Persistent Pointer Information

Xiao Xiao, Qirun Zhang, Jinguo Zhou, Charles Zhang

The Hong Kong University of Science & Technology
Motivation

Computing once, using multiple times!

Storing the pointer information in a database!
1. How to store?
2. What to store?
How to store?

- Using general database?

Not enough!

Not domain-aware!
- Storage size is large
How to store?

- **Group 1**: flow-sensitive, POPL ‘11
- **Group 2**: 1-object + 1-heap sensitive + JDK 1.4, ISSTA ‘02
- **Group 3**: 1-callsite + 1-heap sensitive + JDK 1.6, ISSTA ‘11

<table>
<thead>
<tr>
<th>Program</th>
<th>Language</th>
<th>LOC</th>
<th>#Pointers</th>
<th>#Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>samba</td>
<td>C</td>
<td>2112.7K</td>
<td>1004880</td>
<td>237201</td>
</tr>
<tr>
<td>gs</td>
<td>C</td>
<td>1508.1K</td>
<td>711082</td>
<td>150009</td>
</tr>
<tr>
<td>php</td>
<td>C</td>
<td>1312.4K</td>
<td>673156</td>
<td>146760</td>
</tr>
<tr>
<td>postgresQL</td>
<td>C</td>
<td>1189.2K</td>
<td>584774</td>
<td>131886</td>
</tr>
<tr>
<td>antlr</td>
<td>Java</td>
<td>75.4K</td>
<td>302560</td>
<td>76970</td>
</tr>
<tr>
<td>luindex</td>
<td>Java</td>
<td>67.4K</td>
<td>269878</td>
<td>70426</td>
</tr>
<tr>
<td>bloat</td>
<td>Java</td>
<td>188.4K</td>
<td>625056</td>
<td>129471</td>
</tr>
<tr>
<td>chart</td>
<td>Java</td>
<td>375.1K</td>
<td>890971</td>
<td>234811</td>
</tr>
<tr>
<td>batik</td>
<td>Java</td>
<td>404.5K</td>
<td>766238</td>
<td>137488</td>
</tr>
<tr>
<td>sunflow</td>
<td>Java</td>
<td>326.2K</td>
<td>552974</td>
<td>106456</td>
</tr>
<tr>
<td>tomcat</td>
<td>Java</td>
<td>357.5K</td>
<td>657394</td>
<td>103627</td>
</tr>
<tr>
<td>fop</td>
<td>Java</td>
<td>415.1K</td>
<td>1173406</td>
<td>201122</td>
</tr>
</tbody>
</table>

Group 1 Dacapo-2006
Group 2 Dacapo-9.12
How to store?

We compress points-to information by bzip:

<table>
<thead>
<tr>
<th></th>
<th>Compressed Size</th>
<th>Compression Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Samba</td>
<td>3.2 MB</td>
<td>2 min</td>
</tr>
<tr>
<td>gs</td>
<td>45 MB</td>
<td>11 min</td>
</tr>
<tr>
<td>php</td>
<td>141 MB</td>
<td>37 min</td>
</tr>
<tr>
<td>postgresQL</td>
<td>572 MB</td>
<td>117 min</td>
</tr>
<tr>
<td>antlr</td>
<td>9.2 MB</td>
<td>1 min</td>
</tr>
<tr>
<td>luindex</td>
<td>7.9 MB</td>
<td>1 min</td>
</tr>
<tr>
<td>bloat</td>
<td>33 MB</td>
<td>4 min</td>
</tr>
<tr>
<td>chart</td>
<td>380 MB</td>
<td>27 min</td>
</tr>
<tr>
<td>batik</td>
<td>5,300 MB</td>
<td>260 min</td>
</tr>
<tr>
<td>sunflow</td>
<td>2,200 MB</td>
<td>139 min</td>
</tr>
<tr>
<td>tomcat</td>
<td>1,900 MB</td>
<td>146 min</td>
</tr>
<tr>
<td>fop</td>
<td>15,000 MB</td>
<td>353 min</td>
</tr>
</tbody>
</table>

Information format:

\[ p_1 \, o_1 \, o_2 \, o_3 \ldots \]

\[ p_2 \, o_1 \, o_2 \, o_3 \ldots \]

\[ \ldots \]

\[ p_n \, o_1 \, o_2 \, o_3 \ldots \]
What to store?

