DEBUGGING OPERATING SYSTEMS WITH TIME-TRAVELING VIRTUAL MACHINES (TTVM)

Samuel T. King, George W. Dunlap, and Peter M. Chen
University of Michigan

Presented by: Zhiyong (Ricky) Cheng
Happy Halloween!
Outline

- Background
- Introduction
- Virtual Machine Model
- Time-traveling Virtual Machine
- TTVM-aware GDB
- Performance
- Conclusion
- Questions
Background

- The paper was published in USENIX 2005, and won the best paper reward.
- The name of research group was CoVirt. Now they focus on “Virtual-machine based security services.” Link:  
  http://www.eecs.umich.edu/virtual/
- The main author of the paper, Sam King, had graduated and now teaching “Hot topics in virtualization and security” course at UIUC.
Background (cont’d)

- Although this paper and the previous paper are come from the same research group, this paper is more like a “by-product”.
  - Cannot find source code;
  - Doesn’t match main research interests of the group.
  - Actually, it is the only paper related to OS debugging.
- I borrowed some slides from the original presentation:
  http://www.eecs.umich.edu/virtual/papers/kingo5_1.slides.ppt
Cyclic debugging

- Iterate, revisit previous states
  - Inspect state of the system at each point
Problems with cyclic debugging

- Long runs
  - Each iteration costly
- Non-determinism
  - Code might take different path each time bug executed
  - Bug may not be triggered at all
- Especially relevant for multithreaded apps, Operating Systems
- Debugging is a combination of detective work, guesswork, and systematic search.
Example: NULL pointer

- Detective work: Investigating crash point
- Guesswork: Caused by the NULL pointer?
- What should we do next?
Example: NULL pointer

- Set a conditional watchpoint (Systematic search)
  - ptr might change often
  - Long runs!
Example: NULL pointer

- Conditional watchpoint
  - Different code path, variable never set to NULL

- All these are trying to find the LAST modification
Solution: Debugging with time traveling virtual machine (TTVM)

- Running the OS inside a virtual machine (Virtual machine)
  - Debugging without perturbing its state.
- Able to navigate backward and forward arbitrarily through the execution history. (Time traveling)
  - Go back to the prior points;
  - Fast forward to the crash point;
  - Work in presentence of non-determinism.
  - No need to repeat the entire run. (Comparing to the cyclic debugging)
Debugging with time traveling virtual machine

- Provide what cyclic debugging trying to approx.

\[ \text{ptr} = \text{NULL!} \]
Debugging with time traveling virtual machines

- Reverse equivalent to any debugger motion function
  - Reverse step
  - Reverse breakpoint
  - Reverse watchpoint
- Implement using time travel to previous states
  - Must be identical to “buggy” run
  - Instruction level granularity
VIRTUAL MACHINE MODEL
Typical OS level debugging

- Requires two computers
- OS state and debugger state are in same protection domain
  - crashing OS may hang the debugger
Using virtual-machines for debugging

- Guest OS, operating system running inside virtual machine
- Debugger functions without any help from target OS
  - Works even when guest kernel corrupted
- Leverage convenient abstractions provided by VM – the interface of physical machine.
Notice the difference?
- Where is the debugging stub?
Guest OS and host OS

- Want guest OS to be similar to host OS so bugs are portable.
- Differences are not fundamental, result of VM platform we use (UMLinux):
  - 92% of code are architecture independent.
  - Architecture dependent code different between guest OS and host OS (8% total)
    - Low-level trap handling
    - MMU functionality
    - Device drivers (6% total)
Guest OS and host OS (cont’d)

- Use the same host driver in guest
- Trap and forward privileged instructions from guest (Discuss later)
  - IN/OUT instructions
  - Memory mapped I/O
  - Interrupts
  - DMA
- 98% of Linux code runs unmodified in User-Mode Linux. Therefore can be debugged.
TIME-TRAVELING VIRTUAL MACHINE

