CHESS or “How to track Heisenbugs”

Presentation of:
Finding and reproducing Heisenbugs in concurrent programs

Authors:
Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu


By: Mathias Dürrenberger
What is a Heisenbug?

- Werner Karl Heisenberg (5 Dec 1901 - 1 Feb 1976)
- Uncertainty principle
- The bug disappears when we try to debug the code
What is a Heisenbug?

- The bug only appears when compiler optimizations are enabled.
- Multithreaded systems: occasional/rare race conditions - thread scheduling is essentially non-deterministic.
- Big productivity issue (it can take weeks to find one).
- And often huge associated risks too.
Why Heisebugs matter

- Failure of real time systems can have catastrophic consequences!
- Nuclear power plant
- Engine control systems
- Driving assistance (break control, traction control, ...)
- Locomotive control systems
- etc
How does CHESS work?

- Take full control of scheduling of threads and of asynchronous events
- Capture interleaving nondeterminism of program with a Lamport happens-before graph
- Reduce state space
- Drives program through possible thread interleavings
CHESS Architecture

- Every run takes a different interleaving
- Reproduce the interleaving for every run

Source: Microsoft
How CHESS interacts

- Win32: DLL-shimming (overwriting the import address table of the program under test)
- .Net: extended CLR profiler
- Singularity: static IL rewriter
- Consider sub-components with complex API as part of application under test
Wrappers

Program under test

Subcomponent with complex API
- wrap it or not?

Win32 Wrappers

Windows

Must understand semantics

Is part of tested system

- wrap it or not?
Main idea: replace the scheduler

Basic functions of CHESS scheduler: playback and monitor
- Playback from trace file
- Launch a thread
- Record events during time slice of a thread into trace file

One step further: generate a new thread interleaving
- Use the recorded information to find an interesting next schedule in a smart way (called: search)
- This is an iterative process
Record

- Disturb system under test as little as possible
- We do not want to change the real time behavior in a significant way
- Must understand the semantics of concurrent API's
- Monitor concurrency primitives: `EnterCriticalSection`, `ReleaseCriticalSection`, `CreateThread`, `QueueUserWorkItem`, `CreateTimerQueue`, `asynchronous file I/O`, ...
Example

Thread 1

... 

EnterCriticalSection()  
// Access shared resource

LeaveCriticalSection();

...

Thread 2

... 

SleepEx()

...

Thread is not switched

Thread is switched
Control flow model

- Current executing task
- Map of resource handles to synchronization variables
- Set of enabled tasks
- Set of threads waiting on each synchronization variable
  - Determine if call switches thread
  - Determine if call yields new task/terminates task
Does CHESS deliver?

- Failure of a nightly test run (after many month's of successes)
- Isolate offending unit (comment out the rest in test harness) => 30 minutes
- Running under CHESS => 20 seconds to discover a deadlock
- Run code in standard debugger, with the offending schedule being driven by CHESS
- Several other projects where CHESS has been used successfully within Microsoft
Concluding remarks

- The direction is right, it’s a promising approach
- But does it target the platform that matters?
- Win32 has a bad reputation as real time platform (I.e. interrupt latency)
- What about Linux and other industrial real time OS’s?
- Porting is rather complex (Win32: 134 lib’s with 2512 functions, .net: 64 lib’s with 1270 functions)
Example:
EnterCriticalSection

- Can block (switch the thread) or immediately acquire the resource
- Emulate call with combination of: TryEnterCriticalSection and EnterCriticalSection
- Record outcome in “operation” value
- Update control flow model
- If “try” fails, add current task to set of tasks waiting on that resource, remove from set of enabled tasks
- When released, move all tasks from waiting set to enabled set
Model of concurrent interactions

- Lamport: happens-before graph, use to model relative execution order of threads
- Node := (task, synchronization variable, operation)
- Task :- { thread, threadpool work item, asynchronous callback, timer callback }
- Synchronization variable :- { lock, semaphores, atomic variables accessed, queues }
- Operation :- [ isWrite (changes state of resource), isRelease (unlocks tasks waiting on that resource) ]
Summary: graph node

- Task: take current executing task
- Synchronization variable: get from resource handle map
- Operation: get from call semantics/analysis
- Advantage: is robust by design and fault tolerant
And the edges?

- They are built, based on analysis of inter-thread communication
- Operation: isWrite, isRelease (from contents of node)
- Can this call disable the current task? I.e. result in a thread switch?
- Requires to model the control flow among threads
- Understanding of call semantics needed
CHESS scheduler

- Iterate on: replay, record, search
  - replay: redo a previous path
  - record: behave as fair, nonpreemptive scheduler, let thread run until it yields control, process graph
  - search: systematically enumerate possible thread interleavings, come up with an interesting schedule
- Problem: how to reduce state space?
Reduce state space

- Bound number of preemptions: three is enough to find almost any bug
- Scope preemptions: exclude standard libraries (I.e. C run time)
- Only use preemption at very specific places (when accessing shared variables)
Further readings

Additional materials:
Can be downloaded for MS Visual Studio 2008
References

