Program Slicing

Xiangyu Zhang
What is a slice?

S: ..... = f (v)

- Slice of v at S is the set of statements involved in computing v’s value at S.

Void main ( ) {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    printf ("sum=%d\n",sum);
    printf("I=%d\n",I);
}
Why Slicing

- Debugging
- Testing
- Differencing
- Program understanding
- Software maintenance
- Complexity measurement / Functional Cohesion
- Program integration
- Reverse engineering
- Software Quality Assurance

Old!
What Now

- Security
  - Malware detection;
  - Software piracy
  - ...

- Software Transactional Memory

- Architecture
  - Value speculation

- Program optimization
  - PRE

- Data Lineage

- More to come

A program implement multiple semantic functions. All are not relevant!
Outline

- Slicing ABC
- Dynamic slicing
  - Efficiency
  - Effectiveness
  - Challenges
Slicing Classification

- Static vs. Dynamic
- Backward vs. Forward
- Executable vs. Non-Executable
- More
How to do slicing?

- Static analysis
  - Input insensitive
  - May analysis
- Dependence Graph
- Characteristics
  - Very fast
  - Very imprecise

```python
b = 0
a = 2
for i in range(1, N + 1):
    if ((i++) % 2 == 1):
        a = a + 1
        b = a * 2
    z = a + b
print(z)
```
Why is a static slice imprecise?

- All possible program paths

- Use of Pointers – static alias analysis is very imprecise

- Use of function pointers – hard to know which function is called, conservative expectation results in imprecision
Dynamic Slicing

- Korel and Laski, 1988

- Dynamic slicing makes use of all information about a particular execution of a program and computes the slice based on an execution history (**trace**)
  - Trace consists control flow trace and memory reference trace

- A dynamic slice query is a triple
  - \(<\text{Var}, \text{Input}, \text{Execution Point}>\>

- Smaller, more precise, more helpful to the user
Dynamic Slicing Example

For input N=2,

1: b=0
2: a=2
3: for i=1 to N do
4: if ((i++)%2==1) then
5: a = a+1
   else
6: b = a*2
   endif
done
7: z = a+b
8: print(z)

1_1: b=0
2_1: a=2
3_1: for i = 1 to N do
4_1: if ( (i++) %2 == 1) then
5_1: a=a+1
   endif
3_2: for i=1 to N do
4_2: if ( i%2 == 1) then
5_2: b=a*2
6_1: b=a*2
7_1: z=a+b
8_1: print(z)

[b=0]
[a=2]
[i=1]
[a=3]
[i=2]
[i=2]
[b=6]
[z=9]
[z=9]
Issues about Dynamic Slicing

- Precision — perfect
- Running history — very big (GB)
- Algorithm to compute dynamic slice - slow and very high space requirement.
Backward vs. Forward

1 main( )
2 {
3   int i, sum;
4   sum = 0;
5   i = 1;
6   while(i <= 10)
7       {
8       sum = sum + 1;
9       ++ i;
10      }
11   Cout<< sum;
12   Cout<< i;
13 }

An Example Program & its forward slice w.r.t. <3, sum>
## Executable vs. Non-Executable

| Program Example; begin
| a := 17;
| b := 18;
| P(a, b, c, d);
| write(d) end |
|---|---|---|
| Procedure P(v, w, x, y); x := v; y := w end |

(a) | Program Example; begin
| a := 17;
| b := 18;
| P(a, b, c, d);
| write(d) end |
| Procedure P(v, w, x, y); y := w end |

(b) | Program Example; begin
| b := 18;
| P(a, b, c, d);
| write(d) end |
| Procedure P(v, w, x, y); y := w end |

(c)
Comments

- Want to know more?
  - Frank Tip’s survey paper (1995)

- Static slicing is very useful for static analysis
  - Code transformation, program understanding, etc.
  - Points-to analysis is the key challenge
  - Not as useful in reliability as dynamic slicing

- We will focus on dynamic slicing
  - Precise
    - good for reliability.
  - Solution space is much larger.
  - There exist hybrid techniques.
Outline

- Slicing ABC
- Dynamic slicing
  - Efficiency
  - Effectiveness
  - Challenges
How are dynamic slices computed?

