Increasing software development productivity with reversible debugging

Undo™

http://undo.io

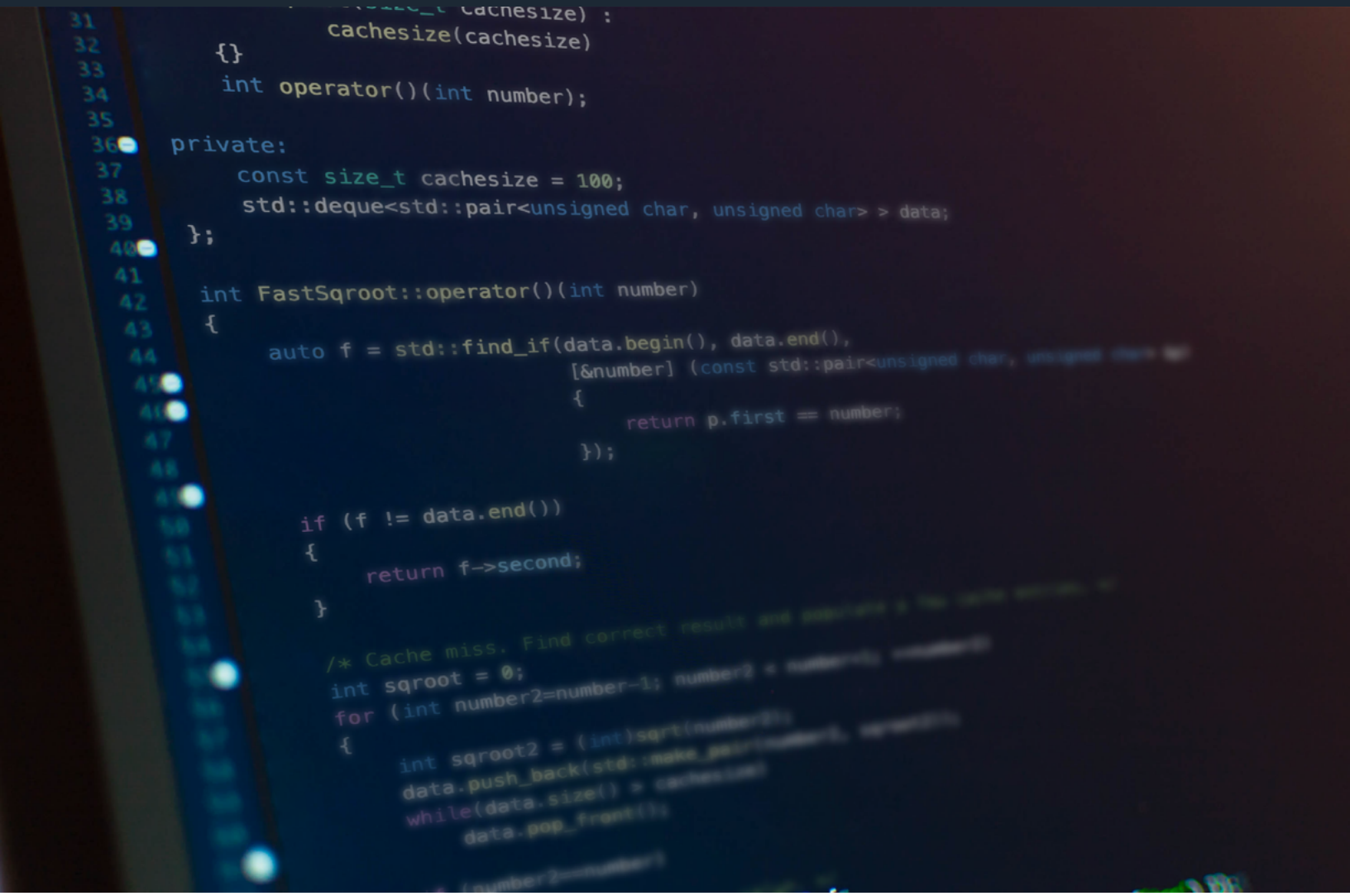


Table of contents

Introduction

In a world increasingly run by software, failures caused by bugs have never been more visible or high profile. Finding and fixing bugs faster, in a more predictable and productive way, is consequently vital for developers and managers.

