Eliminating Synchronization-Related Atomic Operations with Biased Locking and Bulk Rebiasing

Kenneth Russell
Sun Microsystems, Inc.
kenneth.russell@sun.com

David Detlefs *
david.detlefs@alum.mit.edu

Abstract

The Java™programming language contains built-in synchronization primitives for use in constructing multithreaded programs. Efficient implementation of these synchronization primitives is necessary in order to achieve high performance.

Recent research [9, 12, 10, 3, 7] has focused on the run-time elimination of the atomic operations required to implement object monitor synchronization primitives. This paper describes a novel technique called *store-free biased locking* which eliminates all synchronization-related atomic operations on uncontended object monitors. The technique supports the bulk transfer of object ownership from one thread to another, and the selective disabling of the optimization where unprofitable, using epoch-based bulk rebiasing and revocation. It has been implemented in the production version of the Java HotSpot™VM and has yielded significant performance improvements on a range of benchmarks and applications. The technique is applicable to any virtual machine-based programming language implementation with mostly block-structured locking primitives.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization

General Terms Algorithms, Languages, Performance

Keywords Java, synchronization, monitor, lock, atomic, optimization, bias, rebias, revoke, reservation

1. Background and Motivation

The Java programming language contains built-in support for monitors to facilitate the construction of multithreaded programs. Much research has been dedicated to decreasing the execution cost of the associated synchronization primitives.

A class of optimizations which can be termed *lightweight locking* [1, 2, 5] are focused on avoiding as much as possible the use of "heavy-weight" operating system mutexes and condition variables to implement Java monitors. The assumption behind these techniques is that most lock acquisitions in real programs are uncontended. Lightweight locking techniques use atomic operations upon monitor entry, and sometimes upon exit, to ensure correct synchronization. These techniques fall back to using OS mutexes and condition variables when contention occurs.

A related class of optimizations which can be termed *biased locking* [3, 7, 9] rely on the further property that not only are most monitors uncontended, they are only entered and exited by one thread during the lifetime of the monitor. Such monitors may be profitably *biased* toward the owning thread, allowing that thread to enter and exit the monitor without using atomic operations. If another thread attempts to enter a biased monitor, even if no contention occurs, a relatively expensive *bias revocation* operation must be performed. The profitability of such an optimization relies on the benefit of the elimination of atomic operations being higher than the penalty of revocation.

Current refinements of biased locking techniques [12, 10] decrease or eliminate the penalty of bias revocation, but do not optimize certain synchronization patterns which occur in practice, and also impact peak performance of the algorithm.

Multiprocessor systems are increasingly prevalent; so much so that uniprocessors are now the exception rather than the norm. Atomic operations are significantly more expensive on multiprocessors than uniprocessors, and their use may impact scalability and performance of real applications such as javac by 20% or more (Section 6). It is crucial at this juncture to enable biased locking optimizations for industrial applications, and to optimize as many patterns of synchronization in these applications as possible.

1.1 Contributions

This paper presents a novel technique for eliminating atomic operations associated with the Java language's synchronization primitives called *store-free biased locking* (SFBL). It is similar to, and is inspired by, the *lock reservation* technique [9] and its refinements [12, 10]. The specific contributions of our work are:

- We build upon invariants preserved by the Java HotSpot VM to *eliminate repeated stores* to the object header. Store elimination makes it easier to transfer bias ownership between threads.
- We introduce *bulk rebiasing* and *revocation* to amortize the cost of per-object bias revocation while retaining the benefits of the optimization.
- An *epoch-based* mechanism which invalidates previously held biases facilitates the bulk transfer of bias ownership from one thread to another.

Our technique is the first to support *efficient transfer of bias ownership* from one thread to another for sets of objects. Previous techniques do not optimize the situation in which more than one thread locks a given object. The approaches above support optimization of more synchronization patterns in applications than previous tech-

* This work was done while this author was an employee of Sun Microsystems, Inc.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-348-4/06/0010.

bitfields				tag bits	state
hash		age	0	01	unlocked
ptr to lock record				00	lightweight locked
ptr to heavyweight monitor				10	inflated
				11	marked for GC
thread ID	epoch	age	1	01	biasable

Figure 1. Synchronization-related states of an object's mark word.

niques, and allow biased locking to be enabled by default for all applications.

1.2 Organization of this Paper

The rest of this paper is organized as follows. Section 2 describes the lightweight locking technique in the Java HotSpot VM and its invariants. Section 3 describes the basic version of our biased locking technique. Section 4 describes the bulk rebiasing and revocation techniques used to amortize the cost of bias revocation. Section 5 improves the scalability of bulk rebiasing and revocation using epochs. Section 6 discusses results from various benchmarks. Section 7 provides detailed comparisons to earlier work. Section 8 describes how to obtain our implementation, and Section 9 concludes.

2. Overview of Lightweight Locking in the Java HotSpot VM

The lightweight locking technique used by the Java HotSpot VM[4] has not been described in the literature. Because knowledge of some of its aspects is required to understand store-free biased locking (SFBL), we present a brief overview here.

The Java HotSpot VM uses a two-word object header. The first word is called the *mark word* and contains synchronization, garbage collection and hash code information. The second word points to the class of the object. See figure 1 for an overview of the layout and possible states of the mark word.

Our biased locking technique relies on three invariants. First, the locking primitives in the language must be mostly block-structured. Second, optimized compiled code, if it is produced by the virtual machine, must only be generated for methods with block-structured locking. Third, interpreted execution must detect unstructured locking precisely. We now show how these invariants are maintained in our VM.

