Large Spurious Cycles in Global Static Analyses and Their Algorithmic Mitigation

Hakjoo Oh
pronto@ropas.snu.ac.kr

July 10, 2009
ROSAEC Workshop
Motivation

Airac

• Sparrow's buffer-overrun-analysis engine
• an interval-domain-based abstract interpreter
• our platform for domain-specific value analyzer
  • e.g., domain-specific analyzer for JP Morgan’s CDS software is too slow.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>#Basic-Blocks</th>
<th>Time(s) (^1)</th>
<th>#Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip-1.2.4a</td>
<td>7,327</td>
<td>6,541</td>
<td>4601.23</td>
<td>653,063</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>13,093</td>
<td>9,298</td>
<td>23515.27</td>
<td>1,964,396</td>
</tr>
<tr>
<td>less-290</td>
<td>18,449</td>
<td>7,754</td>
<td>46274.67</td>
<td>3,149,284</td>
</tr>
<tr>
<td>tar-1.13</td>
<td>20,258</td>
<td>10,800</td>
<td>75013.88</td>
<td>4,748,749</td>
</tr>
<tr>
<td>make-3.76.1</td>
<td>27,304</td>
<td>11,061</td>
<td>88221.06</td>
<td>4,613,382</td>
</tr>
</tbody>
</table>

We identify a problem (due to lack of enough context-sensitivity) and present its solution.

\(^1\)Pentium4 3.2GHz, 4GB RAM
Spurious Paths & Context-sensitivity

context-insensitivity

context-sensitivity
Spurious Paths Are Inevitable in Practice

- Exponential (or infinite) number of contexts in programs.
- In static analysis, the number of distinguished contexts must be finite.
  - $k$-limiting ($k$-CFA, $k$-suffix)
- $k$-limiting quickly hits a limit in practice.

\[
\begin{array}{c}
0 \rightarrow 1 \rightarrow n-1 \rightarrow n
\end{array}
\]

Any $k$-limiting context-sensitivity inevitably has spurious paths.
Spurious cycles degrade both precision and speed, because

- they model spurious information flow.
- solving cyclic equations is repeatedly applying the equations in vain until a fixpoint is reached.
Spurious Cycles in Reality

In real programs, a single large spurious cycle spans almost all parts of the program.

<table>
<thead>
<tr>
<th>Program</th>
<th>Procedures in the largest cycle</th>
<th>Basic-blocks in the largest cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell-1.0</td>
<td>24/31 (77%)</td>
<td>751/782 (95%)</td>
</tr>
<tr>
<td>gzip-1.2.4a</td>
<td>100/135 (74%)</td>
<td>5,988/6,271 (95%)</td>
</tr>
<tr>
<td>sed-4.0.8</td>
<td>230/294 (78%)</td>
<td>14,559/14,976 (97%)</td>
</tr>
<tr>
<td>tar-1.13</td>
<td>205/222 (92%)</td>
<td>10,194/10,800 (94%)</td>
</tr>
<tr>
<td>wget-1.9</td>
<td>346/434 (80%)</td>
<td>15,249/16,544 (92%)</td>
</tr>
<tr>
<td>bison-1.875</td>
<td>410/832 (49%)</td>
<td>12,558/18,110 (69%)</td>
</tr>
<tr>
<td>proftpd-1.3.1</td>
<td>940/1,096 (85%)</td>
<td>35,386/41,062 (86%)</td>
</tr>
<tr>
<td>apache-2.2.2</td>
<td>1,364/2,075 (66%)</td>
<td>71,719/95,179 (75%)</td>
</tr>
</tbody>
</table>

$k$-limiting is not much effective to mitigate the problem.
Effects on Analysis

spell-1.0 (total #procs:31)
(2,213 LOC, > 30 repetitions)

sed-4.0.8 (total #procs:294)
(26,807 LOC, > 150 repetitions)
Lots of work that eliminate spurious paths.
  - Especially, in data flow analysis.
  - Functional approaches: elimination, iterative, RHS
  - Full call-strings approaches

But they are limited to restricted classes of analysis problems.

$k$-limiting is one of very few options available for more general analyses problems\(^2\) to mitigate spurious paths.
  - Our technique is a simple extension to it, which improves both speed and precision.

