Flow-sensitive Alias Analysis

**Last time**
- Client-Driven pointer analysis

**Today**
- Demand DFA paper
- Scalable flow-sensitive alias analysis
Recall Previous Flow-Sensitive Solution

**Iterative data-flow analysis**

- We’ve seen how IDFA could be use to compute May Points-to and Must Points-to information
- This solution does not scale—can only analyze C programs with 10’s of thousands of lines of code
The Problem

Conservative propagation
– Need to push data-flow facts across every node
The Problem (cont)

**Conservative propagation**
- The analysis doesn’t know which nodes require pointer information ⇒ must propagate information to all reachable nodes
- We need to store, propagate, and compute transfer functions for all pointer information at all program points

**How large can this information be?**
- For programs with 100K to 1M LOC
  - 100’s of thousands of program points
  - Two points-to graphs per program point (for In and Out sets)
  - Each points-to graph can contains 10’s of thousands of pointers (nodes)
  - Each points-to set can contain 100’s or 1000’s of elements (edges)
Exploiting Sparsity

Traditional solution

- Employ a sparse analysis, propagating information directly from defs to uses.

- What if this code were in a loop?

Catch-22

- We need pointer information to compute the def-use chains that would enable a sparse analysis.
Previous Work

**Dynamically compute def-use information** [Chase et al, ’90, Tok & Lin’06]
- High overhead limits scalability
- Scales to 70K LOC

**Semi-Sparse Analysis** [Hardekopf & Lin ’09]
- Separate pointers into two groups: Top-level and Address-taken
- Top-level variables
  - Addresses are never taken
  - Can easily put these variables into SSA form
- Address-taken variables
  - Use traditional IDFA
  - Scales to 300K LOC
A Better Solution

**Staged Flow-Sensitive Analysis** [Hardekopf & Lin ’11]

Precision

- The precision of the auxiliary analysis impacts performance but not precision— as long as the auxiliary analysis is sound
- Use inclusion-based pointer analysis as auxiliary analysis, since it’s the most accurate of the flow-insensitive analyses
Staged Analysis– Performance Problem

**High overhead**
- The number of def-use chains computed by inclusion-based analysis can result in 100’s of thousands of def-use edges for programs with > 1M LOC

**Optimization**
- Identify **access equivalent** variables, those whose def-use chains are identical
- Collapse their def-use chains
- Reduces number of def-use edges by an order of magnitude
Example: CFG

Auxiliary analysis

\[
\begin{align*}
p & \rightarrow \{a\} \\
q & \rightarrow \{b, c, d, e, f\} \\
v & \rightarrow \{e, f\} \\
r & \rightarrow \{a, b, d\} \\
z & \rightarrow \{a, b, c, d\}
\end{align*}
\]
Example: SSA Form

1. \( *p = w \)
   \[ a_1 = \chi(a_0) \]

2. \( *q = x \)
   \[ b_1 = \chi(b_0); \quad c_1 = \chi(c_0); \quad d_1 = \chi(d_0); \quad e_1 = \chi(e_0); \quad f_1 = \chi(f_0) \]

3. \( *r = y \)
   \[ a_2 = \chi(a_1); \quad b_2 = \chi(b_0); \quad d_2 = \chi(d_0) \]

4. \( u = *v \)
   \[ \mu(e_1); \mu(f_1) \]

5. \( s = *z \)
   \[ \mu(a_2); \mu(b_2); \quad \mu(c_0); \mu(d_2) \]

6. \( a_3 = \phi(a_1, a_2); \quad c_2 = \phi(c_1, c_0); \quad b_3 = \phi(b_1, b_2); \quad d_3 = \phi(d_1, d_2) \)

6. \( t = *z \)
   \[ \mu(a_3); \mu(b_3); \mu(c_2); \mu(d_3) \]

Auxiliary analysis
- \( p \rightarrow \{a\} \)
- \( q \rightarrow \{b, c, d, e, f\} \)
- \( v \rightarrow \{e, f\} \)
- \( r \rightarrow \{a, b, d\} \)
- \( z \rightarrow \{a, b, c, d\} \)