Whenever an object is lightweight locked by a `monitorenter` bytecode, a *lock record* is either implicitly or explicitly allocated on the stack of the thread performing the lock acquisition operation. The lock record holds the original value of the object's mark word and also contains metadata necessary to identify which object is locked. During lock acquisition, the mark word is copied into the lock record (such a copy is called a *displaced mark word*), and an atomic compare-and-swap (CAS) operation is performed to attempt to make the object's mark word point to the lock record. If the CAS succeeds, the current thread owns the lock. If it fails, because some other thread acquired the lock, a slow path is taken in which the lock is *inflated*, during which operation an OS mutex and condition variable are associated with the object. During the inflation process, the object's mark word is updated with a CAS to point to a data structure containing pointers to the mutex and condition variable.

During an unlock operation, an attempt is made to CAS the mark word, which should still point to the lock record, with the displaced mark word stored in the lock record. If the CAS succeeds, there was no contention for the monitor and lightweight locking remains in effect. If it fails, the lock was contended while it was held and a slow path is taken to properly release the lock and notify other threads waiting to acquire the lock.

Recursive locking is handled in a straightforward fashion. If during lightweight lock acquisition it is determined that the current thread already owns the lock by virtue of the object's mark word pointing into its stack, a zero is stored into the on-stack lock record rather than the current value of the object's mark word. If zero is seen in a lock record during an unlock operation, the object is known to be recursively locked by the current thread and no update of the object's mark word occurs. The number of such lock records implicitly records the monitor recursion count. This is a significant property to the best of our knowledge not attained by most other JVMs[1].

The Java HotSpot VM contains both a bytecode interpreter and an optimizing compiler. The interpreter and compiler-generated code create activation records called *frames* on a thread's native stack during activation (i.e., execution) of Java methods. We designate these frames as *interpreted* or *compiled*. Interpreted frames contain data from exactly one method, while due to inlining, compiled frames may include data from more than one method.

Interpreted frames contain a region which holds the lock records for all monitors owned by the activation. During interpreted method execution this region grows or shrinks depending upon the number of locks held. In compiled frames, there is no such region. Instead, lock records are allocated by the compiler in a fashion similar to register spill stack slots. During compilation, metadata is generated which describes the set of locks held and the location of their lock records at each potential *safepoint* [15] in compiled code. The presence of lock records allows the runtime system to enumerate the locked objects and their displaced mark words within each frame. This information is used during various operations internal to the JVM, including bias revocation, which will be described later.

The Java Virtual Machine Specification[11] requires that an `IllegalMonitorStateException` be thrown if a monitorexit bytecode is executed without having previously executed a matching monitorenter. The interpreter detects this situation by checking that a lock record exists for an object being unlocked. It is not specified what happens when a monitorenter bytecode is executed in a method followed by removal of the corresponding frame from the stack without executing a monitorexit bytecode. In this case a JVM may legally either throw an exception or not. The Java Hotspot VM's interpreter eagerly detects this situation by iterating through the lock records when removing an interpreted frame and forcibly unlocking the corresponding objects. It then throws an exception if any locked objects were found.

The Java HotSpot client[8] and server[13] optimizing compilers will only compile and inline methods if dataflow analysis has proven that all monitorenter and monitorexit operations are properly paired; in other words, every lock of a given object has a matching unlock on the same object. Attempts to leave an object locked after the method returns, or to unlock an object not locked by that method, are detected by dataflow analysis. Such methods, which almost never occur in practice, are never compiled or inlined but always interpreted.

Because interpreted execution precisely detects unstructured locking, and because compiled execution is proven through monitor matching to perform correct block-structured locking, it is guaranteed that an object's locking state matches the program's execution at all times. It is never the case that an object's locking state claims that it is owned by a particular thread when in fact the method which performed the lightweight lock has already exited. A

[1] An arbitrary Java virtual machine implementation is hereafter referred to as a JVM.

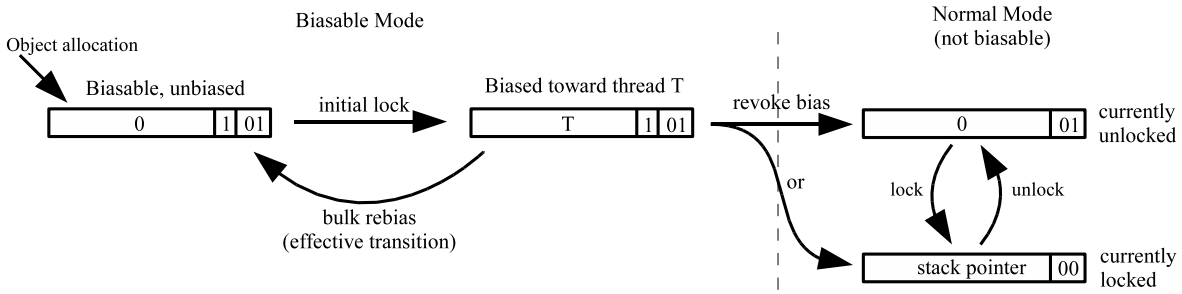

Figure 2. State transitions of an object's mark word under biased locking.

method may not unlock an object unless precisely that activation, and not one further up the stack, locked the object. These are essential properties enabling both the elimination of the recursion count described above as well as our biased locking technique in general. Complications arise in monitor-related optimizations such as lock coarsening in JVMs which do not maintain such invariants[16].

In summary, the following invariants in a programming language and virtual machine are essential prerequisites of our biased locking technique. First, the locking primitives in the language must be mostly block-structured. Second, compiled code, if it exists in the VM, must only be produced for methods with block-structured locking. Third, interpreted execution must detect illegal locking states eagerly. These three invariants imply that an explicit recursion count for the lock is not necessary. Additionally, some mechanism must be present to record a "lock record" for the object externally to the object. In the Java HotSpot VM a lock record is allocated on the stack, although it might be allocated elsewhere.

3. Store-Free Biased Locking

Assuming the invariants in Section 2, the SFBL algorithm is simple to describe. When an object is allocated and biasing is enabled for its data type (discussed further in Section 4), a *bias pattern* is placed in the mark word indicating that the object is *biasable* (figure 1). The Java HotSpot VM uses the value 0x5 in the low three bits of the mark word as the bias pattern.

