USENIX Association

Proceedings of the Java™ Virtual Machine Research and Technology Symposium (JVM '01)

Monterey, California, USA
April 23–24, 2001

Implementing Fast Java™ Monitors with Relaxed-Locks

David Dice
Sun Microsystems, Inc.
Burlington, MA
dice@computer.org

Abstract

The Java™ Programming Language permits synchronization operations (lock, unlock, wait, notify) on any object. Synchronization is very common in applications and is endemic in the library code upon which applications depend. It is therefore critical that a monitor implementation be both space-efficient and time-efficient. We present a locking protocol, the *Relaxed-Lock*, that satisfies those requirements. The Relaxed-Lock is reasonably compact, using only one machine word in the object header. It is fast, requiring in the uncontested case only one atomic compare-and-swap to lock a monitor and no atomic instructions to release a monitor. The Relaxed-Lock protocol is unique in that it admits a benign data race in the monitor unlock path (hence its name) but detects and recovers from the race and thus maintains correct mutual exclusion. We also introduce speculative deflation, a mechanism for releasing a monitor when it is no longer needed.

1. Introduction

The Java language [1] provides monitors [2][12] as the primary synchronization mechanism. Any object can be synchronized upon. In practice, however, most objects pass their lifetimes without ever being involved in synchronization activities. A Java Virtual Machine, or JVM, [3] must therefore provide the *potential* for synchronization but at a very low per-object space overhead. Synchronization operations occur frequently in applications and in the Java runtime library, albeit limited to a small subset of extant objects. It is thus critical that these operations have low latency to avoid degrading performance. Recent research in escape analysis and synchronization removal shows promise, but these techniques apply only to uncontended locking. Even though several recent papers have been published on JVM synchronization, the present work shows that substantial performance improvements can still be made.

The goals for our monitor implementation are thus space-efficiency, time-efficiency and scalability. The design described in this paper scales well to large numbers of threads and processors. It shows low latency for uncontended monitor operations and provides high throughput for contended monitors. It is space-efficient because it uses only one header word per object, and that header word may contain the object's hashCode value as well.

The Relaxed-Lock protocol is used to implement Java-level monitors. It is private to the JVM and is not visible to Java programs.

The design described in this paper was implemented in the Sun Laboratories Virtual Machine for Research (ResearchVM). ResearchVM was previously known as EVM, ExactVM and the Java 2 SDK Production Release. [4] describes another monitor implementation, called the Meta-Lock, based on ResearchVM. Our implementation compares favorably to the Meta-Lock, requiring only one atomic instruction and one memory barrier to lock and unlock an uncontended monitor, while the Meta-Lock algorithm requires two atomic instructions. Like the Meta-Lock, the Relaxed-Lock requires a word in the object header for synchronization. Both algorithms overload this header word to contain both the object's hashCode value and the synchronization word.

The Relaxed-Lock protocol has been implemented in ResearchVM for the Solaris™ Operating Environment, SPARC™ Platform Edition and Solaris IA32™. We present details of the SPARC implementation, but the design is portable to other operating systems and processor architectures. The protocol relies on an atomic compare-and-swap instruction (CAS on SPARC or CMPXCHG on IA32 processors).

2. The Basic Locking Protocol

The paper will first present a simplified form of the protocol and then describe optimizations and reduction to practice. This paper focuses on lock and unlock operations (sometimes called enter and exit, respectively)

that provide mutual exclusion, and doesn't discuss the `wait-notify` aspect of Java monitors.

2.1. Overview

Consider a thread attempting to lock an object. In the Relaxed-Lock protocol each object contains a LockWord field, which is either empty or contains a pointer to a monitor record. A Java thread locks an object with an empty LockWord by allocating a monitor record, writing its unique thread identifier value into the monitor record's Owner field and finally inserting the address of the monitor record into the LockWord using CAS. If the CAS succeeds then the thread has locked the object. We call this *inflation*. Inflation associates a monitor record with an object. If the lock attempt fails, the thread must block itself. To unlock an object upon which no other threads are blocked we deflate the object – we store the empty value into the object's LockWord and return the monitor record to the free pool. We deflate to conserve monitor records. Deflation breaks the relationship between a monitor record and an object.

Now consider multiple threads locking and unlocking a single object where those operations overlap in time. A thread locks an object that is already inflated by applying CAS to try to insert its unique identifier into the monitor record's Owner field. The locking thread may observe that the object's LockWord field refers to a particular monitor record *m*. Since the locking thread fetched this value, the LockWord may have changed – the unlocking thread may have deflated the object and returned *m* to the free pool. Other locking threads may have re-inflated the object with a different monitor record in the meantime. The monitor record *m* may have been reused and may now be associated with a different object. The locking thread holds a *stale* monitor record pointer. The Relaxed-Lock protocol tolerates references to the wrong monitor record. The locking thread will detect the stale pointer and recover as needed.

2.2. Data Structures

The primary data structures related to synchronization are the Execution Environment (*EE*), the object header and the monitor record (*monRec*). Unless otherwise noted, each is initially zero-filled. These structures are implementation specific and are not directly addressable by Java programs.

2.2.1. Execution Environment

An EE is a JVM internal data structure associated with each Java thread. Each EE is attached to a Solaris user-level thread. ResearchVM currently uses a 1:1 thread model – each Java-level thread runs on top of its own Solaris thread. There is only one field in the EE related to synchronization – namely, MonFree, a pointer to the EE's private monitor free list. We use the term thread and EE interchangeably. We say an EE is *waiting* when it has called an object's `wait` method and is awaiting notification. We say an EE is *blocked* when the thread is queued (non-ready) while attempting to lock or re-lock a monitor. In ResearchVM, an EE is uniquely identified by its address.

