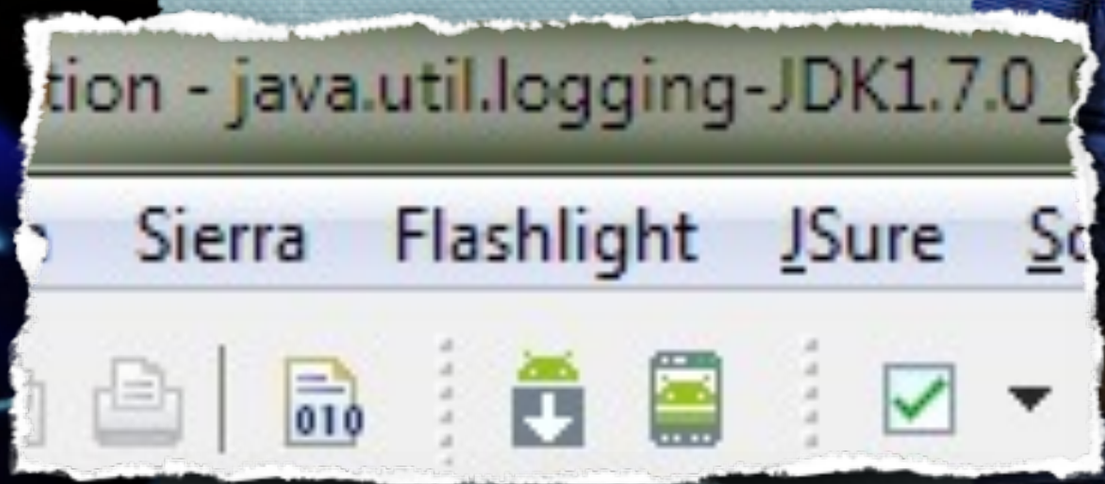


Concurrency-Focused Dynamic Analysis

The perils of ARM and the possibilities for safe/secure mobile applications

Tim Halloran, SureLogic, Inc

Fault diagnosis and verification for safe concurrency

- The role and value of *dynamic analysis*
  - How can data from one program run tell you anything general?
- One run can yield broadly useful modeling information
  - *Unsafe sharing of state*: failure to respect memory model
  - *Safe sharing of state*: use of locks, safe publication, etc.
  - *Performance*: blocking latencies
  - *Potential for deadlock*: lock order anomalies
  - (This builds on novel dynamic techniques, to be described)
- Dynamic results assist model development for sound static analysis
  - (Heritage in CMU Fluid project)
  - Analysis-based verification using sound static composable analyses
  - Minimal explicit models to specify developer intent
  - Example studies: Hadoop concurrency, J.U.C, Accumulo, many Java libs

This talk

- Java memory model (JMM) for explicit concurrency
  - Hardware realities
  - An infringement on source code
  - Accidents and surprises: JVM/x86 and Dalvik/ARM
- Assisting developers and evaluators
  - Structures, models, analysis, tools
- Using concurrency-focused dynamic analysis
  - Collection and querying
  - Interplay with analysis-based verification
  - Performance
- Building an effective and usable tool — some tricks

Why a memory model?
State shared by multiple threads

- What is actually going on:
  - Memory hierarchy
  - Compiler reordering
  - Pipeline reordering and parallel execution
  - Speculative fetching
- Code needs explicit *fences* or *memory barriers*
  - Many kinds: LS, SS, etc.
  - Memory scope of fence
- Developers must respect hardware “rules of the road”