The thread ID may be a direct pointer to the JVM's internal representation of the current thread, suitably aligned so that the low bits are zero. Alternatively, a dense numbering scheme may be used to allow better packing of thread IDs and potentially more fields in the biasable object mark word.

During lock acquisition of a biasable but unbiased object, an attempt is made to CAS the current thread ID into the mark word's thread ID field. If this CAS succeeds, the object is now biased toward the current thread, as in figure 2. The current thread becomes the *bias owner*. The bias pattern remains in the mark word alongside the thread ID.

If the CAS fails, another thread is the bias owner, so that thread's bias must be revoked. The state of the object will be made to appear as if it had been locked by the bias owner using the JVM's underlying lightweight locking scheme. To do this, the thread attempting to bias the object toward itself must manipulate the stack of the bias owner. To enable this a global safepoint is reached, at which point no thread is executing bytecodes. 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. Next, the object's mark word is updated to point to the oldest associated lock record on the stack. Finally, the threads blocked on the safepoint are released. Note that if the lock were not actually held at the present moment in time by the bias owner, it would be correct to revert the object back to the "biasable but unbiased" state and re-attempt the CAS to acquire the bias. This possibility is discussed further in section 4.

If the CAS succeeded, subsequent lock acquisitions examine the object's mark word. If the object is biasable and the bias owner is the current thread, the lock is acquired with no further work and no updates to the object header; the displaced mark word in the lock record on the stack is left uninitialized, since it will never be examined while the object is biasable. If the object is not biasable, lightweight locking and its fallback paths are used to acquire the lock. If the object is biasable but biased toward another thread, the CAS failure path described in the previous paragraph will be taken, including the associated bias revocation.

When an object is unlocked, the state of its mark word is tested to see if the bias pattern is still present. If it is, the unlock operation succeeds with no other tests. It is not even necessary to test whether the thread ID is equal to the current thread's ID. If another thread had attempted to acquire the lock while the current thread was actually holding the lock and not just the bias, the bias revocation process would have ensured that the object's mark word was reverted to the unbiasable state.

Since the SFBL unlock path does no error checking, the correctness of the unlock path hinges on the interpreter's detection of unstructured locking. The lock records in interpreter activations ensure that the body of the monitorexit operation will not be executed if the object was not locked in the current activation. The guarantee of matched monitors in compiled code implies that no error checking is required in the SFBL unlock path in compiled code.

Figure 2 shows the state transitions of the mark word of an object under the biased locking algorithm. The bulk rebiasing edge, which is described further in sections 4 and 5, is only an effective, not an actual, transition and does not necessarily involve an update to the object's mark word. Recursive locking edges, which update the on-stack lock records but not the mark word, and the heavyweight locking state, which involves contention with one or more other threads, are omitted for clarity.

4. Bulk Rebiasing and Revocation

Analysis of execution logs of SFBL for the SPECjvm98, SPECjbb-2000, SPECjbb2005 and SciMark benchmark suites yields two insights. First, there are certain objects for which biased locking is obviously unprofitable, such as producer-consumer queues where two or more threads are involved. Such objects necessarily have lock contention, and many such objects may be allocated during a program's execution. It would be ideal to be able to identify such objects and disable biased locking only for them. Second, there are situations in which the ability to rebias a set of objects to another

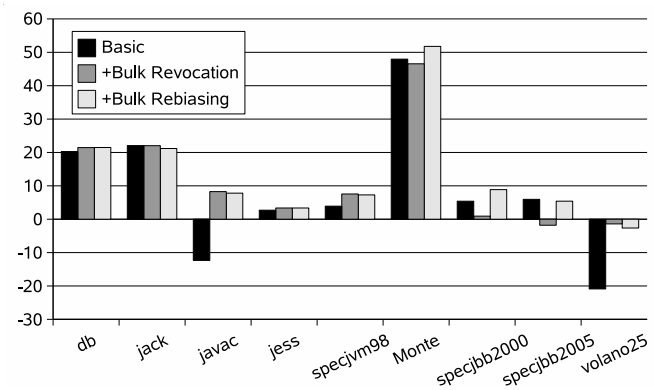

Figure 3. Percentage speedups yielded by basic SFBL algorithm and additions of first bulk revocation and then also bulk rebiasing.

thread is profitable, in particular when one thread allocates many objects and performs an initial synchronization operation on each, but another thread performs subsequent work on them.

When attempting to selectively disable biased locking, we must be able to identify objects for which it is unprofitable. If one were able to associate an object with its allocation site, one might find patterns of shared objects; for example, all objects allocated at a particular site might seem to be shared between multiple threads. Experiments indicate this correlation is present in many programs[6]. Being able to selectively disable the insertion of the biasable mark word at that site would be ideal. However, due to its overhead, allocation site tracking is to the best of our knowledge not currently exploited in production JVMs.

We have found empirically that selectively disabling SFBL for a particular data type is a reasonable way to avoid unprofitable situations. We therefore amortize the cost of rebiasing and individual object bias revocation by performing such rebiasing and revoking in bulk on a per-data-type basis.

Heuristics are added to the basic SFBL algorithm to estimate the cost of individual bias revocations on a per-data-type basis. When the cost exceeds a certain threshold, a *bulk rebias* operation is attempted. All biasable instances of the data type have their bias owner reset, so that the next thread to lock the object will reacquire the bias. Any biasable instance currently locked by a thread may optionally have its bias revoked or left alone.

If bias revocations for individual instances of a given data type persist after one or more bulk rebias operations, a *bulk revocation* is performed. The mark words of all biasable instances of the data type are reset to the lightweight locking algorithm's initial value. For currently-locked and biasable instances, the appropriate lock records are written to the stack, and their mark words are adjusted to point to the oldest lock record. Further, SFBL is disabled for any newly allocated instances of the data type.

