Isolating Cause-Effect Chains

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken







GCC Revisited

Consider the following C program:

```
double bug(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];
}
```

bug.c causes the GNU compiler (GCC) to crash:

```
linux$ gcc-2.95.2 -O bug.c
gcc: Internal error: program cc1 got fatal signal 11
linux$ _
```







Why does GCC crash?

We want to determine the *cause* of the GCC crash:

> *The **cause** of any event ("**effect**") is a preceding event without which the effect would not have occurred.*
>
> — Microsoft Encarta

To prove causality, we must show experimentally that

1. the effect occurs when the cause occurs
2. the effect does *not* occur when the cause does *not* occur.

In our case, the *effect* is GCC crashing.
The *cause* must be something *variable* – e.g. the GCC input.

Isolating Failure Causes

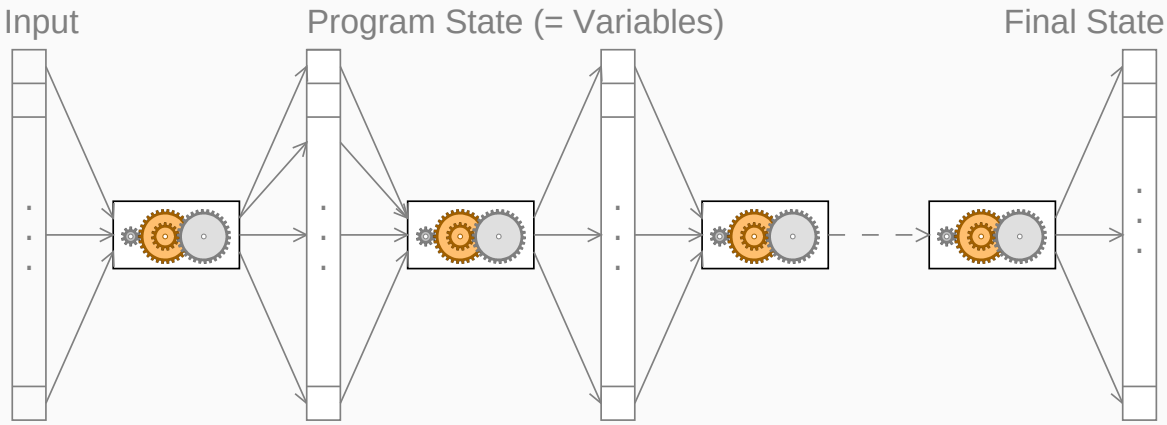
Delta Debugging automatically isolates the *failure-inducing difference* in the GCC input:

#	GCC input	test
1	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✘
5	double **bug**(...) { int i, j; $i = 0$; for (...) { ... }... }	✘
⋮		
19	... $z[i] = z[i] * (z[0] + 1.0);$ ...	✘
18	... $z[i] = z[i] * (z[0] + 1.0);$ ...	✔
⋮		
4	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔
3	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔
2	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔

$+ 1.0$ is the failure cause – after only 19 tests ($\approx$ 2 seconds)

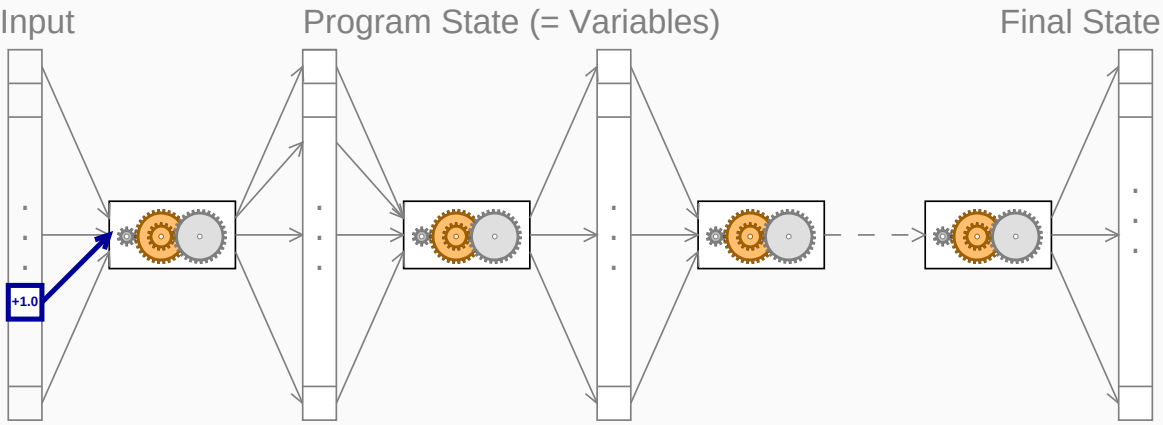
What's going on in GCC?

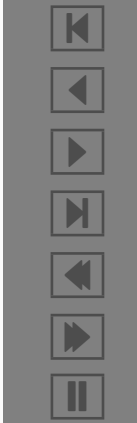
The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



What's going on in GCC?

