2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Time-Travel Debugging for HLS Code

Jonathan Bonsor-Matthews, LightBlue Logic, Cambridge, UK (jonathan@lightbluelogic.com)
Greg Law, Undo, Cambridge, UK (greg@undo.io)
Chirag Goyal, Undo, Delhi, India (chirag@undo.io)
Chris Croft-White, Undo, Cambridge UK (ccroftwhite@undo.io)

***Abstract*— High-Level Synthesis (HLS) code is often developed using a subset of C/C++, along with specialized libraries. This allows HLS engineers to make use of standard software engineering tools and techniques during the development and debug phase of their project. This paper introduces the topic of time-travel debugging, whereby the state of a design can be examined by going backwards and forwards in time. This approach saves huge amounts of effort, allowing the root cause of bugs, including challenging concurrency bugs, to be found with ease and a new codebase to be understood rapidly. Furthermore, it is possible to extract "waveforms" from a time-travel recording and thus allow HLS implementations to be analyzed in a style more familiar to many hardware engineers. We explain the application of time-travel debugging to HLS designs and the advantages, availability and limitations of this approach.**

Keywords*— *Time-Travel Debugging, HLS, SystemC, gdb, Waveforms

I. Introduction

High-level synthesis (HLS) is a design process whereby the behavior of a solution is specified at an architectural and algorithmic level, typically using a subset of a mainstream software programming language, such as C or C++, along with libraries such as SystemC. Several widely-known advantages of writing hardware designs in C or C++ include: faster testing and development by running natively, compared to traditional simulation-based testing; and targeting different Power/Performance/Area (PPA) configurations from the same code by tuning the synthesis tool's configuration.

There is also a lesser-known advantage of writing hardware designs in C or C++: the ability to use industry standard software engineering tools and techniques. There is widespread availability of verification tools: for example, coverage tools (such as gcov), memory checkers (such as valgrind) and various sanitizers (such as AddressSanitizer and ThreadSanitizer); these considerably improve the quality of the behavioral model and add an extra dimension of validation on top of the final Hardware Description Language (HDL) verification stage. At the development and debug stage, a designer can use standard debuggers such as GDB.

Traditionally bugs within HLS designs are found using "brute-force" techniques, such as logging values to a file. It can take a long time to review a log file and adding more data to the output requires rebuilding the code and re-running the test case. Debugging this way is very time consuming and doesn't always find the root cause of an issue.

The objective of this paper is to introduce the powerful, yet still little known, technique of time-travel debugging: being able to debug code both forwards and backwards in time, and demonstrate how it is particularly well suited to analyzing HLS designs.

II. What is Time-Travel Debugging?

A traditional debugger is a computer program which is used to control the flow of another executable. The debugger monitors the execution of the program being debugged, allowing for its execution to be paused at desirable points and for registers, variables and memory to be examined at these points and for the executable to be resumed until another desirable point.

Time-travel debugging provides a significant improvement by enabling the developer to move forwards and backwards through the execution of an application, at will, to quickly home in on an issue.

A time-travel debugger records the execution as it proceeds. Importantly it saves the results of non-deterministic events, including system calls (i.e. user- and kernel-space interactions) and memory-mapped device register accesses. During debugging, the execution can proceed in either a forwards or backwards (reverse) direction: the debugger will use its recording and knowledge of non-deterministic events to re-execute the program in order to recompute memory and register values for any desired point at the program's execution.

These non-deterministic events typically represent a tiny fraction of the instructions executed, meaning the recording's overhead is relatively small [1]. Time-travel debuggers can also make intermittent snapshots, to improve the performance for the developer.

Time-travel debugging is being rapidly adopted by the software industry by developers of large, complex codebases. For example, all the major EDA companies make extensive use of time-travel debuggers [2, 3, 4]. Developers are often faced with exceptionally large code bases (see code discovery in section III), non-repeatable or intermittent failures (see repeated runs as part of a regression flow in section VI) and how to find concurrency bugs (see the examples in section V). Time-travel debugging helps enormously in all these situations and more.

