



Microsoft Research, Redmond, Washington, 2003-10-13

Why does my Program fail?

Causes and effects in 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];  
}
```

Compiling `bug.c`, the GNU compiler (GCC) crashes:

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

What's the error that causes this failure?



Errors



What's the error in GCC?

An **error** is a deviation from what's correct, right, or true. — IEEE Standard Glossary of SE Terminology

To prove that something is an error, we must *show the deviation*:

- *simple* for the failure in question
- *hard* for the program code

General technique: *Deduction*—reasoning from the abstract (code) to the concrete (run): static analysis, verification, ...

Where does GCC deviate from—what?



Causes



What's the cause for the GCC failure?

*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 that

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

General technique: *Experimentation*—constructing a *theory* from a series of experiments (runs)

Can't we automate experimentation?





Isolating Failure Causes

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

#	GCC input	test
1	<code>double bug(...) { int <i>i, j</i>; <i>i</i> = 0; for (...) { ... } ... }</code>	✗
2	<code>double bug(...) { int <i>i, j</i>; <i>i</i> = 0; for (...) { ... } ... }</code>	✓





Isolating Failure Causes

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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓





Isolating Failure Causes

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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓



Isolating Failure Causes



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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
4	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓



Isolating Failure Causes



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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✘
4	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✔
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✔
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✔





Isolating Failure Causes

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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
5	<code>double bug(...) { int i, j; i = 0; for (...) { ... }... }</code>	
4	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓



Isolating Failure Causes



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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
5	<code>double bug(...) { int i, j; i = 0; for (...) { ... }... }</code>	✗
4	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓





Isolating Failure Causes

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

#	GCC input	test
1	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
5	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✗
⋮		
19	<code>... z[i] = z[i] * (z[0] + 1.0); ...</code>	✗
18	<code>... z[i] = z[i] * (z[0] + 1.0); ...</code>	✓
⋮		
4	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
3	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓
2	<code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code>	✓

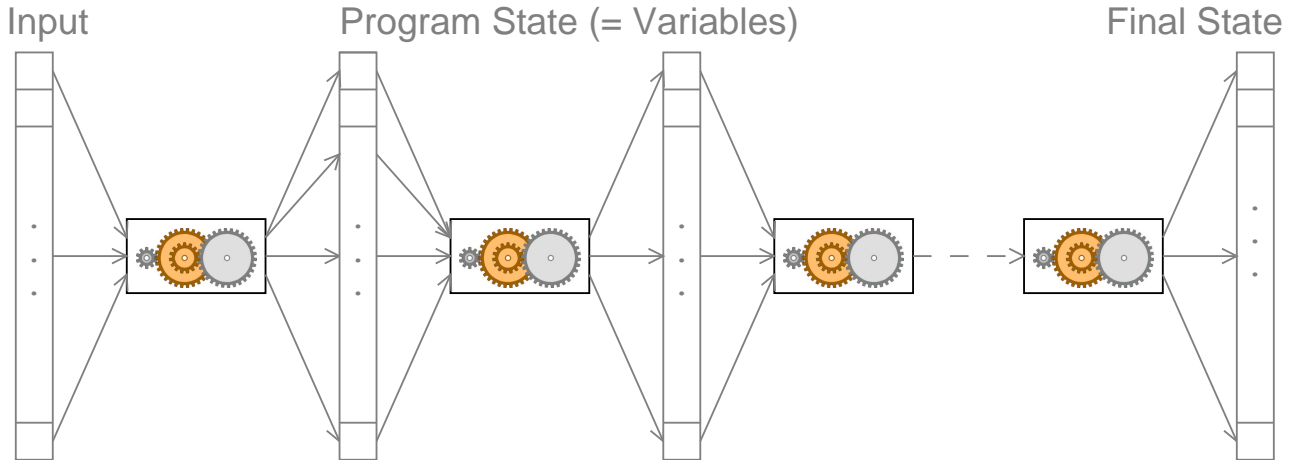
+ 1.0 is the failure cause – after only 19 tests (\approx 2 seconds).



What's going on in GCC?



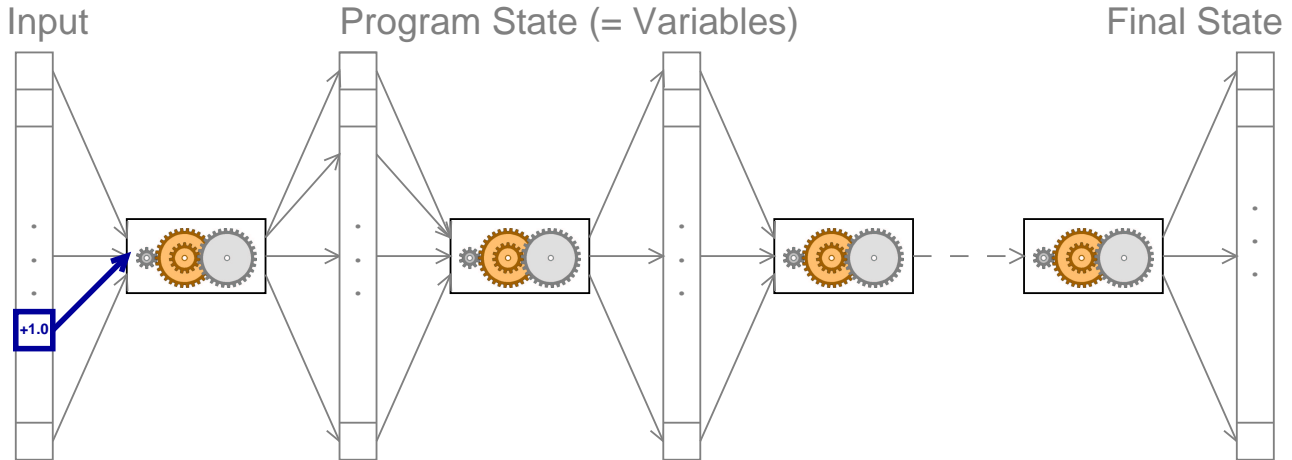
The difference + 1.0 is just the beginning of a *cause-effect chain* within the GCC run.



What's going on in GCC?



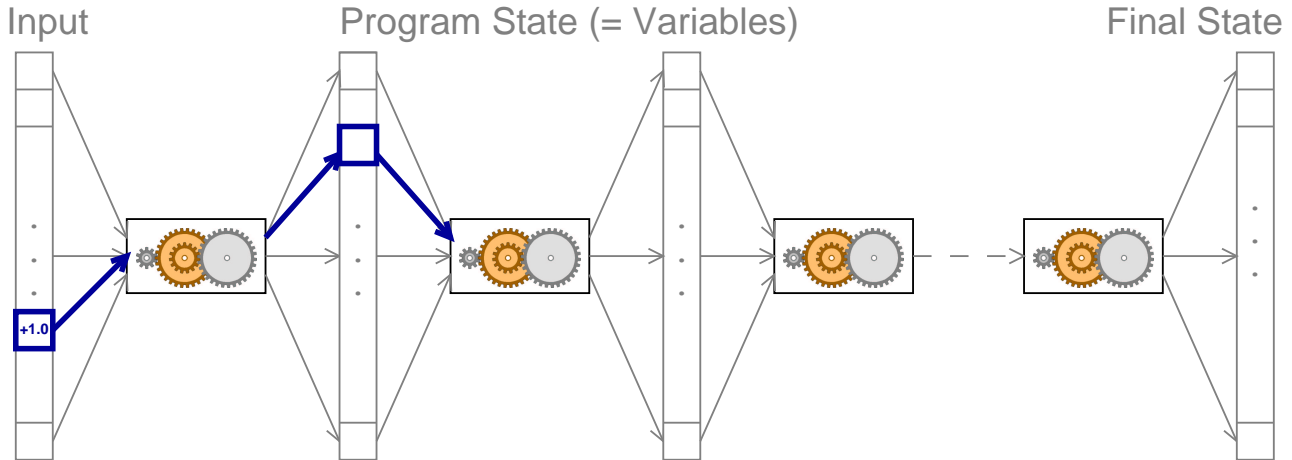
The difference `+ 1.0` is just the beginning of a *cause-effect chain* within the GCC run.



What's going on in GCC?



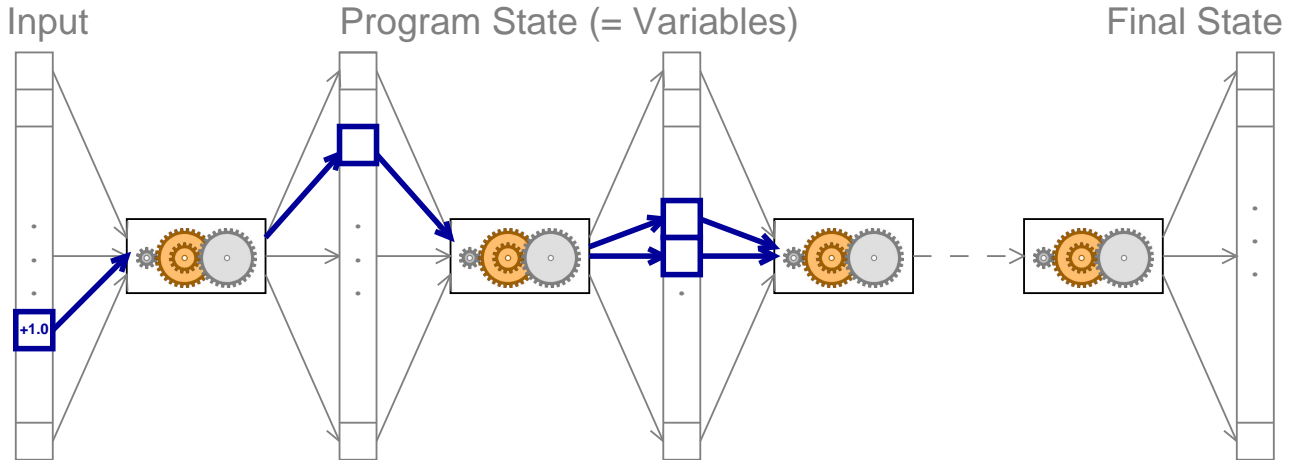
The difference `+ 1.0` is just the beginning of a *cause-effect chain* within the GCC run.



What's going on in GCC?



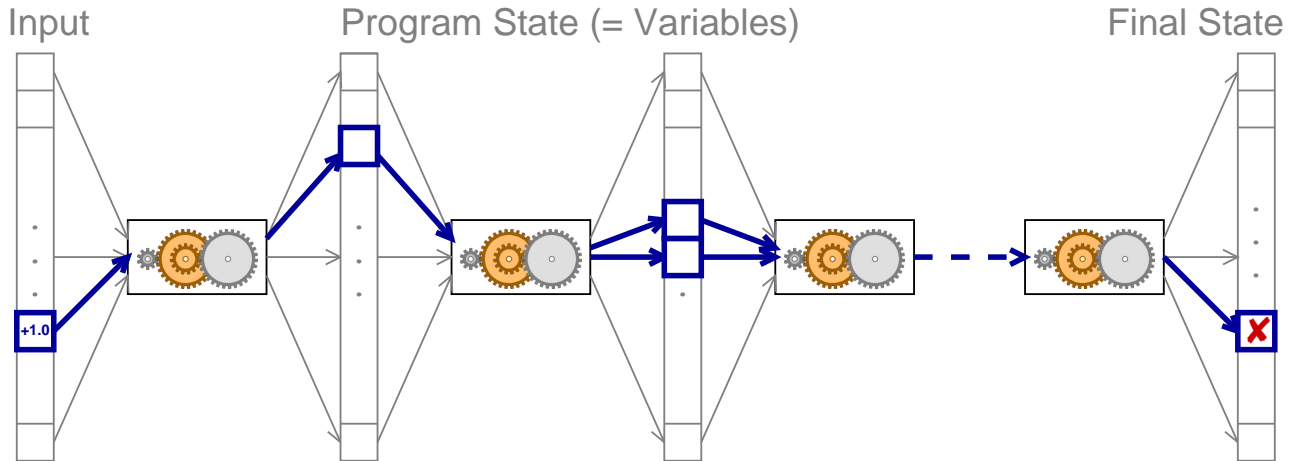
The difference $+1.0$ is just the beginning of a *cause-effect chain* within the GCC run.



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 failure, we must *break* this cause-effect chain.



Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics \Rightarrow OK for small, isolated, managed programs.

But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*





Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics \Rightarrow OK for small, isolated, managed 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];  
    }  
}
```





