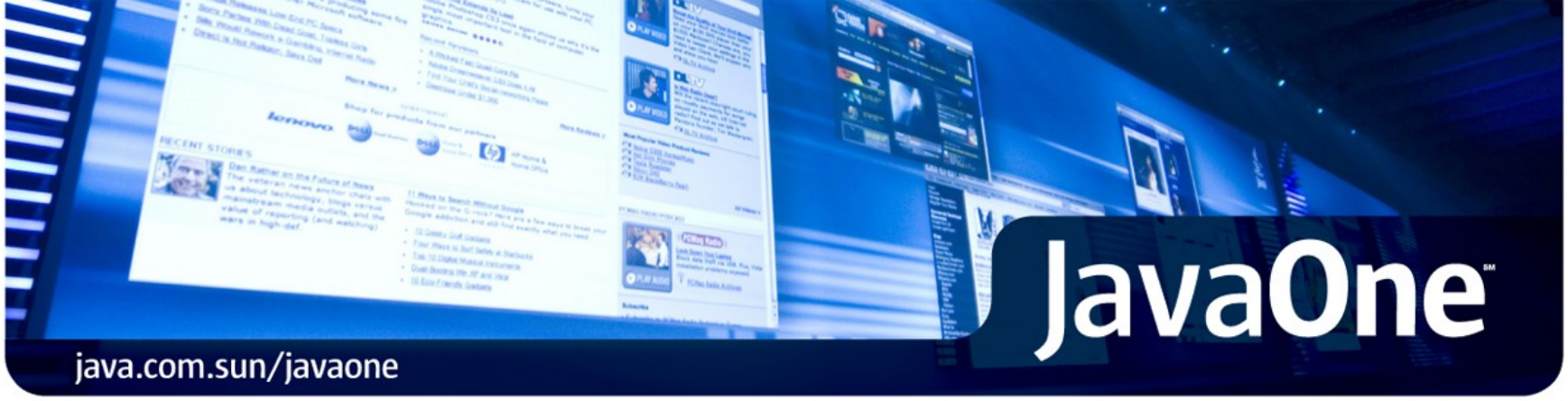


Experiences With Debugging Data Races

Dr. Cliff Click, Distinguished Engineer

Azul Systems

http://blogs.azulsystems.com/cliff

AZUL SYSTEMS

Java

Sun microsystems

Learn what Causes a Data Race

Learn how to Find Data Races

Learn how to Avoid Creating Data Races

GOAL

A Short Debugging Tale

```
if (method.hasCode() != true)
  return false;
code = method.getCode();
...setup;
code.execute();  // Throws NPE rarely!!!
return true;
```

- Example is real; simplified for slide
  - Many more wrapper layers removed
  - Shown "as if" aggressive inlining already
- I've debugged dozens of slight variations
- Apparently I'm not alone:

  Learning from mistakes --
  A Comprehensive Study on Real World Concurrency Bug Characteristics
  http://opera.cs.uiuc.edu/paper/asplos122-lu.pdf

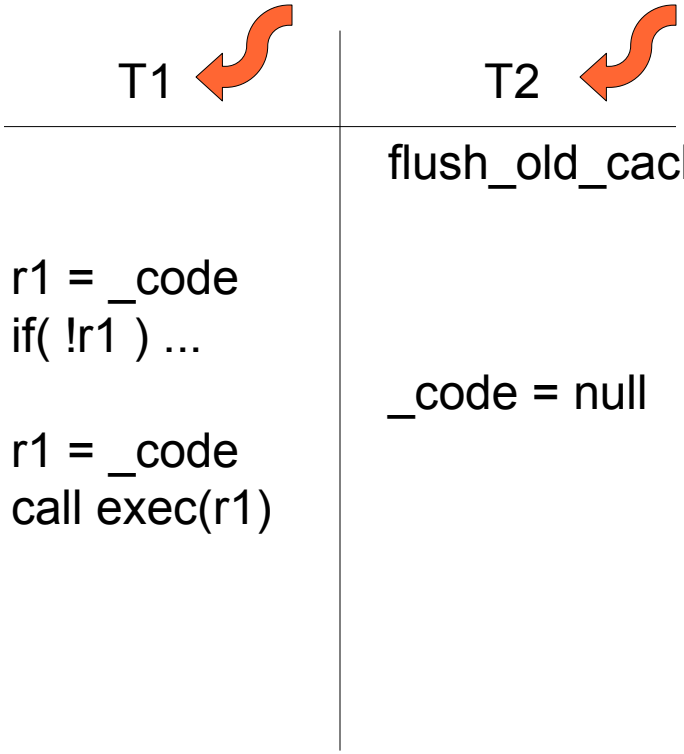
Agenda

- (not very) Formal Stuff
- Reordering Memory – A peek under the hood
- Common Data Races
- Finally(!) Some Debugging Techniques
- Summary & Advice

What IS a Data Race?

