プログラムスライス

井上克郎
大阪大学/
奈良先端科学技术大学院大学
Empirical Evaluation of the Program Slicing for Fault Localization
Background of Research

- Software Systems are becoming large and complex
  - Debugging, testing, and maintaining costs are increasing
- To reduce development costs, techniques for improving efficiency of such activities are essential
Localization

- Handling large source programs is difficult
- If we could select specific portions in the source programs and we can concentrate our attentions only to those portions, the performance of the activities would increase
Program Slicing

• A technique of extracting all program statements affecting the value of a variable
• Specify a variable concerned and extract the affecting statements
• Developers can concentrate their attentions to the extracted statements

Slicing: Extraction
Slice: Collection of extracted statements
Static Slicing

- All statements possibly affecting the value of \textit{Slice Criterion} (a variable concerned)
- Method
  1. Construct Program Dependence Graph (PDG)
     - Nodes: statements in program
     - Edges:
       - \textit{Data Dependence (DD)}: variable definition and its reference
       - \textit{Control Dependence (CD)}: predicate and statement dominated by the predicate
  2. Collect all reachable nodes on PDG to a slice criterion (statement, variable)
s1 begin
s2 a:=3;
s3 b:=3;
s4 readln(c);
s5 if c=0 then
  s6 begin
    s7 d:=functionA(a);
    e:=d
  end;
else
  begin
    s12 d:=functionB(b);
    e:=d
  end;
end.
write(e)
Example of Static Slice

```
s1 begin
  s2   a:=3;
  s3   b:=3;
  s4   readln(c);
  s5   if c=0 then
  s6     begin
  s7       d:=functionA(a);
  s8       e:=d
  s9     end;
  s10    else
  s11    begin
  s12       d:=functionB(b);
  s13       e:=d
  s14     end;
  s15    writeln(e)
  s16end.
```
Example of Static Slice (2)

s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6    begin
s7       d:=functionA(a);
s8       e:=d
s9    end;
s10   else
s11  begin
s12     d:=functionB(b);
s13     e:=d
s14    end;
s15    writeln(e)
s16end.
Debug Supporting Tool

- **Target language**: subset of Pascal
  - conditional, assignment, iterative, input/output, procedure-call, compound statement etc.
  - variables: integer, string, boolean, and arrays of them.

- **Functions**:
  - Calculate static slice.
  - Step execution, referring the values of variables, setting the breakpoints, etc.
Objective

• We aim to empirically evaluate the potential usefulness of the program slicing to the fault localization.
Process of Experiment

- Step1: To conduct the experiments efficiently, we construct software debugging support tool based on the static slicing.

- Step2: We conduct two experimental projects to evaluate the usefulness of the slicing for fault localization.
  - Experiment 1: (with debugging support tool)
  - Experiment 2:
Experiment 1

Objective

- We empirically evaluate the following two hypotheses:
  - (H1) Using slicing technique reduces the fault localizing effort.
  - (H2) There exists some kinds of faults that are localized effectively.
## Experiment 1 Overview

<table>
<thead>
<tr>
<th></th>
<th>Group1 (Three subjects)</th>
<th>Group2 (Three subjects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial1</td>
<td>PG1/Slicing-based fault localization</td>
<td>PG1/Conventional debugger-based fault localization</td>
</tr>
<tr>
<td>Trial2</td>
<td>PG2/Conventional debugger-based fault localization</td>
<td>PG2/ Slicing-based fault localization</td>
</tr>
</tbody>
</table>
Experiment 1

Programs

• W used two programs (PG1 and PG2) which were developed based on the same specification for the inventory control program at wine shop.

• Since they were developed independently, their data structures and the algorithms were not identical.
Experiment 1
Type of Faults

Faults in PG1
• Lack of the output processing,
• Illegal assignment,
• Illegal conditional statement,
• Omission of the initialization,
• Lack of the procedure call,
• Wrong data renovation,
• Wrong parameter for the procedure call, and
• Wrong execution order for some procedure calls.

Faults in PG2
• Illegal conditional statement,
• Illegal conditional statement,
• Wrong reference to array variable,
• Wrong execution order for some procedure calls,
• Wrong parameter for the procedure call,
• Lack of the procedure call,
• Wrong data renovation,
• Illegal output,
• Wrong registration for database.
Experiment 1  
- Analysis for (H1) -

<table>
<thead>
<tr>
<th></th>
<th>Group1</th>
<th>Group2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trial1</td>
<td>122 (min.)</td>
<td>155 (min.)</td>
</tr>
<tr>
<td></td>
<td>(slice)</td>
<td>(no slice)</td>
</tr>
<tr>
<td>Trial2</td>
<td>133 (min.)</td>
<td>114 (min.)</td>
</tr>
<tr>
<td></td>
<td>(no slice)</td>
<td>(slice)</td>
</tr>
</tbody>
</table>