Tracing Data Flow

Classical *program analysis* traces how data propagates in programs.

Requires complete knowledge about entire code and its semantics \Rightarrow OK for small, isolated, managed programs.

But: Real programs are *opaque, parallel, distributed, dynamic, multilingual*—or simply obscure:

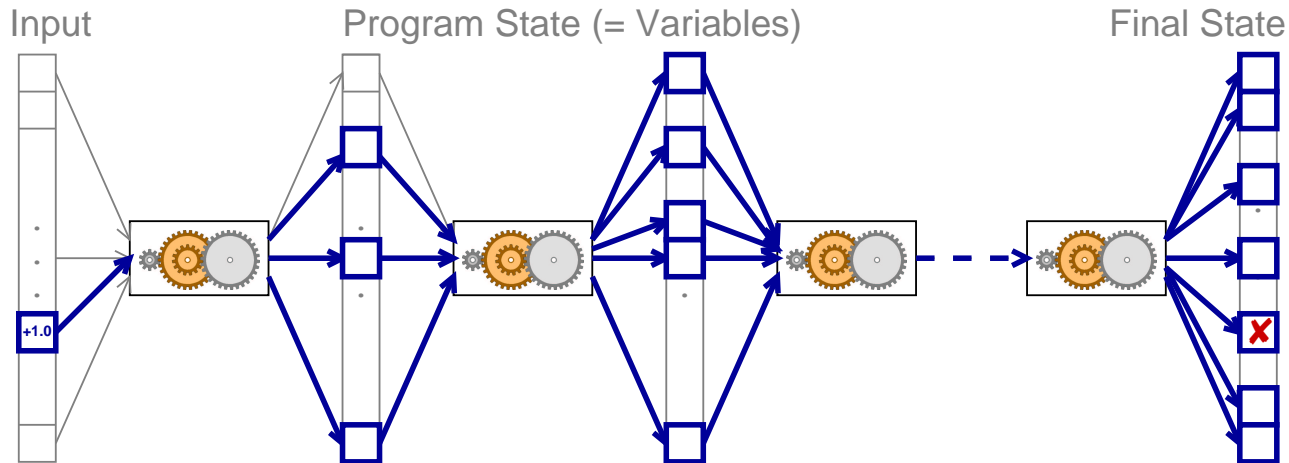
```
struct foo {           // Allocate string
    int tp, len;       int len = 200;
    union {           int bytes = len + 2 * sizeof(int);
        char    c[1];  foo *x = (foo *)malloc(bytes);
        int     i[1];  x->tp = STRING;
        double  d[1];  x->len = len;
    }                 strncpy(x->c, "Some string", len);
}                    }
```



