SIGSOFT 2002/FSE-10, Charleston, November 20, 2002

Isolating Cause-Effect Chains

from Computer Programs

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

A True Story

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 (...) { ... } ... }	✘
2	double **bug**(...) { int i, j; $i = 0$; for (...) { ... } ... }	✔

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 (. . .) { . . . } . . . }	✘
3	double **bug**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	
2	double **bug**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	✔

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 (. . .) { . . . } . . . }	✗
3	double **bug**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	✓
2	double **bug**(. . .) { int i, j; $i = 0$; for (. . .) { . . . } . . . }	✓

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 (. . .) { . . . } . . . }	✘
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 (. . .) { . . . } . . . }	✔

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 (. . .) { . . . } . . . }	✗
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 (. . .) { . . . } . . . }	✓

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 (. . .) { . . . }. . . }	
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 (. . .) { . . . } . . . }	✔

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 (. . .) { . . . }. . . }	✗
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 (. . .) { . . . } . . . }	✓

Isolating Failure Causes

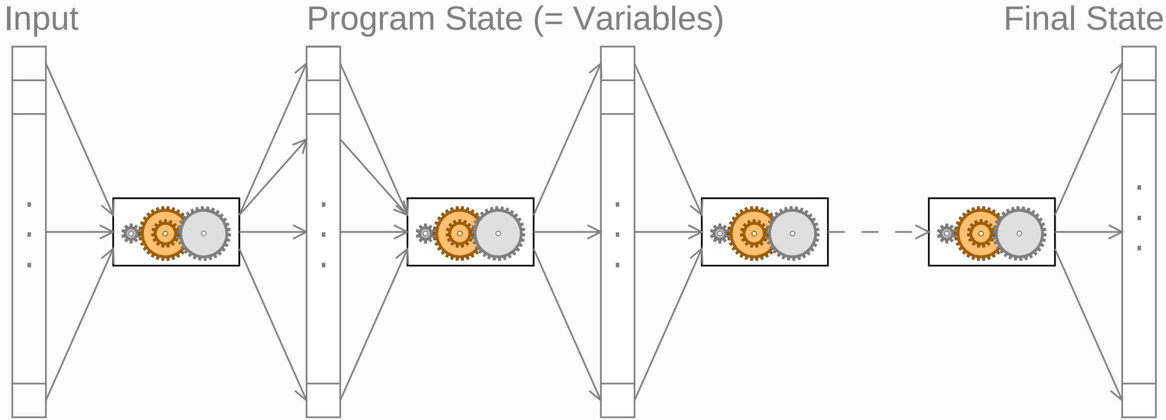
Delta Debugging automatically isolates the *failure-inducing difference* in the GCC input:

#	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)

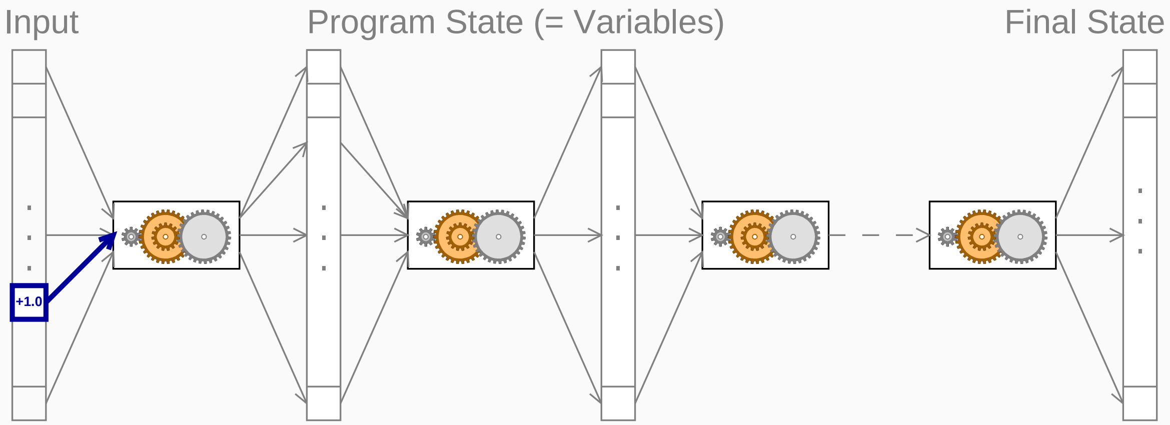
What's going on in GCC?