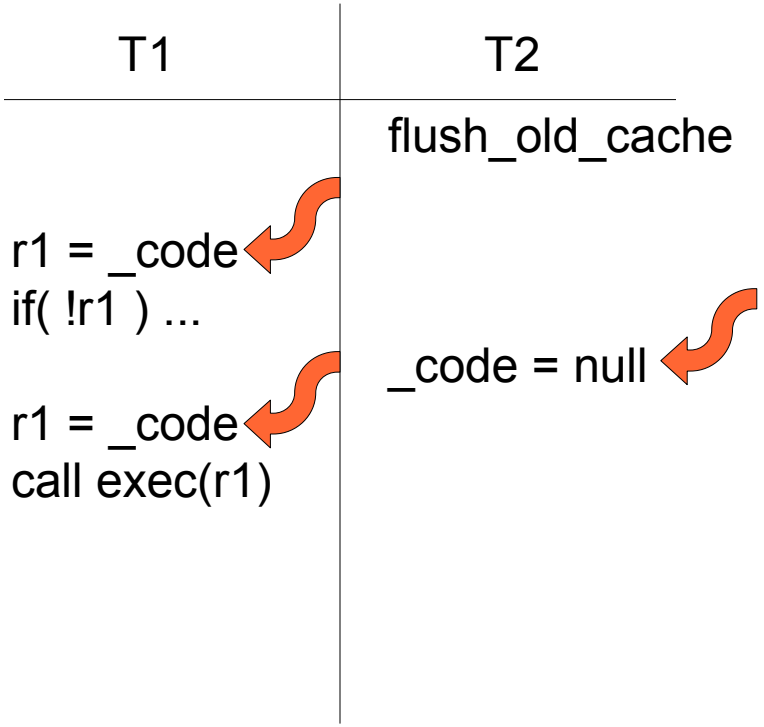
- Formally:
  - Two threads accessing the same memory
  - At least one is writing
  - And no language-level ordering
- Informally:
  - Broken attempt to use more CPUs
  - (but can happen with 1 CPU)
- Generally because 1 CPU is too slow
  - End of frequency scaling :-(
- Multi-core, dual-socket, big server, etc

Timeline of a Data Race



- Two threads

Timeline of a Data Race



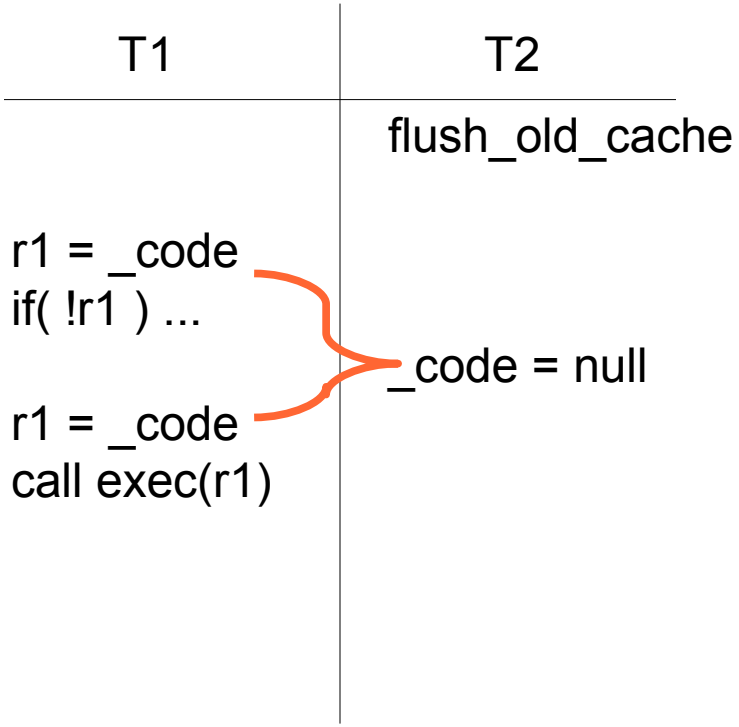
- Two threads
- Accessing same memory

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- Two threads
- Accessing same memory
- At least one is writing

