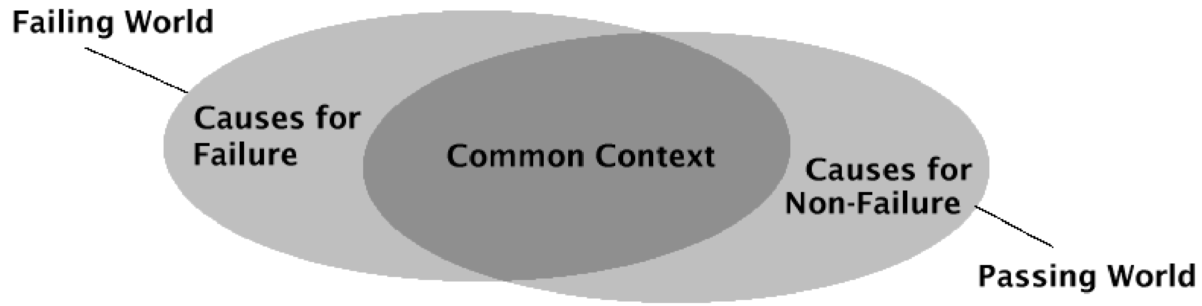
Isolating Failure Causes

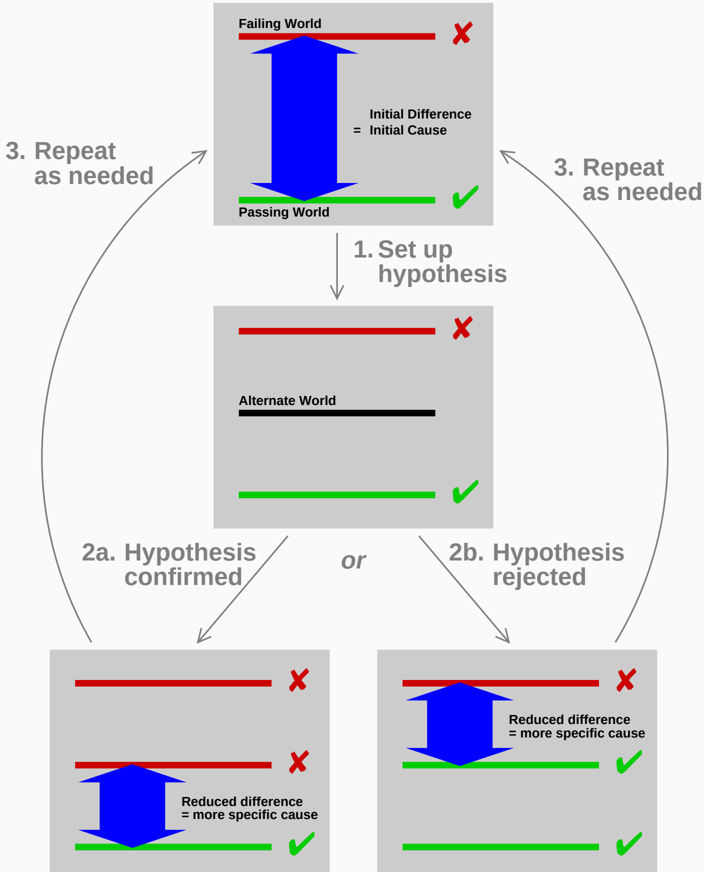
Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Causes and Alternate Worlds

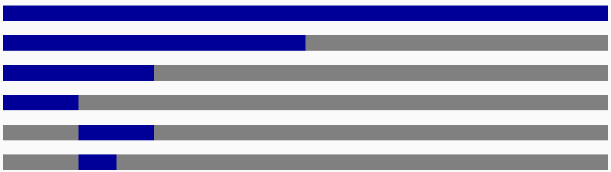


The Narrowing Process



Simplifying HTML Input

Idea: Apply *Divide and Conquer* to simplify HTML pages



1		⟨896 lines⟩	✘
2		⟨448 lines⟩	✘
3		⟨224 lines⟩	✘
4		⟨112 lines⟩	✔
5		⟨112 lines⟩	✘
6		⟨56 lines⟩	✔
⋮			
57	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨40 characters⟩	✘
58	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
59	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✔
60	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨30 characters⟩	✔
61	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨20 characters⟩	✘
62	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨10 characters⟩	✘
⋮			
75	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
76	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
77	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✔
⋮			
90	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	⟨8 characters⟩	✘

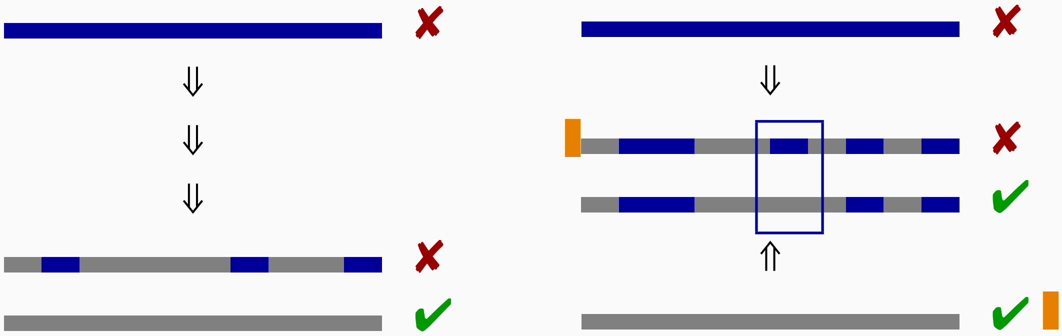
Simplified bug report: `Printing <SELECT> crashes.`

Simplifying vs. Isolating

Problem: To simplify the entire input can be expensive

Alternative approach: We do not simplify the entire input, but the *difference* with respect to a *working input*.