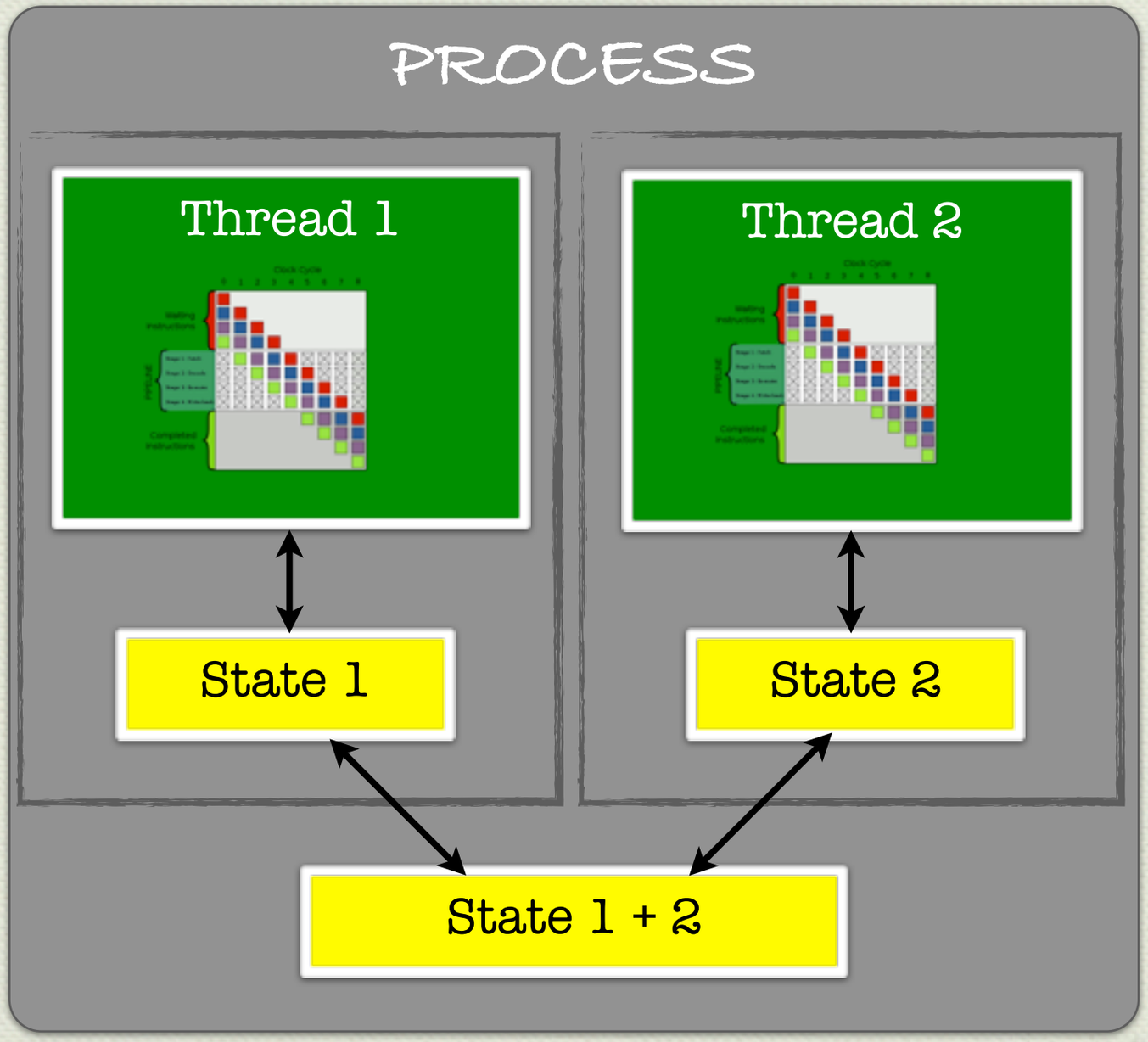


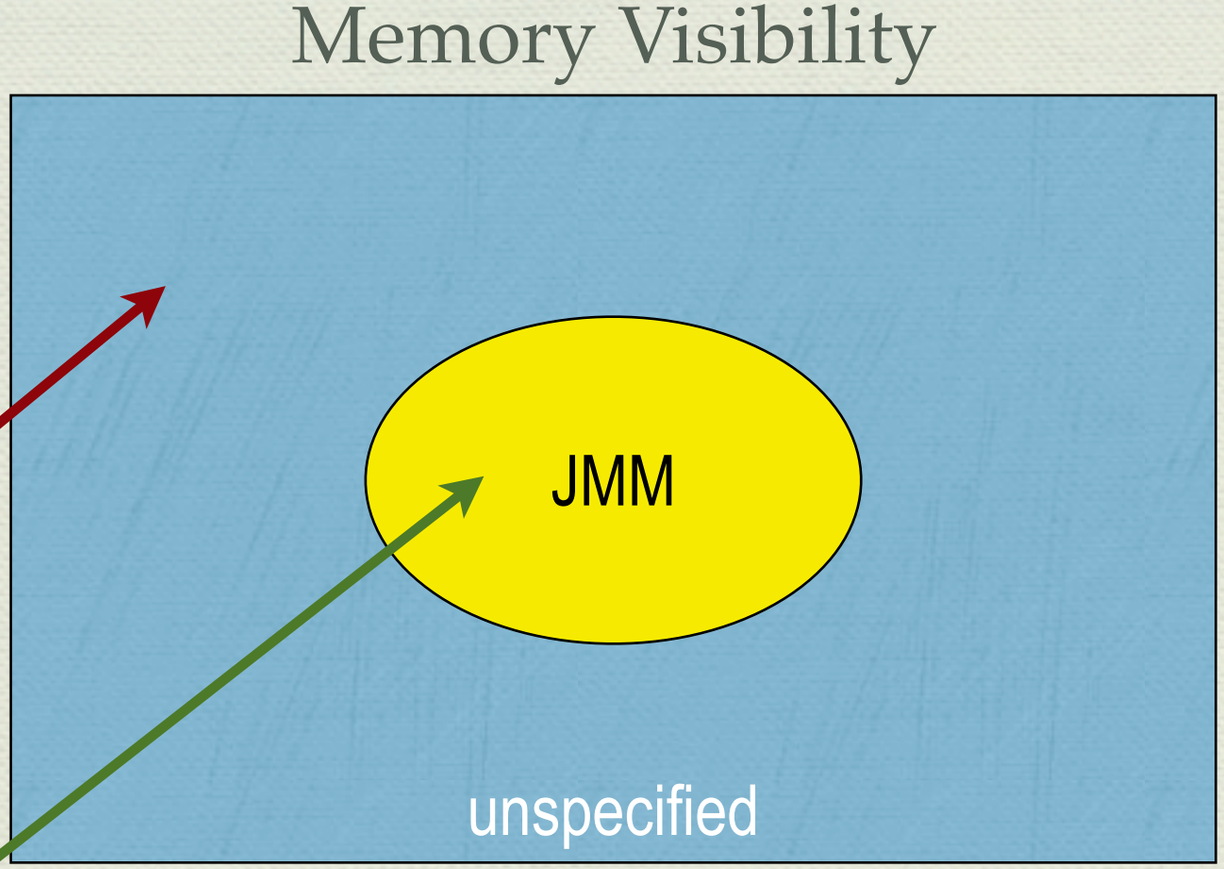
Figure thanks to Bill Scherlis

Abstracting the fence – The Java memory model (JMM)

- From sequential consistency to "**relaxed consistency**"
  - Why? Performance
  - Remedy: Issue fences
- When and how many fences?
  - Lock (monitor) use, volatile field access, and thread start/termination
  - These create happens-before relationships

Required barriers	*2nd operation*			
1st operation	Normal Load	Normal Store	Volatile Load MonitorEnter	Volatile Store MonitorExit
Normal Load				LoadStore
Normal Store				StoreStore
Volatile Load MonitorEnter	LoadLoad	LoadStore	LoadLoad	LoadStore
Volatile Store MonitorExit			StoreLoad	StoreStore

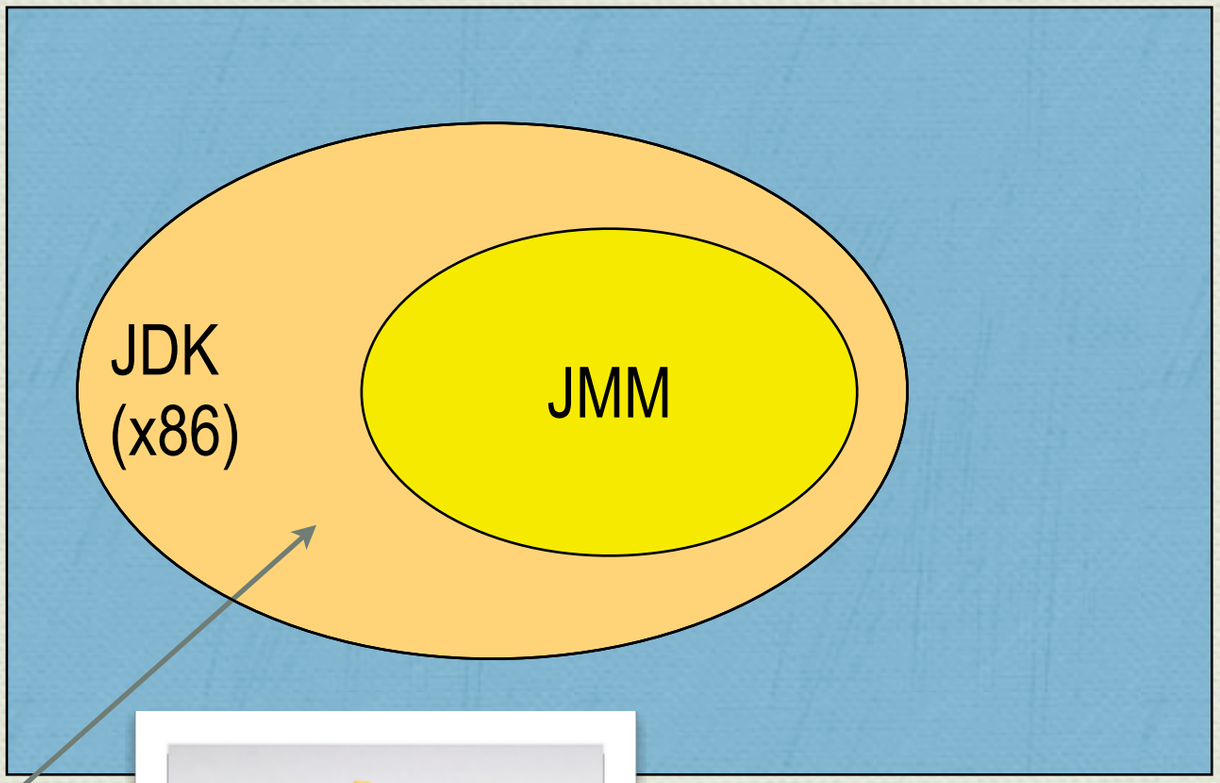
From Doug Lea's JSR-133 Cookbook for Compiler Writers



Nonempty means issue a fence instruction (done by compiler/JVM/JIT)

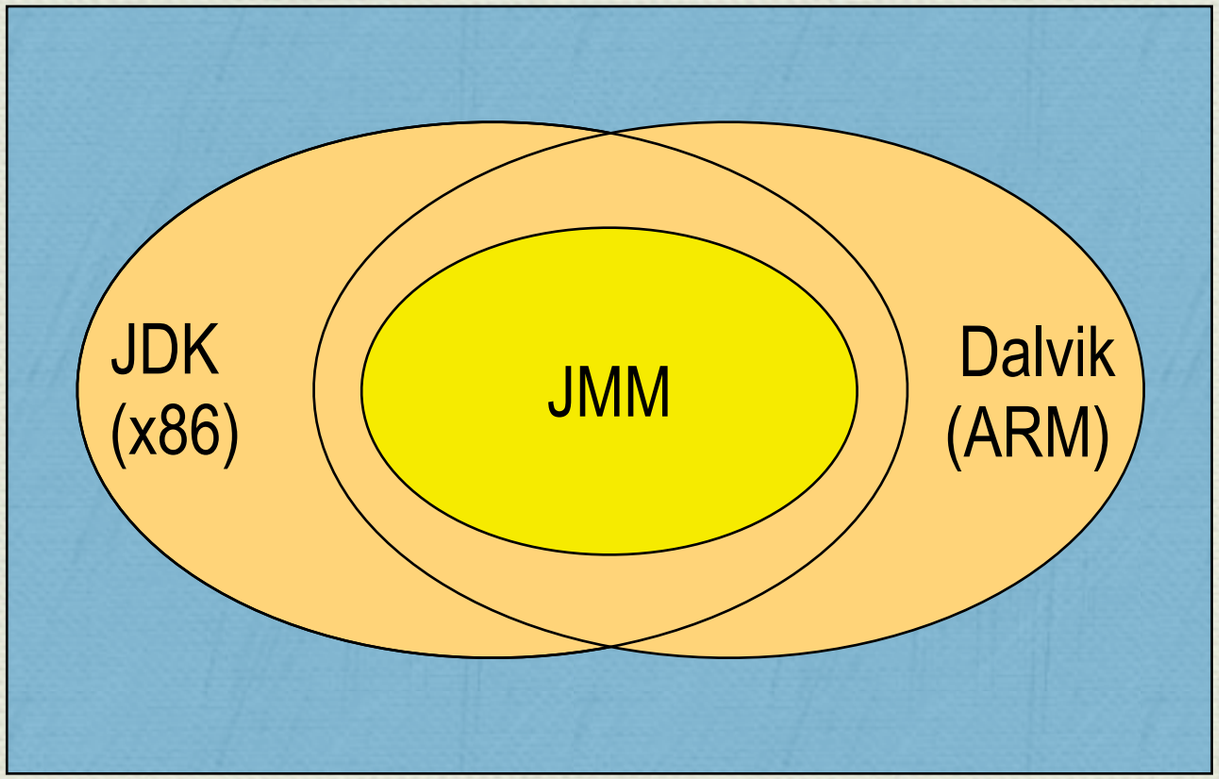
JVM/x86 platform memory model

- The Java platform on Intel x86
  - HotSpot, OpenJDK, IBM J9
- This memory model is more conservative than the JMM
  - Why? Hardware guarantees, engineering choices, etc.
- Is this a problem?
  - No, if we stick to the JMM
  - Yes, if we "run with scissors" in this region of the platform's behavior