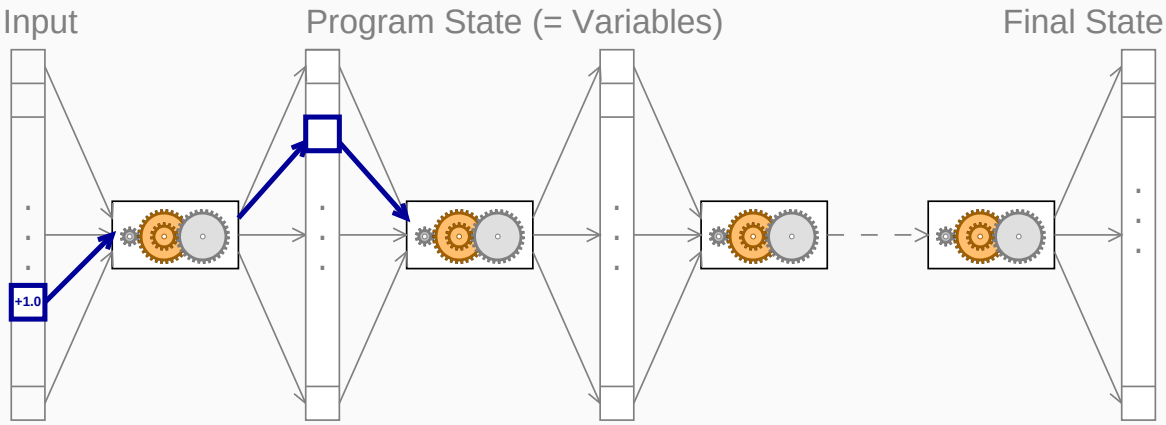
The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.





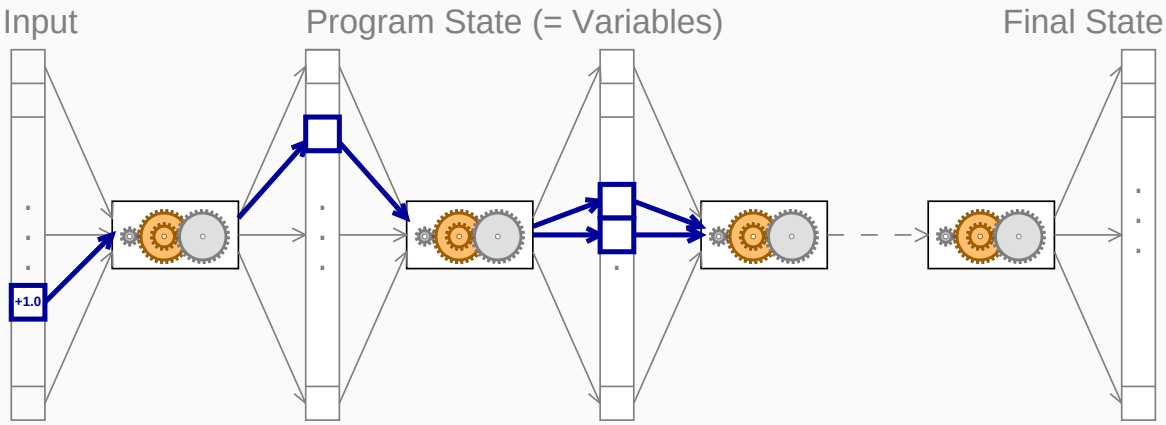
What's going on in GCC?

The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



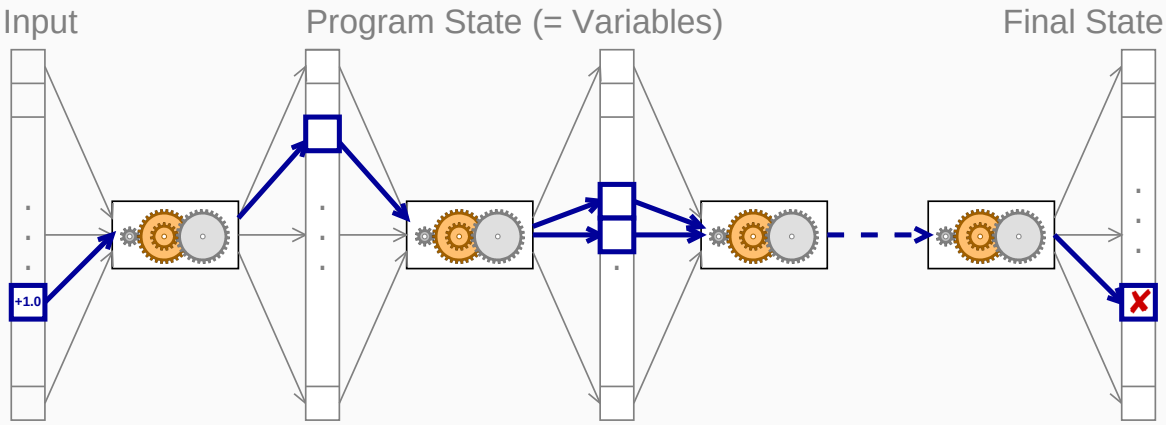
What's going on in GCC?

The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



What's going on in GCC?

The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



To fix the bug, we must *break* this cause-effect chain.

Following Origins

Using *program analysis*, one can follow how effects propagate across the program states.

Requirement: Complete Knowledge about the entire code and its semantics ⇒ good for small, closed programs.

