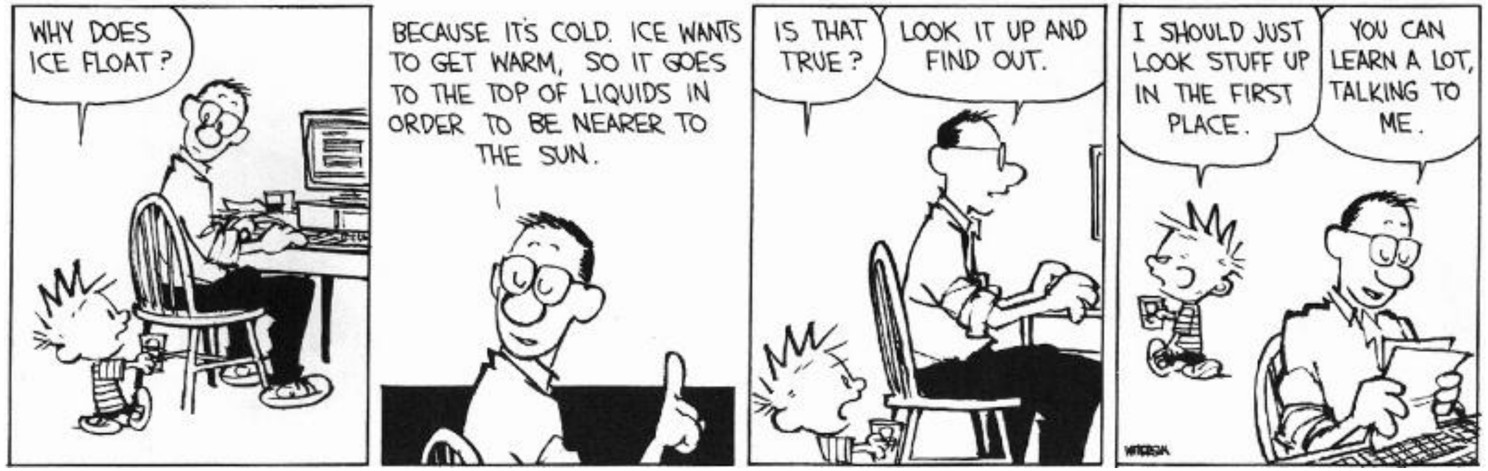
Debugging as Hypothesis Testing



The Story So Far …

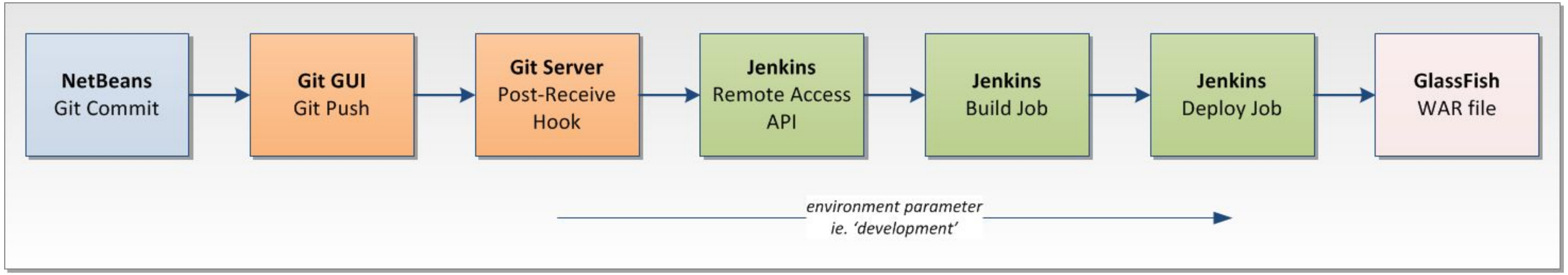
- **Quality assurance** is critical to software engineering.
  - Static and dynamic QA approaches are common
- **Defect reports** are *tracked* and *assigned* to developers for *resolution*
- Modern software is so **huge** that simple debugging approaches *do not work*
- How should we intelligently and scalably approach debugging?

One-Slide Summary

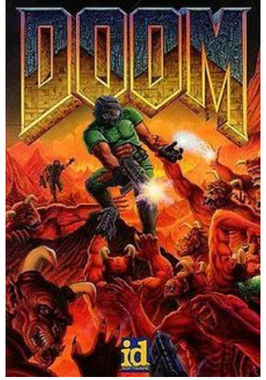
- **Delta debugging** is an automated debugging approach that finds a **minimal interesting subset** of a given set. It is very efficient.
- Delta debugging is based on **divide-and-conquer** and relies heavily on critical assumptions (**monotonicity**, **unambiguity**, and **consistency**).
- It can be used to find which code changes cause a bug, to *minimize* failure-inducing inputs, and even to find harmful thread schedules.

Debugging Case Study

- Consider this deployment pipeline: Git Server to Jenkins to GlassFish application server
  - You have a known-valid test input (NetBeans git commit) that leads to an incorrect WAR file
  - What would you do to determine which pipeline stage has the bug?



Real Life Motivation



- Mozilla developers had a large number of open bug reports in the queue that were not even simplified
- The Mozilla engineers “faced imminent doom”
- Netscape product management sent out the Mozilla Bug-A-Thon call for volunteers: people who would help simplify bug reports.
  - Simplify → turn bug reports into minimal test cases, where each part of the input matters

