Runtime Monitoring II

Lecture 8
CS195

Outline

- On the power of runtime monitoring
- Three more examples:
  - Software fault isolation
  - Data breakpoints
  - Complete machine simulation

Runtime Monitoring

- Runtime monitoring seems simple enough
  - Instrument a program to record what it does
  - Or influence its behavior
- But this is a very powerful technique
  - Review the theory

Computability Theory

- Thm: Some functions are uncomputable
- Proof: By diagonalization

Another View: Language Semantics

Runtime Monitoring \equiv \text{Replacing the Meaning Function}

- For each language primitive and connective, define a new semantic function

Summary

- Runtime monitoring is extremely general
  - In principle
- But, time limited
  - Can't afford to do too much
- Questions
  - What are interesting applications of monitoring?
  - How fast can we make them?
Fault Isolation

• Most OS's provide process isolation
  - Processes are isolated from one another
  - So if one crashes, it doesn't crash everything

• Exception
  - Older versions of Windows
  - And lack of isolation caused real problems . . .

• Protection boundaries are implemented by the OS
  - With hardware support
  - So-called hardware fault protection

An Issue

• How can isolated processes communicate?

• Answer: via remote procedure calls (RPC)

• Problem: Very expensive
  - Requires crossing kernel boundary, context switch
  - Twice: Once for call and once for return

The Problem

• Extensible systems allow third parties to augment an infrastructure with new code
  - Device drivers for an OS
  - More extremely, microkernels
  - Database systems extensible to new types
  - Web server plug-ins

• Choices
  - Run untrusted code in a separate process (slow)
  - Run untrusted code in same process (dangerous)

Software Fault Isolation

• Idea: Rewrite untrusted code to do fault isolation in software

• Trade slower instrumented code for freedom to run in one process
  - No need for a hardware protection boundary

  Slogan: You can still shoot yourself in the foot, but you can't shoot the other guy in the foot

Overview

• Maintain two invariants for isolated code:
  - Any jumps stay within the isolated code
  - Any writes are to data belonging to the isolated code

Segments

• Idea: Divide virtual memory into segments
  - Segment addresses have unique high-order bits

  • Protection subdomains within a process are defined by segments
    - Every write must be within the segment
    - Every jump must be within the segment
      - Except for return from the segment
Two Segments

- Actually, use two segments per fault domain
- One for data
- One for code
  - Prevents fault-isolated code from being self-modifying
  - Bug or feature?

Implementation

- Consider writes
  - Jumps are a little simpler
- Replace each write by the sequence:
  - dedicated-reg ⇐ target address
  - scratch-reg ⇐ (dedicated-reg >> shift-size)
  - scratch-reg == segment-reg
  - trap if not equal
  - store through dedicated-reg

Comments

- This requires several registers:
  - Dedicated-reg holds the address being computed
    - Needed in case code jumps into the middle of instrumentation
  - Segment-reg hold current valid segment
  - Shift-size holds the size of the shift to perform

A Faster Approach

- Idea
  - Don't test the segment bits
  - Just overwrite segment bits with correct segment
  - dedicated-reg ⇐ target-reg & mask-reg
  - dedicated-reg ⇐ dedicated-reg | segment-reg
  - This is sandboxing

Tradeoffs

- This is much faster
  - Only two instructions per instrumentation point
- Loses information about errors
  - Program may keep running with incorrect instructions and data
- Uses five registers
  - 2 for code/data segment, 2 for code/data sandboxed addresses, 1 for segment mask

Optimizations

- Don't instrument statically verifiable writes/jumps
  - Writes to static variables
  - Small jumps relative to program counter
- Allow some slop
  - Some references can fall outside of segment
  - Requires unused buffer regions around segments
  - Example: In load w/offset, sandbox register only
  - Sandboxing reg+offset requires one additional operation
Implementation Details

- Options for implementation
  - Binary instrumentation
    - Most portable & easily deployed
    - Also the hairiest option
  - Modified compiler
    - Less easy to adopt
    - But easier to implement
- Decision: modified compiler
  - Eventually moved to virtual machine

Results

- Works pretty well
  - Overhead ≤ 10% on nearly all benchmarks
  - Often significantly less
- Does what one expects
  - Protects host code
  - Extension code unprotected from itself

Data Breakpoints

- All debuggers provide control breakpoints
  - Stop execution at a particular point
- Less common are data breakpoints
  - Stop execution on reference to an address
- Why?
  - Instrumentation of every read/write is costly
  - Data breakpoints in DBX give 85,000X slowdown

The Basic Strategy

- "Monitored Regions"
  - Areas of memory to be monitored
  - Allows, e.g., monitoring of an entire array
- Instrument write operations
  - Check to see if they reference a monitored region

```
    st  %o0,[%fp - 20]
    sub  %fp,20,%g5
    call  check_1_word,0
    nop  ! Delay slot
```

Detecting Hits

- How do we detect hits to monitored regions?
- Maintain a huge bitmap
  - 1 bit for each word of memory
  - Bit is 1 ⇔ that word is monitored
  - ~3% memory overhead
- This is a standard trick

Optimization: Segmented Bitmaps

- Break bitmap into segments
  - Corresponding to set of high order bits
- Allocate segments lazily
  - First time a word is monitored in the segment
  - This reduces memory overhead
Why Bitmaps?

- Goal: Minimize memory loads to detect hits
- Pure bitmaps need only 1 reference
- Segmented bitmaps need 2 references
  - But likely to be cached
  - Use low order bit of segment pointer to indicate whether segment has any monitored regions

A Tradeoff

- What is the optimal segment size?
  - Larger segments
    - More locality in segment table
  - Smaller segments
    - Do second lookup less often
- 128 word segments were best
  - On machines of the day

Optimizations

- Statically remove write checks
  - Check loop invariant writes once outside of loop
  - Recognize ranges of writes in loops
    - Check once for entire range
- Drawbacks
  - Runtime static analysis is fairly complex
  - Must be repeated as breakpoints change
  - Not always a win
    - Slowdown for some programs

Bottom Line

- Data breakpoints with \( \leq 100\% \) overhead
- Optimizations usually give \( \leq 50\% \) overhead
  - But up to 180\% overhead

Conclusions

- Software fault isolation is a good idea
  - Part of Microsoft's .NET
- Data breakpoints are also a good idea
  - But very expensive
  - Not yet widely used in the form described here