Empirical Study of Usage and Performance of Java Collections

Diego Costa, Artur Andrzejak, Janos Seboek, David Lo

Heidelberg University, Singapore Management University
Empirical Study of Usage and Performance of Java Collections

Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo

Published as:

Paper/slides available at: https://pvs.ifi.uni-heidelberg.de/publications/
Collections

• Collections are objects that **groups** multiple elements into a single unit.
  • Use its metadata to track, access and manipulate its elements
Collections

- Collections are objects that **groups** multiple elements into a single unit.
  - Use its metadata to track, access and manipulate its elements
Collections

• Collections are objects that **groups** multiple elements into a single unit.
  • Use its metadata to track, access and manipulate its elements

```
ArrayList<Integer>
```

![Diagram of Collections](image)
Motivation

• Numerous studies have identified the inefficient use of collections as the main cause of runtime bloat.

Execution Time
+17% Improv.
Configuration of one HashMap instance
[Liu et al. 2009]

Memory Usage
+54% Improv.
Use of ArrayMaps instead of HashMaps
[Ohad et al. 2009]

Energy Consumption
+300% Improv.
Use of ArrayList instead of LinkedList
[Jung et al. 2016]
Collection Frameworks

• The Java Collection Framework offers a **Standard** implementation of the major collection abstractions
  • Stable and reliable framework
  • Easy to use

• However, there exist alternative libraries that provide a myriad of different implementations:
  • Primitive Collections (IntArrayList)
  • Immutable Collections
  • Multimaps (Map<K, Collection>)
  • Multisets (Map<K, int>)
The Java Collection Framework offers a **Standard** implementation of the major collection abstractions

- Stable and reliable framework
- Easy to use

However, there exist alternative libraries that provide a myriad of different implementations:
- Primitive Collections (`IntArrayList`)
- Immutable Collections
- Multimaps (`Map<K, Collection>`)  
- Multisets (`Map<K, int>`)
Collection Frameworks

• The Java Collection Framework offers a **Standard** implementation of the major collection abstractions
  • Stable and reliable framework
  • Easy to use

• However, there exist alternative libraries that provide a myriad of different implementations:
  • Primitive Collections (`IntArrayList`)
  • Immutable Collections
  • Multimaps (`Map<K, Collection>`)  
  • Multisets (`Map<K, int>`)  

  Unsupported features
  Simplified API
Analysis of Performance Impact of Alternative Collections

**Goal**: Can we find alternatives to the Standard collection types which improve performance on time/memory?

1. **Study on Collections Usage**
   - How often do programmers use alternative implementations?

2. **Experimental Evaluation of popular Java Collection Libraries**
   - Are there better alternatives to the most commonly used Collections regarding performance?
Study on Usage of Collections

• Dataset
  • We analyze the GitHub Java Corpus
    • 10K projects
    • 268 MLOC

• Static Analysis
  • Use of Java Parser to extract variable declaration and allocation sites of Types with suffix:
    {List, Map, Set, Queue, Vector}
  • Manually removed false-positives
Developers Rarely Use non-Standard Collections

- Lists: 57%
- Sets: 15%
- Maps: 28%
- ArrayList: 48%
- HashMap: 22%
- HashSet: 10%
Developers Rarely Use non-Standard Collections
Developers Rarely Use non-Standard Collections

- Top 4 represent 86% of all declared instantiations
- Non-Standard collections are declared <4%

Evaluate alternatives to ArrayList, HashMap, HashSet and LinkedList
Commonly Used Element Data Types

From the categorized data types:

- **Strings** are the most commonly held data type, followed by **Numeric**
Commonly Used Element Data Types

From the categorized data types:

- **Strings** are the most commonly held data type, followed by **Numeric**

Evaluate collections holding Strings, Integer and Long
Initial Capacity is Rarely Specified

- ArrayList
- HashMap
- HashSet

Specified
Default
Copy

- List
- Map
- Set
Initial Capacity is Rarely Specified

Evaluate collections with default Initial Capacity
Superior Alternatives

• **Superior Alternative:** a Non-Standard implementation that can outperform a Standard counterpart in terms of execution time and/or memory consumption.

