SOFT620020.01
Advanced Software Engineering

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io

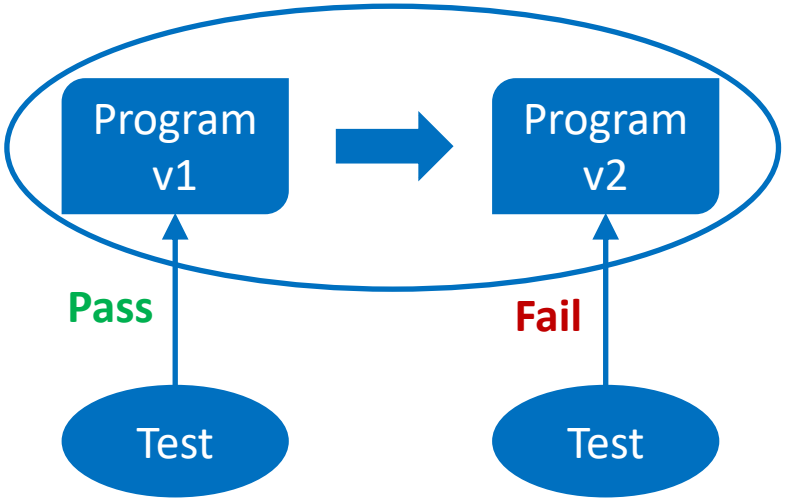
Course Outline

Date	Topic
Sep. 09	Introduction
Sep. 16	Testing Overview
Sep. 23	Guided Random Testing
Sep. 30	Search-Based Testing
Oct. 12	Performance Analysis
Oct. 14	Presentation 1
Oct. 21	Security Testing
Oct. 28	Compiler Testing

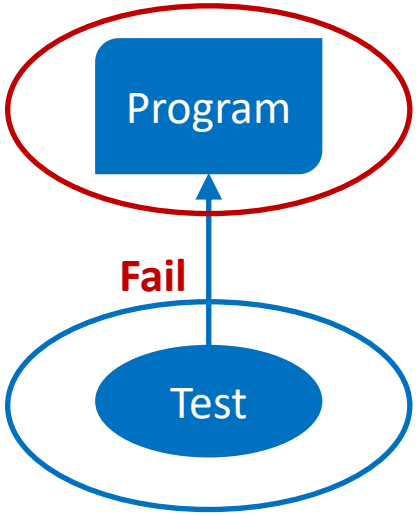
Date	Topic
Nov. 04	Mobile Testing
Nov. 11	Delta Debugging
Nov. 18	**Bug Localization**
Nov. 25	Presentation 2
Dec. 02	Automatic Repair
Dec. 09	Symbolic Execution
Dec. 16	Big Code Analysis
Dec. 23	Presentation 3

Debugging

Locate the buggy code changes that cause the test fail



How to locate the buggy code that cause the test fail?



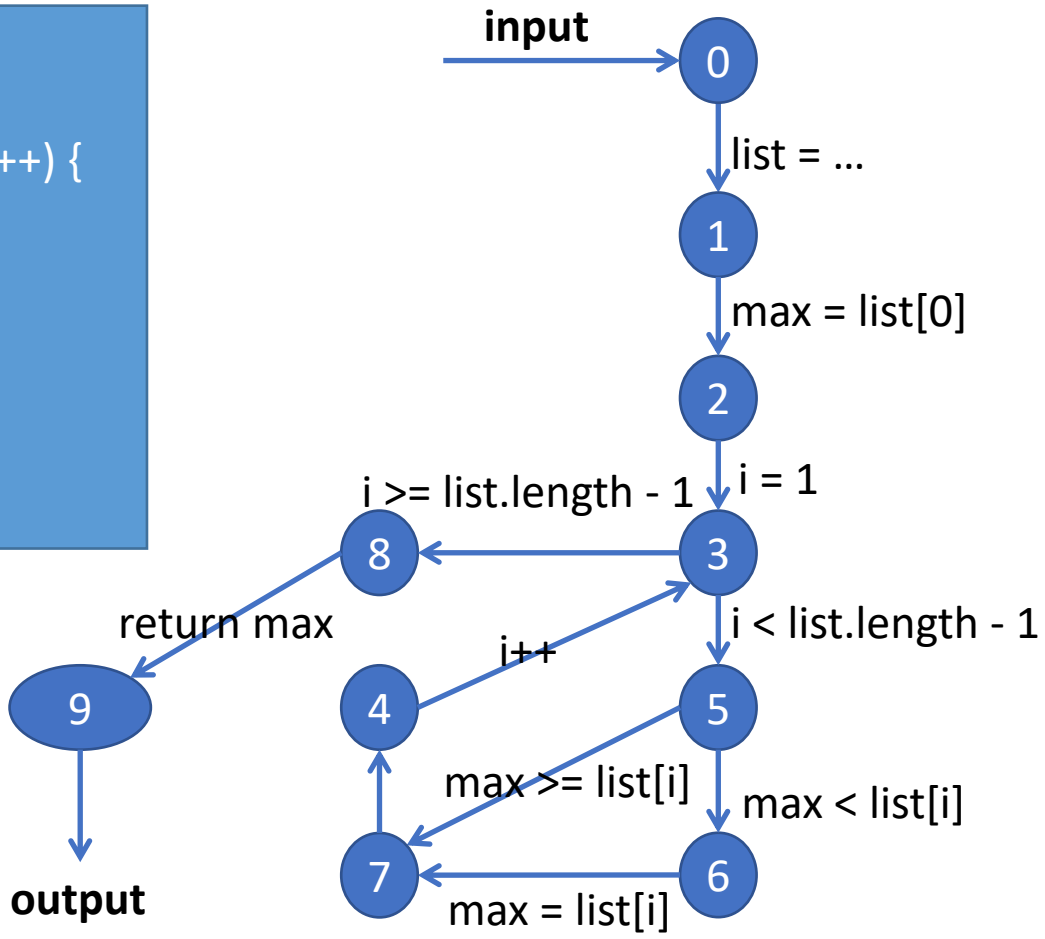
Minimize the failed test

Debugging (cont.)

- Question 1: Bug Localization
  - How do we localize the bugs?
- Question 2: Bug Fixing
  - How do we fix the bugs?

Where is the Bug?

```
public int max (int[] list) {
    int max = list[0];
    for (int i = 1; i < list.length-1; i++) {
        if (max < list[i]) {
            max = list[i];
        }
    }
    return max;
}
```



Isolating Cause-Effect Chains from Computer Programs

Andreas Zeller

FSE 2002, Citation: 662

The GCC Example

Recall that the following example crashes the GCC compiler

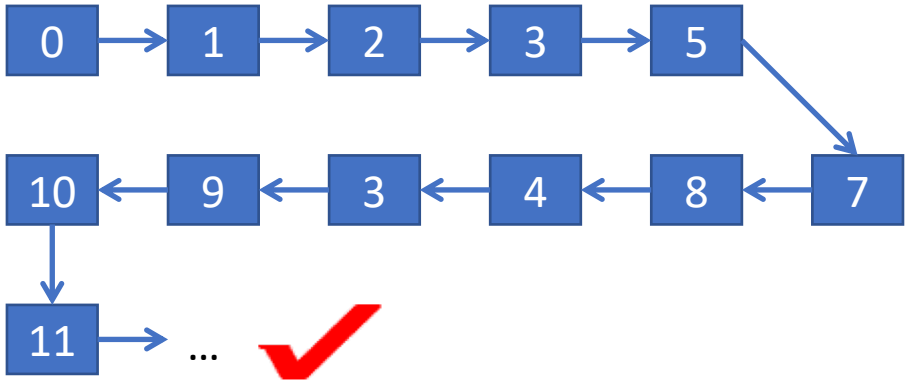
```
#define SIZE 20

double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}

void copy(double to[], double from[], int count)
{
  int n = (count + 7) / 8;
  switch (count % 8) do {
     case 0: *to++ = *from++;
     case 7: *to++ = *from++;
     case 6: *to++ = *from++;
     case 5: *to++ = *from++;
     case 4: *to++ = *from++;
     case 3: *to++ = *from++;
     case 2: *to++ = *from++;
     case 1: *to++ = *from++;
  } while (--n > 0);
  return mult(to, 2);
}

int main(int argc, char *argv[])
{
  double x[SIZE], y[SIZE];
  double *px = x;

  while (px < x + SIZE)
    *px++ = (px - x) * (SIZE + 1.0);
  return copy(y, x, SIZE);
}
```

