

FIXING AN INTERMITTENT MULTICORE BUG WITH SIMICS

NOVEMBER 2009

JAKOB ENGBLOM

WWW.VIRTUTECH.COM

INTRODUCTION

Virtualized Systems Development[™] (VSD) is a development methodology where the actual hardware of a system is replaced with a virtual platform running on a workstation or PC. The virtual platform can run the same binary software as the physical hardware and is fast enough to be used as an alternative to physical hardware for software development.

The virtual platform provides additional benefits to the user compared to physical hardware. For example, the virtual platform offers superior convenience and stability, full insight into the system execution, and better debugging facilities. It provides access to the target system long before prototype hardware is available. The virtual platform supports fault injection and testing with multiple configurations. It also provides checkpoint and restart facilities, allowing system states and bugs to be captured. For multicore, multithreaded, and distributed systems, virtual platforms provide repeatable deterministic replay of any execution, making it much easier to fix intermittent and “random” bugs.

This paper will go through a concrete case-study in how an intermittent multicore multithreaded bug was found, reproduced, and analyzed using Simics. It shows how to use scripting, OS awareness, checkpoints, determinism in concert to do something that is not possible using physical hardware.

THE BUG

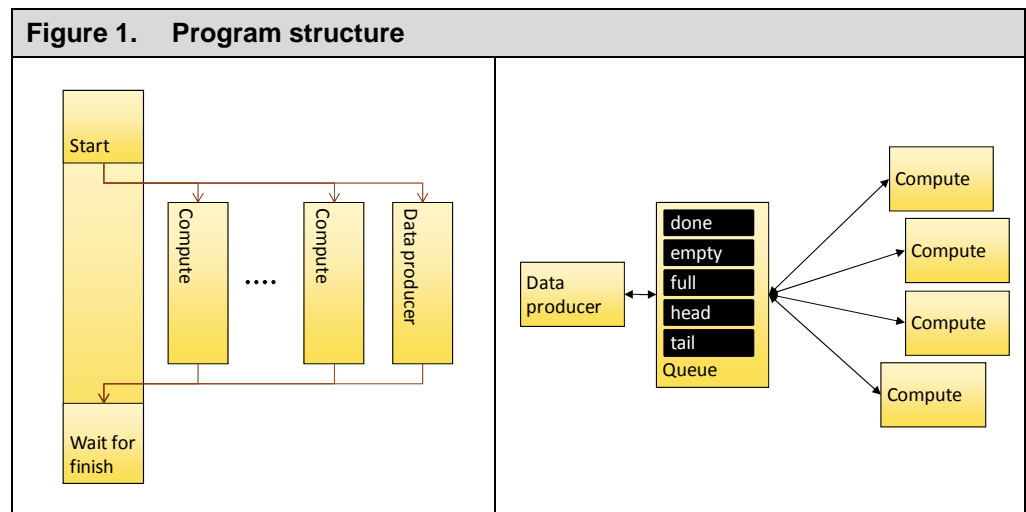
The program that failed is a test program we are using to test and demonstrate how to test multicore scalability. It is a multithreaded program based on pthreads. When it broke, the program had worked very reliably for more than a year on a model of the Freescale MPC8641 target machine, using a 2.6.23 Linux kernel. It was run with 1 to 8 processor cores in the hardware and 1 to 16 threads, and with large variations in its other input parameters and input data. It never crashed, hung, or misbehaved in other ways.

However, in the Fall of 2009, the software stack on which it was running was upgraded to contain a newer Linux kernel. The program was also run on some other target platforms, such as the Freescale QorIQ P4080. Suddenly, we started to see random hangs, where the program would not terminate cleanly but have to be quit from the target Linux command-line.

Apparently, changing the underlying Linux version changed something, causing a latent issue in the program to manifest itself as a bug. This is a common pattern for parallelism bugs: they only manifest themselves as actual visible crashes or freezes or bad computation results once something in the software stack has changed, even though the fundamental issues have been there all the time. In this case, it was the Linux scheduler or pthreads implementation. It shows that just because a program runs fine today it does not have to run fine tomorrow, on a different machine, OS, or in a different physical settings.

