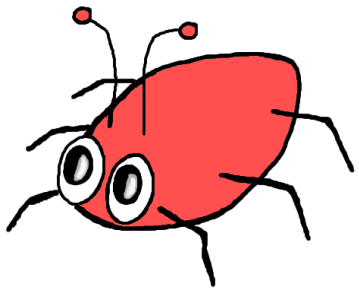
Finding Concurrency Bugs in Java

David Hovemeyer and William Pugh



July 25, 2004

Programmers are Not Scared Enough

- Java makes threaded programming "too easy"
  - Language often hides consequences of incorrect synchronization
- Many (most?) Java programmers play fast and loose with synchronization
- Result: many production concurrent Java programs have serious, avoidable concurrency bugs
  - Programmer intuition about behavior of programs with data races is almost always wrong
  - Program usually works
  - ...until deployed in a mission-critical application?

Our Work

- Develop simple, effective static analysis techniques for finding bugs
  - Including concurrency bugs
  - http://findbugs.sourceforge.net
- Idea: *bug patterns*
  - Deviations from good practice
  - Code idioms that are likely to be errors
- Analyze real applications and libraries
  - Was analysis effective at finding real errors?
  - Was the false positive rate acceptable?
  - Can we convince developers bugs are worth fixing?
  - Can we gain insight on why bugs are introduced?

Concurrency Bug Patterns

Finding and Eliminating Data Races

- Lots of techniques exist to find and eliminate data races:
  - Race-free Java dialects (sound, but restrictive)
  - Sophisticated static analysis (interprocedural, context-sensitive)
  - Dynamic techniques
- How about simple techniques?
  - Java programs usually have relatively simple concurrency patterns
  - Look for violations of most common synchronization idiom
  - Can we find real bugs?

Inconsistent Synchronization

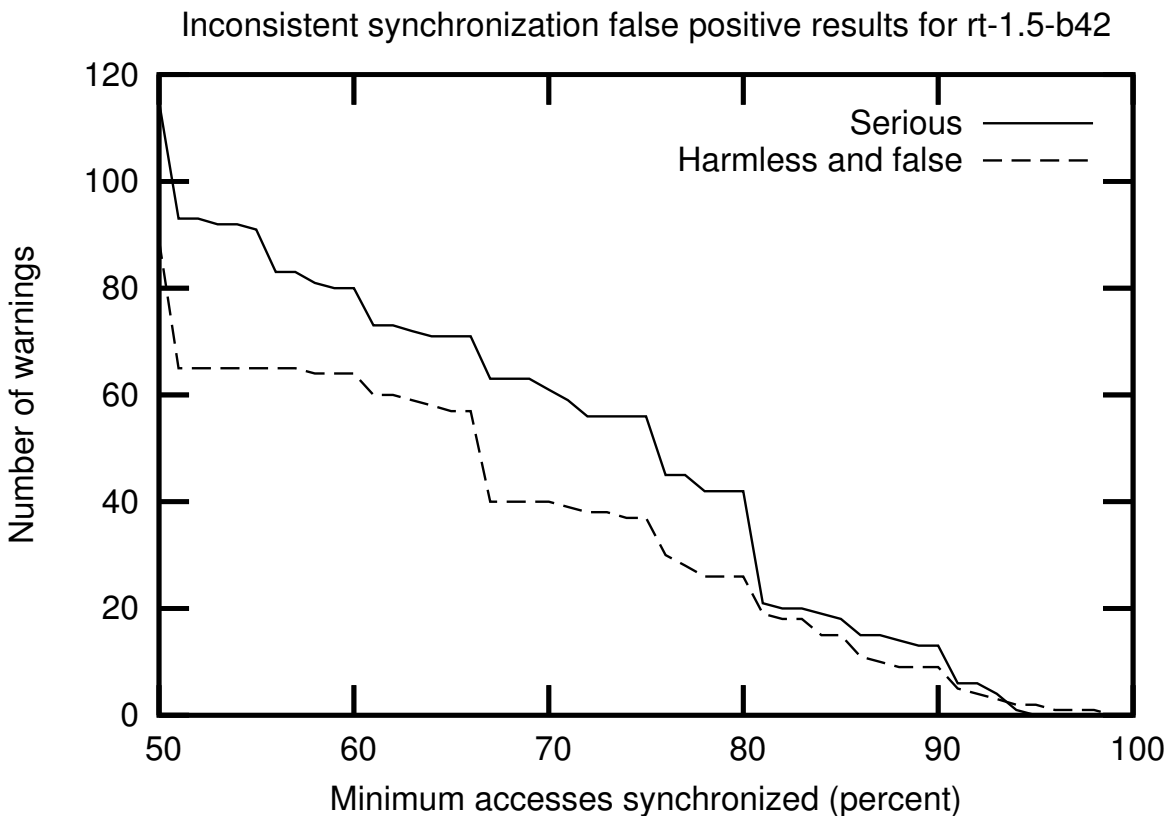
- Common idiom: synchronize on `this` reference
- Track scope of locks intraprocedurally, examine field accesses
  - Ignore accesses in...
  - non-public methods called only from locked contexts
  - methods not likely to be reachable from multiple threads: constructors, finalizers, `readObject()`, etc.
- Report fields where accesses are usually, but not always, synchronized
- Result: this technique finds lots of data races
  - 114 we verified in core J2SE libraries (JDK 1.5, build 42)
  - 2 found in prerelease version of JDK 1.4.2, fixed by Sun in JDK 1.5