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

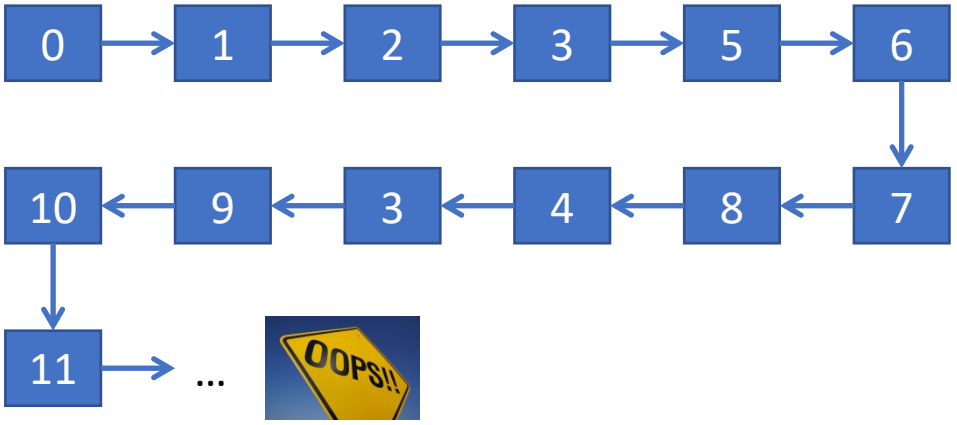
What caused this crash and how to debug the GCC compiler?

We Know

“”




```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```



How to isolate failure-inducing inputs?

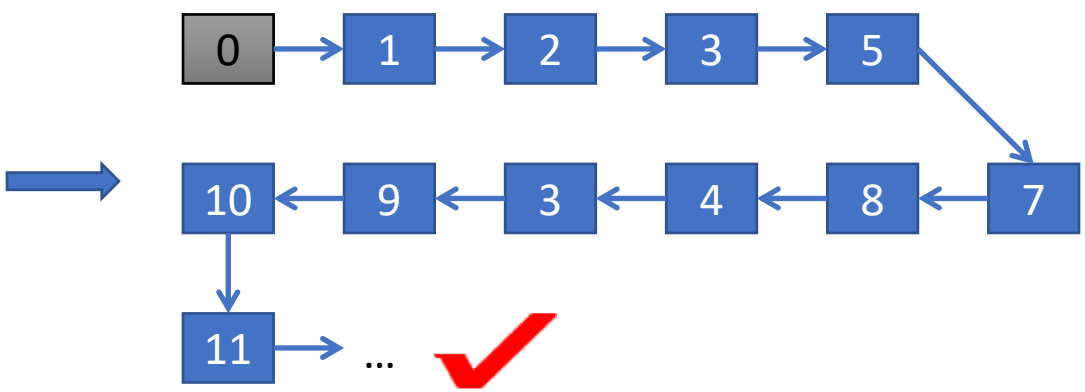
Apply DD on the GCC Example

#	GCC input	*test*
1	double **mult**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	✘
2	double **mult**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	✓

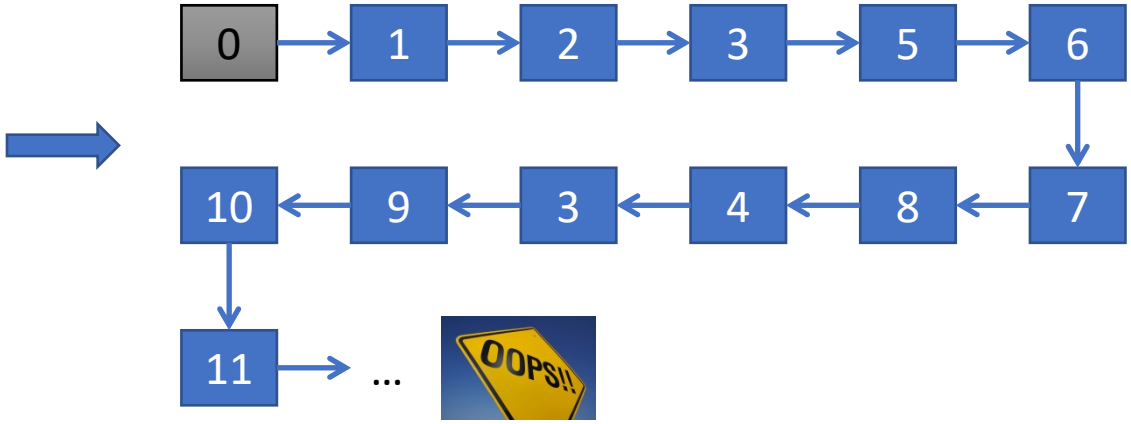
The minimum failure-inducing difference is “+ 1.0”

Now We Know

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

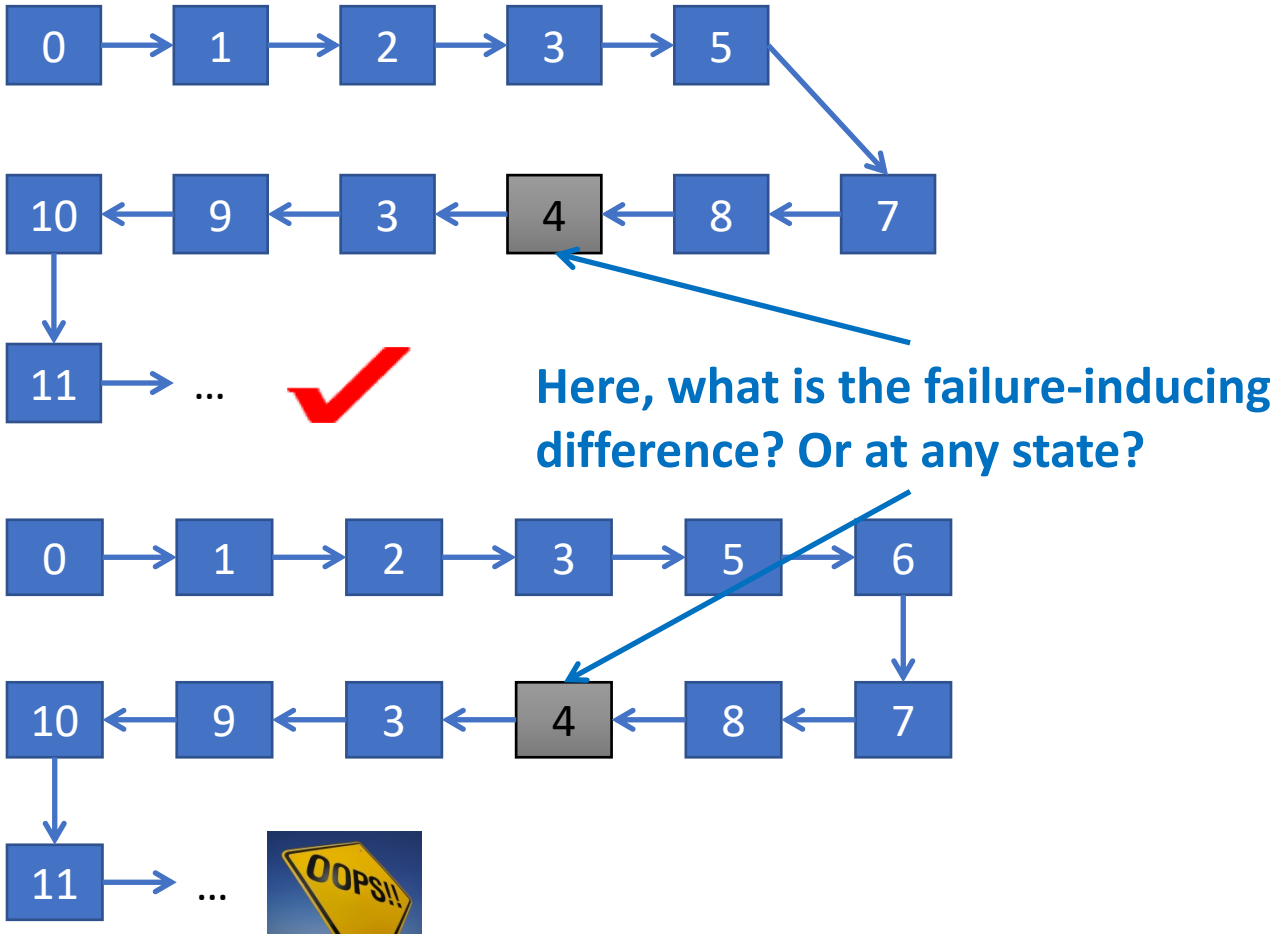



