Time Travel Debugging

Understand complex code and fix bugs faster

Overview

Over the last decade, software development has become mind-blowingly complex – as have the challenges related to finding and fixing bugs. Two problems make debugging increasingly difficult:

1. Complexity hinders developers' ability to understand what the code is doing, and efficiently fix bugs. Complex architectures and large teams mean that developers often have to troubleshoot bugs that have little or nothing to do with the code they created.

2. 91% of developers admit to having software defects which are unresolved because they cannot reproduce them (see: an analyst's study on software reliability).

Traditional methods of debugging and analyzing code are no longer sufficient for the challenges of understanding complex codebases and debugging modern applications.

This technical paper explores what every software engineer working on complex codebases (e.g. multithreaded or multiprocess programs) should know about time travel debugging.

It goes on to outline how upgrading to time travel debugging (TTD) can save you time debugging – enabling you to get changes into the pipeline faster, complete your code deliverables on time, and resolve customer-reported defects in hours, not weeks.

Debugging is a costly productivity killer

As the world becomes increasingly dependent on software, finding and fixing software failures in complex systems has moved from being an inconvenience to a major problem. Delays in shipping code because of a growing backlog of bugs push back product releases and negatively impact engineering productivity and innovation.

25–50%

developer time spent on debugging rather than innovating

While tools exist to prevent bugs when coding, there has been little innovation in tools that help with **debugging** once these bugs surface.

In addition to the productivity cost of excessive lengths of time spent debugging, software errors that cause failures can also cost businesses in other ways.

Visible Costs	Hidden Costs
Product delivery and innovation delays	Employee frustration
Time spent debugging	Backlog of unresolved failing tests
Application downtime	Technical debt
Customer complaints	Customer experience damage
SLA violations	Lost market opportunities
Reputational damage	

So how are developers currently going about the process of debugging?

Traditional debugging options

There is a range of traditional options and approaches currently available to developers to help diagnose errors in a codebase.

Method	Pros	Cons
Programmatic techniques *Assertions and the use of test suites*	Developers modify or write their program in a way that helps them to better observe their code Improves overall code quality and understandability	Requires codebase knowledge Limited for finding certain types of bugs
Logging *Log of program execution*	Easy to use – every developer knows how to insert logging statements into their code (e.g. printf in C or log4j in Java) Easy to navigate and understand program execution Provides clues about what went wrong	Limited in use – not good for thread-related issues Snapshot only – can't tell you the value of variables over time Difficult to determine exact root cause
APM	Valuable for detecting that a problem has occurred	Precise root cause still needs to be investigated and tracked down
Special case analysis tools *Automated tools (e.g. as Coverity, SonarQube, Valgrind, AddressSanitizer, ThreadSanitizer)*	Detect common bugs (e.g. memory access violations, touching unallocated memory, or potential deadlock conditions)	They can only detect those errors that they are designed to detect False positives create "noise" in reports, which can drown out the real errors
Debuggers *Traditional cyclic debugging (e.g. GDB, native IDE debuggers)*	Step forward, inch by inch, to examine code Stop execution at a given point to investigate where it goes and what the values are Can be used as command line or in IDEs	Slow, iterative process May require effort and additional tooling to enable developers to reproduce an issue Not suited to error resolution in live production environments Can't provide any historical information ("what happened?")

Why it's time to transform the way we debug

Traditional debuggers can be painful and inefficient

The most inefficient part of traditional debugging stems from only being able to work forward toward the point of the crash or error. This way of working is slow and results in the need to repeatedly restart the application and work through a complex series of steps to get back to the area of the point of failure.

Credit: monkeyuser.com

The time cost of debugging this way can be high. A lot of reproducing the issue over and over again, a lot of stepping-in and lots of "*Oops, I've stepped over too far. Now I have to restart.*" Working this way is time consuming and inefficient; just at a point when a developer is under pressure to simply fix a bug as quickly as possible.