Timeline of a Data Race



- Two threads
- Accessing same memory
- At least one is writing
- **No language-level ordering**

Timeline of a Data Race

T1	T2
	flush_old_cache
r1 = _code if(!r1) ...	
	_code = null
r1 = _code call exec(r1)	

- OK if:
  - Write before 1st Read OR
  - Write after 2nd Read
- **Broken if in-between**
- Pot-Luck based on OS thread schedule
- Crashes rarely in testing
- More context switches under heavy load
- Crash routine in production

Agenda

- (not very) Formal Stuff
- **Reordering Memory – A peek under the hood**
- Common Data Races
- Finally(!) Some Debugging Techniques
- Summary and Advice

Reordering Memory Ops

T1	T2
_datum = stuff	
_init = true	
	r1 = _init if(!r1) ...
	r2 = _datum

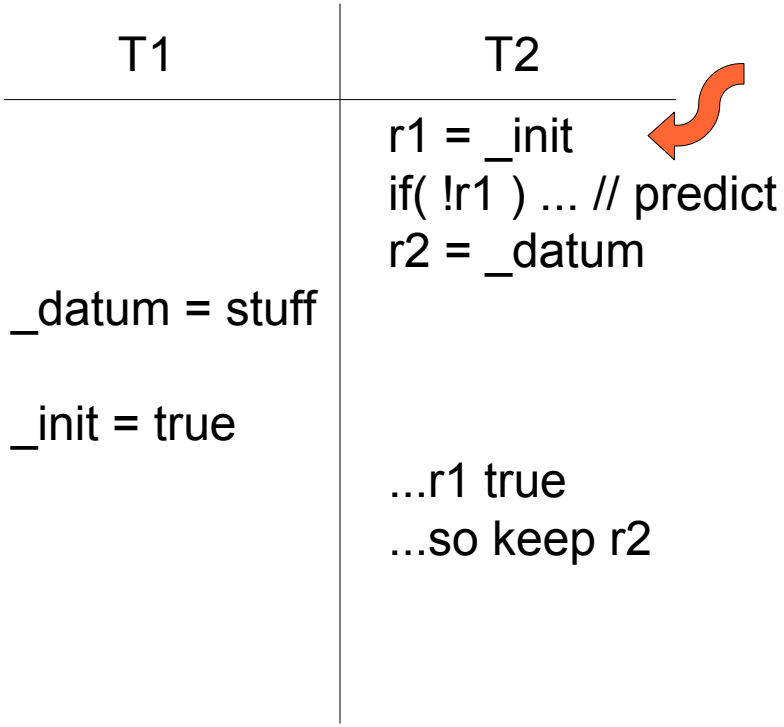
- Writing 2 fields
- Can T2 see stale _datum?
- Yes!

Reordering Memory Ops

T1	T2
	r2 = _datum
_datum = stuff	
_init = true	
	r1 = _init if(!r1) ...