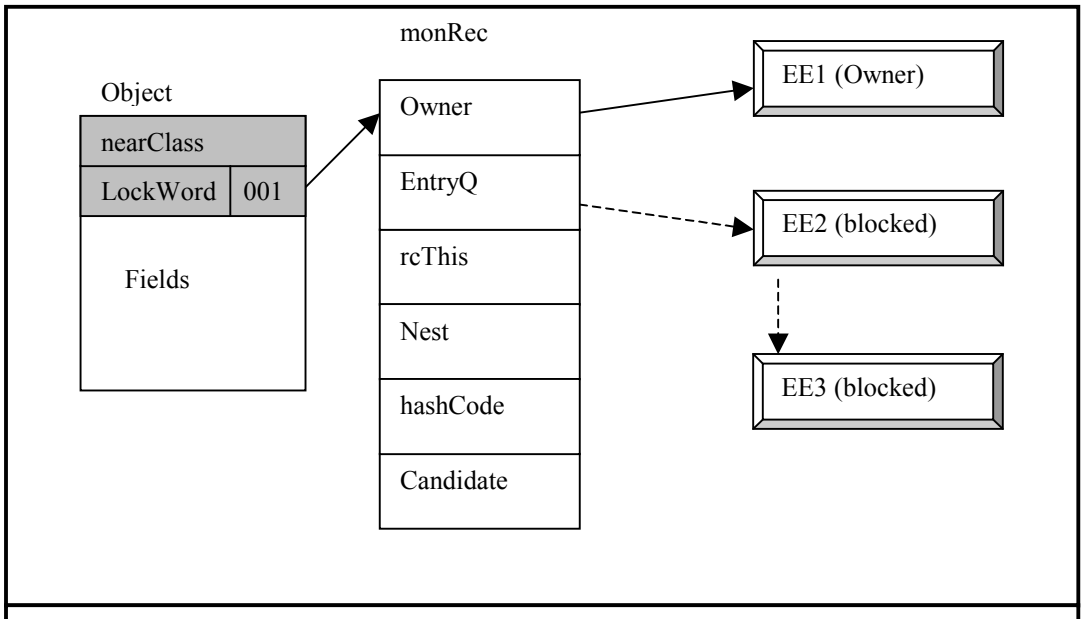

Figure 1. Example of an inflated object. EE1 has locked the object and EE2 and EE2 are blocked on entry.

2.2.2. Object Header

Objects in ResearchVM consist of an object header followed by the constituent fields. The object header consists of a nearClass field, which is used to identify the object type and to implement virtual function calls, and a LockWord field for synchronization. The object constructor initializes the LockWord field to zero. In ResearchVM an object reference is simply a pointer to the object's header.

We distinguish the LockWord state by the low-order bits of the LockWord field. All monRecs are allocated on 8-byte aligned addresses. As such the low-order 3 bits of monRec addresses are known to be zero, allowing encoding of up to 8 states. We use 2: In the *empty* state the LockWord value is zero. All objects are created in the empty state. In the *inflated* state the LockWord contains a pointer to a monRec.

An object may be associated with at most one monRec. Likewise, a monRec may be associated with at most one object at any one time. Many EEs, however, may block on a single object. An EE may block or wait on at most one object at a time. We say an object is locked when the LockWord points to a monRec having a non-NULL Owner field. We say an object is unlocked when its LockWord is empty or its LockWord points to a monRec having a NULL Owner field. We say EE1 owns object *o* when *o*'s LockWord points to some monRec *m* and *m*'s Owner field is set to EE1. An object must be inflated when it is locked or when any threads block or wait on it. Figure 1 shows an example: EE1 owns the object and EE2 and EE2 are blocked on the object.

2.2.3. Monitor Record

The *monRec* is an optional extension to the object header. If space consumption weren't critical, our design would simply embed all the monRec fields directly in the object header itself. Instead, we extend the object on-demand by storing a pointer to a monRec in the LockWord. The monRec is only needed when the object is locked or when threads are waiting or blocking on the object. Like the EE, the monRec is a JVM internal structure and is not a first-class Java object.

A monRec is either free or in use. It is free if it is not associated with any object, and, conversely, it is in use if it associated with an object. Free monRecs reside on per-EE free lists or on a JVM global free list. When a locking thread needs to allocate a monRec with which to inflate an object, it checks its own free list and then, failing that, resorts to the global free list.

The MonFree list is private to the EE. No synchronization is needed to add or remove elements as the list is only accessed by the EE itself. Under normal circumstances, the maximum number of monRecs that an EE needs on its free list is the maximum number of objects that it will lock simultaneously plus one on which to wait. Most EEs reach steady state early in their tenure and never need to grow their lists. In the presence of contention, however, monRecs can migrate between EE free lists. This can result in some EEs "hoarding" monRecs and having an undue number of monRecs in circulation. Unchecked, this could result in unbounded monRec allocation. To compensate, we can either trim and rebalance the free lists at garbage collection (GC) time or, alternatively, mark each monRec with the EE that created it. If an EE deflates an object and observes that the monRec was created by some other EE, it could return that monRec to either the global free list or a special per-EE lookaside free list. The per-EE list is strictly an optimization, but an important one. The Relaxed-Lock protocol would still work correctly with only the global free list.