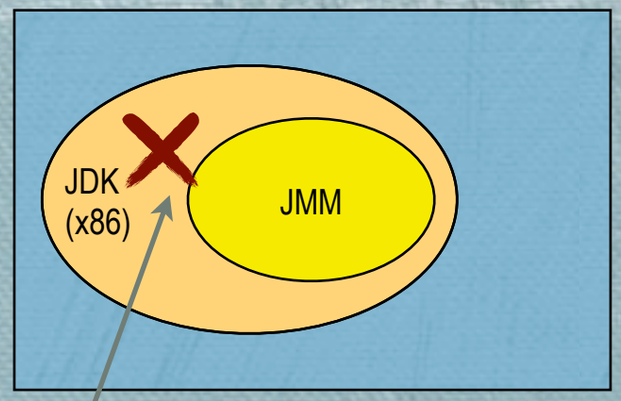


A new (and different) memory model implementation: Dalvik/ARM

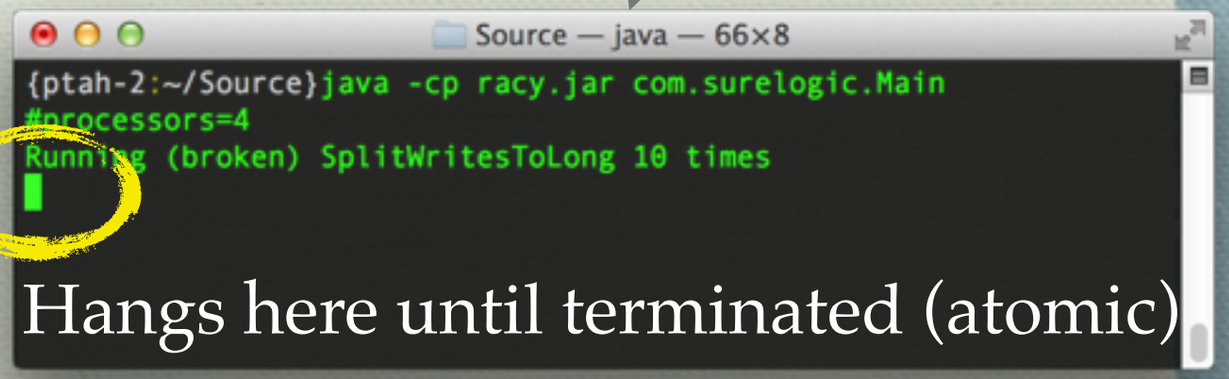
- The Android platform on ARM
  - Uses a different JVM, Dalvik, and byte code format (Dex)
  - Source code is Java
    - Different standard libs
- This memory model is more conservative than the JMM
  - Differs from Java on Intel
  - Why? Different engineering goals for memory and power use, etc.



Split writes of 64-bit fields on JDK/x86



- The JMM recommends that writes to 64-bit values be atomic—but they may be split




```
long value; // shared

// loop in one thread:
value = 0L;  // Broken!

// loop in another thread:
value = -1L; // Broken!

// loop in yet another thread:
if (value != 0L && value != -1L) {
  // output unexpected value and exit
}
```

0L = x00000000_00000000
-1L = xFFFFFFFF_FFFFFFFF

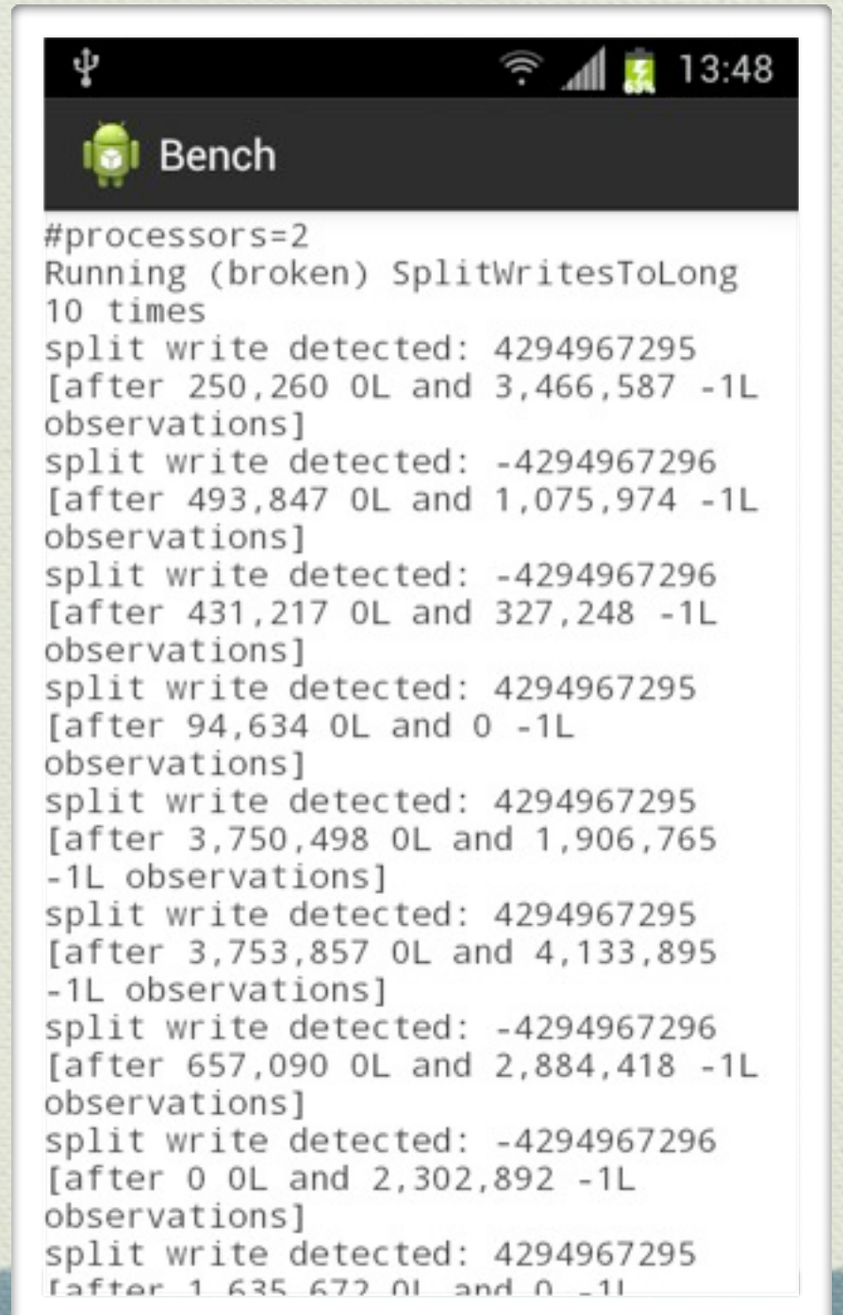
If writes are split the JMM mandates that two atomic 32-bit writes be done so the two possible unexpected values we may see are:

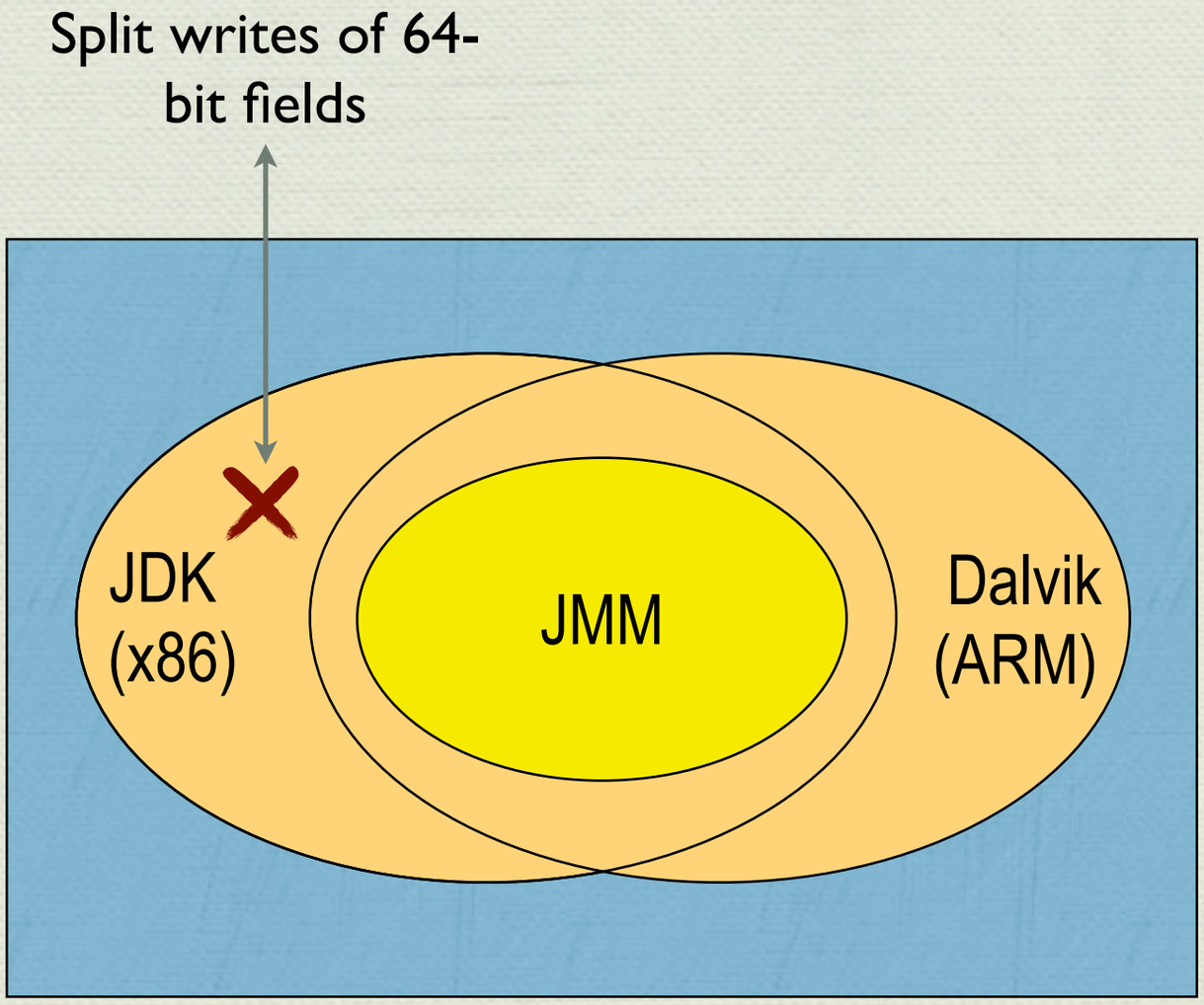
-4294967296 (xFFFFFFFF_00000000)
4294967295 (x00000000_FFFFFFFF)