- **Execution traces**
  - control flow trace -- dynamic control dependences
  - memory reference trace -- dynamic data dependences
- **Construct a dynamic dependence graph**
- **Traverse dynamic dependence graph to compute slices**
How to Detect Dynamic Dependence

- Dynamic Data Dependence
  - Shadow space (SS)
    - Addr → Abstract State

\[
\begin{align*}
  s_1_x &: \text{ST } r1, [r2] \iff \text{SS}(r2)=s_1_x \\
  s_2_y &: \text{LD } [r1], r2 \iff s_2_y \rightarrow \text{SS}(r1)=s_1_x
\end{align*}
\]

Dynamic control dependence is more tricky!
Dynamic Dependence Graph Sizes

<table>
<thead>
<tr>
<th>Program</th>
<th>Statements Executed (Millions)</th>
<th>Dynamic Dependence Graph Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300.twolf</td>
<td>140</td>
<td>1,568</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>67</td>
<td>1,296</td>
</tr>
<tr>
<td>255.vortex</td>
<td>108</td>
<td>1,442</td>
</tr>
<tr>
<td>197.parser</td>
<td>123</td>
<td>1,816</td>
</tr>
<tr>
<td>181.mcf</td>
<td>118</td>
<td>1,535</td>
</tr>
<tr>
<td>134.perl</td>
<td>220</td>
<td>1,954</td>
</tr>
<tr>
<td>130.li</td>
<td>124</td>
<td>1,745</td>
</tr>
<tr>
<td>126.gcc</td>
<td>131</td>
<td>1,534</td>
</tr>
<tr>
<td>099.go</td>
<td>138</td>
<td>1,707</td>
</tr>
</tbody>
</table>

- On average, given an execution of 130M instructions, the constructed dependence graph requires 1.5GB space.
[Agrawal & Horgan, 1990] presented three algorithms to trade-off the cost with precision.
Algorithm One

- This algorithm uses a static dependence graph in which all executed nodes are marked dynamically so that during slicing when the graph is traversed, nodes that are not marked are avoided as they cannot be a part of the dynamic slice.

- Limited dynamic information - fast, imprecise (but more precise than static slicing)
Algorithm I Example

For input N=1, the trace is:

1
2
3
4
5
6
7
8
Algorithm I Example

1: b=0
2: a=2
3: 1 <= i <= N
4: if ((i++)%2 = =1)
5: a=a+1
6: b=a*2
7: z=a+b
8: print(z)

DS={1,2,5,7,8}
Precise!
Imprecision introduced by Algorithm I

Input N=2:
1  for (a=1; a<N; a++) {
2   ...
3   if (a % 2 == 1) {
4       b=1;
5   }
6   if (a % 3 == 1) {
7       b= 2* b;
8   } else {
9       c=2*b+1;
10  }
11 }

Killed definition counted as reaching!
Aliasing!
A dependence edge is introduced from a load to a store if during execution, at least once, the value stored by the store is indeed read by the load (mark dependence edge)

No static analysis is needed.
Algorithm II Example

For input N=1, the trace is:

1
2
3
4
5
6
7
8
Algorithm II – Compare to Algorithm I

- More precise

Algo. I

\[
\begin{align*}
x &= \ldots \\
\ldots &= x
\end{align*}
\]

Algo. II

\[
\begin{align*}
x &= \ldots \\
\ldots &= x
\end{align*}
\]
A statically distinct load/store may be executed several times during program execution. Different instances of a load may be dependent on different store instructions or different instances of a store instructions.