The major fields in the monRec structure are as follows:

- Owner
  The Owner field is a pointer to the EE that has locked the monRec. The field is NULL if no thread owns the monRec – that is, if the object is unlocked.
- Candidate
  The Candidate field is used to implement *futile wakeup throttling*. See below.
- rcThis
  rcThis is a reference count which indicates the number of threads blocked or waiting on the monRec. The rcThis value does not include the current lock owner itself, if there is one. The rcThis field protects a monRec against inadvertent deflation. The field is updated using atomic fetch-and-add instructions.
- Nest
  Java monitors permit recursive (reentrant) locking. A Java thread can re-lock an object that it already owns without blocking. Nest, initially one after inflation, reflects the recursion depth.
- EntryQ
  The EntryQ is a heavyweight Solaris system semaphore [9] on which we block threads attempting to lock the monRec.

Our implementation requires that the monRecs reside in *type-stable-memory*, or TSM [6][7]. Simply put, once a memory location holds a monRec it must always hold a monRec; the memory is now typed. A monRec must tolerate references from threads that hold stale monRec pointers. The referencing thread will, by examining the monRec and the object LockWord, discover that the reference is stale with respect to the object and recover as needed. We can relax the strict TSM requirement

somewhat in ResearchVM. ResearchVM supports a *stop-the-world* GC model. At a stop-the-world point all Java threads are blocked at known safe points – no thread can be executing in a code path where it holds a stale monRec pointer. At these points we can safely reclaim free monRecs and reuse the underlying memory.

2.3. Run-time support

The following example illustrates the binding between Java source code, Java bytecode and the JVM internal service routines. The Java compiler, javac, translates the Java source code into bytecode. The just-in-time compiler (JIT) in the JVM translates the bytecode into native code at run-time. Execution of monitorenter and monitorexit bytecodes results in calls to `monEnter` and `monExit`, which are "C" routines within our JVM. Java provides both `synchronized` methods and `synchronized` statements. The Relaxed-Lock protocol services both forms in the same manner.

Java Source Code

```
[1]    synchronized (obj) { obj.Value ++ ; }
```

Java Bytecode

```
[1]    push obj
[2]    monitorenter
[3]    push obj
[4]    dup
[5]    getfield #Value
[6]    iconst_1
[7]    iadd
[8]    putfield #Value
[9]    push obj
[10]   monitorexit
```

SPARC Native Code

```
[1]    mov   ee, %o0       ! ee is the current thread
[2]    call  monEnter      ! call monEnter (ee,obj)
[3]    mov   obj, %o1      ! delay slot, pass object
[4]    ld    [obj].Value, tmp
[5]    add   tmp,1,tmp     ! increment obj.Value
[6]    st    tmp, [obj].Value
[7]    mov   ee, %o0       ! ee is the current thread
[8]    call  monExit       ! call monExit (ee,obj)
[9]    mov   obj, %o1      ! delay slot, pass object
```

2.3.1. monEnter

A thread can lock an object in one of two ways:

(1) By Inflation

A thread can lock an object that is deflated by successfully inflating the object. The thread uses CAS to attempt to install a monRec in the object's LockWord. The monRec's Owner field has been set to the thread's EE prior to the CAS A successful CAS confers ownership.

(2) By directly locking the associated monRec.

A thread can lock an object that is already inflated by using CAS to transition the Owner field from NULL to the thread's own EE address.

Under the Relaxed-Lock protocol all internal locking is potentially contended. Consider a thread blocked trying to lock a monitor. After waking, the thread must recontend for a monitor by either inflating the object or by attempting to CAS the Owner field in the associated monRec. Waking up doesn't imply ownership of a lock, but rather grants that thread an opportunity to compete for the lock. Blocking and waking threads is simply a way to avoid spinning.

The Relaxed-Lock protocol uses a two-level synchronization model. Java synchronization primitives that don't involve blocking or waking threads are satisfied in the JVM itself. We call this *fast path* synchronization. Locking an uncontended monitor, for instance, requires executing an atomic CAS instruction in the JVM but doesn't involve any heavyweight system synchronization services. On the SPARC processor the fast path lock primitive is only 13 instructions long. The JVM only resorts to the *slow path*, which uses heavyweight Solaris synchronization primitives [9], to explicitly block or wake threads. Most synchronization requests are satisfied via the fast path.

Locking proceeds as follows. First, the `monEnter` routine fetches the object LockWord and examines the low-order bits, which encode the state.

Case 1: Empty (fast path)

The locking thread must allocate a new monRec to install in the object's LockWord [Listing 1, Line 33]. It checks, in the following order, the EE's monRec free list, the global list and, if still unsatisfied, finally constructs a monRec using `malloc`. A subtle but important point is that all monRec on a thread's free list are known to have their Nest field preset to one and the Owner field preset to the EE. Such monRecs are immediately ready to be installed in the object's LockWord field.

Once a monRec is allocated we try to inflate the object by using CAS to install the monRec's address in the LockWord [Listing 1, Line 37]. If the CAS fails then we've encountered interference; another thread changed the LockWord between the initial load and the CAS. In the case of interference, we simply restart the `monEnter` routine and retry the entire operation. The algorithm is lock-free [10] as at least one thread will have made forward progress. Successfully installing the monRec confers ownership to the calling thread. For uncontested locking this is the most frequently executed

path and it largely determines synchronization performance.

Case 2: Inflated and already locked by the calling thread

If the LockWord is inflated then the LockWord value contains a pointer to a monRec. We examine the Owner field in the monRec. If it is equal to the caller's EE then this is a case of reentrant locking. We simply increment the monRec's Nest field and return. This path requires no atomic operations [Listing 1, Line 46].