III. Overview of Time-Travel Debugging Techniques

The power to operate both forwards and backwards, but especially backwards, can be seen from Table I. A very common question is: "how did this variable's value change"? To debug this, it is simply a case of executing until the point of failure and then using a "last" command. Often it isn't necessary to fully understand the code path from the data changing to the failure being found to resolve the bug, thus saving substantial development time.

Table I. Traditional and Reverse Debugging Operations

Command	Forward Function	Reverse Function
Step	Step into next function	Step into previous function
Next	Execute next line	Execute previous line
Finish	Return from current function	Execute until just before current function was called
Break (Condition)	Stop execution at a next given location in code. (Optional condition to only stop in specific desired situations).	Same
Watch	Stop execution if a certain variable or memory location changes	Same
Continue	Execute forwards until next breakpoint or watchpoint	Execute backwards until previous breakpoint or watchpoint
Last	Jump to the next time a variable or memory element changes	Jump to the last time a variable or memory element changed

Another significant advantage of time-travel debugging is in code discovery. Sometimes it is interesting to ask the question "How did the execution get to here?" or "Why does this algorithm do that"? Being able to step forwards and backwards through the code aids learning about the parts of the implementation which matter to a particular problem. A good time-travel debugger allows a user to set bookmarks, to map the flow of code through time and easily return to a certain point in the flow.

Furthermore, it is no longer possible for the developer to become "lost" when the program unexpectedly hits or does not a breakpoint. When this happens (which is common while debugging an unfamiliar codebase) the developer can simply jump back to where they previously were and try again.

A time-travel debugger can be run as a standalone command line application, suiting those familiar with standard debugging approaches. It can also be run through an Integrated Development Environment (IDE), often facilitating the learning curve of the tool, with the application host either running locally or remotely, therefore suiting any compute setup – for example, using a plugin within Visual Studio Code.

Additionally, the technique of thread fuzzing can be employed. This allows the debugger to alter the statistics of the thread scheduling, for example: starving threads (minimizing the execution slots granted to certain threads); randomizing the scheduling order; pre-empting threads in known challenging cases (e.g. in places where there isn't usually a jump instruction or just before and after locking operations); and in a feedback-directed manner to force scheduling and/or starvation at execution points where shared memory modification occurs [5].

IV. Applying Time-Travel Debugging to HLS

HLS-style behavioral models are typically architected to have many small execution units operating in parallel. In SystemC, it is quite common for a modest design to have thousands, or even tens of thousands, of SC_THREADS and/or SC_METHODS executing at the same time. Such parallelism increases the probability of bugs common to multi-threaded applications.

Race conditions (threads simultaneously accessing shared data) and deadlocks (threads mutually and inescapably blocking each other) are easily introduced. Sometimes these can be prevented using formal techniques [6] (however, this often requires an understanding of where the issue may present itself – e.g. that one has already located the bug), or using a custom compiler [7].

Other common issues are very difficult to isolate using traditional techniques, such as: memory corruptions, intermittent bugs and incorrect results.

V. Examples

In this section we present how a developer might use time-travel debugging to find the location of an incorrect value being stored in memory. We then discuss these techniques as applied to Race Conditions and Deadlocks, and code exploration.

Firstly, we provide a simple example to see how rapidly we can find a bug using time-travel debugging. This example makes use of the examples in Accellera's SystemC reference code [8]. Let's say we get to a stage in our debugging where we know that an incorrect value is being stored into **data* at this point in the code (nb the point of this example is to show that we don't need to understand much of the code to find the location of the bug). The yellow arrow denotes the execution is stopped at this point, where the value of **data* is being updated with an incorrect value from *MEM[]*:

```
 99  inline simple_bus_status simple_bus_fast_mem::read(int *data
100                                        , unsigned int address)
101  {
102    *data = MEM[(address - m_start_address)/4];
103    return SIMPLE_BUS_OK;
```