The most obvious way of finding all instances of a certain data type is to walk through the object heap, which is how these techniques were initially implemented (Section 5 describes the current implementation). Despite the computational expense involved, bulk rebiasing and revocation are surprisingly effective.

Figure 3 illustrates the benefits of the bulk revocation and rebiasing heuristics compared to the basic biased locking algorithm[2]. The javac sub-benchmark from SPECjvm98 computes many identity hash codes, forcing bias revocation of the affected objects since there are no bits available to store the hash code in the biasable state (see figure 1). Bulk revocation benefits this and similar situations, here in particular because our early implementations performed relatively inefficient bias revocation in this case. SPECjbb2000 and SPECjbb2005 transfer a certain number of objects between threads as each warehouse is added to the benchmark, not enough to impact scores greatly but enough to trigger the bulk revocation heuristic. The addition of bulk rebiasing, which is then triggered at the time of addition of each warehouse, reclaims the gains to be had.

Note that the addition of both bulk revocation and rebiasing does not reduce the peak performance of biased locking compared to the basic algorithm without these operations. This is discussed further in Section 7.

[2] Machine configuration is "2xAMD" described in Section 6.

5. Epoch-Based Bulk Rebiasing and Revocation

Though walking the object heap to implement bulk rebias and revocation algorithms is workable for relatively small heaps, it does not scale well as the heap grows. To address this problem, we introduce the concept of an *epoch*, a timestamp indicating the validity of the bias. As shown in figure 1, the epoch is a bitfield in the mark word of biasable instances. Each data type has a corresponding epoch as long as the data type is biasable. 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.

With this scheme, bulk rebiasing of objects of class C becomes much less costly. We still stop all mutator threads at a safepoint; without stopping the mutator threads we cannot reliably tell whether or not a biased object is currently locked. The thread performing the rebiasing:

1. Increments the epoch number of class C. This is a fixed-width integer, with the same bit-width in the class as in the object headers. Thus, the increment operation may cause wrapping, but as we will argue below, this does not compromise correctness.
2. Scans all thread stacks to locate objects of class C that are currently locked, updating their bias epochs to the new current bias epoch for class C. Alternatively, based on heuristic consideration, these objects' biases could be revoked.

No heap scan is necessary; objects whose epoch numbers were not changed will, for the most part, now have a different epoch number than their class, and will be considered to be in the biasable but unbiased state.

The pseudocode for the lock-acquisition operation then looks much like:

Listing 1. Biased locking acquisition supporting epoch-based bulk rebiasing.

```
void lock(Object* obj, Thread* t) {
  int lw = obj->lock_word;
  if (lock_state(lw) == Biased
      && bias_epoch(lw) ==
         obj->class->bias_epoch) {
    if (lock_or_bias_owner(lw) == t->id) {
      // Current thread is the bias owner.
      return;
    } else {
      // Need to revoke the object's bias.
      revoke_bias(obj, t);
    }
  } else {
    // normal locking/unlocking protocol,
    // possibly with bias acquisition.
  }
}
```

Above we made the qualification that incrementing a class's bias epoch will "for the most part" rebias all objects of the given class. This qualification is necessary because of the finite width of the epoch field, which allows integer wrapping. If the epoch field is N bits wide, and X is an object of type T, then if 2^N bulk rebiasing operations for class T occur without any lock operation updating the bias epoch of X to the current epoch, then it will appear that X is again biased in the current epoch, that is, that its bias is valid. Note that this is purely a performance concern – it is perfectly permissible, from a correctness viewpoint, to consider X biased. It may mean that if a thread other than the bias holder attempts to lock X, an individual bias revocation operation may be required. But a sufficiently large value of N can decrease the frequency of this situation significantly: objects that are actually locked between one epoch and the next have their epoch updated to the current epoch, so this situation only occurs with infrequently-locked objects. Further, we could arrange for operations that naturally visit all live objects, namely garbage collection, to normalize lock states, converting biased objects with invalid epochs into biasable-but-unbiased objects. (If done in a stop-world collection this can be done with non-atomic stores; in a concurrent marker, however, the lock word would have to be updated with an atomic operation, since the marking thread would potentially compete with mutator threads to modify the lock word.) Therefore, wrapping issues could also be prevented by choosing N large enough to make it highly likely that a full-heap garbage-collection would occur before 2^N bulk rebias operations for a given type can occur.

In practice, wrapping of the epoch field can be ignored. Benchmarking has not uncovered any situations where individual bias revocations are provoked due to epoch overflow. The current implementation of biased locking in the Java HotSpot VM normalizes object headers during GC, so the mark words of biasable objects with invalid epochs are reverted to the unbiased state. This is done purely to reduce the number of mark words preserved during GC, not to counteract epoch overflow.

It is a straightforward extension to support bulk revocation of biases of a given data type. Recall that in bulk revocation, unlike bulk rebiasing, it is desired to completely disable the biased locking optimization for the data type, instead of allowing the object to be potentially rebiased to a new thread. Rather than incrementing the epoch in the data type, the "biasable" property for that data type may be disabled, and a dynamic test of this property added to the lock sequence:

Listing 2. Biased locking acquisition supporting epoch-based bulk rebiasing and revocation.

```
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) {
```

This variant of the lock sequence is the one currently implemented in the Java HotSpot VM.

Epoch-based rebiasing and revocation may also be extended to rebias objects at a granularity between the instance and class level. For example, we might distinguish between objects of a given class based on their allocation site; JIT-generated allocation code could be modified to insert an allocation site identifier in the object header. Each allocation site could have its own epoch, and the locking sequence could check the appropriate epoch for the object:

Listing 3. Code structure supporting allocation site-specific bulk rebiasing.

```
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[obj->alloc_site_id]) {
```

To simplify the allocation path for new instances as well as storage of the per-data-type epochs, a *prototype mark word* is kept in each data type. This is the value to which the mark word of new instances will be set. The epoch is stored in the prototype mark word as long as the prototype is biasable.

