



REVERSIBLE EXECUTION AND DEBUGGING

JUNE 2007

INFO@VIRTUTECH.COM

WWW.VIRTUTECH.COM

INTRODUCTION

It is not possible to run a computer backwards. So when debugging a program, whenever a point of interest has passed, it is gone. The only way to go back is to restart the program from the beginning. This makes debugging very inefficient, since the programmer is investing a lot of effort each time for small amounts of incremental information.



Imagine if the same rules operated in other areas. You are trying to find your way in an unfamiliar city where you are penalized for a wrong-turn by having to leave the city and start again at the city boundary; or you are writing a long document in a word processor, and you make a spelling error and you have to start from the beginning; or you are playing a new single-person video game. Each time you are eliminated, instead of being able to go back to the most recent level, you have to restart the whole game from the beginning, splash screens and all.

But debugging programs has always been like this. Until now. With the reversible execution and debugging features of Simics it is now possible to execute a program in reverse and step backwards through the code.

SIMICS

Simics is the first general-purpose development tool for reversible execution of arbitrary software running on arbitrary systems.

When running software on a computer, once a particular instruction has been performed, it is impossible to go backwards through the code. If it turns out a bug resulted from an error earlier in the program (which is frequently the case), there's no going back. Every programmer has had the experience of stepping over a procedure call without looking into the details, only to discover later that the error originated within that procedure. Or wondering "how did that pointer become zero?" In each case it is necessary to restart the program, and run it to just before problem occurred. That might take a few seconds, or several hours. Or it simply isn't possible – in a parallel or networked system, recreating a particular event can be almost arbitrarily time consuming.

With Simic, it is now possible to step back before *any event* to see what happened. And Simics retains the whole timeline, allowing the programmer to go back-and-forth across any subset of the software to zero in on the problem. With each sweep back-and-forth, new breakpoints, watchpoints, and other instrumentation can be incrementally added until the problem is fully understood.

Since computers themselves do not actually run backwards, it is not possible to provide reversible execution with code running directly on hardware. Instead, a virtual model of the hardware is built that is accurate enough such that the software under development cannot tell the difference. The actual binary code runs unchanged, even for real-time operating systems, device drivers, firmware, network protocol stacks, etc. In addition to having this level of fidelity, the model also runs fast enough that software engineers prefer using it as part of their edit-compile-debug loop.

Classical debugging commands all have an analogous reverse command. For example, all debuggers include a *continue* command that runs the program (forwards) until either a breakpoint is reached or the program terminates. Simics has a corresponding *reverse continue* command which runs the program backwards until either a breakpoint is reached or the program starts. Simics can even *unboot* an operating system, running the code backwards until it reaches the initial hardware reset or power-on state.

Simics has extremely high performance when executing code forwards. Reversible execution would be an interesting academic curiosity if it only executed in reverse at glacially slow speeds, but Simics executes code in reverse almost as fast as it executes code forwards. Which is to say that it has very high performance in reverse too.

Without Simics, a programmer has to struggle to stop the program just before something bad happens, such as detecting corruption of a data structure. It can be extremely difficult to define a breakpoint if a record is added wrongly after the first million correctly added records. The breakpoint would have to be defined to ignore the first million times through the code and stop on just the correct occurrence. With Simics, it is possible to execute the code until the