- Logging and replaying a VM – ReVirt
- Host device drivers in the guest OS
- Checkpointing
- Expected usage model
Logging and replaying a VM

- Based on the previous paper - ReVirt (Dunlap02).
- Re-executes any part of the prior run, instruction by instruction
- Re-creates all state at any prior point in the run
- Logs all sources of non-determinism
  - external input (keyboard, mouse, network card, clock)
  - interrupt point
- Researchers are investigating ways to support replay on multiprocessors. (Lots of questions)
  - Reference 29: A Flight Data Recorder for Enabling Full-system Multiprocessor Deterministic Replay
Host device drivers in guest OS

- Problem: VMM only export a limited set of virtual devices to guest OS.
  - Good for general purposes because it frees guest OSs from needing device drivers.
  - Bad for debugging OSs because cannot work on real device drivers.

- Two approaches to solve the problem:
  - Provide software emulators for the real devices.
  - Extend Revirt and modify VMM.
Host device drivers in guest OS (cont’d)

- Software emulator approach:
  - Device driver issues I/O instruction to VMM.
  - VMM traps instructions and forward to emulator.
  - Must find an accurate software emulator.

- Second approach:
  - Extend ReVirt to log and replay the data returned by IN instructions, memory-mapped I/O instructions and DMA memory loads.
  - VMM must be modified to support x86 IN/OUT instructions and memory-mapped I/O and DMA.
  - Extra traps and logging operations.
  - Possible to corrupt host’s physical memory.
Checkpointing

- Periodic checkpoints for coarse grained time travel.
- Save complete copy of virtual-machine state: simple but inefficient. (Like full backup)
  - CPU registers
  - virtual machine’s physical memory image
  - virtual machine’s disk image
- Instead, use copy-on-write and undo/redo logging. (Like differential backup)
  - Technique applied both to memory and disk
How to time travel backward

checkpoint 1

Write A

redo log

undo log

A
B
B
Adding and deleting checkpoints

- Imagine that checkpoints are nodes of doubly linked list and linked by undo/redo logs.
- Then adding and deleting checkpoints is just same as adding/deleting a node to/from the doubly linked list.

*So far, TTVM is ReVirt + checkpointing*
Expected usage model

- In phase 1, the programmer runs a test to trigger an error. This phase may last a long time.
- In phase 2, the programmer attaches the debugger, switches the system from logging to replay, and prepares to debug the error.
- In phase 3, the programmer debugs the error by time traveling forward and backward through the run.
- Comparing to traditional debugging process, do you notice the difference?
TTVM-AWARE GDB

- Time travel with gdb
- TTVM/debugger interactions
- TTVM on guest applications
- Reverse gdb implementation
Time travel within gdb

- Traditional way of reverse debugging: **reverse execution**.
  - Gain control when crash occurs
  - Traverse up the call stack
  - Or re-run the system with a watchpoint set on the point variable. (only works if the bug is deterministic)

- **Drawbacks**
  - Non-deterministic bugs
  - What if the stack is corrupted?

- Adding reverse commands to gdb can solve the problem
Time travel within gdb (cont’d)

- Adding reverse commands to gdb can solve the problem.
  - Reverse continue
  - Reverse step
  - Goto (any time in the execution)
- Reverse continue takes the virtual machine back to a previous point. For example, the programmer set a watchpoint on the pointer variable and issue the reverse continue command. After executing this command, the debugger would return control to the programmer at the last time the variable was modified.
Example: reverse continue
First pass: count breakpoints; at the end, the programmer sees a list of breakpoints.
Second pass: the programmer pick one bp, and ttvm stops at the bp and return control to programmer.
TTVM/debugger interactions

- Problem: gdb mingle debug state and virtual machine state. (gdb doesn’t know the existence of virtual machine)
- So, TTVM must track all modifications gdb makes to the virtual state. Then, TTVM can make debugging state persistent across checkpoint restores by manually restoring the debug state after the checkpoint is restored.
- Also, TTVM removes any modifications caused by the debugger before taking a checkpoint, so that the checkpoint includes only the original virtual-machine state. (make ReVirt happy).
TTVM on guest applications

- TTVM can also be used to debug multi-threaded guest applications.