*There are cases where split writes occur on 32-bit OpenJDK/Hotspot (versus 64-bit)

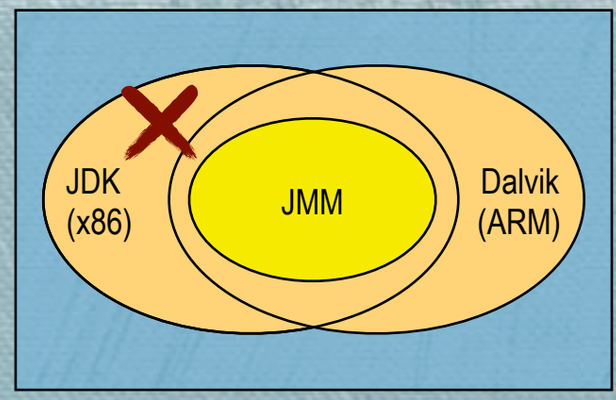
Split writes of 64-bit fields on Dalvik/ARM

Fails quickly





Split writes of 64-bit fields in real world "working" code



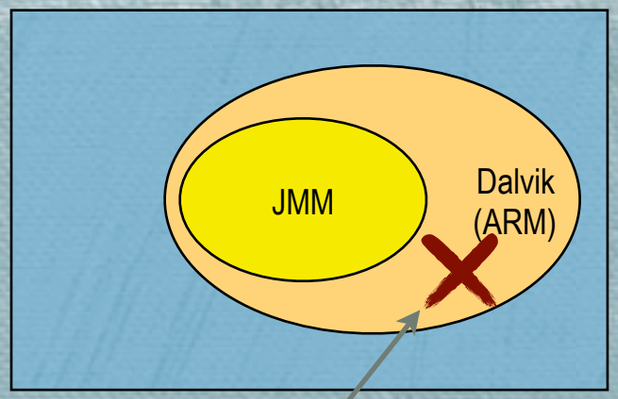
- A bug in an early version of java.util.concurrent concurrency library
  - (Found using SureLogic's JSure verification tool by Greenhouse)
  - Fixed by Doug Lea
  - This class became **AtomicLong** in the Java standard library and is on Android

```
public class SynchronizedLong extends SynchronizedVariable implements
    Comparable, Cloneable {

  protected long value_;
```

```
public long swap(SynchronizedLong other) {
  if (other != this) {
    SynchronizedLong fst = this;
    SynchronizedLong snd = other;
    if (System.identityHashCode(fst) > System.identityHashCode(snd))
      fst = other;
      snd = this;
    }
    synchronized (fst.lock_) {
      synchronized (snd.lock_) {
        fst.set(snd.set(fst.get()));
      }
    }
  }
  return value_;
}
```

This read of the shared long `value_` field is not protected by `lock_` and it could observe a split value (and return it)

Non-volatile boolean flag on Dalvik/ARM



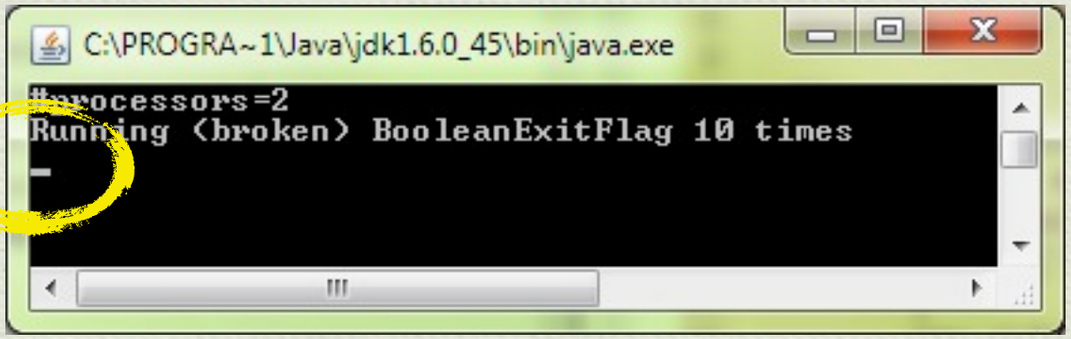
- A boolean set in one thread is used to signal another thread that it should perform some action (e.g., exit cleanly)

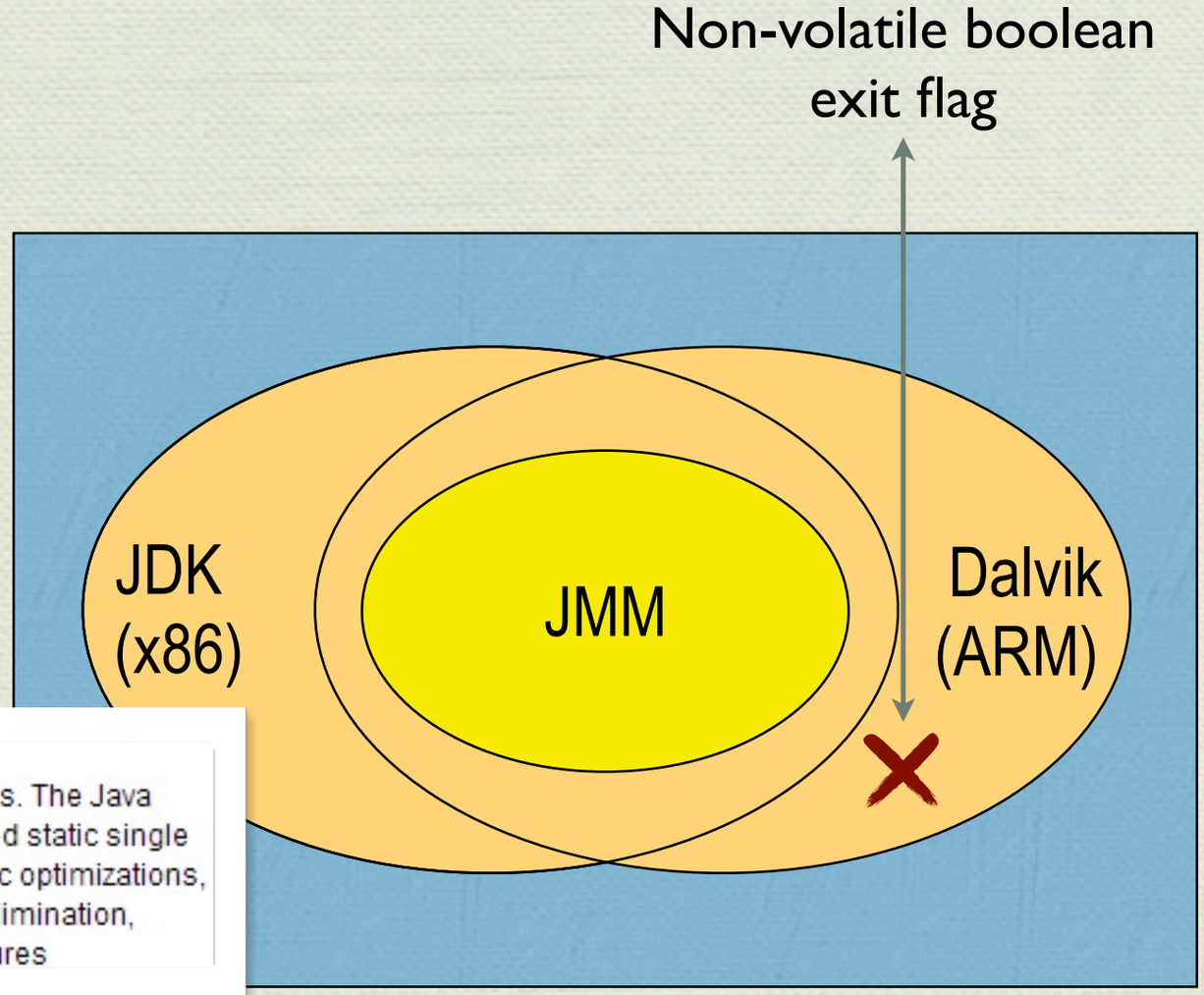
```
boolean flag; // shared

// in one thread:
public void run() {
  while (!flag) { // Broken!
    // do some work
  }
}

// (later) in another thread:
flag = true;
```

