Undo™

Bugs that took weeks now take minutes

Increasing software development productivity with reversible debugging

A white paper by Undo Software

Table of contents

1. Management summary

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 organization. Products ship late (or not at all), developer time is spent on fixing bugs rather than coding and customer fallout from 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 retention.

As code becomes more complex, finding and fixing bugs is correspondingly harder. Bugs that strike intermittently and only under certain conditions are particularly difficult to locate. 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 struggle 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.

1.1 The growing importance of debugging

As software now underpins more and more systems, debugging is moving from being an inconvenience to a major problem for both commercial and technical reasons. Delays in shipping code due to issues caused by bugs has a knock-on effect on product release dates. Additionally, time spent searching for bugs, rather than programming, directly impacts company productivity.

Applications are increasingly complex, multi-threaded, larger, and have a greater number of developers working on them, making tracking down bugs correspondingly more difficult and unpredictable. Multi-threaded programs make programs less deterministic and so more difficult to debug.

1.2 The rising cost of debugging

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

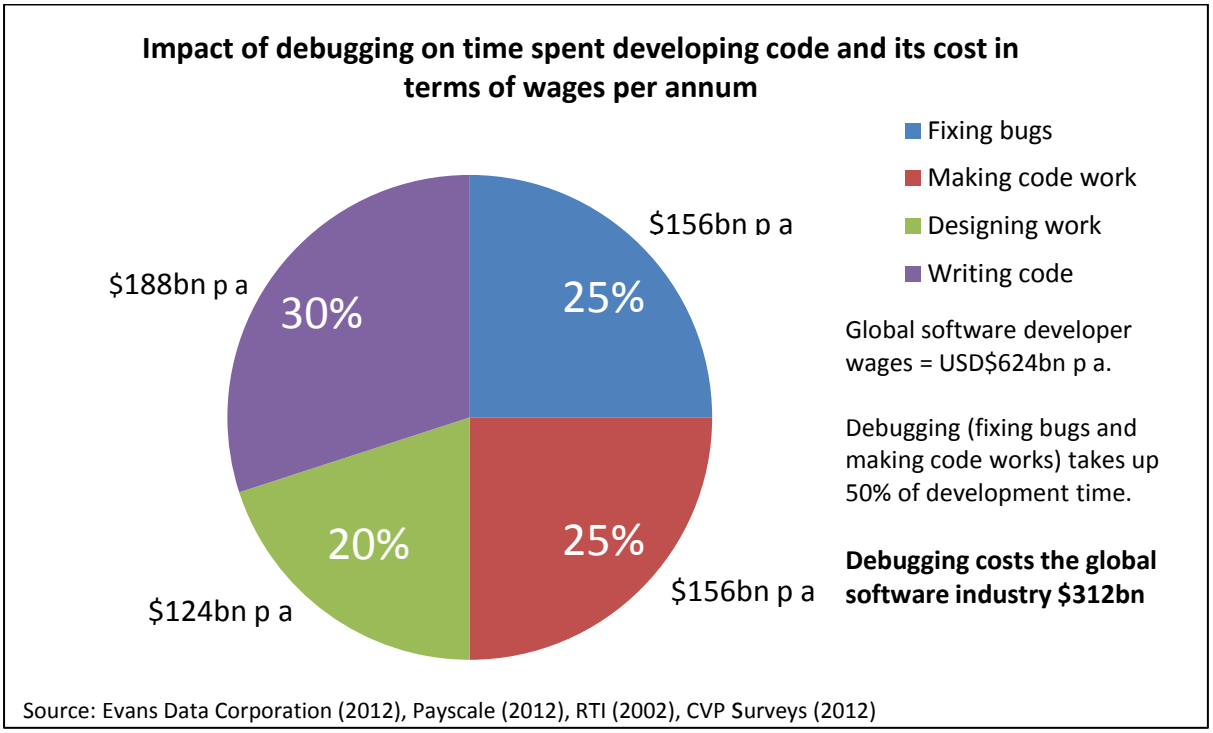

Figure 1: Impact of debugging on time spent developing code and its cost in terms of wages per annum

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.

As Brian Kernighan, co-inventor of the C programming language, wrote,

"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."

2. The options for debugging

2.1 Current options

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.

2.1.i Programmatic techniques

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.

2.1.ii 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 do not 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[1].

2.1.iii General-purposes debuggers

When bugs cannot be found with special-case diagnosis tools, many developers turn to general-purposes debuggers, such as GDB. These let the developer 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 developer walk through the program's execution backwards. Consequently, developers need a different approach, and this is where reversible debugging comes in.

