CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection

Diego Costa, Artur Andrzejak

Heidelberg University

International Symposium on Code Generation and Optimization (CGO 18)
Feb 26\textsuperscript{th} 2018, Vienna Austria
CollectionSwitch: A Framework for Efficient and Dynamic Collection Selection

Diego Costa, Artur Andrzejak

Heidelberg University

Published as:

Paper/slides available at: https://pvs.ifi.uni-heidelberg.de/publications/
Programs = Data Structures + Algorithms

Niklaus Wirth (1984)
Collections

• In Java, the Collections framework provides a reusable set of data structures
  • Widely used and well tested
  • One Interface → multiple implementations

• Developers rarely select/tune their collections [Cha++11]
  • Top-4 most used implementations are selected 95% of the cases [Cos++17]
  • Only 20% of the ArrayList instantiations specify the initial capacity [Cos++17]
Performance Impact of Collections

Inefficient selection of collections as the main cause of runtime bloat

### Execution Time
+17% Improv.
Configuration of one HashMap alloc-site [LS09]

### Memory Usage
+54% Improv.
Use of ArrayMaps instead of HashMaps [OME09]

### Energy Consumption
+38% Improv.
Use of ArrayList instead of LinkedList [Sam++16]

How to better identify and fix such performance inefficiencies?
Motivational Scenario

List<T> myList = new ArrayList<>();
for(T elem : collection) {
    if(!myList.contains(elem)){
        myList.add(elem);
    }
}

List<T> myList = ctx.createList();
for(T elem : collection) {
    if(!myList.contains(elem)){
        myList.add(elem);
    }
}
It is possible to find and switch the collection type to a more suitable variant?

new ArrayList<>()

new HashArrayList<>()
Exemplary Results

• The DaCapo benchmark of Lucene and Avrora
  • Few allocation sites generate millions of collection instances

• **Lucene**: By augmenting 12 allocation sites with our adaptive behavior
  • Reduce execution time by 15%

• **Avrora**: By augmenting 10 allocation sites with our adaptive behavior
  • Reduce peak of memory consumption by 10%
CollectionSwitch

A framework for Dynamic Adaptation of Java Collections

Combines two techniques:

1. Adaptive Allocation-Site
   - Profiles collection instances
   - Searches for a better variant
   - Switches future instantiations to the best variant type

2. Adaptive Collections
   - Instances that switch themselves to the appropriate implementation
Framework Overview

A) How to Enable Adaptive Collections?

B) How to Define the Performance Goals?

C) How to Find a Better Implementation?

Application

Selection Rules

Modify Allocation-sites

Specify Performance Goals

CollectionSwitch

Manage Collections

Run Application

Runtime Profile

Target Machine
A) How to Enable Adaptive Collections

• Using CollectionSwitch in your project

```java
// Original Allocation-Site
List<T> myList = new ArrayList<>();
```

```java
// Using CollectionSwitch
static ListContext ctx = Switch.createContext(ARRAY);
List<T> myList = ctx.createList();
```

 Allows **same-interface** transformations
B) How to define the Performance Goals?

Configurable Selection Rules
- Space and time trade-offs
- Developers input the threshold for selecting a different variant

<table>
<thead>
<tr>
<th>Rule</th>
<th>Improvement</th>
<th>Max Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{time}$</td>
<td>Time -20%</td>
<td>Selects a variant when its predicted time performance is at least 20% better</td>
</tr>
<tr>
<td>$R_{alloc}$</td>
<td>Allocation -20%</td>
<td>Time +125%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Selects a variant when its predicted allocation performance is at least 20% better</td>
</tr>
<tr>
<td></td>
<td></td>
<td>only Array variants! better with max time penalty of 25%</td>
</tr>
</tbody>
</table>

- A variant is selected when it satisfies the rule
  - Criteria satisfied by multiple variants? Select the variant with biggest improvement
C) How to Find a Better Variant?

Application Source-Code

```java
ctx = Switch.createContext(ARRAY);
myList = ctx.createList();
for(;;) {
    myList.contains(obj);
}
myList2 = ctx.createList();
```

Monitoring the Collections Usage

Estimating the Performance of Variants

Adaptive Allocation Context

Performance Models

Array
Monitoring the Collections Usage

```
ctx.createList();

Is monitoring set full?
```

- no
  - Monitor
    - ArrayList
  - Profile
    - size
    - lookup ↑
    - remove...

- yes
  - ArrayList

Monitoring Set

Profile
- size
- lookup ↑
- remove
Monitoring the Collections Usage

ctx.createList();

Is monitoring set full?

no

Monitor

ArrayList

Profile size lookup ↑ remove...

