Overview

- Memory Models, and the JMM in particular
- Memory models involve the compiler
  – an example: Coherence
- Need to make safety guarantees
  – even for improperly synchronized code
- Integration of MM and language
  – what does volatile mean?

What is a memory model?

- If two threads have a data race, what behaviors are allowed?
- Sequential consistency
  – interleave memory operations consistent with original ordering in each thread

MM’s can interfere with optimization

- In each thread, no ordering constraint between actions in that thread
- Compiler could decide to reorder
- Processor architecture might perform out of order
- Sequential consistency prohibits almost all reordering of memory operations
  – unless you can prove accessed by single thread

Some processors support Sequential Consistency

- But most compilers violate it
- Interesting experiment
  – disable all optimization that could violate sequential consistency
  – examine effect on performance

Do programmers care about the details of MM’s?

- If you are writing synchronization primitives
  – You care deeply about the memory model your processor supports
- But if you have synchronized everything properly
  – do you really care?
  – but do you have everything synchronized properly?
The Java Memory Model

- Chapter 17 of the Java Language Specification (and Chap 8 of the VM Spec)
- Describes how threads interact via locks and read/writes to memory
- Done in a style foreign to other work on memory models
- Very hard to understand
  - At first I thought I was just dense
  - Eventually I figured out that no one understands it

The Java Memory Model is dead

- Was intended to have Coherence
  - For each memory location in isolation, SC
  - Unanticipated impact on compiler
- I found a hairball
  - imposes constraints no one intended
  - makes system unusable
- Proof by invocation of Guy Steele
- It will be replaced, not patched
  - but with what?

Coherent memory

- Once you see an update by another thread
  - can’t forget that you’ve seen the update
- Cannot reorder two reads of the same memory location

Reads kill reuse

- Must treat “may reads” as kills
  - a read may cause your thread to become aware of a write by another thread
- Can’t replace c = p.x with c = a

Most JVM’s violate Coherence

- Every JVM I’ve tested that eliminates redundant loads violates Coherence:
  - Sun’s Classic Wintel JVM
  - Sun’s Hotspot Wintel JVM
  - IBM’s 1.1.7b Wintel JVM
  - Sun’s production Sparc Solaris JVM
  - Microsoft’s JVM
- Bug # 4242244 in Javasoft’s bug parade
  - JVM’s don’t match spec

Impact on Compiler Optimizations?

- Preliminary work by Dan Scales, DecWRL
- Made reads kill, have side effects
- Better is probably possible, but will require work
- Reads have side effects
  - but can be done speculatively
  - change intermediate representation
OK, what do we want

- Not going to change Java threading model
  – even if people don’t like it
- Have to keep in mind that most Java programmers haven’t taken an OS course
  – Can’t hold them to high standards
- Incorrectly synchronized programs must have a (safe) meaning
  – can’t allow a cracker to use improperly synchronized code to attack a system.

Rest of the talk

- ⇒ Goals for new memory model
- Weak memory models
  – what can go wrong
- Safety Guarantees
- Changing semantics
- Immutable objects / Atomic object creation
- Future

Goals for new Memory Model

- Preserve existing and/or necessary safety guarantees
  – even in the presence of data races
- Have a clear specification we can reason about
- Allow efficient immutable classes
- New MM should not break “reasonable” existing code

Goals for new MM (continued)

- In code that doesn’t involve locks or volatile variables, use as much as possible of the standard compiler optimization techniques
- Data-race-free programs should be guaranteed sequentially consistent results
  – Constraints not necessary to ensure SC for data-race-free programs should be imposed with “care and deliberation”.

Weak memory models

- Initially, Mem[100] = 200
  Mem[200] = 17
  Mem[300] = 666

- On processor 1:
  Mem[300] = 42
  R1 := Mem[100]
- On processor 2:
  Mem[100] = 300
  R2 := Mem[R1]
  R2 = ?
  17, 42, 666(?)
Not much of a surprise

- Compiler could reorder write instructions
- Processor might reorder write instructions
- Put in a memory barrier...