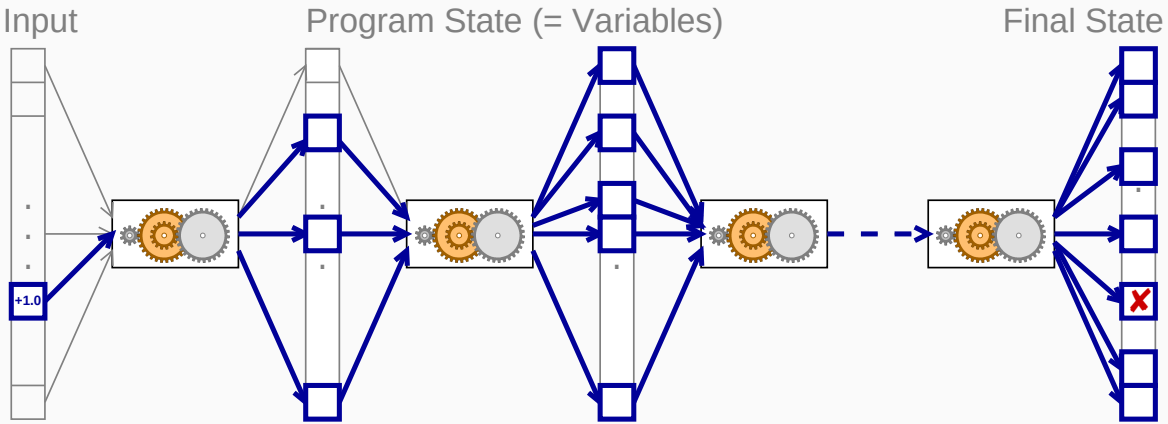
But: real programs are *opaque, parallel, distributed, dynamic, multilingual* – or simply obscure:

```
struct foo {
  int tp, len;
  union {
    char   c[1];
    int    i[1];
    double d[1];
  }
}
```

```
// Allocate string
int len = 200;
int bytes = len + 2 * sizeof(int);
foo *x = (foo *)malloc(bytes);
x->tp = STRING;
x->len = len;
strncpy(x->c, "Some string", len);
```

Small Cause, Large Effect

Effects *accumulate* during execution:



This happens in *static* as well as in *dynamic* analysis.

Scientific Techniques

The techniques we discussed so far can be classified into three groups:

Deduction from the program code to the actual run.

Example: Static program analysis

Observation of the actual run; comparison of observed state vs. expected state

Example: Debuggers, assertions, queries, dynamic program analysis

Experimentation to validate and narrow down causes

Example: Delta Debugging (on input, code, schedule)







Scientific Techniques (2)

Let us recall the causality example:

```
a = compute_value();
printf("a = %d\n", a);  // prints "a = 0"
```

Deduction (static analysis) tells us that a comes from `compute_value()`.

Observation (debugger) tells us that a is not zero.

Experimentation (debugger?) can tell us that

- whatever we change the value of a to, "a = 0" is printed;
- if we change the format to %f, the proper value of a is printed.

Delta Debugging on States

Obviously, a variable value is a cause if we can change it to a value such that the effect no longer occurs.

But which value should we choose?

Basic idea of delta debugging: *Use an alternate run*—a value that comes from a run where the effect does not occur.

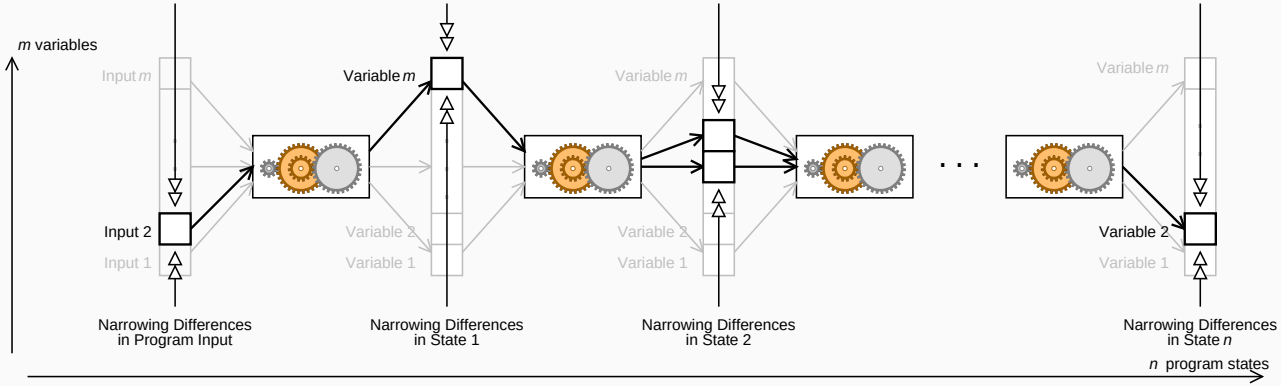






Delta Debugging on States (2)

Within each state, *narrow down* the difference between a sane state and a failure-inducing state—just like we narrow down the difference between a sane input and a failure-inducing input.



Each program state thus becomes an *input* for the remainder of the run.

Sample Revisited

Recall the `sample` program:

```
$ sample 9 8 7
Output: 7 8 9
$ sample 11 14
Output: 0 11
$ _
```







Accessing States

Using a debugger (GDB), we can easily access the `sample` state:

```
$ gdb sample
(gdb) break shell_sort
(gdb) run 7 8 9
Breakpoint 1, shell_sort (a=0x8049838, size=4)
    at sample.c:9
(gdb) info args
a = (int *) 0x8049838
size = 4
(gdb) info locals
i = 1073834752
j = 1074077312
h = 1961
(gdb)
```







Sample States

At the beginning of *shell_sort*, we obtain these states:

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
argc	**4**	**5**
argv[0]	"./sample"	"./sample"
argv[1]	**"9"**	**"11"**
argv[2]	**"8"**	**"14"**
argv[3]	**"7"**	**0x0** (NIL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	**4**	**3**

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
i	**3**	**2**
$a[0]$	**9**	**11**
$a[1]$	**8**	**14**
$a[2]$	**7**	**0**
$a[3]$	1961	1961
$a'[0]$	**9**	**11**
$a'[1]$	**8**	**14**
$a'[2]$	**7**	**0**
$a'[3]$	1961	1961

This state difference is both *effect* (of the input) as well as *cause* (for the failure).

