Undo

—

Record, Replay, Rinse, & Repeat: Easily Rebuilding Programmatic State

Greg Law, co-founder & CTO

tl;dr

- Debugging dominates software development
  - Which means answering the question “what happened?”
- Record & replay is a new approach where the computer can just tell you
- Bugs can be fixed orders of magnitude more quickly

- Most software is not truly understood by anyone

In the beginning

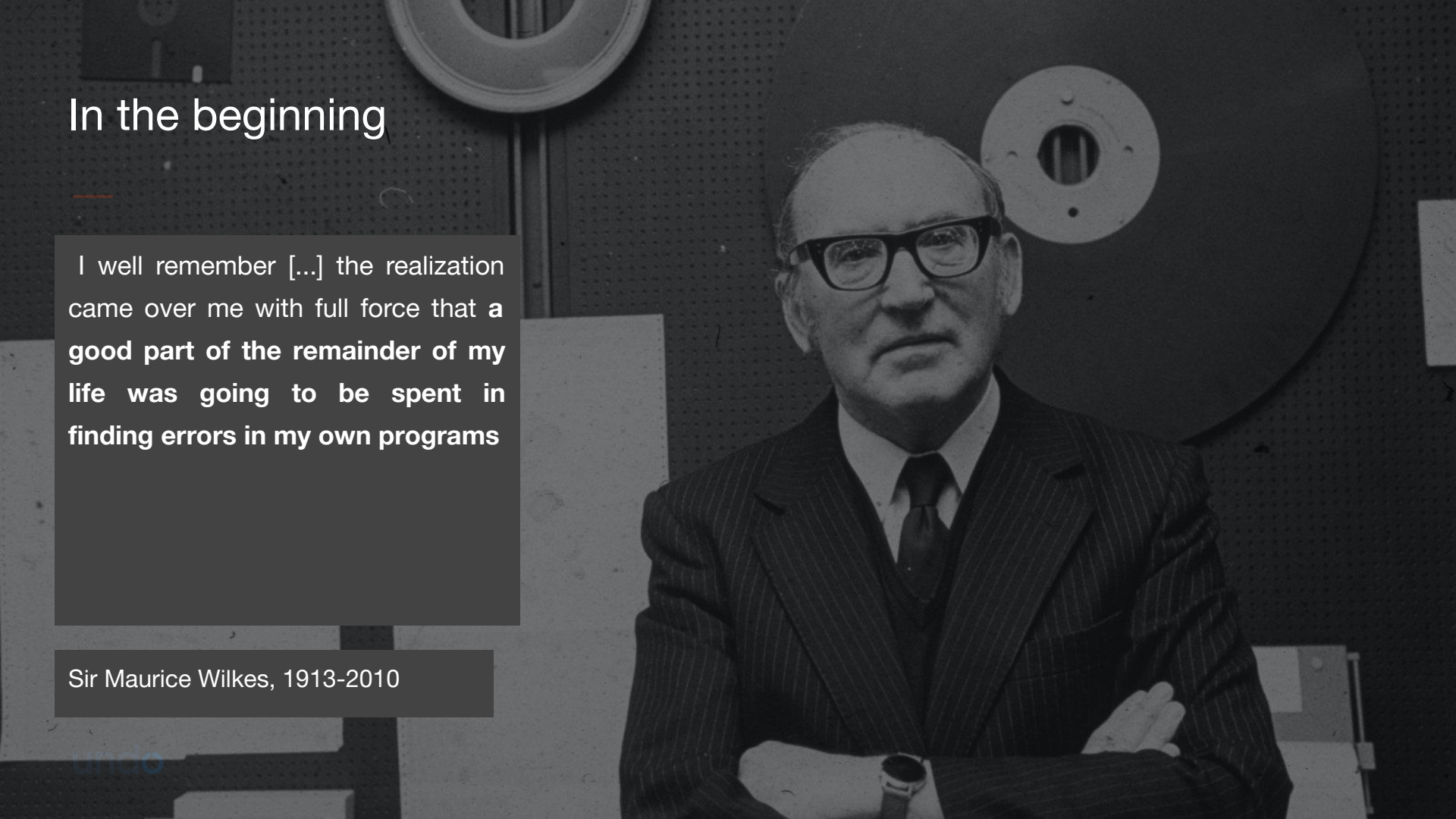