• Can we find a superior alternative to the most commonly used collection types?
Experimental Study on Java Collections

- We selected 6 alternative libraries:
  - Repository Popularity (GitHub)
  - Appearance in previous partial benchmarks

<table>
<thead>
<tr>
<th>Libraries</th>
<th>Version</th>
<th>JCF Compatible</th>
<th>Available at</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trove</td>
<td>3.0.3</td>
<td>yes</td>
<td>trove.starlight-systems.com</td>
</tr>
<tr>
<td>Guava</td>
<td>18.0</td>
<td>yes</td>
<td>/google/guava</td>
</tr>
<tr>
<td>GSCollections</td>
<td>6.2.0</td>
<td>yes</td>
<td>/goldmansachs/gs-collections</td>
</tr>
<tr>
<td>HPPC</td>
<td>0.7.1</td>
<td>no</td>
<td>github</td>
</tr>
<tr>
<td>Fastutil</td>
<td>7.0.10</td>
<td>yes</td>
<td>/vigna/fastutil</td>
</tr>
<tr>
<td>Koloboke</td>
<td>0.6.8</td>
<td>yes</td>
<td>/leventov/Koloboke</td>
</tr>
</tbody>
</table>
Experimental Study on Java Collections

• Seven typical scenarios evaluated
  • populate, iterate, contains, get, add, remove, copy

• Collections holding from 100 to 1 million elements

• Alternatives to the most commonly used collections
  • JDK 1.8.0_65
  • ArrayList, HashMap, HashSet and LinkedList
    • Object collection alternatives
    • Primitive collection alternatives
We create a benchmark suite: **CollectionsBench**

- Open Java Microbenchmark Harness

```java
@Setup
public void setup() {
    fullList = this.createNewList();
    fullList.addAll(values); // Randomly generated
}

@Benchmark
public void iterate() {
    for (T element : fullList) {
        blackhole.consume(element);
        // Blackholes avoid dead-code
        // optimization
    }
}
```
We create a benchmark suite: **CollectionsBench**

Only the instantiation is needed for each collection type*

```java
@Setup
public void setup() {
    fullList = this.createNewList();
    fullList.addAll(values); // Randomly generated
}

@Benchmark
public void iterate() {
    for (T element : fullList) {
        blackhole.consume(element);
        // Blackholes avoid dead-code optimization
    }
}
```

Here we measure:

- Execution time (ns)
- Collection Overhead (allocation)
  - GC Profiler

*Only the instantiation is needed for each collection type*
Experimental Planning

• To accomplish a **steady state** performance evaluation:

```java
@Benchmark
public void iterate()
```

5s

Iteration
Experimental Planning

• To accomplish a **steady state** performance evaluation:

```java
@Benchmark
class Main {
    public void iterate() {
        // Measurement code here
    }
}
```

- Iteration: 5s
- Warmup: 10x Iteration
- Measured: 30x Iteration

Fork 1
Experimental Planning

• To accomplish a **steady state** performance evaluation:
Experimental Planning

• To accomplish a **steady state** performance evaluation:

```
@Benchmark
public void iterate()
```

**5s**

Iteration

**Warmup**

Fork 1

10x Iteration

Fork 2

10x Iteration

**Measured**

30x Iteration

AVG Time ± CI (99%)

AVG Memory allocated ± CI (99%)
Reporting Speedup/Slowdown

• We present the results of alternatives normalized to the Standard implementation performances.
  • Means with overlapping CI are set to zero

• We use the following speedup/slowdown definitions:

\[ S = \begin{cases} \frac{T_{std}}{T_{alt}}, & \text{if } T_{std} > T_{alt} \\ -\frac{T_{alt}}{T_{std}}, & \text{otherwise} \end{cases} \]
Reporting Speedup/Slowdown

• We present the results of alternatives normalized to the Standard implementation performances.
  • Means with overlapping CI are set to zero

• We use the following speedup/slowdown definitions:

\[
S = \begin{cases} 
\frac{T_{std}}{T_{alt}}, & \text{if } T_{std} > T_{alt} \\
-\frac{T_{alt}}{T_{std}}, & \text{otherwise}
\end{cases}
\]

\[S = \{0, 1.5, 1.9, 2, 7.8\} \quad \text{and} \quad S = \{0, -1.5, -1.9, -2, -7.8\}\]
Reporting Memory Overhead

