Slicing - 2

• Dynamic Slicing
  – Slicing a particular execution of a program
  – Questions of precision
  – Preprocessing to obtain dependence info versus on-the-fly calculation tradeoffs
  – Using slicing to find bugs

H. Agrawal, J. Horgan, “Dynamic Program Slicing”, PLDI’90
X. Zhang, R. Gupta, Y. Zhang, “Precise Dynamic Slicing Algorithms”, ICSE’03

Definition

• Given an execution history his of a program P for a test t and a variable var, the
dynamic slice of P wrt <var,his> is the set of all statements s∈his whose execution had
some effect on the value of var as observed
at the end of execution -- Agrawal & Horgan, PLDI’90
  – Similar meaning to static slice but are working with a
    trace from program execution
Dynamic Slicing

1. read(n);
2. k := 1;
3. while k<= n do
   {4.  if (k mod 2 = 0) 5.  x := 17;
4.     else x := 18;
4.     k := k + 1;
6. }
7. write (x);//**

Original Program

Gather dependences on a specific execution. Slicing criterion specifies input and distinguishes statement instances in trace. e.g., (n = 2, x)
Trace will be:
{1,2,3,4,6,7,3,2,3,8}
Slice will be statements{1,2,3,4,5,7,8}
because stmt 7 was the last stmt to define x before exiting the loop.

Dynamic Slicing Approaches

• Agrawal and Horgan present 4 possible algorithms
  • Alg 1: Project PDG on nodes (stmts) seen in the program execution; Do static slicing algorithm on this projection
  • Alg 2: Mark PDG edges with data dependences during program execution; Traverse graph only on marked edges
  • Alg 3: Create separate node for each run-time stmt occurrence s, with outgoing dependence edges ONLY to those statement occurrences on which s is dependent
  • Alg 4: Do algorithm 3, but reuse nodes if their transitive dependences are the same
    – Algms 1+2 are imprecise; Algms 3+4 are precise, with 4 requiring less space than 3
Example - Algm 1

{1. read(x);
2. If (x < 0) then
   {3. y := f1(x);
    4. z := g1(x);}
else {5. If (x = 0) then
    {6. y := f2(x);
     7. z := g2();}
else {8. y := f3(x);
    9. z := g3(x);}
}
10. write (y);
11. write (z);
}

Algm 1: Project PDG on nodes (stmts) seen in the program execution; Do static slicing algorithm on this projection

PDG for x=-1;
cd edges dashed;
dd edges solid

Agrawal et. al, PLDI’90

Algm 1 - Imprecision

{1. read(n);
2. z := 0;
3. y := 0;
4. k := 1;
5. while (k <= n) do
   {6. z := f(z,y)
    7. y := g(y);
    8. k := k + 1
   }9. write(z);
}

PDG for n=1

{1,2,3,4,5^1,6^1,7^1,8^1,5^2,9}

Agrawal et. al, PLDI’90
**Algm 2**

Algm 2: Mark PDG edges with data dependences during program execution; Traverse graph only on marked edges

```plaintext
1. read(n);
2. z := 0;
3. y := 0;
4. k := 1;
5. while (k <= n) do
   {6. z := f(z,y);
      7. y := g(y);
      8. k := k + 1
   }9. write(z);
```

**PDG for n=1**

```
1, 2, 3, 4, 5, 6, 7, 8, 9
```

---

**Algm 3**

Algm 3: Create separate node for each run-time stmt occurrence s, with outgoing dependence edges ONLY to those statement occurrences on which s is dependent

```plaintext
1. read(n);
2. z := 0;
3. y := 0;
4. k := 1;
5. while (k <= n) do
   {6. z := f(z,y);
      7. If (k != n)
         8. y := g(y);
      else 9. y := 2*n;
      10. k := k + 1
   }11. write(z);
```

```
n=2
1, 2, 3, 4, 5, 6, 7, 8, 9, 11
```
Preprocessing Dependences

