Slicing - 2

- **Dynamic Slicing**
  - Slicing a particular execution of a program
  - Questions of precision
  - Preprocessing to obtain dependence info versus on-the-fly caculation 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

Tip, JPL'95

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

Original Program

Gather dependences on a specific execution. Slicing criterion specifies input and distinguishes statement instances in trace.
e.g., (n = 2, $**_1$, x)
Trace will be:
$\{1^1,2^1,3^1,4^1,6^1,7^1,3^2,4^2,5^1,7^2,3^3,8^1\}$
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
  - Algm 1: Project PDG on nodes (stmts) seen in the program execution; Do static slicing algorithm on this projection
  - Algm 2: Mark PDG edges with data dependences during program execution; Traverse graph only on marked edges
  - Algm 3: Create separate node for each run-time stmt occurrence s, with outgoing dependence edges ONLY to those statement occurrances on which s is dependent
  - Algm 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

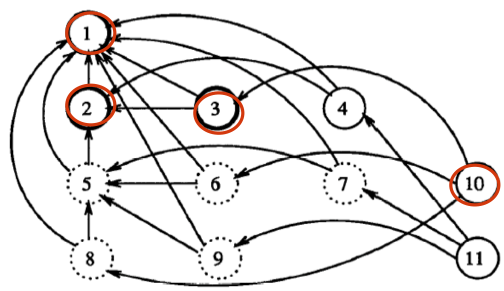
Agrawal et. al, PLDI'90

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

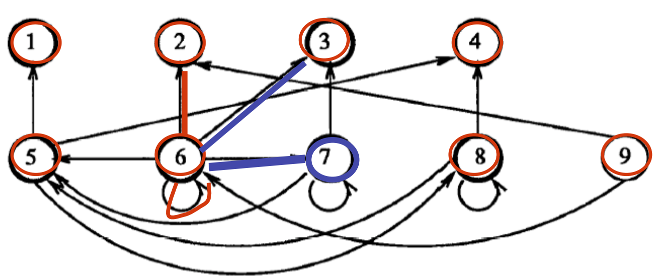


Algm 1 - Imprecision

Agrawal et. al, PLDI'90

```
{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



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

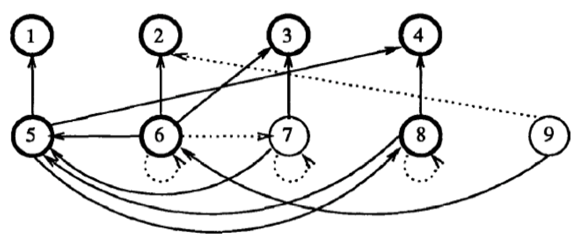
Algm 2

Agrawal et. al, PLDI'90

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

```
{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



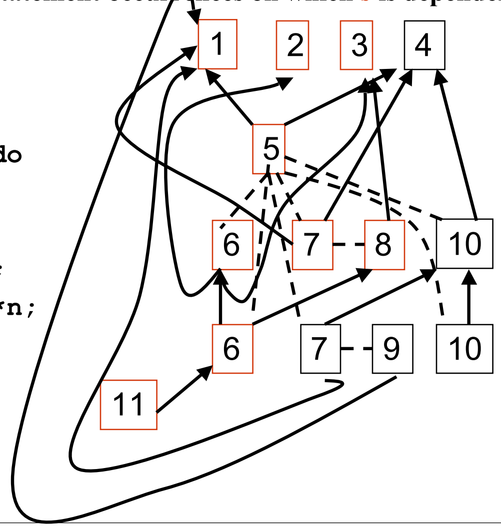
$\{1,2,3,4,5^1,6^1,7^1,8^1, 5^2, 9\}$

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

```
{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



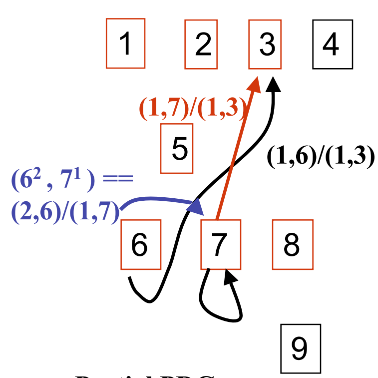
Preprocessing Dependences

- 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 (2^1, 3^2) 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