Small Cause, Big Effect



Another problem—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_lineno	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	X





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_lineno	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_lineno	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_lineno	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*

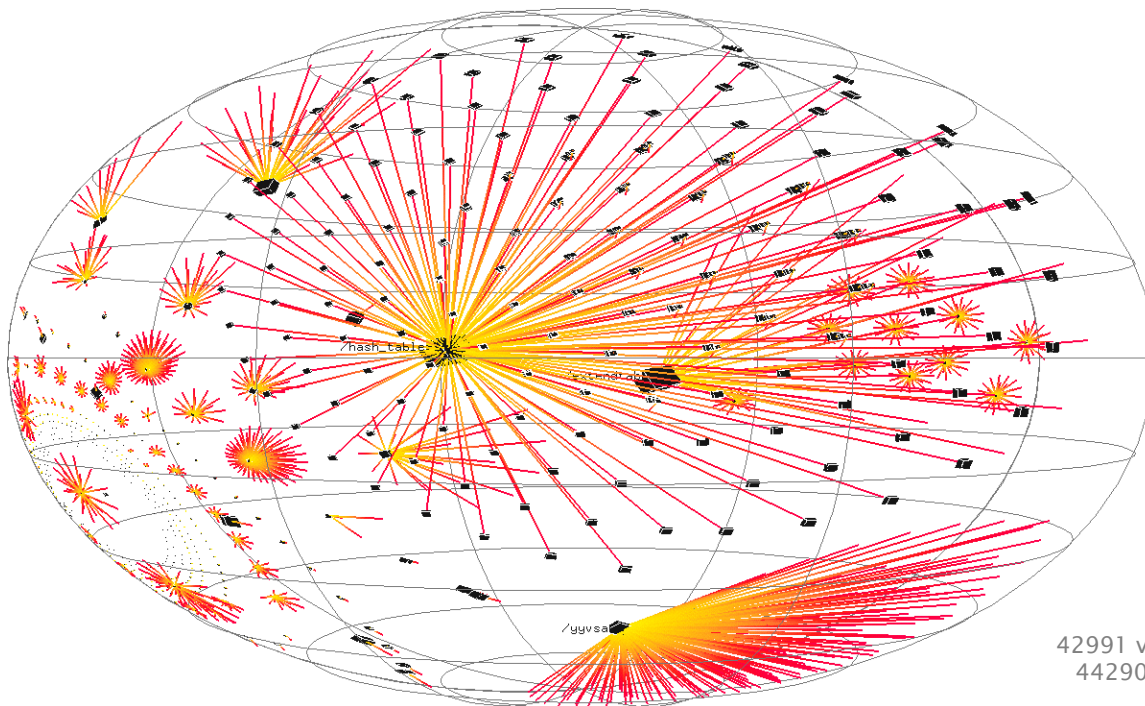
#	reg_rtx_no	cur_insn_uid	last_lineno	first_loop_store_insn	test
1	32	74	15	0x81fc4e4	✗
			?		
5	32	74	15	0x81fc4a0	✓
4	32	74	14	0x81fc4e4	?
3	32	74	14	0x81fc4a0	✓
2	31	70	14	0x81fc4a0	✓

Consequence: determine and apply *structural differences!*



The GCC Memory Graph

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



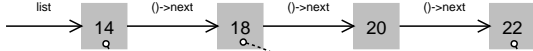
Structural Differences



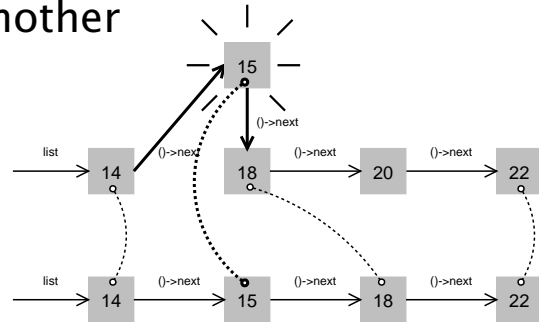
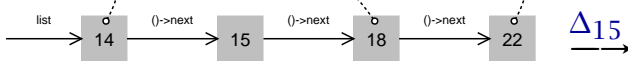
IGOR can compute structural graph differences:

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

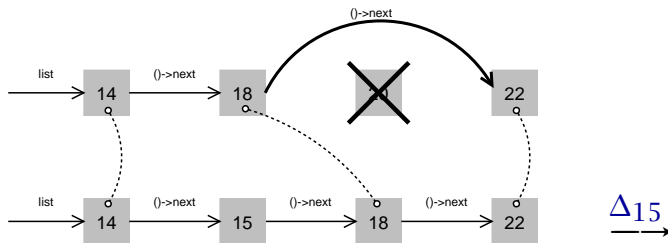
r_v



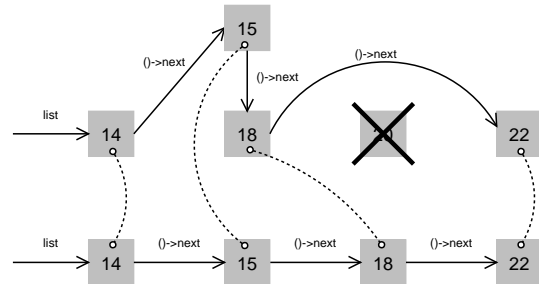
r_x



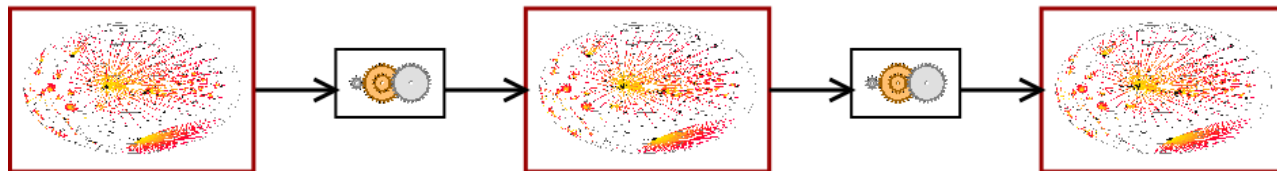
Δ_{20}



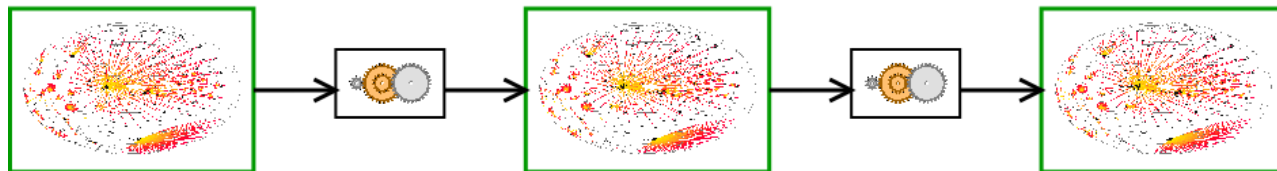
Δ_{20}



The Process in a Nutshell



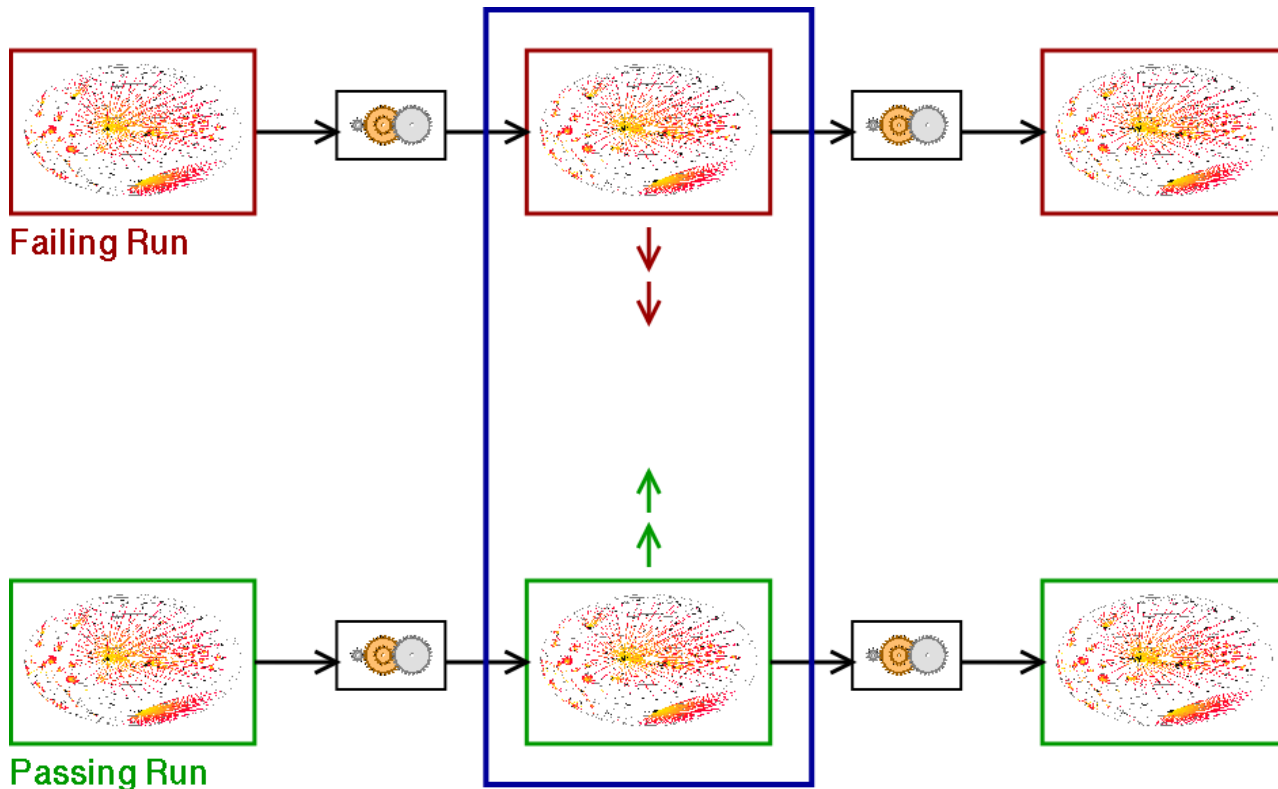
Failing Run



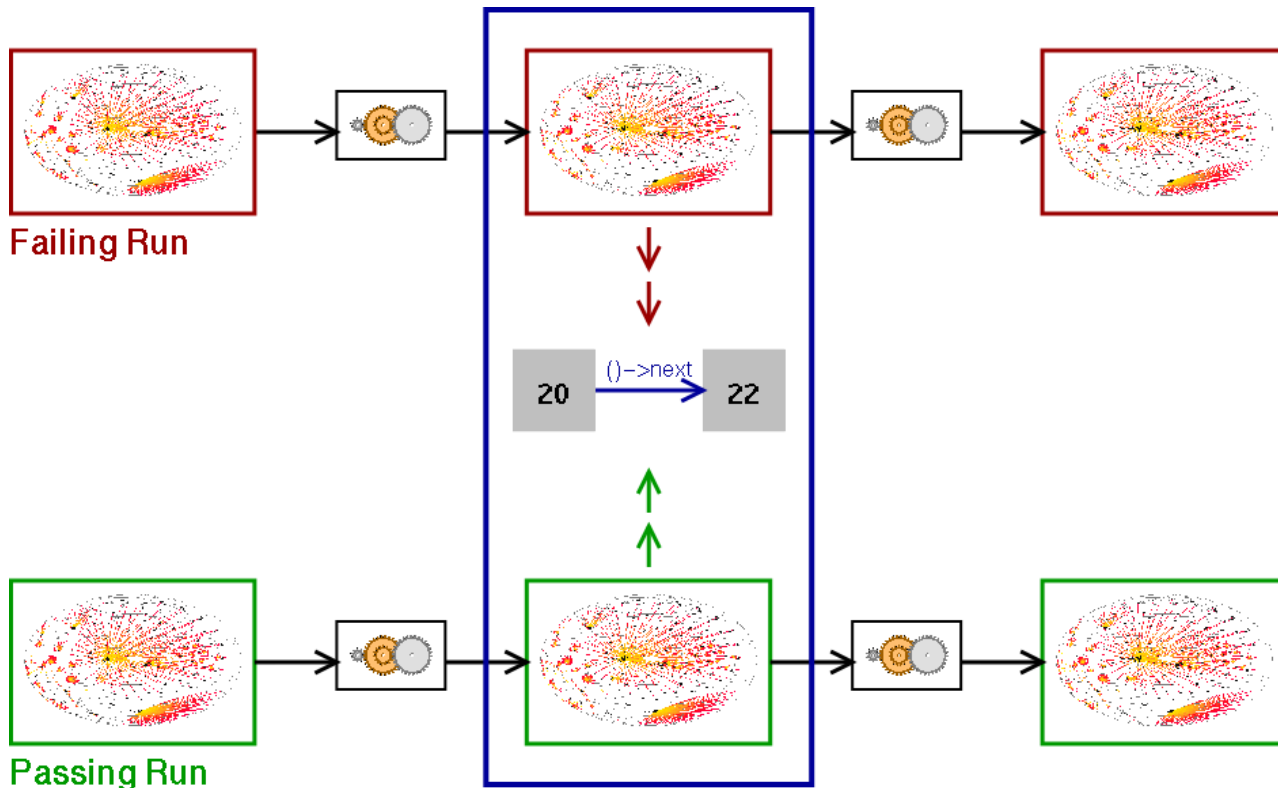
Passing Run



The Process in a Nutshell



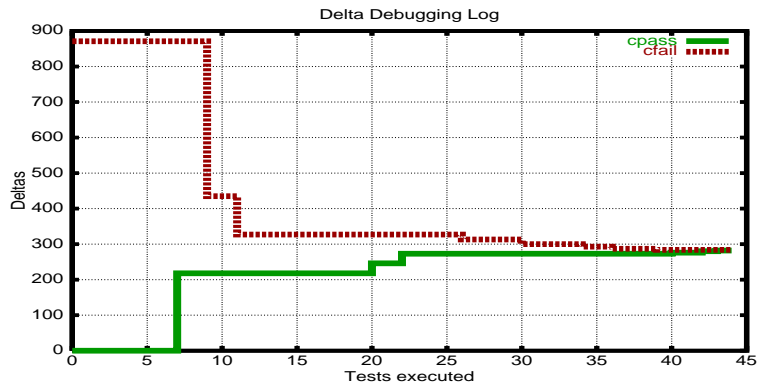
The Process in a Nutshell



Relevant State Differences



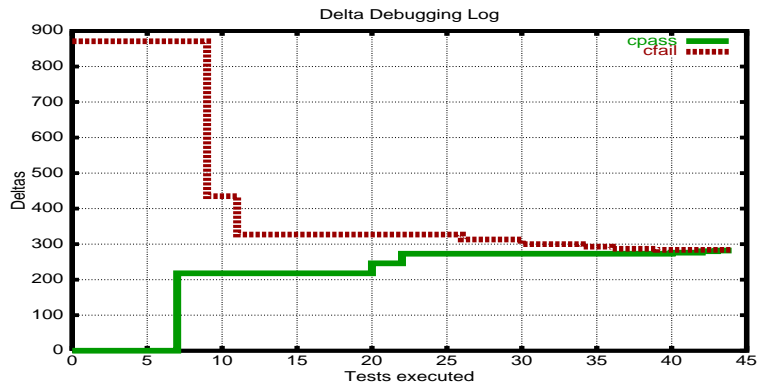
IGOR examines the state of `cc1` in *combine_instructions*:
871 nodes (= variables) are different



Relevant State Differences



IGOR 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



13/22

After 59 tests, IGOR has determined these failure causes:

1. Execution reaches **main**.

Since the program was invoked as “cc1 -0 fail.i”,
variable **argv[2]** is now “**fail.i**”.



The GCC Cause-Effect Chain



After 59 tests, IGOR 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, IGOR 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, IGOR 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, IGOR 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)



Causes vs. Errors

Every failure is caused by some error. But where is the error?