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



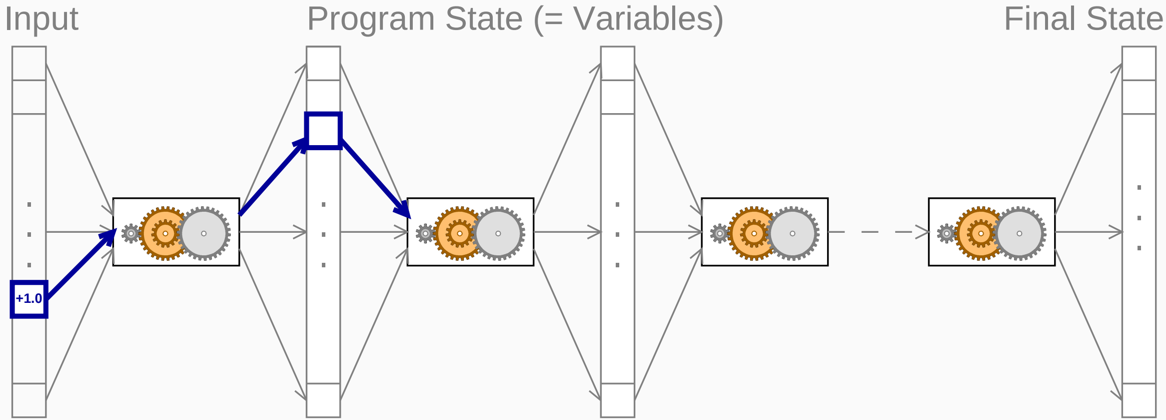
What’s going on in GCC?

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



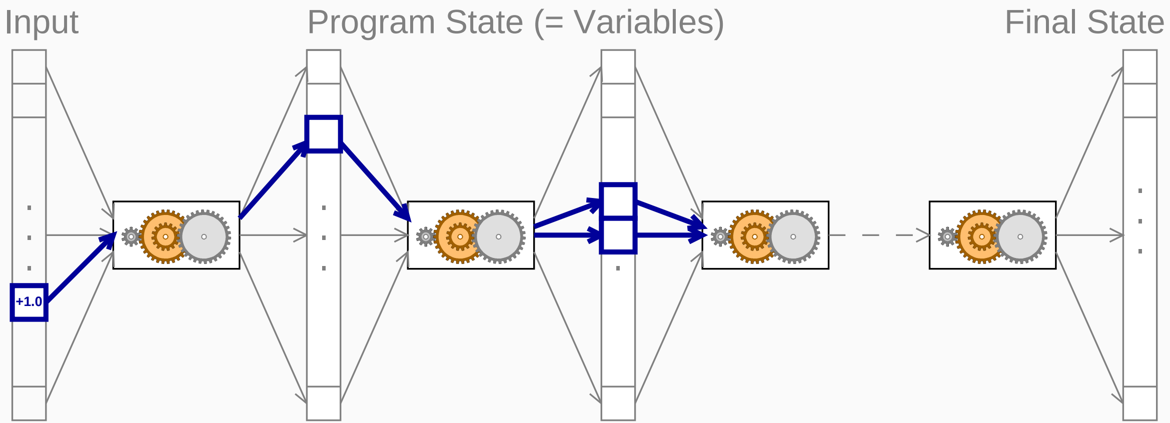
What’s going on in GCC?

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



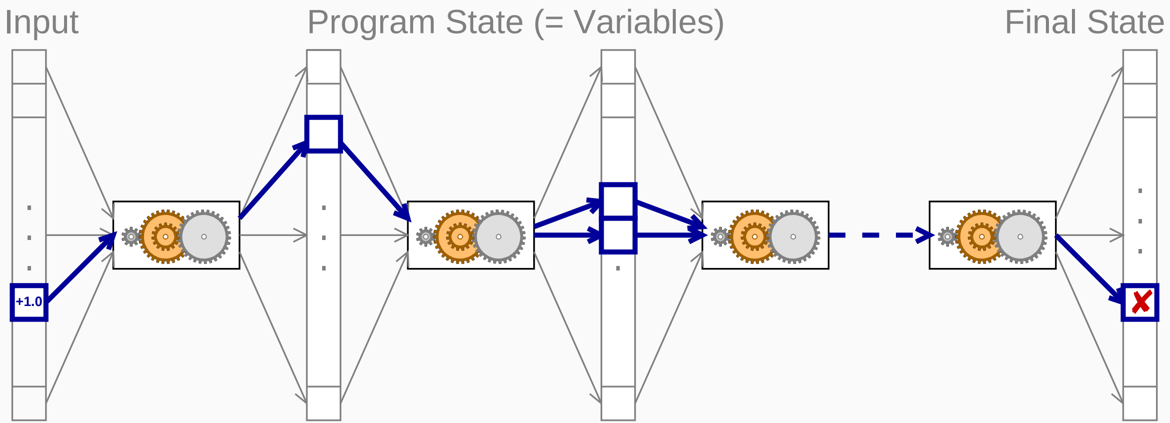
What's going on in GCC?