\(^2\)e.g., infinite domain with non-distributive flow functions
Benefits of Our Extension

- $\text{Normal}_k$: the classical $k$-limiting algorithm
- $\text{Normal}_k$/RSS: the $k$-limiting with our extension
Basic Idea

Change the worklist ordering so that multiple returns are unnecessary.

... → (5) → (2) → (3)
- Normal\(_k\) inserts (4) and (6).
- But, the current analysis of \(f\) is not related to (4), but only to (6).
- Exclusive analysis of the callee is key.
Prioritizing a callee over call-sites that invoke the callee

For example,

\[ W = \{ n_h, c_1, c_2, \cdots \} \]

- prioritize \( n_h \) over \( c_1 \) and \( c_2 \), and select one from \( \{ n_h, \cdots \} \)
Normal\textsubscript{k}/RSS Algorithm

\texttt{FixpointIterate} (\mathcal{W}, \mathcal{T}) =
\texttt{ReturnSite} := \emptyset
repeat
\begin{align*}
S & := \{(\text{call}_{f}^{g,r}, \ldots) \in \mathcal{W} \mid (n_h, \ldots) \in \mathcal{W} \land \text{reach}(g, h) \land \neg \text{recursive}(g)\} \\
(n, \delta) & := \text{choose}(\mathcal{W} \setminus S) \\
m & := \hat{F} \ n \ (\mathcal{T}(n)(\delta)) \\
\text{if } n = \text{call}_{f}^{g,r} \land \neg \text{recursive}(g) \text{ then } \text{ReturnSite}(g) := (r, \delta) \\
\text{if } n = \text{exit}_g \land \neg \text{recursive}(g) \text{ then } \\
\quad (r, \delta_r) := \text{ReturnSite}(g) \\
\quad \text{if } m \not\subseteq \mathcal{T}(r)(\delta_r) \\
\quad \mathcal{W} := \mathcal{W} \cup \{(r, \delta_r)\} \\
\quad \mathcal{T}(r)(\delta_r) := \mathcal{T}(r)(\delta_r) \sqcup m \\
\text{else} \\
\text{for all } (n', \delta') \in \mathcal{N}(n, \delta) \text{ do} \\
\qquad \text{if } m \not\subseteq \mathcal{T}(n')(\delta') \\
\qquad \quad \mathcal{W} := \mathcal{W} \cup \{(n', \delta')\} \\
\qquad \quad \mathcal{T}(n')(\delta') := \mathcal{T}(n')(\delta') \sqcup m \\
\text{until } \mathcal{W} = \emptyset
Purely Algorithmic Technique

Because it is purely an algorithmic technique inside the worklist-based fixpoint iteration routine,

- it is orthogonal to the performance gains of context-sensitive abstract semantics.
- the correctness is obvious enough to avoid the burden of a safety proof of otherwise newly designed abstract semantics.
Correctness & Precision

- The result is not a fixpoint of the given equation system, but still a sound approximation of program semantics.
- Our algorithm is always at least as precise as the original worklist algorithm.
  - by removing unnecessary computations (analyzing more means losing precision)
More Precision: Less Widening Points

We no more apply widenings to interprocedural loopheads created by non-recursive procedure calls.

```c
int g = 0;
int f() { g++; }
int main() {
    f();
    f();
}
```

- Normal worklist algorithm computes $g = [0, +\infty]$
- Our algorithm computes $g = [0, 2]$
We compare speed and precision between $\text{Normal}_k$/RSS and $\text{Normal}_k$.

- $\text{Normal}_k$: Sparrow’s underlying worklist algorithm
- $\text{Normal}_k$/RSS: $\text{Normal}_k$ with our algorithmic technique
11 open-source software packages.

