Debugging operating systems with time-traveling virtual machines

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

University of Michigan

2005 USENIX Annual Technical Conference
Presented by Yeh Tsung-Yu
Outline

• Introduction
• Virtual Machine
• Time-Traveling virtual machine
• TTVM-aware gdb
• Performance
• Experience and lesson learned
• Conclusion and future work
Introduction

• Three reasons of traditional debugger can’t debug OS kernel well!
  – OS’s execution are non-deterministic because of non-deterministic events
    • Interleaving of multiple threads, interrupts, user input, network input.
    • When we re-run program, bug may disappear!
  – OS run for long periods of time.
  – Operating system may corrupt the state of the debugger.
    • Remote kernel debuggers depend on some basic functionality in the debugged OS
      – Such as reading and writing memory locations, setting and handling breakpoints (e.g., through the serial line).
Introduction

• Our contribution:
  – Time-Travel VM can re-execute deterministically to observe the non-determinant bug.
• Example that show traditional debugger can’t debug kernel well:
  – When the guest kernel attempted to call a NULL function pointer.
  – The error had corrupted the stack.
  – So standard debugging tools were unable to traverse the call stack and determine where the invalid function call had originated.
Virtual Machine

• We integrate time travel into a general-purpose debugger (gdb) and our virtual machine.
  – Commands implemented in gdb:
    • Reverse step, reverse breakpoint, reverse watchpoint (go back to the last time a variable was modified).

  – VM used is “User-Mode Linux” (UML)
    • OS first must be ported to run on this architecture.
    • And we also modified UML to use real driver in Host OS.
      – Actually there’re lots of bugs in device driver.
      – Now we can not only debug guest kernel, and also debug device driver.
Virtual Machine

- UML runs as two user processes on the host OS.
  - *Guest-kernel* run *kernel code*.
  - *Guest-user* run *user code*.

![Diagram showing the relationships between gdb, host processes, TTVM functionality, and the host operating system.]
Time-Traveling virtual machine

• TTVM re-execute deterministically by two facilities:
  – Log and replay.
  – In addition, checkpoint is implemented for quickly forward and backward.
Time-Traveling virtual machine

- Log and replay (previous work)
  - TTVM can be replayed deterministically by
    - Start from a checkpoint.
    - Replay all sources of non-determinism, for ex:
      - External input from network
      - Real-time clock
      - The timing of interrupt
  - How to log a interrupt
    - Data return from interrupt must be logged.
    - Interval (number of instructions) between interrupt must be logged.
      - Performance counter accumulate the number of executed instruction.
Time-Traveling virtual machine

• On previous work’s replay, only one checkpoint can be started from.
  • Forward or backward must first restore to the only CK.
  • As a result, we checkpoint more than one time, and make their interval as small as possible for user’s variety choice.

• For checkpointing faster
  – We incrementally take a checkpoint by “copy-on-write”.
    • Save only those modified memory pages.
Time-Traveling virtual machine

- Every memory access will produce two log.
  - Update data is logged to redo list for forward.
  - Old data is logged to undo list for backward.
Time-Traveling virtual machine

• Consider how to move CK(n+2) backward to CK(n).
  • First, restore A, B, C in CK(n)’s undo list
  • Second, restore D, E in CK(n+1)’s undo list

![Diagram showing the sequence of operations and undo/redo logs for different virtual machines CK(n), CK(n+1), and CK(n+2).]
Consider how to delete $CK(n+1)$
Reverse continue is implemented using two execution passes (Figure above).

Example followed: if we select a pointer as watchpoint, then

1. (1) Restore to CK(1),
2. (2) TTVM makes note of breakpoints BP(1), BP(2), and BP(3).
3. (3) Restore to CK(1),
4. (4) Replays execution, but stops at $BP(1)$, $BP(2)$, $BP(3)$ and returns control to the programmer (5).
Performance

• Overhead with log operation
  – Time overhead:
    • 12% time overhead for SPECweb99
    • 11% time overhead for kernel build
    • 3% time overhead for PostMark
  – Space overhead:
    • 85 KB/sec for SPECweb99
    • 7 KB/sec for kernel build
    • 2 KB/sec for PostMark
Performance

- Speed of “replay”
  - For the three workloads, 1-3% longer to replay than it did to log.
  - For workloads with idle periods, replay can be much faster than logging.
Performance

- Overhead for checkpoint

Figure 4: The effect of checkpointing on running time
Experience and lesson learned

- A non-deterministic bug that we countered
  - We use USB serial port hubs to communicate with other machines.
  - And our desktop computers to crash intermittently (usually overnight).

- A bug we found through TTVM
  - Under high load, a buffer in the **tty driver** became full during an interrupt service routine.
  - This caused usb driver request to the device to throttle its communication with the computer.
  - After issuing this request, the driver waited for a response, which caused the call to **schedule**.
  - This bug appeared in the current release of Linux 2.4 and 2.6.
Conclusion and future work

• In previous work, log and replay can achieve
  – Hardware fault-tolerance.
  – Intrusion analysys.

• In our contribution, we make non-deterministic bug won’t disappear and debug
  – Is there other research we can apply this technique?