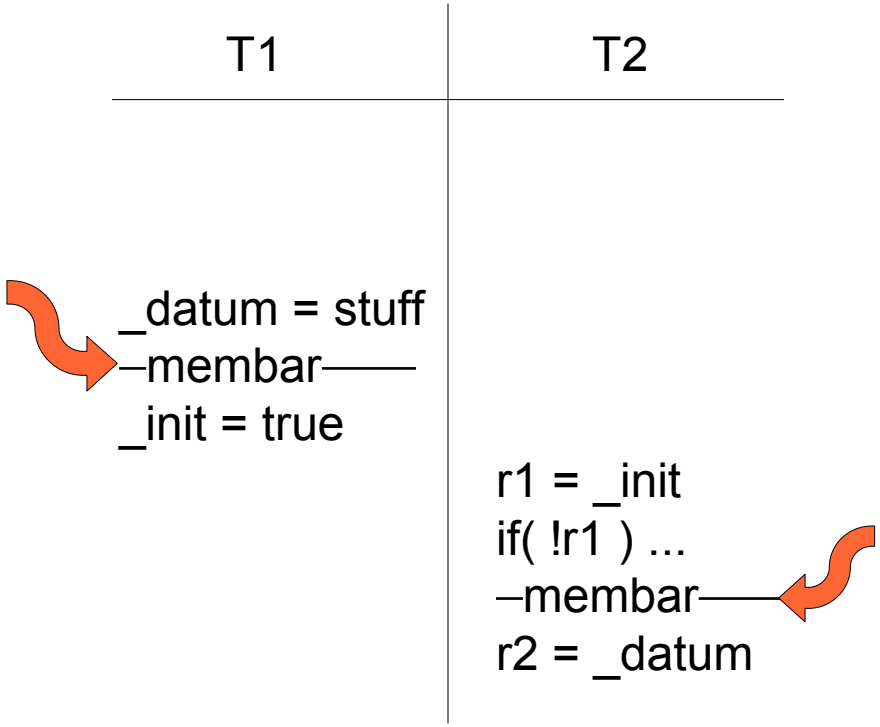
- Writing 2 fields
- Can T2 see stale _datum?
- Yes!
- Compiler can reorder
  - Standard faire for -O
- Java: make _init volatile
- C/C++: dicier
  - no C memory model yet

Reordering Memory Ops



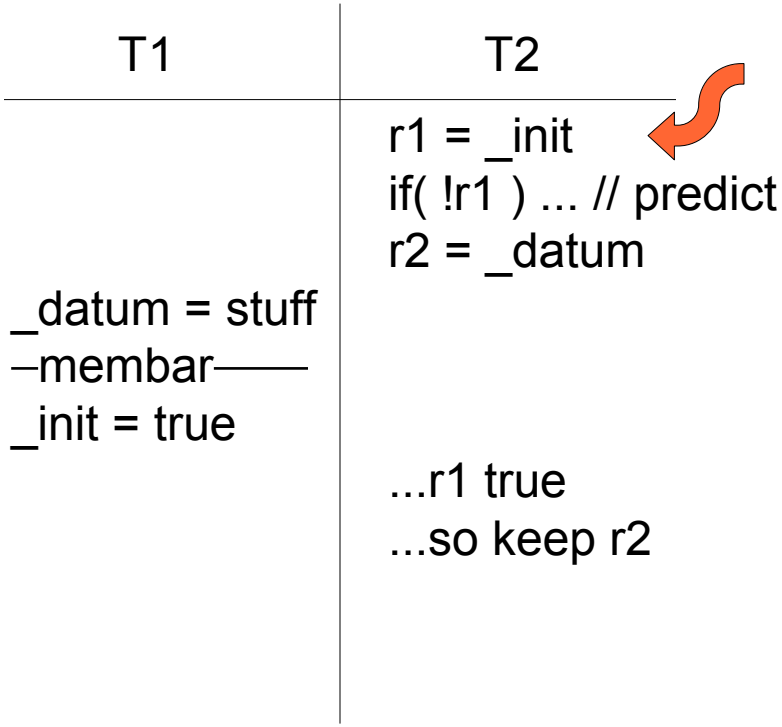
- Writing 2 fields
- Can T2 see stale _datum?
- Yes!
- **Hardware can reorder**
- Load _init misses cache
- Predict r1==true
- Speculatively load _datum early, hits cache
- _init comes back true
- Keep speculative _datum

