A REVIEW OF REVERSE DEBUGGING

Jakob Engblom

Wind River Systems

ABSTRACT

Reverse debugging is the ability of a debugger to stop *after* a failure in a program has been observed and *go back* into the history of the execution to uncover the reason for the failure.

Long the dream of programmers, over the past decade, reverse execution has become a practical technique available in a number of free and commercial tools.

This article will review the history and techniques of reverse debugging, as researched, implemented, and used from the 1970s until today. We will provide some personal insights into reverse debugging, from our own practical use of one such tool, Wind River Simics.

Index Terms— Software Debugging, Computer Simulation, Review, Computing History

1 INTRODUCTION

In this paper, actual *reverse debugging* is defined by the ability of a debugger to plant a breakpoint in a program and then proceed backwards in time until the breakpoint triggers. Reverse debugging is related to various implementation techniques such as record-replay, tracing, deterministic reexecution, reverse execution, checkpointing (or shapshotting), and program scheduling control, but it is really an application of those techniques to solve a debug problem that is core to the issue. Reverse debugging is sometimes known as *bidirectional debugging*.

As shown in Figure 1, reverse debugging is quite different from classic *cyclic debugging*, in which you run and rerun a failing program under debugger control to diagnose a failure. For cyclic debugging to be practical, it is pretty much required that each run of the programs behaves and fails in the same way.

For cyclic debugging to work, program under investigation has to be fundamentally deterministic. This is usually the case for non-interactive single-threaded programs, but not the case for real-time programs, parallel programs, or programs that involve some kind of asynchronous input/output (including files that change their contents between runs). In such circumstances, reproducing an error by rerunning a program is likely to fail (i.e., not hit the bug), or even hit different errors than the initial error that prompted the debug session. Attaching a debugger often disturbs the timing of a parallel program, easily masking errors (so called *Heisenbugs*, where the act of observation changes the system to hide the bug).

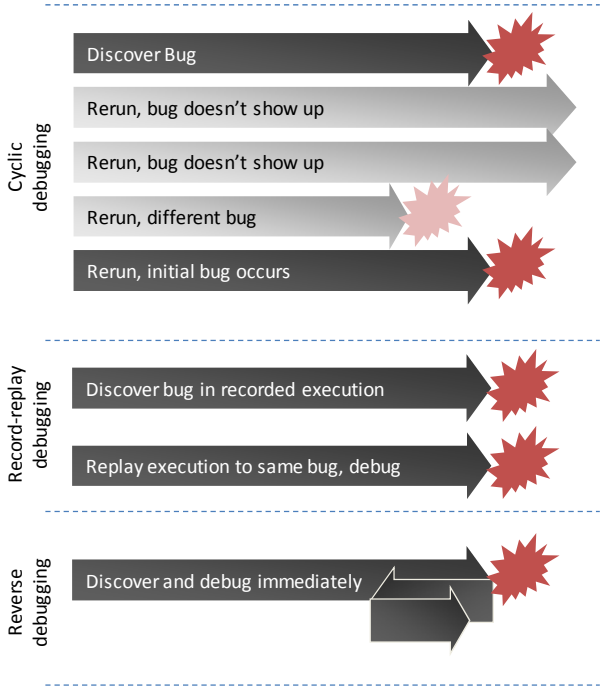

Figure 1 Fundamental debug techniques

Reverse debugging is one approach to tackle the debug of intermittent bugs, based on the idea of rather than trying to reproduce the issue in a separate run from the beginning you work inside the run that has already failed, and reverse back into its execution to diagnose the issue. This does require the ability to reproduce the past, and how this can be achieved is the topic of this paper.

An intermediate form between cyclic and reverse debugging is *record-replay debugging*. In record-replay, a non-deterministic execution is recorded and later replayed. Debugging is still performed in the forward direction: you cannot go back in time to a previous point in the program execution without restarting the replay run, nor can you trigger breakpoints backwards in time. Record-replay debug allows cyclic debugging to be applied to non-deterministic (but still recordable) programs. Record-replay is easier to

implement, since it tends to require fewer changes to the debugger core than actual reverse debugging.