<table>
<thead>
<tr>
<th>Program</th>
<th>LOC</th>
<th>#Basic-Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell-1.0</td>
<td>2,213</td>
<td>782</td>
</tr>
<tr>
<td>barcode-0.96</td>
<td>4,460</td>
<td>2,634</td>
</tr>
<tr>
<td>httptunnel-3.3</td>
<td>6,174</td>
<td>2,757</td>
</tr>
<tr>
<td>gzip-1.2.4a</td>
<td>7,327</td>
<td>6,271</td>
</tr>
<tr>
<td>jwhois-3.0.1</td>
<td>9,344</td>
<td>5,147</td>
</tr>
<tr>
<td>parser</td>
<td>10,900</td>
<td>9,298</td>
</tr>
<tr>
<td>bc-1.06</td>
<td>13,093</td>
<td>4,924</td>
</tr>
<tr>
<td>less-290</td>
<td>18,449</td>
<td>7,754</td>
</tr>
<tr>
<td>twolf</td>
<td>19,700</td>
<td>14,610</td>
</tr>
<tr>
<td>tar-1.13</td>
<td>20,258</td>
<td>10,800</td>
</tr>
<tr>
<td>make-3.76.1</td>
<td>27,304</td>
<td>11,061</td>
</tr>
</tbody>
</table>
Comparison of Speed between Normal$_k$ and Normal$_k$/RSS

**Normal$_0$ vs. Normal$_0$/RSS**

**Normal$_1$ vs. Normal$_1$/RSS**

**Normal$_2$ vs. Normal$_2$/RSS**
Comparison of Speed between Normal\(_k\) vs. Normal\(_{k+1}\)/RSS

<table>
<thead>
<tr>
<th>Method</th>
<th>Normal(_0) vs. Normal(_1)/RSS</th>
<th>Normal(_1) vs. Normal(_2)/RSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell</td>
<td>22</td>
<td>35</td>
</tr>
<tr>
<td>barcode</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td><a href="https://normal.com">https://normal.com</a></td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>gzip</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>jwhois</td>
<td>53</td>
<td>67</td>
</tr>
<tr>
<td>bc</td>
<td>50</td>
<td>62</td>
</tr>
<tr>
<td>AVERAGE</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Motivation

Problem

Solution

Experiments
Comparison of Precision between Normal\(_0\) and Normal\(_0/\text{RSS}\)

<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis</th>
<th>#intervals in the joined memory</th>
<th>([c, c])</th>
<th>([c_1, c_2])</th>
<th>([c, +\infty))</th>
<th>((-\infty, +\infty))</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell</td>
<td>Normal(_0)</td>
<td>345</td>
<td>88</td>
<td>33</td>
<td>143</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normal(_0/\text{RSS})</td>
<td>345</td>
<td>89</td>
<td>35</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>barcode</td>
<td>Normal(_0)</td>
<td>2136</td>
<td>588</td>
<td>240</td>
<td>527</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normal(_0/\text{RSS})</td>
<td>2136</td>
<td>589</td>
<td>240</td>
<td>526</td>
<td></td>
</tr>
<tr>
<td>httptunnel</td>
<td>Normal(_0)</td>
<td>1337</td>
<td>342</td>
<td>120</td>
<td>481</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normal(_0/\text{RSS})</td>
<td>1345</td>
<td>342</td>
<td>120</td>
<td>473</td>
<td></td>
</tr>
<tr>
<td>gzip</td>
<td>Normal(_0)</td>
<td>1995</td>
<td>714</td>
<td>255</td>
<td>1214</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normal(_0/\text{RSS})</td>
<td>1995</td>
<td>716</td>
<td>255</td>
<td>1212</td>
<td></td>
</tr>
<tr>
<td>jwhois</td>
<td>Normal(_0)</td>
<td>2740</td>
<td>415</td>
<td>961</td>
<td>1036</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Normal(_0/\text{RSS})</td>
<td>2740</td>
<td>415</td>
<td>961</td>
<td>1036</td>
<td></td>
</tr>
</tbody>
</table>
We suggest the following implementation guideline:

- When Normal$_k$ hits the memory cost limit: then use Normal$_k$/RSS instead.
  - Normal$_k$/RSS is empirically faster than Normal$_k$
  - Normal$_k$/RSS is at least as precise as Normal$_k$
  - Normal$_k$/RSS requires in addition just as many memory entities as the number of procedures.

- When Normal$_k$ hits the time cost limit: then, if memory permits, consider using Normal$_{k+1}$/RSS instead.
  - Normal$_{k+1}$/RSS can be even faster than Normal$_k$
  - Normal$_{k+1}$/RSS is in principle more precise than Normal$_k$
  - it requires in addition just as many entities as the number of procedures than Normal$_{k+1}$