Sir Maurice Wilkes, 1913-2010

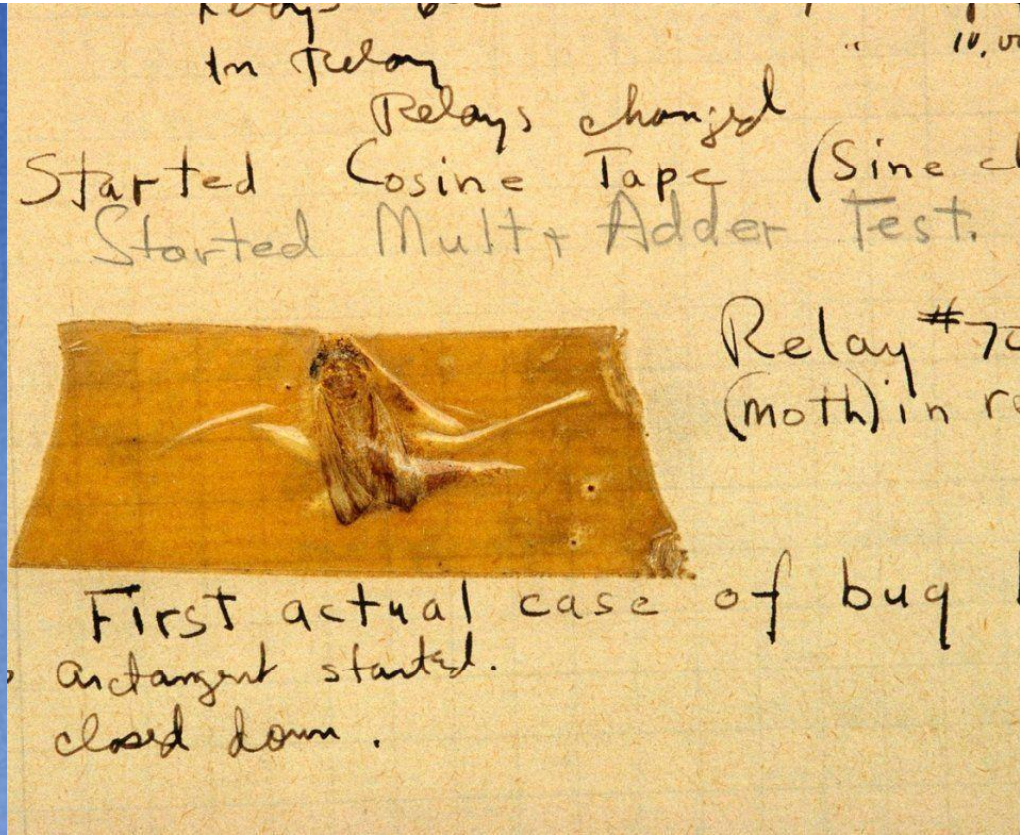

FRIDEN

In the beginning

I well remember [...] the realization came over me with full force that **a good part of the remainder of my life was going to be spent in finding errors in my own programs**

Sir Maurice Wilkes, 1913-2010




Relays changed
Started Cosine Tape
Started Mult + Adder Test.
Relay #
(moth) in
First actual case of bug
closed down.

