JVM Issues

William Pugh

Dept. Of Computer Science
Univ. of Maryland

http://www.cs.umd.edu/~pugh/java

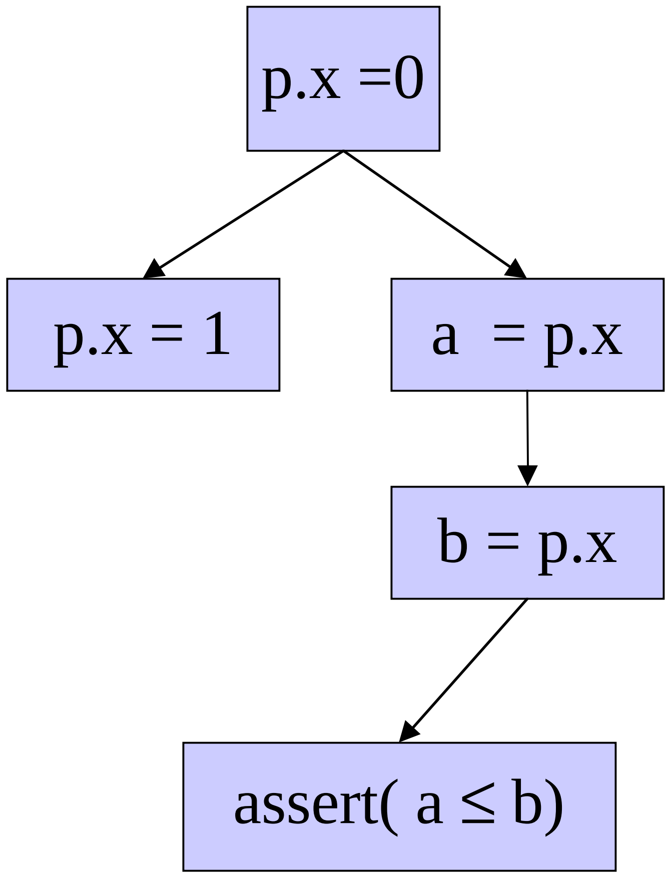
Coherence

Coherence

- Can’t reorder reads of the same variable
- A seemingly reasonable requirement
- Forbids standard compiler optimizations