• For the memory comparison we present the collection overhead reduction per element (with compressed object pointers)
Reporting Memory Overhead

• For the memory comparison we present the collection overhead reduction per element (with compressed object pointers)

For instance

Collection X: 100 bytes per element
Collection Y: 10 bytes per element

• Evaluated on the copy scenario
Reporting Memory Overhead

• For the memory comparison we present the collection overhead reduction per element (with compressed object pointers)

For instance

Collection X: 100 bytes per element
Collection Y: 10 bytes per element

• Evaluated on the copy scenario
Superior Alternatives: LinkedList

• LinkedList was outperformed by every ArrayList alternative

<table>
<thead>
<tr>
<th></th>
<th>contains</th>
<th>add</th>
<th>get</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>3</td>
<td></td>
<td>*</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>Fastutil</td>
<td>3</td>
<td></td>
<td>*</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3</td>
<td></td>
<td>*</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>HPPC</td>
<td>3</td>
<td></td>
<td>*</td>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

JCF LinkedList$Entry consumes 24 bytes

Reduction of Collection Overhead

84%
Superior Alternatives: LinkedList

- LinkedList was outperformed by every ArrayList alternative.

<table>
<thead>
<tr>
<th></th>
<th>contains</th>
<th>add</th>
<th>get</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>3</td>
<td>* * * * *</td>
<td>4 6 3</td>
<td>11 12 16 11 12</td>
<td></td>
</tr>
<tr>
<td>Fastutil</td>
<td>3</td>
<td>* * * * *</td>
<td>4 6 3</td>
<td>3 3 4 3 4</td>
<td></td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3</td>
<td>* * * * *</td>
<td>4 6 3</td>
<td>10 13 16 12 12</td>
<td></td>
</tr>
<tr>
<td>HPPC</td>
<td></td>
<td>* * * * *</td>
<td>3 6 3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

JCF LinkedList$Entry$ consumes 24 bytes

- Reduction of Collection Overhead: 84%

Asymptotic disadvantage
Superior Alternatives: LinkedList

• LinkedList was outperformed by every ArrayList alternative

<table>
<thead>
<tr>
<th></th>
<th>contains</th>
<th>add</th>
<th>get</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>4 6 3</td>
<td>11 12 16 11 12</td>
</tr>
<tr>
<td>Fastutil</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>4 6 3</td>
<td>3  3  4 3  4</td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3</td>
<td>*</td>
<td>*</td>
<td>4 6 3</td>
<td>10 13 16 12 12</td>
</tr>
<tr>
<td>HPPC</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3  6  3</td>
<td></td>
</tr>
</tbody>
</table>

Asymptotic disadvantage
Asymptotic advantage

public boolean remove(Object o)

JCF LinkedList$Entry consumes 24 bytes

Reduction of Collection Overhead 84%
Superior Alternatives: ArrayList

- *GSCollections* provides a superior alternative
  - Faster when populating the list (no time penalty)

<table>
<thead>
<tr>
<th></th>
<th>populate</th>
<th>iterate</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>GSCollec.</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>HPPC</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
</tbody>
</table>

No memory difference
Superior Alternatives: ArrayList

- **GSCollections** provides a superior alternative
  - Faster when populating the list (no time penalty)

### Distinct Array copy calls

**Std:** `Arrays.copyOf();`

**Alt:** `System.arraycopy();`

No memory difference
Superior Alternatives: ArrayList

- **GSCollections** provides a superior alternative
  - Faster when populating the list (no time penalty)

### Performance Comparison

<table>
<thead>
<tr>
<th></th>
<th>populate</th>
<th>iterate</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>GSCollec.</td>
<td></td>
<td></td>
<td>-4</td>
</tr>
<tr>
<td>HPPC</td>
<td></td>
<td></td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>-3</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>-6</td>
<td>-7</td>
</tr>
<tr>
<td>1K</td>
<td></td>
<td>-7</td>
<td>-14</td>
</tr>
<tr>
<td>10K</td>
<td></td>
<td>-14</td>
<td>-14</td>
</tr>
<tr>
<td>100K</td>
<td></td>
<td>-14</td>
<td>-9</td>
</tr>
<tr>
<td>1M</td>
<td></td>
<td>-9</td>
<td></td>
</tr>
</tbody>
</table>

