Lecture 19

Delta Debugging
Cooperative Bug Isolation
Today’s Agenda

- Presentation:
  - Chris on Cooperative Bug Isolation
- Quiz on Delta Debugging
Today’s Agenda

• Delta Debugging:
  • Simplifying Failure Causes => Isolating Failure Causes
  • Applications of Delta Debugging Algorithm
  • Isolating Cause and Effect Chain
Quiz: Delta Debugging
Isolating Failure Causes

Andreas Zeller
Simplifying Input

<SELECT NAME="priority" MULTIPLE SIZE=7>
✔
✔
✔
✔
✔
✔
✔
✔
✔
✔
✔
Simplifying

Input

Failure Cause
Isolating Input

```html
<SELECT NAME="priority" MULTIPLE SIZE=7>

Difference narrowed down

<SELECT NAME="priority" MULTIPLE SIZE=7>

<SELECT NAME="priority" MULTIPLE SIZE=7>
```
Isolating Input

<SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
    <SELECT NAME="priority" MULTIPLE SIZE=7>
</SELECT>
Isolating

Input

Failure Cause

10
Configuration

Circumstance

All circumstances

\[ C = \{ \delta_1, \delta_2, \ldots \} \]

Configuration

\[ c \subseteq C \]

\[ c = \{ \delta_1, \delta_2, \ldots \delta_n \} \]
Tests

Testing function

\[ test(c) \in \{\checkmark, \times, ?\} \]

Initial configurations

\[ test(c_{\checkmark}) = \checkmark \]
\[ test(c_{\times}) = \times \]
Minimal Difference

Goal: Subsets $c_x$ and $c'$

$$\emptyset = c \subseteq c' \subset c'_x \subseteq c_x$$

Difference

$$\Delta = c'_x \setminus c'$$

Difference is 1-minimal

$$\forall \delta_i \in \Delta \cdot test(c' \cup \{\delta_i\}) \neq \checkmark \land test(c'_x \setminus \{\delta_i\}) \neq \times$$
Algorithm Sketch

- Extend $ddmin$ such that it works on two sets at a time — $c'_x$ and $c'_\checkmark$
- Compute subsets
  \[ \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n = \Delta = c'_x \setminus c'_\checkmark \]
- For each subset, test
  - the addition $c'_\checkmark \cup \Delta_i$
  - the removal $c'_x \setminus \Delta_i$
# Test Outcomes

<table>
<thead>
<tr>
<th></th>
<th>![X]</th>
<th>![✓]</th>
</tr>
</thead>
<tbody>
<tr>
<td>$test(c' \setminus \Delta_i)$</td>
<td>$c_x := c' \setminus \Delta_i$</td>
<td>$c'_x := c' \setminus \Delta_i$</td>
</tr>
<tr>
<td>$test(c' \cup \Delta_i)$</td>
<td>$c'_x := c' \cup \Delta_i$</td>
<td>$c'_x := c' \cup \Delta_i$</td>
</tr>
<tr>
<td>otherwise</td>
<td>increase granularity</td>
<td>most valuable outcomes</td>
</tr>
</tbody>
</table>
dd in a Nutshell

\[ dd(c_\downarrow, c_\uparrow) = (c'_\downarrow, c'_\downarrow) \quad \Delta = c'_\downarrow \setminus c'_\uparrow \text{ is 1-minimal} \]

\[ dd(c_\downarrow, c_\uparrow) = dd'(c_\downarrow, c_\uparrow, 2) \]

\[ dd'(c'_\downarrow, c'_\uparrow, n) = \]

\[
\begin{align*}
& (c'_\downarrow, c'_\uparrow) \\
& dd'(c'_\downarrow \setminus \Delta_i, c'_\uparrow, 2) \\
& dd'(c'_\downarrow, c'_\uparrow \cup \Delta_i, 2) \\
& dd'(c'_\downarrow \cup \Delta_i, c'_\uparrow, \max(n - 1, 2)) \\
& dd'(c'_\downarrow, c'_\uparrow \setminus \Delta_i, \max(n - 1, 2)) \\
& dd'(c'_\downarrow, c'_\uparrow, \min(2n, |\Delta|)) \\
& (c'_\downarrow, c'_\uparrow)
\end{align*}
\]

if \(|\Delta| = 1\)

if \(\exists i \in \{1..n\} \cdot test(c'_\downarrow \setminus \Delta_i) = \checkmark\)