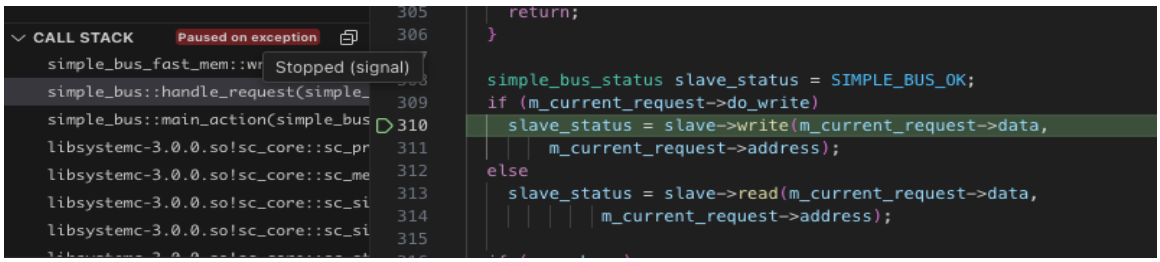
We now have two options for debug, by considering the question: are the data in *MEM[]* incorrect or is the addressing into *MEM[]* wrongly calculated? In practice, we must guess which is the most likely and note the fork in our debugging process to return later if we hit a dead-end (we should bookmark this point in time in our debugger, to return easily later). Here, we'll start with the data corruption option (it's a guess).

We can find out where the *MEM[]* data were last changed by entering the command: *last MEM[(address - m_start_address)/4]* (it is also possible to do this through the GUI). The *last* command sets a watch point on the data of interest and continues the execution in a backwards direction, until the data were last modified:

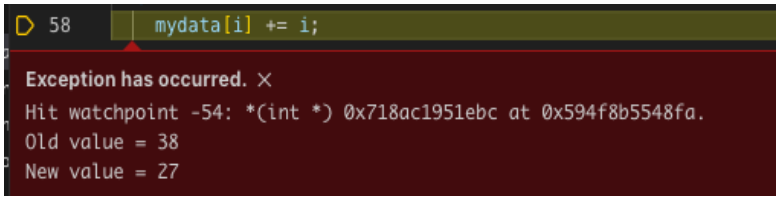
```
106  inline simple_bus_status simple_bus_fast_mem::write(int *data
107                                        , unsigned int address)
108  {
109    MEM[(address - m_start_address)/4] = *data;

Exception has occurred. ×
Hit watchpoint -46: *(int *) 0x594f8ca76d88 at 0x594f8b54fe5d.
Old value = 38
New value = 27
```

This shows us that the debugger wound back execution until the data at *MEM[]* changed: when the program was running forwards at this point, it updated the data at *MEM[]* from 27 to 38. Having now found where *MEM[]* was modified, we can go one step up the call stack to see that **data* here is the *data* passed in through the *request*:



We now need to find where those data were set up, by entering the command: *last *m_current_request->data*. This will, again, cause the execution to continue in a backwards direction until the data were modified:



We see that "*mydata[i] += i*" is the line of code of interest – the understanding phase can now start, to compare what that area of code should do and what it currently does.

Note that we found the location of the problem very quickly. We didn't need to worry about what the intermediate code was doing. We didn't need to keep track of the many threads running in parallel, we let the debugger handle the complexity for us. We were able to follow the dataflow backwards several steps, easily identifying the root cause of the problem. Had we gone down the wrong path in our debugging, that wouldn't have been a problem either – we could have jumped back to a bookmark or moved forwards and then tried backwards again, without restarting entirely!

Race conditions can occur when multiple tasks execute concurrently, with correctness being dependent on the order of events. For example, say two threads are trying to increment a counter: it is critical the read, increment and write-back steps for each thread aren't interrupted by the other thread, otherwise they might both "increment" the counter to the same value (i.e. thread one reads, thread two reads the same values, and then both threads increment from the same value). In a real hardware design, a simple example is a shared arbitrated data bus: for example, if data are being mixed from multiple sources or transmitted from the wrong source.