https://www-archive.mozilla.org/newlayout/bugathon.html

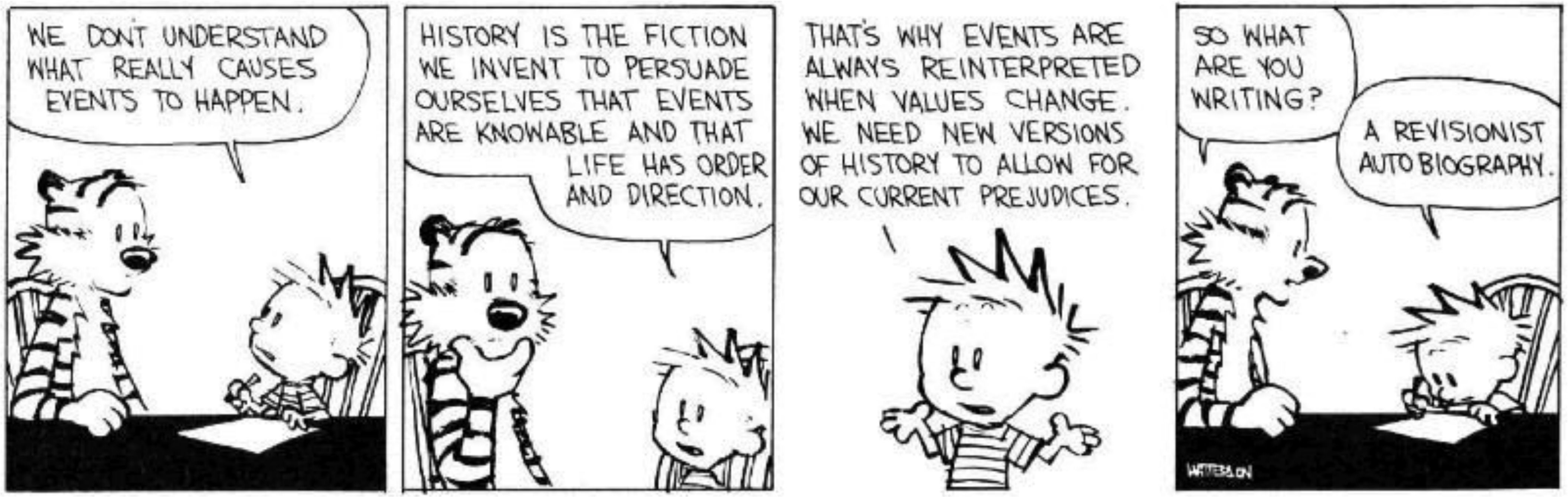
Minimizing a Mozilla Bug

- We want something that can simplify this large HTML input to just “<SELECT>” which causes the crash
- Each character in “SELECT” is relevant (see 20-26)

```
1  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
2  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
3  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
4  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
5  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
6  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
7  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
8  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
9  <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
10 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
11 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
12 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
13 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
14 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
15 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
16 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
17 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
18 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✗
19 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
20 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
21 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
22 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
23 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
24 <SELECT␣NAME="priority"␣MULTIPLE␣SIZE=7> ✔
25 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✔
26 <SELECT NAME="priority"␣MULTIPLE␣SIZE=7> ✗
```

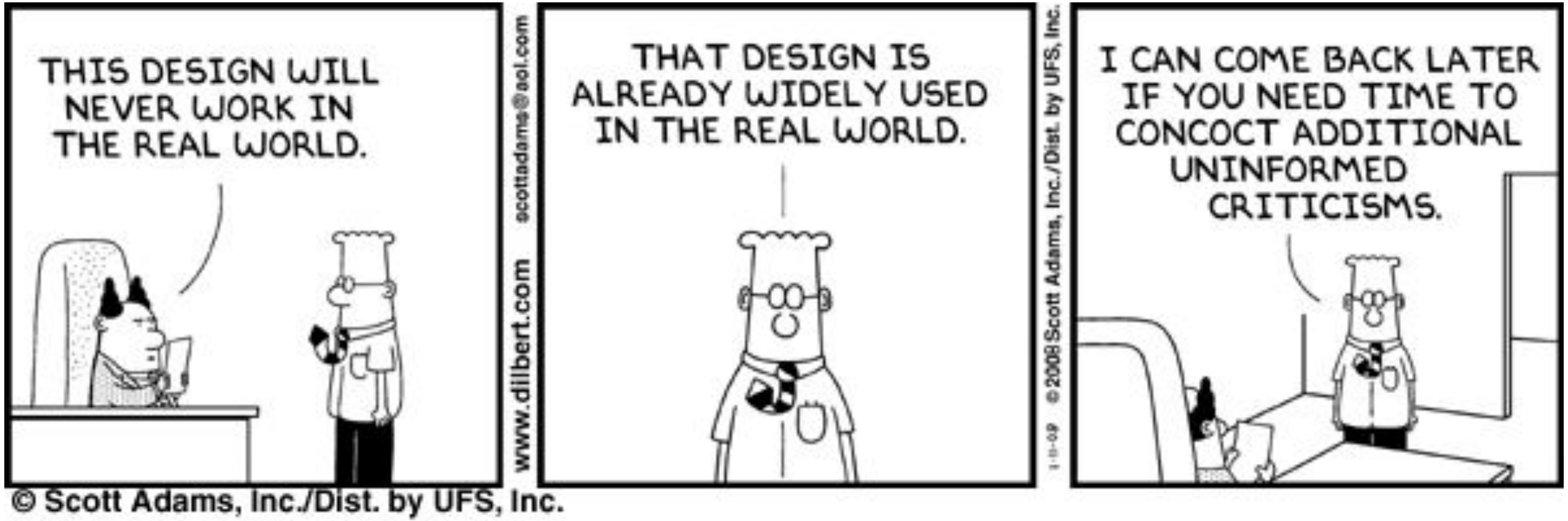
Often people who encounter a bug spend a lot of time investigating which changes to the input file will make the bug go away and which changes will not affect it.

— Richard Stallman, *Using and Porting* GNU CC



Delta Debugging

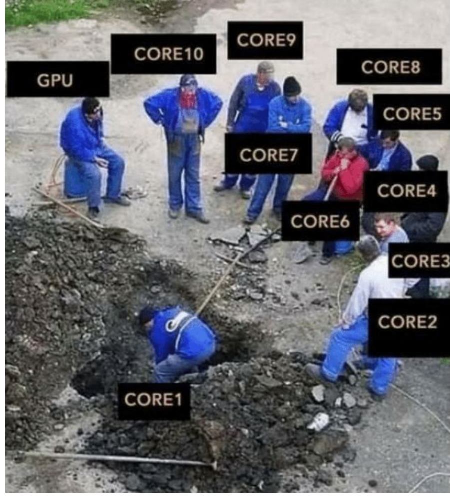
- Three Problems: One Common Approach
  - Simplifying Failure-Inducing Input
  - Isolating Failure-Inducing Thread Schedules
  - Identifying Failure-Inducing Code Changes



Failure-Inducing Input

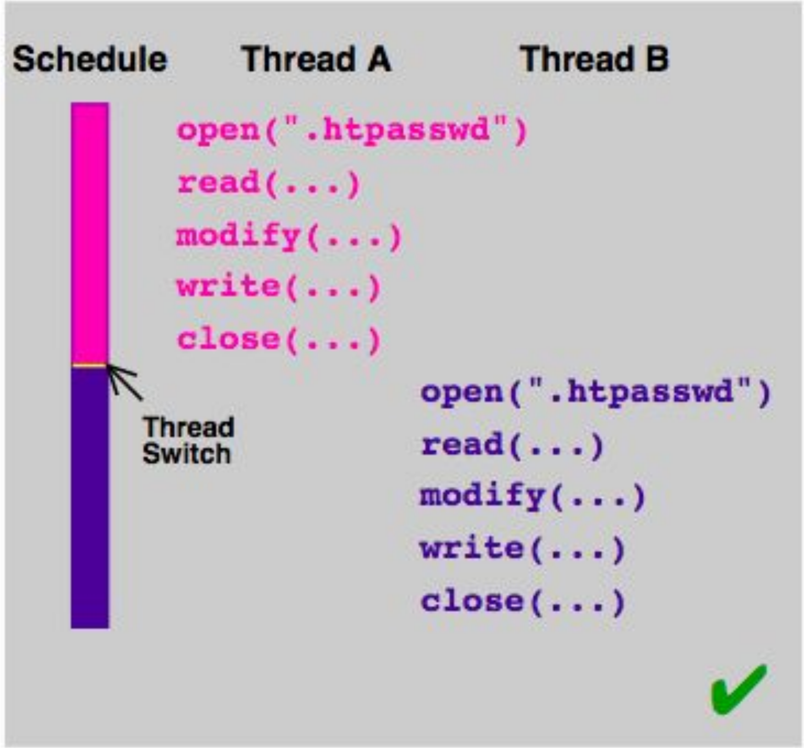
- Having a test **input** may not be enough
  - Even if you know the suspicious code, the input may be too **large** to step through
- This HTML input makes a version of Mozilla crash. Which portion is relevant?