1.1 Going to a Past State

Fundamentally, it is impossible to actually reverse the execution of a computer program. Many machine instructions destroy information; just consider an operation such as XORing a register with itself or writing a new value to a memory location. There is no way to take the state after the operation and infer the state before the operation. Thus, we need to reconstruct history from saved information to obtain the past state of a program or computer system.

There are two ways to reconstruct past state. Either, we *record* it completely in a log, or we *reconstruct* it by executing forward from some saved state. Reconstruction only needs to record the information that cannot be reconstructed, which generates far smaller logs. Most practical solutions have chosen the reconstruction approach as the method to obtain the details of the system state at some particular point in time. Complete log recordings (traces) are feasible for some embedded systems, where extensive trace capabilities exist in hardware, but tend to be too slow if implemented in software.

1.2 Debugger Scope

The scope of reverse debugger solutions varies depending on the problems that the debugger wants to solve and the nature of the implementation. Figure 2 shows an overview of the possible scopes of a debugger, as to what is included in the reverse debug process (i.e., what is reversed in the sense that its past state is presented by the debugger).

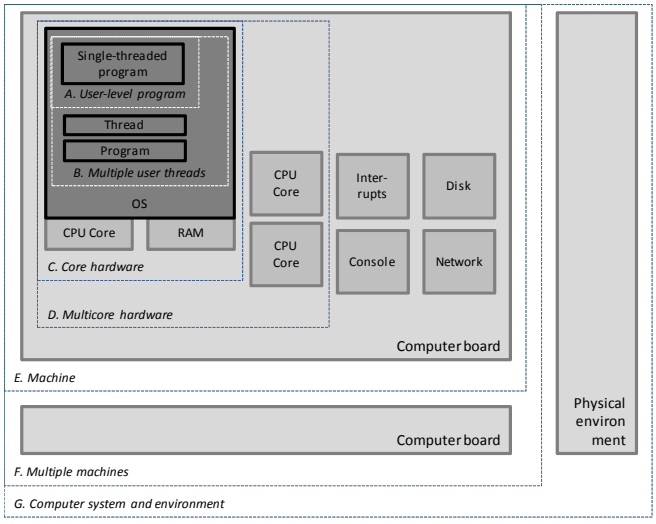

Figure 2 Possible Debugger Scope

User-level reverse debugging works on a single application (*A* and *B* in Figure 2). User-level debug can often be achieved in simpler ways than system-level debug, but suffers from some limitations and can get very complex when dealing with IO.

System-level reverse debugging works not just on a user-level program, but rather on the scope of a full machine (minimally the operating system). Scopes C to G in Figure 2 are all system-level scopes.

An additional issue is the handling of *multiple* software threads or *multiple* hardware processors. It is in general much simpler to debug and reverse a single thread of execution than multiple threads. However, handling multiple threads is pretty much necessary today.

A related issue is the question of the code being run. Do we run *instrumented or unchanged programs* in the debugger? Instrumentation can help collect the information needed to construct the past state of a program. Example instrumentation approaches are compiling programs with extra instrumentation, using instrumented OS libraries, or special scheduling modes. Typically, running the same program binaries as on the real system is better.

Cross-target or host-based. Does the debugger run on the same machine as the program being debugged, or does it target a remote (or simulated) system? Normally for embedded systems, cross-target debug is the most interesting, while for desktop and server systems, host-based is the most natural.

1.3 History and Review

The idea of reverse debugging has been with us since the very beginning of computer programming, but was long dismissed as basically impossible or at least totally impractical. Advances in memory size, computing power, and debugging technology have made it feasible in recent years, even for fairly large systems and software sets.

The first presentation of a practical reverse debugging system that included the ability to step backwards and run backwards to breakpoints appeared in 2000 [1]. Before this, all prior research had focused on how to save and reconstruct old program state for record-replay debugging.