undo

Computers are hard


cadence

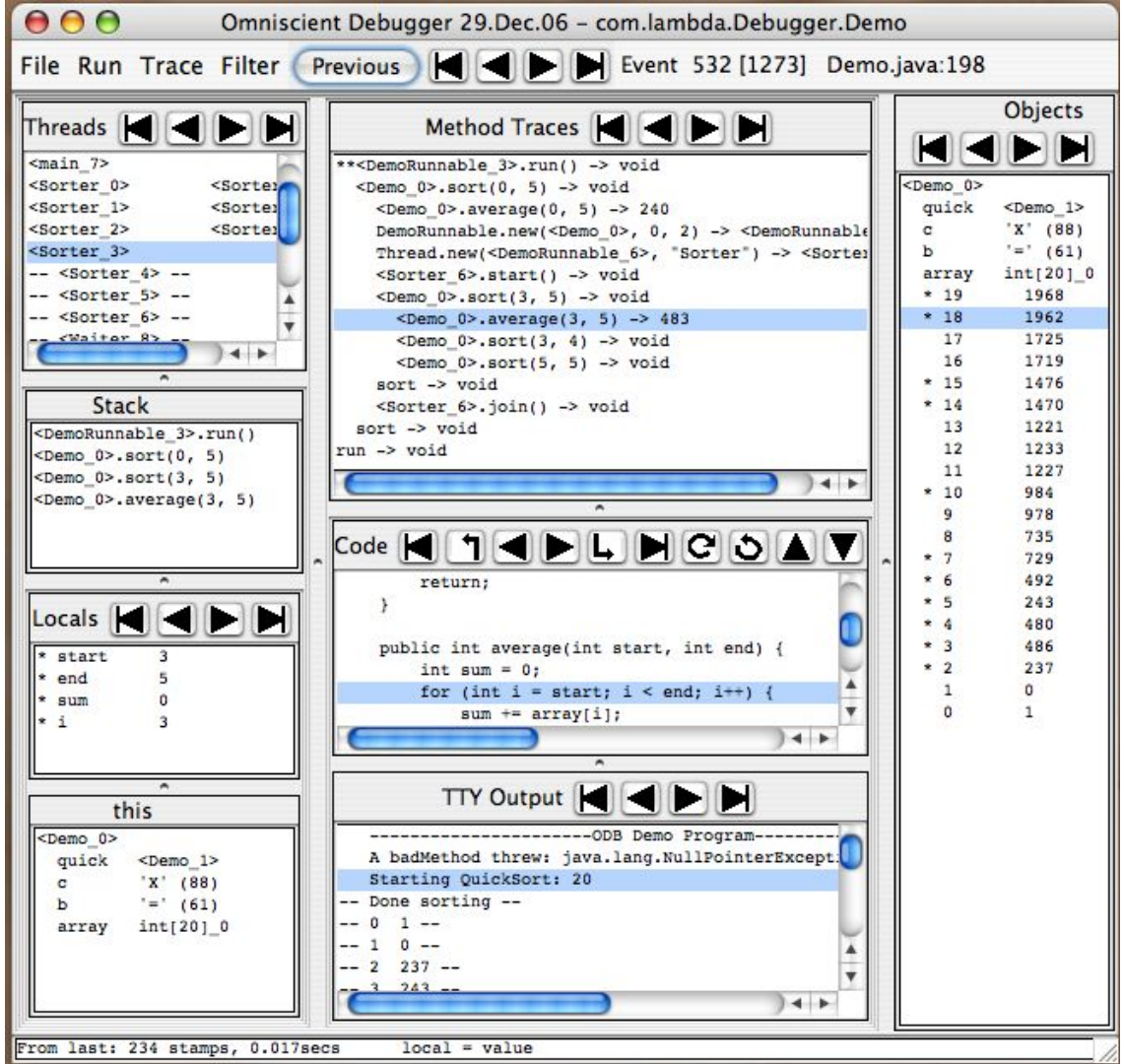
Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?

Brian Kernighan

undo

What happened?

What makes bugs really hard?