Reordering Memory Ops



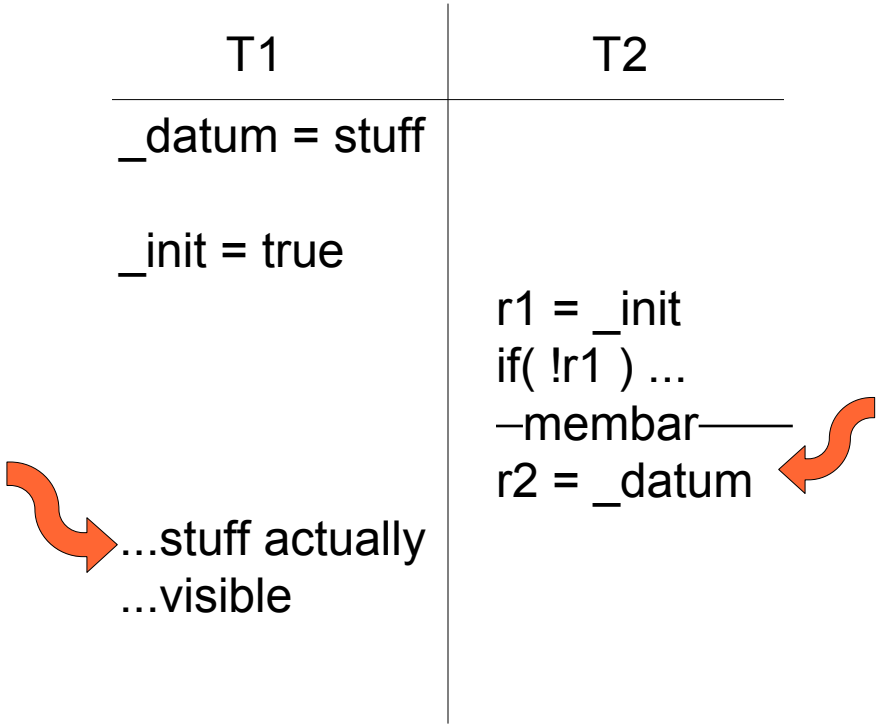
- Writing 2 fields
- Can T2 see stale _datum?
- No!
- Need store-side ordering
- Need load-side ordering
- Included in Java volatile

Reordering Memory Ops



- Writing 2 fields
- Can T2 see stale _datum?
- Yes!
- Missing load-ordering
- Read of _init misses
- Predict branch
- Fetch _datum early
- Confirm good branch

Reordering Memory Ops



- Writing 2 fields
- Can T2 see stale _datum?
- Yes!
- Missing store ordering
- Write of _datum misses
- Write of _init hits cache
- T2 reads _init
- T2 reads stale _datum

Agenda

- (not very) Formal Stuff
- Reordering Memory – A peek under the hood
- **Common Data Races**
- Finally(!) Some Debugging Techniques
- Summary and Advice

Common Data Races

- My experiences only*
- Double-read with write in the middle:

```
if( _p != null )
  ... _p._fld...
```

```
_p = null;
```

- ...and it's usually a null write
- Two writes with reads in the middle:

```
_size=_size*2;
_array=new[_size];
```

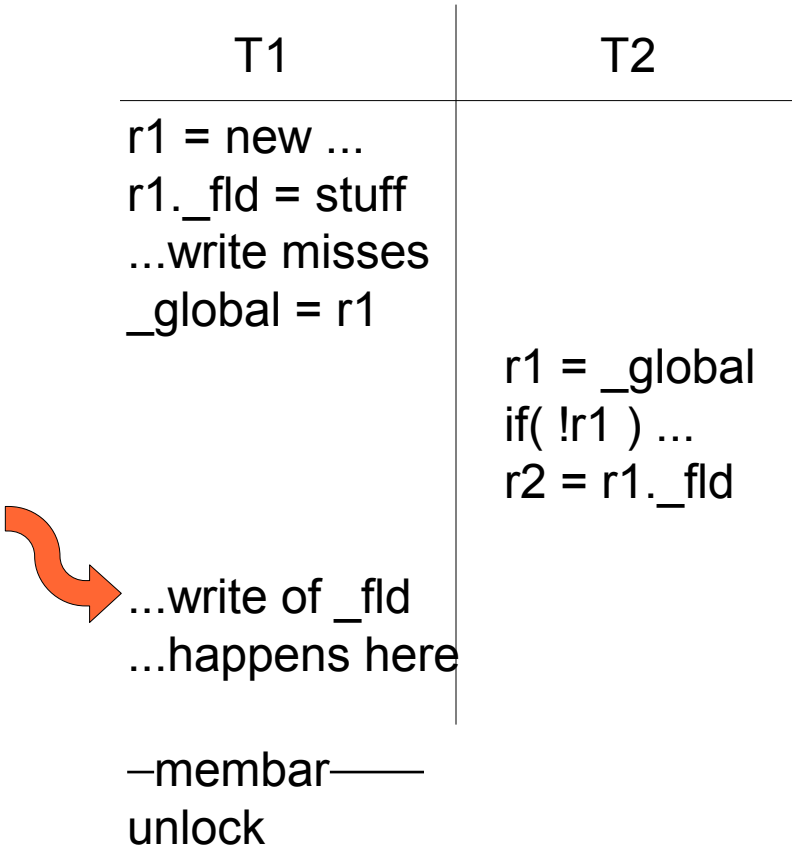
```
..._array[_size-1]...
```

- Double Checked Locking:

```
if( _global == null )
  synchronized(x)
    if( _global == null )
      _global = new ...;
```

