YESTERDAY, MY PROGRAM WORKED. TODAY, IT DOES NOT. WHY?

Andreas Zeller

Presented by: Ming-Ho Yee
November 22, 2016

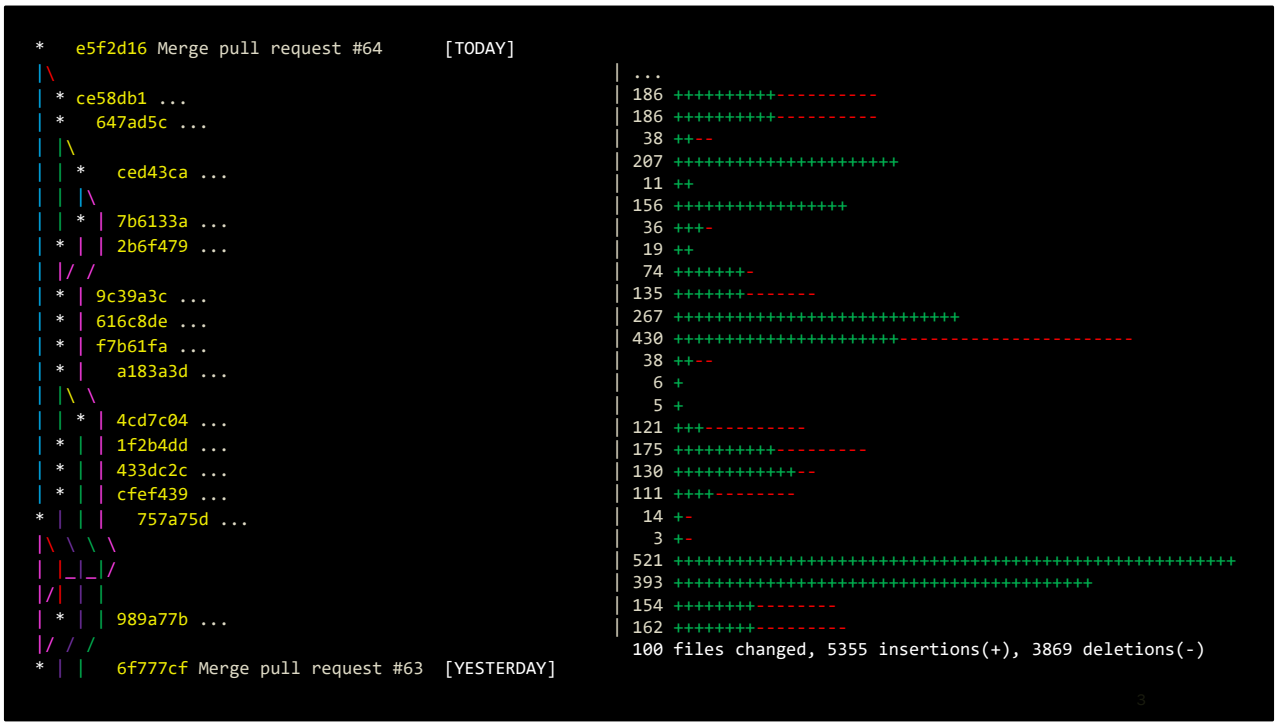
- Unconventional title
  - Doesn't really tell you *what* it contributes, but rather points out the motivation (and does it well)
  - The abstract and introduction also use "marketing tricks"
  - Intro starts with "The GDB people have done it again."
- But overall, the paper was clearly written, and easy to read
  - And the idea isn't very complicated

Impact of Delta Debugging

- Introduced “delta debugging,” an *automated* debugging technique
  - First in a series of papers
- Won the ACM SIGSOFT Impact Paper Award (2009)
- Over 100 citations
  - Still being cited today!

2

- Elegant idea is part of its influence
  - Easy to understand and implement, lots of different applications
- This paper introduced “delta debugging,” an automated debugging technique
  - It was the first in a series of papers
- Won an Impact Paper Award in 2009, and was the keynote at ESEC/FSE 2009
  - Sadly, the video of the keynote is corrupted
- Paper has over 100 citations
  - Still being cited today (2016)



- The motivation is quite simple, and expressed clearly in the title
  - Working program + changes -> no longer works
- Here's an example git log
  - Maybe you're lucky and have version control, and can narrow the bug to a single commit
  - But what if it's a giant commit?
  - What if you don't have version control?
- In the paper, the example was GDB, which did not have version control at the time
  - Almost 200,000 lines of code were changed
- Can we narrow down the changes *automatically*?