In practice, a single logical XOR operation in assembly code computes the bitwise difference between the instance's mark word and the prototype mark word of the data type. A sequence of tests are performed on the result of the XOR to determine whether the bias is held by the current thread and currently valid, whether the epoch has expired, whether the data type is no longer biasable, or whether the bias is assumed not held, and the system reacts appropriately. Listing 4 shows the complete SPARC assembly code for the lock acquisition path of SFBL with epochs.

6. Results

Figure 4 shows the performance impact of our biased locking and epoch-based bulk rebiasing and revocation technique on several industry-standard benchmarks on a variety of processor architectures and configurations[3]. (Figure 5 separates out the Monte Carlo benchmark from the SciMark suite to avoid distorting the graph.) These graphs illustrate not only the effectiveness of the technique, but also the relative cost of atomic operations on various CPUs.

Absolute performance comparisons with previous work [12, 10] are not possible, because as of this writing a JVM implementing the lock reservation technique has not been publicly released[4,5]. Even if such a JVM were available, isolating the effects of the lock reservation or biased locking techniques would be non-trivial. Differences in the optimizations performed by different dynamic compilers can cause even the relative speedup due to this optimization to differ; for example, if the overall generated code quality is low, the relative speedup due to biased locking might be less than if the overall code quality were high. We nonetheless observe that the magnitude of the improvements shown in figures 4 and 5 is comparable to previous work [12, 10].

Some benchmarks clearly improve dramatically, while others show little or only modest improvement. In the SPECjvm98 benchmark suite, the db, jack, javac, and jess benchmarks show the most

[3] 2xP4: 2-CPU 3.06 GHz Hyperthreaded Pentium IV, 4 GB RAM, Solaris 9
2xAMD: 2-CPU 1.8 GHz AMD Opteron, 2 GB RAM, Suse Linux 8 SP3
4xAMD: 4-CPU 2.2 GHz Dual-core AMD Opteron, 16 GB RAM, Solaris 10
2xUS-III: 2-CPU 750 MHz UltraSPARC III, 2 GB RAM, Solaris 8
1xUS-T1: 1-CPU 1.2 GHz 8-core (32-thread) UltraSPARC T1, 32 GB RAM, Solaris 10
32-bit JVMs used on all configurations.

[4] In the week before final submission of this paper, IBM released SPECjbb2005 scores with a JVM, due to ship in September 2006, supporting an `-XlockReservation` command line option for the first time.

[5] Since the first availability of biased locking in the Sun JVM, BEA has introduced an `-Xlazyunlocking` JVM command-line option, the implementation of which is undocumented.

Listing 4. SPARC assembly code for SFBL lock acquisition with epochs.

```
// inputs: Robj = pointer to object, Rmark = object's mark word, Rtemp = temporary
// globals: Gthread = register containing current thread pointer
// effects: sets condition codes before branching to DONE label.
//    EQUAL     => fast lock succeeded
//    NOT EQUAL => fast lock failed, go to slow case in run-time system
// Test whether the lock is currently biased toward our thread and whether the epoch is still valid
// Note that the runtime guarantees sufficient alignment of thread pointers to allow age to be
// placed into low bits
and3 Rmark, biased_lock_mask /* 0b111 */, Rtemp
cmp  Rtemp, bias_pattern     /* 0b101 */
br   notEqual, false /* annul next instruction */, pn /* predict not taken */, CAS_LABEL
ld_ptr [Robj + class_offset], Rtemp
ld_ptr [Rtemp + prototype_mark_offset], Rtemp
or3    Gthread, Rtemp, Rtemp
xor3   Rmark, Rtemp, Rtemp
andcc  Rtemp, ~age_mask, Rtemp
br     equal, true, pt, DONE_LABEL
nop
// The mark has the bias pattern and we are not the bias owner in the current epoch.
// Figure out more details about the state of the mark in order to know what operations can be
// legally performed on the object's mark.
// If the low three bits in the xor result aren't clear, that means the prototype mark is no longer
// biased and we have to revoke the object's bias.
btst biased_lock_mask, Rtemp
br   notZero, false, pn, TRY_REVOKE_BIAS
// Biasing is still enabled for this data type. See whether the epoch of the current bias is still
// valid. If not, attempt to rebias the object toward the current thread.
btst epoch_mask, Rtemp
br   notZero, false, pn, TRY_REBIAS
// Epoch of the current bias is still valid but owner is unknown. Try to acquire bias using an
// atomic operation. If this fails the object's bias will be revoked. Note that we first construct
// the presumed unbiased mark so we don't accidentally destroy another thread's valid bias.
and3 Rmark, biased_lock_mask | age_mask | epoch_mask, Rmark
or3  Gthread, Rmark, Rtemp
cas  Robj,    Rmark, Rtemp
// Test whether bias succeeded; if not, DONE path will revoke bias
cmp Rmark, Rtemp
br  always, false, pt, DONE
nop
TRY_REBIAS:
// Epoch has expired; attempt to acquire bias anew
ld ptr [Robj + class_offset], Rtemp
ld ptr [Rtemp + prototype_mark_offset], Rtemp
or3 Gthread, Rtemp, Rtemp
cas Robj,    Rmark, Rtemp
// Test whether bias succeeded; if not, DONE path will revoke bias
cmp Rmark, Rtemp
br  always, false, pt, DONE
nop
TRY_REVOKE_BIAS:
// Try to reset the mark of this object to the prototype value and fall through to the CAS-based
// fast locking. Note that if CAS fails, it means that another thread raced to revoke the bias of
// this particular object, so it's still okay to continue in the normal locking code.
ld_ptr [Robj + class_offset], Rtemp
ld_ptr [Rtemp + prototype_mark_offset], Rtemp
cas Robj, Rmark, Rtemp
// Fall through to the normal CAS-based lock
CAS_LABEL:
// Normal CAS-based locking code
// ...
DONE:
// Test of condition codes and call to slow case in run-time system if necessary;
// continuation of program
```