Omniscient Debugger 29.Dec.06 – com.lambda.Debugger.Demo
File Run Trace Filter Previous Event 532 [1273] Demo.java:198
Threads
<main_7>
<Sorter_0>
<Sorter_1>
<Sorter_2>
<Sorter_3>
-- <Sorter_4> --
-- <Sorter_5> --
-- <Sorter_6> --
Stack
<DemoRunnable_3>.run()
<Demo_0>.sort(0, 5)
<Demo_0>.sort(3, 5)
<Demo_0>.average(3, 5)
Locals
* start 3
* end 5
* sum 0
* i 3
this
<Demo_0>
quick <Demo_1>
c 'X' (88)
b '=' (61)
array int[20]_0
Method Traces
**<DemoRunnable_3>.run() -> void
<Demo_0>.sort(0, 5) -> void
<Demo_0>.average(0, 5) -> 240
DemoRunnable.new(<Demo_0>, 0, 2) -> <DemoRunnable
Thread.new(<DemoRunnable_6>, "Sorter") -> <Sorter
<Sorter_6>.start() -> void
<Demo_0>.sort(3, 5) -> void
<Demo_0>.average(3, 5) -> 483
<Demo_0>.sort(3, 4) -> void
<Demo_0>.sort(5, 5) -> void
sort -> void
<Sorter_6>.join() -> void
sort -> void
run -> void
Code
return;
}
public int average(int start, int end) {
int sum = 0;
for (int i = start; i < end; i++) {
sum += array[i];
TTY Output
----------ODB Demo Program----------
A badMethod threw: java.lang.NullPointerExcept
Starting QuickSort: 20
-- Done sorting --
-- 0 1 --
-- 1 0 --
-- 2 237 --
Objects
<Demo_0>
quick <Demo_1>
c 'X' (88)
b '=' (61)
array int[20]_0
* 19 1968
* 18 1962
17 1725
16 1719
* 15 1476
* 14 1470
13 1221
12 1233
11 1227
* 10 984
9 978
8 735
* 7 729
* 6 492
* 5 243
* 4 480
* 3 486
* 2 237
1 0
0 1
From last: 234 stamps, 0.017secs local = value

undo

What was the previous state?

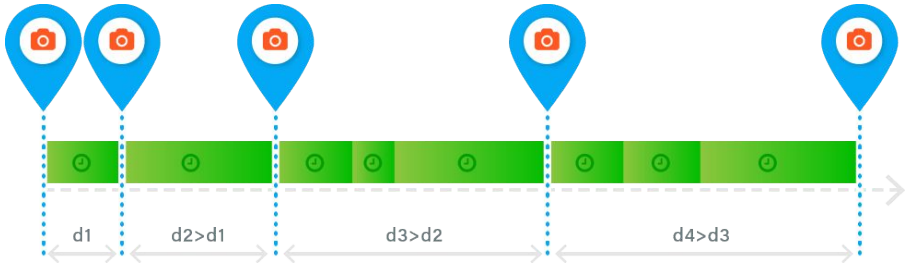
Two options:

1. Save it.
2. Recompute it.

$a = a + 1$ ✓

$a = b$ ✗

Snapshots



Maintain snapshots through history

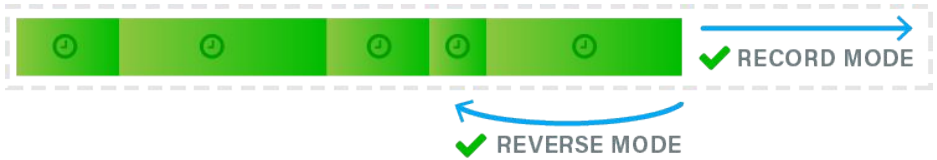
Resume from these - run forward as needed

Copy-on-Write for memory efficiency

Adjust spacing to anticipate user's needs

undo

Event log



Event Log captures non-deterministic state

Stored in memory

Efficient, diff-based representation

Recorded during debug (or Live Recording)