Deduction finds errors—but to prove that some error causes a given failure requires a *fix*.

Where's the technology that fixes errors?





Causes vs. Errors

Every failure is caused by some error. But where is the error?

Deduction finds errors—but to prove that some error causes a given failure requires a *fix*.

Where's the technology that fixes errors?

Experimentation finds causes—but to prove that some failure cause is an error requires a *full specification*.

Without specification, there are no errors—only surprises.





Causes vs. Errors

Every failure is caused by some error. But where is the error?

Deduction finds errors—but to prove that some error causes a given failure requires a *fix*.

Where's the technology that fixes errors?

Experimentation finds causes—but to prove that some failure cause is an error requires a *full specification*.

Without specification, there are no errors—only surprises.

You don't know you found the error until it's fixed:

- Absence of failure proves that the error caused the failure
- The fixed version is (hopefully) correct, right, and true





Isolating the Error

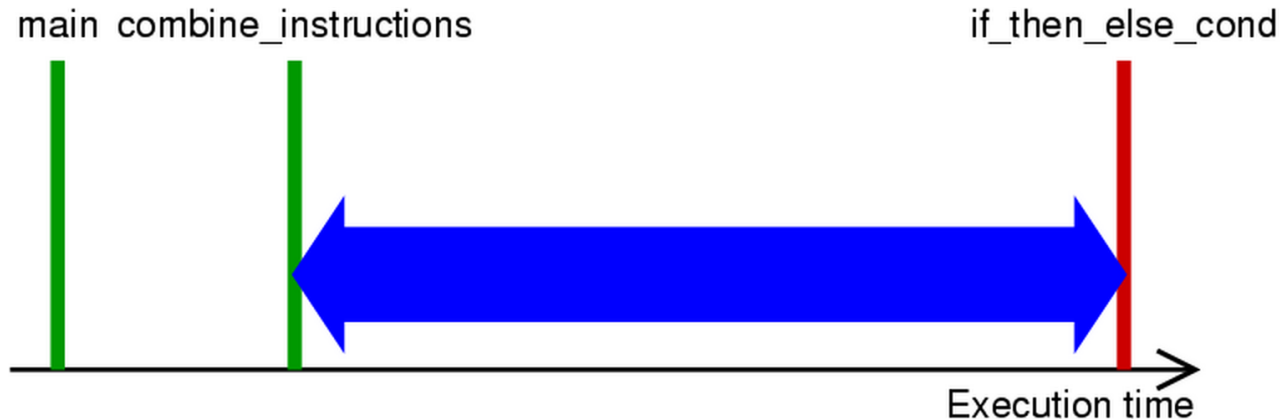
We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.





Isolating the Error

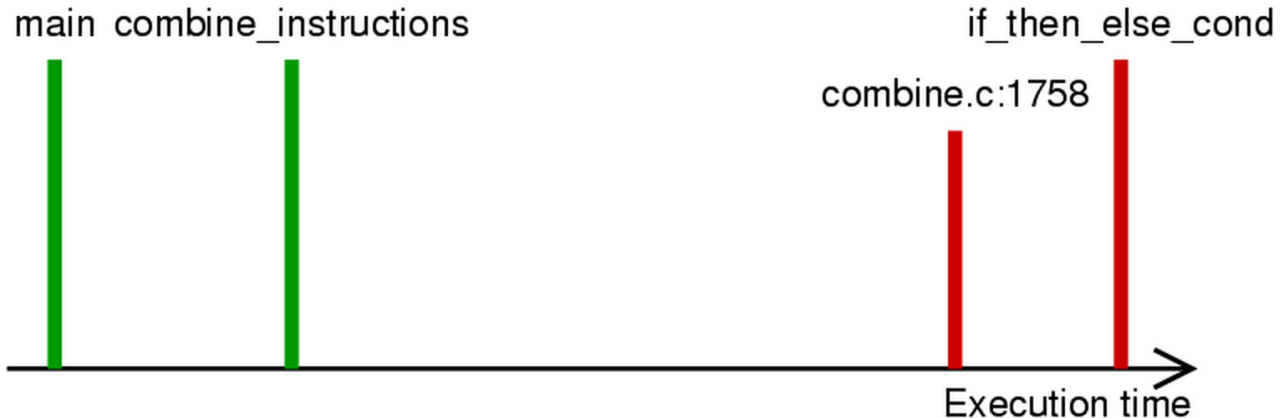
We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.