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



What's going on in GCC?

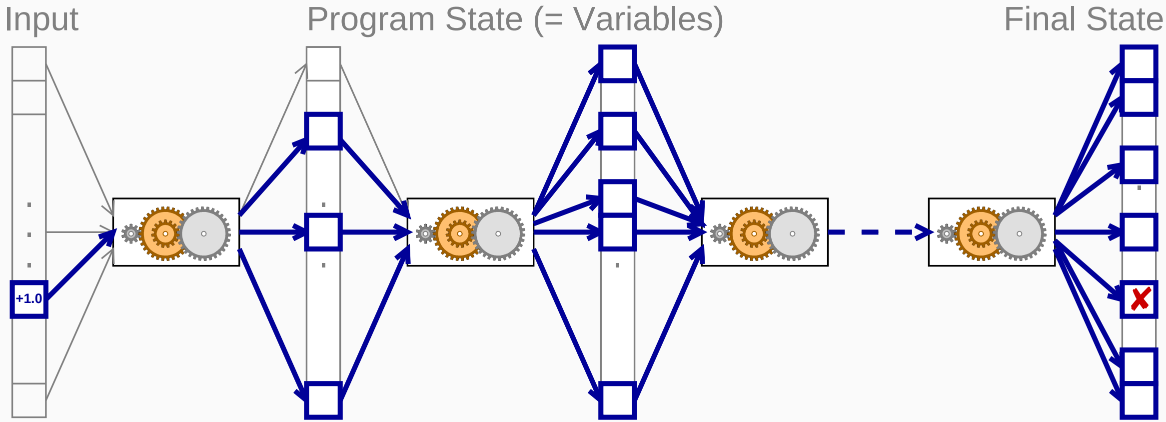
The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



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

Comparing States

Comparing states does not work,
because the differences *accumulate* during execution:



How do we isolate the *relevant* state differences?

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗
2	31	70	14	0x81fc4a0	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗
3	32	74	14	0x81fc4a0	
2	31	70	14	0x81fc4a0	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗
3	32	74	14	0x81fc4a0	✓
2	31	70	14	0x81fc4a0	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗
4	32	74	14	0x81fc4e4	
3	32	74	14	0x81fc4a0	✓
2	31	70	14	0x81fc4a0	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✗
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✓
2	31	70	14	0x81fc4a0	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

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	✓

Relevant State Differences

Using a debugger (GDB), we can examine and alter the program state at various events during a program run.