Replayed to reconstruct any point in history

Saved to create a recording file for later use

undo

Instrumentation

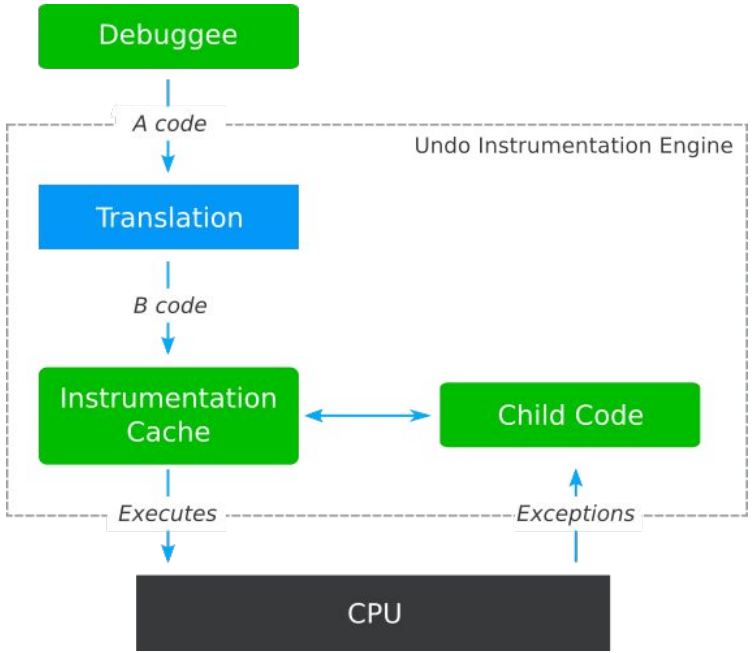
Undo Engine captures *all* non-determinism

Some machine instructions are non-deterministic

`rdtsc`, `cpuid`, `syscall`, etc

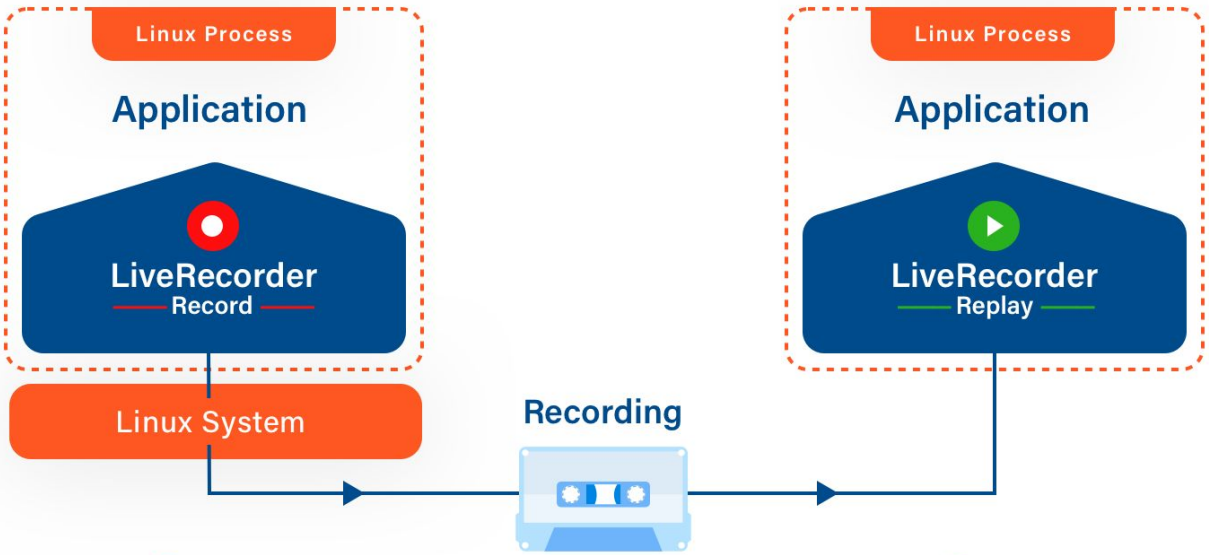
Needs to capture all this and provide precise control over execution in general

Solution: Runtime instrumentation



undo

In-process Virtualization



Record

- Captures all state changes in a running process
- 100% reproduction of execution history

Replay

- Replay & analyze the recording
- Detect root cause of bug
- Reverse debug and resolve

Multiple implementations

For Linux:

- Undo LiveRecorder (C++, Go, Java)
- rr (C++, Go)
- gdb process record

For Windows:

- Microsoft's Time-Travel Debugger (C++, C#, Chakracore JS)
- RevDebug (C#, Java)