This paper will review the technology issues associated with reverse debugging, record-replay debugging, and related techniques. The presentation is structured after the technological issues, rather than presenting work in chronological order.

2 RECORD-REPLAY SOLUTIONS

There are a number of solutions that implement record-replay debugging, but not actual backwards step and/or breakpoints and thus do not quite qualify as reverse debuggers.

The simulator used in the *Data General Eagle* project in 1980 featured the ability to record and replay microcode executions in order to facilitate debugging [2]. It seems that this kind of trace and replay is a common feature of many

older simulation systems, as the idea is fairly obvious and the storage requirements for small-scale simulations modest.

The *Instant Replay* debugger from 1987 allowed the deterministic replay of parallel user-level programs on the BBN Butterfly parallel computer [3]. It required that programs adhered to certain communications styles to work, and relied on instrumenting the OS to capture all thread interactions. By replaying the thread interactions, *reconstruction* could be used to recreate the actual variable values of each thread. This was still considered a useful debugger for issues related threaded programs, in particular deterministic threaded bugs.

The *repeatable scheduling* system from 1996 instrumented system libraries and applied binary code modification to enable deterministic replay of multithreaded applications on a single processor [4]. Quite similar to Instant Replay, but with no requirements on how programs are coded, and no special debugger built around it.

Microsoft Visual Studio 2010 IntelliTrace records a history while debugging programs running on top of the .net CLR virtual machine [5]. It only collects selected data and lets the user move backwards in time to selected points to view past state. A nice feature is the ability to send the history file to other developers to share a debug session.

A weaker form of record-replay debugging is to replay only the stimuli known to cause an issue. In particular in networked systems, it is common practice to subject systems to a generated packet stream for testing (including fuzz testing), and then to replay the same packet stream in case the system fails. Packet streams or other input streams can also be captured on live deployed systems, and brought back to the developer labs to attempt to recreate issues from the field. Such trace replay does not really replay the same *system execution*, but it often works to recreate bugs that depend (mostly) on the nature of the packet stream.

3 TRACE-BASED REVERSE DEBUGGING

Trace-based debug solutions are a special case in that they simply record "everything". Most trace-based approaches use hardware debug support to capture a trace of the execution of a system, at level *C* in Figure 2. Then, in a separate step, a debugger reads the trace and performs reverse debug operations on the trace.

The scope of a trace-based solution is limited by what can be traced in a coherent way. Thus far, this has practically limited it to single processors, as time-stamped coherent traces from multiple processors are very difficult to obtain (that would be level *D* in Figure 2). As debug hardware improves, it is possible that this can be extended to an entire SoC including IO, which would be similar to *E* in Figure 2. The reverse time window is also limited by the capacity of the trace hardware to accept complete trace data.

Hardware trace-based approaches are cross-target, partially system-level (basically, only the processor), multiple-thread, and can work on unchanged programs. The debugger also needs to recreate the target OS state from the trace in order to apply debug operations to OS abstractions.

To implement reverse breakpoints, the debugger scans backwards over the trace until some condition is satisfied. The implementation of reverse is all in the debugger core and its user interface, it does not affect the debugger backend.

The first practical commercial reverse debugger, the *Green Hills Time Machine* [6], was based on trace-based reverse, and came to market in 2003. *Lauterbach CTS* is another hardware-based debugger with reverse abilities [7], but apparently it does not feature reverse *breakpoints* and therefore is probably better classified as a record-replay debugger rather than a true reverse debug system.

There are also a few complete-trace systems based on software, which operate at level *A* or *B* in Figure 2.

The *Omniscient Debugger* presented in 2003 targeted programs running on top of a Java Virtual Machine [8]. It instruments the target program to log all state changes, and used a big lock to serialize execution of multiple threads. The implementation is admitted to be almost worst-case inefficient, but the debugger was still considered useful for some real work. The debug UI did feature backwards breakpoints, making it a true reverse debugger. This approach was user-level, instrumented programs, multiple threads, and host-based.