Prerequisites for Delta Debugging

- Set of all possible changes: $\mathcal{C} = \{\Delta_1, \Delta_2, \ldots \Delta_n\}$
  - A change set $c \subseteq \mathcal{C}$ is called a *configuration*.
  - An empty configuration $c = \emptyset$ is called a *baseline*.
- Function $test : 2^C \rightarrow \{✓, ✗, ?\}$
  - ✓ - Pass
  - ✗ - Fail
  - **?** - Unresolved
- Assumption:

$$test(\emptyset) = ✓ \wedge test(\mathcal{C}) = ✗$$

4

- What do we actually need for delta debugging?
- A set of all possible changes
  - Note: changes ≠ commit, change = changed line
  - A change set is a configuration (no constraints, there are 2^n possible configurations)
  - An empty configuration is a baseline
- A test function that takes a configuration and will PASS, FAIL, or be UNRESOLVED
  - Unresolved is when your program doesn't build, or maybe you get an infinite loop
- Our current situation:
  - Baseline (yesterday) passes
  - Applying all changes (today) fails

Minimal Failure-Inducing Change Set

- **Goal**: Find the minimal failure-inducing change set
- A change set $c \subseteq \mathcal{C}$ is *failure-inducing* if the following holds:

$$\forall c'(c \subseteq c' \subseteq \mathcal{C} \rightarrow test(c') \neq \checkmark)$$

- A failure-inducing change set $B \subseteq \mathcal{C}$ is *minimal* if the following holds:

$$\forall c \subset B\ (test(c) \neq \text{✗})$$

5

- Goal: find the minimal failure-inducing change set
  - What does this mean?
- A change set is failure-inducing if any change set that includes it will fail
  - Actually, it's "does not pass" because it might be unresolved
  - By definition, C is failure-inducing, but that's not very helpful
- Failure-inducing change set is minimal if removing any change means it won't fail (i.e. pass or unresolved)

Three Useful Properties

- **Monotone**

$$\forall c \subseteq \mathcal{C}(test(c) = \text{✗} \rightarrow \forall c' \supseteq c(test(c') \neq \text{✓}))$$
$$\forall c \subseteq \mathcal{C}(test(c) = \text{✓} \rightarrow \forall c' \subseteq c(test(c') \neq \text{✗}))$$

- **Unambiguous**

$$\forall c_1, c_2 \subseteq \mathcal{C}(test(c_1) = \text{✗} \wedge test(c_2) = \text{✗} \rightarrow test(c_1 \cap c_2) \neq \text{✓})$$

- **Consistent**

$$\forall c \subseteq \mathcal{C}(test(c) \neq ?)$$

6

- There are three properties that, if true, we can take advantage of
  - Reduce the amount of searching
- Monotone
  - If a change causes a failure, then any configuration that includes it will also fail (or be unresolved)
  - i.e. it won't start working again
  - Corollary
    - If we have monotonicity, then if we have found a change set that passes, none of its subsets will fail
    - Makes searching more efficient, so you can skip searching a passing configuration's subsets
- Unambiguous
  - Failure is caused by one change set, and not by two disjoint ones
  - i.e. we don't have multiple bugs or multiple causes of the same bug
  - For economy: once we've found a failure-inducing change set, don't need to search the complement
  - Another way to read this constraint: if we have two change sets that fail, they are related
    - Their intersection should be non-empty (and should not pass)

- Consistent
    - We always get a pass or fail, no indeterminate results
- It’s easy if these three properties hold
    - Trickier but still possible if they don’t hold
    - Most important practical problem is inconsistency