else if \(\exists i \in \{1..n\} \cdot test(c'_\downarrow \cup \Delta_i) = \checkmark\)

else if \(\exists i \in \{1..n\} \cdot test(c'_\downarrow \cup \Delta_i) = \times\)

else if \(\exists i \in \{1..n\} \cdot test(c'_\downarrow \setminus \Delta_i) = \times\)

else \(n < |\Delta| \) (“increase granularity”) otherwise
def dd(c_pass, c_fail):
    n = 2
    while 1:
        delta = listminus(c_fail, c_pass)
        deltas = split(delta, n); offset = 0; j = 0
        while j < n:
            i = (j + offset) % n
            next_c_pass = listunion(c_pass, deltas[i])
            next_c_fail = listminus(c_fail, deltas[i])
            if test(next_c_fail) == FAIL and n == 2:
                c_fail = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_fail) == PASS:
                c_pass = next_c_fail; n = 2; offset = 0; break
            elif test(next_c_pass) == FAIL:
                c_fail = next_c_pass; n = 2; offset = 0; break
            elif test(next_c_fail) == FAIL:
                c_fail = next_c_fail; n = max(n - 1, 2); offset = i; break
            elif test(next_c_pass) == PASS:
                c_pass = next_c_pass; n = max(n - 1, 2); offset = i; break
            else:
                j = j + 1
        if j >= n:
            if n >= len(delta):
                return (delta, c_pass, c_fail)
            else:
                n = min(len(delta), n * 2)
Applications

<table>
<thead>
<tr>
<th>Input</th>
<th>Code Changes</th>
<th>Schedules</th>
</tr>
</thead>
</table>

18
Isolating Input

Failure Cause: Isolation: 5 tests
Simplification: 48 tests
Code Changes

From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Report Address <bug-ddd@gnu.org>
Subject: Problem with DDD and GDB 4.17

When using DDD with GDB 4.16, the run command correctly uses any prior command-line arguments, or the value of "set args". However, when I switched to GDB 4.17, this no longer worked: If I entered a run command in the console window, the prior command-line options would be lost. [...]
Version Differences

New version

Old version

Program works

Program fails

Causes
What was Changed

$ diff -r gdb-4.16 gdb-4.17
diff -r gdb-4.16/COPYING gdb-4.17/COPYING
5c5
< 675 Mass Ave, Cambridge, MA 02139, USA
---
> 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
282c282
< Appendix: How to Apply These Terms to Your New Programs
---
> How to Apply These Terms to Your New Programs

…and so on for 178,200 lines (8,721 locations)
Challenges

• Granularity – within some large change, only a few lines may be relevant

• Interference – some (later) changes rely on other (earlier) changes

• Inconsistency – some changes may have to be combined to produce testable code

Delta debugging handles all this
General Plan

• Decompose diff into changes per location (= 8,721 individual changes)

• Apply subset of changes, using PATCH

• Reconstruct GDB; build errors mean unresolved test outcome

• Test GDB and return outcome
Isolating Changes

Delta Debugging Log

• Result after 98 tests (= 1 hour)
The Failure Cause

diff -r gdb-4.16/gdb/infcmd.c gdb-4.17/gdb/infcmd.c
1239c1278
< "Set arguments to give program being debugged when it is started."
---
> "Set argument list to give program being debugged when it is started."

- Documentation becomes GDB output
- DDD expects **Arguments**, but GDB outputs **Argument list**
Optimizations

- History – group changes by creation time
- Reconstruction – cache several builds
- Grouping – according to scope
- Failure Resolution – scan error messages for possibly missing changes
Thread Schedules

The behavior of a multi-threaded program can depend on the thread schedule:

<table>
<thead>
<tr>
<th>Schedule</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>open(&quot;.htpasswd&quot;)</td>
<td>open(&quot;.htpasswd&quot;)</td>
</tr>
<tr>
<td></td>
<td>read(...)</td>
<td>read(...)</td>
</tr>
<tr>
<td></td>
<td>modify(...)</td>
<td>modify(...)</td>
</tr>
<tr>
<td></td>
<td>write(...)</td>
<td>write(...)</td>
</tr>
<tr>
<td></td>
<td>close(...)</td>
<td>close(...)</td>
</tr>
</tbody>
</table>

Schedule Switch

A’s updates get lost!
DEJAVU captures and replays program runs deterministically:

Allows simple reproduction of schedules and induced failures
Schedules as Input

Using DEJAVU, we can consider the schedule as an input which determines whether the program passes or fails.