• Idea: Save space over Algm 4 by associating dependence tags with edges to identify the instance # of a statement in a dependence
  • Data dependence between 1st occurrence of 2 and 2nd occurrence of 3 is denoted (2¹, 3²) on edge (2,3)
  • Split algorithm into
    – Finding dependences in trace and adding labels to graph
    – Calculate the transitive closure of data and control dependence edges
  • Three precise algorithm versions

Precise Executable Slicing

• Zhang et al, ICSE’03 - paper on precise executable slices
• Full Preprocessing (FP): calculate all data dependences for entire trace, label edges in PDG with stmt instances of dependence

```plaintext
{1. read(n);
  2. z := 0;
  3. y := 0;
  4. k := 1;
  5. while (k <= n) do
    {6. z := f(z,y);
     7. y := g(y);
     8. k := k + 1
     }9. write(z);
}
```
Precise Executable Slicing

- **No Preprocessing (NP):** do all data dependence calculation on-demand
  - FP takes too much storage to do all the labeling
  - Traverse trace backwards to find data dependences; cache results to avoid duplicate traversals (NPwoC, NPwC)

- **Limited Preprocessing (LP):** idea is to divide trace into blocks whose defns are summarized so that the backwards traversal can be optimized by skipping irrelevant blocks
  - Each chunk has a summary of downwards exposed defns
  - Can skip over block if none of variables is of interest

Experiments

- **Comparisons between all three approaches and Algms 1+2 of Agrawal & Horgan**
- **Used C programs and some UNIX benchmarks**
- **Technique**
  - Collected traces on 3 inputs per benchmark
  - Computed 25 slices per trace (from 25 different vars at program end)
  - Computed 25 mid-program slices for first input
Dynamic Slices

Zhang et. al, ICSE’03

<table>
<thead>
<tr>
<th>Program</th>
<th>Static Size</th>
<th>Execution Size</th>
<th>AVG</th>
<th>MIN</th>
<th>MAX</th>
</tr>
</thead>
<tbody>
<tr>
<td>124. gcc @ End</td>
<td>585491</td>
<td>170135</td>
<td>6614</td>
<td>2</td>
<td>11860</td>
</tr>
<tr>
<td>124. gcc @ Midpoint</td>
<td>585491</td>
<td>144506</td>
<td>23258</td>
<td>2</td>
<td>7405</td>
</tr>
<tr>
<td>099 .go @ End</td>
<td>95459</td>
<td>61350</td>
<td>5382</td>
<td>4</td>
<td>8449</td>
</tr>
<tr>
<td>099 .go @ Midpoint</td>
<td>95459</td>
<td>55538</td>
<td>3560</td>
<td>2</td>
<td>6500</td>
</tr>
<tr>
<td>134 .perl @ End</td>
<td>316182</td>
<td>21453</td>
<td>785</td>
<td>2</td>
<td>2208</td>
</tr>
<tr>
<td>134 .perl @ Midpoint</td>
<td>316182</td>
<td>20034</td>
<td>601</td>
<td>2</td>
<td>2208</td>
</tr>
<tr>
<td>130 .ll @ End</td>
<td>31829</td>
<td>10958</td>
<td>834</td>
<td>2</td>
<td>834</td>
</tr>
<tr>
<td>130 .ll @ Midpoint</td>
<td>31829</td>
<td>10429</td>
<td>48</td>
<td>2</td>
<td>584</td>
</tr>
<tr>
<td>068 .exprasec @ End</td>
<td>74029</td>
<td>27733</td>
<td>350</td>
<td>2</td>
<td>1204</td>
</tr>
<tr>
<td>068 .exprasec @ Midpoint</td>
<td>74029</td>
<td>21827</td>
<td>295</td>
<td>2</td>
<td>1114</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Benchmarks with #static instructs, #executed instructs, average size of slice (with min/max sizes)