- Reliable and quick

Non-volatile boolean flag on JVM/x86

Thread hangs forever





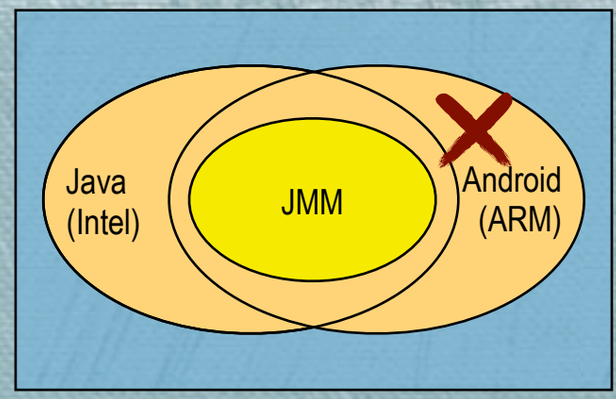
- If the field is not volatile then the JIT hoists the field

Java HotSpot Server Compiler

The server compiler is tuned for the performance profile of typical server applications. The Java HotSpot Server Compiler is a high-and fully optimizing compiler. It uses an advanced static single assignment (SSA)-based IR for optimizations. The optimizer performs all the classic optimizations, including dead code elimination, loop invariant hoisting, common subexpression elimination, constant propagation, global value numbering, and global code motion. It also features

http://www.oracle.com/technetwork/java/whitepaper-135217.html#hotspot

Non-volatile boolean flag in real world "working" code

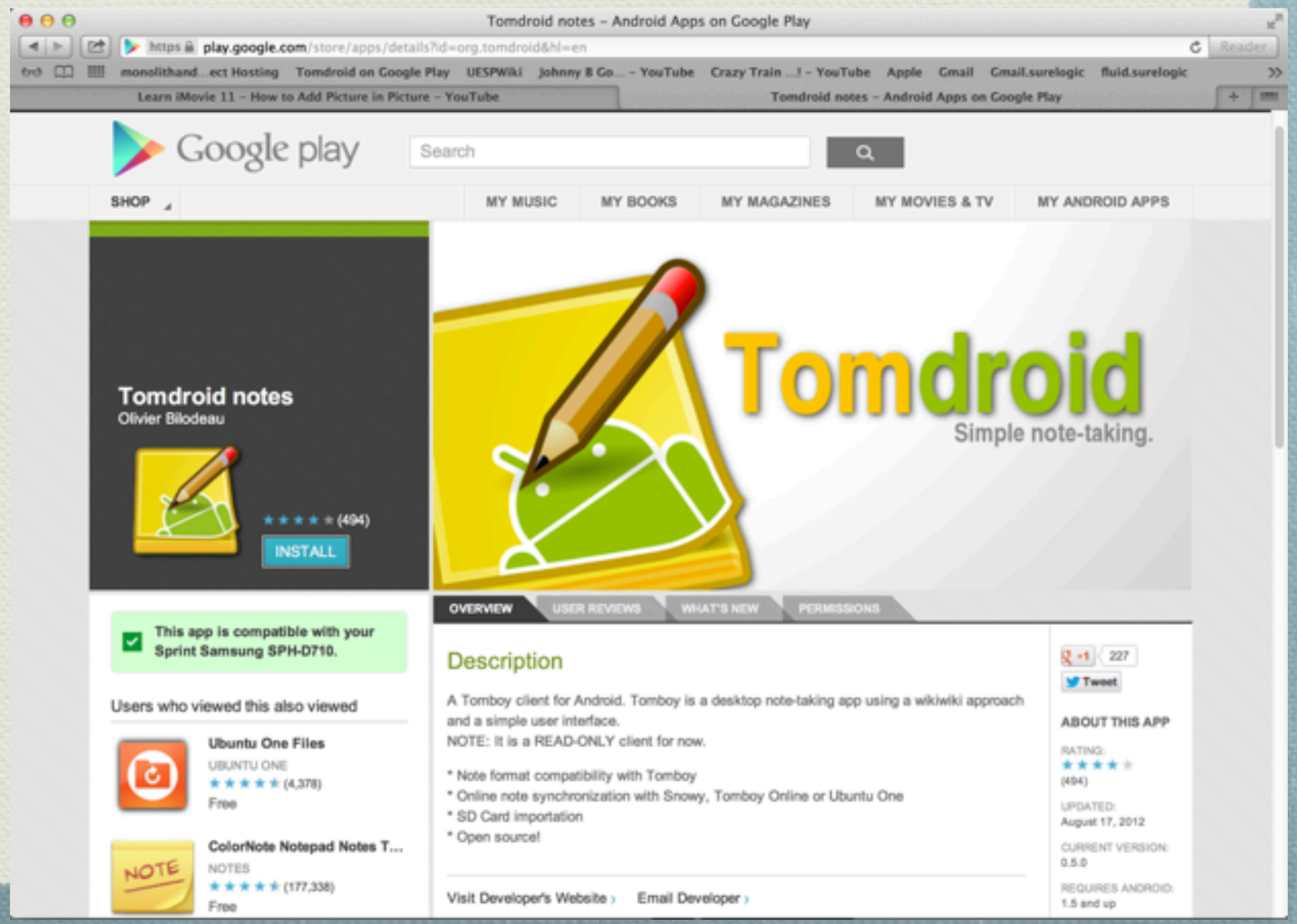


- A bug in the TomDroid notes-taking Android application (50K installs)
  - (Found using SureLogic's Flashlight dynamic analysis tool by Boy)
  - Not yet fixed

```
public abstract class SyncService {
```

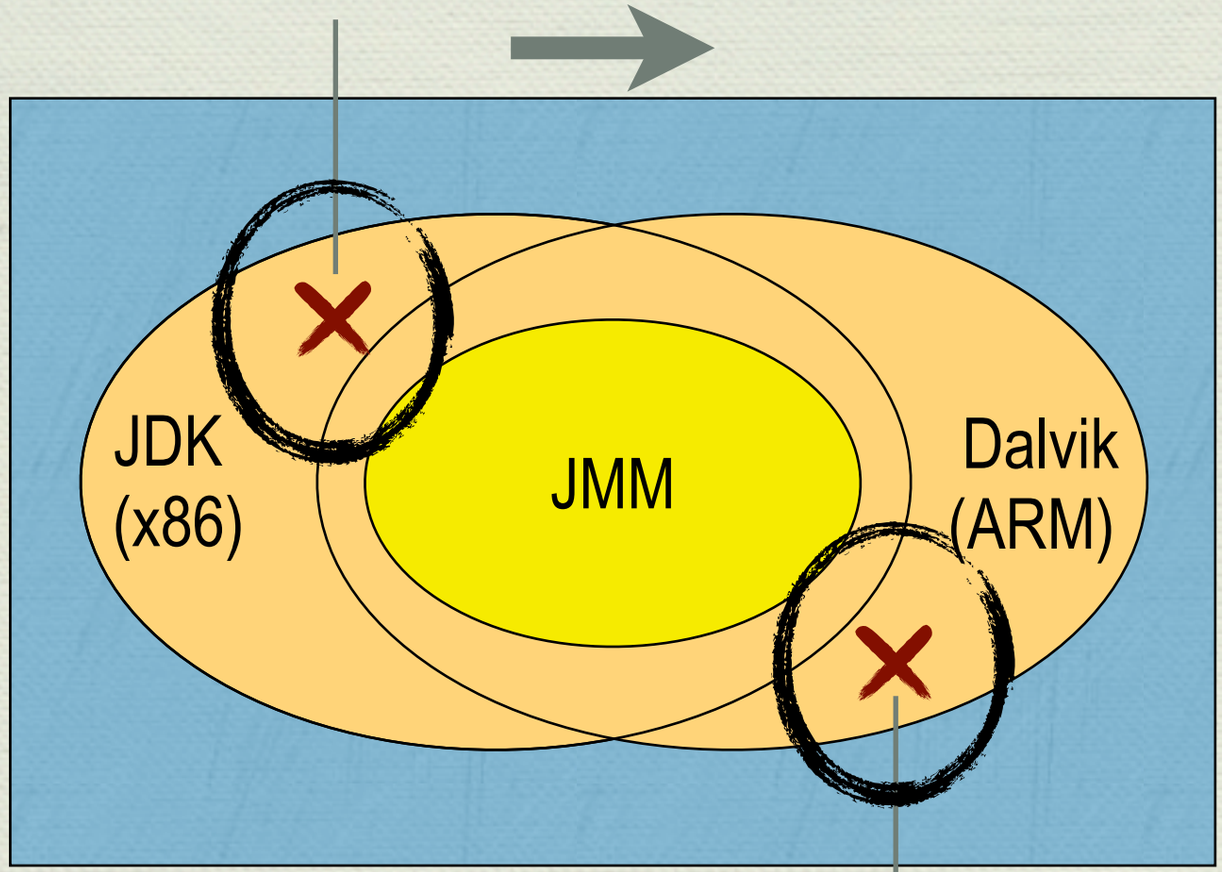
```
public boolean cancelled = false;
```

The shared `cancelled` flag, used to cancel synchronization of notes with a server, is not volatile and may not be seen between the UI thread and the background sync service task



