Lock optimizations on the HotSpot VM

Tõnis Pool
University of Tartu
tonis.pool@gmail.com
Supervised by Eero Vainikko

Abstract

This report gives an overview of different optimisations made by the Java HotSpot VM to decrease the overhead of synchronisation. It lists the major strategies, explains why they work and provides and cites some rudimentary benchmarking to prove that they work.

I. Introduction

Synchronisation is a necessary evil in order to construct distributed and/or parallel systems. In the Java programming language this is achieved by using mutually exclusive locks. I said evil, because acquiring such locks is a relatively expensive operation in terms of CPU cycles needed. Thus any attempt to optimize or minimize such costs bring about an immediate performance gain to any distributed system running on the Java Virtual Machine (VM).

The managed runtime approach of the VM, gives it unique opportunities to analyse the running code and adapt itself to the specific application. This opens up tremendous opportunities to different optimisations. Over the years there have been many optimisations done on the HotSpot Virtual Machine. In this report I want to list the major strategies taken to optimize synchronisation with the goal of giving some deeper insight into the VM and its behaviour when running parallel code.

II. Optimisations

I. Lock Elision

The biggest optimisation to acquiring a lock would be to remove it altogether. This is exactly at the heart of the approach called Lock Elision. This is possible thanks to a technique called Escape Analysis, which means that the HotSpot Server Compiler can analyse the scope of a new object[1]. Based on this information the server compiler might eliminate synchronization blocks (lock elision) if it determines that an object is thread local[2].

This may seem counterintuitive at first, but the use of locking with thread-local objects happens more often than you might think. There are many classes, such as java.lang.StringBuffer and java.util.Vector, which are thread-safe because they might be used from multiple threads, but they are often used in a thread-local manner[3]. Consider the following method:

Listing 1: *Possible Lock Elision Scenario*

```
public String getFruits(String... fruits) {
  Vector v = new Vector();
  for (String fruit : fruits)
    v.add(fruit);
  return v.toString();
}
```

Here the compiler sees that no references to the `Vector v` ever escape the `getFruits` method, thus no other thread can synchronise on it and so it can remove the unnecessary lock-unlock operations.

II. Lock Coarsening

If the code in Listing 1 would be rewritten as:

Listing 2: *Possible Lock Coarsening Scenario*

```
public Vector getFruits(Vector v) {
  v.add("Banana");
  v.add("Melon");
  return v;
}
```

Then the compiler can no longer eliminate the locks because the Vector object may not be thread-local anymore. Some other thread might synchronise on the `Vector v`, while this method is executed. However the compiler can see that the same lock is taken and released repeatedly. Optimisation known as Lock Coarsening means that those individual synchronised blocks are effectively melded together to form a single block reducing the amount of synchronization work.

Doing this around a loop could cause a lock to be held for long periods of times, so the technique is only used on non-looping control flow[4].

III. Adaptive Spinning

Earlier versions of the HotSpot VM relied upon OS mutexes for synchronisation. This often resulted in up to half of the programs runtime spent on useless synchronization[5]. Let's call these OS level locks fat locks. Threads waiting on a fat lock are suspended, when the lock is released threads are notified and queued. The first thread to run gets the lock others are suspended again. This will result in a lot of context switches, which are very expensive[6].

Another option to implement locking is to instead of putting the thread to sleep, keep trying until it gets the lock. These locks are often called spin locks or thin locks, because they are more lightweight then their fat counterparts and they keep the thread active (spinning).

Pseudocode for a very simple spinlock implementation[7]:

Listing 3: *Spinlock pseudocode implementation*

```
public class PseudoSpinLock {
  private static final int LOCK_FREE = 0;
  private static final int LOCK_TAKEN = 1;

  static int lock; //lock memory position

  public void lock() {
    while (cmpxchg(LOCK_TAKEN, [lock])
      == LOCK_TAKEN);
  }

  public void unlock() {
    int old == cmpxchg(LOCK_FREE, [lock]);
    assert (old == LOCK_TAKEN);
  }
}
```

Naturally such active spinning wastes a lot of CPU cycles, but it still might be better than fat locks, if the locks are taken only for very short periods of time.

Adaptive spinning is a "spin then block" strategy, which tries to avoid context switching. It is “adaptive" because the duration of the spin is determined by policy decisions based on factors such as the rate of success and/or failure of recent spin attempts on the same monitor and the state of the current lock owner[4].

If the runtime detects that a lock is generally held a long time, it can recompile the method to only use OS level locks. And conversely it can change a fat lock to lighter spin lock if it detects that the lock is no longer contended[3].

IV. Biased Locking

Adaptive Spinning works on the premise that most locks in most programs are not contended. A further observation has been made, that not only are most locks not contended, they tend to be locked and unlocked by the same thread, making the lock somewhat thread local[8].

This gives away to a class of optimisations called Biased Locking. Which in the heart of it means making consecutive locking and unlocking very cheap for a single thread. In other words biasing the lock towards the thread that first locks it.