2.1.iv 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 backward in time (and forward again) and inspect the program state. They enable developers to solve the murder mystery by letting them see exactly what happened and how the code got here. 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 backward to go straight to the line of code that most recently modified it. Bugs that would have taken a very long time to track down can now be found in minutes.

[1] Source: Coverity blog, April 2014

2.2 Where reversible debugging adds value

2.2.i Immediate and sustained productivity gains

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. Furthermore, reversible debugging adds a relatively simple metaphor to the existing debugger experience, meaning that developers become more productive very quickly, with minimal training required. This is even more true when reversible functionality is added to the debugger that the developer is already using.

Reversible debuggers are not only about debugging. They are tools to allow developers to gain detailed insight into what a program is doing, which can significantly improve the productivity of a developer trying to understand code that they did not write.

2.2.ii Ability to react quickly to customer-reported issues

When problems are encountered, customers demand that fixes are delivered quickly. Reversible debugging can dramatically reduce the time taken to find and solve a bug, or even make it possible to fix particularly difficult bugs that would otherwise be impossible to resolve. This allows software vendors to provide much better customer service and so improve reputation and customer retention.

2.2.iii Debugging of live production systems

Debugging live production systems can be particularly challenging, especially if the circumstances leading to the bug are not yet understood, meaning that it can only be reproduced in production (e.g. at the customer premises). Traditional debuggers can cause failures in the wider system when breakpoints are hit and execution stops. A reversible debugger can be deployed on a live system and the program run uninterrupted, and then the recording wound back allowing the developer to investigate the failure while the live system continues to run.
This is further enhanced by reversible debuggers that allow a recording to be saved to a file for later investigation.

2.3 The business case behind reversible debugging

The recent University of Cambridge research analyzed 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 annually. It found that debugging represents a quarter of the overall budget, a cost of $312 billion per year.

Reversible debugging can deliver significant savings. Mentor Graphics has reduced debugging time by 67% (two thirds) after implementing Undo Software's reversible debugger for Linux, UndoDB.

Take an average software developer earning $90,000 (based on 2012 US labor statistics)[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.