• Persisting points-to information only:
  – Computing aliasing information on-demand
  – A common strategy for existing tools

Not enough!

Pointed-by and aliasing related queries are not efficient.
What to store?

Information flow analysis:
  – Given $p.f = b$, the value of $b$ can flow to all $a$ iff:
    • $a = q.f$ and $(p, q)$ is an alias pair

Using IsAlias(p, q):
  – Querying if $(p, q)$ is an alias pair

Using ListAliases(p):
  – Computing all the aliased pointers of $p$

Statements $p.f = b$ and $a = q.f$ are also distinguished by program points or contexts.
What to store?

Querying with points-to information:

<table>
<thead>
<tr>
<th></th>
<th>IsAlias (s)</th>
<th>ListAliases (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>samba</td>
<td>103.7</td>
<td>55.3</td>
</tr>
<tr>
<td>gs</td>
<td>146.6</td>
<td>81.1</td>
</tr>
<tr>
<td>php</td>
<td>745.1</td>
<td>350.5</td>
</tr>
<tr>
<td>postgresQL</td>
<td>843.2</td>
<td>365.3</td>
</tr>
<tr>
<td>antlr</td>
<td>35.1</td>
<td>26.7</td>
</tr>
<tr>
<td>luindex</td>
<td>28.7</td>
<td>22</td>
</tr>
<tr>
<td>bloat</td>
<td>134.2</td>
<td>105</td>
</tr>
<tr>
<td>chart</td>
<td>207.2</td>
<td>147.9</td>
</tr>
<tr>
<td>batik</td>
<td>117.6</td>
<td>30.3</td>
</tr>
<tr>
<td>sunflow</td>
<td>68.5</td>
<td>26.5</td>
</tr>
<tr>
<td>tomcat</td>
<td>71.3</td>
<td>29.6</td>
</tr>
<tr>
<td>fop</td>
<td>205.9</td>
<td>57.5</td>
</tr>
</tbody>
</table>

Underlying data structure is bitmap.
What to store?

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListPointsTo(p)</td>
<td>Output the points-to set for pointer p</td>
</tr>
<tr>
<td>ListPointedBy(o)</td>
<td>Output the pointers that point to memory o</td>
</tr>
<tr>
<td>IsAlias(p, q)</td>
<td>Decide if the pointer p is an alias of q</td>
</tr>
<tr>
<td>ListAliases(p)</td>
<td>Output the pointers that are aliased to pointer p</td>
</tr>
</tbody>
</table>

- Efficiently supporting all common queries
  - Points-to information is not enough!
- Storing points-to + pointed-by + alias information!
Solution I: Sparse Bitmap

• Using sparse bitmap:
  – Representing all information by matrix
  – Efficient for manipulating sparse boolean matrix

• Points-to matrix: PM
• Pointed-by matrix: PM^T
• Alias matrix: PM × PM^T
Solution I: Sparse Bitmap

Computing alias matrix is inefficient:

<table>
<thead>
<tr>
<th></th>
<th>Computing Time</th>
<th>Storage Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>97.3 s</td>
<td>1.6 G</td>
</tr>
<tr>
<td>luindex</td>
<td>67.2 s</td>
<td>1.3 G</td>
</tr>
<tr>
<td>bloat</td>
<td>1448.7 s</td>
<td>5.1 G</td>
</tr>
</tbody>
</table>

Try BDD?

BDD is much more compact than bitmap.
Solution II: BDD

• Slow for generating alias matrix:
  – Same variable ordering does not work well for both PM and PMᵀ
  – Cannot terminate in 1 hour

• Slow for querying:

<table>
<thead>
<tr>
<th></th>
<th>Storage Size (PM only)</th>
<th>IsAlias (Bitmap)</th>
<th>IsAlias (BDD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>antlr</td>
<td>45M</td>
<td>28.3s</td>
<td>6752.6s</td>
</tr>
<tr>
<td>luindex</td>
<td>40M</td>
<td>23.7s</td>
<td>5146.2s</td>
</tr>
<tr>
<td>bloat</td>
<td>92M</td>
<td>101.2s</td>
<td>32907s</td>
</tr>
</tbody>
</table>
Solution II: BDD

• BDD is compact because:
  – BDD merges all *equivalent* points-to sets;
  – BDD merges *similar* points-to sets (with shared prefix);

• BDD is NOT query-efficient because:
  – BDD does not support set operations for individual elements
  – IsAlias(p, q): We should first take out the points-to sets of p and q, and intersect them.
Solution II: BDD

• Can we design a data structure that retains:

  – The compactness of BDD;

  – The querying efficiency of sparse bitmap;
Merging Equivalent Sets

• Are there still many equivalent pointers after the points-to analysis?

