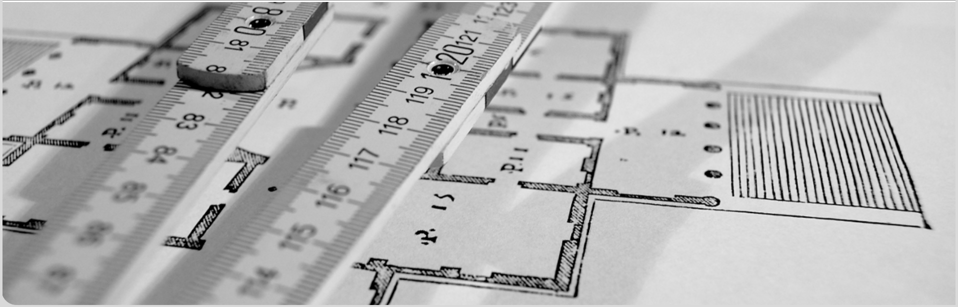
Karlsruher Institut für Technologie

Delta Debugging

A summary of Delta Debugging and its uses based on Andreas Zellers work

Moritz Laupichler | 26. November 2018

FAKULTÄT FÜR INFORMATIK



Motivation

«Everyone knows that debugging is twice as hard as writing a program in the first place.

So if you're as clever as you can be when you write it, how will you ever debug it?»

– Brian Kernighan in "The Elements of Programming Style"

Version Control allows DD

- Version Control has been around since the 80’s
- central terms: configuration, change

The DD idea

Configuration:	Changes:	Configuration:
Yesterday	$\longrightarrow$	Today
	$\longrightarrow$	
Passes tests. ✓	$\longrightarrow$	Tests fail. ✗

Idea: Delta Debugging

Find the minimal set of changes between Yesterday and Today that induces the failure.

The intuitive approach

- $\mathcal{C} = \{\Delta_1, \ldots, \Delta_n\}$: All changes between Yesterday and Today
- $c \subseteq \mathcal{C}$: A configuration (set of changes applied to Yesterday)
- $test : 2^{\mathcal{C}} \rightarrow \{✓, ✗, ?\}$: Result of the tests applied to a configuration

A simple binary search can be conducted to find singular failure inducing changes:

```
1: function SIMPLEDD($c : 2^{\mathcal{C}}$)
2:     if $|c| = 1$ then return $c$
3:     Split $c$ into two halves $c_1$, $c_2$ so that $c_1 \cap c_2 = \emptyset$
4:     if ($test(c_1) = ✗$) then return $simpledd(c_1)$
5:     else return $simpledd(c_2)$
```

Difficulty #1: Interference

- *ddsimple* works for single failure inducing-changes.
  - In each recursion step it applies only the set of changes known to contain a failure inducing change.
- But what if two changes exist that individually pass the tests but their combination induces failure?

Difficulty #1: Interference

Let $c_1, c_2 \in \mathcal{C}$. c_1 and c_2 **interfere** when $\mathit{test}(c_1) = ✓$, $\mathit{test}(c_2) = ✓$ but $\mathit{test}(c_1 \cup c_2) = ✗$.

Difficulty #1: Interference

Idea: Leave one set of changes applied

If a configuration $c = c_1 \cup c_2$ with $test(c_1) = ✓$ and $test(c_2) = ✓$ is found by *simpledd* interference between c_1 and c_2 has been detected. Run *simpledd* on c_1 while leaving c_2 applied and vice versa.

```
1: function $dd_2(c, r : 2^{\mathcal{C}}) : 2^{\mathcal{C}}$
2:     let $c_1, c_2 \subseteq c$ with $c_1 \cup c_2 = c, c_1 \cap c_2 = \emptyset, |c_1| \approx |c_2|$
3:     return $\begin{cases} c & \text{if } |c| = 1, \\ dd_2(c_1, r) & \text{if } test(c_1 \cup r) = ✗, \\ dd_2(c_2, r) & \text{if } test(c_2 \cup r) = ✗, \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise} \end{cases}$
```

Difficulty #2: Inconsistency

- *dd* combines changes arbitrarily (in the case of interference)
- This can lead to inconsistent configurations, i.e. no test outcome can be determined for these configurations

Difficulty #2: Inconsistency

Let $c_1, c_2 \subseteq \mathcal{C}$. An **inconsistency** occurs when $\mathit{test}(c_1 \cup c_2) = ?$.

Idea: More granular subsets

If less changes are applied at once the chances of an inconsistent result are reduced. Hence, if the algorithm cannot find any consistent confgurations reduce the number of changes per subset.

Difficulty #2: Inconsistency

Necessary changes to *dd*:

1. Extend *dd* to work on a number n of subsets $c_1, \ldots, c_n$
2. **Interference** occurs when c_i and its complement $\bar{c}_i$ both pass: $test(c_i) = ✓$ and $test(\bar{c}_i) = ✓$ ($\bar{c}_i = \mathcal{C} \setminus c_i$)
3. Add the case of **preference**: If $test(c_i) = ?$ and $test(\bar{c}_i) = ✓$ we deduce that c_i contains a failure inducing change.
4. Add the case of **Try again**: In any other case repeat the process with $2n$ subsets to improve the chance for consistent configurations.