Simplifying **Isolating**



Larger context – but fewer tests and smaller causes

Isolating a HTML difference

#	Mozilla input	Test
1	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✘
4	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✘
7	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✔
6	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✔
5	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✔
3	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✔
2	`<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>`	✔

Isolated difference: the “<” in “<SELECT>”.

Isolating requires 7 tests, simplifying 26.

Simplification vs. Isolation

Simplification

- make *each part* of the simplified test case relevant
- removing *any part* makes the failure go away

Isolation

- find *one* relevant part of the test case
- removing *this particular part* makes the failure go away

Both simplification and isolation can be handled by delta debugging.

Recalling ddmin

$$ddmin(c_{\text{✘}}) = ddmin'(c_{\text{✘}}, 2) \quad \text{where}$$

$$ddmin'(c'_{\text{✘}}, n) = \begin{cases} ddmin'(\nabla_i, \max(n-1, 2)) & \text{if } \exists i \in \{1, \ldots, n\} \\ & \quad \cdot\, test(\nabla_i) = \text{✘} \\ ddmin'(c'_{\text{✘}}, \min(2n, |c_{\text{✘}}|)) & \text{if } 2n < |c_{\text{✘}}| \\ c'_{\text{✘}} & \text{otherwise} \end{cases}$$

with $c'_{\text{✘}} = \Delta_1 \cup \Delta_2 \cup \cdots \cup \Delta_n, \nabla_i = c'_{\text{✘}} \setminus \Delta_i$, and $\forall \Delta_i, \Delta_j \cdot \Delta_i \cap \Delta_j = \emptyset \wedge |\Delta_i| \approx |\Delta_j|$.

The *ddmin* algorithm must be *extended* to compute *differences.*

A new Algorithm

Let us try to *formalize* our issues.

Again, we have $c_{\text{✔}}$, $c_{\text{✘}}$, C, etc. as defined for *ddmin*.

Our goal is to find two sets $c'_{\text{✔}}$ and $c'_{\text{✘}}$ such that

- $\emptyset = c_{\text{✔}} \subseteq c'_{\text{✔}} \subset c'_{\text{✘}} \subseteq c_{\text{✘}}$ holds and
- the difference $\Delta = c'_{\text{✘}} - c'_{\text{✔}}$ is *1-minimal*.

Δ is *1-minimal* if

$$\forall \delta_i \in \Delta \cdot \mathit{test}(c'_{\text{✔}} \cup \{\delta_i\}) \neq \text{✔} \wedge \mathit{test}(c'_{\text{✘}} - \{\delta_i\}) \neq \text{✘}$$

holds.

Extending ddmin

We must extend *ddmin* such that it works on *two sets at a time:*

- The failing test case $c'_{✘}$ which is to be *minimized* (initially, $c'_{✘} = c_{✘}$ holds), and
- The passing test case $c'_{✔}$ which is to be *maximized* (initially, $c'_{✔} = c_{✔} = \emptyset$ holds).

A Binary Search Approach

Basic idea:

- We split the difference $\Delta = c'_{✘} - c'_{✔}$ into two subsets Δ_1 and Δ_2.

  $\Delta = \Delta_1 \cup \Delta_2$, $\Delta_1 \cap \Delta_2 = \emptyset$, and $|\Delta_1| \approx |\Delta_2|$ holds.

- We test two configurations:

  - $c'_{✘} \setminus \Delta_1 = c'_{✔} \cup \Delta_2$ and
  - $c'_{✘} \setminus \Delta_2 = c'_{✔} \cup \Delta_1$

Possible Outcomes

Starting with $c'_{\checkmark} = c_{\checkmark}$, $c'_{\times} = c_{\times}$; $\Delta = c'_{\times} - c'_{\checkmark} = \Delta_1 \cup \Delta_2$.

Test	Outcome	New $c'_{\checkmark}$	New $c'_{\times}$
$c'_{\times} \setminus \Delta_1 = c'_{\checkmark} \cup \Delta_2$	✘	$c'_{\checkmark}$	$c'_{\times} \setminus \Delta_1$
$c'_{\times} \setminus \Delta_1 = c'_{\checkmark} \cup \Delta_2$	✔	$c'_{\checkmark} \cup \Delta_2$	$c'_{\times}$
$c'_{\times} \setminus \Delta_2 = c'_{\checkmark} \cup \Delta_1$	✘	$c'_{\checkmark}$	$c'_{\times} \setminus \Delta_2$
$c'_{\times} \setminus \Delta_2 = c'_{\checkmark} \cup \Delta_1$	✔	$c'_{\checkmark} \cup \Delta_1$	$c'_{\times}$

Classical binary search with $O(\log_2 |\Delta|)$ tests.

The ddbin Algorithm

Given: $\mathit{test}, c_{✔}, c_{✘} \cdot c_{✔} \subseteq c_{✘} \wedge \mathit{test}(c_{✔}) = ✔ \wedge \mathit{test}(c_{✘}) = ✘$.