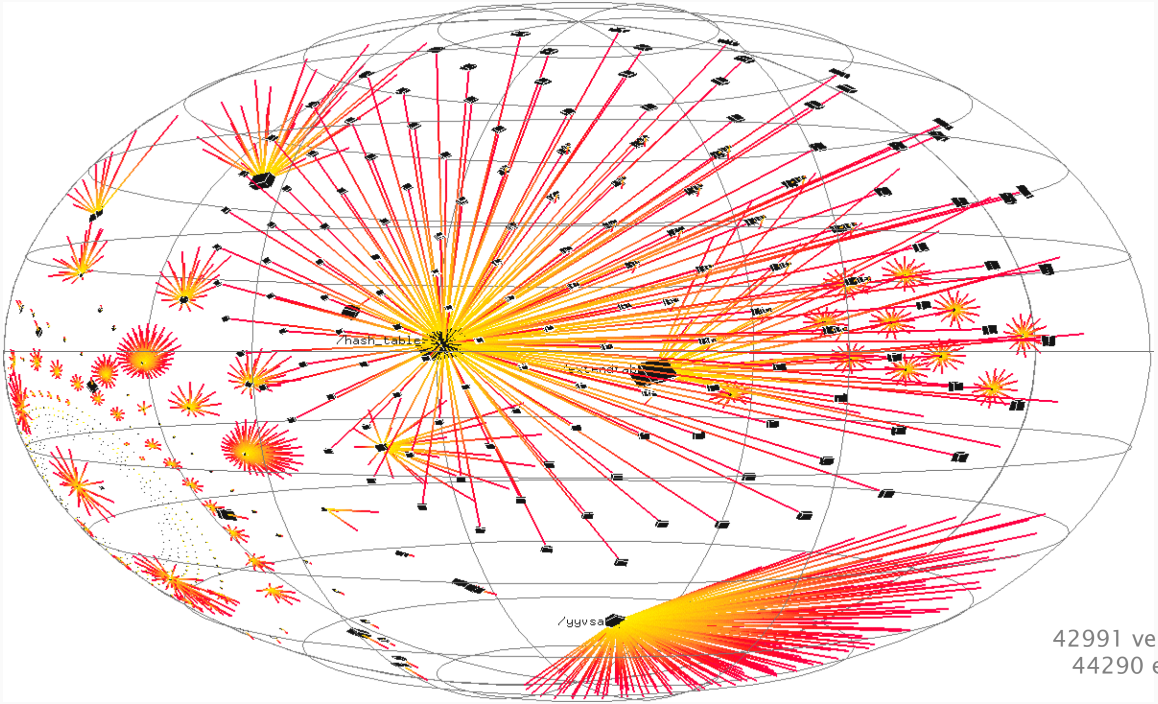
Example: GCC state in the function *combine_instructions*

#	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!*

The GCC Memory Graph

Our HOWCOME prototype extracts the program state as *graph*:
Vertices are *variables*, edges are *references*