Traditional debuggers are out of step with modern application complexity

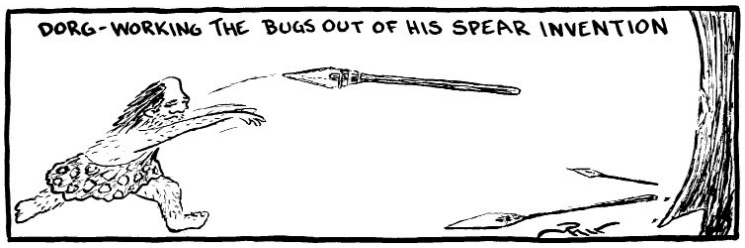
Traditional tools and methods for debugging and analyzing code are no longer adequate for the challenges of understanding complex codebases, and debugging modern applications. Logging, printf, and debuggers were conceived in the 70s and 80s, in an era when programs were comparatively small, with simple processors (no multicore), simple compilers, and few structured forms of testing. These traditional debugging techniques have remained fundamentally unchanged since then.

Despite seeing huge changes over the last few decades in methods of program development, traditional debugging has not evolved in parallel. Debugging code through iterative, line-by-line guesswork has not scaled well with increased application complexity, which can now involve multiple threads on multiple processors, terabytes of data, and billions of instructions from multiple sources.

Debugging forward doesn't make sense

It makes little sense for developers to dogmatically stick to debugging forward, by manually going through the entire stop-rerun-navigate cycle when, in fact, they are trying to simulate stepping backward in their code, to see what went wrong.

> It makes little sense for developers to dogmatically stick to debugging forward when, in fact, they are trying to simulate stepping backward in their code, to see what went wrong.



Credit: Cartoonist: Rank, Joseph

Traditional forward debugging may be how most developers have always done things when using debuggers; but it no longer fits the aim of delivering better software, faster. It's time to upgrade our method of debugging by half a century!

What if developers could simply time travel backward through their code to see exactly what went wrong, instead of having to try and reproduce errors before figuring out what actually happened, and why?

Time travel debugging is a game changer

Time travel debugging (TTD) is the ability to wind back the clock to any point in an application's execution and see exactly what it was doing. Integral to TTD is the ability to reverse debug through program execution history. This transformative debugging capability allows developers to observe and understand the precise conditions that led to a specific bug. By simply letting them rewind the programmatic execution path directly back to the root cause, TTD accelerates finding and fixing bugs. It is also a powerful way for developers to learn about and navigate an unfamiliar codebase.

> Time travel debugging arms developers with the data they need to understand the root cause of errors and to help them fix bugs fast.

Take an example use-case of tracking down some corrupted memory. With time travel debugging, a developer can put a watchpoint (aka data breakpoint) on the variable that contains bad data, and run backward to go straight to the line of code in the thread that most recently modified it.
This "direct to root cause" approach accelerates debugging by eliminating the need for trial and error and repeatedly restarting the program with different breakpoint locations. It reduces multiple iterations down to one loop.

Watch video to see how reverse watchpoints work (1 minute)

But time travel debugging is not just about stepping back a couple of hours in your program's execution. **When integrated into a test suite, it equips developers with the superpower of stepping back to 2 weeks ago (or 2 years ago!)** – when a process failed and was captured in a recording file (akin to video footage). Read more on how this works further down.

Types of bugs that time travel debugging can resolve quicker

Time travel debugging is really powerful for:

- Any bug where time passes between the bug occurring and the symptoms presenting themselves, i.e. assertion failures, segmentation faults, or simply bad results being produced.
- Any bug which occurs intermittently or sporadically – for example, a bug that occurs one time in a thousand, or occurs in a different way on each run of the program. Bugs tend to manifest in this way when the program's execution is non-deterministic due to multithreading and/or interaction with other processes and services.

Most race conditions fall under one or both of these cases; so do many memory or state corruption bugs and non-trivial memory leaks.

Specialized tools (e.g. ThreadSanitizer, Valgrind, etc.) can help with some of these issues, and you should strongly consider using them routinely if you're not already doing so. But you probably wouldn't be reading this paper if they had solved all of your problems.

The following are some common scenarios that TTD can help resolve more quickly and more efficiently than other debugging methods.

Race conditions
Take a bug where code in one thread 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, causing 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 at all 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. Time travel debugging makes it faster: by starting at the end where the corruption is detected, setting a watchpoint and running backward, the source of the corruption can be found much sooner.