- Algo. 2 uses unlabeled edges. Therefore, upon inclusion of the load in the slice it will always include both the stores.
First preprocess the execution trace and introduces labeled dependence edges in the dependence graph. During slicing the instance labels are used to traverse only relevant edges.
## Dynamic Dependence Graph Sizes (revisit)

<table>
<thead>
<tr>
<th>Program</th>
<th>Statements Executed (Millions)</th>
<th>Dynamic Dependence Graph Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300.twolf</td>
<td>140</td>
<td>1,568</td>
</tr>
<tr>
<td>256.bzip2</td>
<td>67</td>
<td>1,296</td>
</tr>
<tr>
<td>255.vortex</td>
<td>108</td>
<td>1,442</td>
</tr>
<tr>
<td>197.parser</td>
<td>123</td>
<td>1,816</td>
</tr>
<tr>
<td>181.mcf</td>
<td>118</td>
<td>1,535</td>
</tr>
<tr>
<td>134.perl</td>
<td>220</td>
<td>1,954</td>
</tr>
<tr>
<td>130.li</td>
<td>124</td>
<td>1,745</td>
</tr>
<tr>
<td>126.gcc</td>
<td>131</td>
<td>1,534</td>
</tr>
<tr>
<td>099.go</td>
<td>138</td>
<td>1,707</td>
</tr>
</tbody>
</table>

- On average, given an execution of 130M instructions, the constructed dependence graph requires 1.5GB space.
```plaintext
N=2:

1: sum=0
2: i=1
3: while ( i<N) do
4: i=i+1
5: sum=sum+i
3: while ( i<N) do
4: i=i+1
5: sum=sum+i
3: while ( i<N) do
6: print (sum)
```
A dynamic dep. edge is represented as by an edge annotated with a pair of timestamps <definition timestamp, use timestamp>
Infer: Local Dependence Labels: Full Elimination

X = Y = X
10, 20, 30

X = Y = X
(10, 10) (20, 20) (30, 30)

X = Y = X
(20, 21)

..., 20
...

..., 20
...

= Y
21
Transform: Local Dependence Labels: Elimination In Presence of Aliasing

\[ X = \]
\[ \ast P = \]
\[ Y = X \]

\[ (10,20) \]

\[ (20,10) \]

\[ (20,20) \]

\[ (20,21) \]

\[ = Y \]

\[ 11,21 \]
Transform: Local Dependence Labels: Elimination In Presence of Aliasing

\[ X = P = Y = X \]
\[ (10,10) \]
\[ (10,11) \]
\[ (20,21) \]

\[ X = P = Y = X \]
\[ (10,20) \]

\[ = Y \]
\[ (10,11) \]
\[ 11,21 \]

\[ X = P = Y = X \]
\[ 10 \]

\[ = Y \]
\[ (10,11) \]
\[ 11,21 \]

\[ X = P = Y = X \]
\[ 20 \]

\[ = Y \]
\[ (20,21) \]
Transform: Coalescing Multiple Nodes into One
**Group:** Labels Across Non-Local Dependence Edges

\[
\begin{align*}
X &= Y = 10 \\
Y &= X = 11, 21 \\
X &= Y = 20 \\
Y &= X = 20, 11 \\
X &= Y = (10, 21) \\
Y &= X = (10, 21) \\
X &= Y = (20, 11) \\
Y &= X = (20, 11)
\end{align*}
\]
# Space: Compacted Graph Sizes