Weak memory models

- Initially,
  Mem[100] = 200
  Mem[200] = 17
  Mem[300] = 666
- On processor 1:
  Mem[300] = 42
  MemBarrier
  Mem[100] = 300
- On processor 2:
  R1 := Mem[100]
  R2 := Mem[R1]?
  R2 = 17, 42, 666(?)

More of a surprise

- The data dependence does not prevent reordering of instructions on processor 2
- How could this happen?
- Spec says it can happen (Alpha, IA-64, …)
- Can it happen in reality?
  - Value prediction
  - Cache memories

Processor weak memory models

- Only on shared memory multiprocessors
- Sun’s TSO (Total store order), PSO (Partial store order) and RMO (Relaxed Memory Order) all strong enough to prevent it
  - Sun Sparc’s all run in TSO order
    - because too much of Sun’s code breaks under any looser model
  - MAJC runs under RMO
    - although some details still up in the air

Processor weak memory models

- Sun’s TSO (Total store order), PSO (Partial store order) and RMO (Relaxed Memory Order) all strong enough to prevent it
  - Sun Sparc’s all run in TSO order
    - because too much of Sun’s code breaks under any looser model
  - MAJC runs under RMO
    - although some details still up in the air
Multithreaded semantics for Java

It does go wrong on some machines

- Multiprocessor Dec Alphas and Intel IA-64 machines
  - at least according to the spec
  - not clear if any current implementations would allow it to happen
- Intel IA-32?
  - not sure; probably allowed by spec
  - not clear if current implements allow it

Same issues, but for object initialization

- Thread 1
  - initialize an object at address X,
  - Make Foo.x reference the object at address X
- Thread 2
  - reads Foo.x, gets X
  - reads field of object at address X, sees pre-initialization value

This is bad!

- If we see an uninitialized value, we might see something that isn’t typesafe
  - seeing a random integer isn’t so great either
- We could put a memory barrier after object initialization
  - but that isn’t enough (as before)
  - need a memory barrier for reading processor

A simple fix

- Allocate objects out of zeroed memory
  - Zero memory during garbage collection
  - All processors know that the memory was initially zero.
- If we see a pre-initialized ptr, we see null
  - zero for numerics, false for boolean
- Matches Java semantics
  - Fields set to default value (null/false/zero) before constructor is executed

Not sufficient

- This fix isn’t sufficient
  - for several reasons
- Consider reading the vtbl ptr of an object
  - points to the virtual function table and class data for object
- If we saw null, virtual method dispatch would generate a segmentation fault for VM
- instanceof and checkedCast could also go wrong

What else can go wrong

- Can see 0 for any world in object header
  - implementation dependent as to what is stored in header
- Can see 0 for array length
  - can throw invalid IndexOutOfBoundsException
- Class loading...
Class loading

- class Foo {
  public static Object x;
  
}  
- On processor 1:
  // First use of Bar
  // loads class Bar
  tmp = new Bar();
  Foo.x = tmp;

- class Bar {
  public int hashCode()
  { ... };
  
}  
- On processor 2:
  Foo.x.hashCode();

Now what can go wrong

- Nothing in code executed by processor 2 to indicate that it might be executing code from a new class
- Any field in Bar’s vtbl or class data could be zero
  - while others could be valid
- Parts of native code for Bar could be zero

Global memory barriers

- Class loading requires global memory barrier
  - each processor must do a memory barrier
  - but initiated by only one processor
- May need to synchronize instruction as well as data caches
- Not cheap/easy to do on many systems

Code generation/specialization

- Generating native code also requires global memory barrier
- In system like HotSpot
  - new code is generated as profile data is collected
  - not just the first time a method is executed