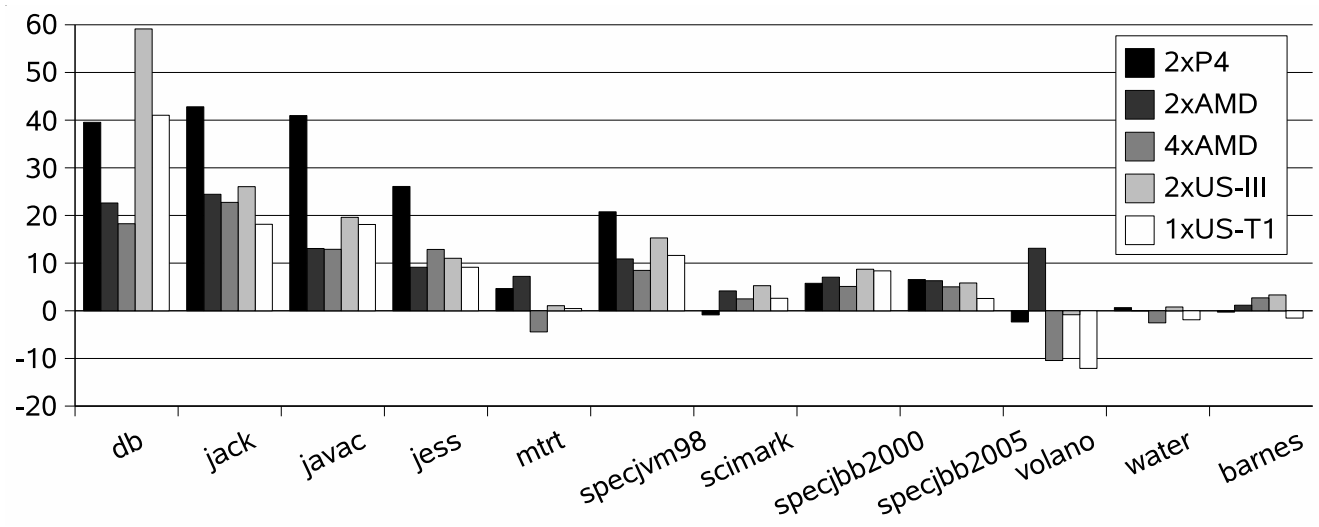

Figure 4. Percentage speedup/slowdown of SPECjvm98, SciMark, SPECjbb, Volano, and SPLASH-2 benchmarks due to biased locking.

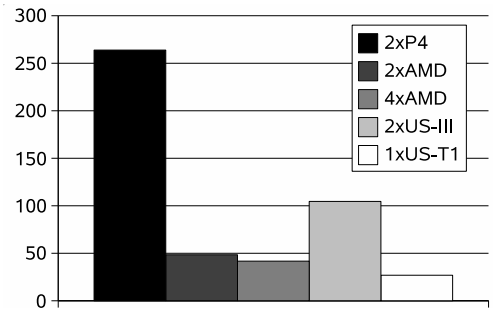

Figure 5. Percentage speedup of Monte Carlo benchmark from SciMark suite due to biased locking.

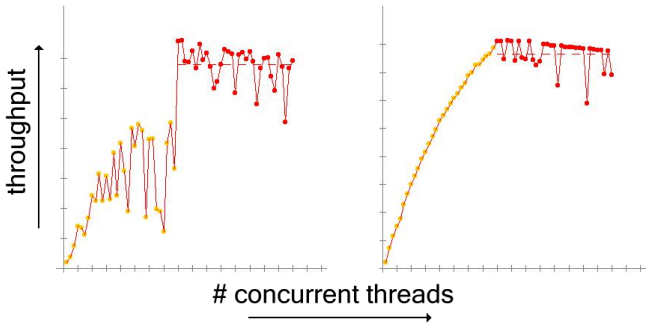

Figure 6. Scalability of a single-JVM SPECjbb2005 run on UltraSPARC T1 before and after the introduction of epoch-based bulk rebiasing and revocation.

improvement, while the others were largely unaffected. The composite score is improved by approximately 10-15%. In the SciMark suite, the Monte Carlo benchmark is greatly improved because its inner loop is dominated by synchronization overhead. The other benchmarks in this suite are largely unaffected by biased locking and due to the benchmark's scoring system only a roughly 5% overall speedup is attained. The SPECjbb2000 and SPECjbb2005 benchmarks net a 5-10% gain on most systems.

The SPLASH-2 [17] benchmarks Water and Barnes are multithreaded scientific applications ported to Java and analyzed by Salcianu and Rinard [14]. These two benchmarks have also been run under the lock reservation technique and its refinements [12, 10]. In our configuration, each of these benchmarks is run with 128 parallel compute threads over a duration of approximately 100 iterations per thread. We note that while our technique yields no speedup for these benchmarks, it also does not suffer the performance penalty of stop-the-owner lock reservation.

The results from these benchmarks indicate the relative cost of atomic operations on various CPUs. The Monte Carlo benchmark is effectively a synchronization microbenchmark and indicates that CMPXCHG is very costly on multiprocessor Intel x86 systems. The SPECjvm98 results support this conclusion, as the largest gains in this suite were also achieved on the multiprocessor Intel system. Multiprocessor UltraSPARC and AMD Opteron systems have better CAS and CMPXCHG performance, a conclusion again largely supported by the data. The db benchmark is an exception, where the largest gain was seen on an older UltraSPARC III system; we believe this may be related to relatively poor associativity of the data cache on this chip. The UltraSPARC T1 processor has a very cheap CAS instruction, so the gains from biased locking are relatively less on this architecture. The db benchmark from SPECjvm98 is again an outlier, which we believe again to be related to the data cache configuration on this chip.