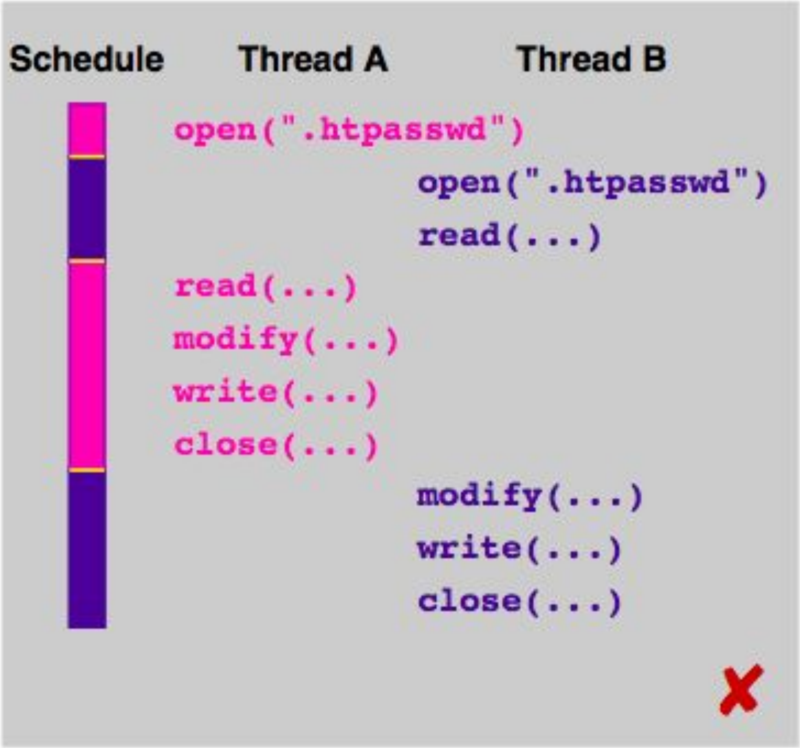
```
<td align=left valign=top>
<SELECT NAME="op_sys" MULTIPLE SIZE=7>
<OPTION VALUE="All">All<OPTION VALUE="Windows 3.1">Windows 3.1<OPTION VALUE="Windows 95">Windows 95<OPTION VALUE="Windows
98">Windows 98<OPTION VALUE="Windows ME">Windows ME<OPTION VALUE="Windows 2000">Windows 2000<OPTION VALUE="Windows
NT">Windows NT<OPTION VALUE="Mac System 7">Mac System 7<OPTION VALUE="Mac System 7.5">Mac System 7.5<OPTION VALUE="Mac
System 7.6.1">Mac System 7.6.1<OPTION VALUE="Mac System 8.0">Mac System 8.0<OPTION VALUE="Mac System 8.5">Mac System
8.5<OPTION VALUE="Mac System 8.6">Mac System 8.6<OPTION VALUE="Mac System 9.x">Mac System 9.x<OPTION VALUE="MacOS X">MacOS
X<OPTION VALUE="Linux">Linux<OPTION VALUE="BSDI">BSDI<OPTION VALUE="FreeBSD">FreeBSD<OPTION VALUE="NetBSD">NetBSD<OPTION
VALUE="OpenBSD">OpenBSD<OPTION VALUE="AIX">AIX<OPTION VALUE="BeOS">BeOS<OPTION VALUE="HP-UX">HP-UX<OPTION
VALUE="IRIX">IRIX<OPTION VALUE="Neutrino">Neutrino<OPTION VALUE="OpenVMS">OpenVMS<OPTION VALUE="OS/2">OS/2<OPTION
VALUE="OSF/1">OSF/1<OPTION VALUE="Solaris">Solaris<OPTION VALUE="SunOS">SunOS<OPTION VALUE="other">other</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="priority" MULTIPLE SIZE=7>
<OPTION VALUE="--">--<OPTION VALUE="P1">P1<OPTION VALUE="P2">P2<OPTION VALUE="P3">P3<OPTION VALUE="P4">P4<OPTION
VALUE="P5">P5</SELECT>
</td>
<td align=left valign=top>
<SELECT NAME="bug_severity" MULTIPLE SIZE=7>
<OPTION VALUE="blocker">blocker<OPTION VALUE="critical">critical<OPTION VALUE="major">major<OPTION
VALUE="normal">normal<OPTION VALUE="minor">minor<OPTION VALUE="trivial">trivial<OPTION VALUE="enhancement">enhancement</SELECT>
</tr>
</table>
```



Thread Scheduling

- Multithreaded programs can be non-deterministic
  - Can we find simple, bug-inducing thread schedules?





Code Changes

- A new version of GDB has a UI bug
  - The old version does not have that bug
- 178,000 lines of code have been modified between the two versions
  - Where is the bug?
  - These days: continuous integration testing helps
    - … but does not totally solve this. Why?

```
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\
```

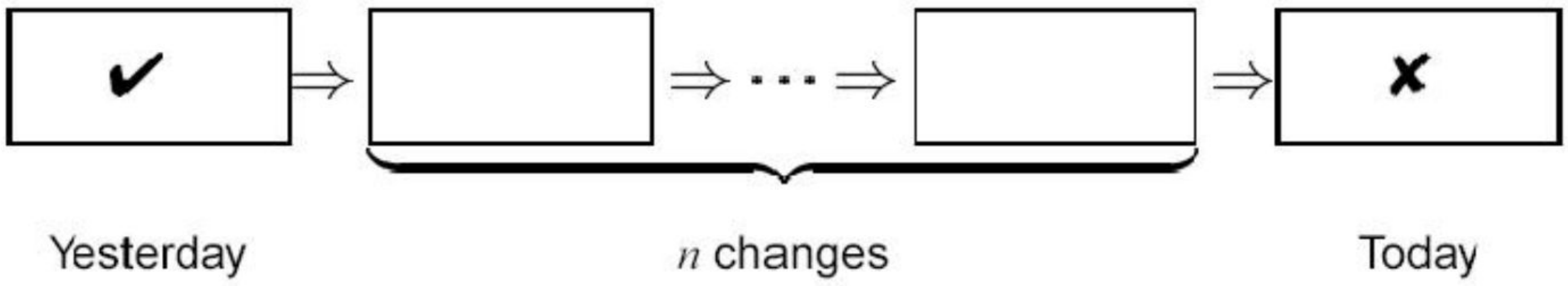
What is a Difference?