```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

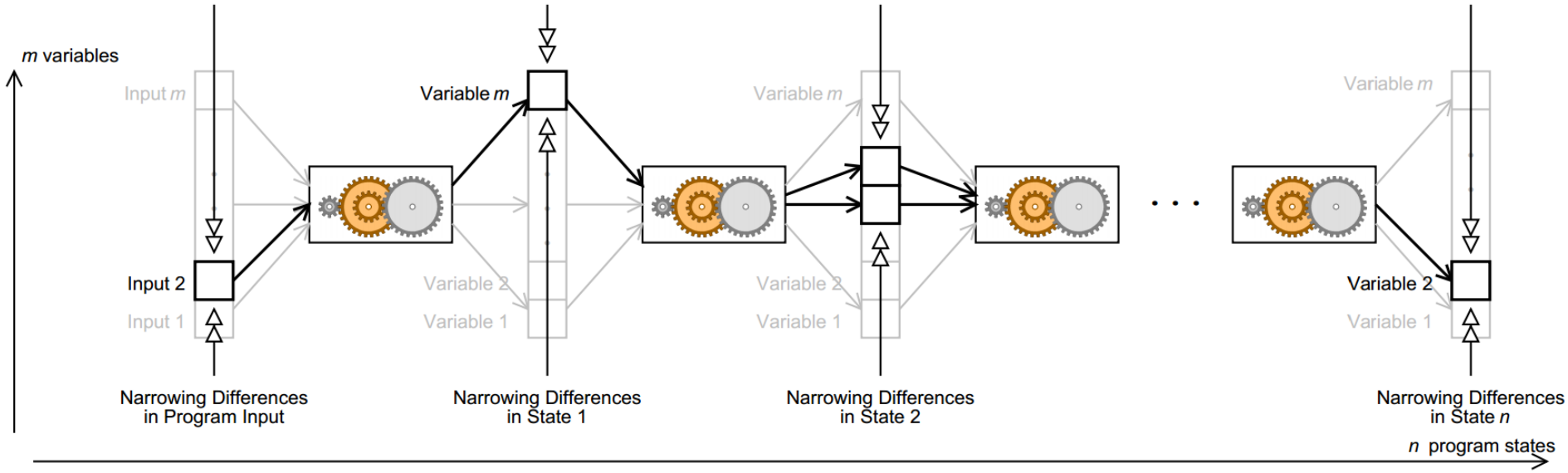


This input difference causes a difference in later program states

The Idea


0
1
2
3
5
10
9
3
4
8
7
11
...
Here, what is the failure-inducing difference? Or at any state?
0
1
2
3
5
6
10
9
3
4
8
7
11
...
OOPS!!

The Idea

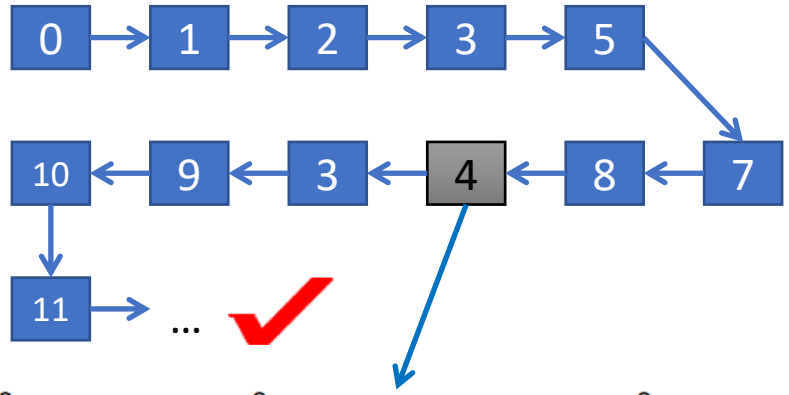


We change a program state (in the trace) in order to find the minimum failure-inducing state difference

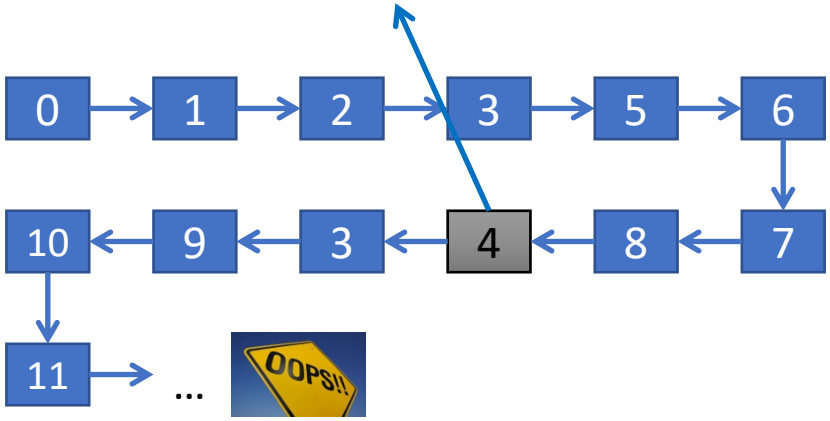
What Are Required

- A debugging tool which allows us to retrieve and alter variables and values in a program state
- A successful test case, a failed test case, and a common program location

Example



	δ_1	δ_2	δ_3	δ_4	
	reg_rtx_no	*cur_insn_uid*	*first_loop_store_insn*	*last_linenum*	*test*
$r_{\times}$	32	74	0x81fc4e4	15	✘
$r_{\checkmark}$	31	70	0x81fc4a0	14	✔



Delta Debugging

	δ_1	δ_2	δ_3	δ_4	
	reg_rtx_no	*cur_insn_uid*	*first_loop_store_insn*	*last_linenum*	*test*
$r_{\times}$	32	74	0x81fc4e4	15	✘
$r_{\checkmark}$	31	70	0x81fc4a0	14	✔

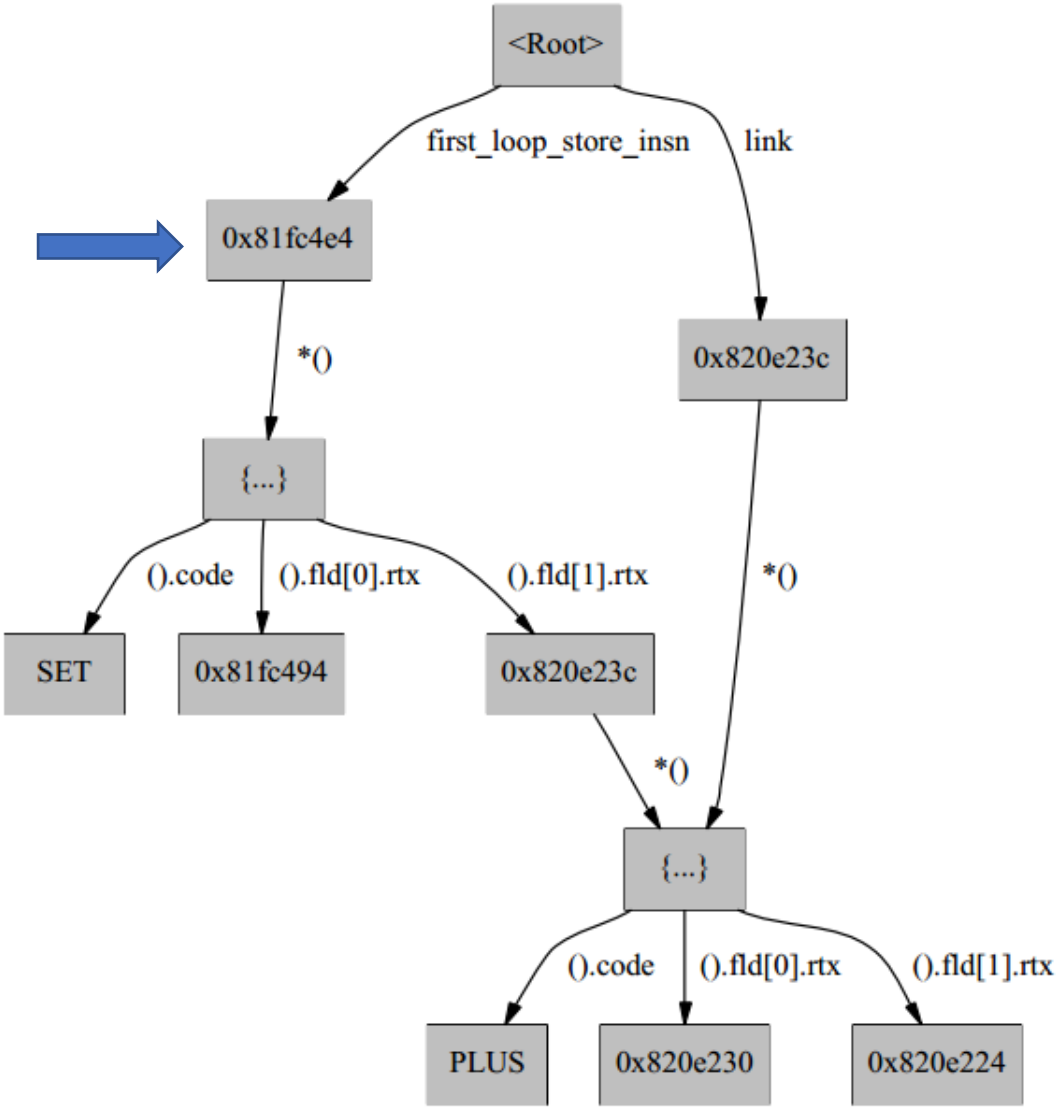
#	*reg_rtx_no*	*cur_insn_uid*	*first_loop_store_insn*	*last_linenum*	*test*
1	32	74	0x81fc4a0	14	✔
2	32	74	0x81fc4e4	14	?
3	32	74	0x81fc4a0	15	✔