The Volano benchmark is adversely affected by the SFBL algorithm. We believe this is due to the relatively high cost of per-object revocation in our system due to the need to reach a global safepoint. Volano in particular starts hundreds of threads to perform I/O, and the cost of a global safepoint increases as the number of concurrent threads increases. This is discussed further in section 7.

Figure 6 illustrates the scalability improvements of epoch-based bulk rebiasing and revocation. The graphs show the performance of a single-JVM run of the SPECjbb2005 benchmark on an UltraSPARC T1 processor. The maximum heap size is set to 3500 MB. In the left graph the bulk rebias and revocation operations are implemented by iterating through the object heap. In the right graph the epoch-based bulk rebias and revocation technique is used. In

Program name	# lock operations	% optimized locks	% opt. locks excl. first bias	% opt. locks fr. earlier work[9]
_201_compress	29712	80.308%	74.239%	31.547%
_202_jess	25128265	99.946%	99.808%	99.289%
_209_db	285377977	99.987%	99.935%	99.963%
_213_javac	77819074	99.918%	97.848%	99.402%
_222_mpegaudio	32131	87.632%	83.785%	35.837%
_227_mtrt	6318146	99.571%	99.523%	99.035%
_228_jack	77013610	99.993%	96.380%	91.947%
SPECjbb2000	786521693	94.258%	89.476%	58.544%
Volano Client	23449530	75.514%	75.481%	84.333%
Volano Server	19292861	76.980%	76.689%	79.755%

Figure 7. Percentages of lock operations optimized by biased locking.

this benchmark, bulk rebias operations tend to occur at the beginning of each measurement period, when another concurrent worker thread is added to the benchmark. (The cost of these operations is included in the throughput computation due to the nature of the benchmark.) The ragged throughput curve in the left graph indicates poor scalability, or alternatively a high degree of variance in throughput, due to the high cost of the associated heap iterations. Epoch-based bulk rebiasing and revocation clearly solve the scalability problems associated with these operations as the heap size increases.

Figure 7 provides as direct a comparison as possible of the effectiveness of biased locking with that of lock reservation [9]. Column 2 shows the number of lock operations executed in typical runs of several benchmarks[6]. The absolute number of lock operations executed for any particular benchmark is not crucial; it more generally indicates whether the benchmark is synchronization-intensive. The percentages in columns 3 and 4 indicate the fraction of these lock operations which were optimizable by our biased locking implementation. Column 3 counts the initial bias of the object toward this overall fraction, while column 4 counts only the number of subsequent successful biased lock acquisitions. We report both numbers in the interest of full disclosure, though the difference is most significant only for the SPECjbb2000 benchmark. The data in column 5 is that presented in Table 5 in [9]; the benchmarks were chosen to match those presented in this earlier work.

These statistics indicate that the effectiveness of biased locking compares favorably to that of lock reservation. The percentage of optimized lock operations in the synchronization-heavy SPECjvm98 benchmarks is roughly equal in both algorithms. Biased locking appears to be able to optimize a much larger percentage of the lock operations in the SPECjbb2000 benchmark. We believe this is attributable to bulk rebiasing, which is heuristically triggered soon after the addition of each new warehouse in a given run. It appears that this benchmark transfers significant numbers of biased objects between threads. However, we did not find that the relative speedup on this benchmark due to biased locking was significantly greater than that due to lock reservation. This indicates that multiple metrics must be used to evaluate such techniques, and also suggests that the additional optimized synchronized lock operations are not in the benchmark's critical code path. On the Volano benchmark, biased locking does not optimize as many lock operations as lock reservation. Note also that this is the benchmark on which biased locking yields a performance degradation rather than a gain. Here there appears to be a correlation between the percentage of optimized locks and the benchmark's overall performance, though this percentage alone is again not sufficient to completely evaluate the algorithm.

[6] Machine configuration is "2xAMD" described earlier.

7. Comparison to Earlier Work

SFBL is similar to, and is inspired by, lock reservation [9] and its refinements [12, 10]. Lock reservation is directly comparable to our basic biased locking technique described in Section 3. Both techniques eliminate all atomic operations for uncontended synchronization and have a severe penalty for bias revocation. Our technique avoids subtle race conditions because objects' headers are not repeatedly updated with non-atomic stores. However, because an explicit recursion count is not maintained, it is more difficult in our technique to determine at any given point in time whether a biased lock is actually held by a given thread.

The global safepoint required for bias revocation in our technique is more expensive than the signal used in lock reservation. It can be a barrier to scalability in applications such as Volano with many threads, many contended lock operations, and ongoing dynamic class loading. However, our experience has been that the combination of these characteristics in an application is rare. We have prototyped a per-thread safepoint mechanism and are investigating its performance characteristics. We also believe a less expensive per-object bias revocation technique is possible for uncontended locks while maintaining the useful locking invariants in the Java HotSpot VM, and plan to investigate this in the future.

Reservation-based spin locks [12, 10] are comparable to our addition of bulk rebiasing and revocation described in Section 4. Both techniques build on top of an underlying biased locking algorithm to reduce the impact of bias revocation. An advantage of reservation-based spin locks is that they largely eliminate, rather than reduce or amortize, the cost of bias revocation. However, reservation-based spin locks do not support transfer of bias ownership between threads. The first thread to lock a given object will always be the bias owner, and other threads will still need to use atomic operations to enter and exit the lock, eliminating the benefits of the optimization for these other threads. In contrast, epoch-based bulk rebiasing allows direct transfer of biases in the aggregate from one thread to another, at the cost of a small number of per-object revocations. Our experience indicates this supports optimization of significantly more synchronization patterns in real programs.

Neither reservation-based spin locks nor our algorithm optimize the case of a single object or small set of objects being locked and unlocked multiple times sequentially by two or more threads, but always in uncontended fashion. Our bulk rebiasing technique optimizes this case in the aggregate, when many such objects are locked in this pattern. Efficient optimization of this synchronization pattern is an important area for future research.