Each object in the Java HotSpot VM has a two-word object header. The first word is called mark word and contains various information about the object state, including synchronisation, garbage collection and hash code information. The second word points to the class of the object[9].

On object creation, if biasing is enabled for that type, it is marked as Biasable by setting the lower 3 bits of the mark word to 101. When a thread wants to lock a Biasable object it makes an atomic CAS (Compare-And-Swap) operation to put its thread id to the object mark word. If the CAS succeeds then the object is considered successfully locked and biased towards that thread. Consecutive locks / unlocks by the biased thread just have to check the mark word if the bias flag is still present, saving many rather expensive atomic operations[10].

When the CAS fails it means that another thread is the bias owner and its bias should be removed and made to look like the object was locked by the lightweight locking scheme. This is done by the following scenario[9]:

1. A global safepoint is reached, at which point all threads are halted
2. The bias owner's stack is walked and the lock records associated with the object are filled in with the values that would have been produced had lightweight locking been used to lock the object
3. The object's mark word is updated to point to the oldest associated lock record on the stack
4. Threads blocked on the safepoint are released

V. Bulk Rebiasing and Revocation

Biased locking optimizes for the case when lock is not contended and usually locked by a single thread. Extending on this, a common scheme found in practice is that a lock is still not contended, but the thread locking the object changes over the objects lifetime. An example would be a situation where some thread initialises an object and passes it over to a thread that continues working on it. To optimize this we'd need a mechanism to "transfer" the bias to the new thread.

Similarly there exist common synchronisation patterns where it is expected that locks are contended and acquired by different threads. An example would be the classic producer-consumer pattern. In those situations it would be beneficial to disable biasing to avoid the expensive bias revocation that needs to halt threads[10].

These both scenarios are handled by Bulk Rebiasing and Revocation. Bulk rebiasing means that all objects of type T have their bias statuses reset, so the next thread that locks the object becomes the new bias owner - effectively transferring the bias. Bulk revocation means disabling biasing all together for all instances of some class, that are detected to be usually contended.

Bulk rebiasing is implemented by adding a timestamp - epoch - to each objects mark word and to corresponding data types. An object is now considered biased toward a thread T if both the bias owner in the mark word is T, and the epoch of the instance is equal to the epoch of the data type. Rebising works by the following scheme[9]:

1. A global safepoint is reached, at which point all threads are halted
2. Increment the epoch number of class C
3. Locate objects of class C that are currently locked in blocked threads, updating their bias epochs or alternatively revoke the biases based on heuristic
4. Threads blocked on the safepoint are released

Implementing bulk revocation by data type is much simpler. A biasable boolean flag is added to the data type, which is then checked on each lock operation. Pseudocode of the lock operation supporting biased locking, bulk rebiasing and revocation is the following[9]:

Listing 4: *Pseudocode of lock operation on HotSpot VM*

```
void lock(Object* obj, Thread* t) {
  int lw = obj->lock_word;
  if (lock_state(lw) == Biased
    && biasable(lw) ==
       obj->class->biasable
    && bias_epoch(lw) ==
       obj->class->bias_epoch) {
    if (lock_or_bias_owner(lw) == t->id) {
      return; // t is the bias owner.
    } else {
      revoke_bias(obj, t);
    }
  } else {
    // normal locking protocol
    // possibly with bias acquisition
  }
}
```

III. Benchmarks

Microbenchmarking the VM is notoriously difficult. It's easy to write a benchmark that seems to test the right thing, but often some unforeseen aspect of the VM will skew the results in a mysterious fashion[11]. Thus I refrain from writing benchmarks myself as much as possible and op for referencing existing ones if possible.

I. Lock Elision

I couldn't find a benchmark to reference for the performance enchantment gained via Lock Elision, thus I wrote one myself. The benchmark is written using the JMH test harness, which is one of the recommended ways of creating micro benchmarks on the Java VM[12]. Source code of the benchmark is here.

The benchmark measures the performance difference of java.lang.StringBuilder and java.lang.StringBuffer. StringBuffer is a thread safe equivalent of StringBuilder, meaning all of it's methods are synchronised.

The heart of the benchmark are the methods concatBuffer and concatBuilder, with the following body.

Listing 5: *cocncatBuffer method*

```
public static String concatBuffer(
  String s1, String s2, String s3) {
    StringBuffer sb = new StringBuffer();
    sb.append(s1);
    sb.append(s2);
    sb.append(s3);
    return sb.toString();
}
```

The concatBuilder method is the same, just the java.lang.StringBuilder data structure is used instead of java.lang.StringBuffer. The results show that using StringBuffer doesn't have a performance overhead and they perform roughly the same, which is expected thanks to Lock Elision.

Table 1: *Lock Elision performance*

MacBook Pro i7 Haswell 2.8GHz - Ops/Sec	
StringBuilder	StringBuffer
224057.578 ± 4751	224746.545 ± 3381

II. Lock Coarsening

Similarly to I Lock Elision, the benchmark for lock coarsening compares the performance difference of java.lang.StringBuilder and java.lang.StringBuffer. The only difference with the previous benchmark is that now concatBuffer and concatBuilder methods don't return java.lang.String, but the underlying data structure, which means that the compiler cannot eliminate the synchronisation, but can use Lock Coarsening. Source code of the benchmark is here.

The results show that using a StringBuffer has more overhead than before, which is expected, because some synchronisation blocks are still present. Though thanks to Lock Coarsening the overhead is not an order of magnitude slower.

Table 2: *Lock Coarsening performance*

MacBook Pro i7 Haswell 2.8GHz - Ops/Sec	
StringBuilder	StringBuffer
242159.381 ± 5703	191549.119 ± 3127

III. Biased Locking

Benchmarking has shown that biased locking improves performance near an order of magnitude on locks that are not contended, by increasing operations per second 10x[13]. Also as expected, when the lock indeed is contended, then biased locking doesn't have any benefit, but at the same time, doesn't bring about any noticeable overhead.

Table 3: *Biased locking performance*

Nehalem 2.8GHz - Ops/Sec		
Threads	-BiasedLocking	+BiasedLocking
1	53,283,461	450,950,969
2	18,519,295	18,108,615

IV. Bulk Rebiasing and Revocation

Figure 1 shows how bulk rebiasing and revocation affects performance on some industry standard benchmark suites. SPECjbb2000 and SPECjbb2005 transfer a certain number of objects between threads, so bulk rebiasing has the largest effect here. Note that the addition of both bulk revocation and rebiasing does not reduce the peak performance of biased locking[9].

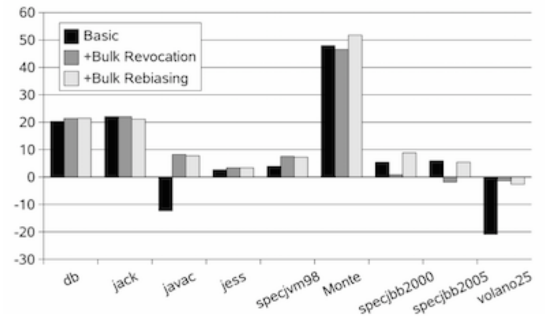

Figure 1: *Percentage speedups yielded by bulk revocation and bulk rebiasing*

IV. Summary

As we can see a modern Java VM is a highly complex machine, capable of adapting itself to each applications needs. The adaptive runtime approach gives the VM insights about the running code that no statically compiled programming language can newer acquire. Over the years the HotSpot VM has grown smarter and smarter in using those insights to boost the programs performance and even fix the programmers mistakes in terms of needles synchronisation.

Hopefully this paper gave some deeper insight about the techniques currently employed by the HotSpot VM. Tough these are surely to be outdated some day, when a new piece of data is discovered and a new or improved optimisation deployed.

References

[1] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff, "Escape analysis for java," *SIGPLAN Not.*, vol. 34, pp. 1–19, Oct. 1999.

[2] "Java HotSpot™ Virtual Machine Performance Enhancements." `http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`.

[3] B. Goetz, "Java theory and practice: Synchronization optimizations in Mustang."

`http://www.ibm.com/developerworks/java/library/j-jtp10185/`, October 2005.

[4] "Java SE 6 Performance White Paper." `http://www.oracle.com/technetwork/java/6-performance-137236.html`.

[5] D. F. Bacon, R. Konuru, C. Murthy, and M. J. Serrano, "Thin locks: Featherweight synchronization for java," *SIGPLAN Not.*, vol. 39, pp. 583–595, Apr. 2004.

[6] D. Dice, "Synchronization in java se 6 (hotspot)." `http://home.comcast.net/~pjbishop/Dave/MustangSync.pdf`, August 2006.

[7] M. Hirt and M. Lagergren, *Oracle JRockit: The Definitive Guide*. Packt Publishing, 6 2010.

[8] K. Kawachiya, A. Koseki, and T. Onodera, "Lock reservation: Java locks can mostly do without atomic operations," *SIGPLAN Not.*, vol. 37, pp. 130–141, Nov. 2002.

[9] K. Russell and D. Detlefs, "Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing," *SIGPLAN Not.*, vol. 41, pp. 263–272, Oct. 2006.

[10] T. K. . C. Wimmer, "Synchronization and Object Locking." `https://wikis.oracle.com/display/HotSpotInternals/Synchronization`, April 2008.

[11] J. Ponge, "Avoiding Benchmarking Pitfalls on the JVM." `http://www.oracle.com/technetwork/articles/java/architect-benchmarking-2266277.html`, October 2014.

[12] `http://openjdk.java.net/projects/code-tools/jmh/`.

[13] M. Thompson, "Biased Locking, OSR, and Benchmarking Fun." `http://mechanical-sympathy.blogspot.com/2011/11/biased-locking-osr-and-benchmarking-fun.html`, November 2011.