There is danger outside the JMM

"Working code" breaks when moved
from JDK/x86 to Dalvik/ARM

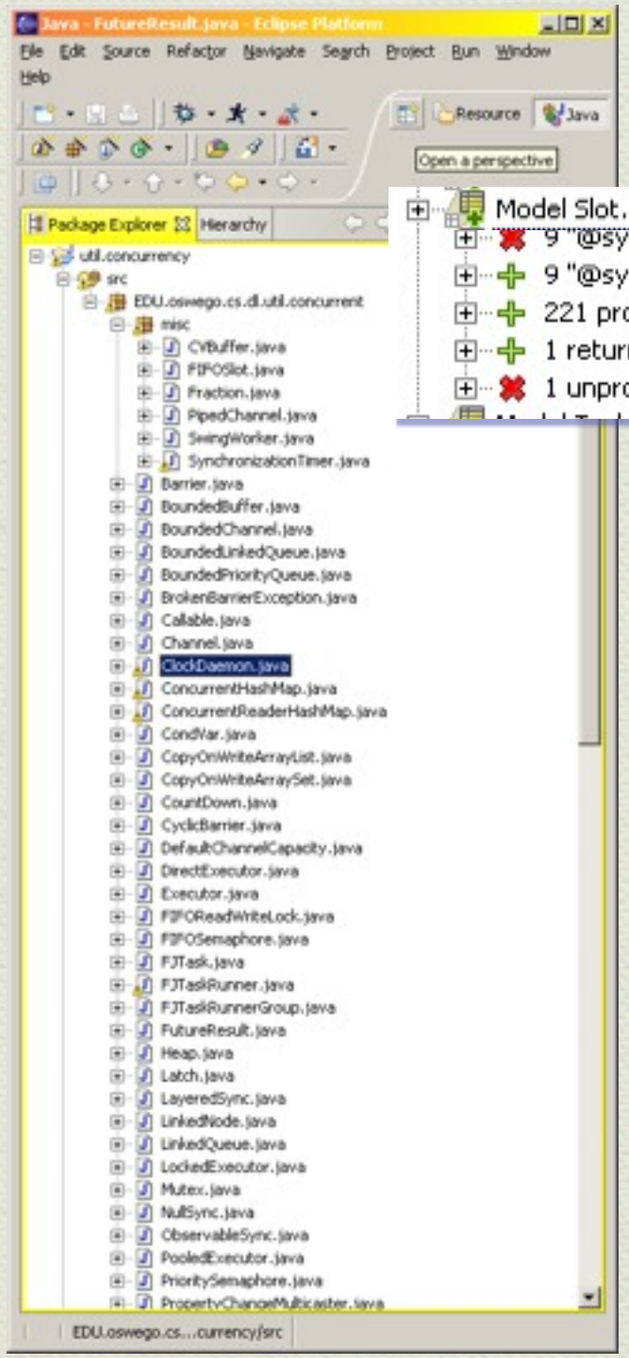


"Working code" breaks when moving
from Dalvik/ARM to JDK/x86

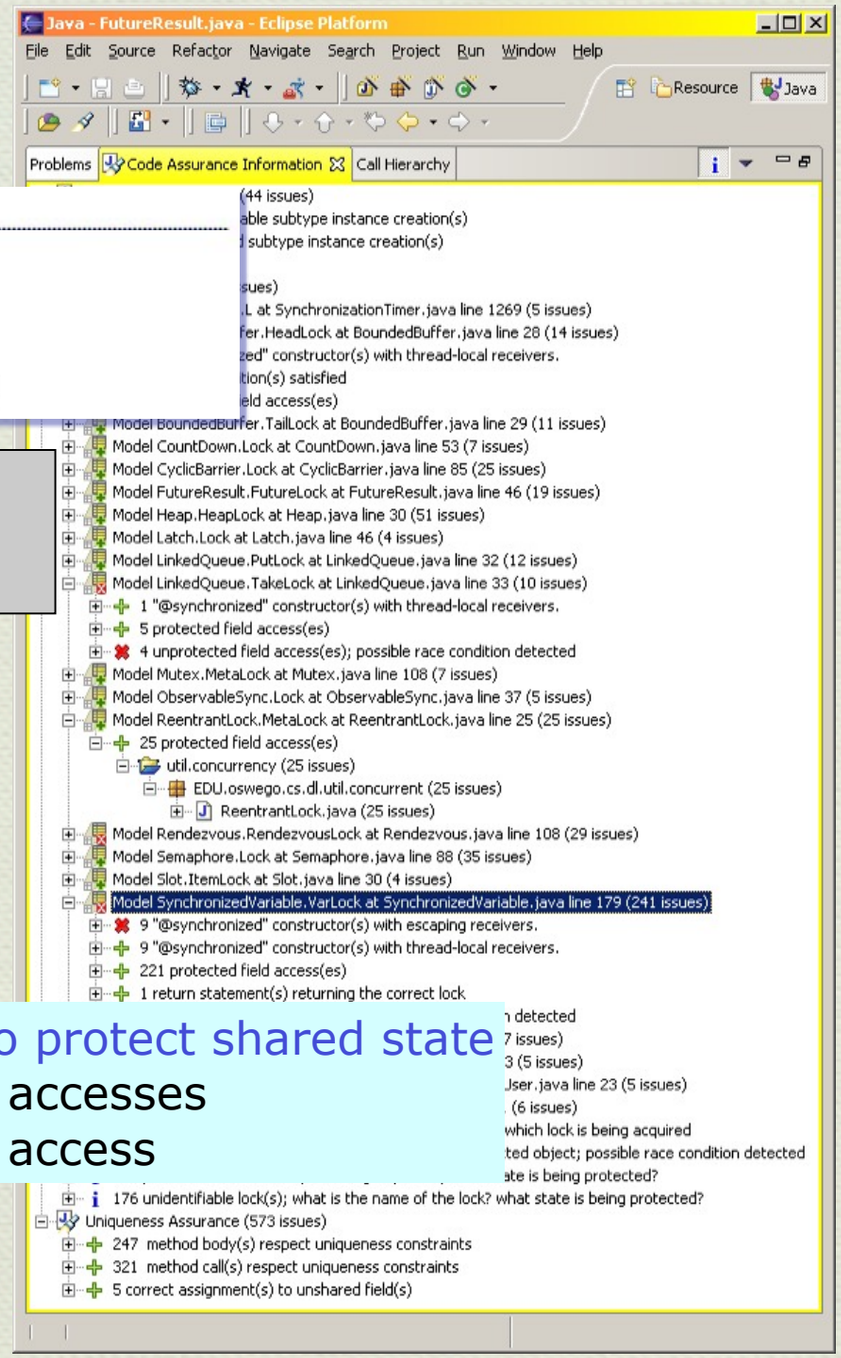
How does the developer handle this?

- Answer 1 – Forget Java! (explicit concurrency) — if you can
  - *Actually*: Essential complexity in languages w/explicit concurrency
- Answer 2 – Test a lot, on multiple platforms
  - *Actually*: Non-determinism (1 in 1m) means less useful
  - *Actually*: "Success" can lead to "running with scissors"
  - *Actually*: When flaws are detected, diagnosis may be hard
- Answer 3 – Outsource concurrency to libraries and frameworks
  - *Actually*: We are doing this
    - *But*: its only partial, and the frameworks and libraries have problems themselves
- Answer 4 – Analysis-based verification (ABV)
  - *Actually*: Starting to emerge into practice

ABV Example: Verification for util.concurrent



- Visual assurance indicators
- Textual warnings
- Drill down analyses



Lock *VarLock* used to protect shared state

- 221 protected accesses
- 1 unprotected access

ABV Example: Analysis-Based Verification for **Hadoop** MapReduce infrastructure

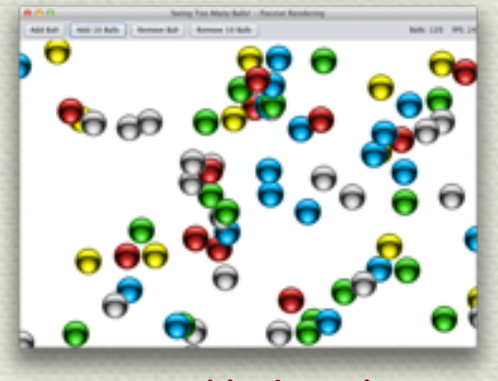
- Difficulties identified
  - State inconsistency
  - Unsafe practices
  - Data exposures
- Assurance given
  - Specific areas of consistency of code with identified intent

Contents

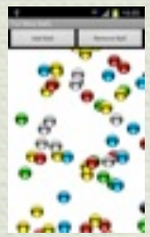
Why dynamic analysis in this non-deterministic setting?