Goal: $c'_{✔}, c'_{✘} = \mathit{ddbin}(c_{✔}, c_{✘})$ such that $c_{✔} \subseteq c'_{✔} \subseteq c'_{✘} \subseteq c_{✘}$, $\mathit{test}(c'_{✔}) = ✔$, $\mathit{test}(c'_{✘}) = ✘$
and each element of $\Delta = c'_{✘} \setminus c'_{✔}$ is *relevant for the failure.*

Let $\Delta = c'_{✘} \setminus c'_{✔} = \Delta_1 \cup \Delta_2$ in

$$\mathit{ddbin}(c_{✔}, c_{✘}) = \mathit{ddbin}'(c_{✔}, c_{✘}) \text{ where}$$

$$\mathit{ddbin}'(c'_{✔}, c'_{✘}) = \begin{cases} (c'_{✔}, c'_{✘}) & \text{if } |\Delta| = 1 \\ \mathit{ddbin}'(c'_{✔}, c'_{✔} \cup \Delta_2) & \text{if } \mathit{test}(c'_{✔} \cup \Delta_2) = ✘ \\ \mathit{ddbin}'(c'_{✘} \setminus \Delta_2, c'_{✘}) & \text{if } \mathit{test}(c'_{✘} \setminus \Delta_2) = ✔ \\ \mathit{ddbin}'(c'_{✔}, c'_{✔} \cup \Delta_1) & \text{if } \mathit{test}(c'_{✔} \cup \Delta_1) = ✘ \\ \mathit{ddbin}'(c'_{✘} \setminus \Delta_1, c'_{✘}) & \text{if } \mathit{test}(c'_{✘} \setminus \Delta_1) = ✔ \end{cases}$$

(Note that $c'_{✘} \setminus \Delta_1 = c'_{✔} \cup \Delta_2$ and $c'_{✘} \setminus \Delta_2 = c'_{✔} \cup \Delta_1$ hold.)

Classical binary search!

ddbin on Mozilla input

#	Mozilla input	Test
1	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✘
4	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✘
7	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✔
6	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✔
5	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✔
3	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✔
2	<SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7>	✔

Unresolved Test Outcomes

Problem: *ddbin* does not handle *unresolved* test outcomes!

Step	GCC input	*test*
1	#define SIZE 20 ... double **mult**(...) { ... }	✘
2	#define SIZE 20	✔
3	double **mult**(...) { ... }	✘
4	double **mult**(...) { int i, j; i = 0; }	✔
5	double **mult**(...) { for(...) { ... } ... }	?

Unresolved Test Outcomes (2)

The *more we change* some input which has a *resolved* test outcome (✔ or ✘),

- the *faster* the progress in narrowing the difference, but
- the *higher* are the chances of unresolved outcomes (**?**).

If we apply *smaller changes* to the input,

- the chance to get an unresolved outcome is *smaller*, but
- the *progress* is smaller, too!

We need a *compromise* between these two approaches!

Unresolved Test Outcomes (3)

Basic idea:

1. Start with *few & large changes* first
2. If all alternatives are unresolved, apply *more & smaller* changes.

This is achieved by splitting the initial Δ not into *two* subsets, but into an *increasing* number of subsets—as in *ddmin*!

Thus, we have to *merge* the binary search of the *ddbin* algorithm with the arbitrary number of subsets as in *ddmin*.

General Delta Debugging

Given: $test, c_{\checkmark}, c_{\times} \cdot c_{\checkmark} \subseteq c_{\times} \wedge test(c_{\checkmark}) = \checkmark \wedge test(c_{\times}) = \times$.

Goal: $c'_{\checkmark}, c'_{\times} = dd(c_{\checkmark}, c_{\times})$ such that $c_{\checkmark} \subseteq c'_{\checkmark} \subseteq c'_{\times} \subseteq c_{\times}$,
$test(c'_{\checkmark}) = \checkmark$, $test(c'_{\times}) = \times$
and each element of $\Delta = c'_{\times} \setminus c'_{\checkmark}$ is *relevant for the failure.*

Let $\Delta = c'_{\times} \setminus c'_{\checkmark} = \Delta_1 \cup \cdots \cup \Delta_n$ in

$$dd(c_{\checkmark}, c_{\times}) = dd'(c_{\checkmark}, c_{\times}, 2) \text{ where}$$

$$dd'(c'_{\checkmark}, c'_{\times}, n) = \begin{cases} dd'(c'_{\checkmark}, c'_{\checkmark} \cup \Delta_i, 2) & \text{if } \exists i \cdot test(c'_{\checkmark} \cup \Delta_i) = \times \\ dd'(c'_{\times} \setminus \Delta_i, c'_{\times}, 2) & \text{if } \exists i \cdot test(c'_{\times} \setminus \Delta_i) = \checkmark \\ dd'(c'_{\checkmark} \cup \Delta_i, c'_{\times}, \max(n-1, 2)) & \text{if } \exists i \cdot test(c'_{\checkmark} \cup \Delta_i) = \checkmark \\ dd'(c'_{\checkmark}, c'_{\times} \setminus \Delta_i, \max(n-1, 2)) & \text{if } \exists i \cdot test(c'_{\times} \setminus \Delta_i) = \times \\ dd'(c'_{\checkmark}, c'_{\times}, \min(2n, |\Delta|)) & \text{if } 2n < |\Delta| \\ (c'_{\checkmark}, c'_{\times}) & \text{otherwise} \end{cases}$$

dd vs. *ddbin* vs. *ddmin*

dd is the most general of all Delta Debugging algorithms:

- If *test* returns ✔ for $c_{✔}$ only, and **?** in all other cases, then *dd* is equivalent to *ddmin*.
- If *test* never returns **?**, then *dd* is equivalent to *ddbin* (= binary search)

Consequence 1: You only need to know about *dd*. Period.

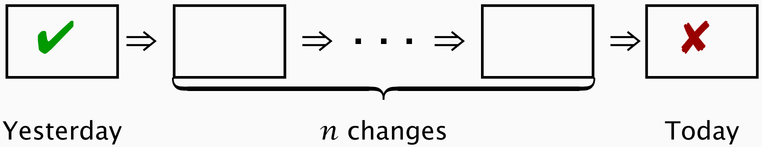
Consequence 2: *We must avoid unresolved test outcomes as good as we can* (e.g. by adding syntactic or semantic knowledge, as in simplification)

Application: Code Changes

```
Date: Fri, 31 Jul 1998 15:11:05 -0500
From: Brian Kahne <bkahne@ibmoto.com>
To: DDD Bug Reports <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. [...]
```

Yesterday, my Program Worked



Assumption: The failure is caused by one of the changes between “yesterday” and “today”.