State Deltas

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
argc	**4**	**5**
argv[0]	"./sample"	"./sample"
argv[1]	**"9"**	**"11"**
argv[2]	**"8"**	**"14"**
argv[3]	**"7"**	**0x0** (NIL)
i'	1073834752	1073834752
j	1074077312	1074077312
h	1961	1961
size	**4**	**3**

Variable	Value	
	in $r_{✓}$	in $r_{✗}$
i	**3**	**2**
$a[0]$	**9**	**11**
$a[1]$	**8**	**14**
$a[2]$	**7**	**0**
$a[3]$	1961	1961
$a'[0]$	**9**	**11**
$a'[1]$	**8**	**14**
$a'[2]$	**7**	**0**
$a'[3]$	1961	1961

We define a set of *deltas* δ_i, each changing variable i:

$$\Delta = c_{✗} - c_{✓} = c_{✗} = \{\delta_{argc}, \delta_{argv[1]}, \delta_{argv[2]}, \delta_{argv[3]}, \delta_{size},$$
$$\delta_i, \delta_{a[0]}, \delta_{a[1]}, \delta_{a[2]}, \delta_{a'[0]}, \delta_{a'[1]}, \delta_{a'[2]}\}$$

Altering State

Using a debugger (GDB), we can easily *change* states:

```
$ gdb sample
(gdb) break shell_sort
(gdb) run 7 8 9
Breakpoint 1, shell_sort (a=0x8049838, size=4)
    at sample.c:9
(gdb) print a[0]
$1 = 7
(gdb) set variable a[0] = 1
(gdb) continue
Continuing.
1 8 9

Program exited normally.
(gdb) _
```







Narrowing Down State

Delta Debugging narrows down failure-inducing state changes:

■ = δ is applied, □ = δ is *not* applied

#	$a'[0]$	$a[0]$	$a'[1]$	$a[1]$	$a'[2]$	$a[2]$	*argc*	*argv*[1]	*argv*[2]	*argv*[3]	i	*size*	Output	Test
1	□	□	□	□	□	□	□	□	□	□	□	□	7 8 9	✔
2	■	■	■	■	■	■	■	■	■	■	■	■	0 11	✘
3	■	■	■	■	■	■	□	□	□	□	□	□	0 11 14	✘
4	■	■	■	□	□	□	□	□	□	□	□	□	7 11 14	?
5	□	□	□	■	■	■	□	□	□	□	□	□	0 9 14	✘
6	□	□	□	■	□	□	□	□	□	□	□	□	7 9 14	?
7	□	□	□	□	■	■	□	□	□	□	□	□	0 8 9	✘
8	□	□	□	□	■	□	□	□	□	□	□	□	0 8 9	✘
Result					■									

Conclusion: $a'[2]$ being 0 (instead of 7) causes the failure.

A Simple Demo

(HOWCOME demonstrator at
http://www.st.cs.uni-sb.de/dd/)







Relevant State Differences

Can we apply this technique to more complex states, too?

Example: GCC state in the function *combine_instructions*

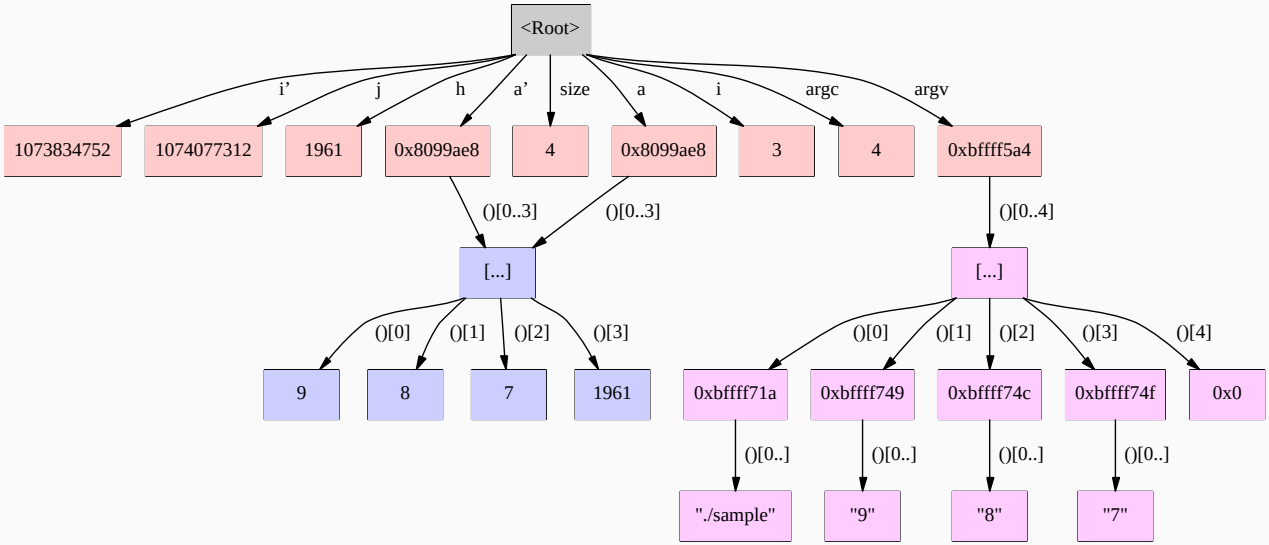
#	reg_rtx_no	cur_insn_uid	last_linenum	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✘
			?		
5	32	74	15	0x81fc4a0	✔
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✔
2	31	70	14	0x81fc4a0	✔