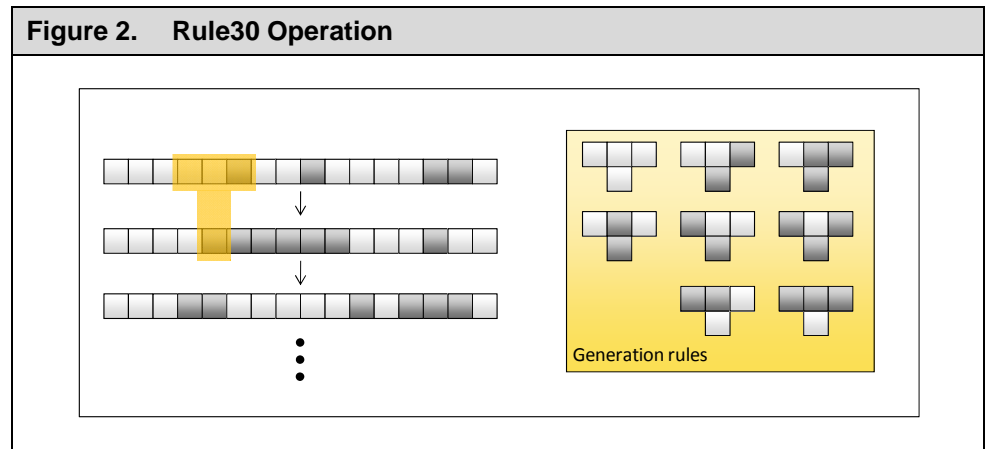
THE PROGRAM

Some notes about the program itself. It is a producer-consumer system programming in C using pthreads, with a single producer thread feeding a variable number of compute threads with data, over a shared queue structure. As shown on the left in Figure 1, the threads are started from a main thread which also waits for all threads to terminate before exiting the program

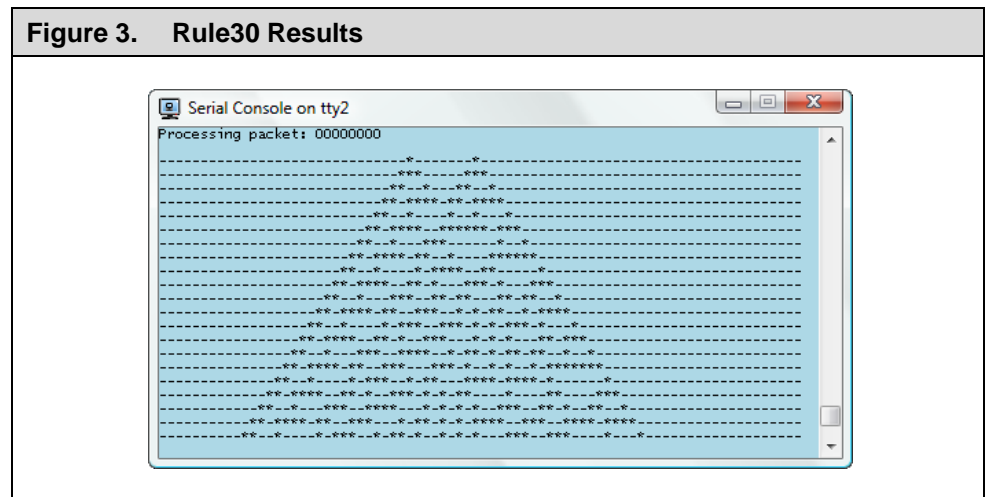


The crucial data structure for this program is the queue, which contains a circular buffer. As shown on the right in Figure 1, the circular buffer is managed using a standard set of `full/empty/tail/head` variables. There is also a variable called `done` which is used as a flag, and set to one when it is time to terminate the compute threads. Note that since this program is used to demonstrate and test scaling, it does actually terminate as a natural part of its execution.

The computation performed is the cellular automaton known as *rule30*. This is an algorithm that computes one line of data at a time, generating very complex and unpredictable patterns from simple rules. As illustrated in Figure 2, each line consists of a number of binary elements, and the value of each element in a line is based on the value of the three elements above, to the left, and to the right of it.



The result of running this algorithm from an input starting with a few bits set is a characteristic “Christmas tree”, as illustrated in Figure 3.



The nature of the computation is irrelevant for the bug in question, but it is important to note that the computation has several parameters guiding how the amount of computation performed for each packet:

- The length of the line, which also the size of the packet processed by each compute thread (the threads are sent a starting line, and then compute successive generations from it).
- The number of generations to compute for each packet.
- The amount of diagnostic output printed on the screen as the program runs.
- The number of packets to process.