<table>
<thead>
<tr>
<th>Program</th>
<th>Graph Size (MB)</th>
<th>Before</th>
<th>After</th>
<th>Before / After</th>
<th>Explicit Dependences (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>300.twolf</td>
<td>1,568</td>
<td>210</td>
<td>7.72</td>
<td>13.40</td>
<td></td>
</tr>
<tr>
<td>256.bzip2</td>
<td>1,296</td>
<td>51</td>
<td>25.68</td>
<td>3.89</td>
<td></td>
</tr>
<tr>
<td>255.vortex</td>
<td>1,442</td>
<td>65</td>
<td>22.26</td>
<td>4.49</td>
<td></td>
</tr>
<tr>
<td>197.parser</td>
<td>1,816</td>
<td>70</td>
<td>26.03</td>
<td>3.84</td>
<td></td>
</tr>
<tr>
<td>181.mcf</td>
<td>1,535</td>
<td>170</td>
<td>9.02</td>
<td>11.09</td>
<td></td>
</tr>
<tr>
<td>164.gzip</td>
<td>835</td>
<td>52</td>
<td>16.19</td>
<td>6.18</td>
<td></td>
</tr>
<tr>
<td>134.perl</td>
<td>1,954</td>
<td>21</td>
<td>93.40</td>
<td>1.07</td>
<td></td>
</tr>
<tr>
<td>130.li</td>
<td>1,745</td>
<td>97</td>
<td>18.09</td>
<td>5.53</td>
<td></td>
</tr>
<tr>
<td>126.gcc</td>
<td>1,534</td>
<td>75</td>
<td>20.54</td>
<td>4.87</td>
<td></td>
</tr>
<tr>
<td>099.go</td>
<td>1,707</td>
<td>131</td>
<td>13.01</td>
<td>7.69</td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1,543</td>
<td>94</td>
<td>25.2</td>
<td>6.21</td>
<td></td>
</tr>
</tbody>
</table>
Breakdowns of Different Optimizations

Effects of optimizations

- Infer
- Transform
- Group
- Others
- Explicit

Average
Efficiency: Summary

- For an execution of 130M instructions:
  - space requirement: reduced from 1.5GB to 94MB (I further reduced the size by a factor of 5 by designing a generic compression technique [MICRO’05]).
  - time requirement: reduced from >10 Mins to <30 seconds.
Generic Compression

- Traversable in compressed form
  - Sequitur
  - Context-based
    - Using value predictors; (M. Burtsher and M. Jeeradit, PACT2003)

- Bidirectional!!
  - Queries may require going either direction
  - The system should be able to answer multiple queries
Compression using value predictors

- Value predictors
  - Last n values;
  - FCM (finite context method).
    - Example, FCM-3

Uncompressed

X Y Z A

Compressed

1

Left Context lookup table

X Y Z A
Compression using value predictors

- Value predictors
  - Last n values;
  - FCM (finite context method).
    - Example, FCM-3

```
<table>
<thead>
<tr>
<th>Uncompressed</th>
<th>Left Context lookup table</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Y Z</td>
<td></td>
</tr>
<tr>
<td>X Y Z B</td>
<td>X Y Z B</td>
</tr>
</tbody>
</table>
```

Compressed

Length(Compressed) = n/32 + n*(1 - predict rate)

Only forward traversable;

CS590F Software Reliability
Enable bidirectional traversal - idea

Previous predictor compression:

Bidirection idea:
Enable bidirectional traversal

- Forward compressed, backward traversed (uncompressed) FCM
  - Traditional FCM is forward compressed, forward traversed

- Bidirectional FCM
Bidirectional FCM - example

Right Context lookup table

Left Context lookup table
Outline

- Slicing ABC
- Dynamic slicing
  - Dynamic slicing practices
  - Efficiency
  - Effectiveness
  - Challenges
The Real Bugs

- Nine logical bugs
  - Four unix utility programs
    - grep 2.5, grep 2.5.1, flex 2.5.31, make 3.80.