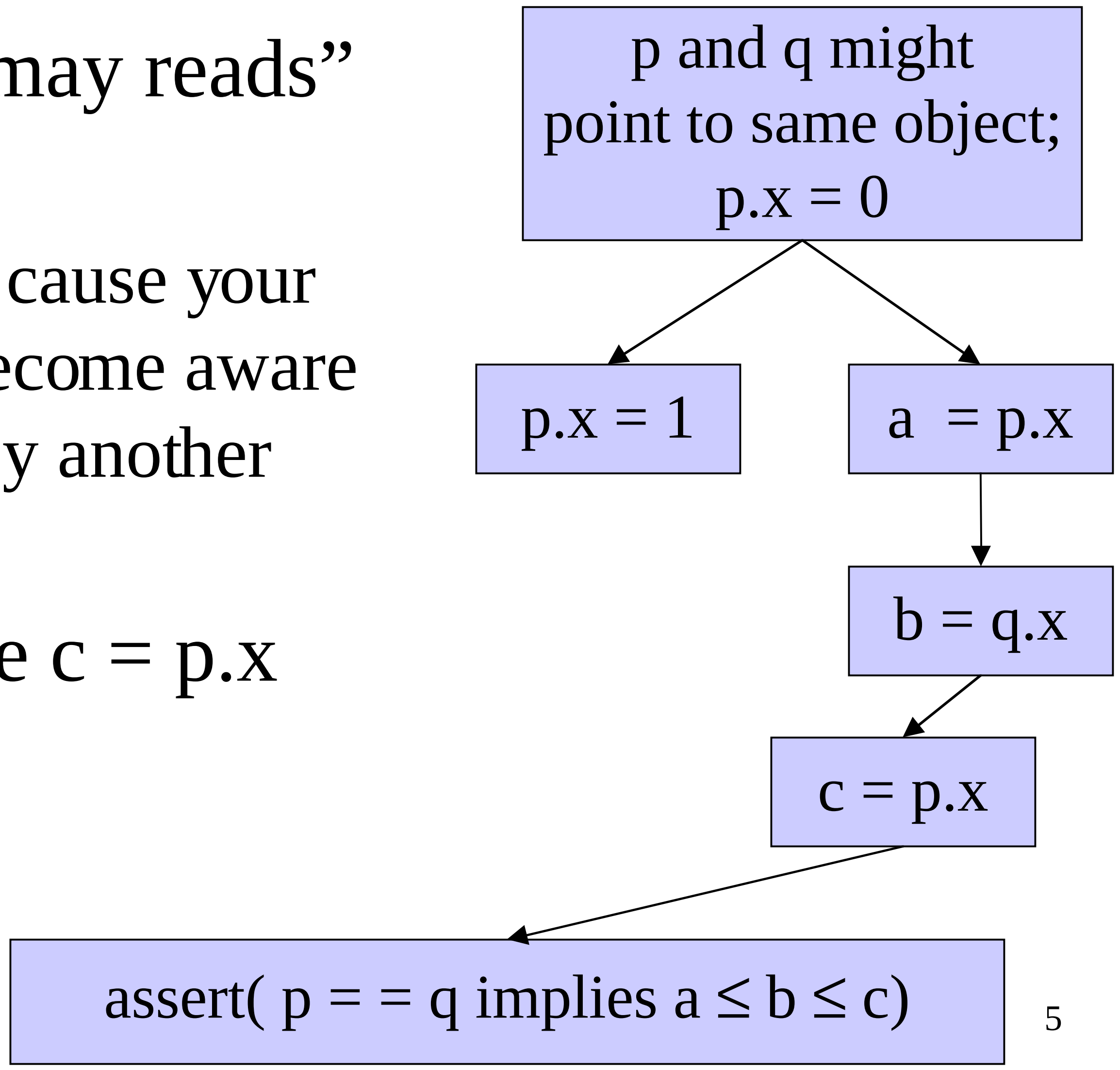
Coherent memory

- 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


p and q might point to same object; p.x = 0
p.x = 1
a = p.x
b = q.x
c = p.x
assert(p = = q implies a ≤ b ≤ c)

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

compress	1.18		mpegaudio	1.44
jess	1.03		richards	0.98
cst	1.01		mtrt	1.02
db	1.04		jack	1.06
si	1.03		tsgp	1.36
javac	0.99		tmix	1.11

Coherence is useful

- Consider

```
int cachedHashCode;
int hashCode() {
    if (cachedHashCode == 0)
        cachedHashCode = computeHC();
    return cachedHashCode;