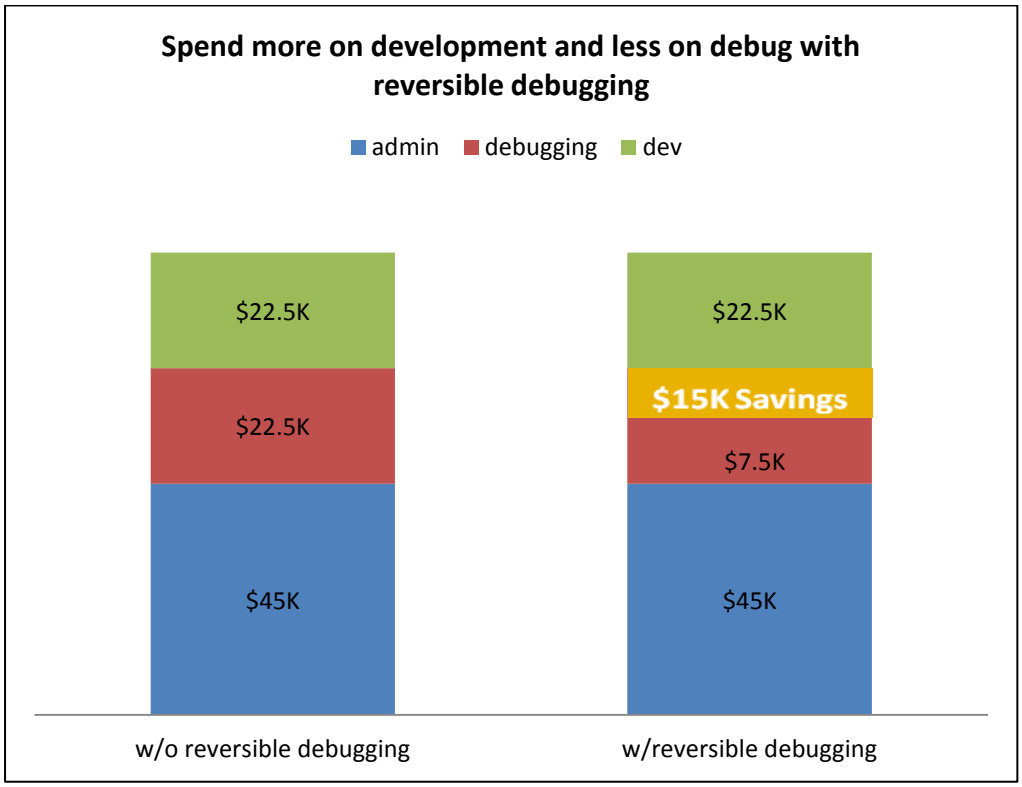

Figure 2: Spend more on development and less on debug with reversible debugging

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

- Delaying product launches.
- The reputational damage to a company when things go wrong.
- Lost customers when failures in your software adversely impact their business.
- The personal cost to developers and managers, in terms of stress and sleepless nights, as they struggle to find and fix bugs.

[2] Source: Bureau of Labor Statistics, U.S. Department of Labor, Occupational Outlook Handbook, 2014-15 Edition

3. Reversible debugging – how it helps in nine common scenarios

3.1 Common debug scenarios

3.1.i 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 step too far? Step forwards a little, and backwards again.

3.1.ii 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.

3.1.iii 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 does not increase the chances of reproducing the intermittent bug, but it does mean that once the bug has been reproduced the developer has at their fingertips all of the information from the program leading up to the bug.

3.1.iv Stack corruption

If a bug results in a corrupted stack a coredump or regular forwards-only debugger can do little to tell the developer where the code was immediately before the stack was corrupted. Using reversible debugging, the developer can simply rewind to see the stack corruption and fix the issue in minutes.

3.1.v Memory leaks

Obscure memory leaks can cause software to run more slowly 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.

3.1.vi 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.

3.1.vii Race conditions

Multi-threaded code is notoriously difficult to write and even harder to debug (see the Kernighan quote above). This is largely due to the non-deterministic and intermittent nature of most race conditions. A reversible debugger allows developers to "home in" on the part of the program's execution history where the bug manifests itself, and to step through the potential complex interactions of race conditions.

3.1.viii Data structure corruption

It is often very difficult to know when data corruption occurs. Using a conventional debugger the response is usually 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 this up there's a strong chance that the bug won't manifest itself next time. Reversible debugging makes things much easier: 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.

3.1.ix Dynamic code

Some applications generate specialized 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.

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

4. Reversible debugging and UndoDB

4.1 Introduction

Undo Software's reversible debugger, UndoDB, allows Linux software developers to record their program's execution, and then "wind the tape" back and forth in real-time in order to get a clear picture of their program's execution. This takes much of the guesswork and trial-and-error out of debugging. Bugs that would have taken weeks to solve can now be solved in minutes. 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 Eclipse, DDD or Emacs, allowing developers to choose their preferred work environment.

UndoDB is based on the ability to efficiently record program execution so that any point in the program's history can be precisely reconstructed. By contrast, other tools either record only certain discrete points (e.g. function entry/exit), or attempt to record the state change on every instruction - with consequent poor performance.

4.2 UndoDB's advantage

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 accessing memory shared with files, devices or other processes.
- Signals.
- Scheduling variation - in a multi-threaded 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.

4.3 How UndoDB works

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.

5. Conclusion

Today, software is central to every organization. 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. The ability to fix bugs quickly is also essential to providing good customer support. Reversible debugging can dramatically reduce the time taken to fix some of the most difficult bugs, resulting in significantly improved customer satisfaction and retention.

With the pressures on software development growing, now is the time to investigate reversible debugging and the benefits it brings.

Undo Software is the leading commercial supplier of Linux and Android reversible debugging tools that enable software developers to record, rewind and replay their C/C++ code to respond quickly to customer critical bugs, increase their productivity and meet their development deadlines. Used by over 1,000 developers at customers that include Cadence Design Systems and Mentor Graphics to solve complex, real-world problems, UndoDB is proven to reduce debugging time from weeks to minutes, while seamlessly integrating into existing development environments. Undo Software is a privately held company headquartered in Cambridge, UK, and is a Gartner "Cool Vendor in Application Development" for 2014. For more information, see: http://undo-software.com/ or follow us on Twitter at www.twitter.com/@undosoft.

UndoDB is sold directly from Undo Software, and can also be found in Rogue Wave's TotalView debugging suite and in ARM's DS-5 development studio from version 5.16.

Undo™

Bugs that took weeks now take minutes

Undo Software

9 Signet Court
Swann Road
Cambridge
CB5 8LA
United Kingdom

sales@undo-software.com

+44 (0)1223 300264