Obviously, the problem is with *ox81fc4e4.*
But is this useful?

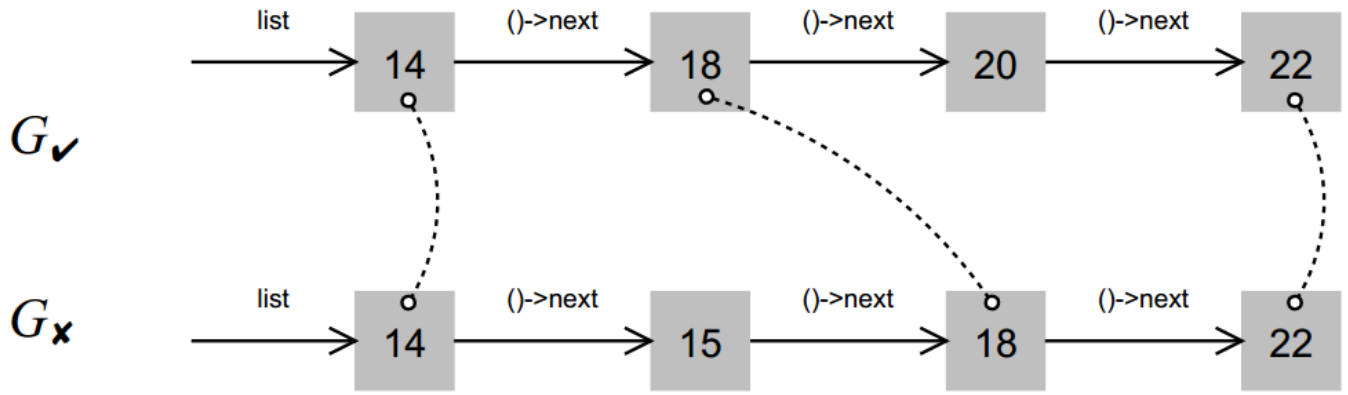
Memory Graphs

- 0x81fc4a0 and 0x81fc4e4 are actually memory references – they cannot be compared directly
- Instead, compare the memory graphs at these two states and find the exact difference

Memory Graphs

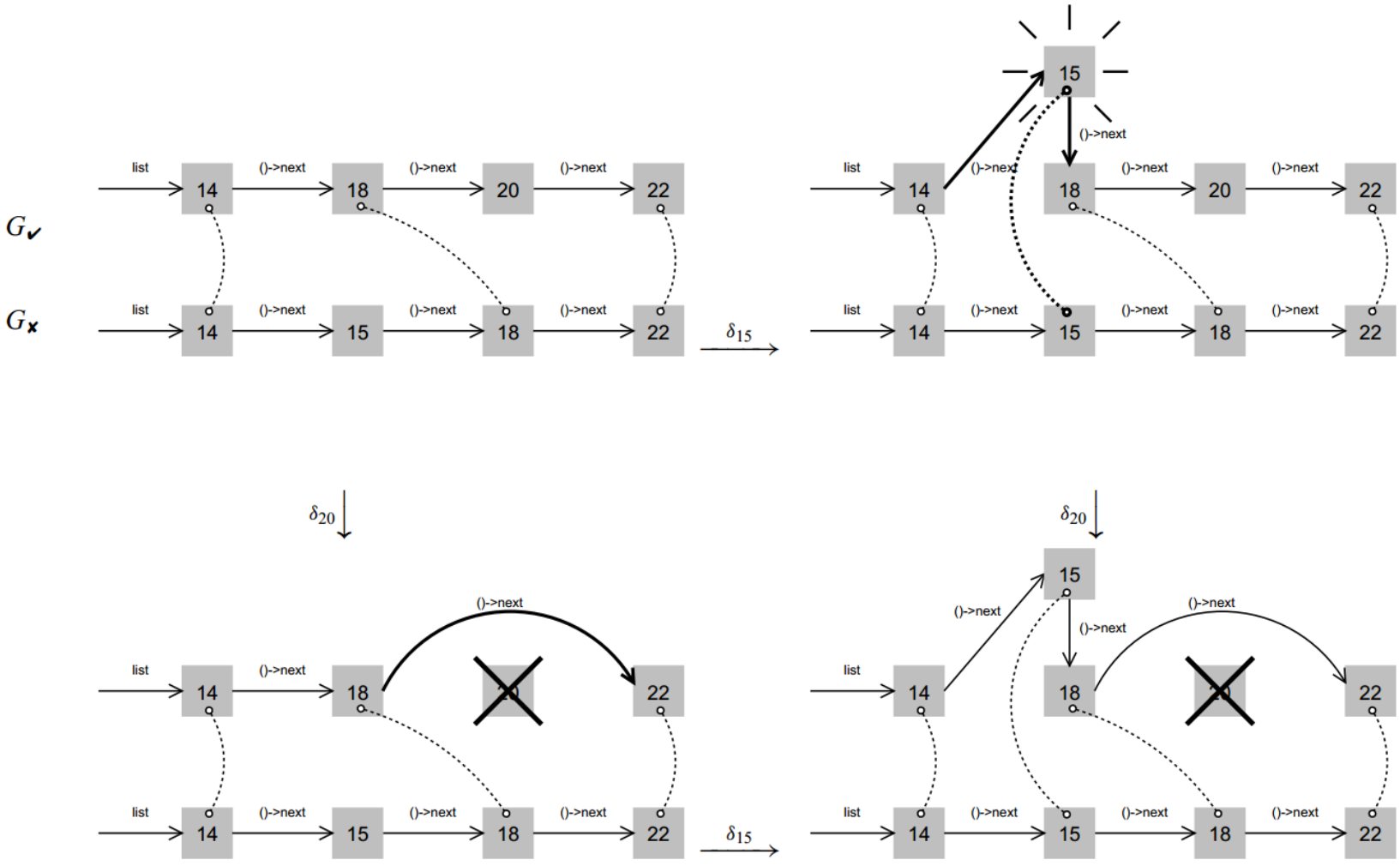


Comparing Memory Graphs



To use Delta Debugging, we need to define a set of changes on the good memory graph so that it transforms to the bad one

Changes on Memory Graphs



Changes on Memory Graphs

- Compute the largest common subgraph of two graphs
- For all vertex which are not part of the common subgraph, either insert or delete the vertex

Recap

We now know

- How to apply Delta Debugging to two program states in two traces

We still need to solve

- Given a sequence of program states, which ones to compare?
- How to present the result to the users?

