Finding Low-Utility Data Structures

Guoqing Xu$^1$, Nick Mitchell$^2$, Matthew Arnold$^2$, Atanas Rountev$^1$, Edith Schonberg$^2$, Gary Sevitsky$^2$

$^1$ Ohio State University
$^2$ IBM T. J. Watson Research Center
The State-of-the-Art

Bloat can manifest itself through many *symptoms*

- temporary objects [Dufour FSE’08, Shankar OOPSLA’08]
- excessive data copies [Xu PLDI’09]
- highly-stale objects [Bond ASPLOS’06, Novak PLDI’09]
- problematic container behaviors [Xu ICSE’08, Shaham PLDI’09, Xu PLDI’10]

... 

*Is there a common way to characterize bloat?*

At the heart of all inefficiencies lie computations, with great *expense*, produce data values that have little *impact* on the forward progress
What’s Common About Bloat

```java
boolean isResourceDir(IResource s) {
    List dirs = computeAllDirs();
    return dirs.contains(s);
}

List computeAllDirs() {
    //find and return all directories under the working dir
    --- from eclipse
}
```

Blame the List object!!
Cost And Benefit

• Absolute cost of a value: total number of instructions executed to produce it
  - \( a = b \) or \( a = b + c \), each has *unit cost*

• Benefit of heap value \( v \):

  \[ v \rightarrow \bigcirc \text{ (output) : benefit is a very large number} \]

  \[ v \rightarrow \text{DEAD END} \text{ (not used) : benefit is 0} \]

  \[ v \xrightarrow{M} v' \text{ : benefit is } M \text{ (amount of work)} \]
Cost Computation

Appears to a problem that can be solved by taint-like flow tracking

\[
\begin{align*}
  b &= 10; & t_b &= 1; \\
a &= b + 3; & t_a &= t_b + 1; \\
c &= b \times 5; & t_c &= t_b + 1; \\
d &= a / c; & t_d &= t_a + t_c + 1;
\end{align*}
\]

Double counted

The cost of \(d\) is 4

It is a *backward dynamic flow* problem

Requires dynamic slicing in general

\[t_d \text{ is 5?}\]
Abstract Dynamic Thin Slicing

**Thin slicing** [Sridharan-PLDI’07]

\[
b = \ldots; \quad b.m = \ldots;
\]

\[
a = b.m;
\]

- Trace required by dynamic (regular and thin) slicing is *unbounded*
  - The entire execution needs to be recorded
  - Many more details than a client would need

- For some problems, equivalence classes exist

<table>
<thead>
<tr>
<th>E1</th>
<th>a = b.m\textsuperscript{1}</th>
<th>a = b.m\textsuperscript{3}</th>
<th>a = b.m\textsuperscript{67}</th>
<th>a = b.m\textsuperscript{23}</th>
<th>a = b.m\textsuperscript{1235}</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>a = b.m\textsuperscript{45}</td>
<td>a = b.m\textsuperscript{217}</td>
<td>a = b.m\textsuperscript{29}</td>
<td>a = b.m\textsuperscript{35}</td>
<td>a = b.m\textsuperscript{9}</td>
</tr>
</tbody>
</table>
Abstract Dynamic Thin Slicing

• Introduce analysis semantics into profiling

Static instruction domain $I$;

Regular dep. graph

Natural numbers domain $N$

Node: $I \times N$

Abstract dep. graph

Bounded abstract domain $D$

Node: $I \times D$

Abstraction function

$f_{instr}(j) : N \rightarrow D$

1, 3, 67, 23, 1235

$f_{a = b.m}(j)$

E1

45, 217, 29, 35, 9

$f_{a = b.m}(j)$

E2

Bounded slicing
Cost Computation

Absolute cost

\[ b = \frac{e}{f} \]

\[ c = \frac{h}{g} \]

\[ a = b + c \]

\#nodes

- Absolute cost is expensive to compute
- It does not make sense to help problem diagnosis
Cost Computation As Abstract Slicing

• $D$ = the set of calling contexts
  – Object-sensitivity is used [Milanova-TOSEM’05]
  – Each element is a chain of receiver object alloc sites

• Further reduce $D$ to a finite bounded domain $D_{\text{cost}}$
  – A set \{0, ..., s – 1\} with fixed size $s$

• $f_{\text{instr}}(j)$ is a three-step function
  – Details can be found in the paper
Relative Abstract Cost

- *Cumulative cost* measures the effort made *from the beginning of the execution* to produce a value.
- It is certain that *the later* a value is produced, *the higher cost* it has.
Abstract benefit: the effort made to transform a value read from a heap loc $I$ to the value written into another heap loc $I'$. 
- The more heap values are produced, the higher the benefit of $I$.
- The more complex the transformation is, the higher the benefit of $I$. 
Case Studies

• sunflow: 9-15% running time reduction
• bloat: 35% running time reduction
• eclipse: 14.5% running time reduction
• derby: 6% running time reduction
• tomcat: 2% running time reduction
• Trade: 2.5% running time reduction

• Summary
  – Computation of data not necessarily used
  – Choices of unnecessary (expensive) operations
  – Certain usage patterns for inner classes are harmful
Overhead

• Implemented in J9 - the IBM commercial JVM
• Space overhead
  – Shadow heap: has the same size as the Java heap
  – Less than 20Mb for the dependence graph
• Time overhead
  – 71x slowdown: track every instruction and perform synchronization when dependence graph is updated
• Reducing overhead
  – Selective profiling
  – Sampling
  – Various static pre-processing techniques
Conclusions

• A run-time analysis that detects bloat by looking for high-cost-low-benefit operations
  – As opposed to symptoms-based bloat detection

• Achieve scalability using abstract slicing
  – A general technique for a variety of backward dynamic flow problem

• Found many optimization opportunities in large real-world Java applications
Thank you