Case 3: Inflated and unlocked

As above, we convert the LockWord value to a monRec pointer. In this case the monRec's Owner field is NULL, indicating the object is unlocked. This can occur when there are threads waiting for notification on the object but the object is unlocked. As there are waiting threads, the monRec's rcThis value will be greater than zero. The acquiring thread attempts to transition the Owner field from NULL to its EE via CAS. If the CAS fails then we fall into the slow path (case 4). If the CAS succeeds then we've locked a monRec, although not necessarily the correct monRec.

Between the initial fetch of the LockWord and the successful CAS, there is a timing window during which another thread may have changed the object's LockWord. We check for this by re-fetching the LockWord and verifying that it is still the same [Listing 1, Line 52]. If the LockWord is unchanged then the thread has successfully locked the object – `monEnter` returns. If the LockWord has changed then the thread has locked the wrong monRec. We say that the monRec pointer is stale with respect to the object. In this case the thread releases the monRec, wakes up any threads that may have blocked because they observed that the monitor was locked, and, finally, retries the entire operation. Transiently locking the wrong monRec is harmless.

Case 3 is an optimization and is not strictly necessary to the protocol. We encounter Case 3 when one or more threads are waiting on an object, the object is unlocked and our thread attempts to the lock the object. Case 3 allows us to lock an object with only one atomic instruction. The protocol would still work correctly if we eliminated case 3 and simply used case 4 (the slow path). Case 4 can handle any inflated object but requires a minimum of 3 atomic instructions.

Case 4: inflated and locked by another thread

The object is contended and we now take the slow path and prepare to block the calling thread. First, `monEnter` atomically increments the rcThis reference count field to indicate that another thread is blocked on the monRec. [Listing 1, Line 64]. While incrementing the reference count another thread may have broken the associating between the object and the monRec. We check for this by re-fetching the object's LockWord and insuring it still points to the expected monRec. This is similar to Valois' `SAFEREAD` technique [10]. Next, we resample the Owner field to see if the owner thread relinquished the object between the original fetch of the LockWord and this point. This closes a timing window and prevents lost wakeups. If the Owner field was observed to be NULL the acquiring thread applies CAS to replace the NULL value with its EE address [Listing 1, Line 74]. If the CAS succeeds then the thread has locked the object; we decrement the rcThis field, set the Nest field to one and return. Otherwise, the thread failed to lock the monitor and it now blocks itself on a heavyweight Solaris semaphore associated with the monRec.

Upon waking, the thread checks to see if has been *flushed*. Flushing is an exceptional condition and is described in more detail below, in the `monExit` section. A thread can wake either because it's been flushed or, normally, because the prior owner has released the monitor and arranged that this thread awake as the "heir apparent" owner of the object. If the object's LockWord still points to the expected monRec then the thread attempts to lock the object by using CAS to store its EE into the Owner field. If the CAS succeeds the thread has locked the object – it then decrements the reference count, sets the Nest field to 1 and returns. If the CAS fails, the thread simply re-blocks on the semaphore. If the thread observes that the object's LockWord has changed, then it has been flushed. The pointer to the monRec is stale; the locking thread decrements the rcThis reference count and restarts the entire `monEnter` path.

2.3.2. MonExit

The `monExit` routine releases the monitor associated with an object [Listing 1, Line 86]. In the Relaxed-Lock protocol only the owner of an object may deflate it, and then only at unlock time. `MonExit` starts by fetching the object's LockWord. If the LockWord is uninflated or if the current thread is not the Owner of the object then `monExit` throws an *IllegalMonitorState* exception as required by the Java Language Specification. `monEnter` then decrements the Nest field. If the monitor is recursively locked, `monExit` simply updates the monRec's Nest field and returns [Listing 1, Line 116].

If decrementing the Nest field results in zero then we must release the object. Note that we don't store the zero value in the Nest field. We leave the value at one to avoid the store – the monRec is immediately ready for its next incarnation and can be added to the per-EE free list without any further processing. If the unlocking thread

observes that the rcThis field is non-zero then it will leave the object inflated as there are other threads legitimately waiting on the monitor. The unlocking thread then marks the Owner field as NULL and, to ensure progress, wakes up one of the threads attempting to lock the monitor.

If the unlocking thread observed that the rcThis field was zero then there *appear* to be no other threads referencing the monitor [Listing 1, Line 96]. In this case we attempt *speculative deflation*. First, we deflate the object by restoring the empty value into the LockWord. This dissociates the object and the monRec. Next, after restoring the LockWord, the unlocking thread re-fetches the rcThis field. In the normal case, the value will still be zero and the thread simply adds the monRec to its free list and returns. If, however, the rcThis field is non-zero then we've misspeculated and must take special action.

We call this speculative deflation, as the unlocking thread previously observed that rcThis was zero, but that might not remain true at the actual point of deflation. In the time between fetching rcThis and deflating the object, another thread may have arrived in `monEnter`, attempted entry and incremented rcThis. The entering thread may also have blocked itself on the monRec's EntryQ. Put another way, the code in `monExit` could inadvertently deflate an object while another thread was in the process of trying to lock that same object. The result of misspeculation is that a locking thread could block on a stale monRec with undesirable consequences – the thread could be stranded indefinitely or it could lock the wrong monRec and violate the mutual exclusion constraint. To recover from misspeculation we must flush the monRec:

Flushing in the unlocking thread:

> The unlocking thread detects misspeculation by noticing that the rcThis field changed from zero to non-zero during deflation. This indicates that threads tried to lock the object while it was deflating. The unlocking thread then wakes up all the threads that attempted to lock the object during the timing window. In our implementation, the flushing thread must wait for all the flushees (victims) to acknowledge the flush. This way the monRec's Owner field stays non-zero and tardy threads can't inadvertently lock the wrong monRec. Once the flush is completed and all blocked threads are known to have vacated the monitor the unlocking thread adds the monRec to its private free list.