- Helps understand large systems and build models
  - Global program properties: deadlock, JMM
  - Gateway to verification — help developers model intent
- Familiar approach to developers (debuggers, profilers, etc.)
  - Low adoption cost
- Performance analysis a challenge
  - E.g., false sharing, lock contention
- Visualize exactly where “bad things” could happen
  - Don’t actually need the race/deadlock to happen

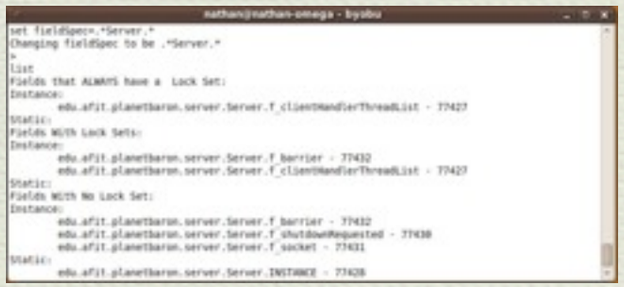
Flashlight concurrency-focused dynamic analysis tool



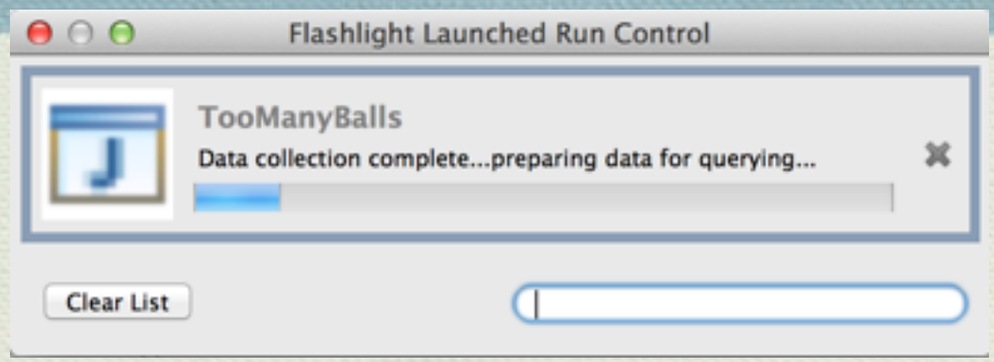
or



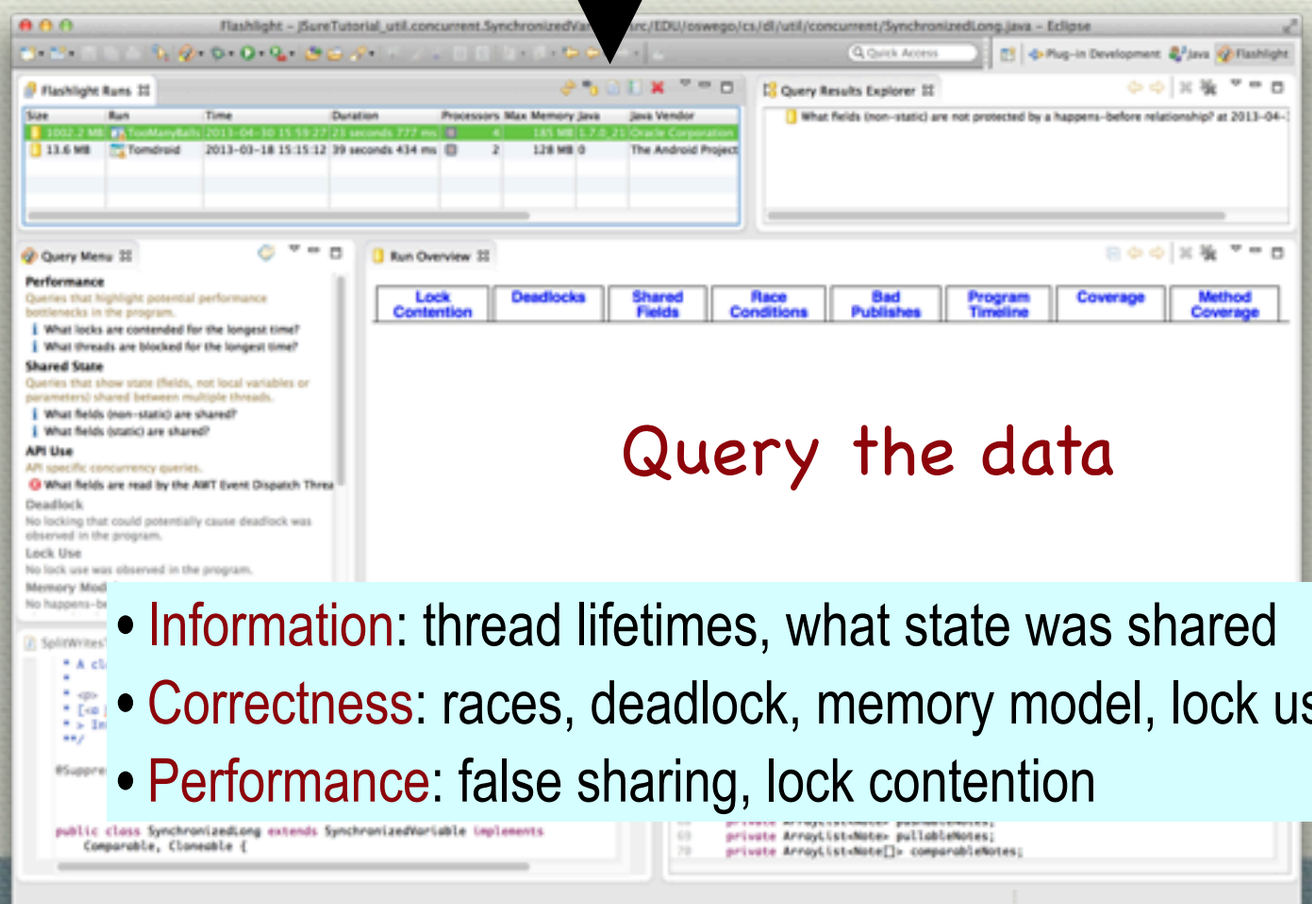
Full instrumentation
(in development and evaluation)



Low-overhead monitoring
(in operations)



Prepare the collected data



Query the data

- Information: thread lifetimes, what state was shared
- Correctness: races, deadlock, memory model, lock use
- Performance: false sharing, lock contention

Lockset query on util.concurrent bug

Query Menu

Deadlock

Queries that show lock use that may cause the program to deadlock.

- Where does a thread hold a lock and acquire another?

Lock Use

Queries about lock use in the program.

- What are the lock edges in this program?
- What fields (non-static) have an empty lock set after object construction?

Memory Model

Queries that show happens-before relationships between threads defined by the Java memory model.

- What fields (non-static) are protected by a happens-before relationship?

Query Results

What fields (non-static) have an empty lock set after object construction? at 2013-05-06 17:38:34

Package/Class/Field	Instances
EDU.oswego.cs.dl.util.concurrent	
SynchronizedLong	
value_	2

What instances of this field have an empty lock set?

Query Results

What instances of this field have an empty lock set? at 2013-05-06 17:40:47

value_ Object	Reads	Writes	Reads Under Construction	Writes Under Construction
SynchronizedLong-6	8	6	0	1
SynchronizedLong-7	12	6	0	1

How often is a lock held when this field is accessed after object construction?

Query Results

How often is a lock held when this field is accessed after object construction? at 2013-05-06 17:40:54

Lock	Times Acquired	value_ Access Count	Percentage Held
SynchronizedLong-6	12	17	70
SynchronizedLong-7	15	17	88

- What fields (non-static) are protected by this lock, and how often?
- What fields (static) are protected by this lock, and how often?
- What objects are protected by this lock, and how often?
- Where are fields accessed while this lock is held?
- Where is this field accessed while this lock is held?
- Where is this field accessed while this lock is not held?

Query Results

Where is this field accessed while this lock is not held? at 2013-05-06 17:41:56

Package/Class/Line
EDU.oswego.cs.dl.util.concurrent
SynchronizedLong
36
107

SynchronizedLong.java

```
public long swap(SynchronizedLong other) {
  if (other != this) {
    SynchronizedLong fst = this;
    SynchronizedLong snd = other;
    if (System.identityHashCode(fst) > System.identityHashCode(snd)) {
      fst = other;
      snd = this;
    }
    synchronized (fst.lock_) {
      synchronized (snd.lock_) {
        fst.set(snd.set(fst.get()));
      }
    }
  }
  return value_;
}

/**
```

Historical Source Snapshot

JSureTutorial_util.concurrent.SynchronizedVariable/src/EDU/oswego/cs/dl/util/concurrent/SynchronizedLong.java

```
93  public long swap(SynchronizedLong other) {
94    if (other != this) {
95      SynchronizedLong fst = this;
96      SynchronizedLong snd = other;
97      if (System.identityHashCode(fst) > System.identityHashCode(snd)) {
98        fst = other;
99        snd = this;
100     }
101     synchronized (fst.lock_) {
102       synchronized (snd.lock_) {
103         fst.set(snd.set(fst.get()));
104       }
105     }
106   }
107   return value_;
108 }
```

Happens-before query on TomDroid bug

Memory Model

Queries that show happens-before relationships between threads defined by the Java memory model.

- What fields (non-static) are protected by a happens-before relationship?
- What fields (non-static) are not protected by a happens-before relationship?
- What fields (static) are not protected by a happens-before relationship?