Delta Debugging

$$dd(c) = dd_2(c, \emptyset)$$

$$dd_2(c, r) =$$

let $c_1, c_2 \subseteq c$ s.t. $\left(c_1 \cup c_2 = c \wedge c_1 \cap c_2 = \emptyset \wedge |c_1| \approx |c_2| \approx \frac{|c|}{2}\right)$

$$\text{in} \begin{cases} c & \text{if } |c| = 1 \\ dd_2(c_1, r) & \text{if } test(c_1 \cup r) = \text{✗} \\ dd_2(c_2, r) & \text{if } test(c_2 \cup r) = \text{✗} \\ dd_2(c_1, c_2 \cup r) \cup dd_2(c_2, c_1 \cup r) & \text{otherwise} \end{cases}$$

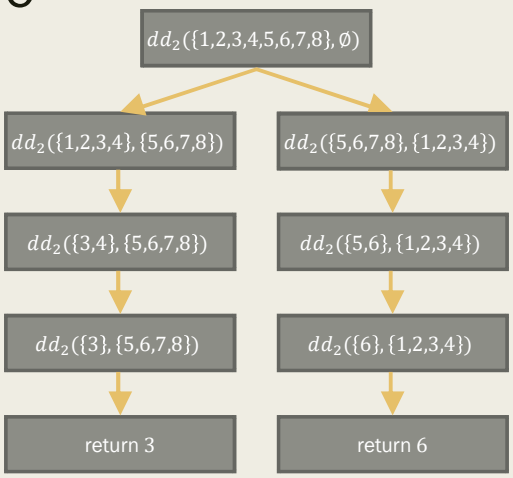
Invariant: $test(r) = \text{✓} \wedge test(c \cup r) = \text{✗}$

7

- This is the pseudocode from the paper
- Start by assuming all three properties true (monotone, unambiguous, consistent)
- Algorithm is divide-and-conquer, like binary search
  - Have a helper function to make the recursion cleaner
- If c has only one change, then it is failure-inducing, so done
- Otherwise, we partition c into disjoint subsets c1, c2 (approximately the same size)
- Check c1, if it fails, c1 contains a failure-inducing change
- Check c2, if it fails, c2 contains a failure-inducing change
- Otherwise, interference
  - Combination of changes in c1 and c2 causes failure
  - Need to search in c1, with *all* changes from c2 applied
    - Will find the changes in c1 that is combined with some changes in c2
  - And then do the same for c2, and union the two changes found
- Invariant: r always passes, c U r always fails
  - We keep r around to handle interference
  - But we're interested in searching c

Delta Debugging Example

Step	Configuration								*test*
1	1	2	3	4	.	.	.	.	✓
2	.	.	.	.	5	6	7	8	✓
3	1	2	.	.	5	6	7	8	✓
4	.	.	3	4	5	6	7	8	✗
5	.	.	3	.	5	6	7	8	✗
6	1	2	3	4	5	6	.	.	✗
7	1	2	3	4	5	.	.	.	✓
Result	.	.	3	.	.	6	.	.	



8

- If there's no interference, it's just binary search
- Here's an example with interference, taken from the paper
    - Number represents an included change, dot represents excluded change
- Let's step through the example
- Start the call with 8 changes. Test each half, both pass, so interference
    - Need to make two recursive calls
- Make two tests, one of them fails, so we know failure is in {3,4} U {5,6,7,8}
- Recurse, and find that 3 is the failure-inducing change
- Now back to the other recursive call
- We only need on test, to se that error is in {5,6} U {1,2,3,4}
- We recurse, see 5 is OK, so 6 is the error
- And we take the union of 3 and 6

Non-Monotonicity and Ambiguity

- **Non-monotone** configuration: a later change might "undo" a failure
  - But "today" is broken, so there exists another failure-inducing change
- **Ambiguous** configuration: multiple failure-inducing change sets
  - Delta debugging will find one of them

9

- The simple algorithm requires three properties to be true
  - What if they don't hold?
- Recall: monotone means adding changes to a broken configuration doesn't fix it
  - This lets us search more efficiently
  - If the configuration isn't monotone, then some change ends up fixing the failure
  - But since "today" is broken, there must be another failure-inducing change
  - Delta debugging will find one of the failures (and it might take more time)
- Recall: unambiguous means there is only one failure-inducing change set
  - OK if there are multiple
  - Algorithm will find one of them
  - Then we can fix it, and run delta debugging again