Isolating the Error

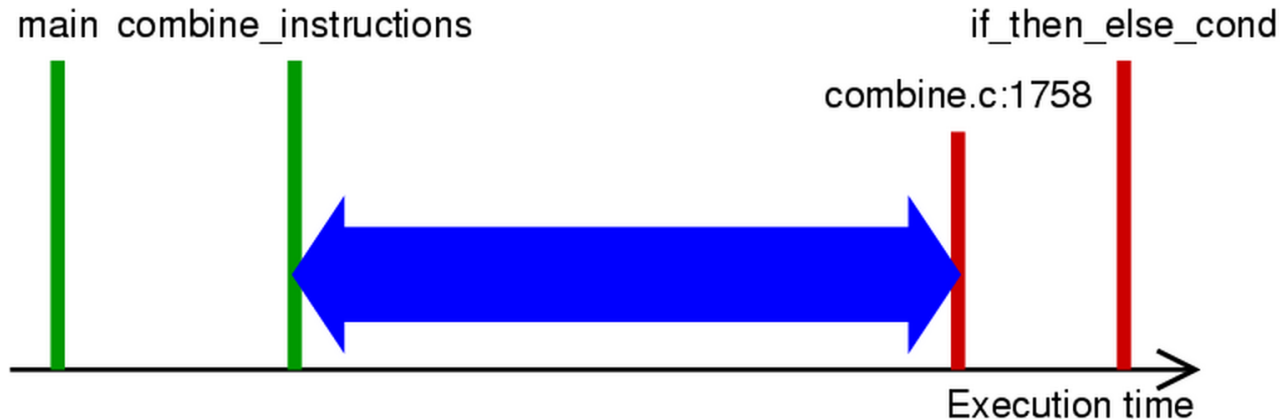
We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.





Isolating the Error

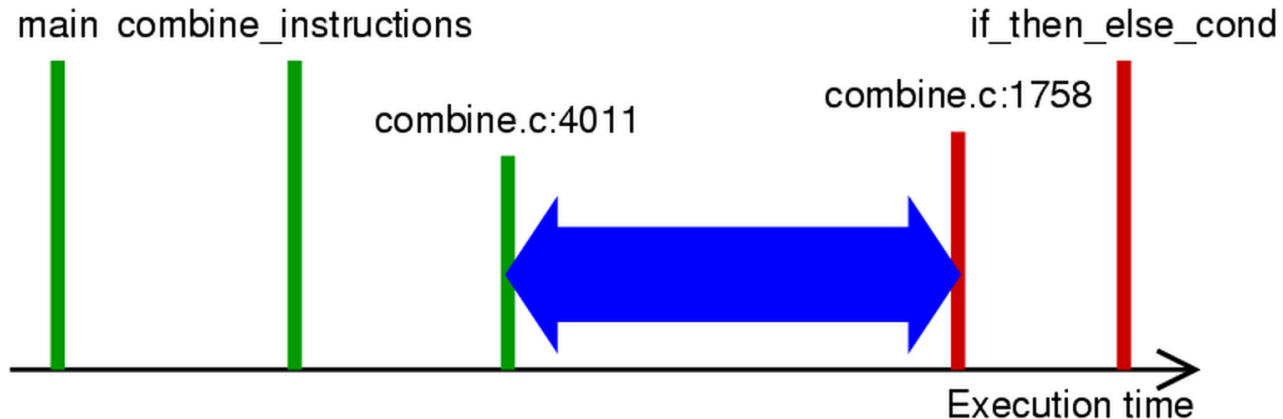
We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.





Isolating the Error

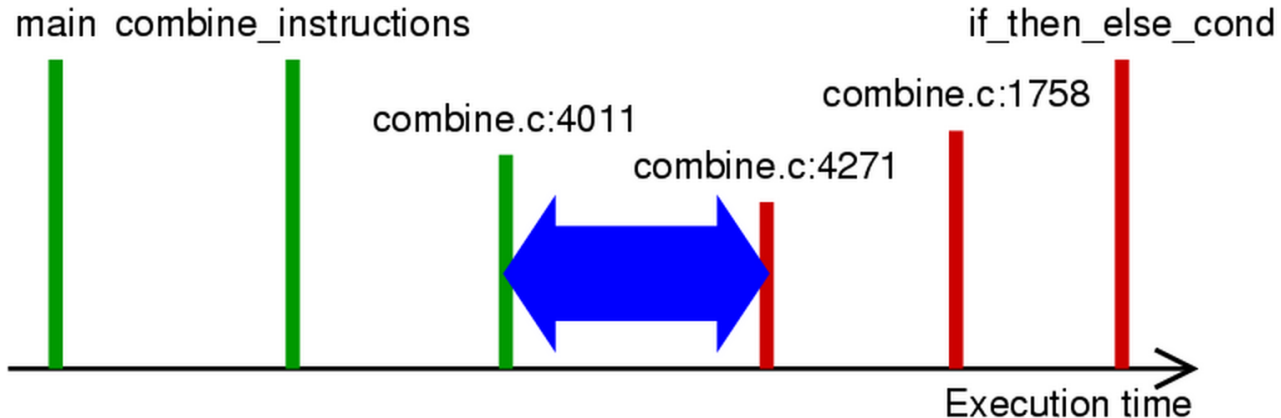
We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.





Isolating the Error

We can narrow down the **error** by (manually) distinguishing **erroneous** and **non-erroneous** causes.



Bad alias in distributive law in lines 4013–4019; fixed in 2.95.3

$$(+ (* a b) c) \Rightarrow (* (+ a c_1)(+ b c_2)) \quad \text{with } c = c_1 = c_2$$





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?*



The screenshot shows the AskIgor web application in a Mozilla browser window. The title bar reads "Askigor - Automated Debugging Service - Mozilla {Build ID: 2002072204}". The address bar shows "http://www.askigor.org/". The main content area features the "AskIgor BETA" logo and a navigation menu with "Debug a Program", "Query Status", and "About Askigor". The "Debug a Program" section contains several form fields:

- What is your program file?** (Linux PC executables only. [More info...](#))
File: - How do you invoke your program such that it fails?** ([More info...](#))
Cmd:
- How do you invoke it such the failure does *not* occur?** ([More info...](#))
Cmd:
- Do the invocations above require additional files?** ([More info...](#))
File 1:
File 2:
File 3:

Below the form fields, it says: "If more than three files are required, please upload them in an archive." A large blue button labeled "Debug it!" is centered at the bottom of the form. At the very bottom of the browser window, the status bar shows "Document: Done (0.664 secs)".

Submit buggy program



Specify invocations



Click on "Debug it"



Diagnosis comes
via e-mail

Up and running
since Summer 2003

56% "pinpoints the bug"
22% "helpful insights"



Delta Debugging Plug-Ins



18/22

The screenshot shows the Eclipse IDE interface. The main editor displays the following Java code:

```
package edu.marlbora.msie.ipl;

/**
 * @author gc-admin
 * To change this generated comment edit the template
 * Window>Preferences>Java>Templates.
 * To enable and disable the creation of type co
 * Window>Preferences>Java>Code Generation.
 */
public class AClass {
    public static void main(String[] args) {
        System.out.println("I am a class");
    }
}
```

The Delta Debugging dialog box is titled "Delta Debugging" and contains the text "Determining failure cause, please stand by..." and a "Cancel" button.