Memory 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 rerun the program, time travel debugging allows developers to go back in time to before the list was corrupted and use a binary search to quickly find out exactly when the corruption occurred. This can bring debugging time down from hours to minutes. Read this routing engineer's story on how he fixed a memory corruption issue in under 10 minutes with time travel debugging.

Segmentation faults (aka segfault)
Core dump / segmentation faults are all too common. They can occur when a program is attempting to read or write to an illegal memory location.

Using old debugging methods, a starting point would be to use a debugger to view a backtrace of a core file. Doing this would let you know roughly where the program was when it crashed. This information would reveal where to start when trying to investigate the cause of the crash, using an iterative process of elimination. By contrast, time travel debugging enables developers to zero in on the root cause of the bug in a systematic way without needing to guess and restart the program several times.

Stack corruption
Buffer overruns and other kinds of defect can corrupt the stack. This decreases the amount of useful information that a backtrace or a conventional debugger is able to extract from a core dump. Using time travel debugging, developers can rewind to see the stack corruption 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 rerun, it may be a different buffer that leaks. A time travel debugger gives developers the chance to work on a single example failure, moving freely backward and forward through the execution history to identify where the missing code should be.

Long run times
Sometimes tracking down a bug can itself be an $O(n^2)$ iteration: running the debugger for 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 time travel debugging, that time-consuming run-restart cycle can be reduced to an O(n) process. Run until you hit the bug, then step backward to see what led to the problem. Did you miss it? Step forward a little, and backward again.

Frequently called functions
Take the example of a function which is called many times, but fails after about a thousand calls due to a segmentation / access violation fault. Setting a breakpoint in the function doesn't work well because it stops at the first occurrence, when really you want it to stop at the last one – but that involves predicting the future! With a time travel 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 specialized code at runtime. Debugging this code is hard for several reasons. The generated code may have no corresponding source code for analysis tools to work with and could be generated at different addresses on different runs. A time travel debugger allows developers to capture and examine a single run in detail, without the headaches associated with rerunning.

These are just some of the many situations where time travel debugging can save developers a considerable amount of time when debugging.

The key benefits of using time travel debugging

Reveal the root cause of bugs that other methods of debugging cannot reach
Where other methods of error detection and debugging can provide indications of what went wrong, TTD helps developers skip past the need to reproduce bugs, so they can just focus on identifying the root cause. Logs, APM, and analysis tools are invaluable for providing snapshots and signals that something has gone wrong, but they often don't reveal the root cause. Compared to logging in particular, TTD gives the full picture of execution history over time (rather than a snapshot moment) exactly as it happened.

Boost developer productivity
Time travel debugging accelerates bug-fix time by at least 25% (although in practice, users easily see 50–200x productivity savings depending on circumstances):

- It fast-tracks developers straight to the exact root cause without wasting time trying to reproduce the error.
- It reduces the number of debugging iterations down to a single loop.

> Everyone who debugs C/C++ should be using time travel debugging. If you're not using it, you're just wasting time.
>
> – Brian Janes, Senior Engineering Director, **Altair**

ALTAIR

Increased observability and understandability
Being able to time travel and replay code execution history allows developers to observe, comprehend, and learn runtime behavior of legacy code or code they did not write themselves. This is essential for modern development teams to greatly improve productivity across both debugging and maintaining code quality.

> When dealing with unfamiliar code, there is a huge productivity benefit in being able to go backwards and forwards over the same section of code until you fully understand what it does.
>
> – Rob Thompson, Senior Software Engineer, **Siemens**