Cumulative Times of 25 Slices

Zhang et. al, ICSE’03

134.perl @ End — (1)
Comparison with Algm 2

• Compared LP with Algm 2
  • Compared slice sizes but that might be dependent upon choice of slicing criterion
    – Measured on average over programs 4-5.6 times size difference in slices
    – When looked at data slice sizes saw more difference than when examined full slices (with control dependence) which were 1-2 times larger for Algm 2
  • Compared size of dynamic dependence graphs obtained
  • Limited comparison of execution times on small numbers of slices showed LP competitive with Algm 2

Fault location with slicing

• Idea: combine delta debugging with slicing to isolate bug causes
  – Combine delta debugging and chops to find failure-inducing code
    • Delta debugging can find failure-inducing input
  – Use forward dynamic slice from failure-inducing input and backward dynamic slice from faulty output to form a failure-inducing chop -- to locate bug cause
    • Forward dynamic slice finds all stmts which are affected by a specific defining input
    • Chop is code in intersection of forward and backward slice of same program

Failure-inducing Chop

FS is forward dynamic slice from input
BS is backward dynamic slice from output;
combine into chop

Gupta et. al, ASE’05

Q: does the faulty code have to lie within the chop?
Why or why not?

Experiments on Siemens suite

Gupta et. al, ASE’05

Table 1: Overview of benchmark programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Versions</th>
<th>LOC</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>print.tokens</td>
<td>lexical analyzer</td>
<td>3</td>
<td>565</td>
<td>4072</td>
</tr>
<tr>
<td>print.tokens2</td>
<td>lexical analyzer</td>
<td>7</td>
<td>510</td>
<td>4037</td>
</tr>
<tr>
<td>schedule</td>
<td>priority scheduler</td>
<td>6</td>
<td>412</td>
<td>2627</td>
</tr>
<tr>
<td>schedule2</td>
<td>priority scheduler</td>
<td>2</td>
<td>307</td>
<td>2683</td>
</tr>
<tr>
<td>replace</td>
<td>pattern replacement</td>
<td>18</td>
<td>563</td>
<td>5542</td>
</tr>
</tbody>
</table>

Benchmarks
Had to exclude some (e.g., errors of omission, no output)
Results

Gupta et. al, ASE’05

Table 4: Average per benchmark: results of fault location using simplified inputs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Avg. RS</th>
<th>In-RS</th>
<th>Avg. FS</th>
<th>In-FS</th>
<th>Ave. FS</th>
<th>In-FS</th>
<th>F-Chop</th>
<th>F-Chop/FS</th>
<th>F-Chop/ALL</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>62</td>
<td>1</td>
<td>53.5</td>
<td>1</td>
<td>40.6</td>
<td>1</td>
<td>0.65</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>print_tokens?</td>
<td>55</td>
<td>0.43</td>
<td>66.14</td>
<td>1</td>
<td>40</td>
<td>0.43</td>
<td>0.73</td>
<td>0.08</td>
<td></td>
</tr>
<tr>
<td>schedule</td>
<td>77.33</td>
<td>0.67</td>
<td>67</td>
<td>1</td>
<td>42.5</td>
<td>1</td>
<td>0.68</td>
<td>0.12</td>
<td></td>
</tr>
<tr>
<td>schedule2</td>
<td>63.2</td>
<td>1</td>
<td>74</td>
<td>1</td>
<td>42.5</td>
<td>1</td>
<td>0.68</td>
<td>0.14</td>
<td></td>
</tr>
<tr>
<td>replace</td>
<td>74.72</td>
<td>0.93</td>
<td>77.89</td>
<td>1</td>
<td>43.75</td>
<td>1</td>
<td>0.68</td>
<td>0.09</td>
<td></td>
</tr>
</tbody>
</table>

In* column shows fraction of inputs out of total inputs for which faulty statement(s) is contained in structure *
1 means success; value varies from 0-1, over the program versions; Other columns shows average size in stmts over program versions; last 2 columns are ratios of sizes.

How to improve?

