Detecting Inefficiently-Used Containers to Avoid Bloat

Guoqing Xu and Atanas Rountev
Department of Computer Science and Engineering
Ohio State University
Container Bloat

• Pervasively used in large Java applications
  – Many types and implementations in JDK libraries
  – User-defined container types

• Container inefficiency becomes a significant source of runtime bloat
  – Many memory leak bugs reported in the Sun Java bug database are caused by misuse of containers [Xu-ICSE’08]
  – Optimizing a misused HashMap reduced the number of created objects by 66% [Xu-PLDI’09]
The Targeted Container Inefficiencies

• Underutilized container – from **jflex**

```java
Vector v = new Vector();
v.addElement(new Interval('
',''));
v.addElement(new Interval('',''));
RegExp1 c = new RegExp1(sym.CCLASS, v);
```

• Overpopulated container – from **muffin**

```java
Map getCookies(Request query){
    StringTokenizer st = ... // decode the query
    while (st.hasMoreTokens()){ ...
        attrs.put(key, value); // populate the map }
    return attrs;  }
Map cookies = getCookies(query);
String con= cookies.get("config");
```
Use Static Analysis for Bloat Detection

- Dynamic analysis for bloat detection
  - Used by all state-of-art tools
  - “from-symptom-to-cause” diagnosis (heuristics based)
  - Dependent of specific runs
  - Many produce false positives
  - The reports may be hard to interpret

- Static information may be helpful to reduce false positives
  - Programmers’ intentions inherent in the code (instead of heuristics)
  - Independent of inputs and runs
Our Approach

• **Step 1**: A base static analysis
  – Automatically extracts container semantics
  – Based on the context-free-language (CFL) reachability formulation of points-to analysis [Sridharan-Bodik-PLDI’06]
  – Abstract container operations into **ADD** and **GET**

• **Step 2**: Dynamic or static analysis comparing frequencies of **ADD** and **GET**
  – For dynamic analysis: instrument and profile
  – For static analysis: *frequency approximation*

• **Step 3**: Inefficiency detection
  – #ADD is *very small* ➔ *underutilized container*
  – #ADD >> #GET ➔ *overpopulated container*
CFL-Reachability Formulation Example

\[
\begin{align*}
&\text{a = new A(); // o}_1 \\
&\text{b = new A(); // o}_2 \\
&\text{c = a;} \\
&\text{id(p)\{ return p;\}} \\
&\text{x = id(a); // call 1} \\
&\text{y = id(b); // call 2} \\
&\text{a.f = new C(); // o}_3 \\
&\text{b.f = new C(); // o}_4 \\
&\text{e = x.f;}
\end{align*}
\]

\[
\begin{align*}
o &\in \text{pts}(v) \text{ if o } \text{ flowsTo } v
\end{align*}
\]
Extracting Container Semantics

For a container object $o_c$,

- **Inside** objects and **element** objects
- An **ADD** is concretized as a *store operation* that
  - Writes an *element* object to a field of an *inside* object
- A **GET** is concretized as a *load operation* that
  - Reads an *element* object from a field of an *inside* object
ReachFrom Relation

- Relation $o \text{ reachFrom } o_c$
  - $o \text{ flowsTo } a$
  - $b.f = a$
  - $b \text{ flowsTo } o_b$
  - $o_b \text{ reachFrom } o_c$

Matched [ ] and ( ) in a reachFrom path

- A reachFrom path automatically guarantees the context- and field-sensitivity
  - Demand-driven
  - Instead of traversing a points-to graph
Detecting Inside and Element Objects

• An **inside object** $o_i$ of a container $o_c$ is such that
  – $o_i \text{ reachFrom } o_c$
  – $o_i$ is created in $o_c$’s class or super class, or other classes specified by the user

• An **element object** $o_e$ of a container $o_c$ is such that
  – $o_e$ is not an inside object
  – $o_e \text{ reachFrom } o_c$
  – All objects except $o_e$ and $o_c$ on this $\text{reachFrom}$ path are inside objects
**addTo** Reachability

- An **addTo** path from an element object $o_e$ to a container object $o_c$
  - $o_e$ flowsTo $a$
  - $b.f = a$ : store achieving **ADD**
  - $b$ flowsTo $o_b$ where $o_b$ is an inside object
  - $o_a$ reachFrom $o_c$

Matched [ ] and { } in an **addTo** path
getFrom Reachability

• A getFrom path from an element object $o_e$ to container object $o_c$
  – $o_e$ flowsTo $a$
  – $a = b.f$ : load achieving GET
  – $b$ flowsTo $o_b$ where $o_b$ is an inside object
  – $o_b$ reachFrom $o_c$

Matched [ ] and ( ) in a getFrom path
Relevant Contexts

• Identify calling contexts to associate each ADD/GET with a particular container object

• The chain of unbalanced method entry/exit edges \( (1 \ (2 \ (2 \ldots\text{ before the store/load (achieving ADD/GET) on each addTo/getFrom path}) \)

• Details can be found in the paper
Execution Frequency Comparison

• Dynamic analysis
  – Instrument the ADD/GET sites and do frequency profiling

• Static analysis: approximation
  – Based on loop nesting information– *inequality graph*
  – Put ADD/GET with relevant contexts onto inequality graph and check the path

• Example
  List l = new List();
  while(*) {
    while(*){ l.add(…); }  
    l.get(…);
  }

  Overpopulated Container
  #ADD >> #GET
Execution Frequency Comparison

• Dynamic analysis
  – Instrument the ADD/GET sites and do frequency profiling

• Static analysis: approximation
  – Based on loop nesting information—inequality graph
  – Put ADD/GET with relevant contexts onto inequality graph and check the path

• Example

```java
while(*){
  List l = new List();
  l.add(…);
  l.add(…);
  l.add(…);
  l.add(…);
}
```

Underutilized Container

# ADD is small
Evaluation

• Implemented based on Soot and the Sridharan-Bodik points-to analysis framework [PLDI’06]

• The static analysis is scalable
  – 21 large apps including eclipse (41k methods, 1623 containers) analyzed

• Low false positive rate
  – For each benchmark, randomly selected 20 container problems for manual check
  – No FP found for 14 programs
  – 1-3 FPs found for the remaining 5 apps
Static Analysis vs Dynamic Analysis

• Static report is easier to understand and verify
  – Optimization solutions can be easily seen for most problems reported by static analysis
  – For most problems reported by dynamic analysis, we could not have solutions
    • Highly dependent on inputs and runs

• Dynamic frequency information can be incorporated for performance tuning
  – Statically analyze hot containers

• Case studies can be found in the paper

16
Conclusions

• The first static analysis targeting bloat detection
• Find container inefficiency based on loop nesting
  – Demand driven and client driven
  – Unsound but has low false positive rate
• Usage
  – Find container problems during development
  – Help performance tuning with the help of dynamic frequency info
• Static analysis to detect other kinds of bloat?
Thank you