Which States to Compare

- We compare states which have identical program counter and local variables
  - The set of local variables cannot be changed
- The paper suggests three places
  - Shortly after the program starts
  - In the middle of the program run
  - Shortly before the failure

GCC: Shortly After Start – *main*

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

27,139 vertices and 27,159 edges

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

27,139 vertices and 27,159 edges

The only difference is in argv[2], i.e., the names of the input source files

GCC: In the Middle – *combine_instructions*

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

42,991 vertices and 44,290 edges

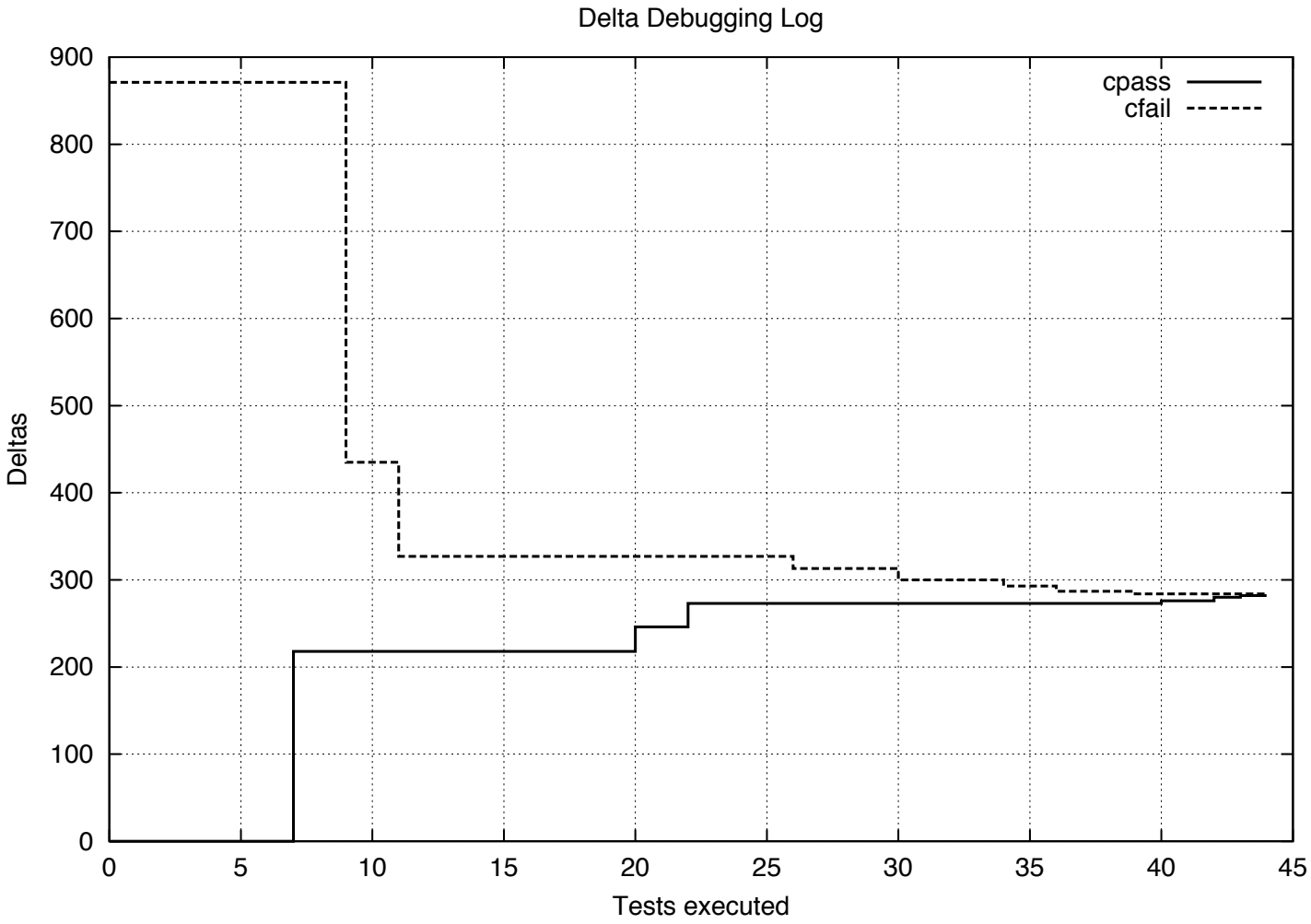
```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

43,147 vertices and 44,460 edges

871 vertices need to be added or deleted. The failure-inducing difference is the insertion of a node in the RTL tree containing a PLUS operator

GCC: In the Middle – *combine_instructions*



GCC: Shortly Before Failure – *if_then_else_cond*

```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

47,071 vertices and 48,744 edges

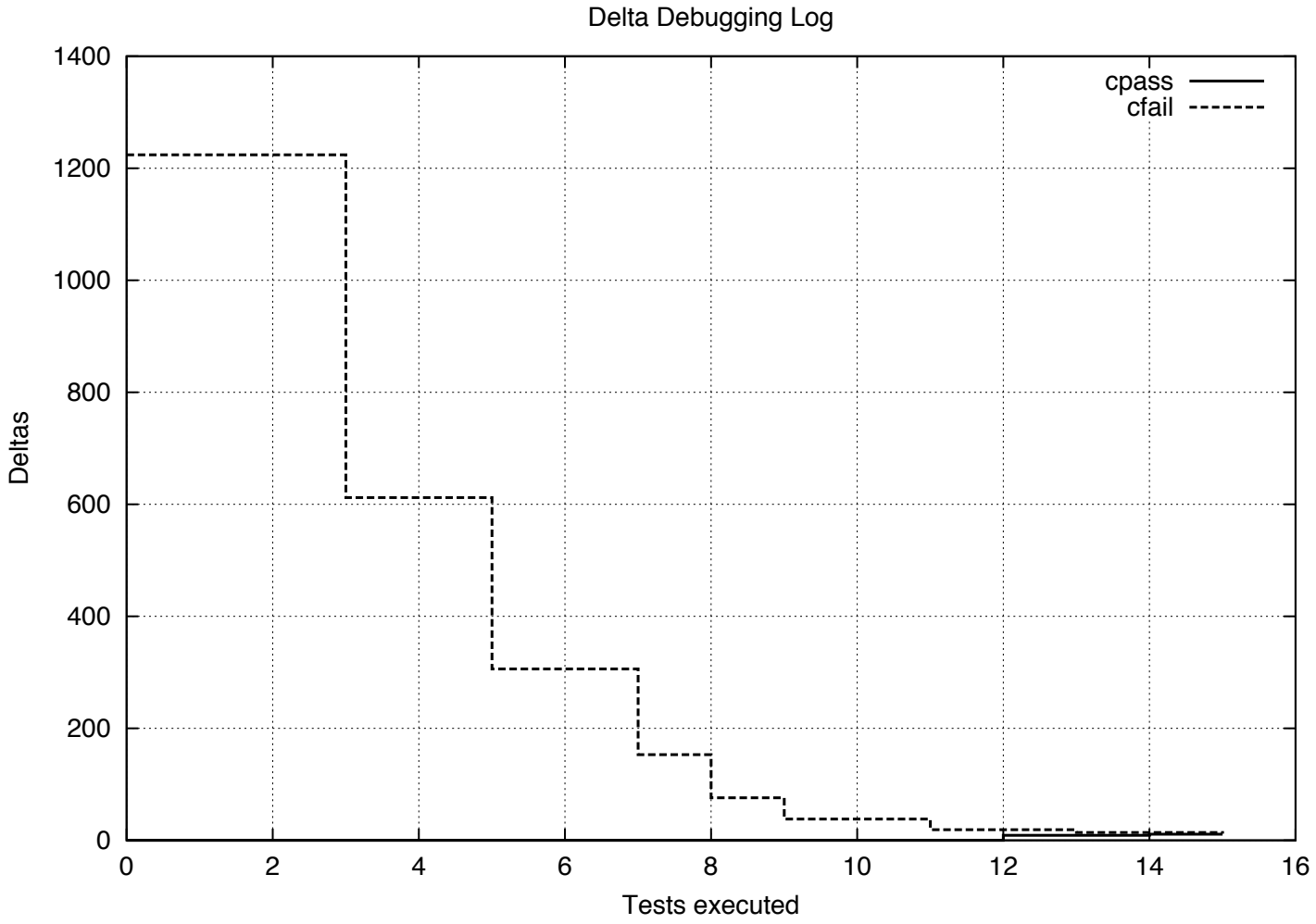
```
double mult(double z[], int n)
{
  int i, j;

  i = 0;
  for (j = 0; j < n; j++) {
    i = i + j + 1;
    z[i] = z[i] * (z[0] + 1.0);
  }
  return z[n];
}
```

Memory graph:

47,313 vertices and 48,744 edges

1,224 vertices need to be added or deleted. The failure-inducing difference is a single pointer adjustment: set variable link->fld[0].rtx->fld[0].rtx = link

GCC: Shortly Before Failure – *if_then_else_cond*



Report from Delta Debugging

Cause-effect chain for ’./gcc/cc1’
Arguments are -O fail.i (instead of -O pass.i)
therefore at main, *argv[2] = ”fail.i” (instead of ”pass.i”)*
therefore at combine_instructions,
  **first_loop_store_insn→fld[1].rtx→fld[1].rtx→*
    fld[3].rtx→fld[1].rtx = ⟨new variable⟩
therefore at if_then_else_cond,
  link→fld[0].rtx→fld[0].rtx = link (instead of i1dest)
therefore the run fails.