Consequence: determine and apply *structural differences!*

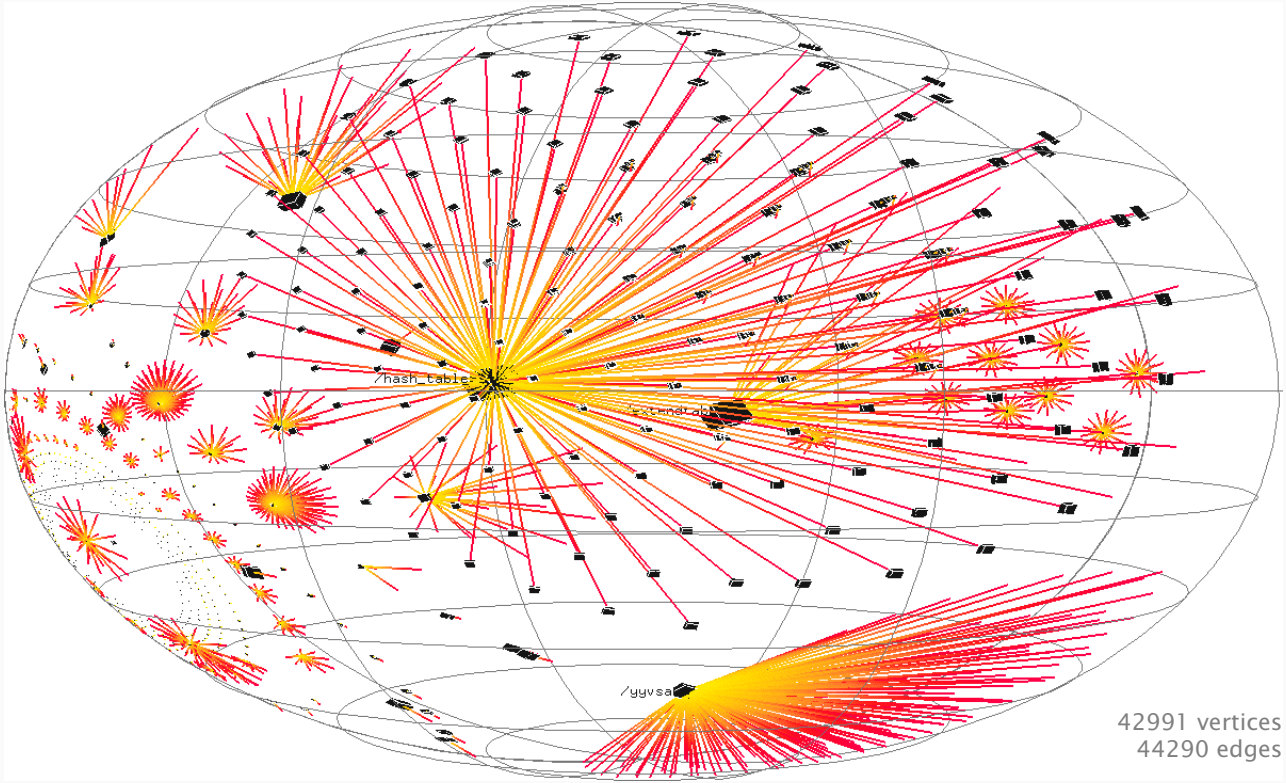
Memory Graphs

Basic idea: represent references (pointers) by edges in a graph.

Example – `sample` memory graph:

