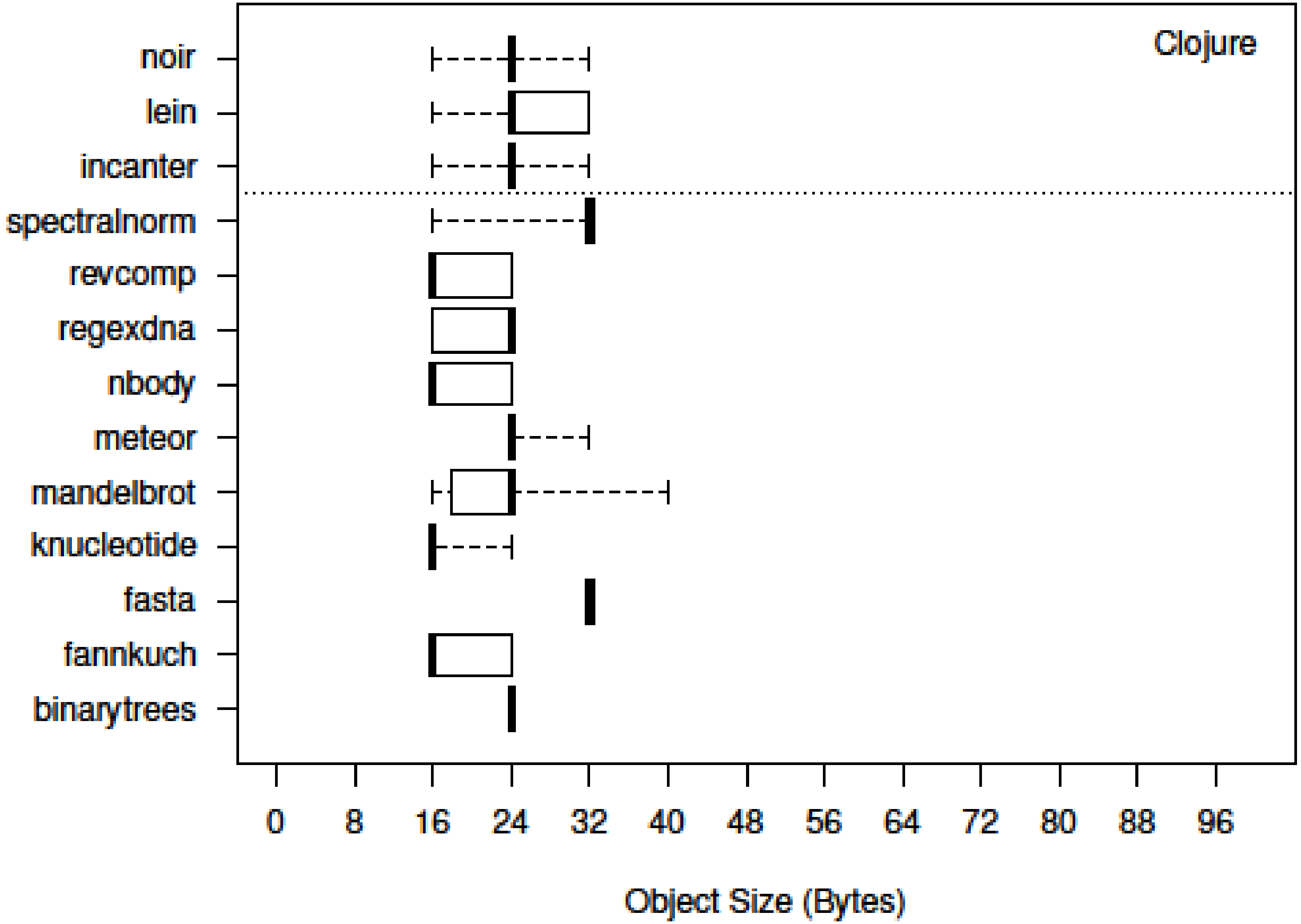
Object Sizes

- Results for the distribution of object sizes (filtered)