- Debugging deals with "a large number of **different** things"
- With respect to debugging, a **difference** is a **change** (in the program configuration or state) that may lead to alternate observations
  - Difference in the **input**: different character or bit in the input stream
  - Difference in **thread schedule**: difference in the time before a given thread preemption is performed
  - Difference in **code**: different statements or expressions in two versions of a program
  - Difference in program **state**: different values of internal variables

Unified Solution

- **Abstract** Debugging Problem:
  - Find which part of something (= which input, which change, etc.) determines the failure
  - "Find the smallest subset of a given set that is still interesting"
- **Divide and Conquer**
  - Applied to: working and failing inputs, code versions, thread schedules, program states, etc.

Yesterday, My Program Worked
Today, It Does Not

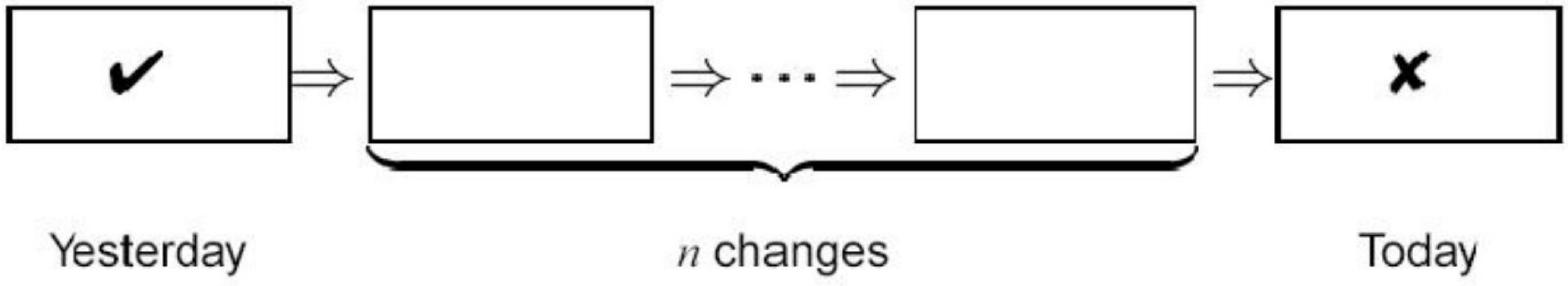


- We will iteratively
  - **Hypothesize** that a small subset is interesting
    - Example: change set {1,3,8} causes the bug
  - Run tests to falsify that hypothesis – how?

Delta Debugging (Interface)

- Given
  - a set $C = \{c_1, \ldots, c_n\}$ (of changes)
  - a function ***Interesting*** : a set of changes → Yes or No
  - Interesting(C) = Yes, Interesting({}) = No
  - Interesting is monotonic, unambiguous and consistent (more on these later)
- The **delta debugging** algorithm returns a minimal "Interesting" subset **M** of C:
  - Interesting(M) = Yes
  - Forall m in M, Interesting(M \ {m}) = No

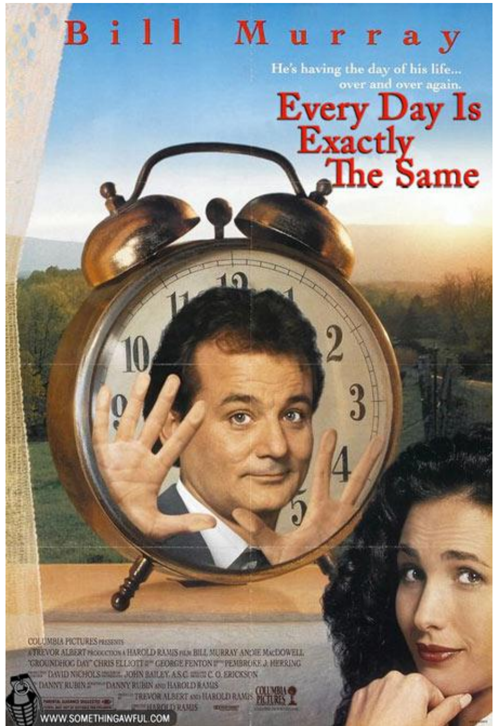
Example Use of Delta Debugging



- C = the set of *n* changes
- Interesting(X) = Apply the changes in X to Yesterday's version and compile. Run the result on the test.
  - If it fails, return “Yes” (X is an interesting failure-inducing change set),
  - otherwise return “No” (X is too small and does not induce the failure)

Naïve Approach

- We could just try all subsets of C to find the smallest one that is Interesting
  - Problem: if |C| = N, this takes 2^N time
  - Recall: real-world software is huge
- We want a polynomial-time solution
  - Ideally one that is more like log(N)
  - Or we'll loop "forever"



Algorithm Candidate

```
/* Precondition: Interesting({c_1 … c_n}) = Yes */
DD({c_1, …, c_n}) =
 if n = 1 then return {c_1}
 let P1 = {c_1, … c_n/2}
 let P2 = {c_n/2+1, …, c_n}
 if Interesting(P1) = Yes
   then return DD(P1)
   else return DD(P2)
```

So far, this is just binary search!

It won't work if you need a big subset (with >1 elements) to be Interesting.

Useful Assumptions

- Any subset of changes may be Interesting
  - Not just singleton subsets of size 1 (cf. bsearch)
- Interesting is **Monotonic**
  - $\text{Interesting}(X) \longrightarrow \text{Interesting}(X \cup \{c\})$
- Interesting is **Unambiguous**
  - $\text{Interesting}(X)$ & $\text{Interesting}(Y) \longrightarrow \text{Interesting}(X \cap Y)$
- Interesting is **Consistent**
  - Interesting(X) = Yes or Interesting(X) = No
  - (Some formulations: Interesting(X) = Unknown)

Delta Debugging Insights

		Interesting(P2)	
		Yes	No
Interesting(P1)	Yes	This	Here
	No	Here	That

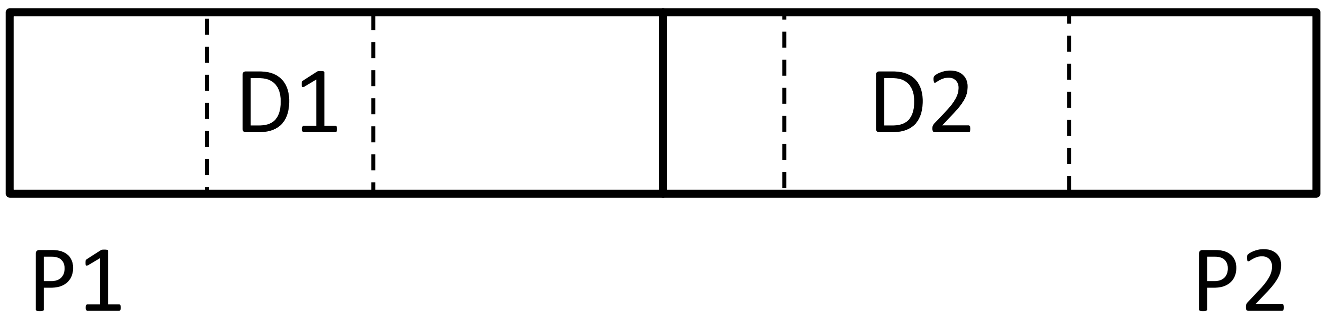
- Basic Binary Search
  - Divide C into P1 and P2
  - If Interesting(P1) = Yes then recurse on P1
  - If Interesting(P2) = Yes then recurse on P2
- **At most one case** can apply (by Unambiguous)
- By Consistency, the only other possibility is
  - (Interesting(P1) = No) *and* (Interesting(P2) = No)
  - What happens in such a case?