OK, so safety is hard

- Hopefully, I’ve convinced you that many safety issues, often taken for granted, are difficult on a SMP with a weak memory model
- Need to formalize the safety issues we will guarantee

Rest of the talk

- Goals for new memory model
- Weak memory models
  - what can go wrong
- ⇒ Safety Guarantees
- Changing semantics
- Immutable objects / Atomic object creation
- Future
Multithreaded semantics for Java

Safety Guarantees

- For reads of fields and arrays
  - type safety
  - not-out-of-thin-air safety
- VM safety - despite lack of synchronization
  - All operations other than reading a field or array are as usual
    - can’t crash/violate VM
    - No new exceptions
    - array length is always correct

Implementing type safety

- Allocate objects out of memory that everyone agrees has been zeroed
  - since memory was zeroed, every processor must have done a memory barrier

Implementing VM safety

- Global memory barrier after class loading and code generation
  - work to make this efficient
- Null vtbl - two solutions
  - check if null; if so, mem barrier and reload
  - Handle SIGSEGV and recover
- Zero array length
  - check if 0; if so, mem bar and reload
    - for bounds check, only check once out of bounds exception is detected

Class loading safety

- Current spec says that before executing getstatic, putstatic, invokestatic or new on a class, you must load the class or verify that another class has loaded it
  - Add: if you verify that another class has loaded it, you must do an acquire so as to see all writes by the thread that initialized it
  - Add invokevirtual, invokespecial, getfield, putfield

Implementing class loading safety

- You don’t really want to check that a class has been loaded before each invokevirtual
- Loading/initializing a class “prepares” it
- Whenever you do a global memory barrier, “prepared” classes become “distributed”
- Before doing a new on a “prepared” class, you must do a global memory barrier

Rest of the talk

- Goals for new memory model
- Weak memory models
  - what can go wrong
- Safety Guarantees
  - Changing semantics
- Immutable objects / Atomic object creation
- Future
Changing semantics

- volatile
  - tighten to make more uses valid
- final
  - change to enable optimizations
- useless locks
  - change to enable optimizations

Changing the semantics of volatile

- C++ spec:
  - There is no implementation independent meaning for volatile
- Existing Java spec
  - Actions on volatile variables are SC
  - but actions on normal variables and volatile variables can be reordered
  - Change semantics of volatile so that
    - read of volatile is treated as acquire
    - write of volatile is treated as release

Example of new use of volatile

- Double-check idiom
  // used (incorrectly) in many places
  if (helper == null) // helper is volatile
    synchronized(this) {
      if (helper == null) {
        helper = new Helper();
      }
    }
  // Would also be fixed by atomic object creation (see later)

Example of new use of volatile

- Advanced Double-check idiom
  if (!initialized) // initialized is volatile
    synchronized(this) {
      if (!initialized) {
        a = new A();
        b = new B();
        b.update(...);
        initialized = true;
      }
    }
  // Not handled by atomic object creation

Changing the semantics of final

- Under current semantics, a memory barrier effects final fields
  - forces them to be reloaded from memory
- Change semantics to allow them to remain in registers
  - also across unknown method calls
- Ugly if objects escapes constructor before final fields initialized

Changing the semantics of useless locks

- Right now, a lock/unlock is always treated as a memory barrier
- Even if the lock/unlock is done on an object not visible to other threads
  - synchronized (new Object()) {} is a memory barrier
- Even if it is a recursive lock
  - e.g., when a synchronized method calls another synchronized method
Multithreaded semantics for Java

What MM semantics allow this?
- Lazy Release Consistency?
- Information only needs to flow
  - from Thread 1 to Thread 2 if
  - Thread 1 does a release on X
  - Thread 2 does an acquire on the same X
- Useful in software DSM systems
  - not too useful in hardware DSM systems
- Very useful for compilers!