The *gdb 7.0* built-in record target which appeared in 2009 is a trace-based reverse debugging solution. It records the effect of every target instruction in a single target thread. It is very slow, but provides the only open-source and free solution for reverse debugging. It lacks a concept of time, and a user cannot move the execution to a particular point in time or run backwards for a certain time. The availability of gdb was also crucial to getting reverse execution features into mainstream Eclipse, in 2009.

Using the gdb-MI or gdb-serial remote protocols, gdb can also be used as a frontend to other commercial reversible debugger backends such as *Simics* [11] and *UndoDB* [12]. In this case, the gdb debugger does not know how the backend implements reverse, only that it understands the commands required to perform reverse. Thus, gdb offers a general frontend that can be used with multiple backends.

4 RECONSTRUCTION-BASED REVERSE DEBUGGING

Figure 3 shows the basic technique used to reach a certain previous point in time in a system execution in a reconstruction-based reverse debugger. The approach works by first setting the system state to a starting point that can be reliable reproduced, and which is *before* the desired target time. Such a system restoration capability is known as *checkpointing* or *snapshotting*, and we will call it

checkpointing in this paper. The system then executes forward until the desired point in time has been reached. In the simplest case, a single checkpoint is maintained at the start of execution [9]. To reduce the user's waiting time most reverse execution approaches add additional checkpoints during the execution forward.

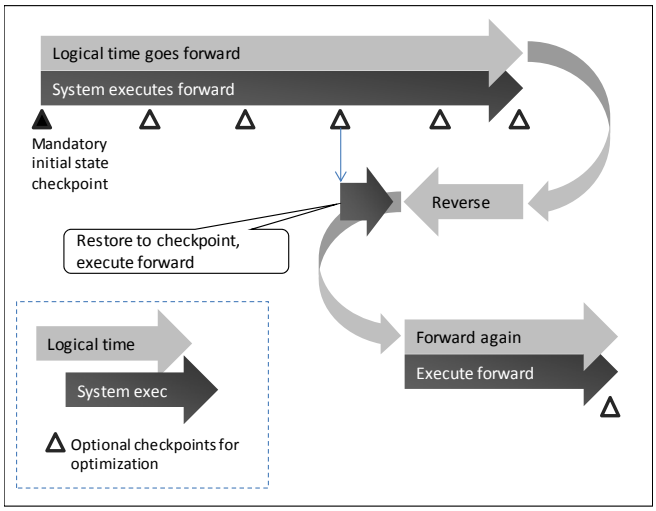

Figure 3 Reconstruction-based Reverse Principle

The execution forward has to be *deterministic* so that we always reach the same final state. If the system is subject to any asynchronous inputs, these have to be provided during the forward execution in order to make it possible to reach the same final state. Such inputs have to be provided with total precision in both data contents and timing, or the execution will diverge and become useless (at least for the purpose of reverse debugging).

4.1 Managing Time

To function, a reverse debugger needs to have a concept of time. Time is used to stop the reexecution process at the right point in time, and to trigger recorded inputs. How time is represented depends on the recording and reconstruction technology used. The most common time bases are executed instruction counts and clock cycles.

In systems with multiple threads or processors, time has to be ordered between the concurrent processes. How to implement the ordering intimately depends on the implementation of the reverse debugging system itself. For example, partial orders based on thread interaction [13], a total serialization of all threads using a single lock [8], or the scheduling scheme of a virtual platform [11] have all been used.

Most (but not all) reverse debug systems exposes time to the user. Typical operations available include being able to mark certain points in the execution to later come back to them (bookmarking), and running backwards or forwards for a certain amount of time. Some debuggers maintain a stack of past stopping points, allowing a user to logically browse backwards in their examination of the debuggee.

4.2 Reverse Step and Breakpoints

The two main operations you want in a reverse debugger is to be able to step backwards and run backwards until a breakpoint hits. The fundamental methods for this were presented as part of the *Bidirectional Debugger* in 2000 [1].