Failure to find and fix bugs quickly has a financial, personal and reputational cost to an organisation. Products ship late (or not at all), developer time is spent on fixing bugs rather than coding, and customer fallout from a failure to fix bugs quickly once the product is already deployed can damage a company's reputation with its customers, translating into lower customer satisfaction and share price.

As code becomes more complex, finding and fixing bugs is correspondingly harder. Particularly difficult to locate are bugs that strike intermittently and only under certain conditions. Replicating the exact scenario that leads to an intermittent bug appearing is extremely time-consuming, and often impossible.

This white paper explains how traditional approaches to debugging are struggling to cope with the scale and complexity of today's software and looks at the technique of reversible debugging, how it works, and where it brings benefits.

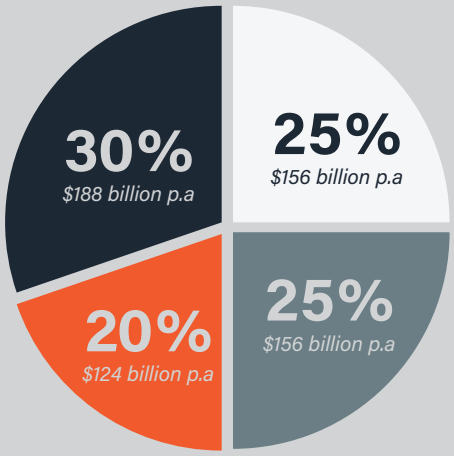
1 The debugging challenge

As software gains in complexity and the world becomes increasingly dependent on ever-more complex software, debugging is moving from being an inconvenience to a major problem for software companies, for both commercial and technical reasons. Delays in shipping code caused by bugs push back product release dates and directly impact company productivity. While tools exist to help developers prevent bugs when writing code, there has been little innovation in development tools that will help you locate and fix bugs once they have been found.

Applications are increasingly complex, multi-threaded, larger, and have a greater number of developers working on them, which makes tracking down bugs correspondingly more difficult and unpredictable. Multi-threaded programs lengthen the time elapsed between the root cause of the bug and its detection as well as making bugs less deterministic and difficult to reproduce.

As software increases in complexity, debugging is taking up more developer time and becoming vital for brand protection. A 2013 study from the Judge Business School of the University of Cambridge, UK[1], found that the global cost of debugging software has risen to $312 billion annually, half of which ($156 billion) is spent on wages.

Impact of debugging on time spent developing code and its cost in terms of wages per annum



- Fixing bugs
- Making code work
- Designing work
- Writing code

→ Global software developer wages = USD $624 billion p.a

→ Debugging (fixing bugs and making code work) takes up 50% of development time.

→ **Debugging cost the global software industry $312 billion p.a**

Source: Evans Data Corporation (2012), Payscale (2012), RTI (2002), CVP surveys (2012)

The study identified that developers spend 50% of their development time fixing bugs or making code work, rather than designing or writing new code. The vast majority of debugging time is spent locating the bug – once it has been found, correcting it is normally relatively simple.

> "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."
>
> — **Brian Kernighan** / *Co-author of the first book on C*

[1] Judge Business School (2013) - Research by Cambridge MBAs finds global cost of debugging to be $312 billion.