X. Zhang, R. Gupta, N. Gupta, “Locating Faults through Automated Predicate Switching”, ICSE’06

- Idea: Simulate changes in program state by sequence of branch outcomes at runtime;
  Using a failing run, find a runtime predicate outcome switch that causes program to succeed
  - Critical predicate
  - Need to look for a predicate evaluation instance to switch because fault may not be in predicate itself, which may evaluation correctly often
- Practical search strategy
Predicate Switching

- Switch only one predicate per run
- Order
  - Last-executed first-switched order
- Prioritization-based order on degree of being influenced by faulty code (works better than LEFS)
- Algorithm
  - Find erroneous output value
  - Rerun program to collect conditional branches; Find predicates for switching using failure-inducing chop, exploring closest predicates first
  - Find critical predicate (stop at first switched predicate causing the program to succeed)

Issues

- What to do about crashes?
  - Pgm executes \( \text{pred} \) and does not crash - OK
  - Pgm executes \( \text{pred} \) and crashes again - CONTINUE
  - Pgm does not execute \( \text{pred} \) - UNCLEAR OUTCOME
- Infinite loops
  - May be caused by a switch - fixed by setting an arbitrary cut-off for \# of instructions
Experiments

- In 15:20 cases found a critical predicate
  - 5 times search failed
  - 11 times was closest predicate

Table 4: Successful/Failed Searches.

<table>
<thead>
<tr>
<th>Program</th>
<th>Found</th>
<th>Where</th>
<th>What</th>
<th>False-Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex 2.5.3</td>
<td>yes</td>
<td>$mem_e @ 1813$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 2.5.1</td>
<td>no</td>
<td>search failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.5.5</td>
<td>no</td>
<td>search failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.5.4</td>
<td>yes</td>
<td>$acc_{3} @ 141$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.5.0</td>
<td>yes</td>
<td>$mem_{1} @ 239$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.5.0</td>
<td>yes</td>
<td>search failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.5.0</td>
<td>yes</td>
<td>$mem_{2} @ 5834$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.4.9</td>
<td>yes</td>
<td>$mem_{3} @ 9102$</td>
<td>193</td>
<td>1</td>
</tr>
<tr>
<td>Hex 1.3.8</td>
<td>yes</td>
<td>$mem_{4} @ 4573$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$acc_{1} @ 956$</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.3</td>
<td>yes</td>
<td>$mem_{5} @ 6583$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.3</td>
<td>yes</td>
<td>$mem_{6} @ 6006$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.2</td>
<td>yes</td>
<td>search failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.1</td>
<td>no</td>
<td>search failed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$mem_{7} @ 9171$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$mem_{8} @ 11833$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$mem_{9} @ 9046$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$mem_{9} @ 2697$</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Hex 1.3.0</td>
<td>yes</td>
<td>$mem_{11} @ 3150$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Zhang et al, ICSE’06

How to locate faulty code?

- Calculate dynamic chop backward from the critical predicate and forward from failure-inducing input
  - Use 2 chops: data-only, full
  - Chops are smaller than slices
  - Where were the faults?
    - 4: in critical predicate
    - 1: in full chop but not in data-only chop
    - 5: in data-only chop (and in full chop)
    - 2: in forward full slice of critical predicate intersected with backward full slice of erroneous input
    - 2: in forward full slice of critical predicate
  - Claim whenever could find critical predicate, then faulty code was in its upwards or downwards slices or chops!
Cause of the fault

• Once know faulty code, need to find its cause
  • Manual search by user
  • Heuristic: examine stmts at smaller dynamic dependence distance from critical predicate first

Table 7: Dependence Distance Based Search.

<table>
<thead>
<tr>
<th>Program</th>
<th>Statements</th>
<th>Dep. Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>s-flex-c4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>s-flex-c5</td>
<td>search failed – error in DP</td>
<td></td>
</tr>
<tr>
<td>s-flex-c6</td>
<td>search failed – error in BP</td>
<td></td>
</tr>
<tr>
<td>s-flex-c7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>s-flex-c8</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>s-flex-c9</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>s-flex-c10</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>s-flex-c11</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Zhang et al, ICSE’06