Debugging operating systems with time-traveling virtual machines

Sam King, George Dunlap & Peter Chen
University of Michigan

Min Xie

Slides modified using Sam King's original version
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, OS
Example: NULL pointer

- Walk call stack
  - Variable not modified

ptr == NULL?
Example: NULL pointer

- Set a conditional watchpoint
  - ptr might change often
Example: NULL pointer

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

- All these are trying to find the LAST modification
Debugging with time traveling virtual machines

- Provide what cyclic debugging trying to approx.

ptr = NULL!
Debugging with time traveling virtual machines (TTVM)

- Reverse equivalent to any debugger motion function
  - Reverse watchpoint, breakpoint, step

- Implement using time travel to previous states
  - Must be identical to “buggy” run
  - Instruction level granularity
Overview

- Virtual machine platform
- ReVirt: virtual machine replay system
- Efficient checkpoints and time travel
- Using time travel for debugging
- Conclude
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
- How similar is the guest OS?
Similarity of guest OS

- Want guest OS to be similar to host OS so bugs are portable
- Differences not fundamental, result of VM platform we use
- Architecture dependent code different between guest OS
  - Low-level trap handling
  - MMU functionality
  - Device drivers
- Use the same host driver in guest
- Trap and forward privileged instructions from guest
  - IN/OUT instructions
  - Memory mapped I/O
  - Interrupts
  - DMA
- 98\% of Linux code runs unmodified in User-Mode Linux
ReVirt: fine grained time travel

- Based on previous work (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
- Low space and time overhead
  - SPECweb, PostMark, kernel compilation
  - logging adds 3-12% time overhead
  - logging adds 2-85 KB/sec
Checkpoints: coarse grained time travel

- Periodic checkpoints for coarse grained time travel

- Save complete copy of virtual-machine state: simple but inefficient
  - CPU registers
  - virtual machine’s physical memory image
  - virtual machine’s disk image

- Instead, use copy-on-write and undo/redo logging
Checkpointing for faster time travel

- Restore back to a prior checkpoint
  - Undo-log associated with this checkpoint n
    - Memory pages modified between checkpoint n and n+1

- Move forward to a future checkpoint
  - Redo-log associated with next checkpoint n+1
    - Memory pages modified between checkpoint n and n+1
How to time travel backward/forward
Sharing Log Page

checkpoint 1  checkpoint 2  checkpoint 3

undo log  redo log
Logging for Disk

- Avoid copying disk blocks into undo/redo logs
  - Maintaining in memory maps to new/old pages
How to time travel backward

checkpoint 1

redo log

undo log
Using time travel to implement reverse watchpoints

- Example: reverse watchpoint
- First pass: count watchpoints
- Second pass: wait for the last watchpoint before current time
Runtime Adding & Deleting Checkpoints

- Delete checkpoints to free up space
  - Assume 3 checkpoints (c₁, c₂, c₃)
  - Merge c₂'s undo log with c₁'s undo log
  - Merge c₂'s redo log with c₃'s redo log

- Optionally add checkpoints during replay to speed up time travel operation
  - Monitor pages changed after last checkpoint -> redo
  - COPY all pages in last checkpoint's undo log -> undo
Using TTVM

- Checkpoint at moderate intervals (e.g., 25 seconds)
  - < 4% time overhead
  - < 6 MB/s space overhead
- Exponentially thin out prior checkpoints (Boothe 00)
- Take checkpoints at short intervals (e.g., 10 seconds)
  - < 27% time overhead
  - < 7 MB/s space overhead
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
  - Live demo
Experiments

- Setup
  - Host OS: Linux 2.4.18 with skas extensions for UML and TTVM modifications
  - Guest OS: UML port of Linux 2.4.20 with host drivers for USB and soundcard devices
Time & Space Overhead
Conclusions

- Programmers want to debug in reverse
- Current debugging techniques are poor substitutes for reverse debugging
- Time traveling virtual machines efficient and effective mechanism for implementing reverse debugging
Questions

