JBrainy: Micro-benchmarking Java Collections with Interference
WORK IN PROGRESS PAPER

Noric Couderc Emma Söderberg Christoph Reichenbach
Collection Selection Problem

The Problem

Here is Java code:

```java
Set countries = new _______; // What should we write here?
s.add("France");
s.add("Sweden");
s.add("Germany");
// ...
s.contains("Norway");
// ...
s.clear();
```

The answer depends of **the context**: How will we use our collection?
Can we use the most popular?

Figure: Distribution of fastest benchmarks
Can we use the most popular?

Figure: Distribution of fastest benchmarks
Collection Selection Problem

The Problem

Here is Java code:

```java
Set countries = new _______; // What should we write here?

s.add("France");
s.add("Sweden");
s.add("Germany");
// ...
s.contains("Norway");
// ...
s.clear();
```

To pick the right implementation class you need to know:

- How you will use your collection: It’s usage profile
- Each implementation class’ strengths and weaknesses: It’s cost profile
Usage Profiles

Synthesized usage profile

- clear()
- equals(Object)
- add(Object)
- removeAll(Collection)
- size()
- hashCode()
- retainAll(Collection)
- iterator()
- toArray()
- contains(Object)
- toArray(Object[])
- containsAll(Collection)
- isEmpty()
- remove(Object)
- addAll(Collection)

number of calls

0 100 200 300 400 500
Cost Profiles

Usually, developers use tables such as this one:

![Figure: Big-O Cheatsheet by Eric Rowell](image)

These can be derived theoretically or empirically.
Cost Profiles

Usually, developers use tables such as this one:

![Big-O Cheatsheet by Eric Rowell](image)

**Figure:** Big-O Cheatsheet by Eric Rowell

These can be derived theoretically or empirically.
Cost Profiles

Usually, developers use tables such as this one:

![Big-O Cheatsheet by Eric Rowell](image)

**Figure:** Big-O Cheatsheet by Eric Rowell

These can be derived theoretically or empirically.
Assumptions

Maybe we need to consider interference between methods called?

$$\text{ArrayList} + [\text{get, add, get, ...}] = \star \star \star \star$$

$$\text{LinkedList} + [\text{get, add, get, ...}] = \star$$
Our solution: JBrainy’s Microbenchmarking

Question

How do we benchmark if there is interference?

Answer

Synthetic benchmarking, with:

- A **uniform** usage profile (Jung et al’s Brainy).
- A **more realistic** usage profile (Our tool, JBrainy).
Hardware

- Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz
- 16 GB of RAM
- Ubuntu 18.04 (Linux 4.18.0-15-generic),
- OpenJDK 10.0.2.
Results: Polyvalence

Figure: Distribution of fastest benchmarks

Collection Selection
Micro-Benchmarking
Results
Conclusion
## Results: Food for Thought

### Lists compared to ArrayList

ArrayList is faster across the board.

- LinkedList: 0.5
- Vector: 0.9

### Sets compared to HashSet

- LinkedHashSet usually faster: particularly:
  - `toArray()`: 2.96
  - `toArray(Object[])`: 2.85
  - `add()`: 2.10
- TreeSet:
  - `clear()`: 1.18

### Maps compared to HashMap

- LinkedHashMap usually faster, particularly:
  - `put()`: 1.28
  - `hashCode()`: 1.20
  - `remove()`: 1.10
- TreeMap:
  - `clear()`: 1.07

### Legend

- `method()`: Implicit traversal
- 3.0: Speedup
Why does this matter?

Our results indicate that **popular** collections are often not the optimal choice, probably because `LinkedHashMap` and `LinkedHashSet` exploit **interference**.
We discussed the issues at play when optimizing collection use.

We introduced our method of micro-benchmarking collections.

Our results indicate that:

- The most **popular** collections do **not** necessarily yield the best performance.
- Interference plays a role in collection performance.

Future work:

- Investigate the reasons for speedups
- Try with more JVMs and more hardware
- Try it on real programs!