2 Traditional debugging tools and techniques

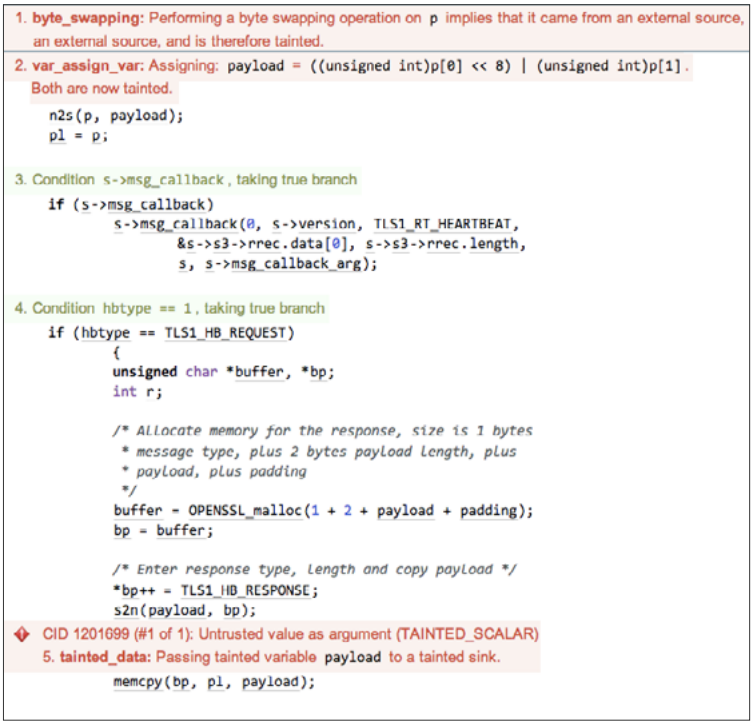
So how can developers make themselves smarter? There are a range of options and approaches available to help them. These can be classified into three groups: programmatic techniques, special-case diagnosis analysis tools, and general-purpose debuggers.

Programmatic techniques

Essentially developers modify or write their program in a way that helps them find bugs. Techniques include print statements, assertions and the use of test suites.

Special case diagnosis/analysis tools

These automated tools (such as Coverity, Purify and Valgrind) detect the most common bugs (memory access violations, touching unallocated memory, or potential deadlock conditions, for example). While they help with particular types of errors, they are not comprehensive to give total coverage. If your bug doesn't fit neatly into one of these categories, such tools don't offer any help. And even instances of very common bugs can elude these tools. The recent Heartbleed bug was a very common form of bug (buffer over-read), yet [every commonly-used detection tool failed to spot it](#).



General-purposes debuggers

When bugs cannot be found with special case diagnosis tools, many programmers turn to general-purposes debuggers, such as GDB. These let the programmer step forwards, inch by inch, through their code and set watchpoints as they go.

However debugging involves thinking backwards, as Brian Kernighan and Rob Pike point out in their book The Practice of Programming:

> "Reason back from the state of the crashed program to determine what could have caused this. Debugging involves backwards reasoning, like solving murder mysteries. Something impossible occurred, and the only solid information is that it really did occur. So we must think backwards from the result to discover the reasons."

Therefore, to be really useful, a debugger needs to help the programmer walk through the program's execution backwards. Consequently, developers need a different approach, and this is where reversible debugging comes in.

3 Introducing reversible debugging

Reversible debuggers enable developers to record all program activities (every memory access, every computation, and every call to the operating system) and then rewind and replay to inspect the program state. This colossal amount of data is presented via a powerful metaphor: the ability to travel backwards in time (and forwards again) to inspect the program state. Essentially they enable developers to solve the murder mystery by letting them rewind their code to walk backwards, as well as forwards, through the program. Take an example use-case of tracking down some corrupted memory. With a reversible debugger, a developer can simply put a watchpoint on the variable that contains bad data, and run backwards to go straight to the line of code that most recently modified it. Bugs that would take a very long time to track down can be found in minutes.

There are many benefits of using reversible debugging:

General productivity

Making reversible debugging part of the development and debugging process improves overall productivity. Common, but difficult to identify bugs can be found more quickly, freeing up developer time. Reversible debuggers that are fully compatible with the open source debugger GDB can be easily integrated into development environments, without the need for extensive training. Reversible debugging can also help developers become familiar with code they did not write, but which they now have to work with.

Intermittent bugs

Sporadic bugs, that strike seemingly at random, are incredibly difficult to find as well as being potentially the most damaging to company reputation. They can easily slip through the net of normal debugging routines, and only surface when code is close to shipping – or worse, has already shipped. By running a reversible debugger until the intermittent bug strikes, developers can then step backwards from the point of failure line by line until the bug itself is found. If necessary multiple instances of the debugger can be run on different servers to increase the chance of the bug manifesting itself.

At customer premises

For many software vendors, their tools are being used on customer sites. If the program crashes, vendors must learn about the circumstances in order to reproduce the issue in-house before investigating it. Unfortunately, this is often impossible, meaning software vendors have no choice but to send an engineer to the customer site to investigate the failure. Running a reversible debugger on-site, on the machine demonstrating the issue, means that, the next time it occurs, the engineer can step back to see what went wrong.