Recap

We now know

- How to apply Delta Debugging to multiple pairs of program states in two traces
- How the results are presented to users (using the minimum failure-inducing differences in chains)

We still need to solve

- Where the bug is?

Bug Localization with DD

Iteration 1:

Cause-effect chain for ’./gcc/cc1’
Arguments are -O fail.i (instead of -O pass.i)
therefore at main, *argv[2] = ”fail.i” (instead of ”pass.i”)*
therefore at combine_instructions,
**first_loop_store_insn→fld[1].rtx→fld[1].rtx→*
fld[3].rtx→fld[1].rtx = ⟨new variable⟩
therefore at if_then_else_cond,
link→fld[0].rtx→fld[0].rtx = link (instead of i1dest)
therefore the run fails.

This is what the programmer wants.

The failure is not what the programmer wants.

The bug must be in between.

Bug Localization with DD

Iteration 2:

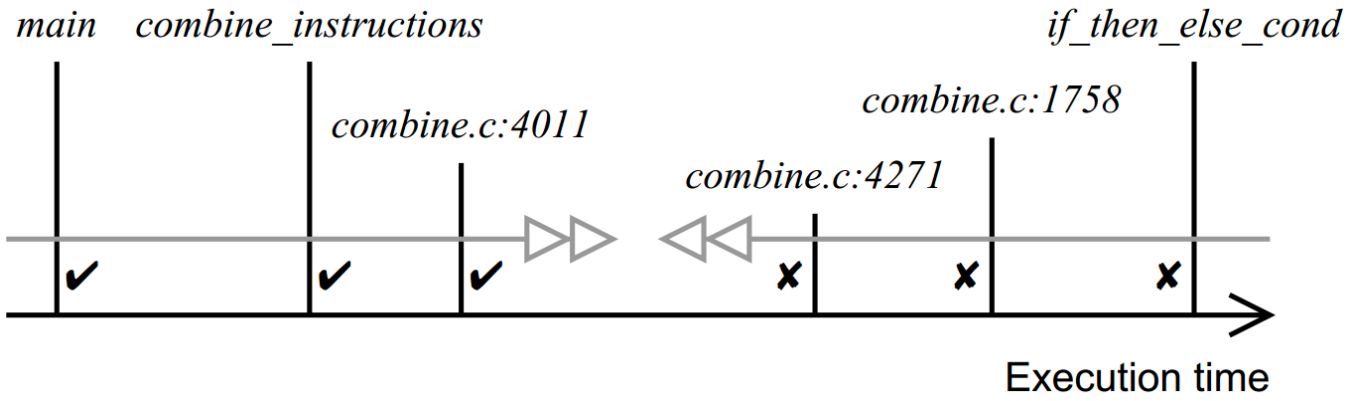
Cause-effect chain for ’./gcc/cc1’
Arguments are -O fail.i (instead of -O pass.i)
therefore at main, *argv[2] = ”fail.i” (instead of ”pass.i”)*
therefore at combine_instructions,
**first_loop_store_insn→fld[1].rtx→fld[1].rtx→*
fld[3].rtx→fld[1].rtx = ⟨new variable⟩
therefore at if_then_else_cond,
link→fld[0].rtx→fld[0].rtx = link (instead of i1dest)
therefore the run fails.

This is what the programmer wants.

The failure is not what the programmer wants.

The bug must be in between.

Bug Localization with DD



The bug is narrowed down to line 4013 to line 4019, where the bug is truly is.

The Bug in GCC

```
(MULT (PLUS a b) c)
```

is transformed into

```
(PLUS (MULT a c1)(MULT a c2))
```

c_1 and c_2 are created as *aliases* of *c*, which causes the cycle in the RTL tree!
To fix the error, one should make c_2 a true copy of c_1.

More Case Studies

Event	Edges	Vertices	Deltas	Tests
`sample` at *main*	26	26	12	4
`sample` at *shell_sort*	26	26	12	7
`sample` at *sample.c*:37	26	26	12	4
`cc1` at *main*	27139	27159	1	0
`cc1` at *combine_instructions*	42991	44290	871	44
`cc1` at *if_then_else_cond*	47071	48473	1224	15
`bison` at *open_files*	431	432	2	2
`bison` at *initialize_conflicts*	1395	1445	431	42
`diff` at *analyze.c*:966	413	446	109	9
`diff` at *analyze.c*:1006	413	446	99	10
`gdb` at *main.c*:615	32455	33458	1	0
`gdb` at *exec.c*:320	34138	35340	18	7

Quick Summary

- Identify minimum difference between two states
- Programmers are required to look at the minimum difference to locate the bug step-by-step

- **Search in space** across a program state to find the infected variable(s)—often among thousands
- **Search in time** over millions of such program states to find the moment in time when the infection began—the moment the defect was executed
  - Holger Cleve and Andreas Zeller, “Locating Causes of Program Failures,” in ICSE, pp. 342-351, 2005.

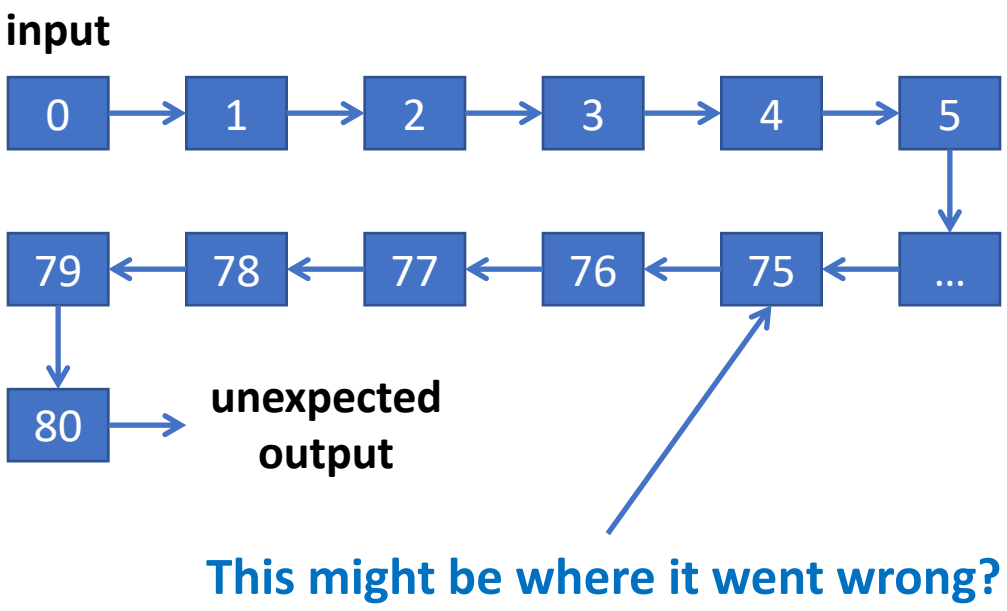
Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique

James A. Jones and Mary Jean Harrold

ASE 2005, Citation: 963

Where is the Bug?