- Challenges:
  - Detecting the current guest process
  - Understand guest kernel task structs.

- This part of implementation is VMM-specific.
Reverse gdb implementation

- Gdb and TTVM communicate via gdb remote serial protocol.
- Implemented as host kernel device driver. (as a stub, must support the `g`, `G`, `m`, `M`, `c`, and `s` gdb commands)
- Gdb need not be modified.
- Virtual machine is out of scene – These reads and writes are transparent to VM.
- So, is this gdb a real debugger?
Reverse gdb implementation (cont’d)

In my opinion, it is not a real debugger.

- The TTVM acts as a remote debugger stub and communicate with gdb via gdb remote serial protocol.
- The gdb (TTVM-aware) sends commands to TTVM and displays whatever is transferred back from TTVM, like a terminal.
- Therefore, gdb is just like a remote controller for the VM logging/replaying system – ReVirt, and TTVM is just like the IR receiver in the middle.
- If this is the case, why not write a dedicated client replaying software?
PERFORMANCE/EXPERIENCES
Performance – Testing environment

- **Machine:**
  - Uniprocessor 3 Ghz Pentium 4,
  - 1 GB memory,
  - 120 GB Hitachi Deskstar GXP disk

- **Host OS:**
  - Linux 2.4.18 with UML running with SKAS(Separate Kernel Address Space) extension
  - TTVM modifications. (device driver hacks, communication with gdb)

- **Guest OS:**
  - UML port of Linux 2.4.20
  - Drivers for the USB and sound card devices
  - 256 MB memory
  - 5 GB disk.

- Both guest and host file systems initialized from RedHat 9 distribution.
Performance – guest workloads

Three guest workloads measured:
- SPEC99web using Apache (Spec99web is benchmark for evaluating performance of web servers)
- 3 successive builds of linux 2.4 kernel where each build executes make clean; make dep; make bzImage;
- PostMark – filesystem benchmark.
Performance – overheads

Time and space overhead of logging

Logging without checkpointing:

<table>
<thead>
<tr>
<th>Workload</th>
<th>Time overhead</th>
<th>Space overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec99web</td>
<td>12%</td>
<td>85KB/sec</td>
</tr>
<tr>
<td>Kernel build</td>
<td>11%</td>
<td>7KB/sec</td>
</tr>
<tr>
<td>PostMark</td>
<td>3%</td>
<td>2KB/sec</td>
</tr>
</tbody>
</table>

Replay without checkpointing:

1 – 3 % longer for all three workloads
Performance (cont’d)

Running time with checkpointing:
Running times are normalized to running the workload without any checkpoints

![Graph showing normalized runtime against checkpoint interval (sec)]
Performance (cont’d)

Space overhead of checkpointing

![Graph showing checkpoint overhead vs checkpoint interval](image-url)
Performance (cont’d)

Time to restore a checkpoint

The large jump at a restore distance of 600 seconds for PostMark is due to restore enough data to trash the host memory.
Experiences with TTVM

- Corrupted debugging information
  - TTVM still effective when stack was corrupted
- Device driver bugs
  - Handles long runs
  - Non-determinism
  - Device timing issues
- Race condition bugs
- Mremap bug
Conclusions

- This paper presents a creative application of virtual-machine logging/replay systems such as ReVirt. “TTVM is the first system that provides practical reverse debugging for long-running, multi-threaded programs such as an operating system.”