THE CHASE

Provocation

The chase of the bug began with a quest to provoke it, which is a precondition for reproducing it. Since we had already seen the bug on the simulator, this was really easy. It was almost a simple matter of rerunning the test cases where we observed the hang – but not quite.

In the existing setups, the simulator would load the target program dynamically from the host machine using the simicsfs pseudo file system. This introduces non-determinism in the execution, and did effect when exactly the bug hit. To avoid this, a checkpoint was prepared with the target machine booted and the target program loaded.

Starting from this checkpoint, scripting to run the program was applied. At this point, we worked like a test department would do: find a precise way to reproduce a bug to hand it back to engineering for analysis and fixing.

The hung executions seemed to happen randomly with no real pattern to them. They definitely did not depend on the program parameters, apart from output. With sufficient volumes of text output to the serial console, the program always terminated cleanly (offering a demonstration of the well-known fact that changing the timing of a program by logging tends to hide bugs – and sometimes cause new bugs to appear, obviously).

Proving the independence from the parameters, running the program with the same parameters for a few times in a row would result in a hung execution only once in a while. Using control-C to quit it and restart would typically let the next run of the program terminated cleanly.

The final script used is shown in Figure 4, which is the complete Simics script needed to open the checkpoint and run the program a number of times on the target machine. Note that we did not bother to minimize the test case to just show the hang, it was enough to capture a case where it happened after a few runs of the target program (in this case, on a 2.6.27 Linux, the hang happened when using five threads, so counts 6, 7, and 8 are really superfluous).

Figure 4. Bug Provocation Script

```
read-configuration "%script%/buggy-rule30-program-setup.ckpt"

$sys = argox8
$con = ($sys.console0.con)
$thread_counts = [1,2,3,4,5,6,7,8]
$packet_count = 200

script-branch {
  local $con = $con
  local $prompt = "# "
  $con.input "\n"

  foreach $t in $thread_counts {
    $cmd = ("./rule30_threaded.elf software_byte %d 50 100 0 %d\n" % [$packet_count, $t])
    $con.wait-for-string $prompt
    echo "* Testing with " + $t + " threads."
    echo ""
    $con.input $cmd
  }
}
```

Reproduction

Given the work in provoking the bug, reproducing it was trivial. Just run the script in Figure 4, from the prepared checkpoint, and the bug hits. Communicating a bug like this from a testing department to an engineering department is as easy as taking the checkpoint and script, and sending them in an email.

In our actual case, the checkpoint also contains two more machines in a network. These probably have nothing to do with the hang, but why take a chance and spend testing effort to remove them? It is much simpler to leave them in there, capturing the precise system setup where the bug manifests itself in the first place.

Analysis

The final step in any debugging is to analyze the problem at hand. This is covered in more detail in the next section.

ANALYZING THE BUG

To diagnose the problem, we wrote some Simics script code that watched the running program and created a high-level semantic trace. Our guess was that the issue had something to do with the queue data structure, as that is where the compute threads receive their signal to terminate. Thus, the script had to detect all changes to the queue, and print the state of the queue after the change.

A short note on debugging and Simics: Simics contains a built-in *debugger* system that handles symbolic information, and which is available from the Simics command-line interface (CLI) and Python interpreter. Simics can set *breakpoints* on writes to, reads from, and the execution of code at any location in memory. Simics has a system for *OS awareness* where code running in the simulator determines the active processes and threads on the virtual platform, and can notify scripts when threads are switched in and out. Obviously, you can read and write any value in memory at any time. This makes it possible to write programs that perform debugging in a completely automated fashion, with no manual intervention needed other than the interpretation of results.

Setting up breakpoints on modification of the queue was not as easy as putting breakpoints on functions, however. The key functions all used a mutex to protect access to the shared queue data structure, and we needed to get at the threads when they actually modified the queue. The function called by compute functions to retrieve work is shown in Figure 5, and since we assume that the program is well-behaved in its use of concurrency primitives, we can use the structure of this function to write the instrumentation code. The code to add data is very similar in structure. Essentially, by looking at writes to the `full` and `empty` variables in the queue, we can be sure to intercept state changes within the lock.

Had the program been written in a different way, with designated functions to manipulate the queue separate from the locking logic, this would have been easier, but changing the program to help debugging it does not really fit in the idea of non-intrusive debugging of a rare hang.