Terminology
- \( \chi(a_0) \) is a May Def
- \( \mu(b_2) \) is a May Use
Example: Def-Use Graph

\[ *p = w \]
\[ *r = y \]
\[ *q = x \]
\[ s = *z \]
\[ t = *z \]
\[ u = *v \]
Example: Optimized Def-Use Graph

Applying the transfer functions

- Use IFDA, but only propagate a variable’s points-to sets across edges whose label contains that variable
- Because def-use graph is an over-approximation, we might propagate information unnecessarily
  - Imprecision in a sound Auxiliary analysis affects performance, not precision
Evaluation

Comparison against state-of-the art

- Staged Flow-Sensitive Analysis (SFS) ➜ New algorithm
- Semi-Sparse Flow-Sensitive Analysis (SSO) ➜ Prior state-of-the-art

Details

- Implemented in LLVM using shared code base
- Both analyses are field-sensitive
- Both use BDDs to store points-to sets
<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Name</th>
<th>Description</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>197.parser</td>
<td>parser</td>
<td>11K</td>
</tr>
<tr>
<td></td>
<td>300.twolf</td>
<td>place and route simulator</td>
<td>20K</td>
</tr>
<tr>
<td></td>
<td>ex</td>
<td>text processor</td>
<td>34K</td>
</tr>
<tr>
<td></td>
<td>255.vortex</td>
<td>object-oriented database</td>
<td>67K</td>
</tr>
<tr>
<td></td>
<td>254.gap</td>
<td>group theory interpreter</td>
<td>71K</td>
</tr>
<tr>
<td></td>
<td>sendmail</td>
<td>email server</td>
<td>74K</td>
</tr>
<tr>
<td></td>
<td>253.perlbmk</td>
<td>PERL interpreter</td>
<td>82K</td>
</tr>
<tr>
<td></td>
<td>nethack</td>
<td>text-based game</td>
<td>167K</td>
</tr>
<tr>
<td></td>
<td>python</td>
<td>interpreter</td>
<td>185K</td>
</tr>
<tr>
<td></td>
<td>176.gcc</td>
<td>C language compiler</td>
<td>222K</td>
</tr>
<tr>
<td></td>
<td>vim</td>
<td>text processor</td>
<td>268K</td>
</tr>
<tr>
<td></td>
<td>pine</td>
<td>e-mail client</td>
<td>342K</td>
</tr>
<tr>
<td></td>
<td>svn</td>
<td>source code control</td>
<td>344K</td>
</tr>
<tr>
<td></td>
<td>ghostscript</td>
<td>Postscript viewer</td>
<td>354K</td>
</tr>
<tr>
<td></td>
<td>gimp</td>
<td>image manipulation tool</td>
<td>877K</td>
</tr>
<tr>
<td></td>
<td>tshark</td>
<td>wireless network analyzer</td>
<td>1,946K</td>
</tr>
</tbody>
</table>
## Results

<table>
<thead>
<tr>
<th>Name</th>
<th>SSO (s)</th>
<th>SFS (s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>197.parser</td>
<td>0.41</td>
<td>0.37</td>
<td>1.11</td>
</tr>
<tr>
<td>300.twolf</td>
<td>0.23</td>
<td>0.41</td>
<td>0.56</td>
</tr>
<tr>
<td>ex</td>
<td>0.35</td>
<td>0.40</td>
<td>0.88</td>
</tr>
<tr>
<td>255.vortex</td>
<td>0.60</td>
<td>0.62</td>
<td>0.97</td>
</tr>
<tr>
<td>254.gap</td>
<td>1.28</td>
<td>1.29</td>
<td>0.99</td>
</tr>
<tr>
<td>sendmail</td>
<td>1.21</td>
<td>1.00</td>
<td>1.21</td>
</tr>
<tr>
<td>253.perlbmk</td>
<td>2.30</td>
<td>1.57</td>
<td>1.46</td>
</tr>
<tr>
<td>nethack</td>
<td>3.16</td>
<td>2.64</td>
<td>1.20</td>
</tr>
<tr>
<td>python</td>
<td>120.16</td>
<td>6.62</td>
<td>18.15</td>
</tr>
<tr>
<td>176.gcc</td>
<td>3.74</td>
<td>3.46</td>
<td>1.08</td>
</tr>
<tr>
<td>vim</td>
<td>61.85</td>
<td>5.53</td>
<td>11.18</td>
</tr>
<tr>
<td>pine</td>
<td>347.53</td>
<td>82.00</td>
<td>4.24</td>
</tr>
<tr>
<td>svn</td>
<td>185.10</td>
<td>10.69</td>
<td>17.32</td>
</tr>
<tr>
<td>ghostscript</td>
<td>OOT</td>
<td>31:56.29</td>
<td>∞</td>
</tr>
<tr>
<td>gimp</td>
<td>OOT</td>
<td>20:22.27</td>
<td>∞</td>
</tr>
<tr>
<td>tshark</td>
<td>OOT</td>
<td>13:48.47</td>
<td>∞</td>
</tr>
</tbody>
</table>