What fields (non-static) are not protected by a happens-before relationship? at 2013-05-06 17:59:01

Package/Class/Field	Count
android.os	
Message	
arg1	9
what	13
org.tomdroid.sync	
SyncService	
cancelled	1
syncErrors	1
syncProgress	1

Historical Source Snapshot

Tomdroid/src/org/tomdroid/sync/SyncService.java

```
59      * and sent to the main UI along with the PARSING_COMPLETE message.
60      */
61     private ErrorList syncErrors;
62     private int syncProgress = 100;
63
64     public boolean cancelled = false;
65
66     // syncing arrays
67     private ArrayList<String> remoteGuids;
68     private ArrayList<Note> pushableNotes;
```

Query Results

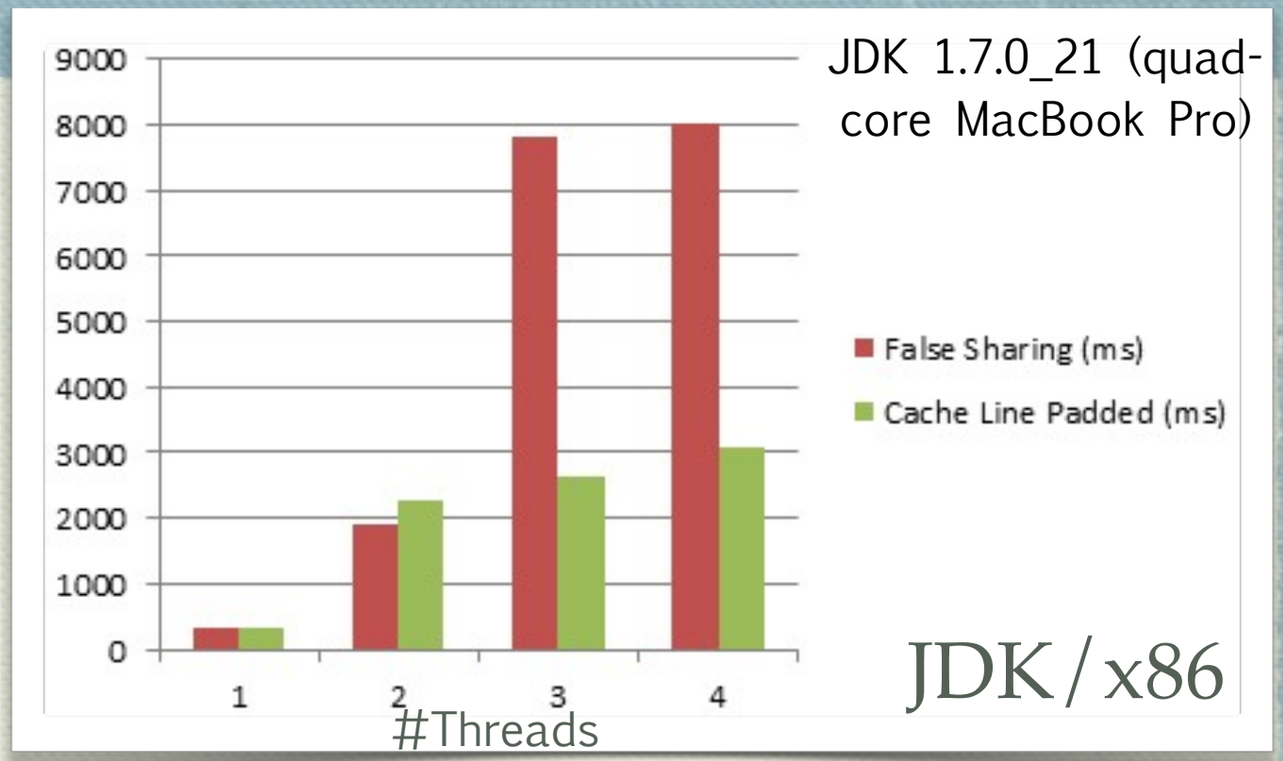
When and by what threads was this field accessed? at 2013-05-06 18:04:01

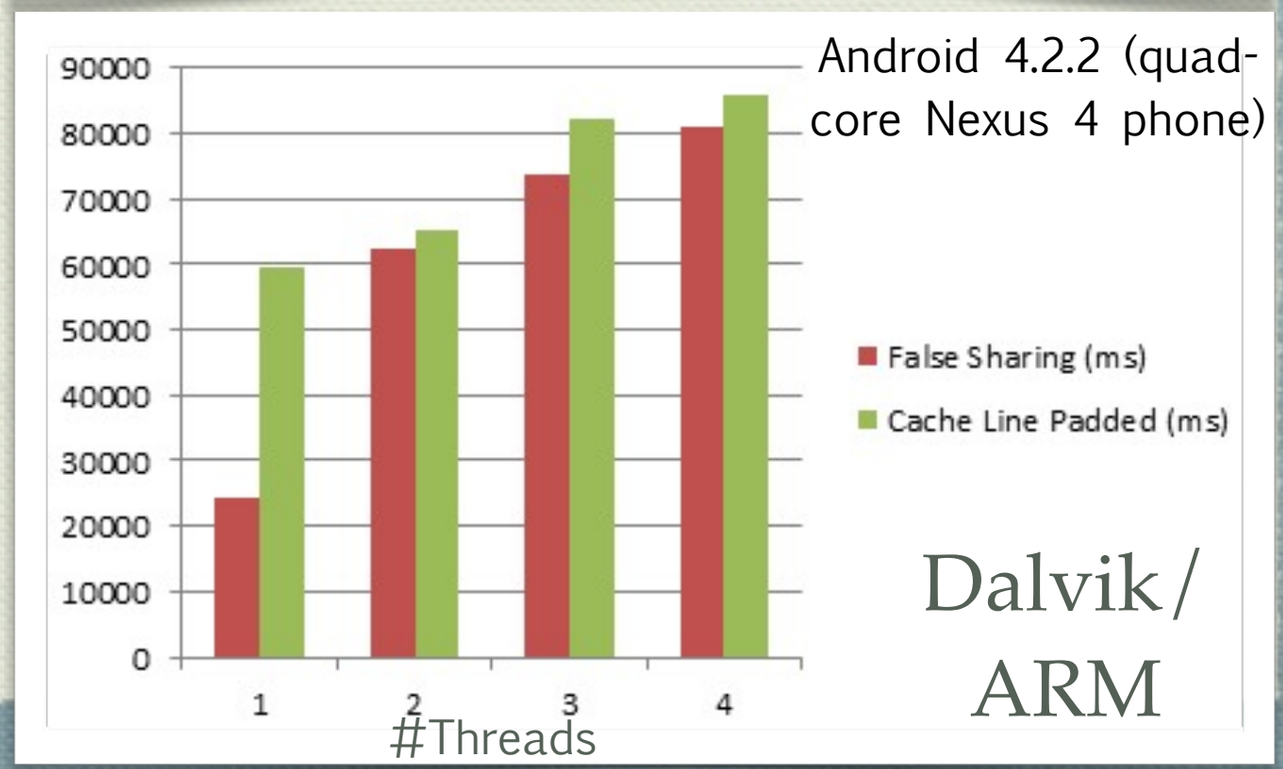
Package/Class/Field/Thread	Happens-Before	Reads	Writes	Reads Under Construction	Writes Under Construction	Start
org.tomdroid.sync						
SyncService						
cancelled						
main		0	3	0	1	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03
main	Yes	0	5	0	0	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03
main	Yes	0	2	0	0	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03
main	Yes	0	1	0	0	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03
main	Yes	0	2	0	0	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03
main	Yes	0	1	0	0	2013-03
pool-17-thread-1	No	1	0	0	0	2013-03

Performance: false sharing

```
public class Store {
   int t1Counter; t1
t2 int t2Counter;
   int t3Counter; t3
t4 int t4Counter;
}
```

- State used in different threads shares a cache line
  - Performance killer for x86
  - But not for ARM
- Hot topic in Java community
  - Padding declarations is a workaround
  - But may slow Android Apps





False sharing query

Performance

Queries that highlight potential performance bottlenecks in the program.

- What objects have the potential for false sharing?
- What threads are blocked for the longest time?

Query Results

What objects have the potential for false sharing? at 2013-05-07 21:44:25

Package/Class/Object/Field/Thread	Reads	Writes	Interleaving %	Start	Stop
com.surelogic.bench.runs					
Store					
Store-11					
t1Counter					
Store$C1-17	0	50	22.00	2056-09-11 23:27:27.078033	2056-09-11 23:27:27.080844
t2Counter					
Store$C2-19	0	50	68.00	2056-09-11 23:27:27.080208	2056-09-11 23:27:27.083288
t3Counter					
Store$C3-21	0	50	66.00	2056-09-11 23:27:27.081073	2056-09-11 23:27:27.085967
t4Counter					
Store$C4-23	0	50	88.00	2056-09-11 23:27:27.082156	2056-09-11 23:27:27.08504

Tricks of the concurrency-focused dynamic analysis trade

- Refactor Java byte code — No JVMTI
  - Enables a range: complete to lightweight selective monitoring
  - Support undocumented JIT patterns — track timing & performance
- Interact with GC in the JVM
  - Filters out thread-confined objects — key to scale-up
- JMM monitoring
  - Based on extensive and flexible monitoring of the libraries
- General query system based on extended SQL
  - Flexible support for interactive tree tables and query “drill-down”
- Correct support at the edges
  - Start-up and tear-down — surprising subtle
  - Can start/stop instrumentation separately from the app
- Android support (Dalvik/ARM)

Wrap-up

- Accidental correctness giving way to errors (x86 ↔ ARM)
- Need to respect JMM — or analogs in other languages
- **Sound static analysis** based on fragmentary models
  - Yields composable analysis-based verification at scale
  - Helps find bugs and identify specific fixes
- Surprisingly, **dynamic analysis** has an important role
  - Understanding, particularly for global properties
  - Performance focus
  - Visualization of missing “fence posts”