Works well in conjunction with live logging & tracing

Logging & tracing give a high-level ‘story’ of a program’s execution

Use it to know where to go in a recording

Apply logging to a recording

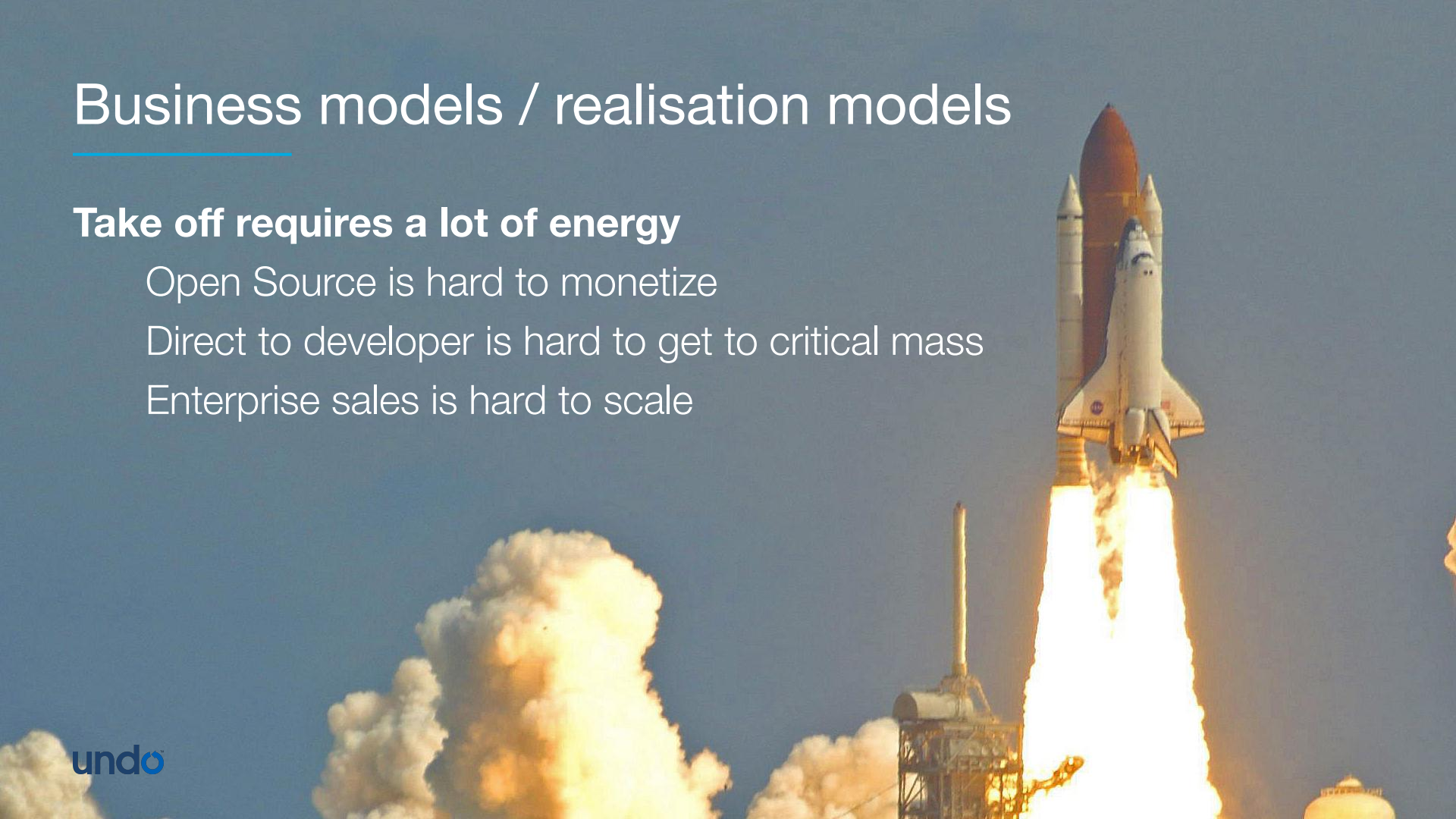
80/20 Rule

~~80/20 Rule~~

Business models / realisation models

Take off requires a lot of energy

- Open Source is hard to monetize
- Direct to developer is hard to get to critical mass
- Enterprise sales is hard to scale



undo

1. Computers are hard & debugging is under-served
2. Record/replay is awesome
3. 80/20 rule does not always apply