In HLS designs, it can often be preferable to avoid the use of data access controls (e.g. locks or mutexes), as they can increase the latency and area and reduce the throughput of a design. It's precisely this desire which increases the chance of race conditions occurring in complicated HLS designs.

Race conditions can be notoriously difficult to solve. Some race conditions happen sporadically, so it can take time to find the failing case and when we do, we want to be able to debug without losing the context – this stops us adding extra debug later (e.g. further logging), but a time-travel recording of the failure event is perfect! There can be a huge number of operations between the error and the same error being noticed.

Typically, several tools are used to help find race conditions [9]:

- Static analysis tools (e.g. Coverity [19]) can identify certain classes of race conditions. Such tools can be expensive and suffer from false negatives (they don't guarantee to catch all such issues) and false positives (they often flag problems that aren't really problems).
- Dynamic analysis tools (e.g. ThreadSanitizer [20, 21] and Helgrind [22]) can identify certain races or deadlocks, by flagging if one thread writes memory after another thread writes to the same memory,

before any thread has read it – this could be a great clue, but it could be a false positive. Also, they tend to be most useful for shallow/trivial race conditions such as an obviously missing lock; often such tools are of little help in identifying the root cause of such races, merely that such races exist. Such tools can slow down a program by 10-100x, which in turn could be enough to prevent the race condition occurring.

- Logging output can help a developer identify what a program is doing and pinpoint issues in flow. However, logging often takes up an enormous amount of disk space, is time consuming to read, typically requires several attempts to get the right values outputted and can, also, slow down or serialize execution to a point that the race condition doesn't occur – Heisenbugs [10]. Tracing frameworks, such as DTrace [24], can provide extra detail, however adding probes requires re-running the executable, which takes significant time, and might not work for non-deterministic systems.

Time-travel debugging works reliably for race conditions. Once a value has been noted as being incorrect, the developer sets a watchpoint on the variable, so the debugger stops whenever that variable changes. By continuing the execution in reverse, the developer can see when the value is changed and quickly build up a picture about the sequencing of the modifications. Critically, the developer doesn't need to understand what all the concurrent threads are doing, nor to understand the entire code base, to navigate backwards to find the cause of failure. Typically, a developer can very quickly find non-atomic accesses to data in this way, and then know immediately the type of unsafe programming patterns in a design to be fixed (e.g. by adding locking or re-architecting to guarantee safety without locking).

Deadlocks occur when two or more threads are blocked waiting for a resource, preventing themselves from sending or freeing the same resource. A simple example in a real hardware design is two modules with a handshake mechanism: module A is waiting for a token from module B and, in parallel, module B is waiting for a token from module A - both modules are blocked by each other from proceeding.

In SystemC, deadlocks can be readily introduced in designs where SC_THREADs are synchronized by waiting on and notifying with an sc_event and/or using an sc_mutex type lock to control data access. The same deadlock pattern is seen using other HLS primitives.

Using a time-travel debugger, deadlocks are relatively simple to debug. The debugger is attached at the point when execution, at least in the offending threads, has stalled. The developer can, optionally, look through the backtrace of all the threads to see which threads are stalled. The developer sets breakpoints at the points where the code needs to wait for the resource (e.g. a "lock" function) and where the code would free the resource (e.g. an "unlock" function). The developer executes the code in reverse to understand the problematic sequencing leading to the deadlock and then can resolve the issue, by fixing the bug or rearchitecting the resource management as appropriate.

Often, engineers are tasked with adding a feature, debugging an issue or joining a team to work within a part of a large codebase, with much of which they are unfamiliar. Using a time-travel debugger, an engineer can start from a point of code in the middle of execution and set a breakpoint. Execution can proceed (in either direction) to this breakpoint, from which point the developer can explore:

- how the execution reached this point,
- the value of variables and memory at this point in time,
- the state of other threads at this time, and
- can use watchpoints to understand where variables and memory are modified.

Good time-travel debuggers also offer the feature to bookmark a relevant point in the code and in time, further allowing an engineering to understand the flow throughout execution. There is no possibility of the developer debugging becoming "lost" and having to restart a long debug session over again – they can always get back to familiar territory, like hitting the "back button" on a web browser.