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>(For all subjects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-equivalent Pointers</td>
<td>18.5%</td>
<td></td>
</tr>
<tr>
<td>Non-equivalent Objects</td>
<td>82.9%</td>
<td></td>
</tr>
</tbody>
</table>
Merging Equivalent Sets

- Compressed points-to matrix:
  - Size can be reduced by 71.5%!

Yes!

<table>
<thead>
<tr>
<th>sambags</th>
<th>php</th>
<th>postgresQL</th>
<th>antlr</th>
<th>luindex</th>
<th>bloat</th>
<th>chart</th>
<th>batik</th>
<th>sunflow</th>
<th>tomcat</th>
<th>fop</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.7</td>
<td>5.3</td>
<td>12.6</td>
<td>16.9</td>
<td>0.8</td>
<td>0.7</td>
<td>4.4</td>
<td>6.5</td>
<td>344.4</td>
<td>228.7</td>
<td>545</td>
</tr>
</tbody>
</table>

Unit: second
Merging Equivalent Sets

• Storage size (MB):

<table>
<thead>
<tr>
<th></th>
<th>samba</th>
<th>gs</th>
<th>php</th>
<th>postgresQL</th>
<th>antlr</th>
<th>luindex</th>
<th>bloat</th>
<th>chart</th>
<th>batik</th>
<th>sunflow</th>
<th>tomcat</th>
<th>fop</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.4</td>
<td>30.1</td>
<td>46.7</td>
<td>54.5</td>
<td>13</td>
<td>11.9</td>
<td>46.6</td>
<td>58.3</td>
<td>172.7</td>
<td>113.4</td>
<td>146.3</td>
<td>255.7</td>
<td></td>
</tr>
</tbody>
</table>

Small enough?

Depends!
1. Aggregation effect can quickly take over;
2. Compressing them with bzip increases decoding time, may not be tolerable for some applications
Merging Similar Sets

• Using similarity property for compression:
  – p is aliased to: a1, a3, a4, a5, a7, a8
  – q is aliased to: a1, a2, a3, a4, a7, a8

• Merging their common aliases:
  – (p, q): a1, a3, a4, a7, a8
  – p: a5
  – q: a2
Merging Similar Sets

• The aliasing relationships are unknown

• Intuitively, pointers with similar points-to sets may have similar aliasing relationships

• Grouping pointers with similar points-to sets instead
Merging Similar Sets

A hypothetical approach:

Internal pairs, such as (b1, b2).

Cross pairs, such as (a1, c2).
Question

Challenges:

• What is the best way of partitioning?

• How to compute and encode cross pairs?
Pestrie Encoding

Solution sketch:

- Partitioning and structuring the pointers in the same group as a tree
- Encoding trees as intervals
- Representing cross pairs as rectangles
Pestrie Encoding

**Step 1: Partitioning and structuring the pointers**

- Pointers that point to the same object can be put into a partition
Pestrie Encoding

Using o1 for partition
Pestrie Encoding

Using o2 for partition
Pestrie Encoding

Using o3 for partition
Pestrie Encoding

• Does the partitioning order of objects matter?

Yes!

*Very similar to BDD, different ordering results in different compression ratio.*

NP-hard problem for finding the best order.

We give a good heuristic in the paper.
Pestrie Encoding

- A running example:
  - Using the order o1, o2, o3, o4, o5 for partition

<table>
<thead>
<tr>
<th>PM</th>
<th>PMᵀ</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
<td>p2</td>
</tr>
<tr>
<td>p3</td>
<td>p3</td>
</tr>
<tr>
<td>p4</td>
<td>p4</td>
</tr>
<tr>
<td>p5</td>
<td>p5</td>
</tr>
<tr>
<td>p6</td>
<td>p6</td>
</tr>
<tr>
<td>p7</td>
<td>p7</td>
</tr>
<tr>
<td>o₁</td>
<td>o₁</td>
</tr>
<tr>
<td>o₂</td>
<td>o₂</td>
</tr>
<tr>
<td>o₃</td>
<td>o₃</td>
</tr>
<tr>
<td>o₄</td>
<td>o₄</td>
</tr>
<tr>
<td>o₅</td>
<td>o₅</td>
</tr>
</tbody>
</table>
Pestrie Encoding