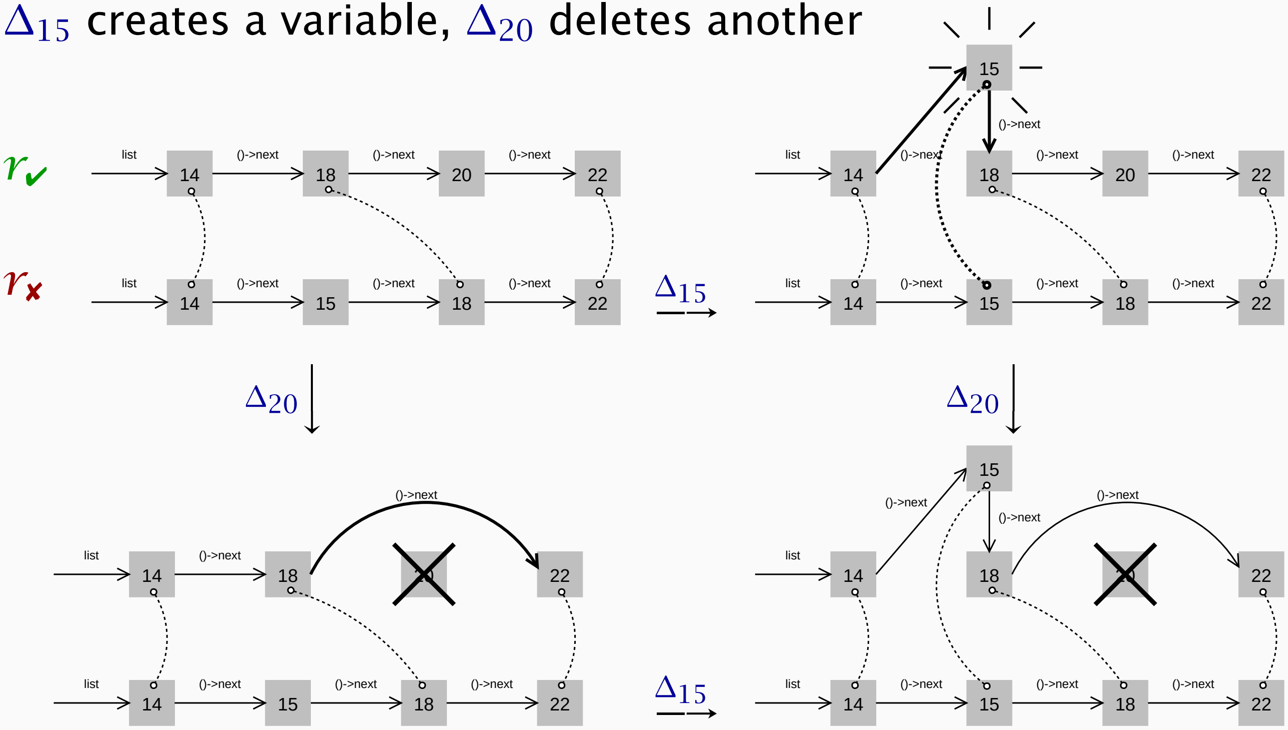


42991 vertices
44290 edges

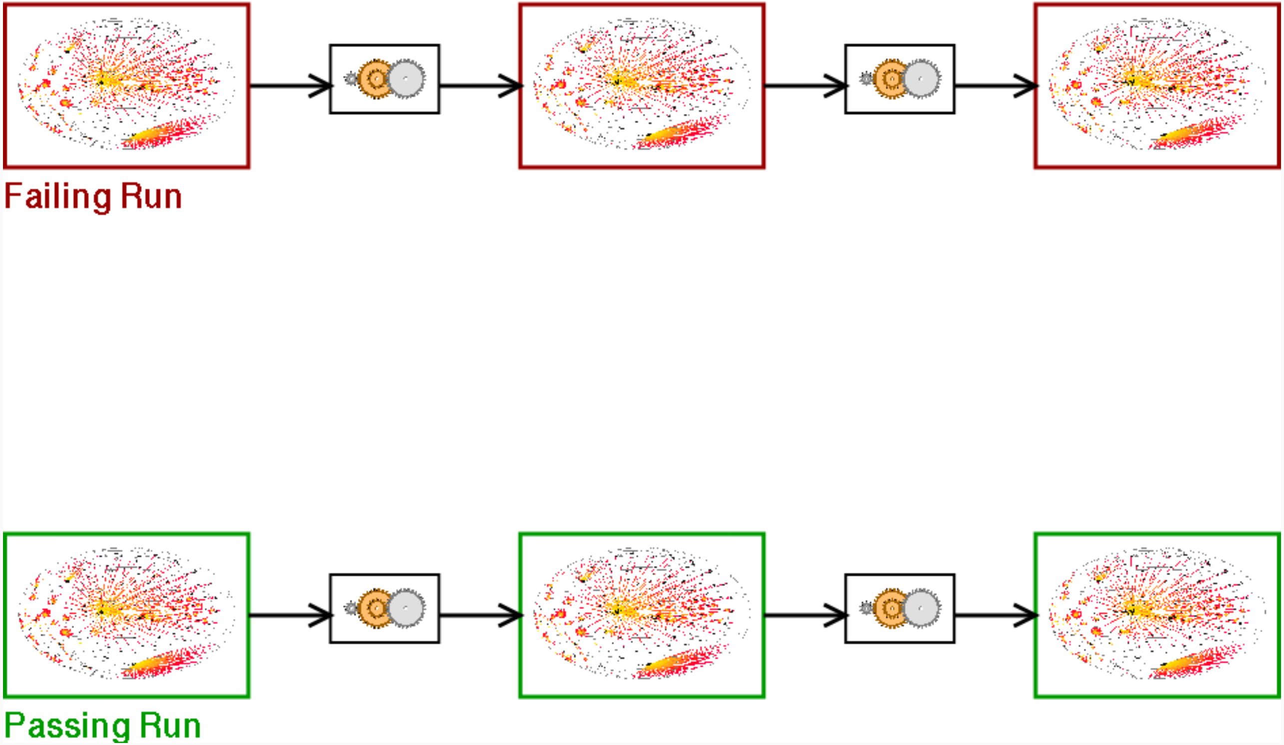
Structural Differences

HOWCOME can compute structural graph differences:

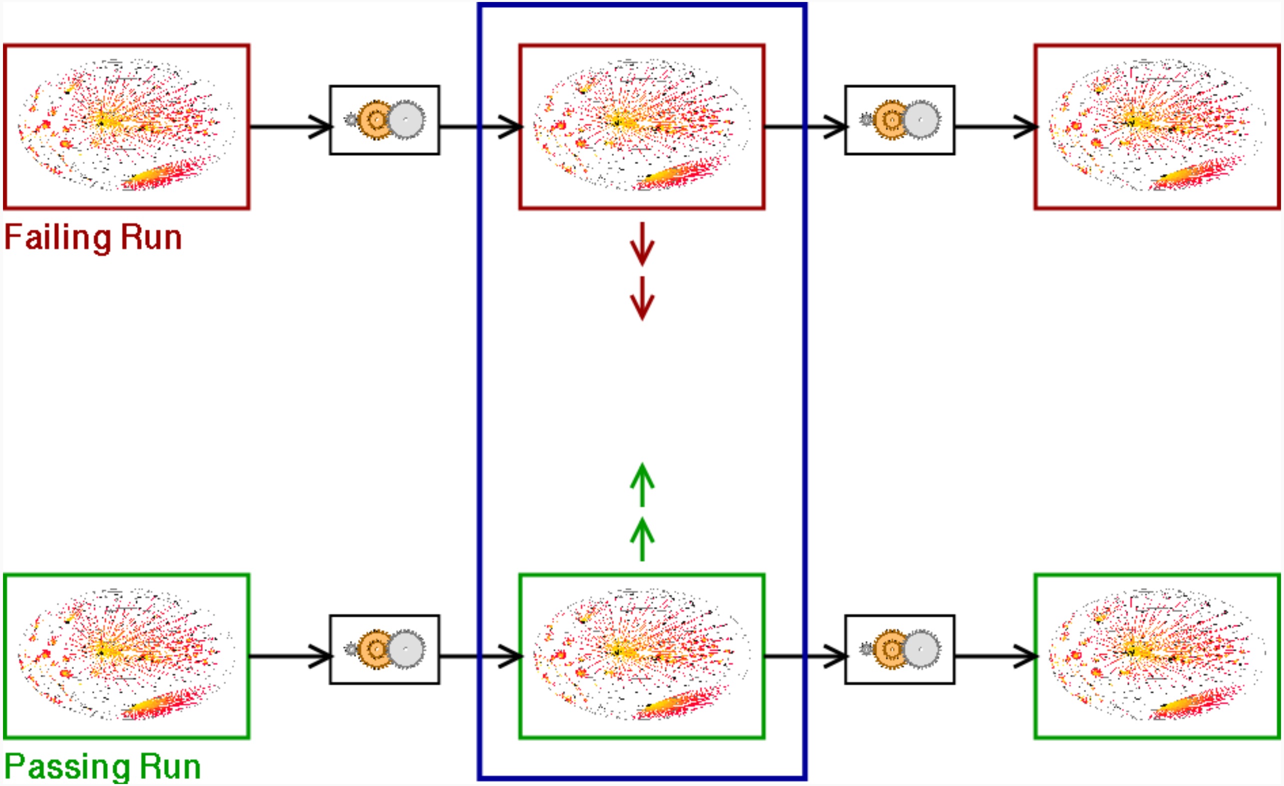
Δ_{15} creates a variable, Δ_{20} deletes another