VI. HLS Debug Best Practices

Many time-travel debuggers offer the facility to record a failure and debug it after the event. This is powerful as it allows users to debug an issue without having to reproduce it themselves: e.g. debugging a customer's failure; debugging a regression failure without needing to check out the exact code version; or debugging an intermittent failure.

Time-travel debugging should be built into a regression flow. It can take a long time to run through a set of tests, some of which might only present intermittent failures. In some setups, reproducing a bug can involve a lengthy process of checking out the code at the right revision and a long build and test cycle.

A regression flow knows when a test has failed. This test can then be re-run through a time-travel debugger's recorder and the failure recorded. The engineer can now simply connect their time-travel debugger to the recording to start work on the issue: no need to reproduce the bug directly themself.

Sometimes the tests fail intermittently in which case the failure recording attempt might need to be repeated multiple times to get the correct recording or all the tests could be recorded (at the expense of some disk space).

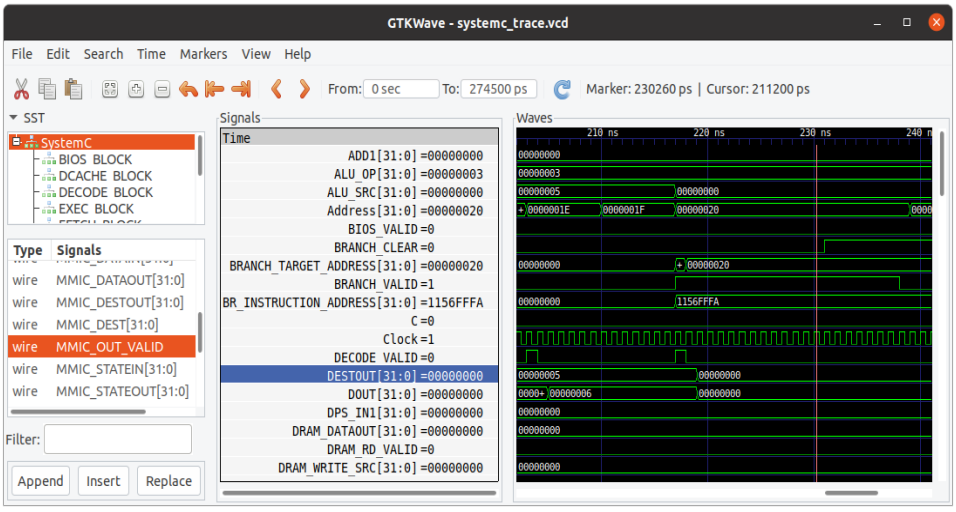
Engineers should write their code thinking about design-for-debug. It is helpful if an error can be detected within the same executable as that which would need to be debugged. Some systems run a test which outputs to a file and then compares that output file with a known reference, flagging an error when a difference is detected. Consider instead a re-architecture so that the executable reads in the known reference itself and flags the error internally should there be a difference between the executable's output and the read-in reference. In the latter case, the debugger can be stopped at the first detection of the error and then a watchpoint (or the last command) can be used to discover quickly where the erroneous value came from and the bug isolated.

Intermediate variables can be helpful for debugging, for example an intermediate timestamp or calculation can be used to set a conditional breakpoint. During compilation to HDL these variables will be optimized out meaning that this approach adds benefit but doesn't harm the final synthesized output.

Assertions should be used readily to act as sanity checkers for assumptions made within a design. If during a test failure (or indeed a successful test!) an assertion fires, the engineer debugging it already has a very good starting point – put the execution in the debugger at the point where the assertion fires and then debug backwards to understand why the assumption was falsified.

VII. Waveforms

It is possible to extract "waveforms" from a SystemC program to be analyzed in a style more familiar to many hardware engineers. The image below shows a screenshot of GTKWave displaying the output of one of Accellera's SystemC examples [8] configured to dump a VCD file.