- Six memory bugs [AccMon project (UIUC)]
  - Six unix utility programs
    - gzip, ncompress, polymorph, tar, bc, tidy.
### Classic Dynamic Slicing in Debugging

<table>
<thead>
<tr>
<th>Buggy Runs</th>
<th>LOC</th>
<th>EXEC (%LOC)</th>
<th>BS (%EXEC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex 2.5.31(a)</td>
<td>26754</td>
<td>1871 (6.99%)</td>
<td>695 (37.2%)</td>
</tr>
<tr>
<td>flex 2.5.31(b)</td>
<td>26754</td>
<td>2198 (8.2%)</td>
<td>272 (12.4%)</td>
</tr>
<tr>
<td>flex 2.5.31(c)</td>
<td>26754</td>
<td>2053 (7.7%)</td>
<td>50 (2.4%)</td>
</tr>
<tr>
<td>grep 2.5</td>
<td>8581</td>
<td>1157 (13.5%)</td>
<td>NA</td>
</tr>
<tr>
<td>grep 2.5.1(a)</td>
<td>8587</td>
<td>509 (5.9%)</td>
<td>NA</td>
</tr>
<tr>
<td>grep 2.5.1(b)</td>
<td>8587</td>
<td>1123 (13.1%)</td>
<td>NA</td>
</tr>
<tr>
<td>grep 2.5.1(c)</td>
<td>8587</td>
<td>1338 (15.6%)</td>
<td>NA</td>
</tr>
<tr>
<td>make 3.80(a)</td>
<td>29978</td>
<td>2277 (7.6%)</td>
<td>981 (43.1%)</td>
</tr>
<tr>
<td>make 3.80(b)</td>
<td>29978</td>
<td>2740 (9.1%)</td>
<td>1290 (47.1%)</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>8164</td>
<td>118 (1.5%)</td>
<td>34 (28.8%)</td>
</tr>
<tr>
<td>ncompress-4.2.4</td>
<td>1923</td>
<td>59 (3.1%)</td>
<td>18 (30.5%)</td>
</tr>
<tr>
<td>polymorph-0.4.0</td>
<td>716</td>
<td>45 (6.3%)</td>
<td>21 (46.7%)</td>
</tr>
<tr>
<td>tar 1.13.25</td>
<td>25854</td>
<td>445 (1.7%)</td>
<td>105 (23.6%)</td>
</tr>
<tr>
<td>bc 1.06</td>
<td>8288</td>
<td>636 (7.7%)</td>
<td>204 (32.1%)</td>
</tr>
<tr>
<td>Tidy</td>
<td>31132</td>
<td>1519 (4.9%)</td>
<td>554 (36.5%)</td>
</tr>
</tbody>
</table>

2.4-47.1% EXEC
Avg 30.9%
Looking for Additional Evidence

- Classic dynamic slicing algorithms investigate bugs through negative evidence of the *wrong output*

- Other types of evidence:
  - *Failure inducing input*
  - *Critical Predicate*
  - *Partially correct output*

- Benefits of More Evidence
  - *Narrow the search for fault*
  - *Broaden the applicability*
Negative: Failure Inducing Input [ASE’05]

The rationale
Given a failed run:

- Identify a minimal failure inducing input ([Delta Debugging - Andreas Zeller])
  - This input should affect the root cause.
- Compute forward dynamic slice (FS) of the input identified above

![Diagram of input and output with a shaded area representing the forward dynamic slice (FS).]
Negative: Critical Predicate [ICSE’06]

```c
970    base = ...
     ...
2565   base[...] = ...
     ...
2667   for ( i = 0; i <= lastdfa; ++i )
2668       {
2673       int offset = base[i+1];
2677       chk[offset] = EOB_POSITION;
2681       chk[offset - 1] = ACTION_POSITION;
2683       }
2684
2685   for ( i = 0; i <= tblend; ++i )
2686       {
2690       else if ( chk[i] == ACTION_POSITION )
2696       else /* verify, transition */
2699       }
```

The rationale
Searching Strategies

Execution Trace:

if (P1)
......
if (P2)
......
(3) → if (P3)
......
(2) → if (P4)
......
(1) → if (P5)
......