Figure 5. The Queue Get Function

```
int rule30_packet_queue_get (rule30_packet_queue_t *queue,
                             rule30_packet_t      *packet_buffer) {
    int returnvalue = 0;
    pthread_mutex_lock (&(queue->mutex));
    while ((queue->empty) && !(queue->done)) {
        pthread_cond_wait (&(queue->notEmpty), &(queue->mutex));
    }

    //
    // We are through. Check state of the queue. First, check for work
    // Then, check for done
    //
    if(queue->empty) {
        if(queue->done) {
            returnvalue = 0;
        } else {
            returnvalue = -1;
        }
    } else {
        memcpy(packet_buffer, &((queue->buf[queue->head]).data), sizeof(rule30_packet_t));

        // Manipulate queue status
        queue->head++;
        if (queue->head == queue->size)
            queue->head = 0;
        if (queue->head == queue->tail)
            queue->empty = 1;
        queue->full = 0;

        returnvalue = 1; // One packet retrieved
    } // if empty

    //
    // Unlock the queue, signal that it is not full anymore
    //
    pthread_mutex_unlock (&(queue->mutex));
    pthread_cond_signal (&(queue->notFull));

    return returnvalue;
}
```

However, even this interception of writes is not as easy as it sounds, as the queue is not a global variable and thus is not available as a symbol in the debugger. Instead, it is dynamically allocated on the heap, and the queue pointer passed around between all functions using the queue, never storing the pointer in a global variable (a good computer science graduate never uses a global variable other than as the means of last resort).

To solve this, the analysis script sets a breakpoint on the line in the setup function that came after the allocation. With the program stopped at that point, the script can read the value of the local variable pointing to the queue,

and determine the addresses of all the interesting member variables in the structure.

Figure 6. Breakpoint Intercepting the Packet Allocation

```
# Simics CLI script code
#
# $ctx is a breakpoint context for the program
# $st is a symbol table holding the debug information
#
$mbp = ($ctx.break ($st.pos (rule30_threaded.c:222)))
$cpu = (wait-for-breakpoint $mbp)
$pq_addr = ($cpu.sym "pq")
#
# Determine addresses of relevant member variables
#
$pq_tail = ($cpu.sym "&(pq->tail)")
$pq_empty = ($cpu.sym "&(pq->empty)")
$pq_full = ($cpu.sym "&(pq->full)")
$pq_head = ($cpu.sym "&(pq->head)")
$pq_done = ($cpu.sym "&(pq->done)")
#
# Set data-access breakpoints
#
$dbp = ($ctx.break -w $pq_done)
$drbp = ($ctx.break -r $pq_done)
$ebp = ($ctx.break -w $pq_empty)
$fbp = ($ctx.break -w $pq_full)
```

The Simics script code in Figure 6 achieves this, putting a breakpoint right after the last line of the code shown in Figure 7. When the breakpoint hits, we read the value of the variable `pq`, as well as determine the addresses of its member variables using a gdb-like symbol-lookup syntax.

Figure 7. Code Intercepted by Figure 6

```
void start_threads( ... ) {
    //
    // Allocate a packet queue to feed work to worker
    // threads.
    //
    pq = rule30_packet_queue_init( QUEUESIZE, "generator to workers queue");
    // Breakpoint hits here, after pq is given a value from the init call
```

Given this set of breakpoints, a Python script was written that put callbacks on all breakpoint triggers, and performed an analysis of the queue state. On each write to a variable, it dumped the state of the queue including computing the number of elements in the circular buffer (without having to run any code on the target). To get an idea for who was active, it also used

OS awareness to find the currently executing thread ID, and symbol lookup debugging to convert the current program counter into a position in the program source code (actually, the important issue was the name of the function we were executing in).

Figure 8. Python Queue Analysis Code

```
def general_breakpoint_handle(target,user_arg,context,bpno,memop):
#
# Get active cpu, thread, value to be written, etc
#
cpu = SIM_get_mem_op_initiator(memop)
tid = process_tracker.iface.tracker.active_trackee(cpu)
pc = cpu.iface.processor_info.get_program_counter()
now = cpu.iface.cycle.get_cycle_count(cpu)
value = SIM_get_mem_op_value_cpu(memop)
(file,line,func) = context.symtable.source_at[pc]
#
# Read values of all variables
#
done = read_32b_variable(pq_done)
empty = read_32b_variable(pq_empty)
full = read_32b_variable(pq_full)
head = read_32b_variable(pq_head)
tail = read_32b_variable(pq_tail)
# We know the buffer is 100 elements in this program
if(tail>head):
    #
    elems = tail - head
elif(tail<head):
    # tail has wrapped
    elems = tail + (100-head)
else:
    # equality, depends on empty/full flags
    elems = 0
    if empty==1:
        elems = 0
    elif full==1:
        elems = 100
    # Special case: we are writing a "1" to full, which is a transition
    elif (target=="full") and (value==1) :
        elems = -100 # mark transition
#
# Print state
#
if (SIM_mem_op_is_write(memop)):
    op = "writing"
else:
    op = "reading"
    value = done
print "[bp] Thread %5d, %s variable %s with value %d." % (tid,op,target,value)
print "    At %s, line %d " % (func,line)
print "    Prev. state: Done: %2d Empty: %2d Full: %2d Tail: %2d Head: %2d Elems:
%2d" % (done,empty,full,tail,head,elems)
```