It is also possible to extract "waveforms" from a time-travel recording without needing to rebuild or re-run the executable. This provides a powerful option for hardware engineers, especially those transitioning to HLS design and implementation. A waveform viewer can be linked to a time-travel debugger, so that by clicking on a transition in the waveform view the debugger can synchronize to the point in the time-travel recording where the transition occurred, allowing full time-travel debug starting from the same moment.

VIII. Public Availability

There are several time-travel debuggers publicly available [11]. HLS designs are most commonly programmed in a C or C++ based language (often including associated libraries such as SystemC) [12]. The most useful known time-travel debuggers for HLS engineers are:

- Linux: rr [13] and Undo [14]
- Windows: Microsoft's Time Travel Debugging Tool [15]

Each time-travel debugger listed above offers support for either Linux or Windows, and offers a different set of features and different performance.

IX. Limitations

A time-travel debugger, as for a traditional debugger, presents a small performance slowdown, but how much depends vastly on the workload [13, 16, 17]. However, in practice a good regression system can be configured to record the failing tests offline, ready for later debug. The overhead of the recording size depends on the length of execution and the number of non-deterministic events but is typically much smaller than standard HDL simulation logs [1].

Thread fuzzing, as discussed in section III, is implemented for standard OS threading mechanisms (e.g. Linux's) pthreads. HLS tools or libraries often have their own scheduler, as an HLS design might have thousands or even tens of thousands of threads running in parallel so using standard OS primitives could be too slow. As such, the thread fuzzing supplied with a time-travel debugger wouldn't work out of the box, but could be implemented by modifying the supplied scheduler (where source code is available).

The examination of thread info is discussed briefly as an optional step when solving deadlocks, in section V). For the same reason as for thread fuzzing, this thread information isn't available directly from a time-travel debugger for HLS tools or libraries using their own threading module, but could be extracted by looking through the scheduler's data structures (again, where source code is available) or ensuring a standard linux threading module (i.e. pthreads) is used.

X. Results

Quantifying the time saved by using such techniques is challenging – it depends on the bug to be found and the codebase to be understood. We have seen data suggesting that finding a bug is often at least four times faster using time-travel debugging than traditional techniques, and sometimes significantly faster still.

This 75% reduction in time spent finding bugs and the speedup of learning code leads to a significant improvement in the effectiveness of the engineers working on a project. This enables faster time-to-market and more time to implement additional features, to add additional coverage closure and to enable more application layers to be brought up before silicon is returned – in short enabling the advertised benefits of Shift Left [23].

Further, by being able to complete an HLS design faster, this feeds well into the shift-left paradigm [18], whereby the HLS implementation can be tested with the entire software stack running natively (say on x86), then against an FPGA synthesis of the same implementation, both significantly before tape-out, and then finally against a silicon production version of the same implementation.

When used for modelling the reduced debugging time leads in turn to increased throughput and can result in a wider range of scenarios and applications being modelled, and thus results in more inefficiencies being discovered prior to chip tape out, and so result in improved PPA characteristics.

XI. Conclusions

We have demonstrated that it is beneficial to apply standard software development tools and techniques to HLS development. We have shown that the concept of time-travel debugging, which allows the flow of an executable to be controlled both backwards and forwards in time, vastly simplifies the task of debugging and code discovery. We have highlighted that the improvement in effectiveness can then lead to faster time to market and better early testing of the implementation through shift-left. One of the key advantages of HLS is to be able to take advantage of the advanced tooling that is available to software developers; time travel debugging is particularly compelling. Finally, producing "waveform" views from a recording allows debug flows with which digital design engineers are more commonly familiar, and so aids the transition to HLS.

Acknowledgment

The authors would like to thank the team within Undo for supplying data and examples to inspire this paper and our connections across the semiconductor industry for sharing their experiences of developing and debugging HLS designs.