The Package Explorer on the left shows the project structure:

- IPLproject
 - edu.marlbora.msie.ipl
 - AClass.java
 - JRE_LIB - C:\j2sdk1.4.0_0

The Outline view on the right shows the class structure:

- edu.marlbora.msie.ipl
 - AClass
 - main(String[])

The Tasks view at the bottom shows 0 items.

C	I	Description	Resource	In Folder	Location

Package Explorer Hierarchy

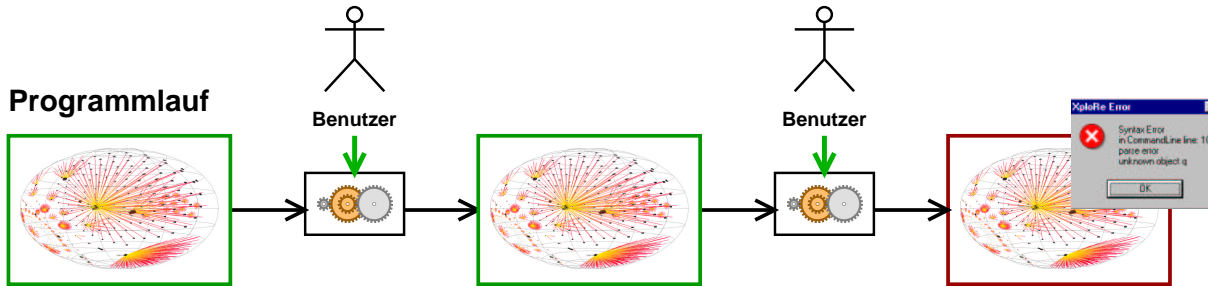
Writable Insert 14 : 4



Delta Debugging in one Run



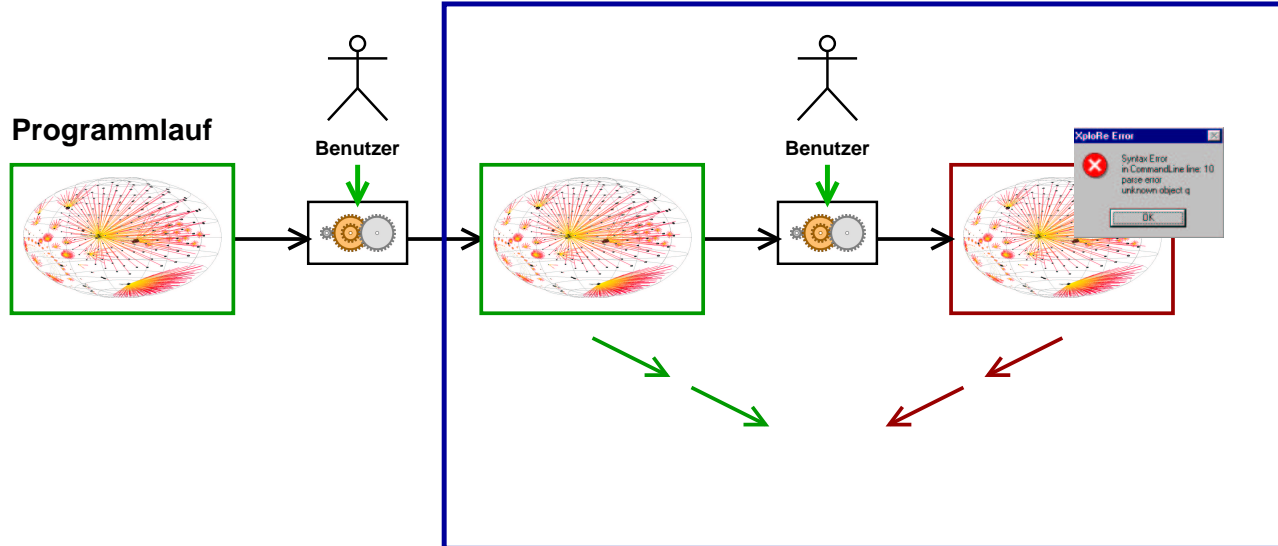
In a reactive program, one single run may suffice:



Delta Debugging in one Run



In a reactive program, one single run may suffice:



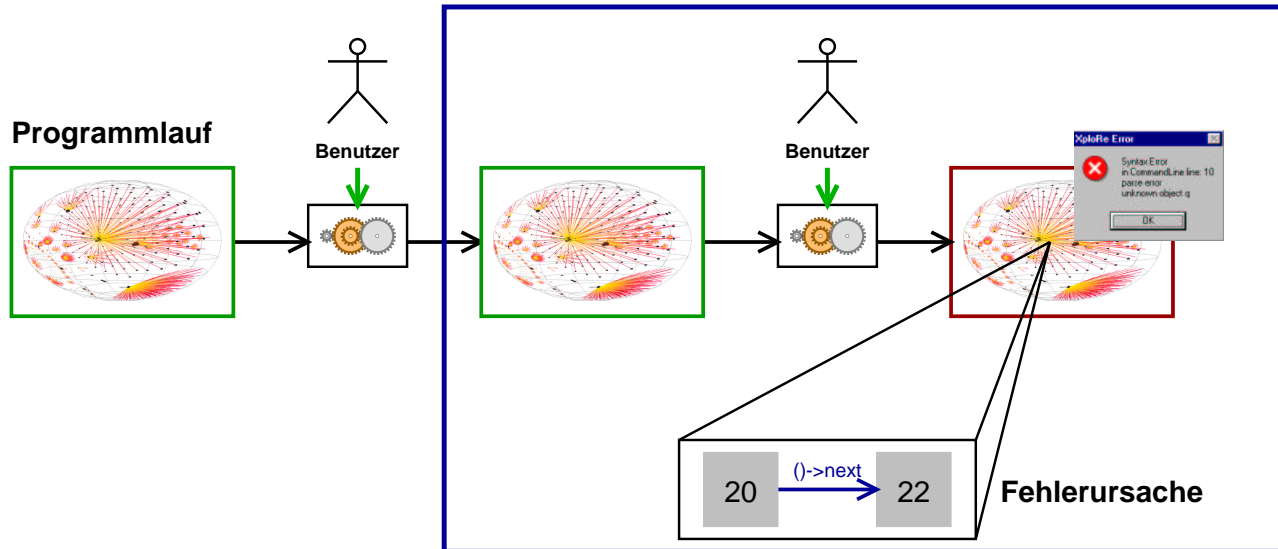
Comparing program state *at different moments in time* again reveals differences...