Flushing in the locking thread:

> The locking thread wakes up and notices that the object's LockWord field is not the expected value. This indicates that the thread has been flushed. When a thread recognizes that it has been flushed, it decrements the monRec's reference count field and retries the entire `monEnter` operation.

Misspeculation occurs rarely as the window of vulnerability is short. [Listing 1, Lines 96-97]. On SPARC, the window – in `monExit`, between the fetch of the rcThis field and the deflating store into the object's LockWord field – is only 6 instructions long. By tolerating this timing window we are able to remove all atomic instructions from the uncontested unlock path. Instead of preventing references to stale monRecs we detect and recover as needed. Misspeculation can't occur when only one thread is accessing an object (no contention) or when the many threads block on an object, in which case the object remains inflated (heavy contention).

The protocol deflates monitors in order to limit the number of monRecs in circulation. An alternative to speculative deflation is to defer deflation until GC-time. The collector could scan and deflate as needed. Deflating at GC-time also makes for a very slightly faster unlock path as we don't need to check reference counts or to deflate. Our implementation uses speculative deflation, however, as it is less coupled to the garbage collection subsystem. In addition, speculative deflation is aggressive, and deflates an object as soon as possible. This minimizes the number of monRec in circulation. Contrast this to Bacon's *Thin Lock* scheme [8][15] where objects, once inflated, stay inflated for their lifetime.

The rcThis field is a hint used to guide deflation. If the rcThis is value is non-zero in `monExit` [Listing 1, Line 96], then the monRec is in use and is not eligible for deflation. It is safe to sample rcThis before releasing the object – while the object is locked rcThis can only increase; it can never transition to zero. We are not at risk of missing deflation and leaking monRecs. If rcThis is zero then the monRec is idle and is eligible for deflation.

2.3.3. Wait-Notify

The `wait-notify-notifyAll` portion of the monitor subsystem isn't described in this paper. We should note, however, that the monRec's rcThis includes the number of threads waiting on the monitor. If there are any threads `waiting` on an object then the object must be inflated. The wait-notify subsystem is largely decoupled from the lock-unlock portion of the synchronization protocol.

3. Augmenting the Basic Protocol

3.1. Safely incrementing rcThis.

As described above the Relaxed-Lock protocol suffers a timing window that permits monRecs to leak. Consider

a `monEnter` call that encounters an inflated and locked monitor. The locking thread must increment the monRec's rcThis reference counter. It may, however, have incremented the rcThis value of a stale monRec pointer. The locking thread recognizes that the pointer is stale and compensates by decrementing the rcThis field. Unfortunately an exiting thread could have observed the rcThis field when it was temporarily (and improperly) non-zero. This, in turn, could cause `monExit` to miss the deflation of the object currently associated with the monRec. Missed deflation would result in leaking monRecs.

To compensate for this timing window `monEnter` puts the potentially leaked monRecs onto a special SuspectList [Listing 1, Line 67]. A monRec is suspect if it may have missed being speculatively deflated. The GC thread scans and cleans the SuspectList at GC-time. Specifically, the GC thread will examine the monRec and the associated object and perform any deflation that may have been missed. The GC thread is able to safely perform the recovery because at GC-time all the Java threads will be stopped at known locations. Because we use a stop-the-world GC mechanism, when the GC thread executes we know that no normal Java threads will be holding stale monRec pointers or executing in a vulnerable region.

The SuspectList allows us to detect and recover from the timing window in monEnter. In the full-length paper we describe another technique that *prevents* a thread holding a stale monRec pointer from incrementing the rcThis field. The idea is based on *word-tearing*; we use mixed-size load, store and CAS operations to access collocated fields in the monRec. Briefly, we define a composite field in the monRec structure. The composite field contains two subfields, rcThis and the Guard, that are separately addressable with load and store instructions. The composite field is addressable with atomic load, store and CAS instructions. The basic idea is the Guard changes when the monRec recycles (deflates). To increment the rcThis subfield we use CAS on the composite field. The CAS will fail if the monRec is stale. The CAS simultaneously validates that the monRec has not recycled and conditionally increments the reference count. Word-tearing is processor and memory-model dependent. It is not portable but is known to work on current SPARC and Pentium™ processors.

3.2. hashCode multiplexing

We now provide a brief sketch of the changes needed to let the LockWord and the object's hashCode cohabit in one header word.

Each Java object may have a hashCode value associated with it [5]. The hashCode, once assigned, is persistent with respect to that object. The hashCode values for a set of objects should have a reasonable distribution as they are often used as keys for hash tables.

ResearchVM realizes hashCode values on-demand. As such, an object's LockWord may be empty, contain the hashCode value or be inflated. We use the low order bits to distinguish the contents. At inflation time, in `monEnter`, the locking thread copies the hashCode from the LockWord and stores it in the monRec [Listing 1, Line 35]. At deflation time the unlocking thread copies the hashCode back into the LockWord [Listing 1, Line 97]. [4] attributes the idea of the displaced header words to Lars Bak in the HotSpot VM. To avoid timing hazards, the first time we inflate an object, if a hashCode has not been assigned, we generate a hashCode and associate it with an object. An inflated object always has a hashCode value.