To step backwards, you essentially use the reconstruction approach to run one step short of the current position. You note the current time t in the system, and ask the reconstruction system to get to time t - 1 (in whatever time units are being used). For systems with multiple threads of control, the step can either be global or within the current thread, that depends on the debugger user interface decisions.

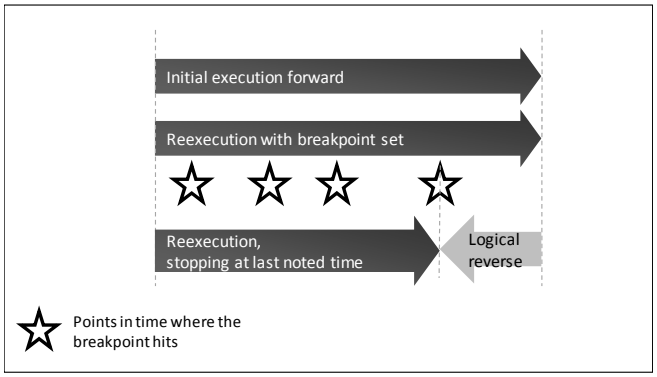

Figure 4 Reconstruction Reverse Breakpoint

To go backwards until a breakpoint hits, a somewhat more complicated approach is needed. All published work use the technique shown in Figure 4. The state is rewound to a checkpoint and then the system is executed forward once with breakpoints set. When a breakpoint hits during this rerun, the time is noted, and execution is resumed without notifying the user. Once the debugger reaches the current point in time, it stops the execution. It then goes back to the start again, and executes forward to the time of the last breakpoint hit seen during the rerun. Thus, it is enough to rerun the program twice to go back to the previous breakpoint. This is far better than stepping backwards a step at a time and checkpoint for breakpoint conditions, which would be quadratic in complexity and thus unusably slow. When using multiple checkpoints spread out in time, the strategy is normally to work backwards one checkpoint at a time.

4.3 Reversible State

The question about just which state to consider part of the reversing process is fundamental to the implementation of a reverse debugger based on reconstruction. Reconstruction can be applied to any level in Figure 2.

Obviously, for any computer program, the core state is the processor registers and memory system used to store the variables, call stack, current program location, and other data that the program operates on. From a hardware perspective, this is *C* or *D* in Figure 2.

When programs interact with the external world, things get more complicated, however. Fundamentally, as noted above, the program that we are reversing needs to get the same inputs at the same points in time that it got in its initial run. This has to be based on recording anything that cannot be reconstructed. How this is implemented varies widely between different reverse debugging systems, and is arguably the point where the most diversity is seen. If we look at Figure 2, we basically have to record any information that crosses the boundary between what we are able to reverse and the outside.

Thread interactions between threads in an OS are an important part of user-level reverse debugging systems. Both data exchange and synchronization have to be recorded. It is also necessary to record the asynchronous task switches where the OS interrupts a thread to schedule another thread, since the order of access to shared data is very important to reproduce to exactly reproduce a parallel program execution. For a *system-level* solution, thread interactions are implicitly reproduced by the reproduction of the operating system execution on top of the hardware.

An interesting twist on the management of thread interaction replay is to bend the rules a bit and change the order of replay of certain segments of code. This can be used to analyze the behavior of program by comparing the results from different replay orderings [14].

Console input and output is most commonly handled by replaying input data and ignoring output data during the reconstruction phase. This means that the display of a program (text or graphics) gets frozen at its final state [12]. This is safe but a bit unsatisfying in cases where output is of interest to the user. To reverse the contents of a display, the reverse debugger has to have control over the output system, which is the case for tools like virtual machines [9] and virtual platforms [11], but rarely for user-level soluions.

*File system*s. For a user-level solution, files leave a record of a program execution that is permanent from run to run of a program. Thus, file access has to be in some way virtualized or redirected so that a program always sees the same version of the file during its reconstruction phase. In a system-level reverse debugger, it is simply a matter of including the disk as part of the system state that needs to be reversed, making it no different from memory [9][11].