The group that used the slicing technique could localize the faults effectively.
## Experiment 1
- Analysis for (H2) -

<table>
<thead>
<tr>
<th>Type of fault</th>
<th>Average time to localize</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Trial 1</strong></td>
<td></td>
</tr>
<tr>
<td>Illegal conditional statement</td>
<td>Group1: 14 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group2: 33 (min.)</td>
</tr>
<tr>
<td>Lack of procedure call</td>
<td>Group1: 19 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group2: 34 (min.)</td>
</tr>
<tr>
<td>Wrong data renovation</td>
<td>Group1: 12 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group2: 19 (min.)</td>
</tr>
<tr>
<td><strong>Trial 2</strong></td>
<td></td>
</tr>
<tr>
<td>Illegal conditional statement</td>
<td>Group2: 19 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group1: 34 (min.)</td>
</tr>
<tr>
<td>Wrong registration for database</td>
<td>Group2: 12 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group1: 19 (min.)</td>
</tr>
</tbody>
</table>

This difference is confirmed by the the Welch test ($\alpha=0.05$)

Slicing is effective to localize these faults.
Experiment 2
- Objective -

- In Experiment 1, we could not collect enough subjects to statistically confirm all hypotheses because of its expensiveness.

- To resolve this limitation, we have also carried out an inexpensive experiment, called Experiment 2, which aimed to examine usefulness of the slicing to the fault localization for small scale programs with more subjects and less management effort.
## Experiment 2

### Overview

<table>
<thead>
<tr>
<th>Target</th>
<th>Group1 (15 subjects)</th>
<th>Group2 (19 subjects)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target</td>
<td>Six programs (P1-P6)</td>
<td>Six programs (P1’-P6’)</td>
</tr>
<tr>
<td></td>
<td>Slicing-based fault localization</td>
<td>Conventional debugger-based fault localization</td>
</tr>
</tbody>
</table>

P1-P6: programs with slicing information  
P1’-P6’: only programs
Experiment 2
Programs

• Six kinds of Pascal programs each of which includes one fault (illegal conditional or illegal assignment statement)
  – (P1) Factorization,
  – (P2) Decision whether the input number is a prime number,
  – (P3) Construction of a Triangle of Pascal,
  – (P4) Numerical operations,
  – (P5) Permutation,
  – (P6) Sorting.
Experiment 2  
- Analysis for (H1) -

<table>
<thead>
<tr>
<th></th>
<th>Group1</th>
<th>Group2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average time</td>
<td>40.73 (min.)</td>
<td>49.11 (min.)</td>
</tr>
<tr>
<td></td>
<td>(slice)</td>
<td>(no slice)</td>
</tr>
</tbody>
</table>

This difference is confirmed by the Welch test ($\alpha=0.05$)

The group that used the slicing technique could localize the faults effectively.
Experiment 2  
- Analysis for (H2) -

<table>
<thead>
<tr>
<th>Type of fault</th>
<th>Average time to localize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Illegal conditional statement in P3</td>
<td>Group1: 7.13 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group2: 11.63 min.)</td>
</tr>
<tr>
<td>Illegal conditional statement in P6</td>
<td>Group1: 3.07 (min.)</td>
</tr>
<tr>
<td></td>
<td>Group2: 4.53 (min.)</td>
</tr>
</tbody>
</table>

These faults are included in such programs that it is very difficult to grasp the correspondence its algorithm to its code.
Findings

- We have empirically evaluated the potential usefulness of the program slicing to the fault localization.

- Number of subjects are small. However, we would say that the program slicing is useful for the fault localization.
Lightweight Semi-Dynamic Slicing Methods
Dynamic Slicing

- All statements actually affecting the value of a slice criterion for an execution with a particular input data
- Useful for debugging with testcase
- Method
  1. Execute program with an input data and record the execution trace
  2. Determine DD and CD on each statement of the trace
  3. Collect reachable statements to a slice criterion (input-data, execution-point, variable)
Example of Dynamic Slicing

```
Example of Dynamic Slicing

s1 begin
s2   a:=3;
s3   b:=3;
s4   readln(c);
s5   if c=0 then
s6     begin
s7       d:=functionA(a);
s8       e:=d
s9     end;
s10    else
s11     begin
s12       d:=functionB(b);
s13       e:=d
s14     end;
s15   writeln(e)
s16end.
```

```
e1 begin
e2   a:=3;
e3   b:=3;
e4   readln(c);
e5   if c=0 then
e6     begin
e7       d:=functionA(a);
e8       e:=d
e9     end;
e10    else
e11     begin
e12       d:=functionB(b);
e13       e:=d
e14     end;
e15   writeln(e)
e16end.
```