Inconsistency

- Sometimes the outcome of a test cannot be determined
  - E.g. change cannot be applied, program will not build, program does not execute correctly
- Approach: split c into smaller subsets
  - $\emptyset$ ("yesterday") and $\mathcal{C}$ ("today") are consistent
  - Want a configuration closer to "yesterday" or "today"

10

- Inconsistency is a big problem
  - When the outcome of a test cannot be determined
    - Change cannot be applied (e.g. conflict), program doesn't build, program doesn't work
  - Non-monotone and ambiguous configurations are minor in comparison, and easily dealt with
  - Inconsistency will be more common, and is harder to manage
- The basic approach is to split c into smaller subsets. Here's why:
  - Yesterday and today are consistent
  - c is already in subsets, but they're all unresolved
  - We want a configuration closer to yesterday (or today), so higher chances of being consistent
  - So apply fewer changes to yesterday (or remove fewer changes from today)
  - We do this by splitting c into smaller subsets

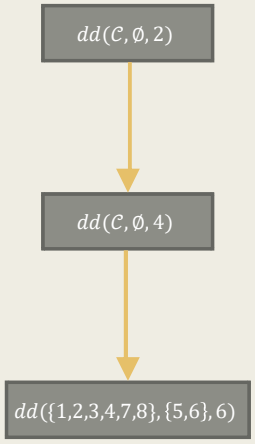
Extending Delta Debugging

- Generalize dd to test n subsets instead of 2
  - Instead of testing c_1 and c_2, test c_i and $\overline{c_i}$, for all c_i
- Cases to consider:
  - **Found** ($test(c_i) = X$)
  - **Interference** ($test(c_i) = ✓ \wedge test(\overline{c_i}) = ✓$)
  - **Preference** ($test(c_i) = ? \wedge test(\overline{c_i}) = ✓$)
    - Failure-inducing change in c_i, but apply all changes in $\overline{c_i}$ for consistency
  - **Try again** (otherwise)
- Resulting set may not be minimal

11

- The paper provides a formal definition of the extended delta debugging algorithm
  - It's a bit overwhelming, and contains all the details, plus some optimizations
- Instead, I want to talk about the overall idea, and highlight the differences
- Original dd considers two subsets, and tests c1 and c2
- Extended dd considers n subsets, and also tests the complement (for all subsets)
  - Like the original algorithm when n=2, but makes twice s many tests
- Two cases same as before
  - If ci fails, then error is in ci, so recurse
  - If ci and !ci pass, then interference, so search subsets of ci with !ci, union subsets of !ci with ci
- New cases
  - Preference, if ci unresolved but !ci passes
    - Failure must be in ci, but !ci needed for consistency
    - Search subsets of ci with !ci
    - Like "half" of interference
  - Try again
    - If all fails, try again with twice as many subsets
- Resulting set may not be minimal (would need to search all subsets)

Step	c_i	Configuration								test
1	$c_{11} = \overline{c_{12}}$	1	2	3	4	.	.	.	.	?
2	$c_{12} = \overline{c_{11}}$	.	.	.	.	5	6	7	8	?
3	c_{21}	1	2	.	.	.	.	.	.	?
4	c_{22}	.	.	3	4	.	.	.	.	?
5	c_{23}	.	.	.	.	5	6	.	.	✓
6	c_{24}	.	.	.	.	.	.	7	8	?
7	$\overline{c_{21}}$	.	.	3	4	5	6	7	8	?
8	$\overline{c_{22}}$	1	2	.	.	5	6	7	8	?
9	$\overline{c_{23}}$	1	2	3	4	.	.	7	8	✗
10	$\overline{c_{24}}$	1	2	3	4	5	6	.	.	?
11	c_{31}	1	.	.	.	5	6	.	.	✓
12	c_{32}	.	2	.	.	5	6	.	.	?
13	c_{33}	.	.	3	.	5	6	.	.	?
14	c_{34}	.	.	.	4	5	6	.	.	✓
15	c_{35}	.	.	.	.	5	6	7	.	?
16	c_{36}	.	.	.	.	5	6	.	8	✗
Result		.	.	.	.	.	.	.	8	