Interference: Interesting(P1) = No *and* Interesting(P2) = No

- By Monotonicity
  - If Interesting(P1) = No and Interesting(P2) = No
  - Then no subset of P1 alone or subset of P2 alone is Interesting
- So the Interesting subset must use a combination of elements from P1 and P2
- In Delta Debugging, this is called **interference**
  - Basic binary search does *not* have to contend with this issue

Interference Insight

(hardest part of this lecture?)



- Consider P1
  - Find a minimal subset D2 of P2
  - Such that Interesting(P1 ∪ D2) = Yes
- Consider P2
  - Find a minimal subset D1 of P1
  - Such that Interesting(P2 ∪ D1) = Yes
- Then by Unambiguous
  - Interesting((P1 ∪ D2) ∩ (P2 ∪ D1)) = Interesting(D1 ∪ D2) is also minimal

Example: {3,6} Is Smallest Interesting Subset of {1, ..., 8}

<u>1</u> <u>2</u> <u>3</u> <u>4</u> <u>5</u> <u>6</u> <u>7</u> <u>8</u> <u>Interesting?</u>

Example: Use DD to find the smallest interesting subset of {1, …, 8}

What do you think DD will do here?
List the first three steps.

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

<u>1</u> <u>2</u> <u>3</u> <u>4</u> <u>5</u> <u>6</u> <u>7</u> <u>8</u> <u>Interesting?</u>

1 2 3 4

5 6 7 8

First Step:
Partition C = {1, …, 8} into
P1 = {1, …, 4} and P2 = {5, …, 8}

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

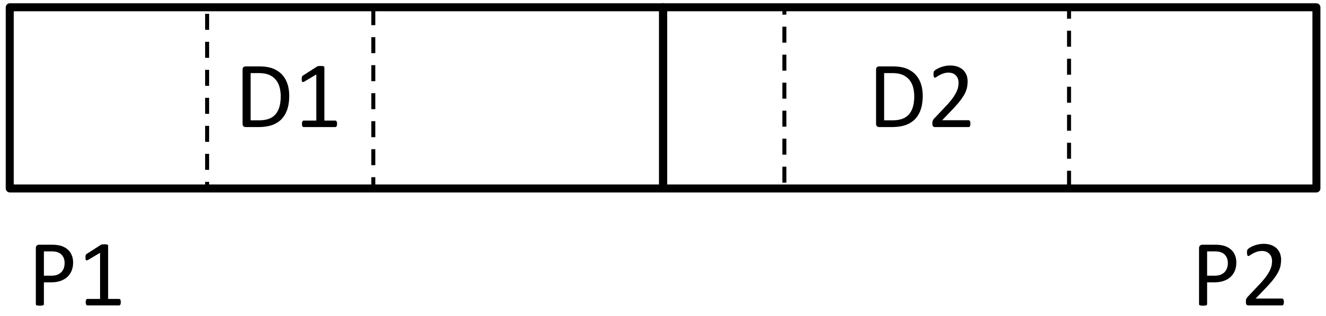
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					???
				5	6	7	8	???

↑

Second Step:
Test P1 and
P2

Example: {3,6} Is Smallest Interesting Subset of {1, ..., 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No



Interference! Sub-Step:
Find minimal subset D1 of P1
such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No

Interference! Sub-Step:
Find minimal subset D1 of P1
such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	2	3	4	5	6	7	8	Interesting?
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	???

Interference! Sub-Step:
Find minimal subset D1 of P1
such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No

Interference! Sub-Step:
Find minimal subset D1 of P1
such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	??

Interference! Sub-Step:
Find minimal subset D1 of P1
such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	2	3	4	5	6	7	8	Interesting?
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	Yes

Interference! Sub-Step:
Find minimal subset D1 of P1 such that Interesting(D1 + P2)

Example: {3,6} Is Smallest Interesting Subset of {1, ..., 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	Yes

Interference! Sub-Step:
Find minimal subset D1 of P1 such that Interesting(D1 + P2)

Are we done?

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	Yes
		3		5	6	7	8	??

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	2	3	4	5	6	7	8	Interesting?
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	Yes
		3		5	6	7	8	Yes

D1 = {3}

Example: {3,6} Is Smallest Interesting Subset of {1, ..., 8}

1	2	3	4	5	6	7	8	Interesting?
1	2	3	4					No
				5	6	7	8	No
		3		5	6	7	8	Yes

D1 = {3}

Just one half.
Need second half!

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No

D1 = {3}

Now find D2!

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2	3	4	5	6			??

D1 = {3}

Now find D2!

Example: {3,6} Is Smallest Interesting Subset of {1, ..., 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2	3	4	5	6			Yes

D1 = {3}

Now find D2!

What's next?

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2	3	4	5	6			Yes
1	2	3	4	5				No
1	2	3	4		6			Yes

D1 = {3}
D2 = {6}

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	2	3	4	5	6	7	8	Interesting?
1	2	3	4					No
				5	6	7	8	No
		3		5	6	7	8	Yes
1	2	3	4		6			Yes

D1 = {3}
D2 = {6}

Final Answer:
How to combine D1, D2?