The schedule difference causes the failure!
We start with runs ✓ and ✗

We determine the differences $\Delta_i$ between thread switches $t_i$: 

- $t_1$ occurs in ✓ at “time” 254
- $t_1$ occurs in ✗ at “time” 278
- The difference $\Delta_1 = |278 - 254|$ induces a statement interval: the code executed between “time” 254 and 278
- Same applies to $t_2, t_3$, etc.
Isolating Relevant Differences

We use Delta Debugging to isolate the relevant differences.

Delta Debugging applies subsets of differences to:

- The entire difference $\Delta_1$ is applied
- Half of the difference $\Delta_2$ is applied
- $\Delta_3$ is not applied at all

DEJAVU executes the debuggee under this generated schedule; an automated test checks if the failure occurs.
Isolating Differences

The Isolation Process

Delta Debugging systematically narrows down the difference.

Dejavu replays the generated schedule.

Test outcome
Example: Raytracer

- Raytracer program from Spec JVM98 suite
- Injected a simple race condition
- Set up automated test + random schedules
- Obtained passing and failing schedule
- 3,842,577,240 differences, each moving a thread switch by ±1 yield point (time unit)
Isolating Cause-Effect Chains

Andreas Zeller
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];
}
What is the cause of this failure?
1. The programmer creates a defect – an error in the code.

2. When executed, the defect creates an infection – an error in the state.

3. The infection propagates.

4. The infection causes a failure.

This infection chain must be traced back – and broken.
Tracing Infections

- For every infection, we must find the earlier infection that causes it.
- Program analysis tells us possible causes
Tracing Infections
Isolating Input

Difference causes failure
Isolating States

Variables

Difference causes failure

Variables
Comparing States

- What is a program state, anyway?
- How can we compare states?
- How can we narrow down differences?
A Sample Program

$ sample 9 8 7
Output: 7 8 9

$ sample 11 14
Output: 0 11

Where is the defect which causes this failure?
```c
int main(int argc, char *argv[]) {
    int *a;

    // Input array
    a = (int *)malloc((argc - 1) * sizeof(int));
    for (int i = 0; i < argc - 1; i++)
        a[i] = atoi(argv[i + 1]);

    // Sort array
    shell_sort(a, argc);

    // Output array
    printf("Output: ");
    for (int i = 0; i < argc - 1; i++)
        printf("%d ", a[i]);
    printf("\n");

    free(a);
    return 0;
}
```
A sample state

• We can access the entire state via the debugger:
  1. List all base variables
  2. Expand all references...
  3. ...until a fixpoint is found
## Sample States

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value in $r^*$</th>
<th>Value in $r^\times$</th>
</tr>
</thead>
<tbody>
<tr>
<td>argc</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>argv[0]</td>
<td>&quot;./sample&quot;</td>
<td>&quot;./sample&quot;</td>
</tr>
<tr>
<td>argv[1]</td>
<td>&quot;9&quot;</td>
<td>&quot;11&quot;</td>
</tr>
<tr>
<td>argv[2]</td>
<td>&quot;8&quot;</td>
<td>&quot;14&quot;</td>
</tr>
<tr>
<td>argv[3]</td>
<td>&quot;7&quot;</td>
<td>0x0 (NIL)</td>
</tr>
<tr>
<td>i</td>
<td>1073834752</td>
<td>1073834752</td>
</tr>
<tr>
<td>j</td>
<td>1074077312</td>
<td>1074077312</td>
</tr>
<tr>
<td>h</td>
<td>1961</td>
<td>1961</td>
</tr>
<tr>
<td>size</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

This state difference is both effect when the input) as well as cause when for the failure).

at shell_sort()
Narrowing State Diffs

■ = \( \delta \) is applied, \( \square \) = \( \delta \) is *not* applied

<table>
<thead>
<tr>
<th>#</th>
<th>( a'[0] )</th>
<th>( a[0] )</th>
<th>( a'[1] )</th>
<th>( a[1] )</th>
<th>( a'[2] )</th>
<th>( a[2] )</th>
<th>( argc )</th>
<th>( argv[1] )</th>
<th>( argv[2] )</th>
<th>( argv[3] )</th>
<th>( i )</th>
<th>( size )</th>
<th>Output</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>7 8 9</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>0 11</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>0 11 14</td>
<td>✗</td>
</tr>
<tr>
<td>4</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>7 11 14</td>
<td>?</td>
</tr>
<tr>
<td>5</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>0 9 14</td>
<td>✗</td>
</tr>
<tr>
<td>6</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>7 9 14</td>
<td>?</td>
</tr>
<tr>
<td>7</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>0 8 9</td>
<td>✗</td>
</tr>
<tr>
<td>8</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>0 8 9</td>
<td>✗</td>
</tr>
</tbody>
</table>