Network traffic, *sensor inputs*, *interrupts* from external sources, and similar interactions that reach out of level *E* in Figure 2 are most commonly handled by recording all inputs. Outputs are typically suppressed during reexecution, as the outside world does not expect or know how to deal with the information anyway. In reverse debugging systems based on full-system simulation and virtual platforms, it is possible to include the entire system including models of the environment into a single reversible whole [11].

4.4 Gaining Control

A key problem that needs to be solved for reconstruction-based reverse debugging is how to get control over the target system in order to be able to coerce it to repeat its execution. Standard computer systems are not repeatable or controllable to any useful extent, and thus some kind of intermediate layer has to be introduced to grab control.

Instrumenting the standard system libraries to capture inputs, outputs, and thread interactions is a necessary part of any user-level recording approach [1][3][4][12][13]. To handle multithreaded code, help from the OS is usually needed to capture events such as thread scheduling and descheduling [3][13]. Usually, it is also necessary to change the user code itself, to add time counters, record values of stores and loads, or coerce the reexecution of a program to do the right thing [1][4][13].

Another approach is to work on programs that are running on top of a language virtual machine (such as a JVM) [5][8]. A VM offers great control over the target software and its interface to the world, as everything that is done is fundamentally interpreted by the VM.

We can also use virtual platforms or virtual machines, and run the target software stack including the operating system. This has been used with both run-time virtual machines like UML [9] and VmWare [17], and development-oriented full-system virtual platforms like Simics [11]. A virtual platform reexecutes at the hardware level, and the software including the operating is reversed as a result. It is in many ways the cleanest solution, but also one that is hard to implement and get to perform well.

4.5 Implementations

Reconstruction of state seems to have been first implemented in the *Instant Replay* system [3], but that was really a record-replay rather than true reverse debug system.

The *Bidirectional Debugger* presented 2000 [1] used Unix fork calls to create checkpoints, and instrumented system libraries and program code to facilitate reexecution. Single-threaded and user-level.

ReVirt from 2002 used user-mode Linux (UML) paravirtual machines to implement a reverse debugger for operating systems [9][10]. The OS needs to be modified to run on UML, but not to run with the debugger. The paper [10] nicely presents a number of examples for when reverse debugging really helps. The solution is level *E* in Figure 2, but only for a single processor.

Simics Hindsight from 2005 used a full-system virtual platform to implement reverse debugging [11]. System-level, multiprocessor and multi-machine, unmodified software stack. A unique aspect of Simics is that it can

reverse a system out to level *G* in Figure 2, and it was the first commercial multiprocessor reverse debugger.

UndoDB from 2006 was the first commercial user-level reverse debugger [12]. It runs on Linux, and can handle multiple threads (apparently by serializing their execution). It uses a novel time base, simulated nanoseconds, which increases by at least one between instructions, but does not correspond to any real time.

The Microsoft *iDNA* framework and the time travel debugger built on top of it was presented in 2006 [13][14]. It is a user-level, multiple thread solution that uses a combination of emulation and binary instrumentation to capture the data needed for replaying a program in a post-mortem debugger.

A small subset of the *Qemu* emulator was used to demonstrate reverse debugging in 2007 [15]. This work targeted the semihosting variant of Qemu, and thus was really user-level (not system-level), single-processor. Based on gdb, this debugger introduced the "change direction" user interface found in *gdb 7.0 reverse debugging*. All other reverse debugging solutions use the more natural and direct reverse (or backwards) step and run commands.

The RogueWave *TotalView* debugger added support for reverse debugging of multithreaded user-level programs in 2008, with *ReplayEngine* [16]. The debugger allows a user to step back in time, as well as run backwards to a certain line (which is a backwards execution breakpoint) and jump to a certain point in time. It requires heavy instrumentation of the runtime system used, and supports a few common HPC communications APIs such an OpenMPI.