Example: {3,6} Is Smallest Interesting Subset of {1, …, 8}

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**Interesting?**
1	2	3	4					No
				5	6	7	8	No
1	2			5	6	7	8	No
		3	4	5	6	7	8	Yes
		3		5	6	7	8	Yes
1	2	3	4	5	6			Yes
1	2	3	4	5				No
1	2	3	4		**6**			Yes

D1 = {3}
D2 = {6}

Final Answer:
{3, 6}

Delta Debugging Algorithm

Initially, empty set; but during run, not empty

Initially, entire set; but during run, a subset

DD(P, $\{c_1, \ldots, c_n\}$) =

Precondition: P is not interesting, but P U $\{c_1, \ldots, c_n\}$ is

Postcondition: minimal subset of $\{c_1, \ldots, c_n\}$ such that "P U this subset" is interesting

if n = 1 then return $\{c_1\}$

let P1 = $\{c_1, \ldots c_{n/2}\}$

let P2 = $\{c_{n/2+1}, \ldots, c_n\}$

if Interesting(P ∪ P1) = Yes then return DD(P, P1)

if Interesting(P ∪ P2) = Yes then return DD(P, P2)

else return DD(P ∪ P2, P1) ∪ DD(P ∪ P1, P2)

Algorithmic Complexity

- If a single change induces the failure
  - DD is logarithmic: 2 * log |C|
  - Why?
- Otherwise, DD is linear
  - Assuming constant time per “Interesting” check
  - Is this realistic? (cf. “AOTBE”)
- If Interesting can return Unknown
  - DD is quadratic: $|C|^2 + 3|C|$
  - If all tests are Unknown except last one (unlikely)

Questioning Assumptions

(assumptions are restated here for convenience)

- All three key assumptions are questionable
- Interesting is **Monotonic**
  - $\text{Interesting}(X) \rightarrow \text{Interesting}(X \cup \{c\})$
- Interesting is **Unambiguous**
  - $\text{Interesting}(X)\ \&\ \text{Interesting}(Y) \rightarrow \text{Interesting}(X \cap Y)$
- Interesting is **Consistent**
  - Interesting(X) = Yes or Interesting(X) = No
  - (Some formulations: Interesting(X) = Unknown)

Not Monotonic

- ~~Montonic: If X is Interesting, any superset of X is interesting~~
- What if the world is not monotonic?
  - For example, Interesting({1,2}) = Yes but Interesting({1,2,3,4}) = No
- Then DD may still find *an* Interesting subset
  - Thought questions: Will it be minimal? How long will it take?

Ambiguity
(a 481 student found this counterexample!)

- ~~Unambiguous: the interesting failure is caused by one subset (and not independently by two disjoint subsets)~~
- What if the world is ambiguous?
- Then DD (as presented here) may *not* find an Interesting subset
- Hint: trace DD on Interesting({2, 8}) = yes, Interesting({3, 6}) = yes, but Interesting({2, 8} ∩ {3, 6}) = no.
  - DD returns {2,6} :-(.

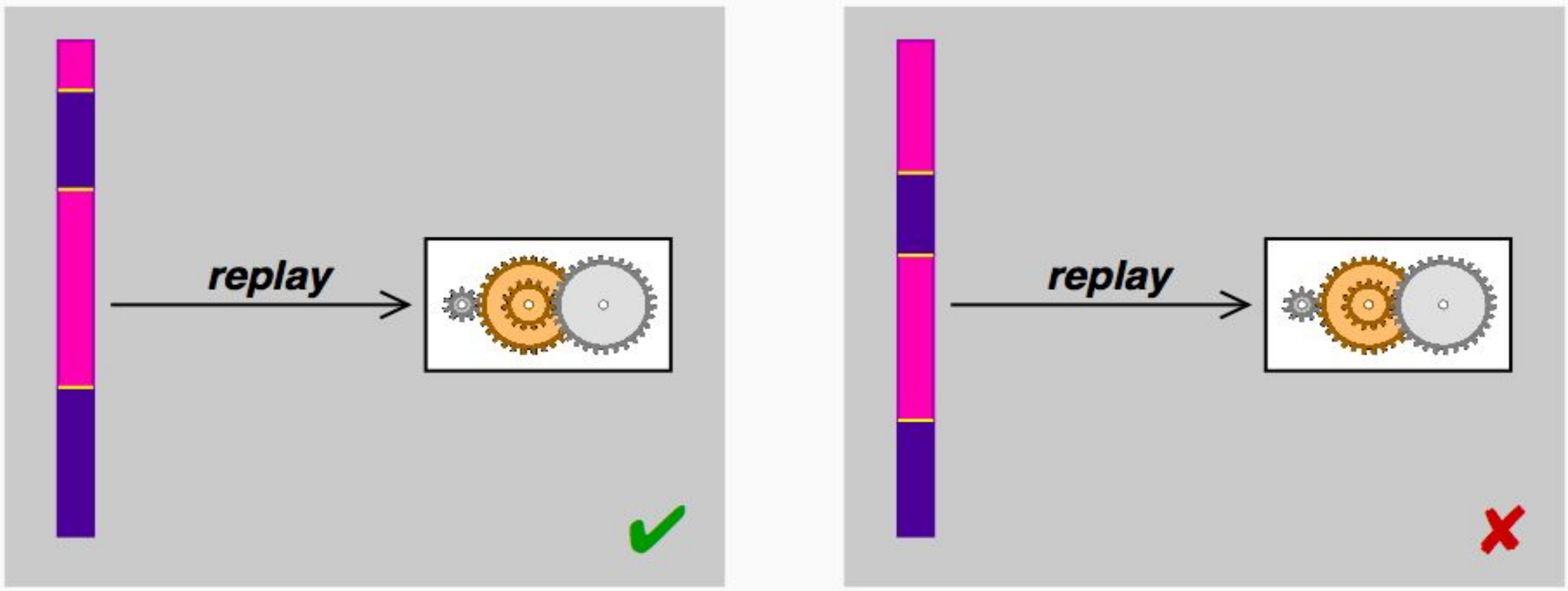
Inconsistency

- ~~Consistent: We can evaluate every subset to see if it is Interesting or not~~
  - What if the world is not consistent?
- Example: we are minimizing changes to a program to find patches that make it crash

  Some subsets may not build or run!
  - Integration Failure: a change may depend on earlier changes
  - Construction failure: some subsets may yield programs with parse errors or type checking errors (cf. HW3!)
  - Execution failure: program executes strangely or does not terminate, test outcome is unresolved