Goal: Finding and examining this *failure-inducing change.*

Procedure: *Delta Debugging*

Trouble Ahead

In case of GDB, we have an *enormous change:*

```
$ 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 a total of *178,200 lines*.

Trouble Ahead (2)

Large changes are not the only source of trouble:

- ✘ **Granularity.** A single logical change can affect *thousands of lines of code*—but only a few lines may be responsible for the failure.

  Example: integration of large third-party changes

- ✘ **Interference.** There can be *multiple* failure-inducing changes that cause the failure only when applied together.

  Example: integration of parallel development lines

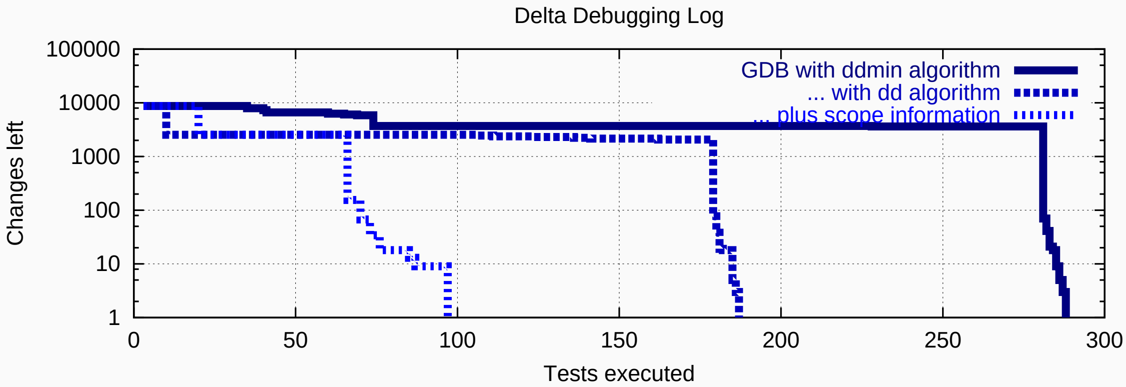
- ✘ **Inconsistency.** The generated configuration may be *inconsistent*—we do not know whether the failure occurs.

  Example: change conflict, construction failure, crash

All these are handled by delta debugging.

Isolating the GDB Change

DIFF split into 8721 changes; 370s/test on 400 MHz PC

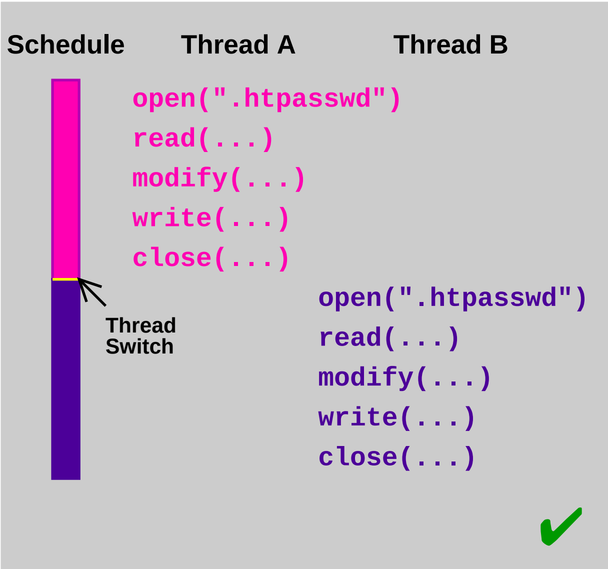


The failure-inducing code change is:

```
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.\n\
---
> "Set argument list to give program being debugged when it is started.\n\
```

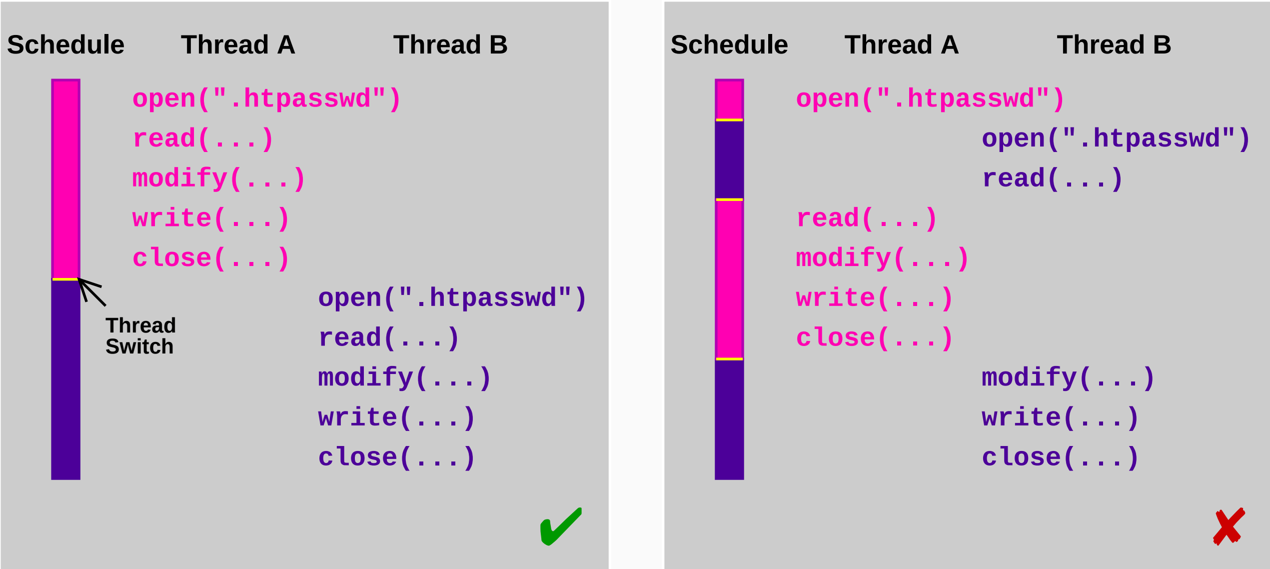
Application: Thread Schedules

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



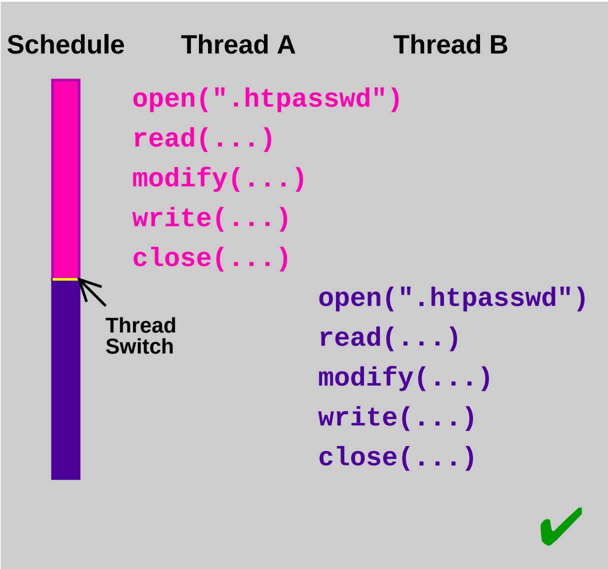
Application: Thread Schedules

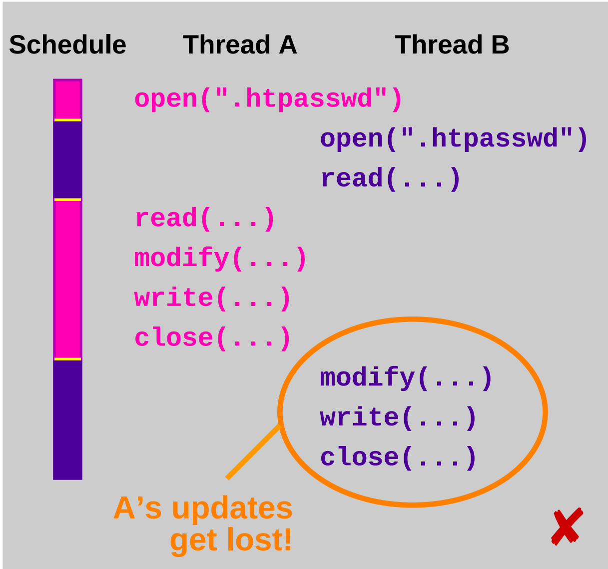
The behavior of a multi-threaded program can depend on the *thread schedule:*



Application: Thread Schedules

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

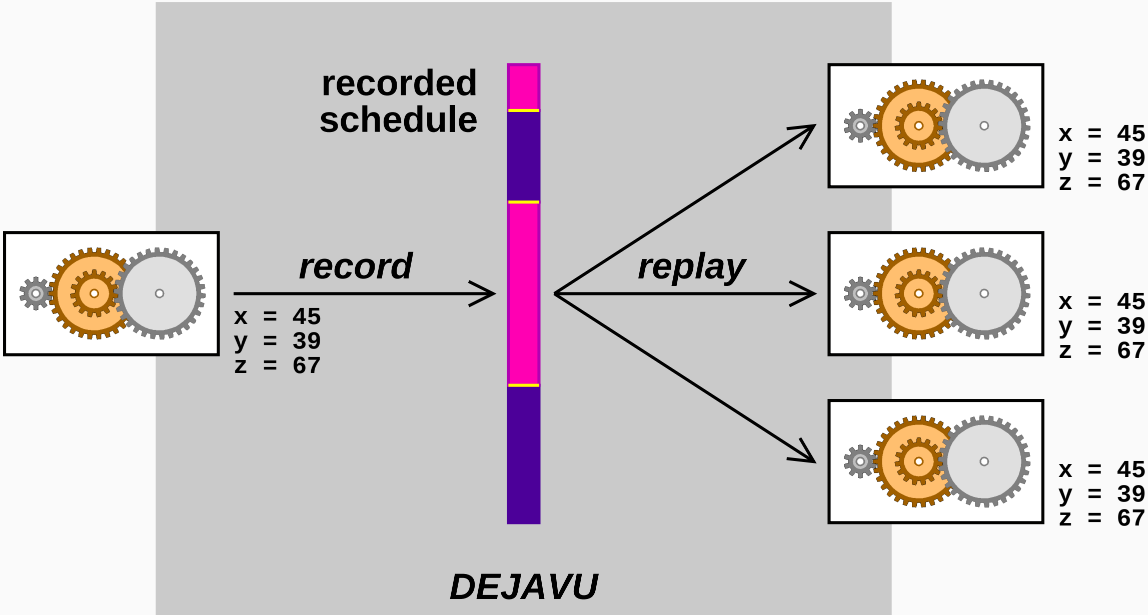




Thread switches and schedules are *nondeterministic:*
Bugs are *hard to reproduce* and *hard to isolate!*

Recording and Replaying Runs

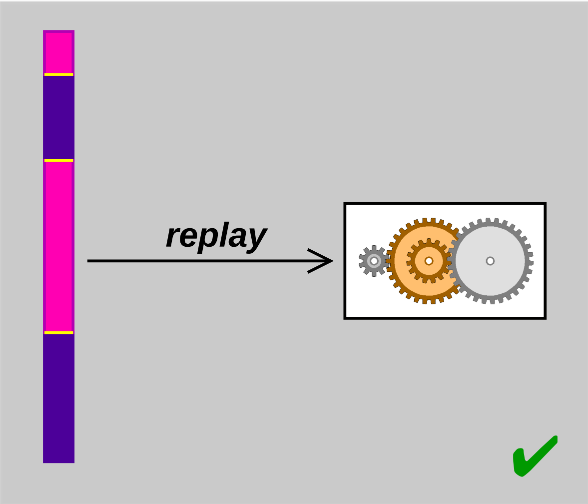
DEJAVU captures and replays program runs deterministically:

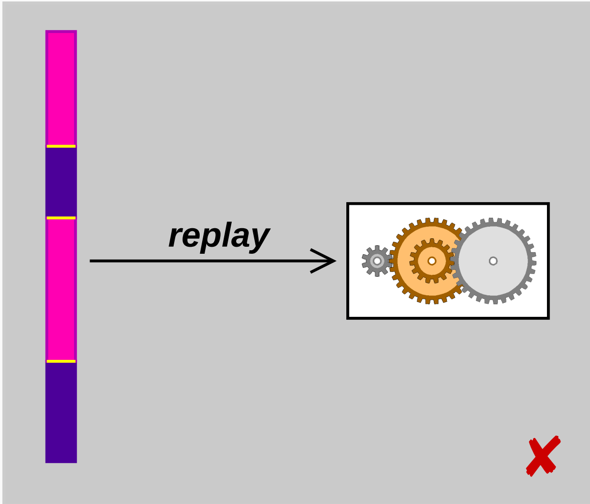


Allows simple *reproduction* of schedules and induced failures

Differences between Schedules

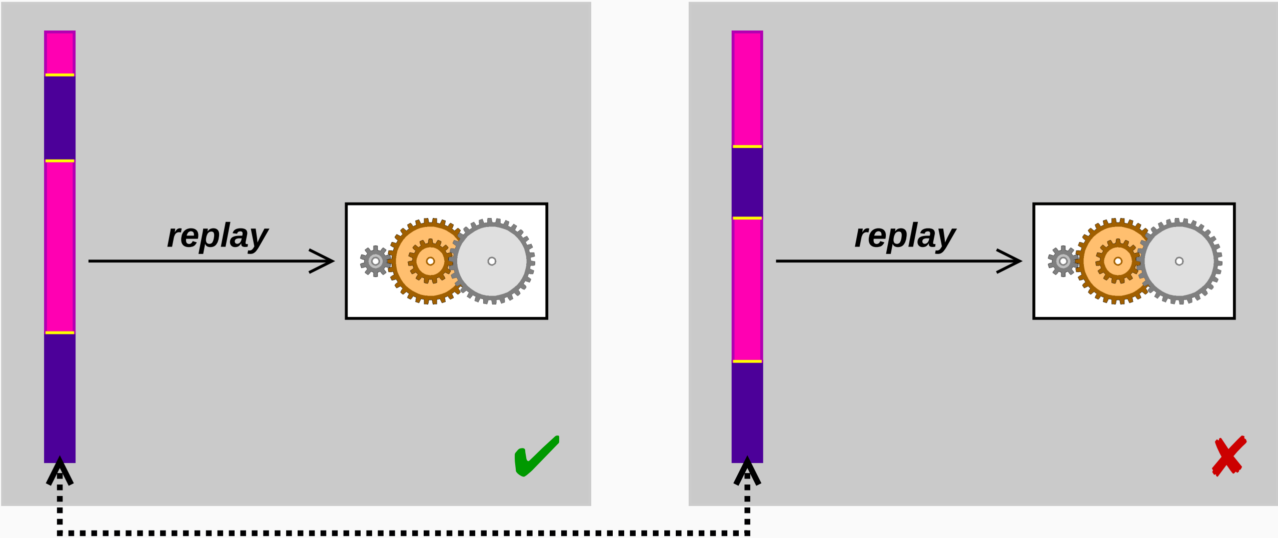