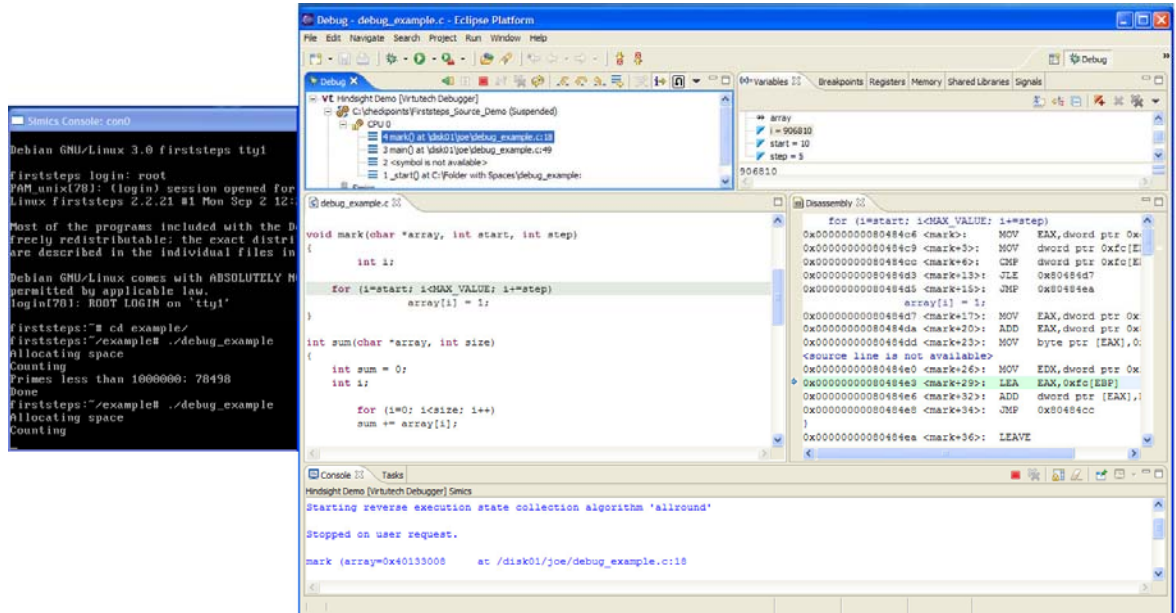
problem is detected, which is typically obvious, and then run in reverse until the root cause is determined, which is almost certainly associated with the record just added. In most cases, fixing the problem once it is identified is straightforward.

Simics is used to debug a program in much the same way as a normal debugger. The program under test is launched in debug mode, the programmer sets breakpoints at interesting places if necessary and the program starts. At some point, a breakpoint is reached, an assertion in the code fails indicating trouble, or the underlying hardware detects a problem such as division by zero. Now, instead of having to rerun the program with additional breakpoints, Simics permits new breakpoints to be added and the program run backwards. If the problem seems to have occurred just before it was detected, the code can be reverse-single-stepped.

EXAMPLE

Debugging using Simics can be performed in a much more direct way than with traditional approaches. Here is an example of a tricky problem.

A kernel mode device driver for a network interface contains a bug. The bug is in the code that handles the packets that are received with incorrect checksums. As part of the processing, a zero is written to the wrong location. Since the driver is running in kernel mode, in this case it overwrites a system data structure. Packets with bad checksums are rare, so the system usually runs correctly under normal operation. Even when bad packets are received, the errant zero might have no effect if it is written to an area that is not in use. But if the zero is written to an active data structure, the system may eventually crash when the overwritten value is accessed and a zero acquired instead.



Suddenly the system halts having attempted to access address zero.

A cursory examination of where the system halted makes it clear that an important data structure has been overwritten. In principle, the problem could be almost anywhere in the kernel, in an interrupt routine, in a device driver or any other code for which access to kernel memory is allowed.

With a traditional debugger, it is almost impossible to debug this sort of kernel problem. Even with a kernel debugger, the only thing to be done is to put a watchpoint on the location and re-boot the system. But this will completely alter the timing of the system and it is unknown when the next bad checksum packet will arrive. The symptoms this time around will be completely different. If bad checksum packets occur sufficiently rarely, the system will just appear to be unstable, crashing occasionally for an unknown reason. Time taken to find the bug: weeks, or perhaps even years.

With Simics it is possible to set a watchpoint on the errant zero. Execution is then reversed until the watchpoint is hit. When execution stops at the watchpoint, it stops in the network device driver bad checksum handling routine. With that bit of information it is probably trivial to spot the error in the code and fix it. Time taken to find the bug: a few minutes.

HOW IT WORKS

At first glance, Simics seems close to magic. Since real computers don't actually run backwards, the only way it seems that Simics could be implemented is to save the old values of every altered register or memory location so that each instruction can be reversed simply by restoring the obliterated values. This would have two serious problems: the amount of data that would be generated and the execution time required to handle the storage. The simulation would be slowed down to unacceptable speeds running forwards and have very poor performance running backwards. This approach still leaves the issue of how to handle devices such as disks, network interfaces and graphics screens. Simics suffers from none of these deficiencies.

