Isolating Cause-Effect Chains
from Computer Programs

Andreas Zeller
Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken
A True Story

Consider the following C program:

```c
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:

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <strong>bug(...)</strong> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✘</td>
</tr>
<tr>
<td>2</td>
<td>double <strong>bug(...)</strong> { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
</tbody>
</table>
Isolating Failure Causes

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✘</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td></td>
</tr>
</tbody>
</table>
# Isolating Failure Causes

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
</tbody>
</table>
# Isolating Failure Causes

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ... }</td>
<td>✔</td>
</tr>
</tbody>
</table>
**Isolating Failure Causes**

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✓</td>
</tr>
<tr>
<td>4</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✓</td>
</tr>
</tbody>
</table>
# Isolating Failure Causes

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <code>bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>double <code>bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>double <code>bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>double <code>bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>double <code>bug(...) { int i, j; i = 0; for (...) { ... } ... }</code></td>
<td>✔</td>
</tr>
</tbody>
</table>
**Isolating Failure Causes**

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✔</td>
</tr>
<tr>
<td>5</td>
<td><code>double bug(...) { int i, j; i = 0; for (...) { ... } ...}</code></td>
<td>✗</td>
</tr>
</tbody>
</table>
# Isolating Failure Causes

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

<table>
<thead>
<tr>
<th>#</th>
<th>GCC input</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>double <strong>bug</strong>(...) { int i, j; i = 0; for (...) { ... } ...}</td>
<td>✔</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>18</td>
<td>... <em>z[i] = z[i] * (z[0] + 1.0);</em> ...</td>
<td>✔</td>
</tr>
<tr>
<td>19</td>
<td>... <em>z[i] = z[i] * (z[0] + 1.0);</em> ...</td>
<td>✗</td>
</tr>
</tbody>
</table>

+ **1.0** is the failure cause – after only 19 tests (≈ 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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
</tbody>
</table>
**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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_lineno</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
</tbody>
</table>
# 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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenumber</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✓</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td></td>
</tr>
</tbody>
</table>
**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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
</tbody>
</table>
# 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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td></td>
</tr>
</tbody>
</table>
**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`

<table>
<thead>
<tr>
<th></th>
<th>reg_rtx_no</th>
<th>curInsn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>?</td>
</tr>
</tbody>
</table>
**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`

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>❌</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>?</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
</tbody>
</table>
**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*

<table>
<thead>
<tr>
<th>#</th>
<th>reg_rtx_no</th>
<th>cur_insn_uid</th>
<th>last_linenum</th>
<th>first_loop_store_insn</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4e4</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>74</td>
<td>15</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4e4</td>
<td>❓</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>74</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>70</td>
<td>14</td>
<td>0x81fc4a0</td>
<td>✔</td>
</tr>
</tbody>
</table>

Consequence: determine and apply *structural differences*!
The GCC Memory Graph

Our HOWCOM prototype extracts the program state as *graph*: Vertices are *variables*, edges are *references*.
**Structural Differences**

HOWCOMETE can compute structural graph differences:

\( \Delta_{15} \) creates a variable, \( \Delta_{20} \) deletes another
The Process in a Nutshell

Failing Run

Passing Run
The Process in a Nutshell

Failing Run

Passing Run
The Process in a Nutshell

Failing Run

Passing Run
**Relevant State Differences**

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

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

Only one variable causes the failure:

```c
$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`.

3. Execution reaches `if_then_else_cond (95th hit)`.
   Since `*first_loop_storeInsn→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?
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/


http://www.askigor.org/