Recursive locks are no-ops

Compilers and Lazy Release Consistency
- Locks/unlocks on thread local objects are no-ops
  - under old semantics, memory barrier required
- Java monitors are recursive
  - recursive locks/unlocks become no-ops
    - under old semantics, memory barrier required

Rest of the talk
- Goals for new memory model
- Weak memory models
  - what can go wrong
- Safety Guarantees
- Changing semantics
  - Immutable objects / Atomic object creation
- Future

Immutable Objects
- Many Java classes represent immutable objects
  - e.g., String
- Creates many serious security holes if Strings are not truly immutable
  - probably other classes as well
  - should do this in String implementation

Why aren’t Strings immutable?
- A String object is initialized to have default values for it’s fields
- Then the fields are set in the constructor
- Thread 1 could create a String object
- pass it to Thread 2
  - which calls a sensitive routine
- which sees the fields change from their default values to their final values

William Pugh, Univ. of Maryland
Making String immutable

• Could make String methods synchronized
  – most programmers don’t think methods for immutable objects need to be synchronized
  – slow down String methods on all platforms
    • only needs to be synchronized on SMP’s with weak memory models
    • doesn’t need synchronization on SPARC or MAJC(?) SMP’s

What we need

• Some way of making a class truly immutable
• With minimal (zero?) performance impact on systems where nothings needs to be done
• Not too ugly

Atomic object creation

• Many naïve programmers assume object creation is atomic
  – Subsumes truly immutable objects
• They are wrong
  – In this code, a could get the value 1, 3 or 0
• No way to make a constructor synchronized
  – wouldn’t work anyway

Should object creation be atomic?

• Advocated by Sun
  – no impact on SPARC/MAJC
• Simple approach would require memory barriers in front of each getfield
  – Factor of 3 slowdown on 2 processor Alpha
• Numbers by Sanjay Ghemawat, DEC SRC
• Simple optimization improves this
  – Factor of 1.87 slowdown on 2 processor Alpha

A solution?

• Guarantee that reads of final fields see the final value, not the initial default value
  – assuming object doesn’t escape before final fields set
• Also fits well with new semantics of final
• Might be much cheaper than full atomic object creation
• Better programming style than assuming atomic object creation?

Not as simple as that

• No way for elements of an array to be final
• For Strings, have to see final values for elements of character array
• So…
  – Read of final field is treated as a weak acquire
    • matching a release done when object is constructed
  – weak in that it only effects things dependent on value read
    • no compiler impact
Multithreaded semantics for Java

Implementing these semantics

- Start with the idea of doing a memory barrier before each getfield of a final field
  - 1666 of 9018 fields in rt.jar are final
  - 2292 could be final
- Only do the memory barrier if object is young
  - Objects are no longer young once a global memory barrier occurs after their construction

Checking for young objects

- Several ways it could be done
  - Here is one
- Put young objects in addresses with sign bit off
- Put old objects and stack allocated objects in addresses with sign bit on
- Conditional memory barrier: if (addr < 0) MemBar;

Guidelines for Compiler Writers

- Don’t assume that
  - if you drop a value cached in a register,
  - you can reload the value and get the same value
  - even though you don’t see any possible writes
- Memory barriers induced by acquire/release
  - moving something past a barrier isn’t symmetric

Rest of the talk

- Goals for new memory model
- Weak memory models
  - what can go wrong
- Safety Guarantees
- Changing semantics
- Immutable objects / Atomic object creation
  - ⇒ Future

Future

- The Java Memory Model will be completely replaced
- Trying to get lots of feedback
  - mailing list, web page
  - road shows
  - BOF at OOPSLA
- Unclear how endgame will be played
  - All Java licensee’s get a voice

Where next?

- Java Memory model mailing list
  - http://www.cs.umd.edu/~pugh/java/memoryModel
  - Lots of discussion going on
- Won’t get changed for next rev of JLS
- Some people at Sun want to avoid a JSR
  - but if changes have a substantial impact on some Java licensees, probably unavoidable