Using DEJAVU, we can consider the schedule as an *input* which determines whether the program passes or fails.





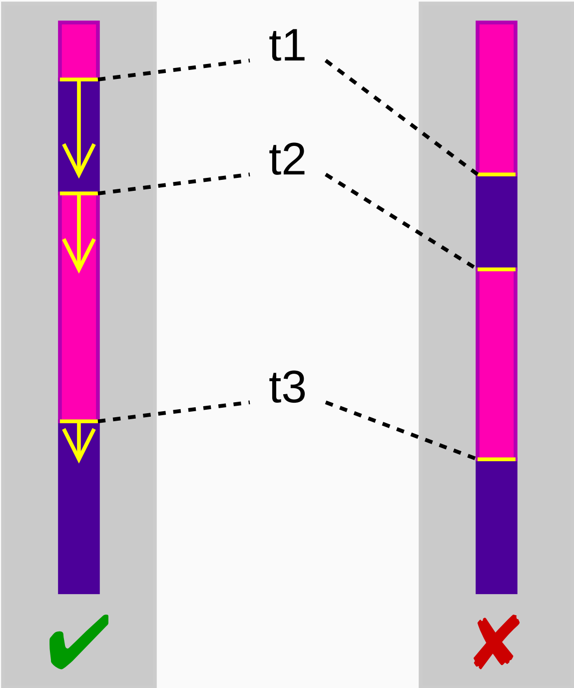
Differences between Schedules

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



The *difference* between schedules is relevant for the failure:
A *small* difference can pinpoint the failure cause

Finding Differences



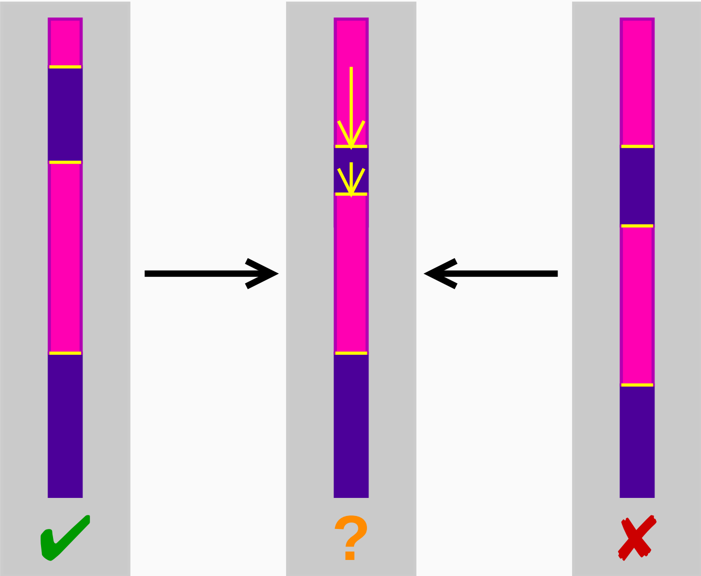
- We start with runs ✔ and ✘
- We determine the differences Δ_i between thread switches t_i:
  - t_1 occurs in ✔ at "time" 254
  - t_1 occurs in ✘ at "time" 278
  - The difference $\Delta_1 = |278 - 254|$ induces a *statement interval:* the code executed between "time" 254 and 278
  - Same applies to t_2, t_3, etc.

Our goal: *Narrow down* the difference such that only a small *relevant difference* remains, pinpointing the root cause

Isolating Relevant Differences

We use *Delta Debugging* to isolate the relevant differences

Delta Debugging applies *subsets* of differences to ✔:

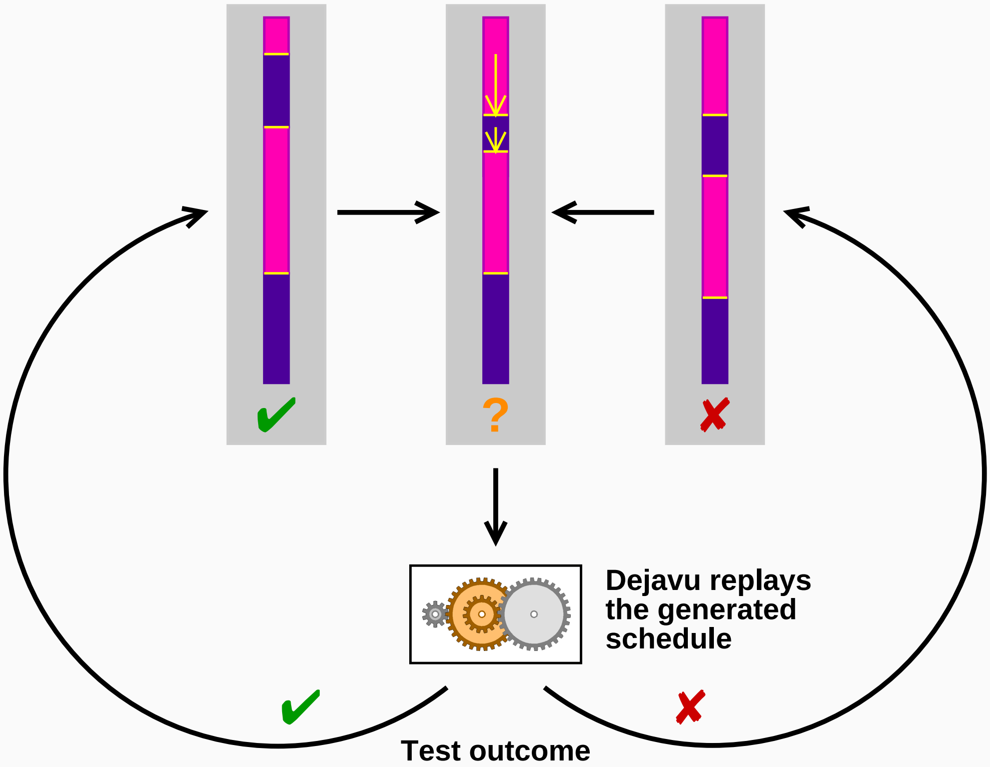


- The *entire* difference Δ_1 is applied