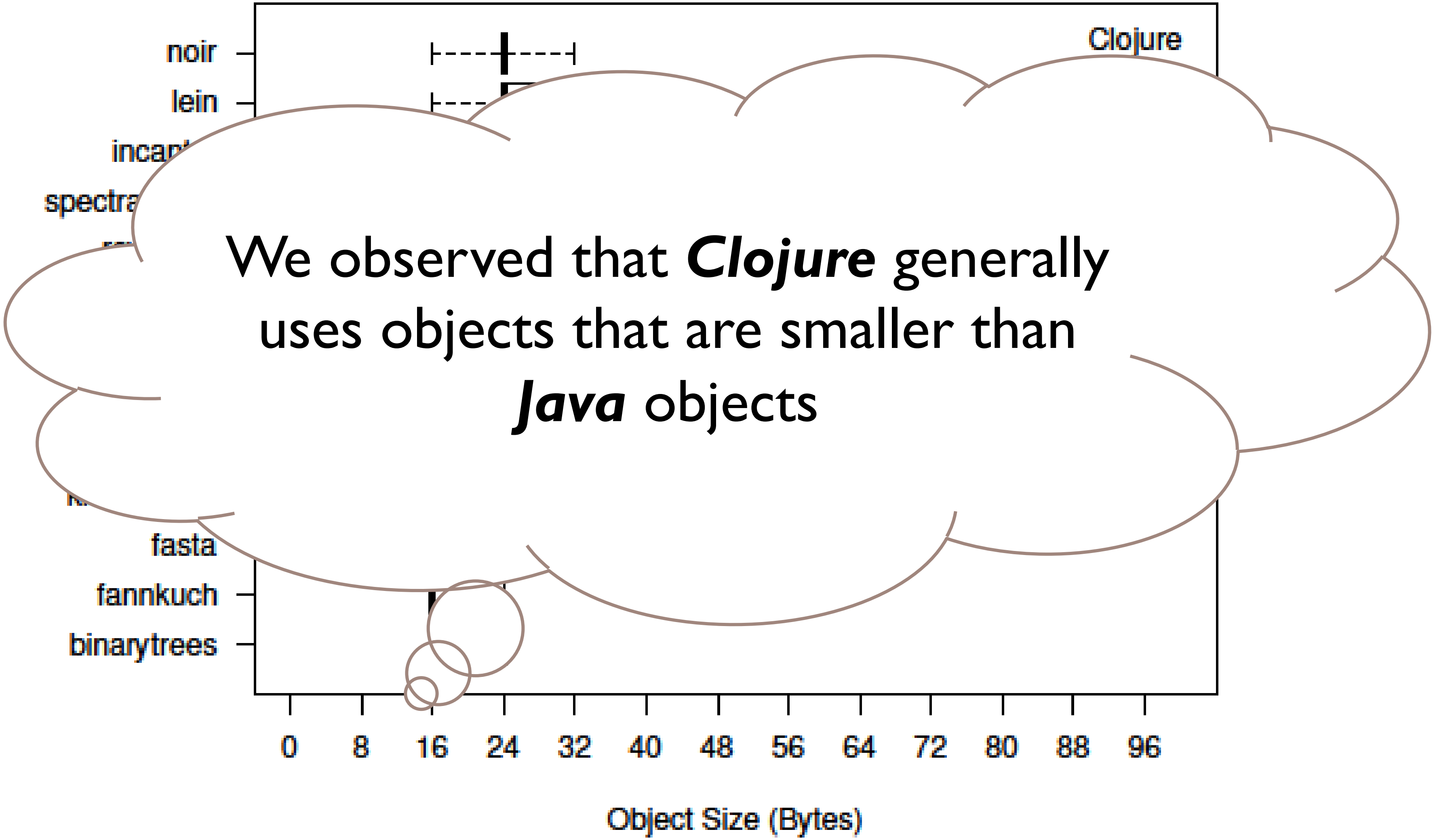
Object Sizes

- Results for the distribution of object sizes (filtered)



Object Sizes

- Results for the distribution of object sizes (filtered)