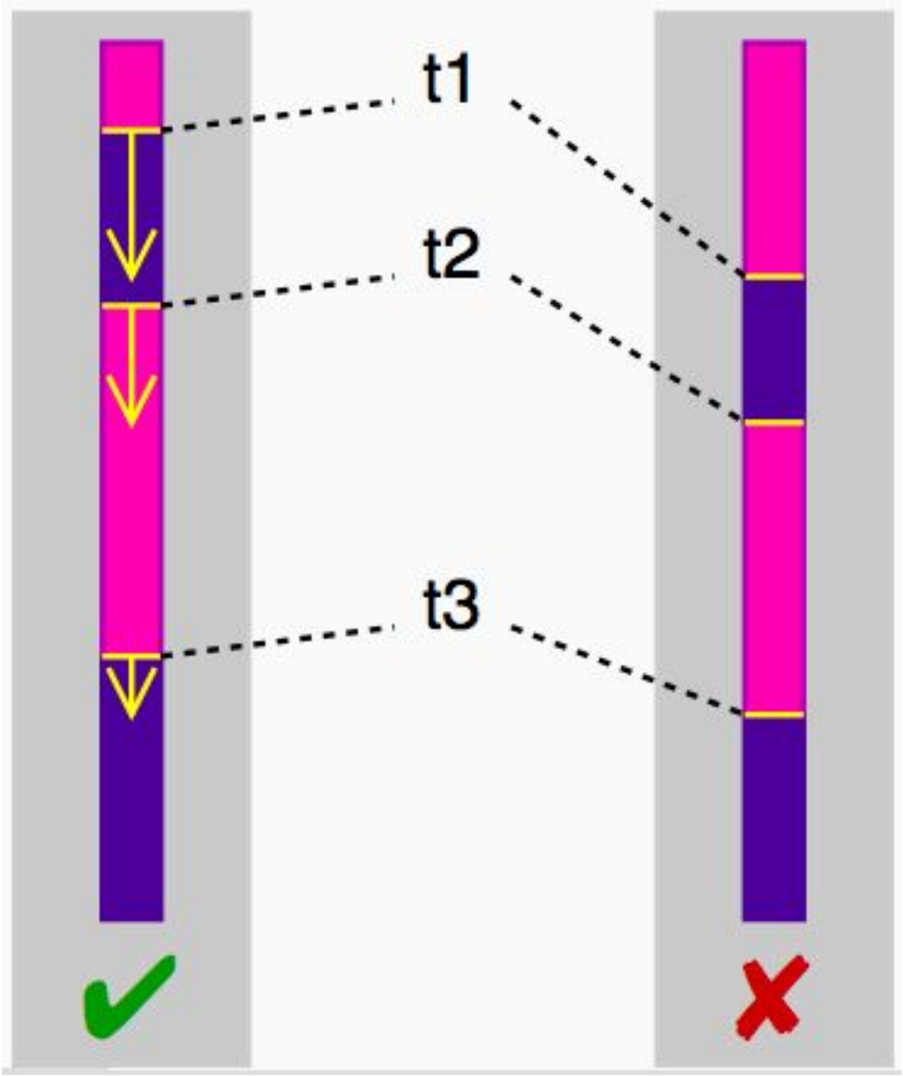
Delta Debugging Thread Schedules

- DejaVu tool by IBM, CHESS by Microsoft, etc.
- The thread schedule becomes part of the input
- We can control when the scheduler preempts one thread



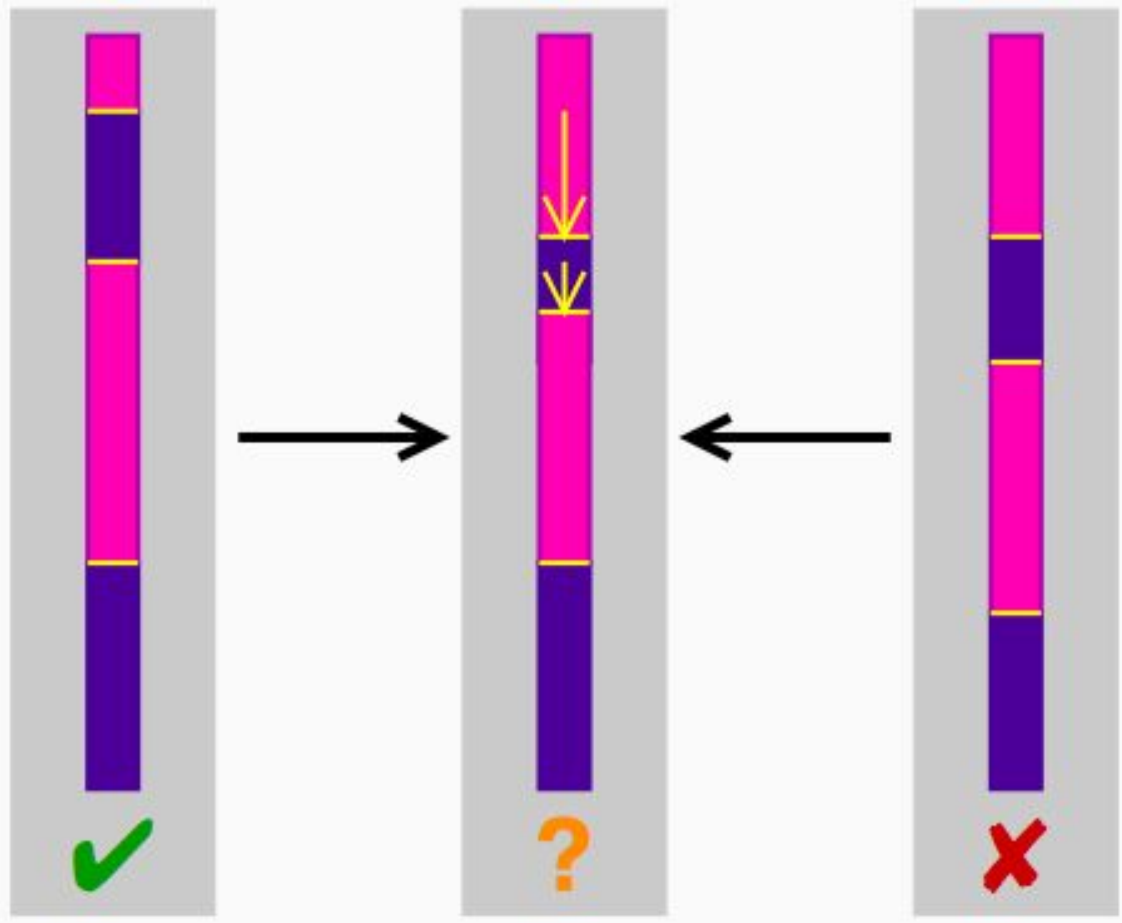
Differences in Thread Scheduling

- Starting point
  - Passing run
  - Failing run
- Differences (for t1)
  - T1 occurs in passing run at time 254
  - T1 occurs in failing run at time 278