- Half of the difference Δ_2 is applied
- Δ_3 is not applied at all

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

The Isolation Process

Delta Debugging systematically narrows down the difference



A Real Program

We examine Test #205 of the SPEC JVM98 Java test suite:
a raytracer program depicting a dinosaur

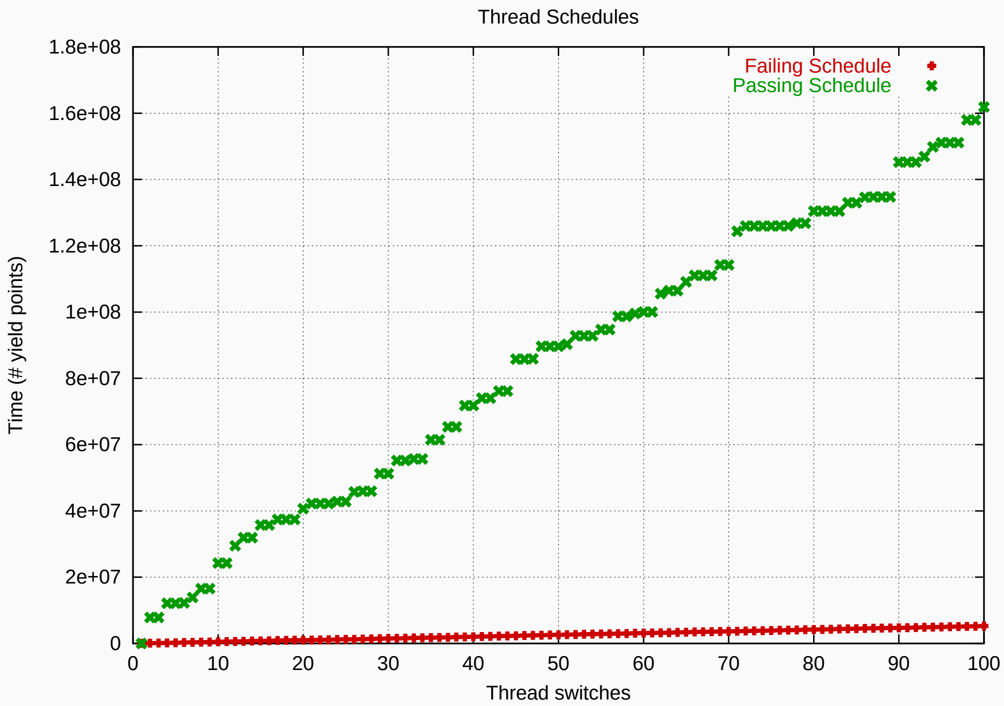
Program is single-threaded—the multi-threaded code is commented out

To test our approach,

- we make the raytracer program *multi-threaded* again
- we introduce a simple *race condition*
- we implement an *automated test* that would check whether the failure occurs or not
- we generate *random schedules* until we obtain both a passing schedule (✔) and a failing schedule (✘)

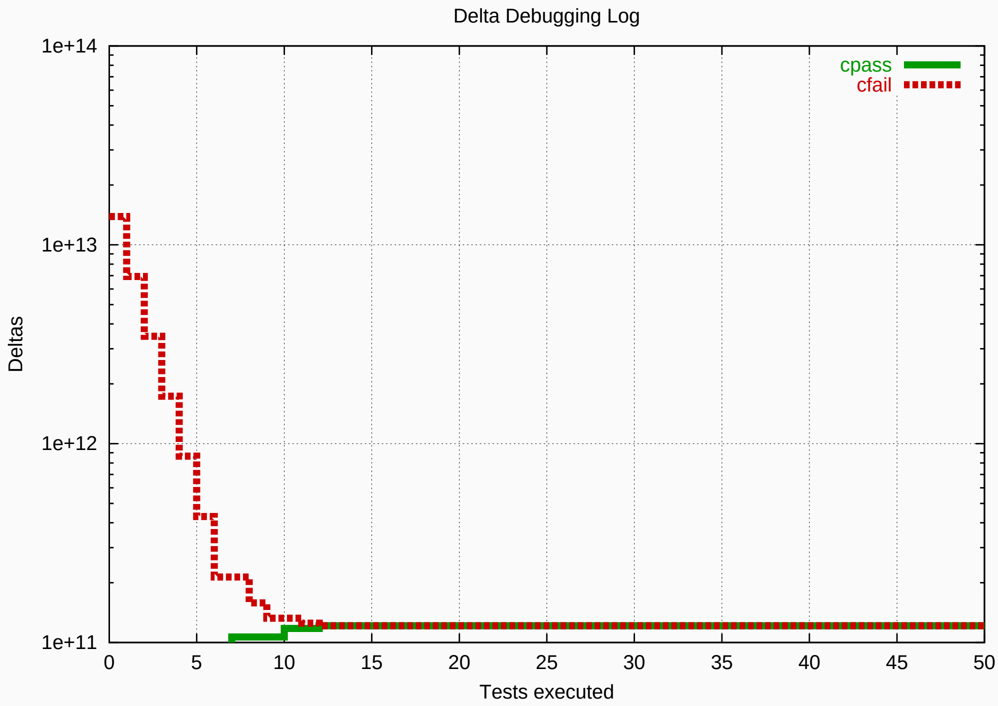
Passing and Failing Schedule

We obtain two schedules with 3,842,577,240 differences, each moving a thread switch by ±1 “time” unit



Narrowing Down the Failure Cause

Delta Debugging isolates one single difference after 50 tests:



The Root Cause of the Failure