    }
```

Weak processor memory models

Weak memory models

- Initially,

  Mem[100] = 200

  Mem[200] = 17

  Mem[300] = 666

- On processor 1:

  Mem[300] = 42

  Mem[100] = 300

- On processor 2:

  R1 := Mem[100]

  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:
- On processor 2:

$$\begin{tikzcd}
\text{Mem[300] = 42} \arrow[dd, bend right] & \text{R1 := Mem[100]} \arrow[d, bend left, "?"] \\
\text{MemBarrier} & \text{R2 := Mem[R1]} \arrow[ul] \\
\text{Mem[100] = 300} \arrow[uur] &
\end{tikzcd}$$

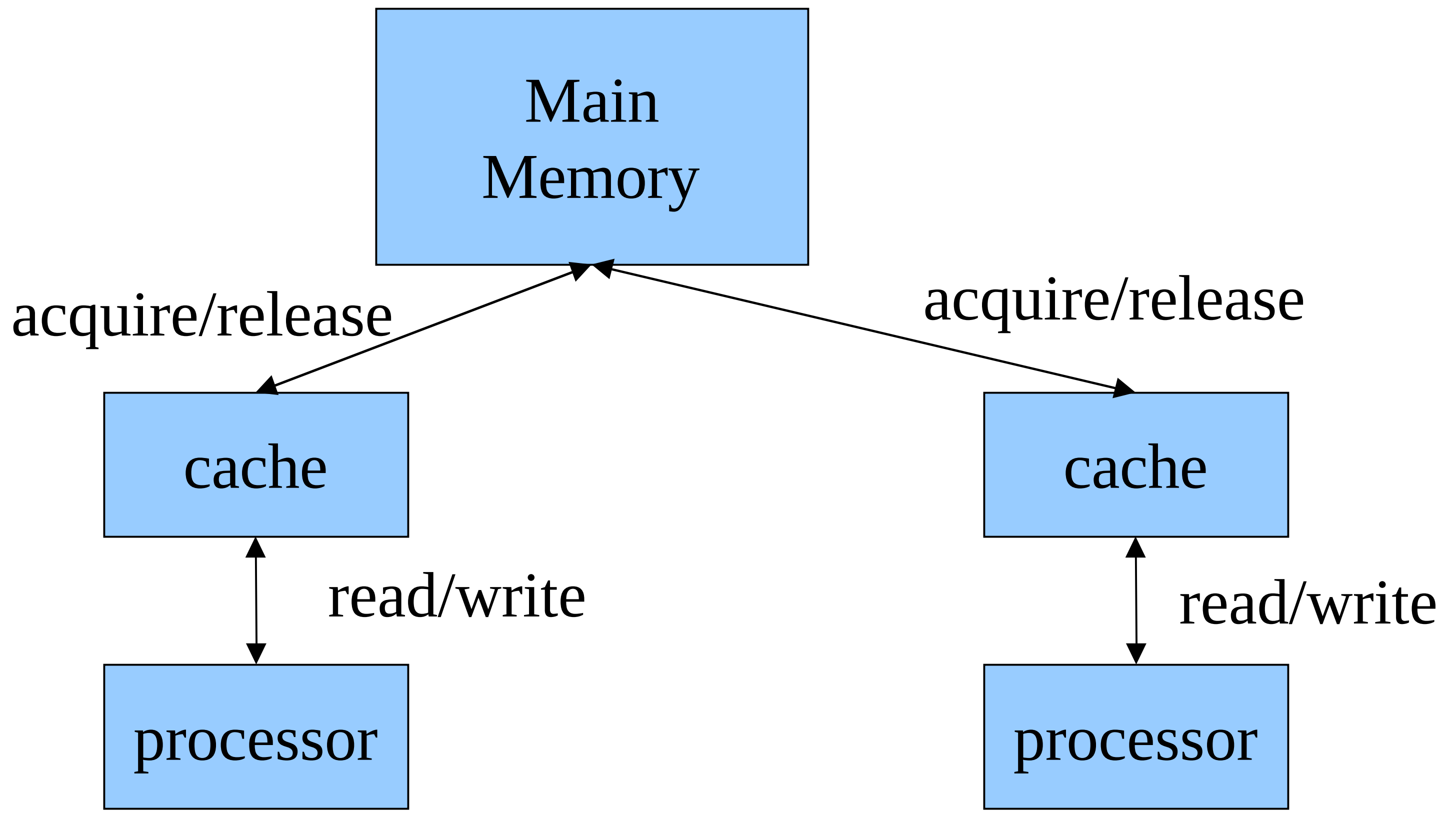
R2 = ?