The dd^+ algorithm

Zellers **extended** *dd* algorithm dd^+ deals with the following cases:

1. "found (in c_i)"
2. "interference"
3. "preference"
4. "try again"

The dd^+ algorithm

Zellers **extended** *dd* algorithm dd^+ deals with the following cases:

1. "found (in c_i)" $\Rightarrow$ continue search in c_i
2. "interference"
3. "preference"
4. "try again"

The dd^+ algorithm

Zellers **extended** *dd* algorithm dd^+ deals with the following cases:

1. "found (in c_i)" $\Rightarrow$ continue search in c_i
2. "interference" $\Rightarrow$ search in c_i while leaving $\bar{c}_i$ applied and vice versa
3. "preference"
4. "try again"

The dd^+ algorithm

Zellers **extended** *dd* algorithm dd^+ deals with the following cases:

1. "found (in c_i)" $\Rightarrow$ continue search in c_i
2. "interference" $\Rightarrow$ search in c_i while leaving $\bar{c}_i$ applied and vice versa
3. "preference" $\Rightarrow$ search in c_i while leaving $\bar{c}_i$ applied
4. "try again"

The dd^+ algorithm

Zellers **extended** *dd* algorithm dd^+ deals with the following cases:

1. "found (in c_i)" $\Rightarrow$ continue search in c_i
2. "interference" $\Rightarrow$ search in c_i while leaving $\bar{c}_i$ applied and vice versa
3. "preference" $\Rightarrow$ search in c_i while leaving $\bar{c}_i$ applied
4. "try again" $\Rightarrow$ repeat search on same change set with twice as many subsets

The dd^+ algorithm

Properties of dd^+:

- dd^+ finds a minimal failure-inducing subset of changes as long as the changes are safe, i.e. combined they result in a consistent configuration.
- dd^+ has at most linear time complexity like *dd*.

Use case for dd^+ Analyzing input that crashes a program

Consider the C program *fail.c* that crashed GCC in version 2.95.2:

```
double mult(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];
}
```

How dd^+ finds the minimal set of C tokens that causes the crash

#	GCC input	test
1	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✗
2	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✓
3	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✓
4	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✓
5	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✗
6	double mult(…) $\{int\ i, j;\ \ i = 0;\ \ for(\dots)\{\dots\}\dots\}$	✓
⋮	⋮	⋮
18	… $z[i] = z[i] * (z[0] + 1.0);$ …	✗
19	… $z[i] = z[i] * (z[0] + 1.0);$ …	✓
20	… $z[i] = z[i] * (z[0] + 1.0);$ …	?
21	… $z[i] = z[i] * (z[0] + 1.0);$ …	?

Using DD to find the reason a program fails

- The *dd*$^+$ algorithm as described above can be used to find out which part of a certain input is responsible for crashing a program.
- From a developers perspective it would be great to know *why* this specific input induces failure.

Using DD to find the reason a program fails

Difficulty: Causalities from the input change to the failure

A minimal change in the input has major consequences for the program execution. How can the debugger know which chain of effects is responsible for the eventual failure?

Using DD to find the reason a program fails

Difficulty: Causalities from the input change to the failure

A minimal change in the input has major consequences for the program execution. How can the debugger know which chain of effects is responsible for the eventual failure?

Idea: Delta Debugging on program states

Use delta debugging on program states of a known succeeding run $r_{\checkmark}$ and a known failing run $r_{\times}$ of the program. For each analyzed pair of program states we would then know what current difference causes the failure down the line.

$\Rightarrow$ A *Cause-Effect-Chain* can be constructed from those comparisons.

Delta Debugging on program states

- A program state is essentially a set of (*variable*, *value*) pairs.
- A program state of $r_{\checkmark}$ and one of $r_{\times}$ can have the following differences (the deltas):
  - A variable only present in one state
  - A different value of a variable present in both states
- With this the runs $r_{\checkmark}$ and $r_{\times}$ can be compared in different significant locations.
- We use delta debugging to find the current (*variable*, *value*) pairs that are responsible for the failure of $r_{\times}$ further down the line.

HOWCOME on *fail.c*

Zellers prototypical *HOWCOME* algorithm returns the following Cause-Effect-Chain for *fail.c*:

1. Execution reaches **main**.
   Since the program was invoked as “ccl -O fail.i” variable *argv*[2] is now **“fail.i”**
2. Execution reaches **combine_instructions**.
   Since *argv*[2] was “fail.i”, variable $*first\_loop\_store\_insn \rightarrow fld[1].rtx \rightarrow fld[1].rtx \rightarrow fld[3].rtx \rightarrow fld[1].rtx$ is now $\langle newrtx\_def \rangle$.
3. Execution reaches **if_then_else_cond (95th hit)**.
   Since $*first\_loop\_store\_insn \rightarrow fld[1].rtx \rightarrow fld[1].rtx \rightarrow fld[3].rtx \rightarrow fld[1].rtx$ was $\langle newrtx\_def \rangle$, variable $link \rightarrow fld[0].rtx \rightarrow fld[0].rtx$ is now *link*.
4. Execution ends.
   Since variable $link \rightarrow fld[0].rtx \rightarrow fld[0].rtx$ was *link*, the program now **terminates with a SIGSEGV signal**. The program fails.