12

- An example with “preference” is just like “interference”
  - Except that instead of making two recursive calls, preference only makes one
  - Doesn’t need to search the complement
- Here’s an example showing try again
- We start by searching the entire set of changes C, and want 2 subsets
  - Note that c1 = !c2, c2 = !c1
- Both tests are unresolved, so we try again with 4 subsets
  - None of them fail, so we try the 4 complements
  - None of the complements pass, so we try again
- Note that 5,6 passed, and the complement failed
  - So we only need 6 subsets, we just keep r=5,6 and don’t search them
  - Finally we’re left with 8 as the failure-inducing change

Avoiding Inconsistency

- **Grouping related changes**
  - E.g. group changes time, file or directory, identifiers referenced
- **Predicting test outcomes**
  - Try to predict if a test is unresolved, instead of running it
  - E.g. order changes, assume a change requires all previous changes
- GDB case study:
  - Original run: 470 tests in 48 hours
  - Reducing inconsistencies: 289 tests in 20 hours

13

- As we saw, inconsistency can be pretty annoying to deal with
  - And it's probably very common
  - If we consider a bunch of changes to a file, it's "obvious" that many need to be made together
- Paper outlines two general approaches for reducing inconsistency
- Grouping related changes
  - Changes will be related, so we can try to guess
  - Group by time, file or directory, the identifiers they reference, functions
- Predicting inconsistency
  - E.g. order all changes, assume that a change requires all previous changes
- The paper discusses two case studies
  - Using these two techniques greatly reduced the time to find the bug
  - For GDB, they grouped by directory/file, and common identifiers
  - If a build failed, they would scan errors for identifiers and automatically apply changes that affected those identifiers

Related Work

- Andreas Zeller has published several papers related to delta debugging
  - Reducing failure-inducing input
  - Finding a failure-inducing thread schedule
  - Isolating cause-effect chains

14

- Mentioned earlier that this paper was the first in a series
  - Andreas Zeller continued research in this area, applying the delta debugging algorithm for other purposes
  - Reducing failure-inducing input
    - Giant test case, e.g. compiling a C file, make it smaller and easier to understand
  - Failure-inducing thread schedule
    - To debug multi-threaded applications
    - First, use DEJAVU tool to record and replay thread schedules
    - Then use delta debugging to find the difference between a working and failing schedule
  - Cause-effect chains
    - Think of execution as a series of program states (with variables and values)
    - Isolate the relevant variables and values
    - Similar to program slicing, but apparently more precise

Implementations

- Eclipse plug-ins
- MyDD Python module
- Delta
- C-Reduce
- Lithium
- WALA's JS Delta
- Flix delta debugging

15

- Andreas Zeller's research group has a bunch of Eclipse plug-ins and a Python module
    - Plug-ins for debugging changes, input, and program state
- Most delta debugging implementations are used for minimizing input
- Delta and C-Reduce have been used to find compiler bugs
- Lithium has been used for Firefox
- WALA, the Watson Libraries for Analysis, from IBM, has a delta debugger for reducing JS files
- Flix, the project I worked on at Waterloo, also has a delta debugging tool for reducing the input facts

Conclusions

- Delta debugging can automatically find find failure-inducing changes
- Domain knowledge can help reduce inconsistencies
- The delta debugging technique can be used to minimize test input
  - Many implementations exist

16

- Delta debugging is nice because it can do all of this automatically
  - You just need a set of changes, a test function, and an implementation of delta debugging
- Dealing with inconsistencies becomes a problem, but delta debugging can cope
  - If we use domain knowledge to group related changes, we can speed up delta debugging
- There are other applications of the technique, instead of just finding changes
  - Can minimize test cases, examine program state
  - Many implementations use delta debugging to minimize test input

Discussion

- Is delta debugging actually useful for finding changes?
  - Most implementations use delta debugging to reduce test input
  - Inconsistencies seem like they would be very, very common
- What are other applications of delta debugging?

17

- Most of the implementations are for reducing test input
  - Maybe because it's easier to implement. But is it more useful?
  - Who has heard of git bisect? Who has heard of delta debugging?
  - My hunch is that inconsistencies are very very common
    - I try to make my commits "atomic", and even then, they can be very large
  - GDB case study uses a clever technique for reducing inconsistencies, by searching for error messages
    - Skeptical about how easy this was to implement, and it seems very ad-hoc
- Open-ended question: what are other applications of delta debugging?