SIEMENS

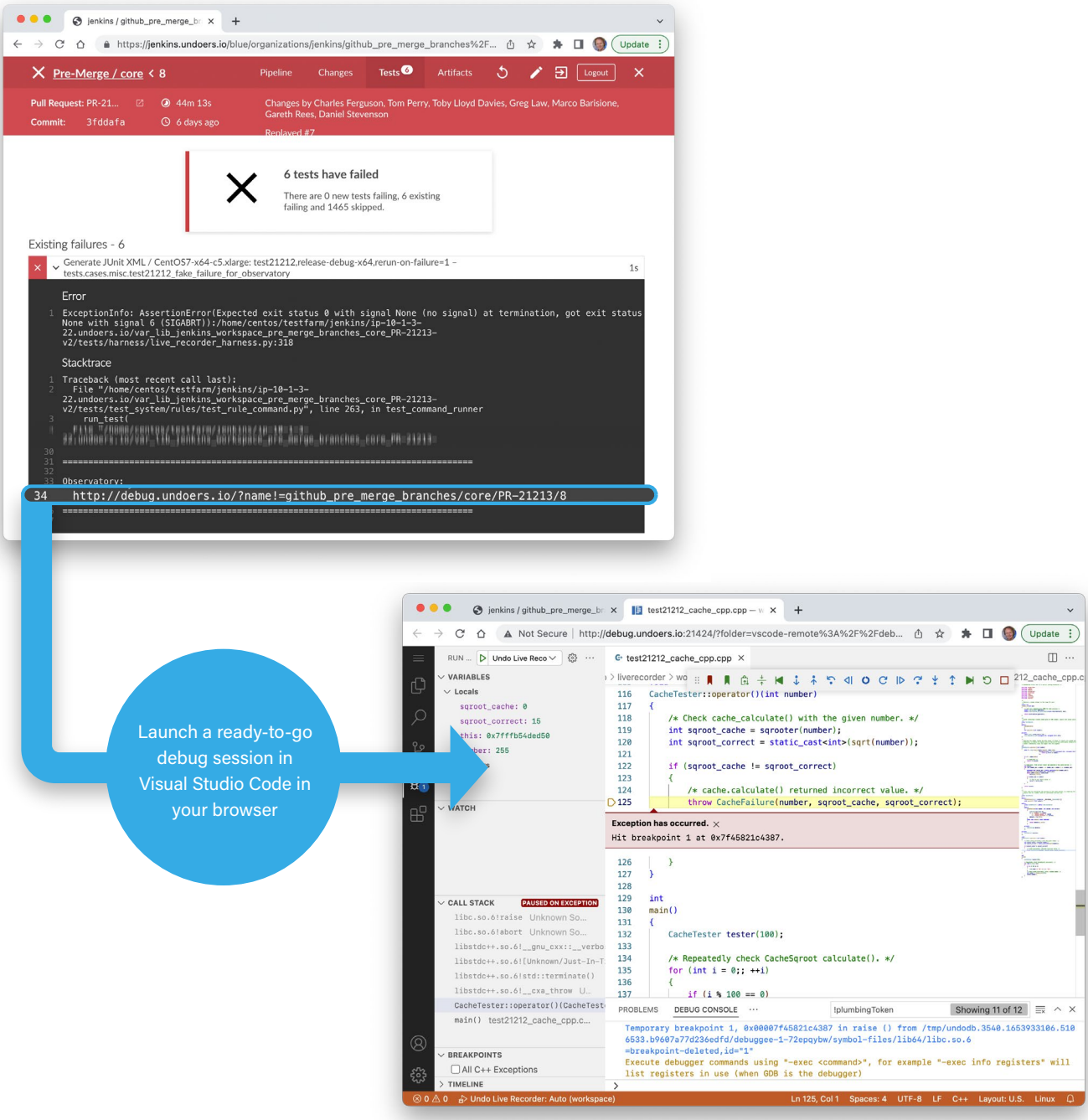
Introducing LiveRecorder

LiveRecorder makes bugs 100% reproducible with time travel debugging.

LiveRecorder lets developers start debugging test failures instantly. LiveRecorder provides a one-click workflow from a test failure to a time-travel debugger placed exactly at the point of failure – skipping the tedious steps usually required to reproduce the problem. Developers working on complex C/C++, Go, and Java software can now save a huge amount of time diagnosing the root causes of new regressions, legacy bugs, and flaky tests. Bugs that took days or weeks to isolate can now be resolved in hours.