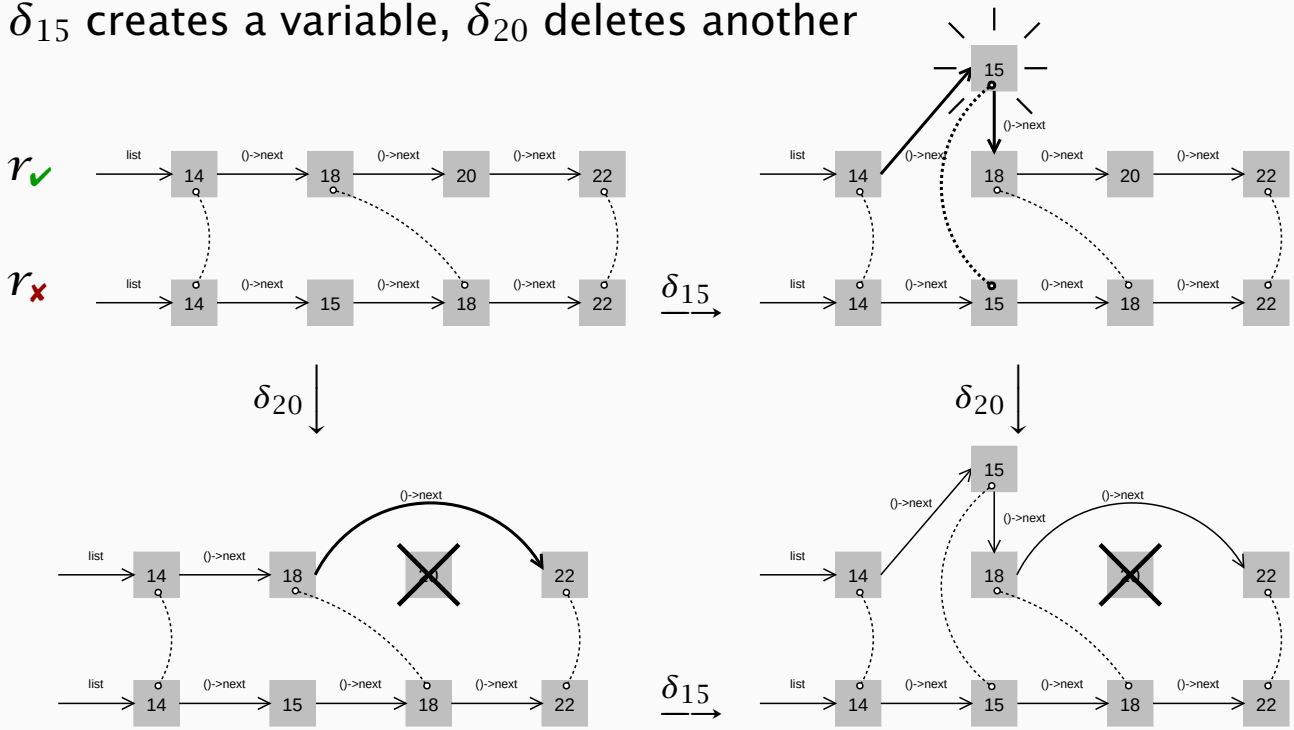


The GCC Memory Graph

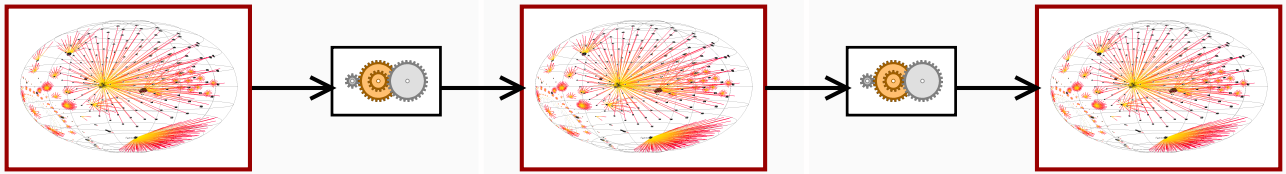


Structural Differences

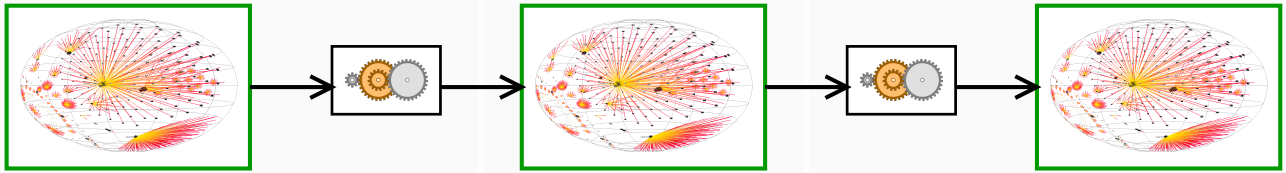
HOWCOME can compute structural graph differences:
δ_{15} creates a variable, δ_{20} deletes another