We compute the hashCode as the XOR of the object's current address and a global gcHash value. This calculation is extremely fast and doesn't impact synchronization performance. The gcHash value is recomputed using a Park-Miller [16] random number generator at each GC-point. In a sense, the address provides a spatial component to the hashCode and the gcHash value contributes a temporal component. The gcHash component is particularly important as, when using a copying garbage collector, heap addresses tend to be reused and don't have a good distribution.

3.3. Futile wakeup throttling

Consider the following policies used to activate a successor thread when a thread unlocks an object. In *directed handoff* the unlocking thread explicitly picks a successor from the list of blocked threads, marks that thread as the owner of the object and then wakes it. The distinguished successor, by virtue of waking, knows that it owns the object. Directed handoff is strictly fair, assuming a LIFO list. When multiple threads repeatedly contend for the same object, however, the directed handoff policy results in high levels of context switching. Assume a typical parallel program that executes the following loop: lock a shared object, execute serial work, unlock the object, execute parallel work. If multiple threads execute the loop they will contend for the shared lock. If the locking is fine-grained and duration of the "execute parallel work" and "execute serial work" phases is short then cost of the context switching will dominate performance

In *competitive handoff*, when a thread unlocks an object, it marks the object as available and then makes a *potential* successor thread (sometimes called the heir apparent) ready. The successor, upon waking up, must

compete for the object like other threads. Waking a successor ensures progress. Competitive handoff is inherently unfair as one thread may dominate the lock. By avoiding excessive context switching and by keeping "hot" threads running it usually provides the best system throughput. Competitive handoff relies on system-level thread preemption to provide a coarse level of fairness. The ResearchVM Meta-Lock and the Relaxed-Lock protocol both use competitive handoff.

Competitive handoff suffers from the futile wakeup problem. When multiple threads repeatedly compete for the same lock, one thread tends to dominate and the remaining threads tend to migrate back and forth between the monitor's EntryQ and the system ready queue. In particular, the successor threads will often wake up, fail to grab the lock and re-block on the monitor. It fails to acquire the lock because the previous owner, which made the successor ready, has reacquired the lock in the interim. This is a *futile wakeup*. The threads eventually make progress, but suffer from degraded performance because of the excessive context switching. To avoid this effect the Relaxed-Lock protocol uses *futile wakeup throttling*. Instead of permitting an unbounded number of successors to be ready, we permit at most one. At any one time, we only need one heir apparent to ensure that the computation makes forward progress. Having more than one heir apparent is unnecessary and inefficient. Throttling greatly reduces the futile wakeup rate.

To implement throttling we use a field in the monRec called Candidate. Candidate is set to one to indicate that the next time the thread releases the monitor it should also wakeup a successor. The field is set to zero to indicate that no wakeup is needed. To be precise, zero means that either no threads are blocked on the monitor or that a successor has been make ready but has not yet come awake. Throttling greatly improves the performance of Java applications that have many threads contending heavily for a single object. [Listing 1, Lines 19 and 72]. Throttling is an optimization and is not fundamental to the Relaxed-Lock scheme.

To demonstrate the utility of throttling consider a Java program, RandBash, that has 24 threads, running in parallel, each of which loops calling the `java.util.Random.nextInt` method for a shared object 1000000 times. The `nextInt` method calls a worker routine that is synchronized. On an 8-way 333 MHz SPARC system running Solaris 2.8 we have the following results:

JVM	seconds
ResearchVM with Meta-Lock	141
ResearchVM with Relaxed-Lock, throttling enabled	40
ResearchVM with Relaxed-Lock, throttling disabled	119

Table 1 Throttling Results.

3.4. Fast Assembly Language Paths

In order to improve performance, our implementation, like Meta-Lock, uses fast, specialized forms of `monEnter` and `monExit` to handle uncontested locking. These routines are written in assembly language. Space permits us from describing these further.

4. Results

	JVM		
Benchmark	**Meta-Lock**	**Relaxed-Lock**	
VolanoMark 2.1.2 -- The VolanoMark benchmark, created by Volano LLC, predicts the performance of an internet chat server. We tested it with the rooms parameter set to 10 and the message count parameter set to 100. VolanoMark executes a large number of uncontended synchronization operations.	14925	15346	msgs/sec
pBOB 1.2 (Portable BOB) -- pBOB was created by IBM to model the performance of object databases. It uses a random number generator to create the synthetic workload. The random number generator class is protected by a static monitor and is highly contended. pBOB scores reflect the throughput of a large number of threads passing through a single critical section.	49282	137349	tpmBOB

Contend – Contend, like pBOB, tests the ability of a JVM to handle high levels of contention on a single monitor. It uses 24 threads executing concurrently, each of which iterates 100000 time over a loop. The loop body consists of a parallel portion, which takes 1.55 μsecs to complete and a serial portion, protected by a global monitor, which also requires 1.55 μsecs to complete.		23.771	8.993	secs
SPECjvm98				
_201_compress	LZW compress and decompress	39.205	39.209	secs
_202_jess	Java Expert Systems Shell	18.278	17.316	secs
_209_db	Simulates a database – search and update	60.931	59.045	secs
_213_javac	Java source to bytecode compiler	32.659	31.867	secs
_222_mpegaudio	Compress an audio file	42.916	41.602	secs
_227_mtrt	Multithreaded ray tracer	11.466	10.288	secs
_228_jack	Self-generating parser generator	23.643	22.456	secs
Java Grande Forum Benchmark, Version 2.0. Section1:Method:Same:SynchronizedInstance		5027309	6727437	calls/sec
Sync – Sync is a single-threaded micorbenchmark that times the execution of uncontested synchronization operations. The synchronized statements and methods are empty.				
1M normal calls to an empty method		30	30	msecs
No waiting threads, 1M synchronized method calls		251	179	msecs
No waiting threads, 1M synchronized method calls – nested		125	114	msecs
No waiting threads, 1M synchronized statements		244	207	msecs
One waiting thread, 1M synchronized method calls		451	246	msecs
One waiting thread, 1M synchronized method calls – nested		296	114	msecs
One waiting thread, 1M synchronized statements		462	212	msecs
CHashMapTest – CHashMapTest, written by Doug Lea, exercises his mostly-concurrent reading, exclusive writing HashMap package. As run, it has 4 threads concurrently applying 1000000 updates each to a HashMap having 100000 elements.		19.249	5.733	secs

Table 2 Benchmark Results.

