IBM Software Group

Java Synchronization :
Not as bad as it used to be!

Mark Stoodley
J9 JIT Compiler Team

Why was synchronization in Java slow?

- **Early implementations relied on OS**
  - OS synchronization mechanisms are heavy-weight
- **Thread-safety provided in class library**
  - Whether you need it or not
  - Sometimes deeply buried: lack of awareness

How VMs and JITs Reduce the Cost

1. **Make each lock/unlock operation faster**
   - Flat locks: only use OS monitor if contended
   - Spin loop: before *inflating* to an OS monitor
   - Fast path to avoid some runtime function calls
2. **Do fewer lock/unlock operations**
   - Lock coarsening reduces repetitive, recursive locking

Relying solely on OS monitors: bad idea

1. **OS monitor per Java object impractical**
   - Pool of OS monitors accessed via monitor table
   - Need to lock the table to access it (extra cost)
2. **Locks often uncontended (many studies)**
3. **Long dead time to wake thread up even if contended lock is available "soon"**

1. Flat locks: Most locks are not contended

- **Add lock word to object header**
  - Thread ID (TID)
  - Recursive count (CNT)
  - Flat Lock Contended (FLC) bit
- **Atomic compare and swap to lock object:**
  - If lock word is 0, TID is stored to lock word in header
  - If TID not my thread ID, then set FLC bit and go get an OS monitor (inflated lock)
  - Otherwise, increment CNT field

Flat locks are good!

- **If lock isn't contended, no OS monitor needed**
- **Cost of atomic instruction is much lower**
- **Memory penalty for lock word in every object**
  - Benefits considered "worth the cost"

2. Spin loop: Don't give up too easily

- **Observation: most locks held for short time**
  - Even though contended now, probably available "soon"
- **So, spin for a while before acquiring OS monitor**
- **In fact, several tiers of loops:**
  - Sequence of spins, lock attempts, and yields before blocking

Flat lock + Spin loop even better!!

- No OS monitor even for many contended locks
- Probably faster than blocking thread
- BUT spinning uses processor resources for which threads might be competing
- Complex trade-off to balance spinning versus yielding versus blocking

3. Fast Path in Generated Code

- **Lock/Unlock usually needs a function call**
- **To save the function call:**
  - Do one atomic compare-and-swap to grab the lock directly if possible
  - Otherwise, callout to spin on the flat lock
- **Important: memory coherence actions mandated by Java spec**
  - Lock: no later reads have started
  - Unlock: all earlier writes have completed

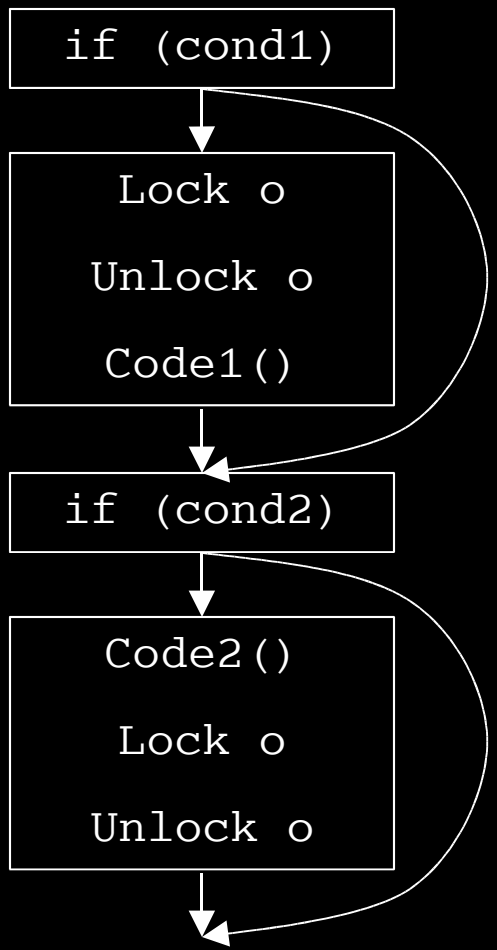
How VMs and JITs Reduce the Cost

1. **Make each lock/unlock operation faster**
   - Flat locks: only use OS monitor if contended
   - Spin loop: before *inflating* to an OS monitor
   - Fast path to avoid some runtime function calls
2. **Do fewer lock/unlock operations**
   - Lock coarsening reduces repetitive, recursive locking