- Each shadowed area:
  - Partial Equivalent Set
  - A tree structure

- Red edges
  - Cross edges

- Black edge:
  - Tree edges

- Edge numbers:
  - Ignore them at the moment

Pestrie: A trie variant
Interval Encoding

**Step 2: Encoding tree by interval label**

- Interval label for node $v$: $[I_v, E_v]$
- $I_v$: the pre-order of $v$ in DFS traversal
- $E_v$: the largest pre-order in the sub-tree of $v$
Pestrie Decomposition

Step 3: Encoding cross pairs

(p, q) is a cross pair iff:

– p and q are reachable from $u \rightarrow x$ and $u \rightarrow y$;

Two edges of root $u$

For example, $(p4, p7)$ is an alias
Pestrie Decomposition

• Encoding all cross pairs from the cross edges $u \rightarrow x$ and $u \rightarrow y$

• $S_1$ (Nodes reachable from $u \rightarrow x$): $[I_1, E_1]$

• $S_2$ (Nodes reachable from $u \rightarrow y$): $[I_2, E_2]$

• All cross pairs for $S_1 \times S_2$: $[I_1, E_1, I_2, E_2]$
Generating Persistence file

- Generating persistent file:
  - Including *points-to, pointed-by, alias* information
  - Rectangles are stored as points, lines, and rectangles
Decoding & Querying

• Decoding and constructing query structure:
  – Time: $O((n+R)\log n)$
  – $R$ is #rectangles, $n$ is #pointers

• Querying performance:
  – IsAlias: $O(\log n)$
  – ListAliases: $O(K)$, $K$ is the size of answer set
  – ListPointsTo: $O(K)$
  – ListPointedTo: $O(K)$
Storage Size

Bitmap:
Storing points-to and alias matrices

BDD:
Storing points-to matrix

PesTrie:
- Points-to, pointed-to, and alias matrices
- 10.5X smaller than bitmap
- 17.5X smaller than BDD

Unit: MB

PesTrie
Bitmap
BDD

samba gs php postgresQL antlr luindex bloat chart batik sunflow tomcat fop

26.9 MB

200
150
100
50
0

300
250
200
150
100
50
0

Unit: MB
IsAlias Query

Pestrie is:

1.6X faster than bitmap (with alias information)
2.8X faster than bitmap (on-demand)
ListAliases Query

Pestrie and bitmap (with alias information) are almost equally fast.

Pestrie is **123.6X** faster than bitmap (on-demand)
ListPointsTo Query

Unit: second

- Pestrie
- Bitmap
- BDD
Construction Time

First group: 0-CFA heap sensitivity
Second group: 1-CFA heap sensitivity
Third group: 1-CFA heap sensitivity JDK 1.6

Unit: second
Decoding Time

**Unit: second**

Bitmap decoding:

- Loading points-to and alias matrices;
- Constructing pointed-to matrix.
Querying Memory

Unit: MB

- samba
- gs
- php
- postgresQL
- antlr
- luindex
- bloat
- chart
- batik
- sunflow
- tomcat
- fop

Pestrie

Bitmap
More Details in Our Paper

• Complete construction algorithm;
• Pruning strategies;
• Proofs;
• Querying algorithms;
• Optimization theory;
• Preparing points-to matrix;
• More experiments and explanations.
Summary

• We study the problem of persisting points-to, pointed-to, and aliasing information.

• We design Pestrie persistence scheme:
  – Compact size
  – Fast querying
  – Efficient construction and decoding
Q & A

Thank You
Additional Slides
Partitioning Order

• Recall our aim:
  – Grouping the pointers with similar points-to sets

• Recall the HITS algorithm:
  – High quality hub has links to many authority pages
  – Authority page has many links to high quality hubs

Heuristic ===> Pages with similar authority values point to hubs with similar qualities
Partitioning Order

• Authority pages are analogy of pointers
• Hub pages are analogy of objects

• Using hub values to rank the objects

\[ H_o = \sqrt{\sum_{p \in pt^T(o)} |pt(p)|^2} \]