References

[1] Undo, "Documentation", undo.io, https://docs.undo.io/TechnicalDetails.html (accessed June 16, 2025)
[2] Undo, "How Synopsys saw a boost in customer satisfaction & developer productivity after investing in Undo", unfo.io, https://undo.io/case-studies/synopsys-boosts-customer-satisfaction-and-developer-productivity/ (accessed June 16, 2025)
[3] Undo, "Siemens EDA accelerates defect resolution with Undo", undo.io, https://undo.io/case-studies/siemens/ (accessed June 16, 2025)
[4] Undo, "Cadence tracks down mission-critical bugs on customer sites within hours, not months", undo.io, https://undo.io/case-studies/cadence-design-systems/ (accessed June 16, 2025)
[5] Undo, "Expose Concurrency Bugs With Thread Fuzzing", undo.io, https://undo.io/resources/thread-fuzzing-wild/ (accessed June 16, 2025)
[6] S. Beyer and D. Strasser, "Detecting Harmful Race Conditions in SystemC Models Using Formal Techniques", DVCON 2015
[7] N.Blanc and D. Kroenig, "Race analysis for SystemC using model checking", Proc. IEEE/ACM Int. Conf. Comput.-Aided Des., Nov. 2008, pp. 356–363.
[8] Accellera, "SystemC Reference Implementation", github.com, https://github.com/accellera-official/systemc (accessed June 16, 2025)
[9] Undo, "Debugging Race Conditions in C/C++", undo.io, https://undo.io/resources/debugging-race-conditions-cpp/ (accessed June 16, 2025)
[10] Wikipedia, "Heisenbug", wikipedia.org, https://en.wikipedia.org/wiki/Heisenbug (accessed June 16, 2025)
[11] Wikipedia, "Time travel debugging", wikipedia.org, https://en.wikipedia.org/wiki/Time_travel_debugging (accessed June 16, 2025)
[12] Wikipedia, "High-level synthesis", wikipedia.org, https://en.wikipedia.org/wiki/High-level_synthesis (accessed June 16, 2025)
[13] rr, "rr", rr-project.org, https://rr-project.org/ (accessed June 16, 2025)
[14] Undo, "Undo", undo.io, https://undo.io/ (accessed June. 16, 2025)
[15] Microsoft, "Time Travel Debugging – Overview", microsoft.com, https://learn.microsoft.com/en-us/windows-hardware/drivers/debuggercmds/time-travel-debugging-overview (accessed June 16, 2025)
[16] Undo, "Undo Performance Benchmarks", undo.io, https://undo.io/resources/undo-performance-benchmarks/ (accessed June 16, 2025)
[17] Microsoft, "Use time travel debugging to record and replay ASP.NET apps on Azure VMs", microsoft.com, https://learn.microsoft.com/en-us/visualstudio/debugger/debug-live-azure-virtual-machines-time-travel-debugging?view=vs-2022 (accessed June 16, 2025)
[18] Cadence, "Virtual Platforms to Shift-Left Software Development and System Verification", cadence.com, https://community.cadence.com/cadence_blogs_8/b/fv/posts/shift_2d00_left-using-helium-virtual-and-hybrid-studio-for-hw-and-sw-co_2d00_verification (accessed June 16, 2025)
[19] Black Duck, "Coverity Scan Static Analysis", coverity.com, https://scan.coverity.com (accessed June 16, 2025)
[20] Clang, "Clang 20.0.0git documentation", clang.llvm.org, https://clang.llvm.org/docs/ThreadSanitizer.html (accessed June 16, 2025)
[21] Google, "ThreadSanitizerCppManual", github.com, https://github.com/google/sanitizers/wiki/threadsanitizercppmanual (accessed June 16, 2025)
[22] Valgrind, "Helgrind: a thread error detector", valgrind.org, https://valgrind.org/docs/manual/hg-manual.html (accessed June 16, 2025)
[23] Larry Smith, "Shift-Left Testing", drdobbs.com, https://www.drdobbs.com/shift-left-testing/184404768 (accessed June 16, 2025)
[24] Wikipedia, "DTrace", wikipedia.org, https://en.wikipedia.org/wiki/DTrace (accessed Sep 1, 2025)