### Conclusions?
Results (cont)

Big picture
- Two orders of magnitude improvement
  2006: 70K LOC
  2011: 2,000K LOC
Related Work

Previous staged pointer analyses

– Auxiliary analysis partitions the program [Kahlon ’08]
– Auxiliary analysis prunes the program [Guyer & Lin ’03, Fink et al ’06]
– Complementary to this solution
Future Work

Stage the Client-Driven Pointer Analysis

– A sparse FICI will be much more scalable than the current implementation
Wild Idea

Staged Flow-Sensitive Pointer Analysis: A family of algorithms

- We can select other Auxiliary analyses
  - Instead of inclusion-based (FICI), consider a FICS analysis
  - Resulting analysis would be more precise than a FSCI analysis
- How scalable?
- How precise?

<table>
<thead>
<tr>
<th>FI</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Steens, Anders</td>
<td>SSO, SFS</td>
</tr>
<tr>
<td>Whaley</td>
<td>Client</td>
</tr>
</tbody>
</table>
The Big Picture

Many dimensions of pointer analysis precision

- Flow-sensitive
- Path-sensitive
- Heap model
- Field-based
- Arrays

- Context-sensitive
- Field-sensitive
- Object-based
- Shape analysis

Language effects

- Different languages have different usage patterns
  - eg. C often passes pointers to functions (why?)
  - Das’ Steensgaard’s analysis with one-level flow [Das ’00]
- Modern languages (Python, Java) add more dynamicism
Cottage Industry

Could churn out endless number of new analyses
– Language $\times$ precision dimension (huge space)
The Problem

**Practical use**

– Pointer analyses are difficult to reuse

– Pointer analyses are difficult to write and debug
  – “Around 20 pointer analyses available in LLVM”

– How much precision do you need?
  – Depends on the client and the program
  – Eg. Cisco’s question: To parallelize IOS, which pointer analysis should we use?

– We need to better understand the impact on clients
  – It’s hard to do this without already having multiple pointer analyses
The Vision

**Turnkey Pointer Analysis**
- We need pointer analysis that is so easy to use that everyone can use it
- Should be client-driven
- Needs to be much more adaptive than Guyer’s Client-Driven analysis, which only looked at two dimensions of precision
- Requires careful study of multiple clients
- If successful, would be a game changer
A Step Towards a Solution

Tunable Pointer Analysis (TPA)
  – Decouples control flow sensitivity from core pointer analysis algorithm

Diagnostic tool:
  – TPA can simulate other pointer analysis algorithms
  – TPA can be used to learn about the precision needs of client analyses
  – TPA can be used to help develop and test new pointer analyses by providing a set of known results
Tunable Pointer Analysis

**Useful pointer analysis:**
- Sufficiently scalable for clients such as model checking and software verification

**Valuable research tool:**
- Guide the research community: What precisions are important?
- Identify new techniques for applying adaptive precision
Next Time

Lectures
– Modern uses of compilers
– Traditional uses of compilers

Projects
– You should have received feedback from me
– Submit next iteration of proposals by Friday April 3rd

Assignments
– Assignment 4 due Friday April 10th