- **No memory difference**

- **Distinct Array copy calls**
  - Std: `Arrays.copyOf();`
  - Alt: `System.arraycopy();`

- **HPPC adds each element instead of copying the array**
Superior Alternatives: HashSet

- Koloboke provides a superior alternative
  - Fastutil is a solid 2\textsuperscript{nd} option

<table>
<thead>
<tr>
<th></th>
<th>populate</th>
<th>iterate</th>
<th>contains</th>
<th>add</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSCollect.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guava</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Koloboke</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>Trove</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Std HashSet$\text{Node}$ object consumes \textbf{32 bytes}

Reduction of Collection Overhead

75%
Superior Alternatives: HashSet

- Koloboke provides a superior alternative
- Fastutil is a solid 2nd option

<table>
<thead>
<tr>
<th></th>
<th>populate</th>
<th>iterate</th>
<th>contains</th>
<th>add</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GSCollec.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guava</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPPC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Koloboke</td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trove</td>
<td>3</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>

Impact of memory efficiency on time: Koloboke performs a memory copy of its HashTable

Koloboke performs a memory copy of its HashTable

Std HashSet$Node object consumes 32 bytes

Reduction of Collection Overhead

75%
Superior Alternatives: HashMap

- Standard HashMap is a **solid** implementation
- No superior alternatives on time

<table>
<thead>
<tr>
<th></th>
<th>Fastutil</th>
<th>GSCollect.</th>
<th>Guava</th>
<th>HPPC</th>
<th>Koloboke</th>
<th>Trove</th>
</tr>
</thead>
<tbody>
<tr>
<td>populate</td>
<td>-3</td>
<td>-7</td>
<td>-9</td>
<td>-8</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>iterate</td>
<td>-7</td>
<td>-9</td>
<td>-7</td>
<td>-8</td>
<td>-7</td>
<td>-3</td>
</tr>
<tr>
<td>contains</td>
<td>-6</td>
<td>-4</td>
<td>-4</td>
<td>-2</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>remove</td>
<td>-3</td>
<td>-3</td>
<td>-11-13-15</td>
<td>-4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>copy</td>
<td>-2</td>
<td>-3</td>
<td>-4</td>
<td>-3</td>
<td>9</td>
<td>11</td>
</tr>
</tbody>
</table>

100 1K 100K 1M 100 1K 100K 1M 100 1K 100K 1M 100 1K 100K 1M
Superior Alternatives: HashMap

- Standard HashMap is a **solid** implementation
  - No superior alternatives on time

<table>
<thead>
<tr>
<th></th>
<th>populate</th>
<th>iterate</th>
<th>contains</th>
<th>remove</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>GS Collec.</td>
<td>-3</td>
<td>-7</td>
<td>-9</td>
<td>-7</td>
<td>-7</td>
</tr>
<tr>
<td>Guava</td>
<td>-7</td>
<td>-9</td>
<td>-7</td>
<td>-8</td>
<td>-7</td>
</tr>
<tr>
<td>HPPC</td>
<td>-6</td>
<td>-4</td>
<td>-4</td>
<td>-2</td>
<td>-3</td>
</tr>
<tr>
<td>Koloboke</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-11</td>
</tr>
<tr>
<td>Trove</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td></td>
<td>-11</td>
<td>-13</td>
<td>-15</td>
<td>-4</td>
<td>-11</td>
</tr>
</tbody>
</table>

- Fastutil provides a superior alternative on memory consumption

**Reduction of Collection Overhead**

Std. HashMap$Node$ object consumes **32 bytes**
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint
Object vs Primitive Collections