*Learning From Mistakes – ASPLOS 2008

Double Checked Locking



- Initializing global singleton
- Can T2 see stale _fld?
- Yes!
- Misplaced store-ordering
- Unlock puts barrier AFTER both writes
  - Not between
- Actually missing load-ordering as well
  - True data dependency
  - Only IA64 might reorder?

More On Double-Read

- `if( _p != null ) {..._p._fld...}`
- Compiler likes to CSE both loads together
  - No bug if CSE'd together
  - C: Crashes in debug build, not product build
  - Java: Crashes before high-opt JIT kicks in
- Crashes when context-switch between reads
- i.e., just as heavy load hits system
- If you survive startup, might last a long time
- Bug can persist for years
  - Plenty of personal experience here...

Getting Clever w/HashMap

- Using a collection unsafely, and catching NPE
  - But not catching rarer AIOOB
  - Bug bit both a customer AND in-house engineer
- Idea: HashMap w/single writer, many readers
  - Thinking: No locking needed since 1 writer
  - Readers sometimes see ½ of 'put'
  - Throw NPE occasionally; Fix: catch NPE & retry
- Writer is mid-resize, reader hashes to larger table
  - But does lookup on smaller table,
  - Throws AIOOB – not caught, program crash
- Reader calls size(), size() calls resize, and...
  - Other Reader throws AIOOB – not caught, program crash
  - Or list corrupted; cyclic..
    - touching threads hang forever spinning on the cycle

Agenda

- (not very) Formal Stuff
- Reordering Memory – A peek under the hood
- Common Data Races
- **Finally(!) Some Debugging Techniques**
- Summary and Advice

Visual Inspection

- Easy to get started with
  - Appears to be the *State of the Practice*
- Works on core files; after the fact
- Very slow per LOC
- Just obtaining the code is often a problem
- Sometimes can make a more directed search
  - e.g., Stack trace points out where somebody failed
  - Play mental Sherlock Holmes w/self
- Requires Memory Model expert, domain expert
- **Does Not Scale**

Visual Inspection

- **Biggest Flaw: Not Knowing The Players**
- Maintainer cannot name shared variables
  - Or *which* threads can access them
  - Or *when* they are allowed to touch
- Sometimes suffices to Make Access Explicit
  - Large Flashy Comments on shared variables
  - At least the Players become obvious
- Avoiding Double-Read:
  - Often requires changing accessor patterns
  - No more "if(isReady()) ... get()" pattern
  - Return flag & value in 1 shot, cache in a local variable:

```
tmp = get();
if( tmp != null ) ...tmp._fld...
```

Visual Inspection

- 2nd Biggest Flaw: forgetting "The Cycle"
- Concurrent code does: {start, do, end}
- Carefully inspect: two threads both doing {start, do, end}
- But code in a cycle!
  - ...start, do, end, stuff, start, do, end, stuff...
- Inspect: two threads both doing {end, stuff, start}
- Inspect: {start, do, end} vs {end, stuff, start}
  - (chasing each others' tail)

Printing

- Printing / Logging / Events
  - "Make noise" at each read/write of shared variable
  - Inspect trace after crash
  - Serialized results…
  - …but I/O blocks; changes timing, hides data-race bugs
  - Also OS can buffer per-thread; WYSI not WYG
- Per-Thread Printing
  - Write "event tokens" (enums) to per-thread ring-buffer
  - Per-thread buffer: No contention to write
  - Tokens: no complex String creation, no object creation
  - Ring buffer: much less overhead & no blocking
  - Less likely to hide bug

It Takes Two to Tangle...