X output()

Control Flow Distance Ordering

if (P4)
......
(1) → if (P4)
......
(2) → if (P1)
......
(3) → if (P2)

Dependence Distance Ordering

output()
Slicing with Critical Predicate

Given a failed run:

- Identify the critical predicate
  - The critical predicate should **AFFECT / BE AFFECTED BY** the root cause.
- Compute bidirectional slice (BiS) of the critical predicate

![Diagram of slicing with critical predicate]
All Negative Evidence Combined

failure inducing input

Input

Output

BS

FS


BiS(CP)

BS^FS

CP

X

CS590F Software Reliability
<table>
<thead>
<tr>
<th>Buggy Runs</th>
<th>BS</th>
<th>BS^FS^BiS (%BS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>flex 2.5.31(a)</td>
<td>695</td>
<td>27 (3.9%)</td>
</tr>
<tr>
<td>flex 2.5.31(b)</td>
<td>272</td>
<td>102 (37.5%)</td>
</tr>
<tr>
<td>flex 2.5.31(c)</td>
<td>50</td>
<td>5 (10%)</td>
</tr>
<tr>
<td>grep 2.5</td>
<td>NA</td>
<td>86 (7.4%*EXEC)</td>
</tr>
<tr>
<td>grep 2.5.1(a)</td>
<td>NA</td>
<td>25 (4.9%*EXEC)</td>
</tr>
<tr>
<td>grep 2.5.1(b)</td>
<td>NA</td>
<td>599 (53.3%*EXEC)</td>
</tr>
<tr>
<td>grep 2.5.1(c)</td>
<td>NA</td>
<td>12 (0.9%*EXEC)</td>
</tr>
<tr>
<td>make 3.80(a)</td>
<td>981</td>
<td>739 (81.4%)</td>
</tr>
<tr>
<td>make 3.80(b)</td>
<td>1290</td>
<td>1051 (75.3%)</td>
</tr>
<tr>
<td>gzip-1.2.4</td>
<td>34</td>
<td>3 (8.8%)</td>
</tr>
<tr>
<td>ncompress-4.2.4</td>
<td>18</td>
<td>2 (14.3%)</td>
</tr>
<tr>
<td>polymorph-0.4.0</td>
<td>21</td>
<td>3 (14.3%)</td>
</tr>
<tr>
<td>tar 1.13.25</td>
<td>105</td>
<td>45 (42.9%)</td>
</tr>
<tr>
<td>bc 1.06</td>
<td>204</td>
<td>102 (50%)</td>
</tr>
<tr>
<td>tidy</td>
<td>554</td>
<td>161 (29.1%)</td>
</tr>
</tbody>
</table>

Average=36.0% * (BS)
Positive Evidence

- Correct outputs produced in addition to wrong output.
- $\text{BS}(O_{\text{wrong}}) - \text{BS}(O_{\text{correct}})$ is problematic.

...  
10. $A = 1$ (Correct: $A=3$)  
...  
20. $B = A \% 2$  
...  
30. $C = A + 2$  
...  
40. Print $(B)$  
41. Print $(C)$

$\text{BS}(C@41)= \{10, 30, 41\}$
$\text{BS}(B@40)= \{10, 20, 40\}$
$\text{BS}(C@41)-\text{BS}(B@40)$
$= \{30, 41\}$
Assign a confidence value to each node, $C(n) = 1$ means $n$ must contain the correct value, $C(n) = 0$ means there is no evidence of $n$ having the correct value. Given a threshold $t$, BS should only contain the nodes $C(n) < t$.

- If a node $n$ can only reach the correct output, $C(n) = 1$.
- If a node $n$ can only reach the wrong output, $C(n) = 0$.
- If a node $n$ can reach both the correct output and the wrong output, the CONFIDENCE of the node $n$ is defined as:

$$C(n) = 1 - \log |\text{range } (n)| \div |\text{Alt } (n)|$$

- $\text{Alt}(n)$ is a set of possible LHS values at $n$, assigning any of which to $n$ does not change any same correct output.
  - $|\text{Alt}(n)| \geq 1$;
  - $C(n)=1$ when $|\text{Alt}(n)| = 1$. 