VmWare Workstation added reverse debugging in 2008 [17]. System-level, single-processor solution that supported data breakpoints in revers. Based on recording the non-deterministic aspects of a single-processor VmWare machine, with limitations to which kinds of devices could be used. It noteworthy that VmWare dropped this support with VmWare Workstation 8 in 2011, presumably since the requirements of reverse execution interfered with the requirements for normal VM use cases.

5 FINAL REMARKS

This paper has offered a review of the current state and historical growth of reverse debugging technology. Over time, many different implementations have been tried, and some have made it all the way to users and the marketplace. The implementations differ in their scope, target systems, and capabilities, but all offer something better than traditional cyclic debugging. The space of this paper has only allowed a broad overview of the field, and the reader is encouraged to visit the referenced sources for more information.

6 REFERENCES

[1] Boothe, B, "Efficient Algorithms for Bidirectional Debugging", *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI)*, pp. 299-310, May 2000.

[2] Kidder, T, *The Soul of a New Machine*, Hachette Book Group USA, NY, 1981.

[3] LeBlanc, T, and Mellor-Crummey, J, "Debugging Parallel Programs with Instant Replay", *IEEE Transactions on Computers*, pp. 471-482, Volume 36, Issue 4, April 1987

[4] Russinovich, M, and Cogswell, B, "Replay for concurrent non-deterministic shared-memory applications", *Proc. ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI)*, pp. 258-266, June 1996.

[5] Huff, Ian, "IntelliTrace in Visual Studion 2010 Ultimate", *MSDN Blogs*, May 13, 2009 (http://blogs.msdn.com/b/ianhu/archive/2009/05/13/historical-debugging-in-visual-studio-team-system-2010.aspx)

[6] Lindahl, M, "The Device Software Engineer's Best Friend", *IEEE Computer*, May 2006.

[7] Lauterbach Context Tracking Systems, http://www.lauterbach.com/cts.html, 2005.

[8] Lewis, B, and Ducasse, M, "Using Events to Debug Java Programs Backwards in Time", *Proc. of the ACM SIGPLAN 2003 Conference on Object-oriented programming, systems, languages, and applications (OOPSLA)*, pp. 96-97, October 2003.

[9] Dunlap, G, King, S, Cinar, S, Bazrai, M, and Chen, P, ""ReVirt: enabling intrusion analysis through virtual-machine logging and replay", *Proc. of the 5th symposium on Operating systems design and implementation (OSDI)*, pp. 211-224, 2002.

[10] King, S, Dunlap, G, and Chen, P, "Debugging Operating Systems with Time-Traveling Virtual Machines", *Proceedings of USENIX 2005 Annual Technical Conference*, pp. 1-15, 2005.

[11] Engblom, J, Aarno, D, and Werner, B, "Full-System Simulation from Embedded to High-Performance Systems", in *Processor and System-on-Chip Simulation*, Leupers, Rainer and Temam, Olivier (eds), pp. 25-45, Springer Verlag, 2010.

[12] UndoDB Man Page, http://undo-software.com/product/undodb-man-page, visited July 2012.

[13] Bhansali, S, et. al, "Framework for Instruction-level Tracing and Analysis of Program Executions", *Proc. of the 2nd International Conference on Virtual Execution Environments (VEE)*, ACM Press, June 2006.

[14] Narayanasamy, S, et. al, "Automatically Classifying Benign and Harmful Data Races Using Replay Analysis", *Proc. ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI)*, June 2007.

[15] Jacobowitz, D, and Brook, P, "Reversible Debugging", *GCC Developer's Summit* 2007.

[16] Gottbrath, C, "Reverse Debugging with the TotalView Debugger", *Cray User Group Conference*, Helsinki, Finland, May 2009.

[17] Lewis, E, "VMware Workstation 6.5: Reverse and Replay Debugging is Here!" (http://www.replaydebugging.com/2008/08/vmware-workstation-65-reverse-and.html), August 2008