```
public class Scene { ...
    private static int ScenesLoaded = 0;
    (more methods...)
    private
    int LoadScene(String filename) {
        int OldScenesLoaded = ScenesLoaded;
        (more initializations...)
        infile = new DataInputStream(...);
        (more code...)
        ScenesLoaded = OldScenesLoaded + 1;
        System.out.println("" +
             ScenesLoaded + " scenes loaded.");
        ...
    }
    ...
}
```

Consequence

Still, processor speed doubles almost every 18 months (Moore's Law).

Consequence: We can now afford approaches that were way too expensive only a few years ago.

Currently, the computer spends 99.9% of its time just waiting for the programmer to move the mouse.

We can exploit this to have

- Expensive program analysis
- Automated testing
- Automated debugging

Concepts

- In contrast to simplification, *isolation* finds only *one* relevant part of the test case;
  removing *this particular part* makes the failure go away
- Isolation is *much more efficient* than simplification.

Concepts (2)

➠ The general Delta Debugging algorithm *dd* extends *ddmin* to isolate failure-inducing differences.

➠ *dd* becomes an efficient binary search as soon as there are no unresolved test outcomes

➠ Delta Debugging can be applied to *arbitrary circumstances* of the program run:

- Program input
- Program code
- Program environment (i.e., the thread schedule)

as long as there is an automated test and a passing configuration to compare with.