- **Want: Who Dun It and When Dey Did It**
- Per-Thread Printing w/TimeStamp
  - Slap a System.nanoTime in the per-thread event buffer
- Post-process the crash
  - Sort all ring-buffers by nanoTime
  - Print a time-line just before the crash
  - AND after the crash
  - 99% chance the "guilty thread" stands out
- Rather heavy-weight technique
  - Need to know where to target it

Tools

- **Not Ready for Prime Time**
  - Most tools simply don't scale
  - Academic toys: 10x slowdowns, high false positive rates
  - Require PhD to use
- Recommend: FindBugs
  - Scales to production use
  - Simple pattern-matching
  - Definitely limited in scope
  - Handful young companies making a go in this space
- Visual Inspection & PrintF
  - Still State of the Practice

Agenda

- (not very) Formal Stuff
- Reordering Memory – A peek under the hood
- Common Data Races
- Finally(!) Some Debugging Techniques
- **Summary and Advice**

Writing Data Races

- Often hidden by Good Programming Practice
- Already solving a Large, Complex Problem
- Using abstraction, accessors
  - Giving meaning to memory access
  - In context of Large, Complex Problem
- Need more speed
- So introduce Concurrency, Threads
- Patterns like Double-Checked Locking
  - Or double-read w/rare null writer
  - Or forgetting that multiple calls to a thread-safe collection are ***not*** thread-safe between calls:

```
if( cache.get(Key) == null )
    cache.put(Key,compute_value()); // BUGGY!
```

The Pitfall

- End up adding Concurrency to L.C.P.
- **Fail to recognize Concurrency it's own (subtle) Complex Problem**
- Needs its own wrappers, access control
- Interviews w/Data-Race Victims:
  - Often they cannot name the shared vars
  - Don't know which thread can touch what
  - ... or when
  - Surprised by the interleaving that triggers the bug
- Fix starts with Gathering this info and asserting Access Control over shared vars

Other Techniques: Locked Collections

- Azul Flag: Lock unlocked Collections
  - Supposed to be no contention
  - Thin-lock cost is low
  - Some slowdown, but survive the race
  - (actually use Azul HTM to avoid lock overhead)
- Azul Flag: Throw exception if racing in Collection
  - Requires extra word, ½ thin-lock cost
  - Catches BOTH reader and writer
    - ... at moment of race!
- Available in JSR-166 contributions?
  - "UncontendedLock"

Other Techniques

- Formal proofs?
  - Still not ready for prime-time
  - Although hardware designers make it work for them
- Statistical
  - I get X fails/month; what happens that often?
- No good answers yet
- Best answers so far:
  - Don't Get Clever w/ Concurrency
  - Document, Document, Document

Don't Go There...

- Best answer: don't write concurrency bugs!
- Use the 'immutable' object pattern
- Use private data
- Use well-tested java.util.concurrency.*

But When You Must...

- Admit to self: ***Here Be Dragons***
- Design API around concurrent access
  - It's not a bolt-on after-the-fact check-list kind of feature
- Think Before You Write, and ...
- Document, Document, Document!

Summary

- Hardware Reorders, Compilers Reorder
  - Must control for both
  - Java beats C/C++/C#/Fortran/pthreads here
  - Weaker hardware memory models coming soon...
- Must fence in both threads
  - Error if missing either read-side fencing OR write-side fencing
- Common Errors:
  - Double-read and rare null-writer
  - Double-checked locking
  - Single-writer-many-reader in HashMap (& other unsafe collections)
- Visual Inspection, PrintF
  - It's the State of the Practice ***AND*** State of the Art – ugh!
  - But must make 'printf' very lightweight
    - Tokens (no String construction), per-thread ring buffer (no locks)

Summary

- Stick with well-tested java.lang.concurrency.*
- Document inter-thread communication
  - Read & Write of Shared Variable is communication!
- No more separate accessors for 'is_ready' and 'get'
  - Value can change between test and get
- And of course,

Document, Document, Document!

For More Information

- JVM Challenges and Directions in the Multi-Core Era - TS 6206
- Towards a Coding Style for Scalable Nonblocking Data Structures - TS 6256
- http://blogs.azulsystems.com/cliff
- http://e2e.azulsystems.com
- http://sourceforge.net/projects/high-scale-lib
- http://www.azulsystems.com

THANK YOU

Dr. Cliff Click, Distinguished Engineer

Azul Systems

http://blogs.azulsystems.com/cliff