Reservation-based spin locks appear to adversely impact the peak performance of the lock reservation optimization as can be seen in the published results for db and jack [12, 10]. In contrast, epoch-based bulk rebiasing and revocation appear to reduce the adverse impacts of the biased locking optimization without impacting peak performance, as shown in Sections 4 and 6. We believe the high cost of per-object bias revocation in our system is responsible for the negative impact on the Volano benchmark, and plan to reduce this cost in the future. Nonetheless, feedback from customers indicates that our current biased locking implementation yields good results in the field with no pathological performance problems.

Speculative locking [7], another biased locking technique, eliminates all synchronization-related atomic operations, but requires a separate field in each object instance to hold the thread ID. This space increase makes the technique unsuitable for most data types. Additionally, speculative locking does not support the transfer of bias ownership from one thread to another, nor selective disabling of the optimization where unprofitable.

Previous lightweight locking techniques[1, 2, 5] exhibit quite different performance characteristics for contended and uncontended locking and contain very different techniques for falling back to heavyweight operating system locks under contention. Some of these techniques use only one atomic operation per pair of lock/unlock operations rather than two. Nonetheless, all of these techniques use at least one atomic operation per lock/unlock sequence so are not directly comparable to SFBL. Potentially, any of these techniques could be used as the underlying synchronization technique for SFBL or a similar biased locking technique.

8. Availability

Our technique is implemented in the current development version of the Java HotSpot VM. Binaries for various architectures and source code can be downloaded from `http://mustang.dev.java.net/`. The current build contains the per-data-type epoch-based rebiasing and revocation presented here. The biased locking optimization is currently enabled by default and can be disabled for comparison purposes by specifying `-XX:-UseBiasedLocking` on the command line.

9. Conclusion

Current trends toward multiprocessor systems in the computing industry make synchronization-related atomic operations an increasing impediment to the scalability of applications. Biased locking techniques are crucial to continued performance improvement of programming language implementations.

We have presented a new biased locking technique which optimizes more synchronization patterns than previous techniques:

- It eliminates repeated stores to the object header. Store elimination makes it easier to transfer bias ownership between threads.
- It introduces bulk rebiasing and revocation to amortize the cost of per-object bias revocation while retaining the benefits of biased locking.
- Epoch-based bulk rebiasing and revocation yield efficient bulk transfer of bias ownership from one thread to another.

Our technique is applicable to any programming language and virtual machine with mostly block-structured locking and a few invariants in the interpreter and dynamic compiler. It yields good performance increases on a range of benchmarks with few penalties, and customer feedback indicates that it performs well on Java programs in the field. We believe our technique can be extended to optimize even more synchronization patterns.

Acknowledgments

We thank David Cox, our manager, for supporting this work; Gilad Bracha, Dave Dice, Paul Hohensee, Vladimir Kozlov, and Tom Rodriguez for reviewing early drafts of this paper; and Martin Rinard and Alex Salcianu for access to their Java ports of the Water and Barnes benchmarks. We also thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y. S., and White, D. An efficient meta-lock for ubiquitous synchronization. In proceedings of OOPSLA '99, November 1999, pp. 207–222.

[2] Bacon, D. F., Konuru, R., Murthy, C., and Serrano, M. Thin locks: featherweight synchronization for Java. In proceedings of PLDI '98, June 1998, pp. 258–268.

[3] Bacon, D. F. and Fink, S. Method and apparatus to provide concurrency control over objects without atomic operations on non-shared objects. U.S. Patent Number 6,772,153, issued August 3, 2004. Assignee: International Business Machines Corporation.

[4] Bak, L. and Lindholm, T. G. Method and apparatus for concurrent thread synchronization. U.S. Patent Number 6,167,424, issued December 26, 2000. Assignee: Sun Microsystems, Inc.

[5] Dice, D. Implementing fast Java monitors with relaxed locks. In proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01), April 2001, pp. 79–90.

[6] Dice, D. Personal communication.

[7] Gomes, B. A., Bak, L., and Stoutamire, D. P.. Method and apparatus for speculatively locking objects in an object-based system. U.S. Patent Number 6,487,652, issued November 26, 2002. Assignee: Sun Microsystems, Inc.

[8] Griesemer, R. and Mitrovic, S. A compiler for the Java HotSpot™ virtual machine. The School of Niklaus Wirth, "The Art of Simplicity", January 2000, p.133–152.

[9] Kawachiya, K., Koseki, A., and Onodera, T. Lock reservation: Java locks can mostly do without atomic operations. In proceedings of OOPSLA '02, November 2002, pp. 130–141.

[10] Kawachiya, K. Ph.D thesis, Graduate School of Media and Governance at Keio University, 2005.

[11] Lindholm, T. and Yellin, F. The Java™ Virtual Machine Specification, Second Edition. Addison-Wesley, 1999.

[12] Onodera, T., Kawachiya, K., and Koseki, K. Lock reservation for Java reconsidered. In proceedings of ECOOP '04, June 2004, pp. 559-583.

[13] Paleczny, M., Vick, C., and Click, C. The Java HotSpot™ server compiler. In proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01), April 2001.

[14] Salcianu, A., and Rinard, M. Pointer and escape analysis for multithreaded programs. Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Snowbird, Utah, June 2001.

[15] Schmidt, R. W. System and method for facilitating safepoint synchronization in a multithreaded computer system. U.S. Patent Number 6,523,059, issued February 18, 2003. Assignee: Sun Microsystems, Inc.

[16] Stoodley, M. Accelerating Java synchronization in Just-In-Time compiler-generated code. 3rd Workshop on Compiler-Driven Performance, October 2004. `http://www.cs.ualberta.ca/~amaral/cascon/CDP04/`

[17] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A. The SPLASH-2 programs: characterization and methodological considerations. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 24–36, Santa Margherita Ligure, Italy, June 1995.