Go with the Flow: Profiling Copies to Find Run-time Bloat

Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Gary Sevitsky

Presented by Wei Fang
February 10, 2015
Bloat Example

- A commercial document management server
  - Inserting a single small document in the database
  - 25000 method invocations and 3000 temporary objects
  - Decodes a cookie to extract only 1 or 2 elements
  - 1000 method invocations and 35 temporary objects
Bloat Example

• A commercial document management server

  – Inserting a single small document in the database
  – 25000 method invocations and 3000 temporary objects
  – Manual tuning to reduce the object creation rate by 66%

  – Decodes a cookie to extract only 1 or 2 elements
  – 1000 method invocations and 35 temporary objects
  – A hand-optimized specialized version invokes 4 invocations and constructs 2 temporary objects

![Diagram of deserialization and tokenization process]
Copying as Indicator of Bloat

- Chain of data flow that carries values from one storage location to another
- Often via temporary objects
- The same data contained in many locations
- Wasted space and operations
- Bloat often stems from excessive work done along data flows
Copy Profiling

- A flat copy profile is the analog of execution time profile, except that it counts copies rather than time.
Copy Profiling

Observation: copy activity is often concentrated in a small number of methods

- In the document management server, the top 50 methods explain 82% of total copies, but only 24% of total running time.
Copy Chain Profiling

• Understanding the chains as a whole

• A copy chain is
  – A sequence of copies that carries a value through two or more heap locations
  – **Node**: a heap location (instance field and static field)
  – **Edge**: a sequence of copies that transfers a value from one heap location to another
  – An edge abstracts away intermediate copies via stack locations

• We augment a chain with
  – A producer node (source of data)
    • A new expression
    • A computation instruction creating new data
  – A consumer node **C** (sink of data)
    • Shows that the data goes to a computation instruction or a native method
class List{
    Object[] data; int count = 0;
    List(){
        data = new Object[1000]; // O3
    }
    void add(Object o){
        data[count++] = o;
    }
    Object get(int i){ return data[i]; }
    List clone(){
        List newL = new List(); // O2
        for(int j = 0; j < count; j++)
            newL.add(get(j));
    }
}

static void main(){
    List l = new List(); // O1
    for(int i = 0; i < 1000; i++)
        l.add(new Integer(i)); // O4
    List l2 = l.clone();
    for(int i = 0; i < 1000; i++) {
        System.out.println(l2.get(i));
    }
}
Copy Graph Profiling

• Nodes
  – Allocation site nodes “O_i”
  – Instance field nodes “O_i.f”
  – Static field nodes “A.f”
  – Consumer node “C”

• Edges annotated with two integers
  – Frequency and the number of copied bytes

• Example
  – Copy chain: O4 ➔ O3.ELEM ➔ O3.ELEM ➔ C
  – Copy graph:
Context Sensitive Copy Graph

• Imprecision from context insensitive copy graph
  – Invalid copy chains may be derived due to nodes merging

• 1-object-sensitive naming scheme
  – An object is named as *its allocation site* + the *allocation site of the receiver object* of the method containing the allocation site

• Example:
Client Analyses to Find Bloat

• Hot chain detector
  – Recover hot copy chains from copy graph based on heuristics

• Clone detector
  – Find pairs of objects \((O_1, O_2)\) with a large volume of copies occurring between the two object sub-graphs reachable from them

• Not assigned to heap (NATH) analysis
  – Find allocation site nodes that do not have outgoing edges
Detect Real World Bloat

- **DaCapo bloat**
  - Data copy profile: 28% instructions executed are copies
  - 50% of all data copies came from a variety of `toString` and `append` methods

- **What we found from hot copy chains**
  - Most of these calls centered around code of the form `Assert.isTrue(cond, "bug: " + node)`
  - The second argument is printed only when `cond` evaluates to true

- **Elimination of these strings resulted in**
  - 65% reduction in objects created
  - 29% - 35% reduction in running time
Detect Real World Bloat (Cond.)

• Eclipse 3.1
  – A large framework-intensive application
  – Performance problems result from the pile-up of wasteful operations in its plugins

• What we found from NATH analysis report
  – Millions of objects were created solely for the purpose of carrying data across one-level method invocations

• 9.3% running time reduction
## Copy Graph Characteristics

- **Memory overhead**
  - less than 27M for DaCapo,
  - 150M for IBM DMS for 1-object-sensitive analysis