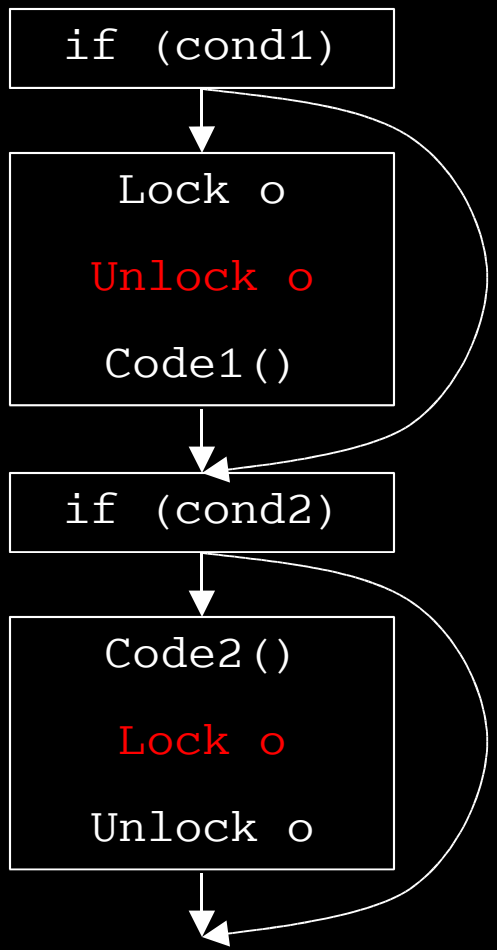
Lock Coarsening

- **Goal is to reduce repetitive locking on the same object in a Java method**
  - Inlined synchronized methods ➔ repetitive syncs
  - Recursive locking also removed as a perq
- ***Coarsen* region of code over which lock is held**
  - ✘ Safety concerns
  - ✘ Maybe increases contention
  - ✓ Fewer Lock and Unlock operations
  - ✓ Better code

Lock Coarsening Example



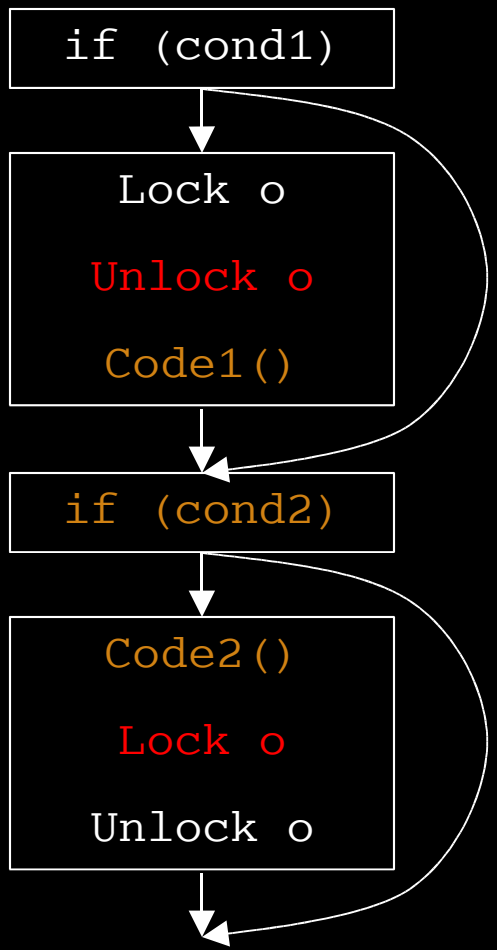
Lock Coarsening Example



removable on one path out?

removable on one path in?

Lock Coarsening Example



removable on one path out?

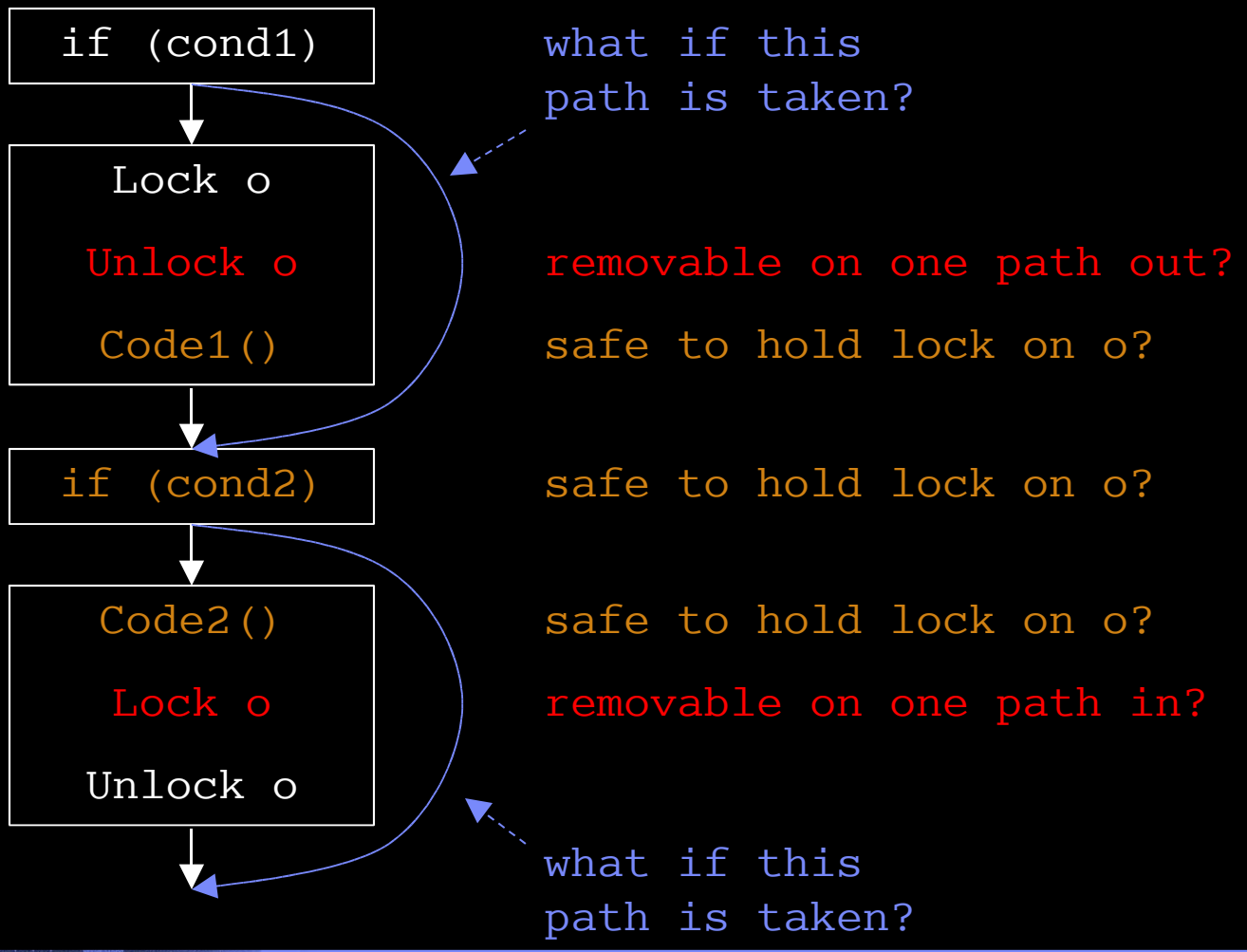
safe to hold lock on o?

safe to hold lock on o?

safe to hold lock on o?

removable on one path in?

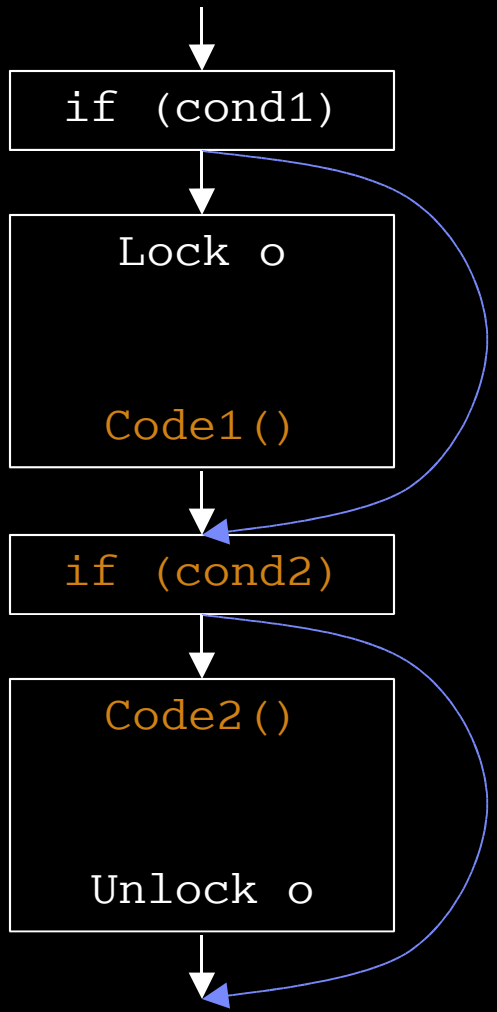
Lock Coarsening Example



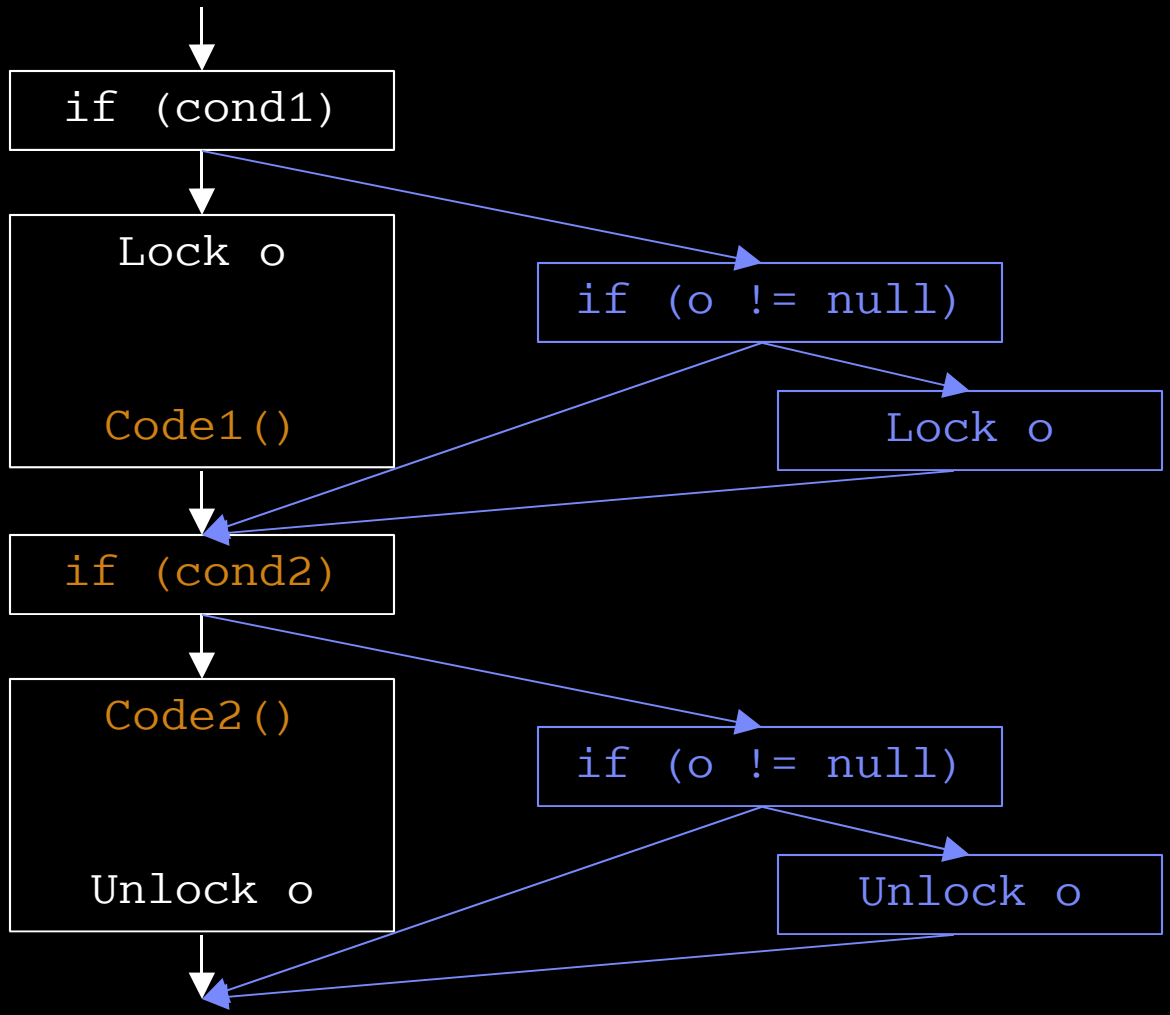
Barriers to coarsening a locked code region

1. **Lock/Unlock operation on another object**
2. **Volatile field accesses**
3. **Loop headers and exits**
4. **Method invocations**
5. **Unresolved field accesses**

Remove unnecessary Locks and Unlocks



Ensure consistency of coarsened region



Not as easy as it looks: Malicious Code

- **One (of N) complications to address**
- **Production JITs cannot ignore bytecode hackers**
- **VM spec: must detect unbalanced locking**
  - MONITORENTER and MONITOREXIT bytecodes not guaranteed to be balanced
  - VM must throw exception if MONITOREXIT attempted on unlocked (or unowned) object
- **Can detect balanced uses in a method, but called methods can do anything**

So what's the problem?

```
{
   synchronized(o) {
      o.foo();
   }
   synchronized(o) {
      o.bar();
   }
}
```

So what's the problem? Consider foo() and bar():

```
{
   synchronized(o) {
      o.foo();
   }
   synchronized(o) {
      o.bar();
   }
}
```

```
BYTECODE for foo() {
   aload 0
   MONITOREXIT
   ret
}
BYTECODE for bar() {
   aload 0
   MONITORENTER
   ret
}
```

foo() unlocks the object: so Unlock should cause exception

```
{
   Lock o;
      o.foo();
   Unlock o;
   Lock o;
      o.bar();
   Unlock o;
}
```

```
BYTECODE for foo() {
   aload 0
   MONITOREXIT
   ret
}
BYTECODE for bar() {
   aload 0
   MONITORENTER
   ret
}
```

Coarsening removes the exception!!

```
{
   Lock o;
      o.foo();


      o.bar();
   Unlock o;
}
```

```
BYTECODE for foo() {
   aload 0
   MONITOREXIT
   ret
}
BYTECODE for bar() {
   aload 0
   MONITORENTER
   ret
}
```

Nice, elegant solution

- **If there are calls that we can't fully peek**
  - Can't remove the Unlock: replace with a guarded Unlock,Lock
  - Guard checks if current thread holds the lock, same CNT
    - If guard fails, unlock the object then re-lock the object
- **If there are virtual calls we've fully peeked into, wrap guarded Unlock,Lock in a side-effect guard that gets NOP-ed**
- **Lock can be safely removed**

Guarded Unlock (assume foo looked safe when compiled)

```
{
   Lock o; saveLockWord = o.lockWord & (~FLCMASK);
   o.foo();  // virtual invocation
   if (foo overidden) {
      if (o not locked || (o.lockWord & (~FLCMASK)) != saveLockWord) {
         Unlock o; Lock o;
      }
   }
      o.bar();
   Unlock o;
}
```

Summary

- **VMs and JIT compilers have improved synchronization cost in two ways:**
  - Make Locks/Unlocks faster
    - Flat locks, spin loop, fast path
  - Remove recursive, repetitive locking
    - Lock coarsening
    - Complication: Malicious code

Java Synchronization Not So Bad !!

Questions?

(Lunch Time !!)