Profile size lookup ↑ remove

Profile size lookup ↑ remove

Profile size lookup ↑ remove

Profile size lookup ↑ remove

Monitoring Set

Are most of the profiles complete?

yes

Analyze Profiles

Clear Monitoring Set

no

Wait for ∆t time

Are most of the profiles complete?

yes

Analyze Profiles

Clear Monitoring Set

no

Monitor

ArrayList
Monitoring the Collections Usage

Parametrized ratio $r$
- $r = 0.3$ too unstable
- $r = 1.0$ too slow
- $r = 0.6$ good compromise

Instances have been assigned for garbage collection

Are most of the profiles complete?
- yes: Analyze Profiles → Discard Profiles
- no: Wait for $\Delta t$ time
Estimating the Performance

We compare variants $V$ performance according to the total cost metric $TC(V)$

$$tc(V) = \sum_{op} N_{op} \times cost_{op,V}(s)$$

$$TC(V) = \sum tc(V)$$

Collection Profile
- Maximum size $s$ of the collection
- Amount of called operations $N_{op}$
Estimating the Performance

• Polynomial function of the collection size $s$

$$\text{cost}_{op,V}(s) = \sum_{k=0}^{d} a_k s^k$$

• We design a series of benchmarks to calculate the coefficients

  • + 30 variants
  • Single Operation Scenario
  • Measurement Variables
    • Execution Time
    • Memory allocation

<table>
<thead>
<tr>
<th>Factor</th>
<th>Levels/Categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>[10, 100, 200, ..., 1M]</td>
</tr>
<tr>
<td>Operations</td>
<td>populate, contains, iterate, middle, remove</td>
</tr>
<tr>
<td>Data Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Data Distribution</td>
<td>Uniform</td>
</tr>
</tbody>
</table>
Estimating the Performance

• Selects a variant $V_{new}$ to replace the current $V_{cur}$ when it satisfies the Performance Rule

\[
\begin{align*}
R_{alloc} & \quad \text{Improvement} \quad \text{Max Penalty} \\
\text{Allocation} = -20\% & \quad \text{Time} = +125\%
\end{align*}
\]

\[
\left( \frac{TC_{alloc}(V_{new})}{TC_{alloc}(V_{cur})} \leq 0.8 \right) \land \left( \frac{TC_{time}(V_{new})}{TC_{time}(V_{cur})} \leq 1.25 \right)
\]
Adaptation on Instance Level

• CollectionSwitch can also switch to an adaptive variant
  • Second level of adaptation

• Adaptive Collections
  • Small sizes: Memory efficient implementation (array)
  • Large sizes: Time efficient implementation (hash)

<table>
<thead>
<tr>
<th>Variant</th>
<th>Transition</th>
<th>Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdaptiveSet</td>
<td>Array -&gt; Hash</td>
<td>40</td>
</tr>
<tr>
<td>AdaptiveMap</td>
<td>Array -&gt; Hash</td>
<td>50</td>
</tr>
</tbody>
</table>

Transition is done by copying the elements
Evaluating the Model I

- Micro-benchmarks
  - Population of the collection
  - 100 searches of a random element

![Graph showing execution time vs collection size with different collection types and improvement rule](image)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{time}$</td>
<td>Time = 20%</td>
</tr>
</tbody>
</table>
Evaluating the Model II

- Micro-benchmarks
  - Population of the collection
  - 100 searches of a random element

<table>
<thead>
<tr>
<th>Rule</th>
<th>Improvement</th>
<th>Max Penalty</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_{time}$</td>
<td>Time = 20%</td>
<td>Time = 20%</td>
</tr>
<tr>
<td>$R_{alloc}$</td>
<td>Allocation = 20%</td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing execution time comparison between CollectionSwitch and HashSet](image1)

![Graph showing memory allocation comparison between CollectionSwitch and HashSet](image2)
Evaluating the Performance Improvement

- DaCapo benchmarks
  - Real applications

<table>
<thead>
<tr>
<th>Bench</th>
<th>Input Size</th>
<th>#Target. Alloc.</th>
<th>Original Run</th>
<th>CollectionSwitch</th>
<th>R_{time}</th>
<th>R_{alloc}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>T(s)</td>
<td>M(MB)</td>
<td>T_1(s)</td>
<td>M_1(MB)</td>
<td>T_2(s)</td>
</tr>
<tr>
<td>avrora</td>
<td>large</td>
<td>7</td>
<td>4.1</td>
<td>4.2</td>
<td>72.1</td>
<td>-</td>
</tr>
<tr>
<td>bloat</td>
<td>large</td>
<td>17</td>
<td>30.3</td>
<td>28.9</td>
<td>96.9</td>
<td>-</td>
</tr>
<tr>
<td>fop</td>
<td>default</td>
<td>15</td>
<td>0.5</td>
<td>0.5</td>
<td>57.0</td>
<td>+7%</td>
</tr>
<tr>
<td>h2</td>
<td>large</td>
<td>10</td>
<td>40.1</td>
<td>38.3</td>
<td>-6%</td>
<td>508.7</td>
</tr>
<tr>
<td>lusearch</td>
<td>large</td>
<td>12</td>
<td>3.6</td>
<td>3.1</td>
<td>-15%</td>
<td>269.4</td>
</tr>
</tbody>
</table>
Evaluating the Overhead

- DaCapo
  - No significant overhead

- Estimation of Variants Performance
  - Below 300 ns

- Memory Overhead
  - Footprint of each Allocation Context ~1 Kb.

\[
R_{imp} \quad \text{Rule} \quad \text{Improvement} \quad \text{Time} = \infty\%
\]
Summary

• Selecting the appropriate collection is critical for designing efficient Java applications

• CollectionSwitch selects collection at runtime through:
  • Adaptive allocation-sites
  • Adaptive collections

• Improvement on execution time and memory of real applications
Thank You!

diego.costa@informatik.uni-heidelberg.de


• [OME09] Ohad Shacham, Martin Vechev, and Eran Yahav. *Chameleon: adaptive selection of collections.* (PLDI '09)

• [Sam++16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. 2016. *Energy profiles of Java collections classes.* (ICSE '16)