Three-step method:

1. **Record** CI / System Test failures to capture the execution of failing test runs, including intermittent failures; store the recordings for later analysis and cross-team collaboration.
2. **Replay** recordings: jump from the test failure (or bug report) straight into a ready-to-go, fully set up, debug session in your web browser.
3. **Resolve** bugs fast by tracing from symptom to root cause in one cycle: go back to any point in the execution history to inspect application state (including contents of all the variables and the heap) and see exactly what your software did.



One-click workflow: from Jenkins (or a bug report) straight into in VS Code to debug the recording (workflow supported for C/C++ development)

Because recordings are portable, developers can pick up a recording and replay it anytime on any machine.

With LiveRecorder, software teams can **significantly improve developer productivity and code quality.**

- **Save time on debugging**, submit code deliverables on schedule, and free up time to build new features
- **Fix known bugs before they hit customers**, by reducing the number of defects bypassing testing

How is LiveRecorder different from other debugging methods?

- **LiveRecorder makes debugging predictable** – compared to using logging, core dumps, printf or standard debuggers.
    - A recording will always behave in the same way anytime, anywhere, and for everyone; no more, "*I can't reproduce on my machine*"!
    - Developers can finally see what the software really did (not what it was expected to do).
    - By removing the guesswork in root cause analysis, LiveRecorder brings certainty to the process of debugging.
- LiveRecorder allows developers to explore how the code is executed – every thread, every variable, every I/O – so **new recruits can debug complex codebases just as efficiently as more experienced team members**.
- LiveRecorder is **the only debugging method that enables effective asynchronous collaboration across teams and time zones**: QA can share links to bug reports with recordings attached (not just just logs or a core dump), developers can share with colleagues links to moments in time in the recording, etc.

- Supports applications written in C/C++, Go, and Java (running on Linux)
- Works on any user-mode compiled code on x86
- Debug in Visual Studio Code (other IDEs also supported e.g. CLion, Eclipse, IntelliJ) or in the command line or Emacs editor

LiveRecorder boosts developer productivity

Save time debugging

Collaborate asynchronously on bug fixing

Learn an unfamiliar codebase faster

LiveRecorder fits in your existing development workflow – no software to install and no set-up required.

And once inside a recording in your VS Code interface (or in the command line), LiveRecorder incorporates the full functionality expected of modern debuggers (such as scripting, conditional breakpoints and watchpoints, full inspection of globals and locals). It also allows these features to be used with the program running in reverse or forward.

Learn more about LiveRecorder

Or to explore some of the features of time travel debugging, try UDB free for 60 days.

Final Thoughts

Most organizations building complex software are still severely limited in their delivery velocity by the rate at which software bugs can be discovered and resolved.

Time travel debugging empowers developers to capture and inspect application state at any point in time during the process failure. By allowing developers to see exactly what happened, it saves a huge amount of time spent on trying to reproduce the problem. It shortens the iterative "*reproduce–guess–test–restart*" traditional debugging cycle down to one loop. This leaves developers free to fast-track to the exact root cause and fix the problem.

Large development teams at companies including SAP, Siemens, Synopsys, Juniper Network, and Palo Alto Networks are already using this technology and see time- and cost-savings that free up developers to code more productively and deliver new features. Time is of the essence while troubleshooting bugs, and every day that can be saved makes a big difference to getting to market faster.

The pressure to increase the speed of software development is growing relentlessly. Now is the time to switch to time travel debugging – a faster, more efficient method of debugging, fit for the 21st century. It's time to Debug Different.

About Undo

Undo is the time travel debugging company for Linux. We equip developers with the technology to understand complex code and fix bugs faster.

Developers spend far too much time figuring out what code actually does – either to understand other people's code or to find and fix bugs. Debugging can be especially time-consuming when software failures cannot be reproduced.

Time travel debugging solves this problem entirely by making bugs 100% reproducible. By bringing time travel debugging to CI and System Test, Undo's LiveRecorder enables developers to save time diagnosing the root causes of new regressions, legacy bugs, and flaky tests.

Thousands of developers across leading technology firms including SAP, Juniper Networks, and Siemens use LiveRecorder to improve developer productivity, development velocity, and software quality.

REQUEST A DEMO

Alternatively, try the technology for yourself. Download your free trial of UDB to explore some of the features of time travel debugging.