All tests were run on an 8-Way 333MHz SPARC system running Solaris 2.8 in the Interactive Scheduling class. The Relaxed-Lock JVM used fast assembly language paths, speculative deflation, hashCode multiplexing and futile wakeup throttling.

As shown above, the Relaxed-lock protocol takes 179 milliseconds to complete 1 million calls to an empty synchronized method. For comparison, optimized "C" code runs 1 million `mutex_lock-mutex_unlock` pairs in 236 milliseconds. We should note, however, that for "C" code a large component of the cost is the control transfer through the procedure linkage table.

5. Conclusions

5.1. Recap

We have presented the Relaxed-Lock protocol that supports Java monitor semantics. It has been validated in the context of ResearchVM. It is time-efficient, reasonably space-efficient and holds up well under contention. For uncontended locking it requires only one atomic CAS to lock an object and only a memory barrier to unlock and object. On most processors atomic instructions are very expensive so the number of atomic instructions in the lock-unlock code path determines synchronization performance. ResearchVM with Relaxed-Lock actually has a longer lock-unlock path in terms of instruction count than the Meta-Lock form, but the Relaxed form has lower latency and can sustain a higher throughput because it

has one less atomic instruction. As we use fewer atomic instructions, The Relaxed-Lock protocol incurs less memory bus traffic and scales better on multiprocessor systems. It also provides predictable performance and is free of pathologies. The synchronization portion of the JVM stands alone and is largely independent of the rest of the JVM, and in particular the garbage collector.

5.2. Future work

In the future we may investigate using model checkers to formally validate the Relaxed-Lock protocol. Also of interest is making the monRec a first-class Java object. This would greatly simplify management of the monRec pool. In the current implementation the monRec contains a Solaris semaphore, and therefore can't be moved by the garbage collector. To avoid this problem we'd put a system semaphore in each EE and do away with the semaphore in the monRec. An EE would always block on its own semaphore. The monRec EntryQ would then become a pointer to an explicit list of EEs blocked on an object. Meta-Lock uses an explicit linked list of EEs. This would also give the JVM considerable control over short term scheduling policy. We would also be able to eliminate the rcThis field and use the explicit linked list pointer as indication that the monRec was idle.

Flushing, while rare, victimizes the unlocking thread. To keep the monRec out of circulation the exiting thread must wait until all flushees rendezvous. Intuitively, this seems unfair. One remedy would be to hand off the monRec to a special thread dedicated to flushing.

5.3. Acknowledgements

I'd like to thank Ole Agesen, Paula J. Bishop, Alex Garthwaite, Maurice Herlihy, Paul Hohensee and Doug Lea for useful suggestions.

6. References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series, Addison-Wesley, 1996.

[2] C.A.R. Hoare. Monitors: An Operating Systems Structuring Concept. *CACM* 17(10), pp. 549, October 1974.

[3] Timothy Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series, Addison-Wesley, 1996.

[4] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, Derek White. An Efficient Meta-Lock for Implementing Ubiquitous Synchronization.. In *Proceedings of ACM SIGPLAN '99 Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA), 1999.

[5] Ole Agesen. Space and Time-Efficient Hashing of Garbage-Collected Objects. *Theory and Practice of Object Systems*, Volume 5, Number 2, 1999, pp. 119.

[6] Michael Greenwald and David Cheriton. The Synergy Between Non-Blocking Synchronization and Operating Systems Structure. *Proceedings of the Second Symposium on Operating Systems Design and Implementation (OSDI)*, USENIX, Seattle, October 1996, pp. 123

[7] Michael Greenwald. Ph. D. Thesis. Non-Blocking Synchronization and System Design. Stanford University, 1999.

[8] David F. Bacon, Ravi Konuru, Chet Murthy and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. *In Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998. pp. 258.

[9] B. Lewis, D. Berg. *The Threads Primer: A Guide to Multithreaded Programming*. Sunsoft Press, Prentice Hall, 1996.

[10] John Valois. Lock-Free Data Structures. Ph. D. Thesis, Rensselaer Polytechnic Institute, 1995.

[11] David L. Weaver, Tom Germond, editors. *The SPARC Architecture Manual, Version 9*. SPARC International, Prentice-Hall, 1994.

[12] Peter A. Buhr, Michel Fortier, Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, 27(1), pp. 63-107. March 1995.

[13] Y. Oyama, K. Taura, A. Yonezawa. Executing Parallel Programs with Synchronization Bottlenecks Efficiently. University of Tokyo, 1998.

[14] Tamiya Onodera, Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proceedings of ACM SIGPLAN '99 Conference on Object-Oriented Programming Languages, Systems and Applications* (OOPSLA), 1999.

[15] R. Dimpsey, R. Arora, K. Kuiper. Java server performance: A case study of building efficient, scalable JVMs. IBM Systems Journal, Volume 39, no. 1, 2000 – *Java Performance*.