Result: ■

Conclusion: \( a'[2] = 0 \) instead of 7 causes the failure.
Complex State

• Accessing the state as a *table* is not enough:
  • References are not handled
  • Aliases are not handled
• We need a *richer* representation
A Memory Graph

```
<Root>

i'  j  h  a'  size  a  i  argc  argv

0x8099ae8  4  0x8099ae8  3  4  0xbffff5a4

1073834752  1074077312  1961  0x8099ae8  4  0x8099ae8  3  4  0xbffff5a4

[...]

0[0..3]

0[0]  0[1]  0[2]  0[3]

9  8  7  1961

[...]

0[0..4]


0xbffff71a  0xbffff749  0xbffff74c  0xbffff74f  0x0

[/sample]  "9"  "8"  "7"
```
Unfolding Memory

• Any variable: make new node
• Structures: unfold all members
• Arrays: unfold all elements
• Pointers: unfold object being pointed to
  • Does $p$ point to something? And how many?
Comparing States

passing run

failing run
Comparing States

- Basic idea: *compute common subgraph*
- Any node that is not part of the common subgraph becomes a *difference*
- Applying a difference means to create or delete nodes – and adjust references
- All this is done within GDB
Applying Diffs

$\delta_{15}$ creates a variable, $\delta_{20}$ deletes another
## Results: GCC Transitions

<table>
<thead>
<tr>
<th>#</th>
<th>Location</th>
<th>Cause transition to variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(Start)</td>
<td>argv[3]</td>
</tr>
<tr>
<td>1</td>
<td>toplev.c:4755</td>
<td>name</td>
</tr>
<tr>
<td>2</td>
<td>toplev.c:2909</td>
<td>dump_base_name</td>
</tr>
<tr>
<td>3</td>
<td>c-lex.c:187</td>
<td>finput→_IO_buf_base</td>
</tr>
<tr>
<td>4</td>
<td>c-lex.c:1213</td>
<td>nextchar</td>
</tr>
<tr>
<td>5</td>
<td>c-lex.c:1213</td>
<td>yyssa[41]</td>
</tr>
<tr>
<td>6</td>
<td>c-typeck.c:3615</td>
<td>yyssa[42]</td>
</tr>
<tr>
<td>7</td>
<td>c-lex.c:1213</td>
<td>last_insn→fld[1].rtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[1].rtx→fld[3].rtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[1].rtx.code</td>
</tr>
<tr>
<td>8</td>
<td>c-decl.c:1213</td>
<td>sequence_result[2]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[0].rtvec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→elem[0].rtx→fld[1].rtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[1].rtx→fld[1].rtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[1].rtx→fld[1].rtx</td>
</tr>
<tr>
<td></td>
<td></td>
<td>→fld[1].rtx→fld[1].rtx</td>
</tr>
<tr>
<td>9</td>
<td>combine.c:4271</td>
<td>x→fld[0].rtx→fld[0].rtx</td>
</tr>
</tbody>
</table>
To isolate failure causes automatically, use
- an *automated test case*
- a means to *narrow down the difference*
- a *strategy for proceeding*.

One possible strategy is Delta Debugging.
Concepts (2)

★ Delta Debugging can isolate failure causes

- in the (general) input
- in the version history
- in thread schedules
- in program states

★ Every such cause implies a fix – but not necessarily a correction.
Dear students,

I updated the lecture schedule. Most notable changes are:

- I removed (R) signs from several papers making them as optional.
  - Reps' et al.'s profiling paper for 4/8,
  - Lanza et al.'s paper on metrics and visualization for 4/20,
  - Boshernitsan's paper on source transformation for 4/29
  If you are signed up for these papers, you are still scheduled to present. However, I won't discuss these papers in depth during my lecture.

- I switched the order between Lanza et al.'s and Murphy et al.'s paper.

- For next monday, I will talk about using delta-debugging for isolating cause-effect chain. It's likely that we will have more discussion on regression testing on next wednesday instead. If you are signed up for presenting Orso et al's paper, you are still on for monday.

Thanks!
Miryung