The Process in a Nutshell

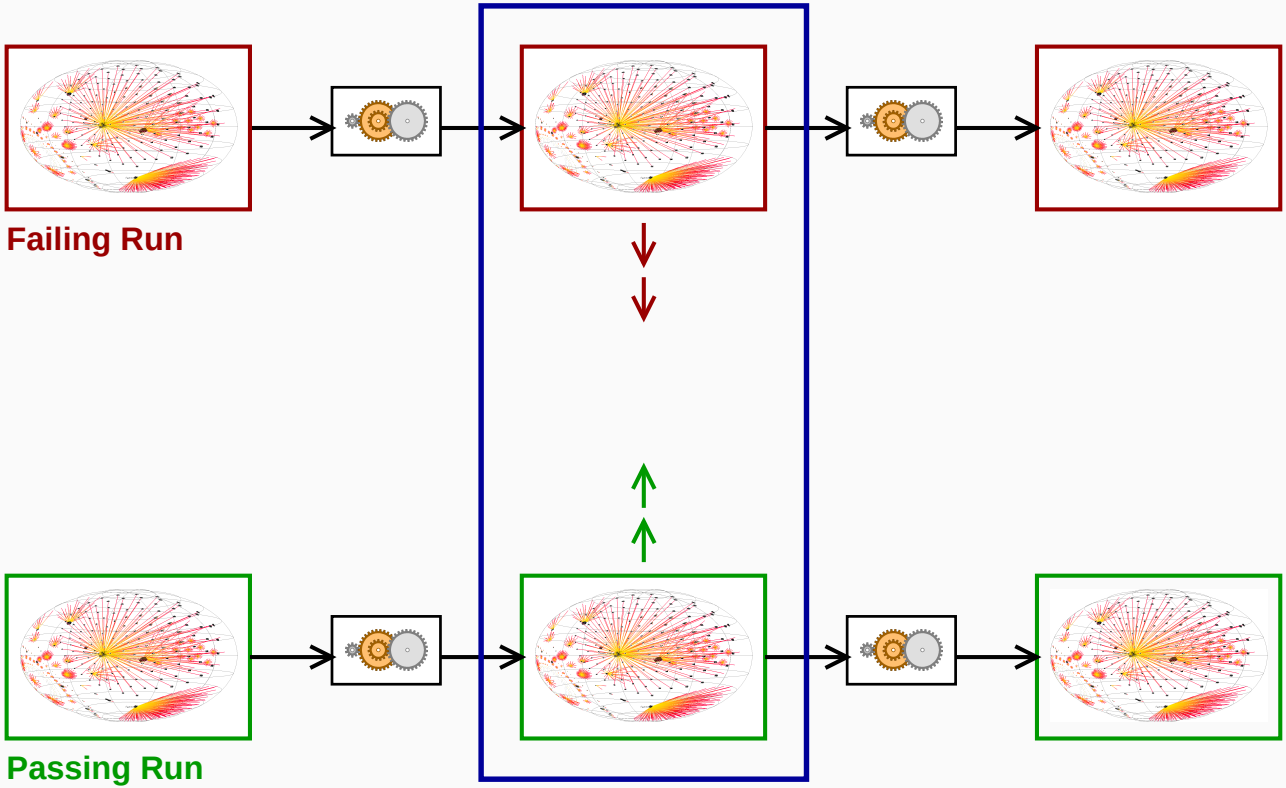


Failing Run

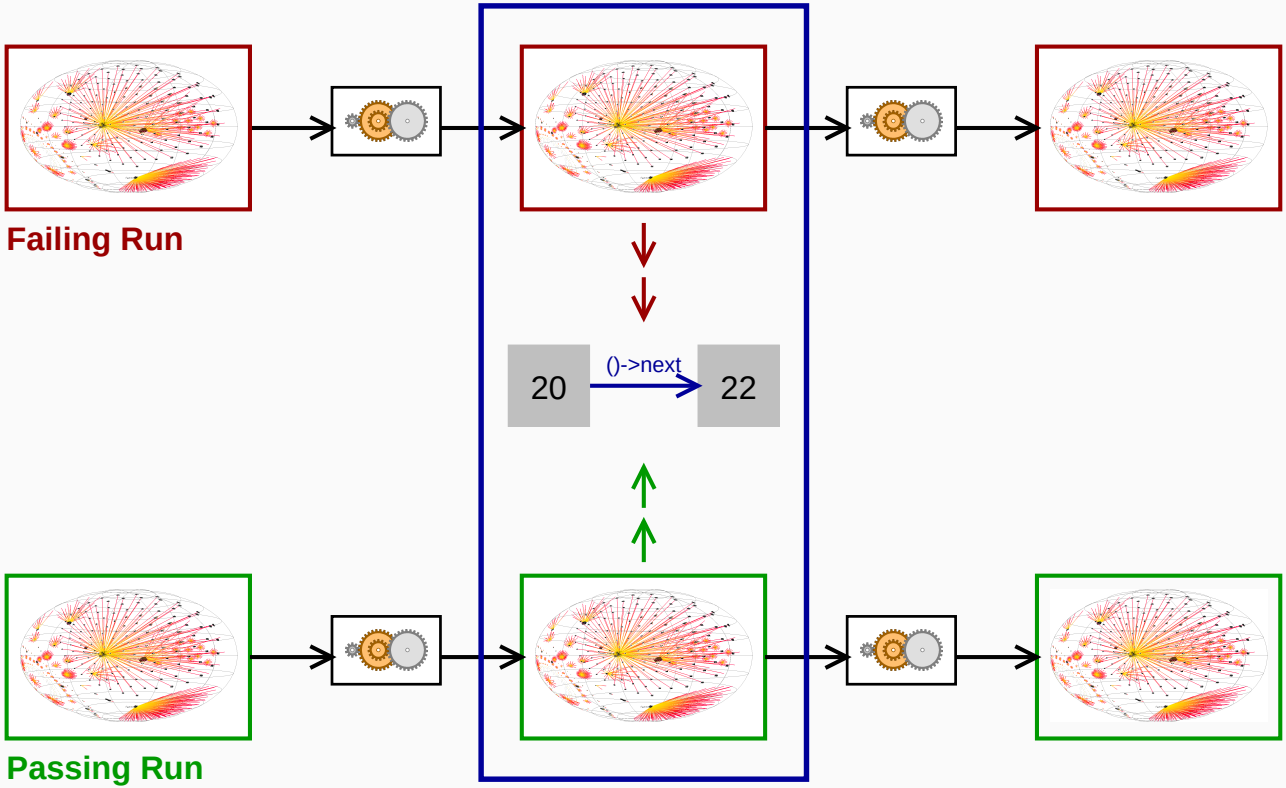


Passing Run

The Process in a Nutshell

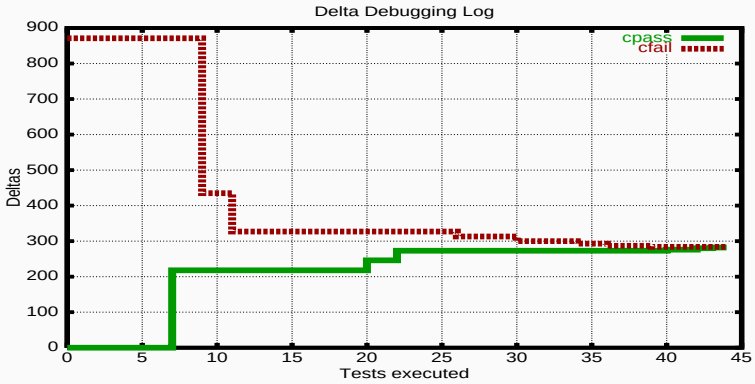


The Process in a Nutshell



Relevant State Differences

HOWCOME examines the state of cc1 in *combine_instructions*: 871 nodes (= variables) are different



