Hardware Support for Software Debugging

Mohammad Amin Alipour
Benjamin Depew

Department of Computer Science
Michigan Technological University
**Hardware Support for Software Debugging**

Presented at the 23rd Systems and Software Technology Conference (SSTC), 16-19 May 2011, Salt Lake City, UT. Sponsored in part by the USAF. U.S. Government or Federal Rights License
Outline

• Motivation
  • Economy of Software Defects
  • Debugging Concurrency Issues
  • Flaws of Software Debugging
• Hardware-Based Software Debugging
  • Post-mortem Analysis: Replay Architectures
    • Flight Data Recorder
    • Rerun
  • Runtime Verification: Hardware Monitors
    • iWatcher
    • Log Based Architecture
  • Concurrency Debugging - ReEnact
• Conclusions
Cost of Software Defects

• Financial Costs
  • In a study by NIST in 2002 it was found that software bugs caused 59.5 billion dollars damage to the US economy

• Other Costs
  • Patriot Missile Defense System Failure
    • Feb 25, 1991
    • Caused killing of 28 American Soldiers
  • Radiation Treatment Overdose
    • June 1985 – January 1987
    • 6 patients overdosed with up to 100x normal radiation levels
    • Speculated to be because of software bugs
Concurrency Defects

• Moore's Law Breaking Down
  • Processor manufactures switching to adding more processing cores on the processor die

• Leveraging multi-core processors by using multithreaded applications introduces more potential runtime bugs in the software:
  • Data Race
  • Deadlock
  • Starvation/Livelock

• Notoriously difficult to debug the conditions listed above
Software Debugging - Common Techniques

• Static
  • Analysis of source code
  • Methods such as: model checking, program analysis
  • Not practical for hardware support

• Dynamic
  • Monitor executing programs dynamically
  • Instrument program and check for violations
  • Practical for hardware support
Software Instrumentation Example

Software Instrumentation inserts instructions within a program to trace the state that the program is in at a given point in time.

- **Static:**
  - Done at the source code level
  - Example: Assert statements

- **Dynamic:**
  - Done at object code level
  - Examples: Valgrind, PIN

```assembly
sub $0xff, %edx
counter++;
cmp %esi, %edx
counter++;
jle <L1>
counter++;
mov $0x1, %edi
counter++;
add $0x10, %eax
```

From: Cohn "Pin Tutorial"
Limitations of Software-Based Dynamic Verification

- **Performance**: Instrumentation can cause runtimes to be longer by an order of magnitude