The code to implement this was written in Python, as that is more suitable than the Simics CLI for complicated scripting and callback handling. It could

have been written in C or C++ as a custom Simics module, but that takes much more time to write than using Python. The core trace logic is shown in Figure 8. The function `general_breakpoint_handle` is called from the actual handlers, which add information on which variable was hit (the argument `target`). A Simics memory breakpoint intercepts the operation *before* the write completes, and this we read the old values of the variables from memory. However, we know the new value being written from the memory operation.

An example of the information collected is shown in the trace excerpt in Figure 9.

Figure 9. Trace Excerpt

```
...  
[bp] Thread 921, writing variable empty with value 0.  
At rule30_packet_queue_add, line 103  
Prev. state: Done: 0 Empty: 1 Full: 0 Tail: 1 Head: 0 Elms: 1  
[bp] Thread 920, writing variable empty with value 1.  
At rule30_packet_queue_get, line 157  
Prev. state: Done: 0 Empty: 0 Full: 0 Tail: 1 Head: 1 Elms: 0  
...
```

This trace indicated that the queue rarely held more than a few elements. Out of curiosity, we also ran the same script on the original Linux 2.6.23-based platform, and there, the queue was almost always full. It looks as if the 2.6.23 Linux gives the producer thread priority for the mutex, allowing it to come in and deposit new data as soon as the compute threads had consumed a few items. On the 2.6.27 Linux, the opposite is true, with the producer thread running behind and inserting a few items before the consumers greedily grab them and compute on them.

Clearly, the Linux kernel can exhibit quite different behavior in different versions. I guess that's why this is called "soft real time"... There is an important parallel programming lesson here: the scheduler is very important, and a smart adaptive scheduler can wreak havoc with software that was accidentally tuned for a different scheduler.

Back on the 2.6.27 Linux and the hung execution, it turned out to be quite fortunate that we had a reproducible test case containing both working and non-working runs of the program. This let us compare the traces.

The most interesting behavior noted was that when the program hung, the done flag was set with a queue that was empty. Looking at the code in Figure 10, it did seem to have this case covered, checking specifically for the case that done was set with an empty queue. In such a case, it signals the conditional variable notEmpty, which we can see the loop in Figure 5 waiting on in case the queue is empty and done is not set.

Figure 10. Function to set the Done Flag

```
void rule30_packet_queue_signal_done(rule30_packet_queue_t *q) {
    //
    // Grab lock, set the done signal atomically
    //
    pthread_mutex_lock (&(q->mutex));
    q->done = 1;
    pthread_mutex_unlock (&(q->mutex));
    //
    // Signal any threads waiting for data to wake up
    // and discover that we are indeed done
    //
    pthread_cond_signal (&(q->notEmpty));
}
```

To zero in on this issue, we extended the debug script to also detect the case of done being set to one, and starting to also follow all reads from the variable. Looking at the reads is necessary to see which threads get the signal to terminate. However, looking at all reads from done during the entire program execution will generate additional trace points of no value, so we made this work conditional, as shown in Figure 11.

It should be noted that the execution semantics of Simics guarantees that the entire script code will be run atomically without the target machine moving at all. Essentially, the script is called between steps in the execution of the memory-access instruction, and Simics only resumes execution of the target system once the script has finished. Thus, there is no race between the script and the target system, and the target system is completely unaffected by the presence, absence, or nature of the inspection script.

