Tracing Data Flows to Find Concurrency Errors

Presented by: Benjamin Ylvisaker

Senior Scientist
GrammaTech, Inc.
531 Esty Street,
Ithaca, NY 14850
Tel: 607 273-7340
E-mail: benjaminy@grammatech.com
GrammaTech Profile

- Spun out of Cornell
  - Tim Teitelbaum, CEO and co-founder, Emeritus faculty at Cornell
  - Tom Reps, President and co-founder, Faculty at U. Wisconsin

- Focus
  - Program analysis and manipulation
  - Source and binaries

- Some customers
  - JPL (site license), Mitre, Draper, NASA, Airbus
New Tools for Static Concurrency Bug Detection

- Detection of data races
  - DARPA-funded research

- Detection of deadlock and other misuses of locks
  - NASA-funded research
  - In partnership w/ Gerard Holzmann at JPL
    - Power of 10
Agenda

- Why multi-core is important
- Why concurrent programming is hard
- How static analysis can help find concurrency defects
Soon (almost) All Processors will be Multi-core

- Scaling of single-threaded performance has fallen off a cliff in the last couple of processor generations

- All processor vendors are moving to multi-core designs
  - Even embedded processors

- But there are some major obstacles to adoption
  - Applications need to be explicitly concurrent
    - Automatic parallelization still not mainstream
  - (Correct) concurrent programming is difficult
Concurrency Adds a New Source of Complexity

There are six possible interleavings of two threads with two instructions each.

With three instructions each, there are twenty possible interleavings.
Non-deterministic Ordering in the Real World

- Real-world threads execute billions of instructions per second

- Interleavings are determined by real-world events and the system scheduler

- Ordering of events and scheduling choices are effectively non-deterministic

- Correctness of execution can depend on relative ordering
  - Race conditions are a major source of unintended time/scheduling dependence
Eliminating Data Races

- Programs can be designed to be less sensitive to scheduling variation
  - Less sensitive => traditional software QA is more effective

- Potential data races and lock misuse are major sources of unintended sensitivity to scheduling variation

- CodeSonar helps eliminate potential data races and lock misuse
Data Races

- A data race arises when:
  1. Multiple threads of execution access a shared piece of data
  2. At least one thread changes the value of the data
  3. Access is not separated by explicit synchronization

- Data races can leave a system in an inconsistent state

- Data races can lurk undetected and only show up in rare circumstances with mysterious symptoms
Example Data Race

Thread 1

count := count + 1;

Thread 2

count := count - 1;

load value from memory into register
increment value
store value back in memory
Data Races are Hard to Debug

- Rare occurrence means little chance of detection during testing

- Diagnosis is difficult

- Reproducibility is a major problem

- Developers tend to assume each thread executes in-order (sequential consistency)
  - Effects of thread interaction easy to miss
We Built Data Race Detection on an Existing Static Analysis Tool -- CodeSonar

- Static bug finder

- Uses symbolic execution for whole-program path-sensitive analysis
  - Bottom-up in the call graph (callees analyzed before callers)
  - Equivalent paths are summarized together to save space
  - Precise pointer analysis and feasible inter-procedural path extraction
Finding Data Races at their Source

- We use a lockset-style approach
  - For each shared memory location, all accesses must be protected by a single lock

- During symbolic execution, find what locks are held when shared memory locations are accessed

- Find thread entry points (with library models)

- For each pair of thread entry points and each shared memory location, intersect the sets of locks to find possible data races
Data Race at input.c:142

Jump to warning location ↓

No properties have been set.

warning details...

Show Events | Change View | Options

thread 1
main

(int argc, char *argv[])

Event 1: Thread 1 starts here. ▼ hide

290
int main (int argc, char *argv[])
291{
292
int i;
293
/
* Parse command line arguments conforming with

340
RealSide = board.side;
341
dbg_printf("Waking up input...\n");
342
dbg_printf("input_status = %d\n", input_status);
343
}

input_wakeup

(void)

void input_wakeup(void)

pthread_mutex_lock(&input_mutex);
input_status = INPUT_NONE;

Data Race
This code writes to input_status.

- The other thread reads from input_status. See other access.
- The following locks are currently held: input_mutex.
  - None of these locks are held by the other thread when it accesses
    input_status so a race may occur.

The issue can occur if the highlighted code executes.

Show: All events | Only primary events

thread 2
input_func

(void)

void *input_func(void *arg __attribute__((unused)))

Event 22: Thread 2 starts here. ▼ hide

119
void *input_func(void *arg __attribute__((unused)))
120{
121
char prompt[MAXSTR] = "";
122
while ( !(flags & QUIT)) {
123
if ( !(flags & XBOARD)) {
124
sprintf(prompt,"\$s (\$d) : ",
RealSide ? "Black" : "White",
(RealGameCnt+1)/2 + 1);
125
}
126
}
127

pthread_mutex_lock(&input_mutex);
128
gnuchess_getline(prompt);
129
input_status = INPUT_AVAILABLE;
130
pthread_cond_signal(&input_cond);
131
pthread_mutex_unlock(&input_mutex);
132

pthread_mutex_lock(&wakeup_mutex);
133
/*
  * Posix waits can wake up spuriously
  * so we must ensure that we keep waiting
  * until we are woken by something that has
  * consumed the input
  */
134

while (input_status == INPUT_AVAILABLE) {

Data Race
This code reads from input_status.

- The other thread writes to input_status. See other access.
- The following locks are currently held: wakeup_mutex.
  - None of these locks are held by the other thread when it accesses
    input_status so a race may occur.

The issue can occur if the highlighted code executes.

Show: All events | Only primary events
No, that Data Race is Not Benign

- Double-checked locking for lazy initialization

```c
if (!init_flag) {
    lock();
    if (!init_flag) {
        my_data = ...;
        init_flag = true;
    }
    unlock();
}
tmp = my_data;
```

- See Boehm, “How to Miscompile Programs with ‘Benign’ Data Races”
How CodeSonar Detects Deadlocks

- Most commonly adopted approach to avoiding deadlock is to assign a partial ordering to the resources
  - Proposed by Dijkstra in 1965 as a solution to the Dijkstra/Hoare Dining Philosophers Problem

- If it is possible for lock A to be held when lock B is acquired, A is “before” B

- CodeSonar examines the code and issues a Conflicting Lock Order warning if any pair of locks can be acquired in different orders by different threads
Additional Concurrency Checks

- Process starvation
- Unknown Lock
- Missing Lock, Missing Unlock, Lock/Unlock Mismatch
- Double Lock, Double Unlock
- Try-lock that will never succeed
Conclusions

- Multi-core processors are inevitable
  - Explicitly concurrent programming is the only reliable way to harness the performance of multi-cores today

- Concurrency errors are insidious
  - Difficult to reproduce, diagnose, and eliminate
  - Even apparently benign data races can have surprisingly detrimental consequences

- We are bringing research in static detection of concurrency defects to industrial-strength bug finding tools
Thanks for Your Attention

Questions?