- **Running time overhead**
  - On average 37X

### Program Characteristics

<table>
<thead>
<tr>
<th>Program</th>
<th>$T_{orig(s)}$</th>
<th>$N_0$</th>
<th>$E_0$</th>
<th>$M_0(Mb)$</th>
<th>$T_0(s)$ ($\times$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>8.9</td>
<td>12516</td>
<td>56703</td>
<td>503.7</td>
<td>284.2 (31.9)</td>
</tr>
<tr>
<td>bloat</td>
<td>157.5</td>
<td>14058</td>
<td>14471</td>
<td>502.2</td>
<td>9812.2 (62.4)</td>
</tr>
<tr>
<td>chart</td>
<td>32.5</td>
<td>18113</td>
<td>12810</td>
<td>502.5</td>
<td>1053.2 (32.4)</td>
</tr>
<tr>
<td>flop</td>
<td>3.6</td>
<td>12419</td>
<td>7675</td>
<td>501.8</td>
<td>38.2 (10.6)</td>
</tr>
<tr>
<td>pmd</td>
<td>46.6</td>
<td>11289</td>
<td>8418</td>
<td>501.7</td>
<td>1542.4 (33.1)</td>
</tr>
<tr>
<td>jython</td>
<td>74.7</td>
<td>25653</td>
<td>21893</td>
<td>503.2</td>
<td>2826.1 (37.8)</td>
</tr>
<tr>
<td>xalan</td>
<td>64.8</td>
<td>13505</td>
<td>28678</td>
<td>502.6</td>
<td>3030.5 (46.8)</td>
</tr>
<tr>
<td>hsqldb</td>
<td>13.5</td>
<td>12294</td>
<td>9102</td>
<td>501.7</td>
<td>350.0 (25.9)</td>
</tr>
<tr>
<td>luindex</td>
<td>12.1</td>
<td>10154</td>
<td>10227</td>
<td>501.6</td>
<td>583.4 (48.2)</td>
</tr>
<tr>
<td>lusearch</td>
<td>19.2</td>
<td>8390</td>
<td>13849</td>
<td>501.5</td>
<td>662.8 (34.5)</td>
</tr>
<tr>
<td>eclipse</td>
<td>124.7</td>
<td>34074</td>
<td>52957</td>
<td>506.5</td>
<td>4343.8 (34.8)</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>1800*</td>
<td>17146</td>
<td>12637</td>
<td>502.4</td>
<td>1800*</td>
</tr>
<tr>
<td>IBM DMS</td>
<td>1800*</td>
<td>147517</td>
<td>87531</td>
<td>519.6</td>
<td>1800*</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>1-object-sensitive ($c = 4$)</th>
<th>1-object-sensitive ($c = 8$)</th>
<th>1-object-sensitive ($c = 16$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$N_4$</td>
<td>$E_4$</td>
<td>$M_4(Mb)$</td>
</tr>
<tr>
<td>antlr</td>
<td>48556</td>
<td>112907</td>
<td>506.9</td>
</tr>
<tr>
<td>bloat</td>
<td>54960</td>
<td>35678</td>
<td>504.3</td>
</tr>
<tr>
<td>chart</td>
<td>69438</td>
<td>25951</td>
<td>504.6</td>
</tr>
<tr>
<td>flop</td>
<td>47893</td>
<td>11095</td>
<td>503.1</td>
</tr>
<tr>
<td>pmd</td>
<td>43740</td>
<td>15576</td>
<td>503.0</td>
</tr>
<tr>
<td>jython</td>
<td>95493</td>
<td>32256</td>
<td>505.8</td>
</tr>
<tr>
<td>xalan</td>
<td>52485</td>
<td>55367</td>
<td>504.9</td>
</tr>
<tr>
<td>hsqldb</td>
<td>47666</td>
<td>13432</td>
<td>503.0</td>
</tr>
<tr>
<td>luindex</td>
<td>39319</td>
<td>17695</td>
<td>502.8</td>
</tr>
<tr>
<td>lusearch</td>
<td>32354</td>
<td>22163</td>
<td>502.6</td>
</tr>
<tr>
<td>eclipse</td>
<td>131065</td>
<td>124043</td>
<td>512.3</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>66102</td>
<td>23909</td>
<td>503.3</td>
</tr>
<tr>
<td>IBM DMS</td>
<td>193707</td>
<td>180187</td>
<td>533.7</td>
</tr>
<tr>
<td>Program</td>
<td>#Chains</td>
<td>Length</td>
<td>#NATH Sites</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>antlr</td>
<td>250680</td>
<td>2.60</td>
<td>811</td>
</tr>
<tr>
<td>bloat</td>
<td>6955316</td>
<td>4.00</td>
<td>1160</td>
</tr>
<tr>
<td>chart</td>
<td>29490</td>
<td>1.16</td>
<td>1652</td>
</tr>
<tr>
<td>fop</td>
<td>275835</td>
<td>3.36</td>
<td>1282</td>
</tr>
<tr>
<td>pmd</td>
<td>436397</td>
<td>2.96</td>
<td>1062</td>
</tr>
<tr>
<td>jython</td>
<td>6827057</td>
<td>4.00</td>
<td>493</td>
</tr>
<tr>
<td>xalan</td>
<td>93263</td>
<td>2.60</td>
<td>1218</td>
</tr>
<tr>
<td>hsqldb</td>
<td>8595</td>
<td>1.80</td>
<td>828</td>
</tr>
<tr>
<td>luindex</td>
<td>30183</td>
<td>2.24</td>
<td>749</td>
</tr>
<tr>
<td>lusearch</td>
<td>10640</td>
<td>3.8</td>
<td>302</td>
</tr>
<tr>
<td>eclipse</td>
<td>10070910</td>
<td>1.24</td>
<td>3030</td>
</tr>
<tr>
<td>SPECjbb</td>
<td>21468</td>
<td>2.00</td>
<td>575</td>
</tr>
<tr>
<td>IBM DMS</td>
<td>1937646</td>
<td>3.75</td>
<td>4695</td>
</tr>
</tbody>
</table>

**Table 5.** Copy chains and NATH objects. All copy graph paths with length \( \leq 5 \) are traversed to compute hot chains. The columns show the total number of generated chains, the average length of these chains, and the number of NATH allocation sites and NATH run-time objects.
Conclusions

• Copy activity is a good indicator of bloat

• Profiling copies
  – Data copy profiles show performance problems
  – Copy graph profiling helps pinpoint certain performance bottlenecks in an application

• Three client analyses based on copy graph

• Experimental results
  – Problems were found in real world large applications
  – Although incurring significant overhead, the tool works for large scale long-running programs
Thank you!