4 The business case

The recent University of Cambridge research analysed the financial cost of debugging, and how it could be reduced. The math is simple. The global cost of software development is $1.25 trillion. It found that debugging represents a quarter of the overall budget, representing $156 billion in wages, with overhead costs doubling this to $312 billion.

Reversible debugging can deliver significant savings. Mentor Graphics has reduced debugging time by 66% (two thirds) after implementing Undo Software's reversible debugger for Linux, UndoDB.

Take an average software developer earning $90,000[2]. Currently they spend a quarter of their time debugging, costing $22,500 in wages. Reducing that by two thirds, creates a saving of $15,000 and increasing available developer time.

Spend more time developing and less time debugging when you use a reversible debugger

Of course, this solely focuses on part of the financial impact of finding and fixing bugs. It ignores the costs of:

- Delaying product launches.
- Running recall programs after software has been released.
- The reputational damage to a company when things go wrong.
- Lost customers if your tool is responsible for delays or issues.
- The personal cost to developers and managers, in terms of stress and sleepless nights, as they struggle to find and fix bugs.

[1] US Bureau of Labor Statistics

There are several main scenarios where reversible debugging can help:

Long run times

Sometimes tracking down a bug can itself be an $O(n^2)$ iteration: running the debugger 5 minutes until the bug manifests itself, setting a breakpoint earlier in the code and running again for 4 minutes, setting an earlier breakpoint and rerunning, etc. With reversible debugging, that time-consuming run-restart cycle can be reduced to an O(n) process. Run until you hit the bug, then step backwards to see what led to the problem. Did you miss it? Step forwards a little, and backwards again.

Frequently-called functions

Take the example of a function which is called many times, but fails after about a thousand calls with a fault such as SIGSEGV or SIGFPE. Setting a breakpoint in the function doesn't work well because it stops at the first occurrence, when you really want it to stop at the last occurrence – but that involves predicting the future! With a reversible debugger it's possible to run to the end and only then set a breakpoint. When running in reverse, the first breakpoint you hit is the last time that code was executed.

Dynamic code

Some applications generate specialised code at runtime. Debugging such code is hard because source code analysis tools are obviously unable to help, there is none of the normal debug information to locate functions, and the code could be generated at different addresses on different runs. A reversible debugger allows the developer to examine a single run in detail without the headaches associated with re-running.

Intermittent bug

An intermittent bug might only strike in 1 in 300 runs. If the developer investigating the bug discovers the need to set a different breakpoint or add another logging command to help understand the problem, it will take a lot of runs before the bug is hit again, so progress will be very slow. A reversible debugger can't help to make the bug appear sooner, but once it does appear the entire history of the run can be examined.

Dynamically generated code, stack corruption

Often an issue, a bug creates code that corrupts the stack. GDB cannot cope, and the coredump provides no information. Using reversible debugging, the developer can rewind to see the stack corrupting and fix the issue in minutes.

Memory leaks

Obscure memory leaks can cause software to run slower over time and potentially even crash. Memory leaks are hard to debug using conventional tools because there is a large gap in time between the allocation of a buffer and the point where it should be freed. It's also not clear where the fault is – the problem is likely to be an absence of code where it should be. Worse, if the program is re-run it may be a different buffer that leaks.

A reversible debugger gives the developer the chance to work on a single example failure, moving freely backwards and forwards through the history to identify where the missing code should be.

Real-time, network protocols

Software can fail when it receives data in unexpected formats. But it may not be possible to step through the code using a debugger if it is communicating with an external program or device which has real-time constraints – the other device may simply give up. With a reversible debugger there is no need to stop during the initial "recording" phase, because it is always possible to rewind later.

Race conditions

Take a bug where some code accesses shared data but claims the wrong lock. This shows up as a threading bug where two threads are accessing data A but one of them has locked data B by mistake, so there is a race condition between the threads. Using a conventional debugger the bug will show up as a corruption of data A, but the cause won't be obvious. Typically the response is to run again with watchpoints set, but this can result in a lot of false positives unless a complex condition is defined to filter out the OK accesses, and having set all that up there's a strong chance that the bug won't manifest next time. Reversible debugging makes it faster: by starting at the end where the corruption is detected, setting a watchpoint and running backwards, the source of the corruption can be found much sooner.