(1) Execute Trace with Input c=0
Example of Dynamic Slicing (cont.)

```
e1 begin
  e2 a:=3;
  e3 b:=3;
  e4 readln(c);
  e5 if c=0 then
    e6 begin
      e7 d:=functionA(a);
      e8 e:=d
    end;
  else
    begin
      e7 d:=functionB(b);
      e8 e:=d
    end;
  end.
end.
```

(2) Determine DD and CD

Slicing Criterion (c=0, s15, e)

```
s1 begin
  s2 a:=3;
  s3 b:=3;
  s4 readln(c);
  s5 if c=0 then
    s6 begin
      s7 d:=functionA(a);
      s8 e:=d
    end;
  else
    begin
      s7 d:=functionB(b);
      s8 e:=d
    end;
  end.
end.
```

(3) Collect Statements
Static and Dynamic Slicing

- Analysis cost: \textbf{static} < \textbf{dynamic}
  - Recording execution trace is exhaustive
  - Determining DD & CD on execution trace is expensive

\[
\text{PDG} \ll \text{Execution Trace}
\]

- Slice size: \textbf{static} > \textbf{dynamic}
  - Static slicing considers all possible flows
  - Dynamic slicing only considers one trace
Unifying Static and Dynamic Information

Static information + Lightweight dynamic information

Efficient and effective slicing
Approach to Call-Mark (CM) Slicing

• (static slice—unexecuted program statements)

• Unexecuted statements are explored by
  – Checking activation of procedure/function calls
  – Delete unexecuted call statements and associated statements
  – The associated statements: execution dependency (statically determined)
Execution Dependency and CED

• \( s_1 \) is \textit{executionally dependent} (ED) on \( s_2 \) iff \( s_1 \) cannot be executed when \( s_2 \) is not executed
  – Easily obtained by flow analysis

• CED(s) is a set of caller statements on which \( s \) is executionally depending
  – If any of CED(s) is known to be unexecuted, then we know that \( s \) is never executed
  – Also by flow analysis
Example of CED

s1  functionA;

s2  if a=1 then

s3       begin

s4       b:= c;

s5       functionB;

CED(s2) = \{s1\}

CED(s4) = \{s1, s5\}
Steps for Call-Mark Slicing

(1) Construct PDG and compute CED for each statement (pre-execution analysis)

(2) Prepare a flag for each call statement, and execute program with input data. Mark the flag if the call statement is executed

(3) Delete unexecuted nodes and associated edges from PDG

    if any flag in CED(s) is not marked, s is know to be unexecuted

(4) Collect reachable statements to a slice criterion
Example of Call-Mark Slice (Step1)

s1  begin
s2    a:=3;
s3    b:=3;
s4    readln(c);
s5    if c=0 then
s6        begin
s7            d:=functionA(a)
s8            e:=d
s9        end;
s10       else
s11      begin
s12        d:=functionB(b)
s13        e:=d
s14      end;
s15    writeln(e)
s16  end.

CED(s2) = {}  
CED(s7) = {s7}
CED(s8) = {s7}
CED(s12) = {s12}
CED(s13) = {s12}
CED(s15) = { }
Example of Call-Mark Slice (Step2)

```
s1 begin
s2   a:=3;
s3   b:=3;
s4   readln(c);
s5   if c=0 then
s6     begin
s7       d:=functionA(a);  X Flag(s7)
s8       e:=d
s9     end;
  X Flag(s12)
s10   else
s11   begin
s12     d:=functionB(b);
s13     e:=d
s14   end;
  X Flag(s12)
```

Execution with input c=0

Flag(s7) : marked
Flag(s12) : not marked
Example of Call-Mark Slice (Step 3 & 4)

CED(s2) = {}
...
CED(s7) = {s7}
CED(s8) = {s7}
CED(s12) = {s12}
CED(s13) = {s12}
CED(s15) = {}

s1 begin
s2 a:=3;
s3 b:=3;
s4 readln(c);
s5 if c=0 then
s6 begin
s7 d:=functionA(a);
s8 e:=d
s9 end;
s10 else
s11 begin
s12 d:=functionB(b);
s13 e:=d
s14 end;
s15 writeln(e)
s16end.
Implementation of Call-Mark Slicing

• Implement steps (1) - (4)
  – flag \(\leftrightarrow\) each call statement

• Flags are not necessary to be associated with caller codes

• Modify calling mechanism and do not modify other codes
  – Steal the return addresses from the calling stack
  – Determine which caller statements are actually executed
Architecture of Osaka Slicing System
Approach to Dependence-Cache Slicing

- Limitation of static analyses for arrays and pointer variables

1: a[0]:=0;
2: a[1]:=3;
3: readln(b);
4: a[b]:=2;
5: c:=a[0]+4;
6: writeln(c);
Overview of Dependence-Cache Slicing