An idea whose time has come

It was clear from the onset of the computer revolution that program debugging would be a fundamental challenge.

The very first paper on debugging by Gill, dating from 1950, defines both the *trace* (output of events of interest) and *post mortem dump* (or what we today would call a *core dump*, a “dump” of the raw state of the computer at the time of an error).

Debugging today and in the 1950s differ surprisingly little. Traces and core dumps are still of central importance. Of course, core dumps have become interactive, in that a program (a debugger) can now freeze and inspect another program (the one being debugged), either on the same system or on a remote one.

The obvious potential in reversible execution for the problem of debugging was clear at least by the late 1960s.

Notably: “What is obviously needed is some method to foresee the occurrence of an error and take some diagnostic steps accordingly. Since this is obviously not possible, then the next best method would be to backup execution of the program to the point of the error after it is noticed and to re-execute the program under control of some diagnostic routine” *Zelkowitz, 1971*

That a solution could be had from a combination of tracing and core dumping was evident from the start. Ever since then, various attempts at achieving this goal have been tried. Except for very limited applications, these efforts were unsuccessful.

Simics already has a checkpoint and restore capability, which allows the state of the whole system to be saved and the simulation to be re-started from the checkpoint. Simics builds on these capabilities. An obvious naïve implementation would checkpoint the simulation after every instruction. Reversible execution could be implemented by restoring saved checkpoints one after another. The performance would be appallingly slow and the amount of space required to store the checkpoints would be prohibitive.

However, interactive debugging is, by definition, interactive. Simics executes millions of instructions almost instantaneously so that the processing is not perceptible to a user. This speed allows a more practical implementation.

At regular intervals, Simics takes checkpoints. To implement a reverse single-step, Simics restores a previous checkpoint and then runs forward one instruction shy of the current program counter. Since Simics is so fast, the effect is that a reverse single step appears instantaneous, like a forward single step.

To make a fully practical solution, there are issues with peripheral devices, preserving determinism, supporting distributed simulation, supporting multiple processors, multiple clock domains, managing breakpoints and watchpoints,

handling large I/O operations, and so on. Efficiently handling all these challenges is where various patented secret sauces come into play. As with many things, the idea seems simple but the devil is in the details.

INCREASED PRODUCTIVITY

By enabling a developer to reverse execute code, Simics is vastly increasing the productivity of software engineers. More than 50 percent of a programmer's time is spent debugging, but there have not been any notable new technologies since source-code debugging became available in the late 1970s. It is too early to have quantitative data on how much time Simics will save, but veteran programmers, upon seeing a demonstration, immediately note that they could have literally saved years of their life if the technology had been available throughout their careers.

Since over half of a programmer's time or more is spent debugging and testing, and if makes debugging twice as efficient, aggregated across a large software organization, the benefits of Simics would result in a 25 percent cost reduction of the operational expenses, plus additional benefits that accrue from decreased time-to-market. Simics will make a very real impact on the cost of software development.

ALL DEBUGGING WILL BE DONE THIS WAY

As electronic systems become more powerful and mature, we expect all software debugging will be supported by an underlying virtualization layer that is necessary to provide the power of reversible execution and debugging. Simics will be a major driving force behind the adoption of virtualized software development as opposed to older, hardware-based approaches which are becoming obsolete for complex systems.

In the longer term, all debugging will come to be done using this approach: reversible execution as a result of an underlying virtualization layer. Debugging with only forward execution will become as obsolete as assembly language debugging once source-level debuggers were available.

Finally, a reverse gear for programmers.



CONTACT INFORMATION

Corporate Headquarters	Virtutech, Inc. 1740 Technology Drive #460 San Jose, CA 95110 Tel: 408- 392-9150 Fax: 408- 608-0430
European Sales and Support Office	Virtutech AB Drottningholmsvägen 14, 3tr SE-112 42 STOCKHOLM SWEDEN tel +46 8 690 07 20 fax +46 8 690 07 29
Internet	info@virtutech.com http://www.virtutech.com

© Copyright 2005, 2007 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics,, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 1740 Technology Drive #460, San Jose, CA 95110, USA