Something has to be fixed along this trace



Coverage-Based Set Union and Intersection

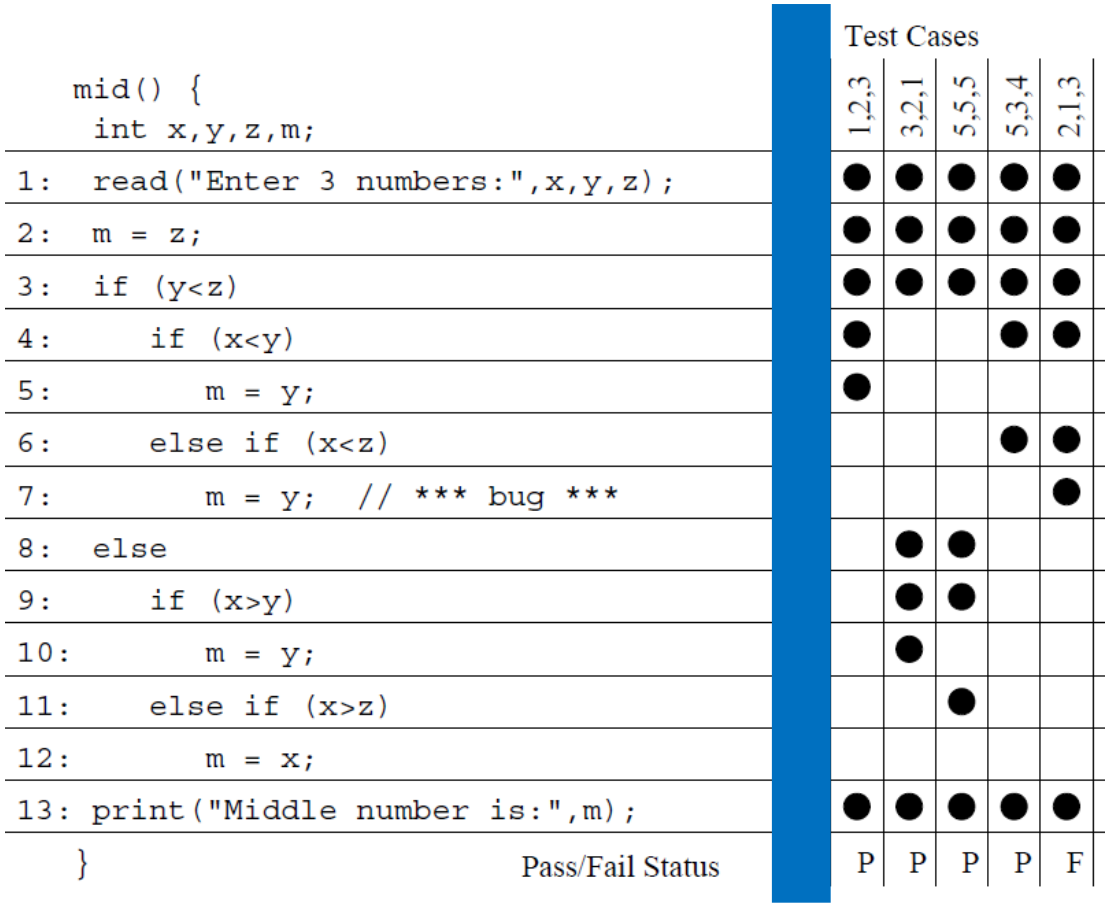
- A statement is more likely to be buggy if it is visited more often in failed test cases and less often in passed test cases

- Assume there are one or more passed test cases {p_0, p_1, p_2, …} and one failed test case f
- **Set Union**: the bug is contained in the set

  {statements executed by f} – {statements executed by any p_i}
- **Set Intersection**: the bug is contained in the set

  {statements executed by all p_i} – {statements executed by f}

Example: Apply Set Union and Intersection



		Test Cases				
		1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
	`mid() {` `int x,y,z,m;`					
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●
4:	`if (x<y)`	●			●	●
5:	`m = y;`	●				
6:	`else if (x<z)`				●	●
7:	`m = y; // *** bug ***`					●
8:	`else`		●	●		
9:	`if (x>y)`		●	●		
10:	`m = y;`		●			
11:	`else if (x>z)`			●		
12:	`m = x;`					
13:	`print("Middle number is:",m);`	●	●	●	●	●
	`}` Pass/Fail Status	P	P	P	P	F

Example: Apply Set Union and Intersection

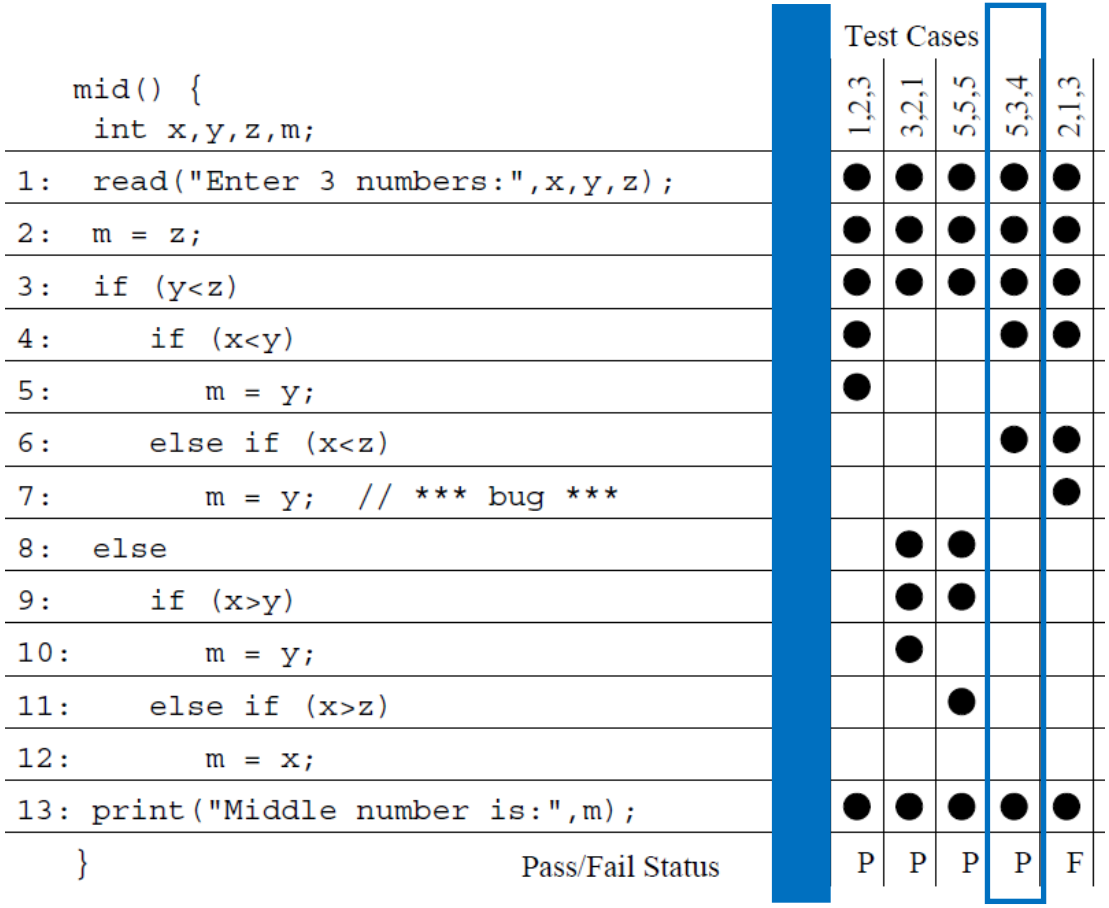
		Test Cases					
	`mid() {` `int x,y,z,m;`	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●	●
4:	`if (x<y)`	●	●			●	●
5:	`m = y;`		●				
6:	`else if (x<z)`	●				●	●
7:	`m = y; // *** bug ***`	●					●
8:	`else`			●	●		
9:	`if (x>y)`			●	●		
10:	`m = y;`			●			
11:	`else if (x>z)`				●		
12:	`m = x;`						
13:	`print("Middle number is:",m);`	●	●	●	●	●	●
	`}` Pass/Fail Status	P	P	P	P	P	F

Set union/intersection performs poorly in practice – sensitive to the test cases.

Nearest Neighbor

- Pick a passed test case p which is the nearest to the failed test case f
- Distance between f and p can be defined as: the number of statements in f which are not in p
- Report the blamed set as {statements executed by f} – {statements executed p}

Example: Apply Nearest Neighbor



		Test Cases				
		1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
	`mid() {` `int x,y,z,m;`					
1:	`read("Enter 3 numbers:",x,y,z);`	●	●	●	●	●
2:	`m = z;`	●	●	●	●	●
3:	`if (y<z)`	●	●	●	●	●
4:	`if (x<y)`	●			●	●
5:	`m = y;`	●				
6:	`else if (x<z)`				●	●
7:	`m = y; // *** bug ***`					●
8:	`else`		●	●		
9:	`if (x>y)`		●	●		
10:	`m = y;`		●			
11:	`else if (x>z)`			●		
12:	`m = x;`					
13:	`print("Middle number is:",m);`	●	●	●	●	●
	`}` Pass/Fail Status	P	P	P	P	F

Example: Apply Nearest Neighbor

	Test Cases					
mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●
2: m = z;	●	●	●	●	●	●
3: if (y<z)	●	●	●	●	●	●
4: if (x<y)	●	●			●	●
5: m = y;		●				
6: else if (x<z)	●				●	●
7: m = y; // *** bug ***	●					●
8: else			●	●		
9: if (x>y)			●	●		
10: m = y;			●			
11: else if (x>z)				●		
12: m = x;						
13: print("Middle number is:",m);	●	●	●	●	●	●
} Pass/Fail Status	P	P	P	P	P	F

Nearest neighbor is also sensitive to the test cases.

SDG Ranking Technique

What if the bug is not found in the initial blamed set?