CS590F Software Reliability
Confidence Analysis: Example

• If a node $n$ can only reach only the correct output, $C(n) = 1$.
• If a node $n$ can only reach the wrong output, $C(n) = 0$.
• If a node $n$ can reach both the correct output and the wrong output, the CONFIDENCE of the node $n$ is defined as:

$$C(n) = 1 - \log_{|\text{range}(n)|} |\text{Alt}(n)|$$

• $\text{Alt}(n)$ is a set of possible LHS values at $n$, assigning any of which to $n$ produces the same correct output.

\[10. \ A = 1 \ (\text{Correct: } A=3)\]

\[C(10) = 1 - \log_2 \frac{|\text{range}(A)|}{2} = \log_{|\text{range}(A)|} 2\]

\[20. \ B = A \ % \ 2\]

\[C(20) = 1\]

\[30. \ C = A + 2\]

\[C(30) = 0\]

\[40. \ \text{Print (B)}\]

\[C(40) = 1\]

\[41. \ \text{Print (C)}\]

\[C(41) = 0\]
Computing $\text{Alt}(n)$

$\text{C}(S1) = 1 - \log |\text{alt}(S1)| = 1$

$\text{alt}(S1) = \text{alt}(T@S2) \land \text{alt}(T@S3) = \{9\}$

$\text{alt}(T@S2) = \{9\}$

$\text{alt}(T@S3) = \{1, 3, 9\}$

$\text{alt}(S2) = \{10\}$

$\text{C}(S2) = 1 - \log |\text{alt}(S2)| = 1$

$\text{alt}(S3) = \{0, 1\}$

$\text{C}(S3) = \ldots = 1 - \log_3 2$

$(Y, T) = (0, 3), (0, 9), (1, 1), (2, 5), (2, 8)$

$(X, T) = (6, 5), (9, 8), (10, 9)$
Evaluation on injected bugs

- We pruned the slices by removing all the statements with $C(n)=1$

<table>
<thead>
<tr>
<th>Program</th>
<th>BS</th>
<th>Pruned Slice</th>
<th>Pruned Slice / BS</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_tokens</td>
<td>110</td>
<td>35</td>
<td>31.8%</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>114</td>
<td>55</td>
<td>48.2%</td>
</tr>
<tr>
<td>replace</td>
<td>131</td>
<td>60</td>
<td>45.8%</td>
</tr>
<tr>
<td>schedule</td>
<td>117</td>
<td>70</td>
<td>59.8%</td>
</tr>
<tr>
<td>schedule2</td>
<td>90</td>
<td>58</td>
<td>64.4%</td>
</tr>
<tr>
<td>gzip</td>
<td>357</td>
<td>121</td>
<td>33.9%</td>
</tr>
<tr>
<td>flex</td>
<td>727</td>
<td>27</td>
<td>3.7%</td>
</tr>
</tbody>
</table>

Average=41.1%
Effectiveness

- BS = 30.9% * EXEC
- BS^FS^BiS = 36% * BS
  - For many memory type bugs, slices can be reduced to just a few statements.
- Pruned Slice = 41.1% * BS
  - For some benchmarks, the pruned slices contain only the dependence paths leading from the root cause to the wrong output.
Comments

- False positive
  - FS > PS / Chop > DS

- False negative
  - DS > FS=PS=Chop

- Cost
  - PS/Chop > FS > DS
Challenges

- Execution omission errors

  Input x=-1
  y=10
  if (x>0) /*error, should be x<0*/
  y=y+1
  print(y)

- For long running programs, multithreading programs

- Making slices smaller
  - More evidence?
Next

- Background (done)
- Ideas, papers (start from next lecture)
- Will try to schedule a lecture on static tools.
  - Probably in late March.