Java theory and practice: Synchronization optimizations in Mustang

Escape analysis can help optimize synchronization

Brian Goetz

In the September installment of *Java theory and practice*, columnist Brian Goetz examined escape analysis, an optimization that has been on the "to-do" list for many JVMs for quite some time and which is expected in HotSpot in the Mustang (Java™ SE 6) release. Escape analysis can be used to convert heap-based object allocation into the less-expensive, stack-based allocations, but it can also be used to make other optimization decisions as well, including optimizing the use of synchronization. This month Brian introduces some of the synchronization optimizations slated for Mustang. *Note: This column describes features of a future version of Sun's HotSpot JVM implementation. Specific features discussed herein may or may not appear in Java SE 6 ("Mustang"); some may be delayed until Java SE 7 ("Dolphin").*

View more content in this series

Whenever mutable variables are shared between threads, you must use synchronization to ensure that updates made by one thread are visible to other threads on a timely basis. The primary means of synchronization is the use of synchronized blocks, which provide both mutual exclusion and visibility guarantees. (Other forms of synchronization include volatile variables, the Lock objects in java.util.concurrent.locks, and atomic variables.) When two threads want to access a shared mutable variable, not only must both threads use synchronization, but if they are using synchronized blocks, those synchronized blocks must use the same lock object.

In practice, locking falls into two categories: mostly contended and mostly uncontended. Locks that are mostly contended are the "hot" locks in an application, such as the locks that guard the shared work queue for a thread pool. The data protected by these locks is needed constantly by many threads, and you can expect that when you go to acquire this lock, you may have to wait until someone else is done with it. Mostly uncontended locks are those that guard data that is not accessed so frequently, so that most of the time when a thread goes to acquire the lock, no other thread is holding that lock. Most locks are not frequently contended, so improving the performance of uncontended locking can improve overall application performance substantially.
The JVM has separate code paths for contended ("slow path") and uncontended ("fast path") lock acquisition. A lot of effort has already gone into optimizing the fast path; Mustang further improves both the fast path and the slow path and adds a number of optimizations that can eliminate some locking entirely.

**Lock elision**

The Java Memory Model says that one thread exiting a synchronized block happens-before another thread enters a synchronized block protected by that same lock; this means that whatever memory operations are visible to thread A when it exits a synchronized block protected by lock M are visible to thread B when it enters a synchronized block protected by M, as shown in Figure 1.

For synchronized blocks that use different locks, we cannot assume anything about their ordering -- it is as if there was no synchronization at all.

**Figure 1. Synchronization and visibility in the Java Memory Model**

It stands to reason, then, that if a thread enters a synchronized block protected by a lock that no other thread will ever synchronize on, then that synchronization has no effect and can therefore be removed by the optimizer. (The Java Language Specification explicitly allows this optimization.)

This scenario may sound unlikely, but there are cases when this is clear to the compiler. Listing 1 shows a simplified example of a thread-local lock object:

**Listing 1. Using a thread-local object as a lock**

```java
synchronized (new Object()) {
    doSomething();
}
```

Because the reference to the lock object disappears before any other thread could possibly use it, the compiler can tell that the above synchronization can be removed because it is impossible for two threads to synchronize using the same lock. While no one would ever directly use the idiom in Listing 1, this code is very similar to the case where the lock associated with a synchronized block can be proven to be a thread-local variable. "Thread-local" doesn't necessarily mean that it is implemented with the `ThreadLocal` class; it could be any variable that the compiler can prove is never accessed by another thread. Objects referenced by local variables and that never escape from their defining scope meet this test -- if an object is confined to the stack of some thread, no
other thread can ever see a reference to that object. (The only way an object can be shared across threads is if a reference to it is published to the heap.)

As luck would have it, escape analysis, which we discussed last month, provides the compiler with exactly the information it needs to optimize away synchronized blocks that use thread-local lock objects. If the compiler can prove (using escape analysis) that an object is never published to the heap, it must be a thread-local object and therefore any synchronized blocks that use that object as a lock will have no effect under the Java Memory Model (JMM) and can be eliminated. This optimization is called lock elision and is another of the JVM optimizations slated for Mustang.

The use of locking with thread-local objects happens more often than you might think. There are many classes, such as StringBuffer and java.util.Random, which are thread-safe because they might be used from multiple threads, but they are often used in a thread-local manner.

Consider the code in Listing 2, which uses a Vector in the course of building up a string value. The getStoogeNames() method creates a Vector, adds several strings to it, and then calls toString() to convert it into a string. Each of the calls to one of the Vector methods -- three calls to add() and one to toString() -- requires acquiring and releasing the lock on the Vector. While all of the lock acquisitions will be uncontended and therefore fast, the compiler can actually eliminate the synchronization entirely using lock elision.

Listing 2. Candidate for lock elision

```java
public String getStoogeNames() {
    Vector v = new Vector();
    v.add("Moe");
    v.add("Larry");
    v.add("Curly");
    return v.toString();
}
```

Because no reference to the Vector ever escapes the getStoogeNames() method, it is necessarily thread-local and therefore any synchronized block that uses it as a lock will have no effect under the JMM. The compiler can inline the calls to the add() and toString() methods, and then it will recognize that it is acquiring and releasing a lock on a thread-local object and can optimize away all four lock-unlock operations.

I’ve stated in the past that trying to avoid synchronization is generally a bad idea. The arrival of optimizations such as lock elision adds yet one more reason to avoid trying to eliminate synchronization -- the compiler can do it automatically where it is safe, and leave it in place otherwise.

Adaptive locking

In addition to escape analysis and lock elision, Mustang has some other optimizations for locking performance as well. When two threads contend for a lock, one of them will get the lock and the other will have to block until the lock is available. There are two obvious techniques for implementing blocking; having the operating system suspend the thread until it is awakened later or using spin locks. A spin lock basically amounts to the following code:
while (lockStillInUse);

While spin locks are CPU-intensive and appear inefficient, they can be more efficient than suspending the thread and subsequently waking it up if the lock in question is held for a very short time. There is significant overhead to suspending and rescheduling a thread, which involves work by the JVM, operating system, and hardware. This presents a problem for JVM implementers: if locks are only held for a short time, then spinning is more efficient; if locks are held for a long time, then suspension is more efficient. Without information on what the distribution of lock times will be for a given application, most JVMs are conservative and simply suspend a thread when it fails to acquire a lock.

However, it is possible to do better. For each lock, the JVM can adaptively choose between spinning and suspension based on the behavior of past acquires. It can try spinning and, if that succeeds after a short period of time, keep using that approach. If, on the other hand, a certain amount of spinning fails to acquire the lock, it can decide that this is a lock that is generally held for "a long time" and recompile the method to use only suspension. This decision can be made on a per-lock or per-lock-site basis.

The result of this adaptive approach is better overall performance. After the JVM has had some time to acquire some profiling information on lock usage patterns, it can use spinning for the locks that are generally held for a short time and suspension for the locks that are generally held for a long time. An optimization like this would not be possible in a statically compiled environment because the information about lock usage patterns is not available at static compilation time.

**Lock coarsening**

Another optimization that can be used to reduce the cost of locking is lock coarsening. Lock coarsening is the process of merging adjacent synchronized blocks that use the same lock object. If the compiler cannot eliminate the locking using lock elision, it may be able to reduce the overhead by using lock coarsening.

The code in Listing 3 is not necessarily a candidate for lock elision (although after inlining addStoogeNames(), the JVM might still be able to elide the locking), but it could still benefit from lock coarsening. The three calls to add() alternate between acquiring the lock on the Vector, doing something, and releasing the lock. The compiler can observe that there is a sequence of adjacent blocks that operate on the same lock, and merge them into a single block.

### Listing 3. Candidate for lock coarsening

```
public void addStooges(Vector v) {
    v.add("Moe");
    v.add("Larry");
    v.add("Curly");
}
```

Not only does this reduce the locking overhead, but by merging the blocks, it turns the contents of the three method calls into one big synchronized block, giving the optimizer a much larger basic
block to work with. So lock coarsening can also enable other optimizations that have nothing to do with locking.

Even if there are other statements between the synchronized blocks or method calls, the compiler can still perform lock coarsening. The compiler is allowed to move statements into a synchronized block -- just not out of it. So if there were intervening statements between the calls to add(), the compiler could still turn the whole of addStooges() into one big synchronized block that uses the Vector as the lock object.

Lock coarsening may entail a trade-off between performance and responsiveness. By combining the three lock-unlock pairs into a single lock-unlock pair, performance is improved by reducing instruction count and reducing the amount of synchronization traffic on the memory bus. The cost is that the duration for which the lock is held may be lengthened, increasing the amount of time other threads might be locked out and also increasing the probability of lock contention. However, the lock is held for a relatively short period of time in either case, and the compiler can apply heuristics based on the length of the code protected by synchronization to make a reasonable trade-off here. (And depending on what other optimizations are enabled by the larger basic block, the duration for which the lock is held may not even be lengthened under coarsening.)

**Conclusion**

Uncontended locking performance has improved with nearly every JVM version. Mustang continues this trend by improving both the raw performance of uncontended and contended locking and introducing optimizations that can eliminate many lock operations.
Related topics

- **Urban performance legends, revisited** (Brian Goetz, developerWorks, September 2005): An exploration of optimization of escape analysis, which enables lock elision.
- **Thin Locks**: IBM Researcher David Bacon developed the now-standard "thin lock" algorithm that is used to reduce the "fast path" lock acquisition cost.
- **Implementing fast Java monitors with relaxed locking**: Sun researcher Dave Dice presents an improved algorithm for intrinsic locks.

© Copyright IBM Corporation 2005
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)