1) Find statements of distance 1 from the initial blamed set (in forward and backward direction), and rank them the highest
2) Find statements of distance 2 from the initial blamed set, and rank them the second highest
3) Find statements of distance 3 and so on

Tarantula

Each statement in the program is assigned a *suspiciousness* score

$$suspiciousness(e) = \frac{\frac{failed(e)}{totalfailed}}{\frac{passed(e)}{totalpassed} + \frac{failed(e)}{totalfailed}}$$

where *failed(e)* is the number of failed test cases that executed statement *e* one or more times, *totalfailed* is the number of failed test cases

Example: Apply Tarantula

	Test Cases						suspiciousness	rank
mid() { int x,y,z,m;	3,3,5	1,2,3	3,2,1	5,5,5	5,3,4	2,1,3		
1: read("Enter 3 numbers:",x,y,z);	●	●	●	●	●	●	0.5	7
2: m = z;	●	●	●	●	●	●	0.5	7
3: if (y<z)	●	●	●	●	●	●	0.5	7
4: if (x<y)	●	●			●	●	0.63	3
5: m = y;		●					0.0	13
6: else if (x<z)	●				●	●	0.71	2
7: m = y; // *** bug ***	●					●	0.83	1
8: else			●	●			0.0	13
9: if (x>y)			●	●			0.0	13
10: m = y;			●				0.0	13
11: else if (x>z)				●			0.0	13
12: m = x;							0.0	13
13: print("Middle number is:",m);	●	●	●	●	●	●	0.5	7
} Pass/Fail Status	P	P	P	P	P	F		

Empirical Study

- Task: locate the buggy statement
- Five techniques: set union, set interaction, nearest neighbor, tarantula, and delta debugging
- Two measures
  - Effectiveness: the rank of the buggy statement
  - Efficiency: the time consumption of each technique

Objects of Analysis

Program	Faulty Versions	Procedures	LOC	Test Cases	Description
print_tokens	7	20	472	4056	lexical analyzer
print_tokens2	10	21	399	4071	lexical analyzer
replace	32	21	512	5542	pattern replacement
schedule	9	18	292	2650	priority scheduler
schedule2	10	16	301	2680	priority scheduler
tcas	41	8	141	1578	altitude separation
tot_info	23	16	440	1054	information measure

122 out of 132 versions are used for the analysis.

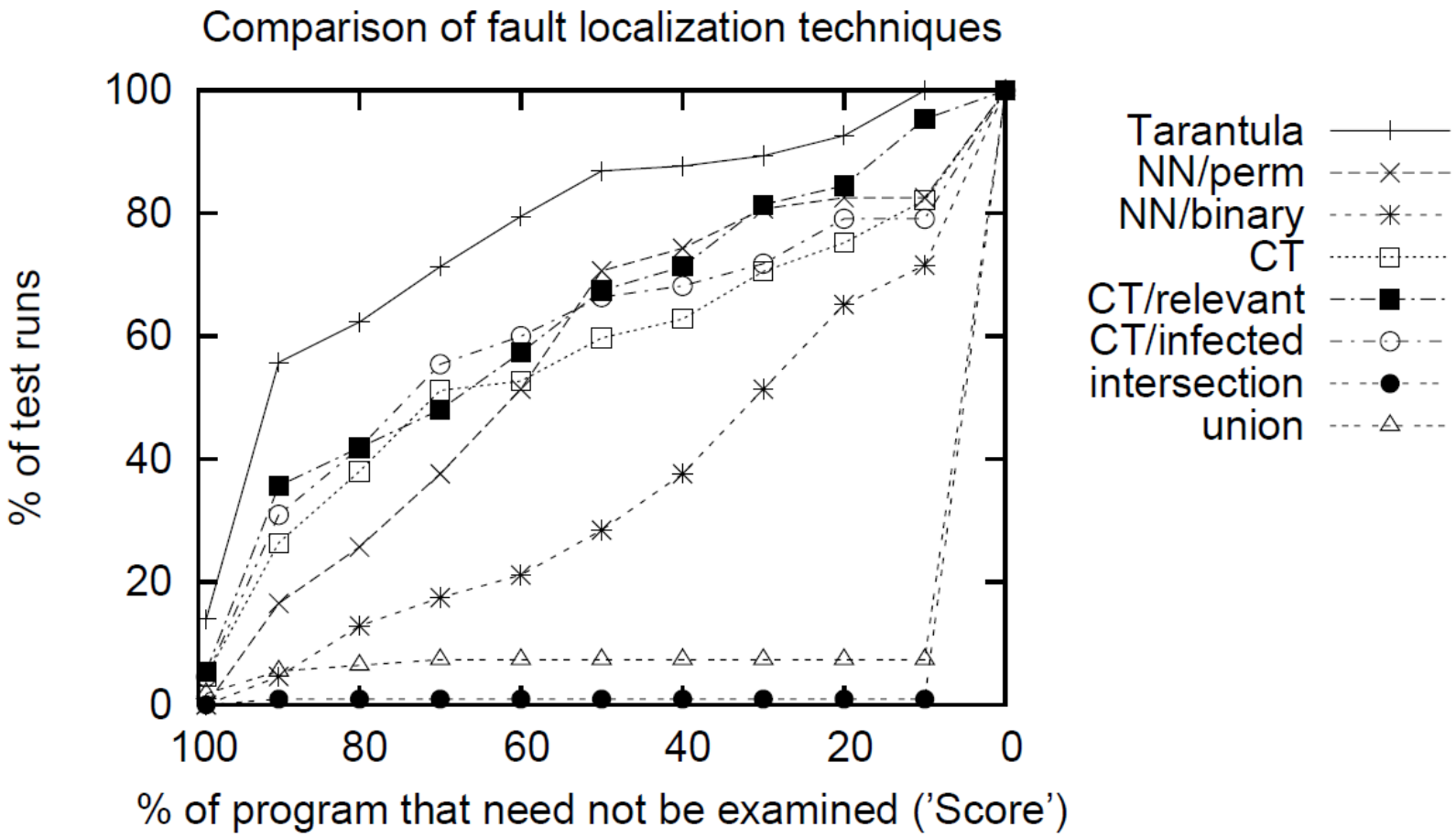
Effectiveness Results

NN: Nearest Neighbor (with different distance calculation)
CT: Delta Debugging (with different ranking techniques)

Score	Tarantula	NN/perm	NN/binary	CT	CT/relevant	CT/infected	Intersection	Union
99-100%	13.93	0.00	0.00	4.65	5.43	4.55	0.00	1.83
90-99%	41.80	16.51	4.59	21.71	30.23	26.36	0.92	3.67
80-90%	5.74	9.17	8.26	11.63	6.20	10.91	0.00	0.92
70-80%	9.84	11.93	4.59	13.18	6.20	13.64	0.00	0.92
60-70%	8.20	13.76	3.67	1.55	9.30	4.55	0.00	0.00
50-60%	7.38	19.27	7.33	6.98	10.08	6.36	0.00	0.00
40-50%	0.82	3.67	9.17	3.10	3.88	1.82	0.00	0.00
30-40%	0.82	6.42	13.76	7.75	10.08	3.64	0.00	0.00
20-30%	4.10	1.83	13.76	4.65	3.10	7.27	0.00	0.00
10-20%	7.38	0.00	6.42	6.98	10.85	0.00	0.00	0.00
0-10%	0.00	17.43	28.44	17.83	4.65	20.91	99.08	92.66

The score defines the percentage of the program that need not be examined to find a faulty statement in the program

Effectiveness Results



Efficiency Results

Program	Tarantula (computation only)	Tarantula (including I/O)	Cause Transitions
print_tokens	0.0040	68.96	2590.1
print_tokens2	0.0037	50.50	6556.5
replace	0.0063	75.90	3588.9
schedule	0.0032	30.07	1909.3
schedule2	0.0030	30.02	7741.2
tcas	0.0025	12.37	184.8
tot_info	0.0031	8.51	521.4

I/O time: read and parse the coverage information about the test cases

Reading Materials

- A. Zeller. Isolating cause-effect chains from computer programs. In FSE, pp. 1-10, 2002.
- H. Cleve and A. Zeller. Locating causes of program failures. In ICSE, pp. 342-351, 2005.
- J. A. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In ASE, pp. 273-282, 2005.
- J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In ICSE, pp. 467-477, 2002.
- M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In ASE, pp. 30-39, 2003.
- B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In PLDI, pp. 141-154, 2003.
- B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In PLDI, pp. 15-26, 2005.

Q&A?

Bihuan Chen, Pre-Tenure Assoc. Prof.

bhchen@fudan.edu.cn

https://chenbihuan.github.io