17, 42, 666(?)

More of a surprise

- The data dependence does *not* prevents reordering of instructions on processor 2
- How could this happen?
- Spec says it can happen on Alpha
- Can it happen in reality?
  - Value prediction
  - Cache memories

Processor weak memory models



Processor weak memory models

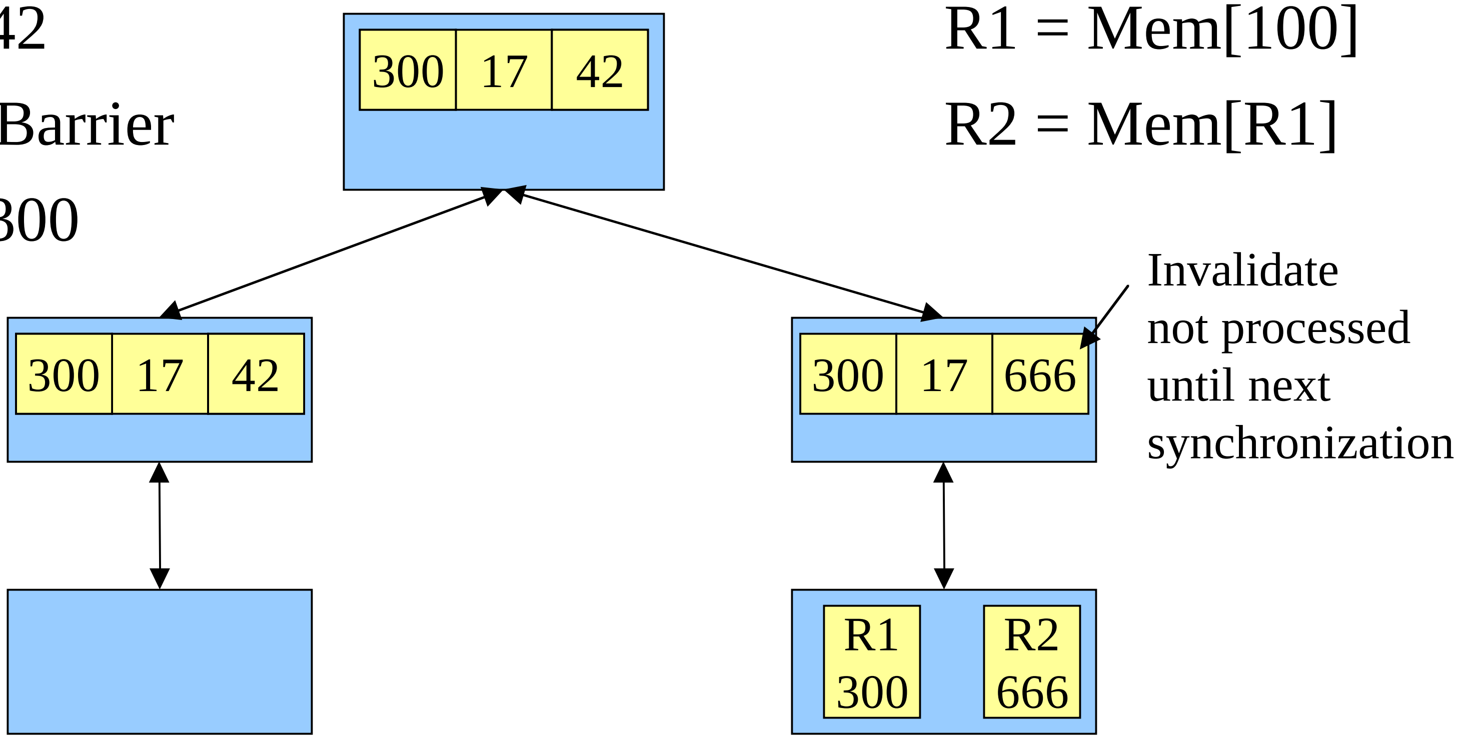
Mem[300] = 42

Release/MemBarrier

Mem[100] = 300

R1 = Mem[100]

R2 = Mem[R1]



What machines can it happen on?

- 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

Intel machines

- Intel IA32 architecture …
  - see later slides
- Intel IA64 doesn’t allow it if second store is st.rel
  - probably not if memory barrier separating two writes
    - need clarification

Alpha processors

- Multiprocessor Alphas 21264 systems can exhibit this behavior
  - although very unlikely to occur
- Future Alpha processors may exhibit more it more frequently
  - new optimizations:
    - value prediction
- Alpha architects feel very strongly that chip should not support anything strong than RC

How it can happen on an Alpha

Initially: p = & x, x = 1, y = 0	
Thread 1	**Thread 2**
y = 1	
memoryBarrier	i = *p
p = & y	
Can result in: i = 0	

- Assume T1 runs on P1 and T2 on P2.
- P2 has to be caching location y with value 0.
- P1 does y=1 which causes an “invalidate y” to be sent to P2.
- This invalidate goes into the incoming “probe queue” of P2; as you will see, the problem arises because this invalidate could theoretically sit in the probe queue without doing an MB on P2.
- The invalidate is acknowledged right away at this point (i.e., you don't wait for it to actually invalidate the copy in P2's cache before sending the acknowledgment).
- Therefore, P1 can go through its MB.
- And it proceeds to do the write to p.
- Now P2 proceeds to read p.
- The reply for read p is allowed to bypass the probe queue on P2 on its incoming path (this allows replies/data to get back to the 21264 quickly without needing to wait for previous incoming probes to be serviced).
- Now, P2 can dereference p to read the old value of y that is sitting in its cache (the inval y in P2's probe queue is still sitting there).

How it can happen on an Alpha

- The short form
  - writes send immediate invalidate messages
    - which are processed lazily
  - memory barrier is a local operation
    - forces processing of invalidate messages

Intel IA-32 Memory Model

Intel IA-32 memory model

- Intel Architecture Software Developer’s Manual
  - Volume 3: System Programming
    - Chapter 7
- x386: sequentially consistent
- x486 and Pentium: TSO
- P6: “write ordered with store buffer forwarding”

write ordered with store buffer forwarding

- Reads can be performed in any order
- writes are always performed in program order
- Writes and following reads can be reordered
- In other words, relaxes R→R and W→R

Note from spec:

- [Software] should insure that accesses to shared variables that are intended to control concurrent execution among processors are explicitly required to obey program ordering through the use of appropriate locking or serializing operations (refer to Section 7.2.4., “Strengthening or Weakening the Memory Ordering Model”).

Lock prefix

- used for atomic read/modify/write
- In x386 - x586, locks bus
  - very expensive
- In x686, if location cached, handled internally
  - can it be reordered?

Virtual Machine Safety Issues

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 word 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 some 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

Weak Processor MM

- I believe that the reordering allowed by the Alpha is a mistake
- It will be very painful to handle in the JVM
  - Will we every know if it has been handled correctly?
- I strongly urge architects to avoid this
  - Weak ordering is not sufficient
- But Java must run on an Alpha SMP

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

- 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

Which Compiler Optimizations are Legal?

Legal Compiler Optimizations

- Would like to prove that in code without synchronization or volatile fields
  - we can do all optimizations we would do for single threaded programs
- Not possible
  - Counter example, see next slide
- Would like to prove it for as many optimizations as possible

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

Performance Issues

Cost of useless synchronization

- In Volano
  - syncs on thread local Buffered I/O streams
  - about 3% improvement from removal
- In DB benchmark
  - lots of syncs on Vector
  - about 20% improvement from removal
  - but can double speed by replacing hand-coded shell sort with built-in merge sort

Benefit of Double Check

- Application Isolation paper by Grzegorz Czajkowski
  - needed synchronization for every access of static field
  - using double-check to eliminate synchronization gave about 10% performance improvement