- Is it possible to debug device drivers without the device being present? Is it possible to replay all the interaction (both requests and responses) in such a way that the debugger can later supply the values as if the device is? ReVirt only logged one side of the communication on the assumption that the identical output could be obtained by providing identical input. However, it could potentially be useful to log runs at several locations and then debug in a lab where the device is not available.
Questions

- In this paper, they mention that a performance counter on the Intel P4 was used to count the number of branches during logging. In ReVirt they talked about it being the branch_retired counter of the Athlon. Which was it actually? Or did they change hardware between the experiments?
Questions

- "Replay occurs at approximately the same speed as the logged run." Some bugs only show up after a long runtime of an application under heavy load (for example, a difficult-to-find bug in a Web server) While checkpoints can be used to skip forward in time quickly, they do not necessarily catch all accesses to a particular variable that is corrupted. Is it possible to do this faster?
Questions

- The first example (the USB driver) doesn't sound like it should need time-travelling debugging. The stack trace is intact, and variables' values can be seen easily. The debugger in the kernel was working fine (the failure didn't break the kernel debugger itself, or any of its dependencies). In my experience, it's usually very easy to figure out the logic that leads to such things; the difficulty is usually what policy should be used to *FIX* the problem, not to find out how the problem occurs in the first place. Why is this a compelling example in favour of time-travelling debugging?
Questions

- If give the symbol table of the OS source, can we debug the source code and let it run step by step just like most IDEs do? To this question, I have used a windows kernel debugger called windbg, but it is really horrible.
Questions

- The VMM must be modified to support running real device drivers in the guest OS. Can a VMM run multiple different guest operating systems in this way? And the device drivers in guest OS will be physical device-specific or not?

- In the system structure for this paper, how would guest-user host process and guest-kernel host process interact with each other? Why not make the guest-user host process above guest-kernel host process?
Questions

- How do they make the guest OS look like a single process to the host OS?

- They run the same guest and host OS's. Was the reason their port required few changes because both OS's ran on the same machine? So if you ran Windows as the guest and Linux as the host, as long as you used versions that were designed for the same machine (i.e. x86), then the port would require few changes? I guess I don't understand why they say "UML's VMM is similar enough to the hardware interface that most code is identical between a host OS and a guest OS". What is the significance of this, and what happens if they are different OS's and this isn't the case?

- TTVM needs to track all modifications that gdb makes to the virtual state to make debugging state persistent across checkpoint restores. How do they do this?
Questions

- Is it efficient to implement the reverse continue by two-passes replaying? And if reverse step is needed for every reversed instruction, every reverse step needs a traveling from the nearest checkpoint, which seems too inefficient.

- If we have the stack dump of the crash point, why do we really need the reverse-traveling to debug? So I think, cases we really need record-and-replay is debugging parallel-concurrent programs or losing corrupted stack dump.
Questions

- Using emulators/VMMs for OS debugging is an established technique. However, the authors approach makes a qualitative difference by very clever use of visualization. The paper does not clearly indicate the debugging process of OS in a multi processor system. Is it possible to use this approach in a multi processor system? If so how check pointing can be done in this case?
Multi-processor support

- 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
Questions

- The authors point out that a para-virtualized OS has the potential to diverge in behavior when compared to a non-paravirtualized version of the same OS. Further the authors claim that bugs can be 'lost' as the two versions (PV and vanilla) of the OS diverge. The authors claim that in this case 92% of the non-driver code is identical between vanilla Linux and UML. Is this LoC metric an accurate way to measure the degree of divergence between the two kernels? It is probable that the 8% of code changed reflects modifications to 'core' parts of the OS code that could drastically change the behavior of the OS. Is there perhaps a better way to measure this divergence?
Questions

- TTVM uses a two pass algorithm to locate previous breakpoints, watchpoints, &c. In a way, what they present here is a framework that compresses an execution history and then interprets the resulting uncompressed execution stream to do useful debugging things (reverse breakpoints in this case). What else could be done with this framework to help debug OSes?
Questions
Thank You ^_^