```
{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);
}
```



Partial PDG

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

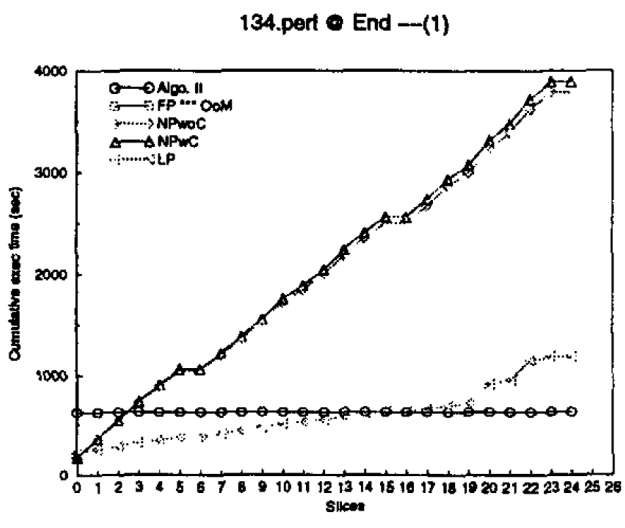
Zhang et. al, ICSE'03

Program	Static	Instructions Executed	PDS		
			AVG	MIN	MAX
126.gcc @ End	585491	170135	6614	2	11860
126.gcc @ Midpoint	585491	144504	2325.8	2	7405
099.go @ End	95459	61350	5382	4	8449
099.go @ Midpoint	95459	55538	3560	2	6900
134.perl @ End	116182	21451	765	2	2208
134.perl @ Midpoint	116182	20934	601	2	2208
130.li @ End	31829	10958	834	2	834
130.li @ Midpoint	31829	10429	48	2	584
008.espresso @ End	74039	27333	350	2	1304
008.espresso @ Midpoint	74039	21027	295	2	1114
Average					

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

Cumulative Times of 25 Slices

Zhang et. al, ICSE'03



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

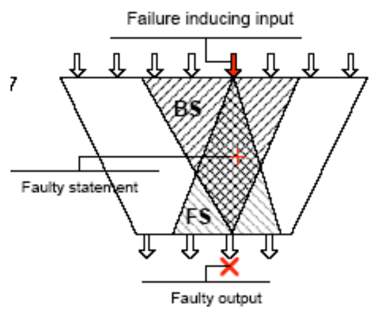
- Idea: combine delta debugging with slicing to isolate bug causes
  - Combine delta debugging and chops to find failure-inducing code
    - Delta debugging can find failure-inducing input
  - Use forward dynamic slice from failure-inducing input and backward dynamic slice from faulty output to form 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

N.Gupta, H. He, X. Zhang, R. Gupta, "Locating Faulty Code Using Failure-inducing Chops", ASE'05

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.

Program	Description	Versions	LOC	Tests
print_tokens	lexical analyzer	5	565	4072
print_tokens2	lexical analyzer	7	510	4057
schedule	priority scheduler	6	412	2627
schedule2	priority scheduler	2	307	2683
replace	pattern replacement	18	563	5542

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.

Program	Avg. BS	In BS	Avg. FS	In FS	Avg. FChop	In FChop	FChop/BS	FChop/ALL
print_tokens	62	1	52.2	1	40.6	1	0.65	0.07
print_tokens2	55	0.43	66.14	1	40	0.43	0.73	0.08
schedule	77.33	0.67	67	1	49.83	0.67	0.64	0.12
schedule2	62.5	1	74	1	42.5	1	0.68	0.14
replace	74.72	0.93	77.89	1	50.78	0.93	0.68	0.09

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 *pred* and does not crash - OK
  - Pgm executes *pred* and crashes again- CONTINUE
  - Pgm does not execute *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**

Zhang et al, ICSE'06

Table 4: Successful/Failed Searches.

Program	Found	Where	Which	False +ves
flex 2.5.319(a)	yes	gen.c @ 1813	0	0
flex 2.5.319(b)	no	search failed		
flex 2.5.319(c)	no	search failed		
grep 2.5	yes	grep.c @ 532	0	0
grep 2.5.1 (a)	yes	search.c @ 549	0	0
grep 2.5.1 (b)	no	search failed		
grep 2.5.1 (c)	yes	dfa.c @ 2854	2	0
make 3.80 (a)	yes	read.c @ 6162	143	1
make 3.80 (b)	yes	remake.c @ 652	1	0
bc-1.06	yes	storage.c @ 176	9	0
tar-1.13.25	yes	prepargs.c @ 81	0	0
tidy	yes	parser.c @ 3496	0	0
s-flex-v4	yes	flex.c @ 2978	0	0
s-flex-v5	no	search failed – error in DP		
s-flex-v6	no	search failed – error in DP		
s-flex-v7	yes	flex.c @ 9171	0	0
s-flex-v8	yes	flex.c @ 11833	0	0
s-flex-v9	yes	flex.c @ 5046	0	0
s-flex-v10	yes	flex.c @ 2687	1	0
s-flex-v11	yes	flex.c @ 3559	0	0

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.

Program	Statements	Dep. Distance
s-flex-v4	1	0
s-flex-v5	search failed – error in DP	
s-flex-v6	search failed – error in DP	
s-flex-v7	2	1
s-flex-v8	2	0
s-flex-v9	1	0
s-flex-v10	3	1
s-flex-v11	3	2

Zhang et al, ICSE'06