- Control dependences are analyzed statically
- Data dependences are collected dynamically at program execution
  - Use dependence cache for each variable
- $\text{PDG}_{DC}$ is constructed when program halts
- $\text{PDG}_{DC}$ is traversed from a slice criterion
Dependence Analysis

Input:  b=0

1:  a[0]:=0;
2:  a[1]:=3;
3:  readln(b);
4:  a[b]:=2;
5:  c:=a[0]+4;
6:  writeln(c);

<table>
<thead>
<tr>
<th></th>
<th>a[0]</th>
<th>a[1]</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Evaluation (1)

- Experiments with several sample programs

<table>
<thead>
<tr>
<th>Program</th>
<th>Static</th>
<th>Call-Mark</th>
<th>D-Cache</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1(85lines)</td>
<td>21</td>
<td>17</td>
<td>15</td>
<td>5</td>
</tr>
<tr>
<td>P2(387lines)</td>
<td>182</td>
<td>162</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>P3(871lines)</td>
<td>187</td>
<td>166</td>
<td>61</td>
<td>8</td>
</tr>
</tbody>
</table>
Evaluation (2)

Pre-Execution Analysis Time
\((ms\ by\ Celeron\ 450MHz\ with\ 128MB)\)

<table>
<thead>
<tr>
<th>Program</th>
<th>Static</th>
<th>Call-Mark</th>
<th>D-Cache</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>11</td>
<td>14</td>
<td>5</td>
<td>N/A</td>
</tr>
<tr>
<td>P2</td>
<td>213</td>
<td>215</td>
<td>19</td>
<td>N/A</td>
</tr>
<tr>
<td>P3</td>
<td>710</td>
<td>698</td>
<td>48</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Evaluation (3)

**Execution Time**
*(ms by Celeron 450MHz with 128MB)*

<table>
<thead>
<tr>
<th>Program</th>
<th>Static</th>
<th>Call-Mark</th>
<th>D-Cache</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>47</td>
<td>47</td>
<td>51</td>
<td>174</td>
</tr>
<tr>
<td>P2</td>
<td>43</td>
<td>43</td>
<td>45</td>
<td>4,540</td>
</tr>
<tr>
<td>P3</td>
<td>4,700</td>
<td>4,731</td>
<td>4,834</td>
<td>216,464</td>
</tr>
</tbody>
</table>
Evaluation (4)

Slice Construction Time

(\textit{ms} by Celeron 450MHz with 128MB)

<table>
<thead>
<tr>
<th>Program</th>
<th>Static</th>
<th>Call-Mark</th>
<th>D-Cache</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>0.4</td>
<td>0.6</td>
<td>0.3</td>
<td>76.0</td>
</tr>
<tr>
<td>P2</td>
<td>1.9</td>
<td>1.8</td>
<td>0.7</td>
<td>101.0</td>
</tr>
<tr>
<td>P3</td>
<td>3.0</td>
<td>3.0</td>
<td>1.2</td>
<td>24,969.3</td>
</tr>
</tbody>
</table>
Discussions

• Analysis cost:
  static $\leq$ call-mark $<$ d-cache $<<$ dynamic

Slice size:
  static $>$ call-mark $>$ d-cache $>$ dynamic

• Reasonable slice results with reasonable analysis time

• Promising approach to get effective program localization
Related Works

• Optimized Approaches for Dynamic Slicing (Agrawal & Horgan)
  – Still large execution overhead

• Hybrid Slicing (Guputa & Soffa)
  – Collect all traces between break points and proc. calls
  – Need to specify break points/ Trace can be huge

• Parametric Slicing (Field & Ramalingam)
  – Generalize static and dynamic slicing by symbolic execution with input data subset
  – Practicability and usefulness are unknown
On-Going Works

- Compiler-based lightweight semi-dynamic slicing environment
- Java program analysis
  - Bytecode analyses
  - Alias analysis for Java programs
  - GUI for alias information
課題

・ プログラムスライスの研究動向調査
・ プログラムスライスの原点の論文の要約
・ プログラムスライスの応用に関する調査 田中、岡本
  - 応用分野
  - 商用システム
1: prod := 1;
2: sum := 0;
3: x := 1;
4: while x <= 10 do begin
5: prod := prod * x;
6: sum := sum + x;
7: x := x + 1;
   end;
8: mean := sum / 10;
9: writeln(prod, sum, mean);
1 get(low, high, step, A)
2 min := A[low];
3 max := A[low];
4 sum := A[low];
5 i := low + step;
6 While i <= high do
7     if max < A[i] then
8         min := A[i];
9     end if
10    if min > A[i] then
11       min := A[i];
12     end if
13    sum := sum + A[i];
14    i := i + step;
15 end loop;
16 put (min, max, sum);