Figure 11. Conditional Tracking of Reads from the Done Flag

```
done_write_seen = False
def done_bp_callback(user_arg, context, bpno, memop):
    print ""
    print "    - Termination: Write to 'done' flag seen - "
    print "    - turning on tracing of reads from it    - "
    global done_write_seen
    done_write_seen = True
    general_breakpoint_handle("done", user_arg, context, bpno, memop)

def done_read_bp_callback(user_arg, context, bpno, memop):
    if(done_write_seen):
        print ""
        print "    - Termination: Read from 'done' flag seen - "
        general_breakpoint_handle("done", user_arg, context, bpno, memop)
    else:
        # Do nothing
        pass
```

Using this script, we get to a final analysis result where we note something quite interesting. In the case with five threads that hang, three out of five compute threads actually read the `done` flag and terminate. A simplistic analysis would assume that all threads are hanging on the `notEmpty` conditional variable in `rule30_packet_queue_get`, and that only one thread gets released when it is signaled, which is not the case. To hang, a thread actually has to be inside the conditional wait there, but there is nothing that makes that necessary.

For example, threads can be waiting to grab the initial mutex lock at the start of the `rule30_packet_queue_get` function, or they might be doing actual compute work on a packet they retrieved before the `done` flag was set. Adding lots of debug printouts of the code increases the likelihood of threads being somewhere else than in the waiting loop, and thus it is logical that lots of debug printouts hides the error. It certainly serves to illustrate just how chaotic parallel programs can be.

The trace of the program when it hangs is shown in Figure 12.

Figure 12. Final Trace Output from Hung Execution

```
...
[bp] Thread 923, writing variable full with value 0.
    At rule30_packet_queue_get, line 158
    Prev. state: Done: 0 Empty: 1 Full: 0 Tail: 0 Head: 0 Elems: 0

    - Termination: Write to 'done' flag seen -
    - turning on tracing of reads from it -
[bp] Thread 928, writing variable done with value 1.
    At rule30_packet_queue_signal_done, line 62
    Prev. state: Done: 0 Empty: 1 Full: 0 Tail: 0 Head: 0 Elems: 0

    - Termination: Read from 'done' flag seen -
[bp] Thread 926, reading variable done with value 1.
    At rule30_packet_queue_get, line 125
    Prev. state: Done: 1 Empty: 1 Full: 0 Tail: 0 Head: 0 Elems: 0

    - Termination: Read from 'done' flag seen -
[bp] Thread 923, reading variable done with value 1.
    At rule30_packet_queue_get, line 125
    Prev. state: Done: 1 Empty: 1 Full: 0 Tail: 0 Head: 0 Elems: 0

    - Termination: Read from 'done' flag seen -
[bp] Thread 927, reading variable done with value 1.
    At rule30_packet_queue_get, line 125
    Prev. state: Done: 1 Empty: 1 Full: 0 Tail: 0 Head: 0 Elems: 0

    ... and then nothing more ...
```

In the end, the solution to the problem is to use a `pthread_cond_broadcast` call rather than `pthread_cond_signal`. In retrospect, it seems strange that this ever worked reliably... but it did. It only requires some luck with the Linux scheduler, it seems.

LESSONS LEARNED

There are some general lessons to be learned from this experience.

About parallel software in general:

- Writing parallel software in C using a raw API like pthreads is hard, and it is easy to make mistakes.
- Mistakes can often be masked by other behaviors in the system, and only manifest themselves in rare circumstances or when something apart from a program itself changes.
- Parallel software should be tested on a range of machines, both considering hardware characteristics like core counts and speeds, and software characteristics such as operating system versions and compiler versions.
- Good instrumentation code is program and problem specific.

About virtual platforms and debugging parallel software:

- Once a bug has been successfully provoked on a virtual platform, it is trivial to reproduce.
- Using a virtual platform makes it very simple to capture and communicate even intermittent bugs between different groups or departments.
- Programmable debug features in a virtual platform makes it possible and fairly easy to write custom bug analysis and tracing code.
- Using scripting as the basis for debugging saves time as it makes reproduction of any particular setup trivial.
- A virtual platform allows an arbitrary amount of analysis code to be run in response to a breakpoint, without changing the behavior of the target system.
- You want to use a scripting-style language like Python to write custom scripts and inspection functionality. Writing the same code in C or C++ would take much more time and result in less flexible code.

The power of scripting in a debugger is well-known, and debuggers like gdb, adb, and Sniff have supported scripting for a very long time. Adding the control offered by a virtual platform, and you have a very powerful debug system for complex software.

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	Europe, Middle-East, Africa sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	Japan sales_japan@virtutech.com
China sales_china@virtutech.com	
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, 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., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA