Biased Locking in HotSpot

The biased locking scheme in HotSpot arose from the following paper authored by myself, Mark Moir, and Bill Scherer. Briefly, the compare-and-swap (CAS) operations normally used to acquire a Java monitor incurs considerable local latency in the executing CPU. Since most objects are locked by at most one thread during their lifetime, we allow that thread to *bias* an object toward itself. Once biased, that thread can subsequently lock and unlock the object without resorting to expensive atomic instructions. Obviously, an object can be biased toward at most one thread at any given time. (We refer to that thread as the bias holding thread). If another thread tries to acquire a biased object, however, we need to *revoke* the bias from the original thread. (At this juncture we can either rebias the object or simply revert to normal locking for the remainder of the object's lifetime). The key challenge in revocation is to coordinate the revoker and the revokee (the bias holding thread) -- we must ensure that the revokee doesn't lock or unlock the object during revocation. As noted in the paper, revocation can be implemented in various ways - signals, suspension, and safepoints to name a few. Critically, for biased locking to prove profitable, the benefit conferred from avoiding the atomic instructions must exceed the revocation costs. Another equivalent way to conceptualize biased locking is that the original owner simply defers unlocking the object until it encounters contention. Biased locking bears some semblance to the concept of *lock reservation*, which is well-described in Kawachiya's dissertation. It's also similar in spirit to "opportunistic locks" (oplocks) found in various file systems; the scheme describe in Mike Burrows's "How to implement unnecessary mutexes" (In Computer Systems: Theory, Technology and Applications, Dec. 2003); or Christian Siebert's one-sided mutual exclusion primitives.

Biased locking is strictly a response to the latency of CAS. It's important to note that CAS incurs *local* latency, but does not impact scalability on modern processors. A common myth is that each CAS operation "goes on the bus", and, given that the interconnect is a fixed a contended resource, use of CAS can impair scalability. That's false. CAS can be accomplished locally -- that is, with no bus transactions -- if the line is already in M-state. CAS is usually implemented on top of the existing MESI snoop-based cache coherence protocol, but in terms of the bus, CAS is no different than a store. For example lets say you have a true 16-way system. You launch a thread that CASes 1 billion times to a thread-private location, measuring the elapsed time. If you then launch 16 threads, all CASing to thread-private locations, the elapsed time will be the same. The threads don't interfere with or impede each other in any way. If you launch 16 threads all CASing to the same location you'll typically see a massive slow-down because of interconnect traffic. (The sole exception to that claim is Sun's Niagara, which can gracefully tolerate sharing on a massive scale as the L2$ serves as the interconnect). If you then change that CAS to a normal store you'll also see a similar slow-down; as noted before, in terms of coherency bus traffic, CAS isn't appreciably different than a normal store. Some of the misinformation regarding CAS probably arises from the original implementation of lock:cmpxchg (CAS) on Intel processors. The lock: prefix caused the LOCK# signal to be asserted, acquiring exclusive access to the bus. This didn't scale of course. Subsequent implementations of lock:cmpxchg leverage cache coherency protocol -- typically snoop-based MESI -- and don't assert LOCK#. Note that lock:cmpxchg will still drive LOCK# in one extremely exotic case -- when the memory address is misaligned and spans 2 cache lines. Finally, we can safely use cmpxchg on uniprocessors but must use lock:cmpxchg on multiprocessor systems. Lock:cmpxchg incurs more latency, but then again it's a fundamentally different instruction that cmpxchg. Lock:cmpxchg is *serializing*, providing bidirectional mfence-equivalent semantics. (Fence or barrier instructions are never needed for uniprocessors) This fact might also have contributed to the myth that CAS is more expensive on MP systems. But of course lock:cmpxchg incurs no more latency on a 2x system than on an 8x system.

While digressing on the topic of bus operations, lets say that a load is followed closely in program order by a store or CAS to the same cache line. If the cache line is not present in the issuing processor the load will generate a *request-to-share* transaction to get the line in S-state and the store or CAS will result in a subsequent *request-to-own*transaction to force the line into M-state. This 2nd transaction can be avoided on some platforms by using a prefetch-for-write instruction before the load, which will force the line directly into M-state. It's also worth mentioning that on typical classic SMP systems, pure read-sharing is very efficient. All the requesting processors can have the cache line(s) replicated in their caches. But if even one processor is writing to a shared cache line, those writes will generate considerable cache coherence traffic; assuming a write-invalidate cache coherence policy (as opposed to write-update) the readers will continually re-load the cache line just to have it subsequently invalidated by the writer(s). Put differently, loads to a cache line are cheap if other processors are loading from but not storing to that same line. Stores are cheap only if no other processors are concurrently storing to or loading from that same line. (We can draw an imprecise analogy between cache coherency protocols and read-write locks in that for a given cache line there can only be one writer at any given time. That's the processor with the line in M-state. Multiple readers of the line allowed and of course the lifetime of a reader can't overlap a write. Unlike traditional read-write locks, however, the cache coherency protocol allows writers to invalidate readers, so we can't push the analogy too far. In a twisted sense, the coherency protocol is obstruction-free). Coherency bandwidth is a fixed and contended global resource, so in addition to local latency, excessive sharing traffic will impact overall scalability and impede the progress of threads running on other processors. A so-called coherency miss -- for example a load on processor P1 where processor P2 has the cache line in M-state -- is typically much slower than a normal miss (except on Niagara). Recall too, that acquiring a lock involves a store (CAS, really) to the lock metadata, so if you have threads on processors P1 and P2 iterating, acquiring the same, the lock acquisition itself will generate coherency traffic and result in the cache "sloshing" of the line(s) holding the metadata. Generally, excessive coherency traffic is to be avoided on classic SMP systems. But as usual, there's an exception to any rule, and in this case that exception is Sun's Niagara, which can tolerate sharing gracefully. We should now return back to the topic of CAS latency, as that issue motivates the need for biased locking.

CAS and the fence (AKA barrier or MEMBAR) instructions are said to be serializing. On multiprocessor systems serializing instructions control the order in which which stores are loads are visible with respect to memory. Such instructions are often implemented in a crude fashion on most current processors. Typically, a serializing instruction will bring the CPU to a near halt, kill and inhibit any out-of-order (OoO) instructions, and wait for the local store buffer to drain. That's OK for simple processors, but it results in considerable loss of performance on more sophisticated OoO processors. There's no *fundamental* reason why CAS or fence should be slow. Until recently, the trend was toward worse CAS and fence performance, but it appears that this trend may be reversing. IBM made considerable improvements in fence latency between the Power4 and Power5. Likewise, Intel made remarkable improvements for both lock:cmpxchg and mfence latency in the recent Core2 processors.

In the case of biased locking -- since we know that the object is unshared -- it's reasonable to assume that the actions of the bias holding thread would keep cache lines underlying the object's lock word and data in M- or E-state

Interestingly, if a a memory barrier, or "fence" instruction is sufficiently faster than CAS we can implement biased locking revocation with a Dekker-like mechanism. If you're interested in delving deeper you might find the following of interest: Asymmetric Dekker Synchronization. (That same document also describes an optimization used in HotSpot where we were able to eliminate a MEMBAR from the state transition code path on multiprocessor systems).

Kenneth Russell and David Detlefs (formerly of SunLabs) have a paper appearing in OOPSLA'06 -- Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing -- where they describe a scheme to reduce the cost of revocation.

It's important to note that biased locking is permitted under the the Java Memory Model as clarified by JSR133. Refer to the Java Language Specification, 3rd edition, Chapter 17, Section 4. Doug Lea's JSR-133 Cookbook is an excellent resource for JVM developers or curious Java programmers.

Finally, there's not currently space in the mark word to support both an identity hashCode() value as well as the thread ID needed for the biased locking encoding. Given that, you can avoid biased locking on a per-object basis by calling System.identityHashCode(o). If the object is already biased, assigning an identity hashCode will result in revocation, otherwise, the assignment of a hashCode() will make the object ineligible for subsequent biased locking. This property is an artifact of our current implementation, of course.

As an aside, the latency of atomic read-modify-write operations such as CAS was in part what motivated myself and Alex Garthwaite to develop our Mostly Lock-Free Malloc (ISMM 2002) mechanism, which uses processor-local allocation without requiring atomics or the binding of threads to processors.

See also US07814488 (2002).