- However, TTVM is unlikely to beat the current debugging techniques in a short period.
  - First, TTVM relies on the accuracy of other systems such as VMM, ReVirt and their extensions/hacks. If any of these systems goes wrong, TTVM cannot give the accurate debug information.
Conclusions (cont’d)

- Second, I’ll put a question mark on TTVM’s efficiency. The paper says that “Taking a reverse single step took about 12 seconds on average” without mentioning the size of logging data. Also, no information of goto command’s efficiency.
- PostMark is able to trash the host memory during the reply tests. So, how long can we “time travel” using TTVM?
- I don’t know why they choose to hack/extend gdb instead of implementing a dedicated replay client software. The first approach apparently will add more overheads and constraints:
  - Must follow gdb remote serial protocol
  - Two execution passes for reverse commands
  - And more...
QUESTIONS
Virtual machine model

- The authors claim that 98% of the host OS code base can be debugged in the guest OS. How did they come up with this number? What kind of code is the last 2%, and why can it not be debugged?

- The authors state that "applying the techniques in this paper to a non para-virtualized VMM such as VMware would enable reverse debugging to work for any host OS bug." Why and how? Would "any host OS bug" include timing-dependent bugs? Are fully virtualized VMMs likely to be more similar to the OS that runs on the hardware than a para-virtualized VMM (a possible reason)? Does VMware provide reverse debugging?
Virtual machine model

- Does the fact that the OS being debugged is running on top of a virtual machine and another OS affect the bugs and debugging?
- In terms of reverse debugging, what are the pros/cons of adopting a guest OS like UML that creates a separate host process for guest processes compared to UMLinux that creates a single host process that encompasses both guest OS and guest processes?
Performance

- It is never mentioned what is a typical period of time their technique is capable of logging, could this be due to poor results?
- It is mentioned that the space overhead of iterative checkpointing is reasonable (2-6 MB/sec at 25 seconds checkpoints, and about 4-7 MB/sec at 10 seconds). What is unreasonable? I thought these numbers are huge, am I missing something? (Maybe the Halloween Party J)
Performance

- They say the space overhead for time travel is low, but if we take an average case of 40KB/sec and run it for a day, it's already at 3.3GB. Do that for a year and it's 1.2TB (worst-case is ~2.5TB)! Is it just me or does this seem like a high space overhead?
Multiprocessors / Extensibility

- Like ReVirt, this solution's major hindrance is the lack of multiprocessor support. Since this paper was published within the last 2 years, it's a shame the authors didn't engineer a solution that would apply to current generation hardware; they didn't even list it as potential future work but did make a brief comment on page 4. Is it just too onerous to track the fine-grained interleaving order of memory operations of these multiprocessor architectures?

- Is this technique scalable to a distributed system?
The authors say that “researchers are investigating ways to support replay on multiprocessors.” (p4) With today’s computers commonly having multiple processors, it is completely unimportant if it can’t support them. What advances have been made to support multiple processors?

A big problem to debuggers for multi-processor/multi-thread system is the time stamp of the application. In another word, it is very difficult to replay what happened at the bugging point. The little difference of running on different PE or the reduce power schedule will change the situation of application running. Did TTVM solve this problem?
Multiprocessors / Extensibility

- Checkpointing does not change
- Must be able to support replay
  - Topic of ongoing research
  - Support at hardware level, flight data recorder ()
    - Fast, limited amount of time recorded
- Software level, page protections to track sharing
  - Might be slow?
- Might not allocate all processors to one OS
Feasibility

- Running a debugger on a virtual machine below the OS seems like a great idea and this paper models it well. Has this been implemented? Would this be used only for developers while trying to debug their OS or could it be extended to run on every machine where it would automatically pick up bugs and send the logs to the developers to be debugged (similar to Microsoft’s error reporting but much more in depth)? Could this be extended even further to automatically pick up bugs and automatically learn to fix common ones?
Feasibility

- They keep stating that they found their debugger easy to use etc. but they're the ones who created the tool! (Of course they would find it intuitive!) Do people use this TTVM debugger? Do others exist?

- Ok, it's been mentioned many times that debugging operating systems is a very tough thing to do (and hurray for printf). This paper was released in 2005... so... what are the limitations of this method? (it requires the use of a VM?)

- This paper is fairly new (2005), how did people conduct OS debugging before using VMs? What were the most effective and competing methods?
Thank You!