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

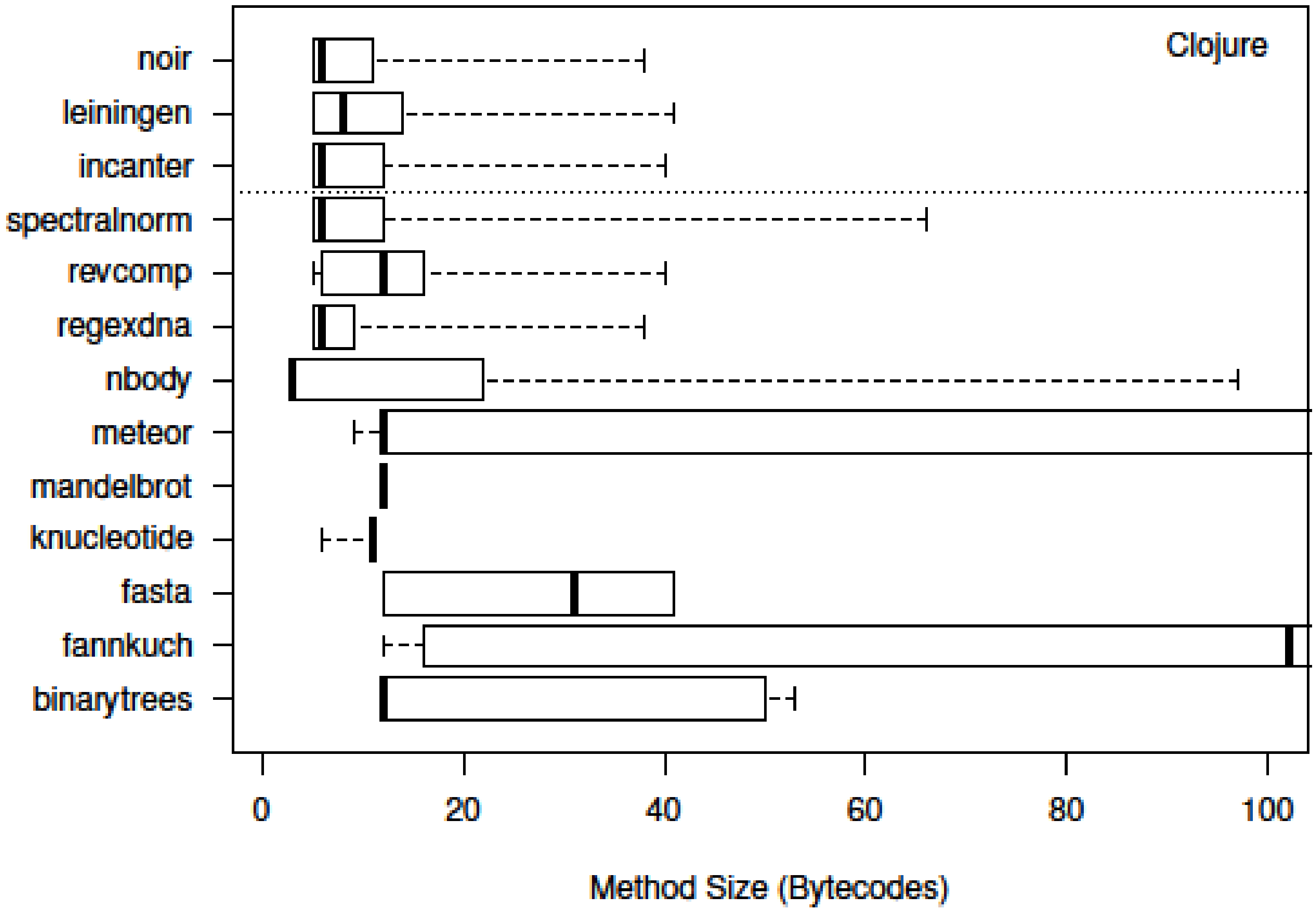
- Aim of study is to investigate the reasons for the poor performance of JVM languages
- Benchmarks in 5 JVM languages were profiled
- JVM languages do have distinctive characteristics related to their features
- Next step is to optimise performance using the observed characteristics

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

Questions?

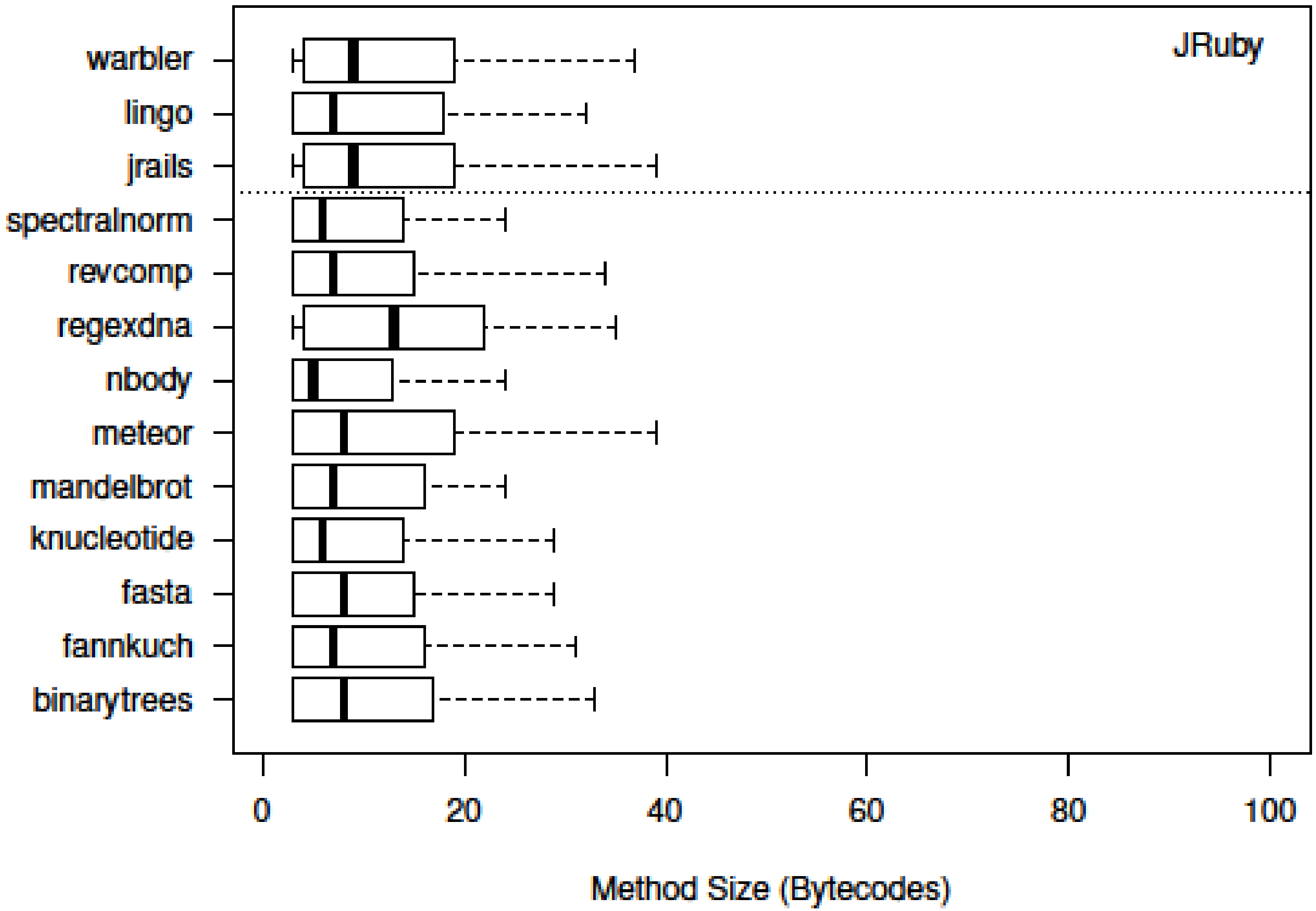
More Method Size Graphs

- Results for the distribution of method sizes (filtered)

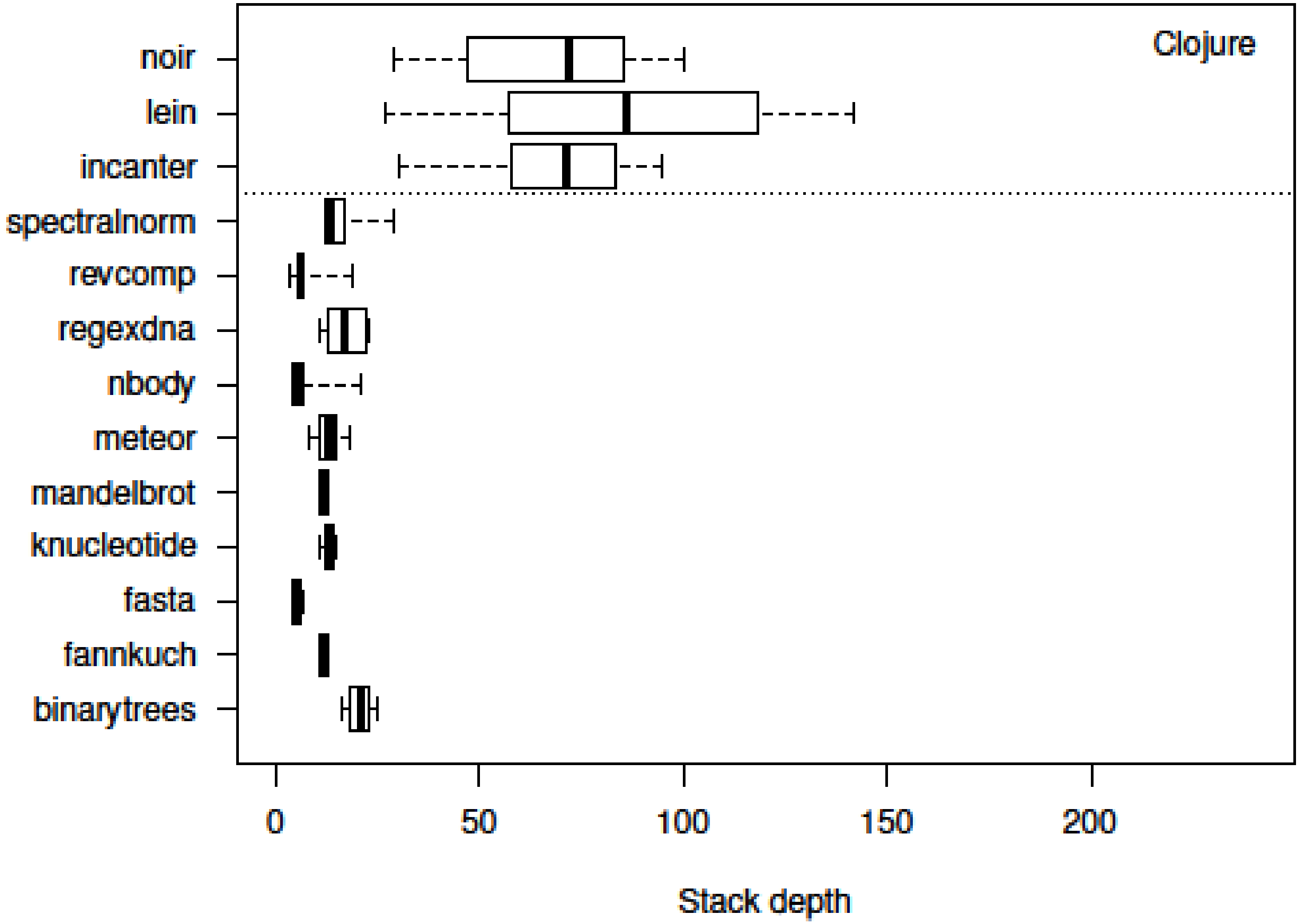


More Method Size Graphs

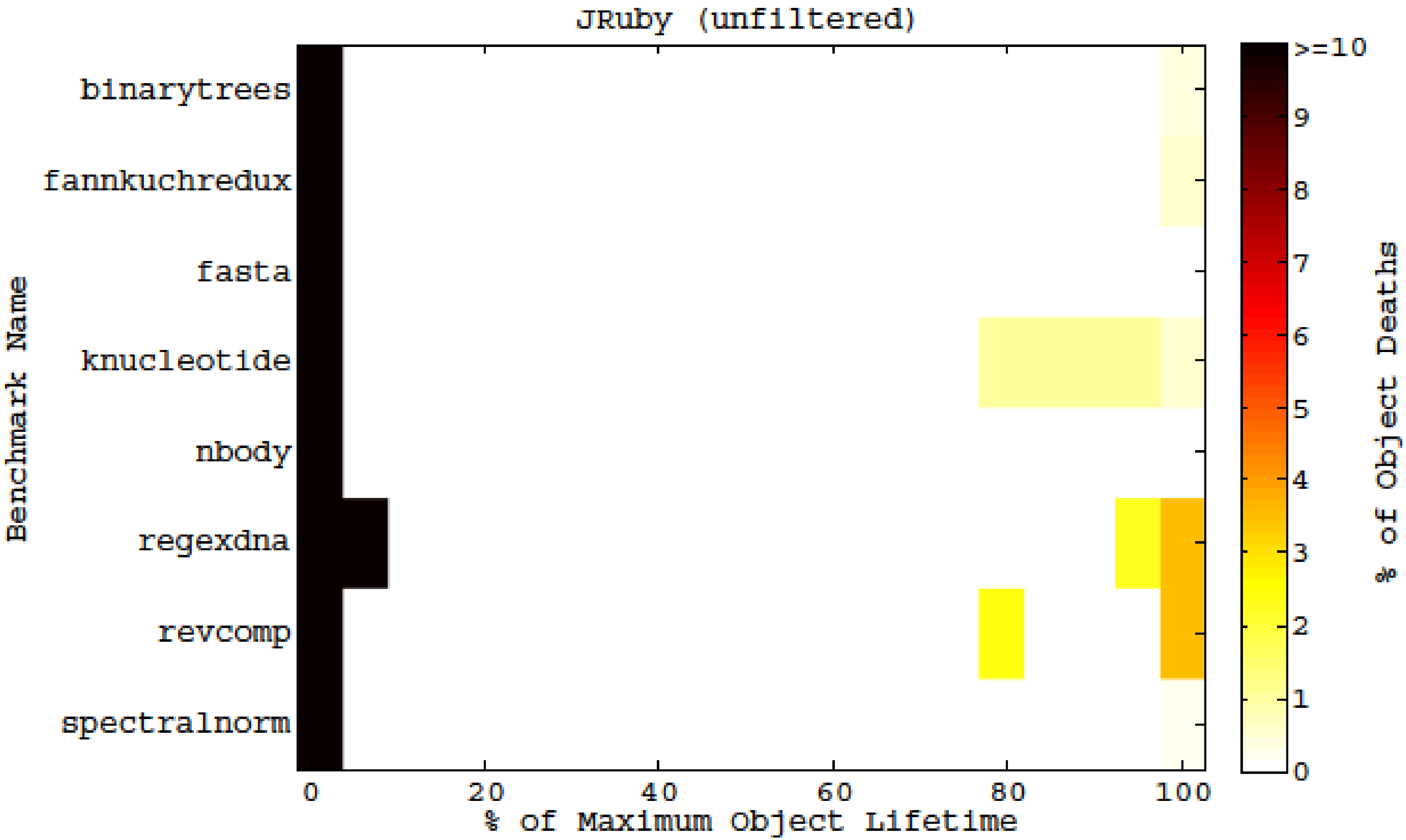
- Results for the distribution of method sizes (unfiltered)



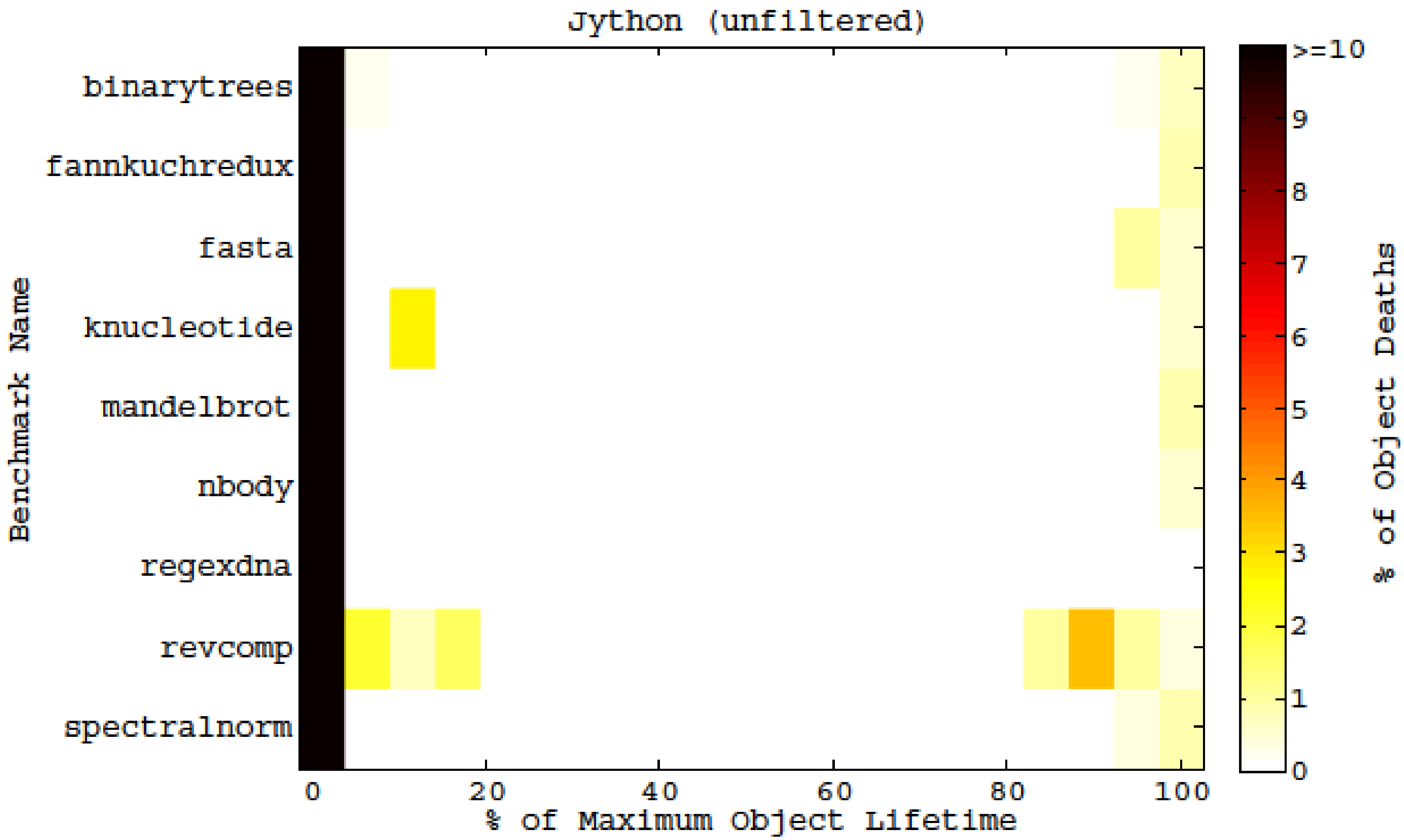
More Method Stack Depth Results



More Object Lifetime Graphs

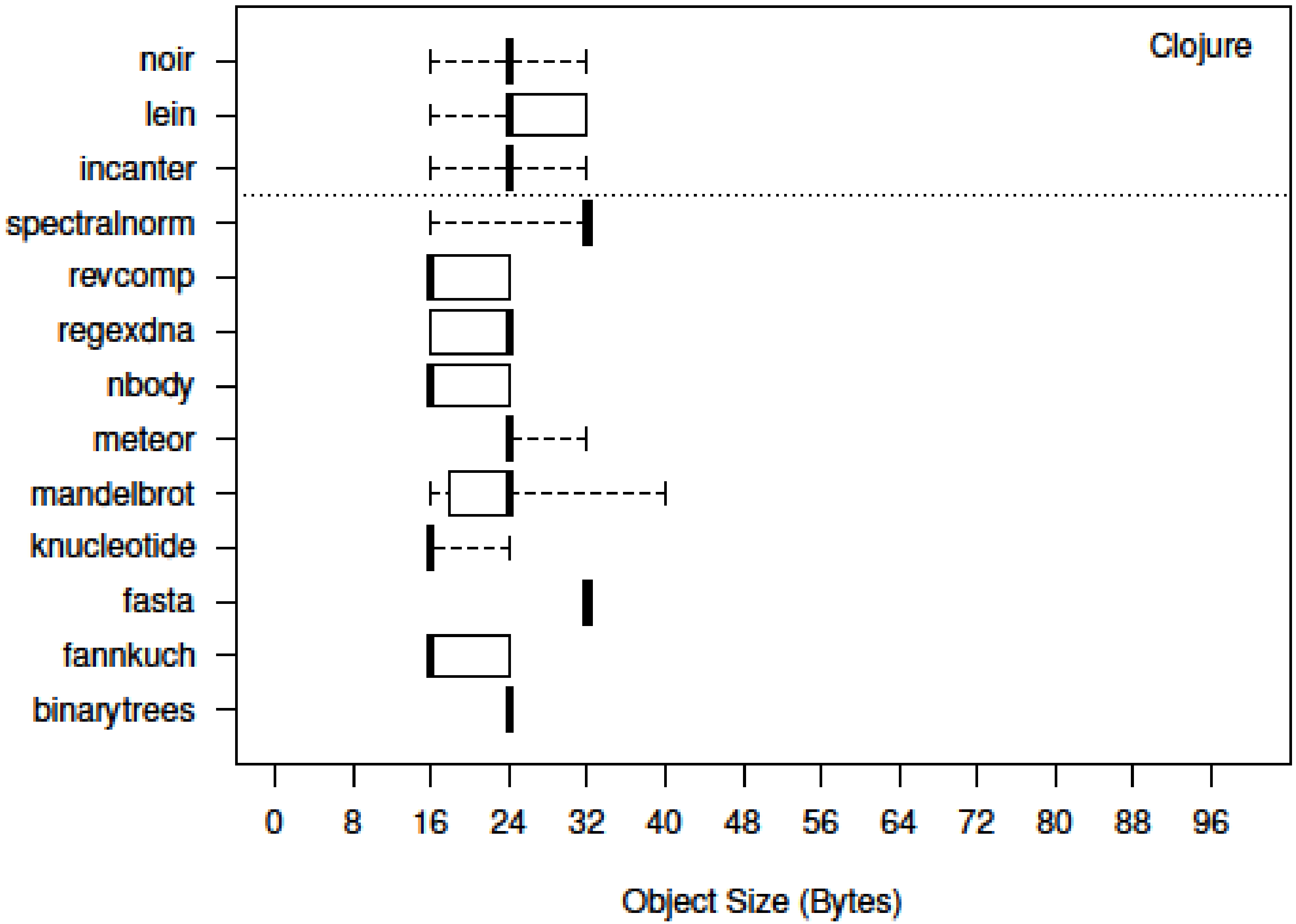


More Object Lifetime Graphs



More Object Size Graphs

- Results for the distribution of object sizes (filtered)



More Object Size Graphs

- Results for the distribution of object sizes (unfiltered)