Delta Debugging in one Run



In a reactive program, one single run may suffice:

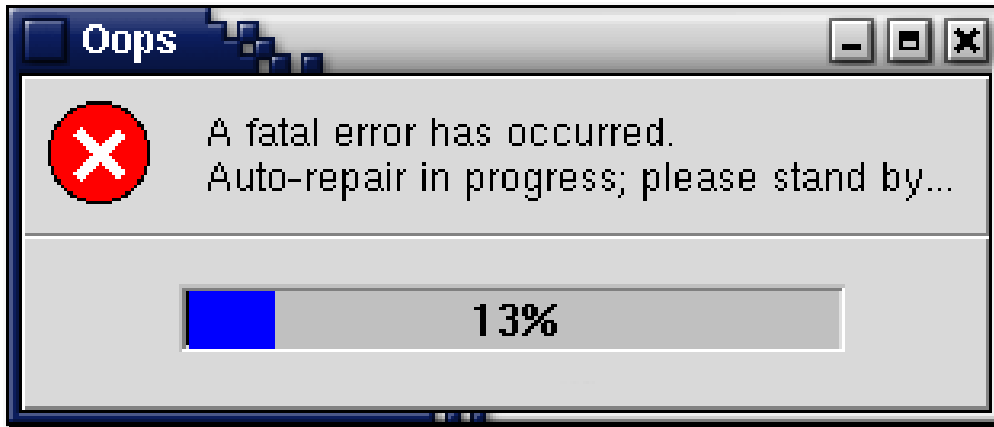


Comparing program state *at different moments in time* again reveals differences, which may be narrowed down to causes.

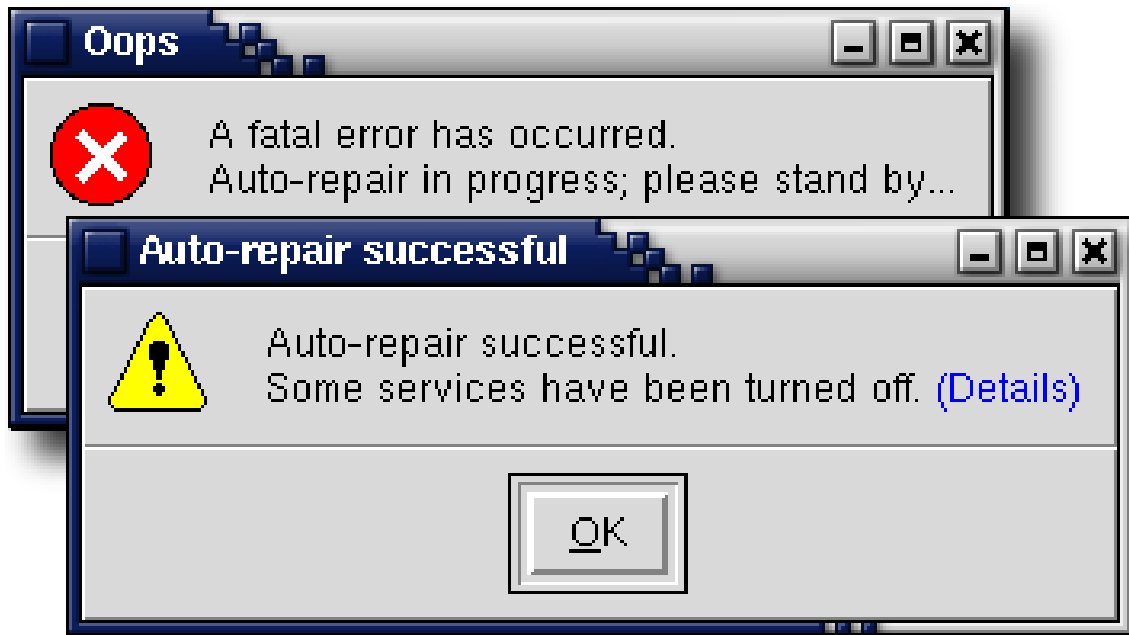
Applications: interactive programs, servers, device drivers...



Self-Repairing Programs



Self-Repairing Programs



Self-Repairing Programs



Oops

A fatal error has occurred.
Auto-repair in progress; please stand by...

Auto-repair successful

Auto-repair successful.
Some services have been turned off. [\(Details\)](#)

Auto-repair details

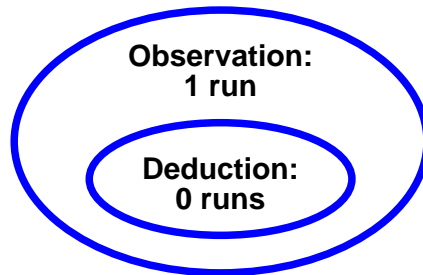
Sub-Pixel anti-aliasing has been turned off,
as it caused a fatal error. [\(Reactivate\)](#)

OK



Past and Future

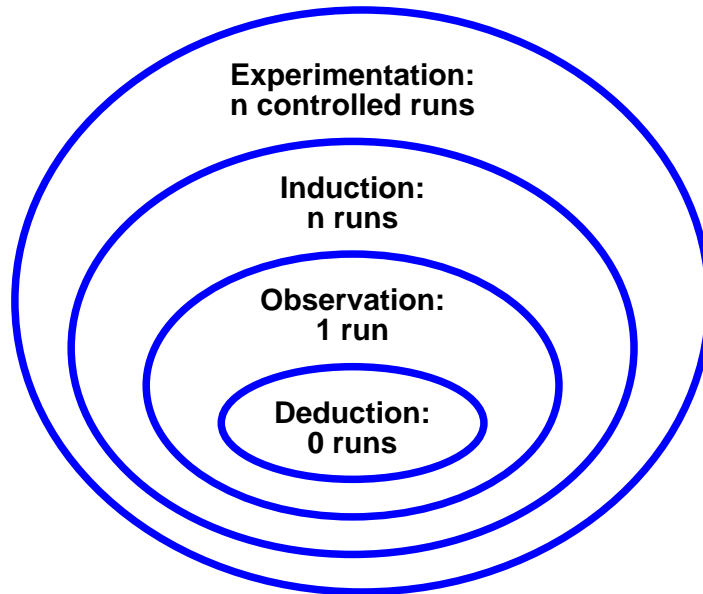
Past 20 years: *deduction* and *observation* techniques



Past and Future



Past 20 years: *deduction* and *observation* techniques



Next 20 years: *induction* and *experimentation*?





Conclusion

- ⇒ We may be able to guarantee the absence of errors— but never the *absence of surprises*.
- ⇒ Failure causes can be isolated *automatically*. . .
 - if we have an automated test
 - where at least one test case passes
- ⇒ Systematic *experimentation* can significantly *augment* “classical” program analysis.
- ⇒ Via automation, debugging becomes a *well-understood and systematic discipline*.
- ⇒ Book “Why does my program fail?” (MK) in Fall 2004

<http://www.askigor.org/>



Read More

Why does my Program Fail? A Guide to Automated Debugging. Morgan Kaufmann Publishers, Fall 2004.

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.

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/>





About this Presentation

This presentation was created by Andreas Zeller, Professor of Computer Science at Saarland University, Saarbrücken, Germany. Contact him at

<http://www.st.cs.uni-sb.de/~zeller/>

This presentation, its source code, and additional material can be downloaded at

<http://www.st.cs.uni-sb.de/papers/fse2002/>

This presentation is licensed under the Creative Commons Attribution License. To view a copy of this license, visit

<http://creativecommons.org/licenses/by/1.0/>

or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA.