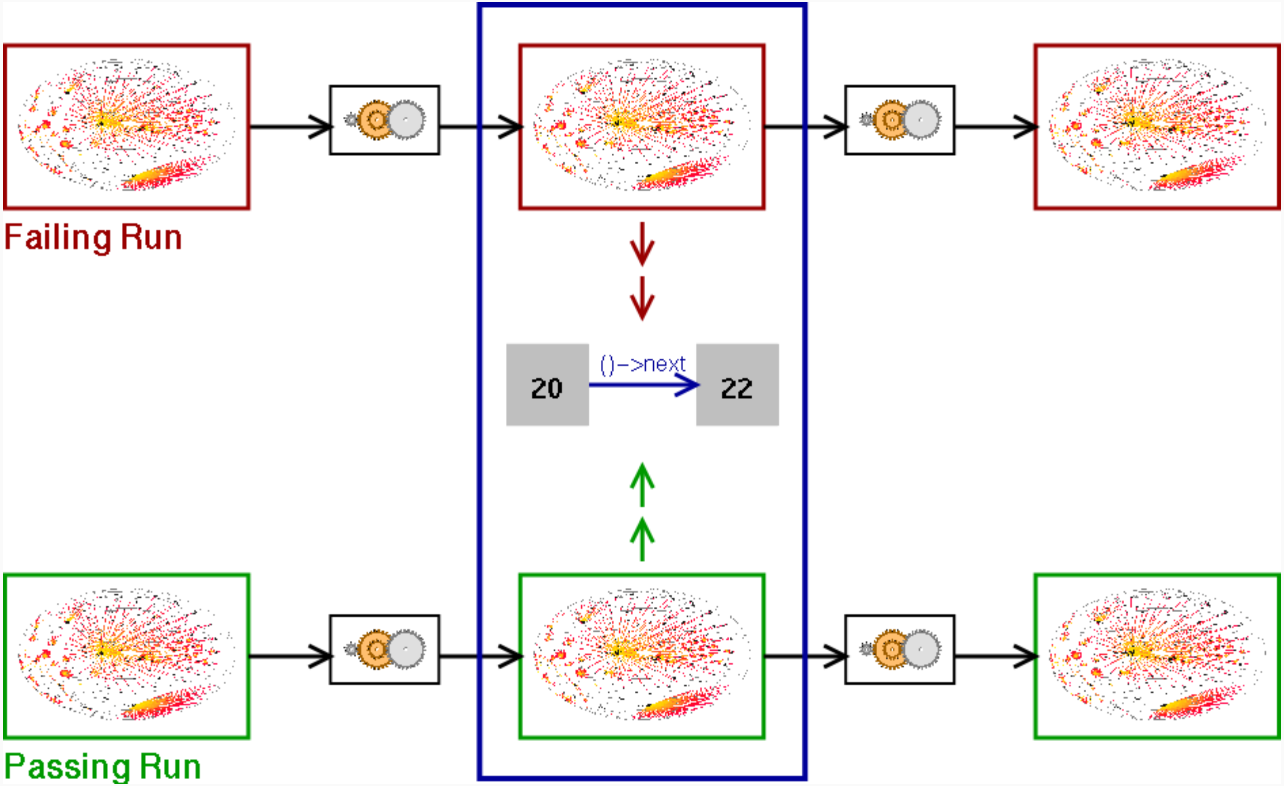
The Process in a Nutshell



The Process in a Nutshell



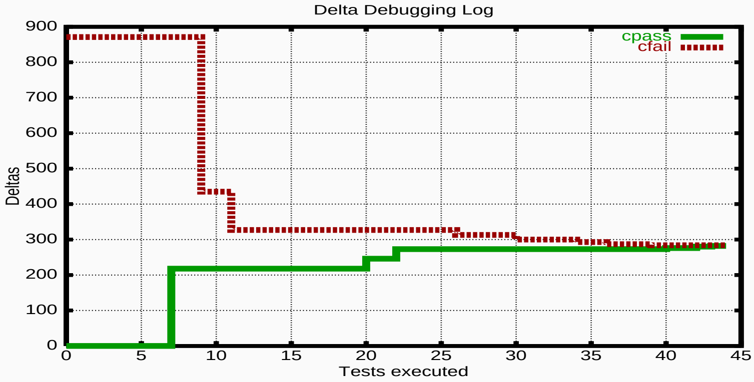
The Process in a Nutshell



Relevant State Differences

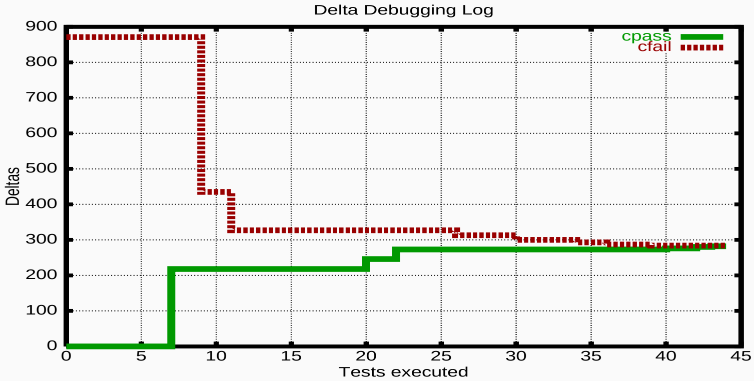
HOWCOME examines the state of cc1 in *combine_instructions*:
871 nodes (= variables) are different

Delta Debugging Log



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`”**.

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**⟩.

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.**

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

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

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

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

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?