[16] S.K.Park, K.W. Miller. Random Number Generators: Good Ones Are Hard to Find. *CACM* 31(10), pp. 1192, October 1988.

7. Appendix – Listing

```
[1]    typedef struct _monRec {                // monitor record
[2]        struct _ExecEnv * volatile Owner ;  // EE, NULL iff free
[3]        volatile int Candidate ;            // wait indicator -- futile wakeup throttle
[4]        volatile int rcThis ;               // ref count -- holds monRec persistent
[5]        int Nest ;                          // recursive locking
[6]        int hashCode ;                      // hoisted from object.LockWord
[7]        struct _monRec * MonNext ;          // LL chain: next free monRec
[8]        lwp_sema_t EntryQ ;                 // heavy system sync object used for entry
[9]        // Other fields elided …
[10]   } monRec ;
[11]
[12]   typedef struct _ExecEnv {               // EE
[13]       monRec * volatile MonFree ;         // Linked list of free monitors
[14]       // Other fields elided ...
[15]   } ExecEnv ;
[16]
[17]   void monWakeup (ExecEnv * ee, monRec * m)
[18]   {
[19]       if (m->Candidate == 1 && CAS(&m->Candidate, 1, 0) == 1) {
[20]           _lwp_sema_post (&m->EntryQ) ;
[21]       }
[22]   }
[23]
[24]   int monEnter (ExecEnv * ee, Object * obj)
[25]   {
[26]       monRec * m ;
[27]       monRec * nxt ;
[28]       intptr_t raw;
[29]
[30]     retry:
[31]       raw = obj->LockWord ;
[32]       if (!INFLATED(raw)) {           // Case 1: Empyt - fast path
[33]           m = ee->MonFree ;
[34]           if (m == NULL) m = ExtendFreeList (ee);
[35]           m ->hashCode = raw ;
[36]           nxt = m->MonNext ;
[37]           if (CAS(&obj->LockWord, raw, MKMON(m)) == raw) {
[38]               ee->MonFree = nxt ;     // successfully installing "m" confers ownership
[39]               return OK;
[40]           }
[41]           goto retry ;                // CAS failed: interference - retry
[42]       }
[43]
[44]       m = MONREC(raw) ;
[45]       if (m->Owner == ee) {           // Case 2: Inflated and locked by calling thread
[46]           m->Nest ++ ;
[47]           return OK ;
[48]       }
[49]
[50]       // Case 3: inflated and unlocked. optimization - not strictly necessary.
[51]       if (m->Owner == NULL && CAS(&m->Owner, NULL, ee) == NULL) {
[52]           if (obj->LockWord == raw)  {
[53]               m->Nest = 1 ;
[54]               return OK ;
[55]           }
[56]           m->Owner = NULL ;       // "m" is stale wrt obj. Recover as needed
[57]           MEMBAR(StoreLoad) ;
[58]           monWakeup (ee, m) ;
[59]           goto retry ;
[60]       }
```

```
[61]
[62]        // Case 4: inflated and locked by another thread.
[63]        // Slow path ... apparent contention: do this the hard way
[64]        Adjust (&m->rcThis, 1) ;         // atomic fetch-and-add
[65]        if (obj->LockWord != raw) {      // Similar to Valois' SAFEREAD
[66]            Adjust (&m->rcThis, -1) ;    // "m" is stale wrt obj
[67]            MarkSuspect (ee, m) ;
[68]            goto retry ;
[69]        }
[70]
[71]        while (obj->LockWord == raw) {
[72]            m->Candidate = 1 ;
[73]            MEMBAR(StoreLoad) ;
[74]            if (m->Owner == NULL && CAS(&m->Owner, NULL, ee) == NULL) {
[75]                Adjust (&m->rcThis, -1) ;
[76]                m->Nest = 1 ;
[77]                return OK ;
[78]            }
[79]            _lwp_sema_wait (&m->EntryQ) ;
[80]            // We're awake - recontend for the object
[81]        }
[82]        FlushAcknowledge (ee, m) ;  // we've been flushed
[83]        goto retry ;
[84]    }
[85]
[86]    int monExit (ExecEnv * ee, Object * obj)
[87]    {
[88]        monRec * m ;
[89]        intptr_t raw ;
[90]        int nn ;
[91]        raw = obj->LockWord ;
[92]        m = MONREC(raw) ;
[93]        if (INFLATED(raw) && m->Owner == ee) {
[94]            nn = m->Nest - 1 ;
[95]            if (nn == 0) {
[96]                if (m->rcThis == 0) {
[97]                    obj->LockWord = m->hashCode ;   // attempt speculative deflate
[98]                    MEMBAR(StoreLoad) ;             // publish store
[99]                    if (m->rcThis == 0) {           // resample ref count
[100]                       ReturnToFreeList (ee, m) ;  // recycle m
[101]                       return OK ;                 // fast path exit
[102]                   } else {
[103]                       FlushAndFree (ee, m) ;      // misspeculated - expel blocked threads
[104]                       return OK ;
[105]                   }
[106]               } else {
[107]                   m->Owner = NULL ;
[108]                   MEMBAR(StoreLoad) ;
[109]                   monWakeup (ee, m) ;
[110]                   return OK ;
[111]               }
[112]           }
[113]           // The following memory barrier is unrelated to the Relaxed-Lock protocol.
[114]           // The Java Memory Model promises release consistency.
[115]           MEMBAR(StoreLoad);
[116]           m->Nest = nn ;
[117]           return OK ;
[118]       }
[119]       return THROW(ILLEGAL_MONITOR_STATE) ;
[120]   }
[121]
```

Listing 1 "C" Code for monEnter() and monExit()