Only one variable causes the failure:

```
$m = (struct rtx_def *)malloc(12)
$m->code = PLUS
first_loop_store_insn->fld[1]...rtx = $m
```

The GCC Cause-Effect Chain

After 59 tests, HOWCOME has determined these failure causes:

1. Execution reaches **main.**
   Since the program was invoked as "`cc1 -O fail.i`",
   variable **argv[2]** is now **"`fail.i`"**.

2. Execution reaches **combine_instructions.**
   Since argv[2] was "`fail.i`",
   variable ***first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx** is now ⟨**new rtx_def**⟩.

3. Execution reaches **if_then_else_cond (95th hit).**
   Since *first_loop_store_insn→fld[1].rtx→fld[1].rtx→fld[3].rtx→fld[1].rtx was ⟨new rtx_def⟩,
   variable **link→fld[0].rtx→fld[0].rtx** is now **link**.

4. Execution ends.
   Since variable link→fld[0].rtx→fld[0].rtx was link,
   the program now **terminates with a SIGSEGV signal**.
   The program fails.

Total running time: 6 seconds (+ 90 minutes of GDB overhead)

Challenges

How do we capture C program state accurately?

Does p point to something, and if so, to how many of them?

Today: Query memory allocation routines + heuristics

Future: Use program analysis, greater program state

How do we determine relevant events?

Why focus on, say, combine_instructions?

Today: Start with *backtrace* of failing run

Future: Focus on *anomalies + transitions*; user interaction

How do we know a failure is the failure?

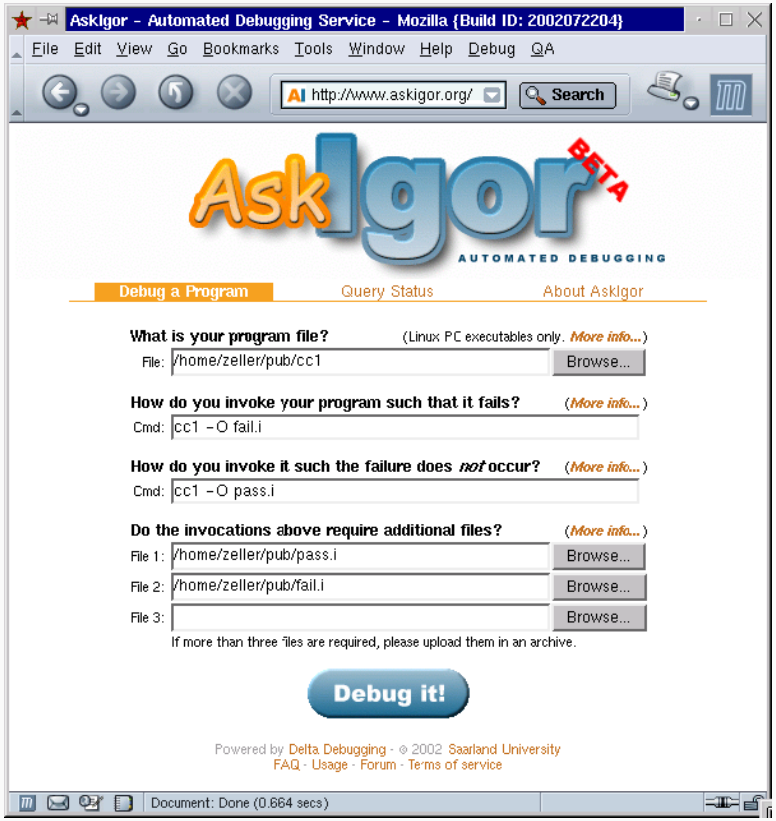
Can't our changes just induce new failures?

Today: Outcome is "original" only if backtraces match

Future: Also match output, time, code coverage

And finally: *When does this actually work?*

www.askigor.org



Submit buggy program

⇓

Specify invocations

⇓

Click on “Debug it”

⇓

Diagnosis comes via e-mail

Up and running since 2002-10-25

Concepts

- Applying delta debugging on program states leads to much *higher precision* than “classical” analysis.
- Memory graphs allow representing and comparing complex program states.
- Delta debugging requires a passing run as reference.
- Next lecture:
  - Extraction and comparison of memory graphs
  - Narrowing down infection sites
  - Limits and Drawbacks

References

- Andreas Zeller, *Isolating Cause-Effect Chains from Computer Programs*, Proc. FSE 2002, pp. 1–10 (2002) http://www.st.cs.uni-sb.de/papers/fse2002/
- —, earlier version, unpublished (includes `sample` example), http://www.st.cs.uni-sb.de/papers/icse2002/
- *HOWCOME demonstrator*, http://www.st.cs.uni-sb.de/dd/
- *Delta Debugging Web Site*, http://www.st.cs.uni-sb.de/dd/
- *AskIgor Web Site*, http://www.askigor.org/