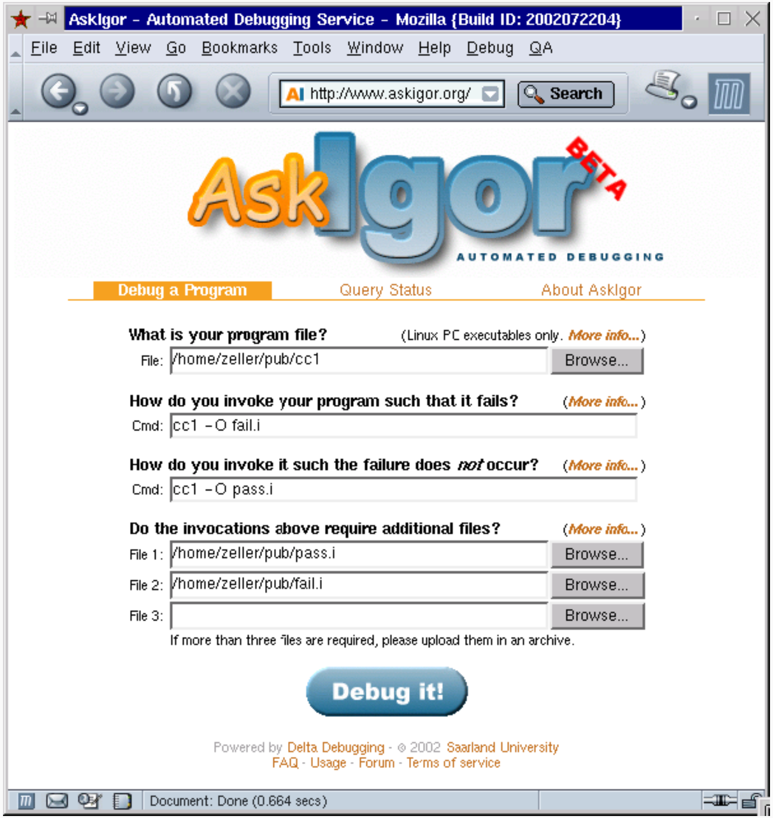
Can't our changes just induce new failures?

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



What is your program file? (Linux PC executables only. *More info...*)

File: /home/zeller/pub/cc1

How do you invoke your program such that it fails? (*More info...*)

Cmd: cc1 -O fail.i

How do you invoke it such the failure does *not* occur? (*More info...*)

Cmd: cc1 -O pass.i

Do the invocations above require additional files? (*More info...*)

File 1: /home/zeller/pub/pass.i

File 2: /home/zeller/pub/fail.i

File 3:

If more than three files are required, please upload them in an archive.

Powered by Delta Debugging · © 2002 Saarland University
FAQ · Usage · Forum · Terms of service

Document: Done (0.664 secs)

Submit buggy program

⇓

Specify invocations

⇓

Click on “Debug it”

⇓

Diagnosis comes
via e-mail

Up and running
since 2002-10-25

Conclusion

- ✔ Cause-effect chains explain the causes of program failures *automatically* and *effectively.*
- ✔ Systematic *experimentation* leads to much *higher precision* than “classical” analysis.
- ✔ Via automation, debugging becomes a *well-understood, systematic discipline.*

- ✘ We need *a passing execution* as a reference.
- ✘ *Large testing costs* can be prohibitive.
- ✘ *Preventing bugs* is still an issue!

`http://www.askigor.org/`

Read More

Automated Debugging. Morgan Kaufmann Publishers, Summer 2003.

Isolating Cause-Effect Chains from Computer Programs. Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2002), Charleston, Nov. 2002.

Isolating Failure-Inducing Thread Schedules. (w/ J.-D. Choi) Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002), Rom, July 2002.

Simplifying and Isolating Failure-Inducing Input. (w/ R. Hildebrandt) IEEE Transactions on Software Engineering 28(2), February 2002, pp. 183–200.

Automated Debugging: Are We Close? IEEE Computer, Nov. 2001, pp. 26–31.

Visualizing Memory Graphs. (w/ T. Zimmermann) Proc. of the Dagstuhl Seminar 01211 "'Software Visualization"', May 2001. LNCS 2269, pp. 191–204.

Simplifying Failure-Inducing Input. (w/ R. Hildebrandt) Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2000), Portland, Oregon, August 2000, pp. 135-145.

Yesterday, my program worked. Today, it does not. Why? Proc. ACM SIGSOFT Conference (ESEC/FSE 1999), Toulouse, Sep. 1999, LNCS 1687, pp. 253–267.

`http://www.askigor.org/`