- Reducing collection footprint: overhead + element footprint

- Smaller Collection Overhead
- Smaller Element Footprint

Object collection
ArrayList<>

Primitive collection
IntArrayList

Integer (16 bytes)
int (4 bytes)
Object vs Primitive Collections

• Reducing collection footprint: overhead + element footprint

Object collection
ArrayList<>  Primitive collection
IntArrayList

Smaller Collection Overhead
Smaller Element Footprint

Object[]  Integer (16 bytes)

int[]  (4 bytes)
Object vs Primitive Collections

• Reducing collection footprint: overhead + element footprint

Object collection
ArrayList<>

Primitive collection
IntArrayList

Smaller Collection Overhead

Smaller Element Footprint

Collection Footprint Reduction

80%
Superior Alternatives: Primitive Collections

• We found superior alternatives to all three abstraction types
  • Data Type: Integer to int
• Performance varies considerably from distinct libraries for multiple reasons

For instance, ArrayList primitive implementations

<table>
<thead>
<tr>
<th>Libs</th>
<th>populate</th>
<th>iterate</th>
<th>contains</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td>3 3 3</td>
<td></td>
<td>3 3 3</td>
<td></td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3 3 3</td>
<td></td>
<td>3 3 4</td>
<td></td>
</tr>
<tr>
<td>HPPC</td>
<td>3 3 3</td>
<td></td>
<td>3 3 3</td>
<td>-5 -7 -8 -9 -7</td>
</tr>
<tr>
<td>Trove</td>
<td>3 3 3</td>
<td></td>
<td></td>
<td>-3 -8 -9 -7 -4</td>
</tr>
</tbody>
</table>
Superior Alternatives: Primitive Collections

- We found superior alternatives to all three abstraction types
  - Data Type: Integer to int
- Performance varies considerably from distinct libraries for multiple reasons

For instance, ArrayList primitive implementations

<table>
<thead>
<tr>
<th>Libs</th>
<th>populate</th>
<th>iterate</th>
<th>contains</th>
<th>copy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td>3 3 3</td>
<td></td>
<td>3 3 3</td>
<td></td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3 3 3</td>
<td></td>
<td>3 3 4</td>
<td></td>
</tr>
<tr>
<td>HPPC</td>
<td>3 3 3</td>
<td></td>
<td>3 3 3</td>
<td>-5 -7 -8 -9 -7</td>
</tr>
<tr>
<td>Trove</td>
<td>3 3 3</td>
<td></td>
<td></td>
<td>-3 -8 -9 -7 -4</td>
</tr>
</tbody>
</table>

5x slower than for(IntProcedure) commonly implemented
Superior Alternatives: Primitive Collections

- We found superior alternatives to all three abstraction types
  - Data Type: Integer to int
- Performance varies considerably from distinct libraries for multiple reasons

For instance, ArrayList primitive implementations

<table>
<thead>
<tr>
<th>Libs</th>
<th>populate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fastutil</td>
<td>3 3 3</td>
</tr>
<tr>
<td>GSCollec.</td>
<td>3 3 3</td>
</tr>
<tr>
<td>HPPC</td>
<td>3 3 3</td>
</tr>
<tr>
<td>Trove</td>
<td>3 3 3</td>
</tr>
</tbody>
</table>

```
for (int = offset; i-- > 0; )
```

30% more branch misses
Superior Alternative: Primitive Collections

- GSCollections, Koloboke and Fastutil provide solid superior alternatives

<table>
<thead>
<tr>
<th>Collection</th>
<th>Footprint Reduction</th>
<th>Integer → int</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArrayList</td>
<td>80%</td>
<td></td>
</tr>
<tr>
<td>HashMap</td>
<td>76%</td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>84%</td>
<td></td>
</tr>
</tbody>
</table>

Time impact (1M elements):
- ArrayList:
  - +2x populate
  - +4x contains
  - +2x remove
- HashMap:
  - +2x populate
  - -2x iterate
  - +6x/+4x copy
- HashSet:
  - +4x populate
  - +4x/+3x iterate
  - +33x/+7x copy
Summary

• There are performance opportunities on Alternative Collection Frameworks
  • Time/memory improvement with moderate refactor effort

• We provide a Guideline to assist developers on:
  • Identifying superior alternative implementations
  • Which scenarios an alternative could lead to a substantial performance improvement

• CollectionsBench is open-source and available at GitLab
Thank you for your time

Questions?

diego.costa@informatik-uni.heidelberg.de