- **Accuracy**: Unable to catch certain software defects such as memory alignment

```c
int x, *p;
/* assume invariant: x == 1 */
...
p = foo(); /* causes a bug: p points to x incorrectly */
*p = 5; /* line A: unintended corruption of x */
...
InvariantCheck(x == 1); /* line B */
z = Array[x];
...```
Performance Slowdown

Luk et al "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation" PIDI'05.
Hardware Based Software Debugging

• Hardware is faster
  • Runtime overhead of using hardware techniques for dynamic software verification is less than that of software techniques
  • The tradeoff is hardware will become more complex and possibly more expensive

• Details of actual execution that could inaccessible at the software level
  • Compiler Optimizations
  • Software Libraries
  • Can see what is happening at the machine level to expose these issues
Post-mortem Analysis

- Debugging requires understanding of the cause of the failure, one common way is reproduction of failure from logging a program's actions

- In sequential programs, reproduction of failure is a relatively straightforward process
  - Check-pointing statements of interest
  - Trace through checkpoints using debuggers

- In concurrent programs, reproduction of failure is difficult
  - Non-determinism of thread execution leads to many possible program execution states
  - Need to preserve the order in which threads execute relative to each other to reproduce the runtime which lead to the failure
Flight Data Recorder (FDR)

- Modeled after the principles of an actual flight data recorder, FDR constantly records information during a program’s execution.

- Assumes sequentially consistent memory model for the base hardware.

- Maintains a record of a set number of previous instructions.

- If a trigger occurs while recording, a log file including a core dump and the record FDR has been keeping is output to a file for replay after program execution.
FDR architecture

FDR Evaluation

Using Wisconsin workload simulations on a simulated machine four 1Ghz processors and a log interval of 1 billion instructions:

• Speed
  • Runtime of the workloads was found to increase less than 2%

• Hardware Cost
  • 7% overhead of system memory used for logging the longest intervals in this system, assuming 512MB memory per processor
  • At least 4KB additional memory needed on the die of the processor for this setup
Rerun

- Based off the approach that FDR uses, also relies on sequential consistency memory model.

- Instead of recording a set interval of instructions, Rerun keeps track of the time a thread executes without memory access conflicts with other threads.

- "Episodic Memory Race Recording"
  - Determines the length of an episode by ending a current episode and beginning a new one when threads access the same memory locations.
  - Orders the episodes of multiple threads to produce a faithful deterministic replay of the whole program, which is done using a Lamport Scalar Clock.
Rerun Architecture

Rerun Evaluation

• Speed
  • Speed overhead of using FDR not noted in the paper, given the small log file sizes and minimal hardware costs, should be minimal

• Hardware Cost
  • 166 bytes per core

• Other notes
  • Other main focus was to keep log file sizes minimal for scalability to large numbers of cores, in which it was noted that Rerun was on par with FDR’s log file size
Runtime Monitoring

- Efficient run-time monitoring of programs can be exploited to
  - Automatically detect invalid software states during runtime
  - Potentially recover from invalid software states to improve reliability

- Some security flaws and safety violations can be reported and in some cases prevented by keeping track of the execution of the program
  - Taint Analysis
  - Buffer Overflow
iWatcher Description

• Hardware architecture support for memory monitoring

• It allows programmers to associate a monitoring function to a portion of memory

• It can be used to detect and prevent memory related faults such as buffer overflow, memory leaks and stack smashing

• Leverages Thread Level Speculation (TLS)
  • To support breakpoints and rollbacks when a trigger occurs
  • To execute monitoring code in parallel with the program code
iWatcher Usage

- The iWatcher system provides system calls as the interface for a programmer to monitor memory locations
  - `iWatcherOn()` method specifies the memory addresses to be monitored, types of accesses (read, write or both), monitoring function to trigger in the occurrence of such event
  - `iWatcherOff()` disables monitoring of specific memory addresses

- Programmers can insert these system calls into the source code wherever monitoring is needed

- iWatcher will invoke the specified monitoring function upon a triggering access to the watched memory
iWatcher Architecture

iWatcher Evaluation

• Speed
  • 4-80% runtime overhead on experimental data
  • 66-175% runtime overhead on heavy dynamic load monitoring

• Hardware Cost
  • 2 bits per word in L1 and L2 caches
  • 2040 bytes for RWT and VWT tables

• Other notes
  • Other main focus was to keep log file sizes minimal for scalability to large numbers of cores
Log Based Architecture (LBA)

- The motivation is that when a program and its monitor run on the same shared resources like register files and processors, they compete for resources which slows down the program execution.

- Based on decoupling program and its monitor by leveraging idle cores in multi-core processors to handle the monitoring of an application.

- Information to monitor is passed from the core running the application to the core monitoring the application using a log buffer with compression.
LBA

Application          Lifeguard

Operating System

Core 1 Capture
Compress

Log Buffer (e.g. L2 cache)

Core 2 dispatch
decompress

Courtesy: Chen, et al "Log-based architectures for general-purpose monitoring of deployed code". In ASID '06.
LBA Evaluation

• Speed
  • The experiments show 4 to 19 times speedup comparing to a comparable software-only technique, Valgrind

• Hardware Cost
  • It requires twice the number of cores necessary to run the application without instrumentation in order to function correctly

• Variety of Lifeguard Functionality
  • AddrCheck
  • TaintCheck
  • LockSet

• Lifeguards can be toggled on or off during runtime
Concurrency Debugging - ReEnact

- Leverages modified Thread-Level Speculation (TLS) hardware

- Create partial orderings of threads in a multithreaded program using logical vector clocks

- Using these orderings, ReEnact is able to detect and often repair data race conditions in a multithreaded program

- Experiments were done with four processors, one thread per processor

- Slowdown during non-bug detected runs was only 5.8%
Conclusions

• There is current architecture support for limited hardware monitoring
  • Currently up to four hardware watch-points
  • x64 architecture supports additional addressing for debug registers, but currently not implemented in hardware

• All hardware debugging methods observed require architecture modifications to current architectures in order to function, therefore are tested through simulations
Conclusions

• In order for new hardware debugging practices to be leveraged in industry, current architecture support will need to be expanded

• With the growing need for debugging multithreaded applications on multi-core processors, will there be enough demand to justify the cost of increasing architecture support for hardware debugging?
Questions?
References


• Chen, et al "Log-based architectures for general-purpose monitoring of deployed code". In ASID '06.


• Luk, et al "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation" PLDI'05.
Acronyms

• LBA: Log Based Architecture
• NIST: National Institute of Standards and Technology
• FDR: Flight Data Recorder
• TLS: Thread Level Speculation
• EEMR: Episodic Memory Race Recording
• RWT: Range Watch Table
• VWT: Victim Watch Flag Table
• HW: Hardware
• MTS: Maximum Time Stamp
• DMA: Direct Memory Access
• CPU: Central Processing Unit