Data structure corruption

Corruption of a linked list leads to a crash, but it is difficult to see when the corruption occurs. Rather than having to continually re-run the program, reversible debugging allows the developer to go back in time before the list was corrupted and use a binary search to quickly find out when the list got corrupted. This brings debugging time down from over an hour to less than 10 minutes.

These are examples of where reversible debugging aids developers and are by no means exhaustive.

5 Introducing UndoDB

UndoDB is an interactive reversible debugger for C/C++ on Linux and Android that works on any user-mode compiled code, on x86 and ARM. It takes the guesswork out of debugging by allowing developers to step or run their program backwards as well as forwards in time. It incorporates the full functionality expected of modern debuggers (such as scripting, conditional breakpoints and watchpoints, full inspection of globals and locals) and also allows these features to be used with the program running in reverse. Bugs can be fixed in minutes, not weeks.

UndoDB is a drop-in replacement for GDB and therefore seamlessly integrates into a developer's work flow. UndoDB can be used at the command line, from any popular IDE (CLion, Eclipse, Emacs etc.), allowing developers to choose their preferred work environment.

To explain UndoDB's advantage, we need to introduce the concept of determinism. A deterministic process is one which always produces the same output when fed with the same starting state. The insight which drives UndoDB is that computers are mostly deterministic (which explains why they are not very good at generating random numbers). If a program behaves deterministically, there is no need to record its intermediate states, because they can be reconstructed at any time simply by running the program from the beginning.

Real programs are not completely deterministic, and to correctly replay a program all the sources of non-determinism must be captured. Sources of non-determinism include:

- Inputs from outside the program, e.g. from user interaction, files, network sockets, real-time clocks etc. Usually these interactions take the form of system calls or accesses to memory-mapped files.
- Signals.
- Scheduling variation - in a multithreaded program where more than one thread is unblocked, the OS can decide to schedule the threads in any order, or even simultaneously on a multicore machine.
- The CPU itself, which may have certain instructions whose effects are not predictable from the program state, e.g. the x86 CPUID and RTDSC instructions.

The program being debugged with UndoDB is instrumented on-the-fly to identify all sources of non-determinism. Instrumentation also provides a timebase, so that any point in a program's run can be identified via a count of "simulated nanoseconds" (which correspond very approximately to real-time nanoseconds). The "event log" captures all the information needed to reconstruct the effects of non-determinism. For example, if the program executes a read() system call, UndoDB will capture the new buffer contents into its event log. If the same section of code is later replayed, the read() is not executed again - instead its effect is simulated by copying the saved buffer from the event log.

6 Conclusion

Today, software is central to every organisation. Finding and fixing bugs has never been more important – meaning that application software debugging tools are no longer a "nice to have", but are a business and development necessity.

Reversible debugging provides a viable, cost-effective way of locating bugs as developers can now record, rewind and replay their code. This makes it simpler to quickly find and fix customer-critical bugs, deliver to ever-shortening deadlines and boosts overall productivity.

By reducing debugging time by two thirds companies can quickly see top line savings, freeing up developers to code more productively, thereby increasing efficiency and safeguarding corporate reputation. With the pressures on software development growing, now is the time to investigate reversible debugging and the benefits it brings.

About us

Undo's products are used by thousands of developers to solve complex, real-world problems for leading technology companies, from embedded to enterprise and High Performance Computing to banking. Its unique record, rewind and replay technology enables Linux and Android developers to see exactly what their program did at every step in its execution. Developers can now rapidly respond to failures in production and test environments, increase their development productivity by at least 25% and dramatically improve software quality.

Undo is a privately held company headquartered in Cambridge, UK. It was one of the winners of the 2016 London Accenture FinTech Innovation Lab and its technology has won numerous awards including Best Software Product at the ARM Innovation Challenge and Gartner's Cool Vendor in Application Development. For more information, see http://undo.io or follow us on Twitter at twitter.com/@undosoft.

Undo

9 Signet Court
Swann's Road
Cambridge, UK
CB5 8LA

http://undo.io
linkedin.com/undoltd
twitter.com/undosoft