Ranking Warnings: The Hypothesis

- We believed that programmers would strive to synchronize all mutable field accesses for objects intended to be thread-safe
- Therefore:
  - The higher the percentage of synchronized accesses,
  - The more likely unsynchronized accesses would indicate genuine bugs
- So, we gave higher priority to warnings of fields synchronized between 75% and 99% of the time

Ranking Warnings: The Reality

- The hypothesis was incorrect:



Interpretation

- Fields synchronized, e.g., 95% of the time not significantly more likely to be bugs than fields synchronized 50% of the time
  - Programmers are deliberately using race conditions to communicate values between threads
- Some examples of unsynchronized accesses:
  - Set methods (very common)
  - Get methods (very common)
  - Copying internal collection to an array
- Programs work "most of the time"
  - Many bugs may not be exploitable in practice
  - Still, not a comforting thought

Double Checked Locking

- A common technique to avoid locking in lazy initialization of a singleton:

```
if (singleton == null) {
  synchronized (lock) {
    if (singleton == null)
      singleton = new Singleton();
  }
}
```

- JVM can reorder writes!
  - Without acquiring lock, may see incompletely initialized object
- Still widely used
  - We found 78 doublecheck instances in core J2SE libraries
  - And 4 doublecheck instances in JBoss

Detecting Double Checked Locking

- State machine driven pattern recognition over bytecode
  - Bytecode closely matches source
- Look for:
  1. Load of field
  2. Null comparison
  3. Monitorenter
  4. Load of field
  5. Null comparison
  6. Object creation, Store to field

Unconditional Wait

- Triggered when monitor wait is done immediately upon entering a synchronized block:
  1. monitorenter
  2. invokevirtual Object.wait()
- Often means condition was checked without the lock held
- Usually a novice thread programmer error, but...
- 2 occurrences in JBoss!

Example

- Example (JBoss 4.0.0 DR3)

```
// If we are not enabled, then wait
if (!enabled) {
  try {
    log.debug("Disabled, waiting for notification");
    synchronized (lock) {
      lock.wait();
    }
  }
  catch (InterruptedException ignore) {}
}
```

Other Bug Patterns

- Some concurrency bug patterns more useful for finding mistakes in novice code:
- Wait Not In Loop
  - Monitor waits must be in a loop which checks the condition
    - Other threads can run between wakeup and reacquiring lock
    - Java allows spurious wakeups
    - Monitors used for multiple conditions
- Two Lock Wait
  - Waiting with two locks creates possibility of deadlock
  - Found bug in J2SE CORBA ORB implementation
- More patterns described in paper

Conclusions

Conclusions

- Trivial static inspection reveals a large number of concurrency bugs in widely-used applications and libraries
- Why are these bugs there?
  - Benign reason: everyone makes mistakes
  - Sinister reason: programmers are too willing to take chances
- Many bug patterns can be easily automated
  - With tuning, false positive rate is acceptable (usually less than 50%)

Recommendations

- Once introduced, bugs are difficult and expensive to fix
  - Especially true of concurrency bugs, where reproducibility is low and likelihood of introducing other errors is high
- We should make early detection tools (static and dynamic) easy to use
  - Tools that don't require developers to change working style more likely to be adopted in practice
- Simplicity helps:
  - If analysis is simple, it's usually easy to explain results to user
  - It makes sense to fix obvious bugs before tackling subtle bugs