Differences in Thread Scheduling

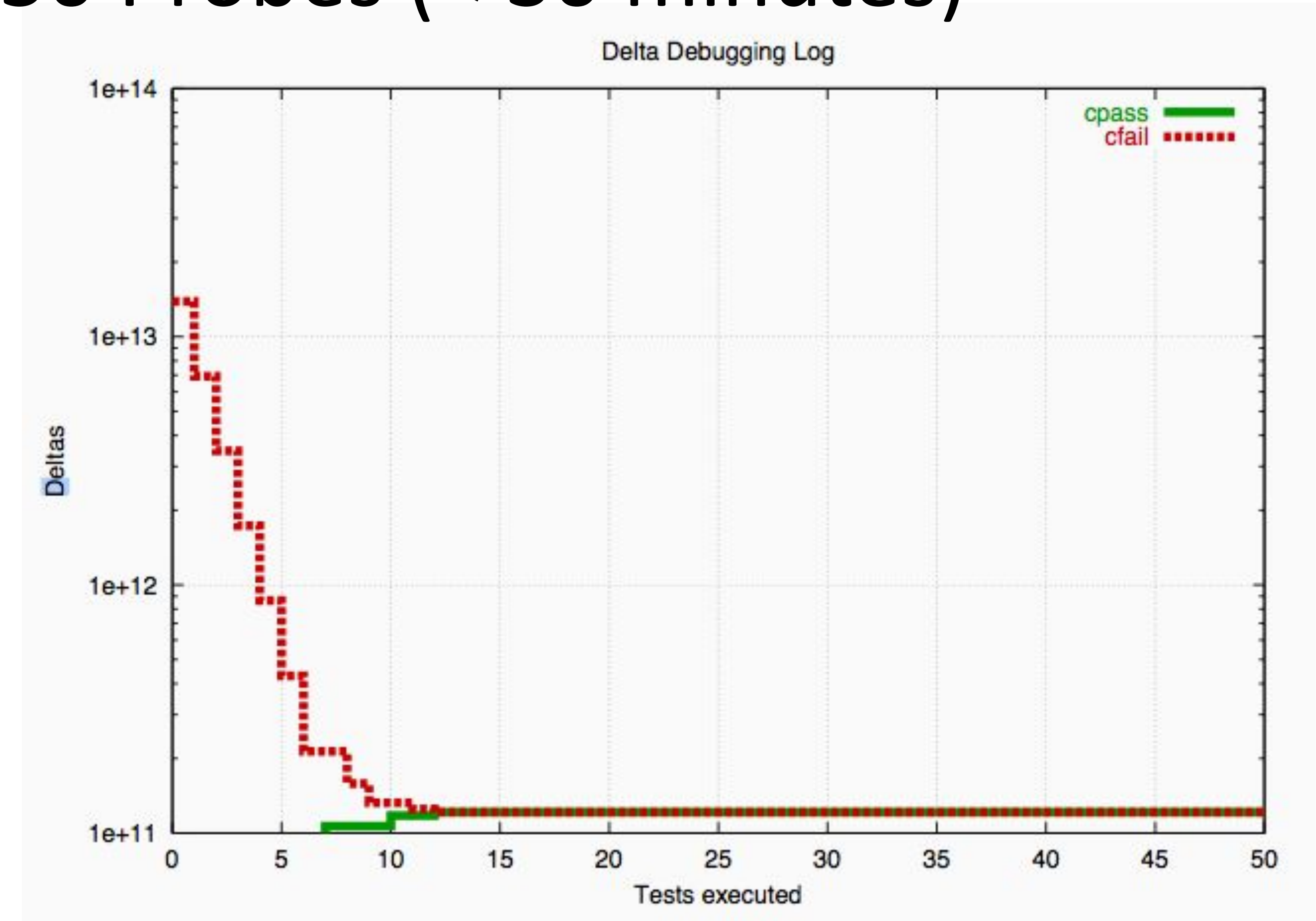
- We can build new test cases by mixing the two schedules to isolate the relevant differences



Does It Work?

- Test #205 of SPEC JVM98 Java Test Suite
  - Multi-threaded raytracer program
  - Simple race condition
  - Generate random schedules to find a passing schedule and a failing schedule (to get started)
- Differences between passing and failing
  - 3,842,577,240 differences (!)
  - Each difference moves a thread switch time by +1 or -1

DD Isolates One Difference After 50 Probes (< 30 minutes)



Pin-Pointing The Failure

- The failure occurs iff thread switch #33 occurs at yield point 59,772,127 (line 91) instead of 59,772,126 (line 82) → race on *which variable*?

```
25  public class Scene { ...
44      private static int ScenesLoaded = 0;
45      (more methods...)
81      private
82      int LoadScene(String filename) {
84          int OldScenesLoaded = ScenesLoaded;
85          (more initializations...)
91          infile = new DataInputStream(...);
92          (more code...)
130         ScenesLoaded = OldScenesLoaded + 1;
131         System.out.println("" +
                ScenesLoaded + " scenes loaded.");
132         ...
134     }
135     ...
733 }
```

should be "Critical Section" but is not

Minimizing Input

- GCC version 2.95.2 on x86/Linux with certain optimizations crashed on a legitimate C program
  - Note: GCC crashes, not the program!

```c
double mult( double z[], int n )
{
    int i;
    int j;
    for (j= 0; j< n; j++) {
        i= i+j+1;
        z[i]=z[i]*(z[0]+0);
    }
    return z[n];
}

int copy(double to[], double from[], int count)
{
    int n= (count+7)/8;
    switch (count%8) do {
        case 0: *to++ = *from++;
        case 7: *to++ = *from++;
        case 6: *to++ = *from++;
        case 5: *to++ = *from++;
        case 4: *to++ = *from++;
        case 3: *to++ = *from++;
        case 2: *to++ = *from++;
        case 1: *to++ = *from++;
    } while (--n > 0);
    return (int)mult(to,2);
}

int main( int argc, char *argv[] )
{
    double x[20], y[20];
    double *px= x;

    while (px < x + 20)
        *px++ = (px-x)*(20+1.0);

    return copy(y,x,20);
}
```

Figure 4: A program that crashes GCC-2.95.2.

Delta Debugging to the Rescue

- With 731 probes (< 60 seconds), minimized to:

```
t(double z[], int n) {
  int i, j;
  for (;;j++) { i=i+j+1; z[i]=z[i]*(z[0]+0); }
  return z[n];  }
```

- GCC has many options
  - Run DD again to find which are relevant

–ffloat-store
–fforce-mem
–fno-inline
–fkeep-static-consts
–fstrength-reduce
–fcse-skip-blocks
–fgcse
–fschedule-insns2
–fcaller-saves
–fmove-all-movables
–fstrict-aliasing
–fno-default-inline
–fforce-addr
–finline-functions
–fno-function-cse
–fthread-jumps
–frerun-cse-after-loop
–fexpensive-optimizations
–ffunction-sections
–funroll-loops
–freduce-all-givs
–fno-defer-pop
–fomit-frame-pointer
–fkeep-inline-functions
–ffast-math
–fcse-follow-jumps
–frerun-loop-opt
–fschedule-insns
–fdata-sections
–funroll-all-loops
–fno-peephole

https://www.cs.purdue.edu/homes/xyzhang/spring07/Papers/hdd.pdf

Go Try It Out: Eclipse Integration

Automated Debugging in Eclipse

We realized two Eclipse plug-ins that automatically determine why your program fails:

- in the input and
- in the program history.

These plug-ins integrate with JUnit tests: As soon as a test fails, they automatically determine the failure cause. You don't even have to press a button—just wait for the diagnosis.

DDinput: Failure-Inducing Input

Find out which part of the input causes your program to fail:

> *The program fails when the input contains <SELECT>.*

This plug-in applies Delta Debugging to program inputs, as described in Simplifying and Isolating Failure-Inducing Input.

Available for download.

DDchange: Failure-Inducing Changes

Find out which change causes your program to fail:

> *The change in Line 45 makes the program fail.*

This plug-in applies Delta Debugging to program changes, as described in Yesterday, my program worked. Today, it does not. Why?.

Available for download.

Questions?

- HW4 is due Wed
- .. and consider starting to work on HW5 (DD)!