An Efficient Meta-lock for Implementing Ubiquitous Synchronization

Ole Agesen*, David Detlefs*, Alex Garthwaite*, Ross Knippel+, Y.S. Ramakrishna+, Derek White*

*Sun Microsystems Laboratories
One Network Drive
Burlington, MA 01803-0902

+Sun Microsystems
901 San Antonio Road
Palo Alto, CA 94303-4900

<first>.<last>@sun.com

ABSTRACT

Programs written in concurrent object-oriented languages, especially ones that employ thread-safe reusable class libraries, can execute synchronization operations (lock, notify, etc.) at an amazing rate. Unless implemented with utmost care, synchronization can become a performance bottleneck. Furthermore, in languages where every object may have its own monitor, per-object space overhead must be minimized. To address these concerns, we have developed a meta-lock to mediate access to synchronization data. The meta-lock is fast (lock + unlock executes in 11 SPARC™ architecture instructions), compact (uses only two bits of space), robust under contention (no busy-waiting), and flexible (supports a variety of higher-level synchronization operations). We have validated the meta-lock with an implementation of the synchronization operations in a high-performance product-quality Java™ virtual machine and report performance data for several large programs.

Keywords: object-oriented language implementation, synchronization, concurrent threads.

1 INTRODUCTION

Shared-memory multi-processor systems have become mainstream, even in the personal computer market. Simultaneously, the concurrent object-oriented Java™ programming language [3] has experienced explosive growth. As a result, significant effort is being devoted to implementing both the sequential and parallel features of this language, e.g., [4, 18, 19, 25, 28]. This paper focuses on the latter area, proposing a new implementation of synchronization operations that, we believe, possesses an attractive set of trade-offs between space, time, and assumptions about the underlying hardware. Implementors of the Java language's synchronization operations are challenged on two fronts:

- *Frequency.* Most Java-based programs synchronize extremely frequently. This occurs because standard class libraries, including commonly used data types such as vectors and buffers, have been designed for the general multi-threaded case. To give just one example, we measured the SPECjvm98 version of javac [29], a source-to-bytecode compiler, and found that on a high-performance virtual machine featuring exact garbage collection and other innovations (E.G.C.JVM[1]), this program executes 765,000 synchronization operations per second.
- *Ubiquity.* The Java language, unlike most concurrent languages, does not define a particular type of synchronizable object such as a monitor or a mutex. Instead, all objects, including, for example, strings and arrays, can be synchronized upon. This design offers the programmer a simpler and more regular language, but presents an obstacle to the language implementor: synchronization must be implemented at a low per-object space cost. More precisely, the *ability* to synchronize must be provided at a low space cost; *actual* synchronization can use additional space since, in practice, programs synchronize on a small fraction of objects. For example, javac, discussed above, synchronizes on about 6% of allocated objects.

Frequency demands time-efficiency while ubiquity demands space-efficiency. This paper presents a new synchronization scheme that, we believe, attains good all-around performance: synchronization executes at close to the speed of the hardware operations while reserving only two bits in each object. Our algorithm, in its basic form, uses a two-level ("meta") locking scheme, with an optional extension that fuses the two levels for higher performance in uncontended cases.

We assume the arbitrary interleaving implied by preemptive thread scheduling: no other assumption makes sense on multiprocessors, and, moreover, even on uniprocessors lack of preemption can lead to unfair (and unintuitive) thread scheduling. It may seem that synchronization operations could be elided for many programs with only a single thread. In reality, however, no program written in the Java language is single-threaded, since, in addition to the main thread created by user code, the class libraries create special threads to handle finalization and various forms of weak references [30]. More significantly, perhaps, commonly used graphics libraries, such as the Abstract Windows Toolkit (AWT), create threads.

1. E.G.C.JVM, our research JVM, known in other publications as EVM and ExactVM, is embedded in Sun's Java 2 SDK *Production* Release, available at http://www.sun.com/solaris/java/. Henceforth, the term "JVM" will refer to implementations of the Java Virtual Machine specification.

The rest of this paper is organized as follows. Section 2 informally describes the Java language's synchronization operations, at both the source and bytecode levels. Section 3 reviews previous work most closely related to our synchronization algorithm. Section 4 describes our basic two-level algorithm, and Section 5 discusses extensions, including a fast path that fuses the primary and meta-lock levels, and other optimizations that are important for good performance. Section 6 presents performance data to quantify the behavior of our algorithm. Section 7 offers final conclusions and some directions for further work.

2 BACKGROUND

Java virtual machines (JVMs) do not execute source code directly. Instead, they execute bytecode obtained from binary class-files that are produced by a source-to-bytecode compiler such as javac. Thus, JVMs must implement the bytecode-level synchronization operations, not the source-level synchronization operations. The distinction is important because the bytecode-level operations are more general than the source-level operations.

2.1 Source-level synchronization

The Java language provides mutual exclusion in two syntactic forms. A *synchronized method* of an object obtains a lock on the object, executes the method, and releases the lock. A *synchronized statement*, `synchronize (exp) { ...actions... }`, evaluates the expression to obtain an object that is locked for the duration of the specified actions.

The synchronization operations in the Java language are reentrant (recursive): synchronized statements on the same object can nest, synchronized methods can invoke other synchronized methods, and the two can be mixed. Consequently, the underlying lock and unlock operations must do some form of counting. Both synchronized methods and statements are "block structured," forcing perfect nesting of locking operations. At the source level, there is no way to express unbalanced locking operations. In particular, exception-throwing and returning out of locked regions unlock as necessary. However, as we shall see below, the story is different at the bytecode level.

To facilitate communication between threads, the Java language defines `wait`, `notify`, and `notifyAll` operations. Like locking, these operations are performed relative to an object. Prior to executing `wait` and `notify[All]`, a thread must first lock the target object. Informally, `wait` fully releases the lock and suspends the current thread. This allows other threads to obtain the lock. The waiting thread becomes eligible to run again when another thread performs a `notify` operation on the object, a specified time has elapsed, or an asynchronous interrupt is delivered to the thread. The `notify` operation wakes one waiting thread, whereas `notifyAll` wakes all waiting threads (when no threads are waiting, both operations are no-ops). When a waiting thread wakes up, it reacquires the lock the same number of times it held it prior to `wait`. The lock reacquisition puts the thread into competition with other threads attempting to acquire the lock, including both other awakened waiters and threads attempting to execute a synchronized method or statement. Once a waiting thread has reacquired the lock, the `wait` operation completes. To simplify matters, in this paper we shall not discuss interrupts further, except to note that in most of our implementation, an interrupt is handled similarly to a time-out.

2.2 Bytecode-level synchronization

Having described synchronization at the source level, we now turn to the bytecode level. In bytecode, method synchronization is performed as part of the call/return sequence: if the `acc_synchronized` attribute is set on a method, the call sequence must acquire the lock (one more time) and the return sequence must release it once. Statement synchronization is expressed using a bytecode pair, `monitorenter` and `monitorexit`, which lock and unlock, respectively, the object referenced by a value popped from the JVM's "operand stack." Unfortunately, while the bytecode representation of synchronized methods is inherently well-nested, there is no such guarantee for `monitorenter` and `monitorexit`. Nothing prevents bytecode from containing instruction sequences like "lock *A*, lock *B*, unlock *A*, unlock *B*," which has no equivalent Java source code. The loss of block structure at the bytecode level makes it difficult to stack allocate locking-related data structures. Consequently, to handle non-LIFO locking and unlocking, our implementation uses a free-list allocator (see Section 5.1).

Conceivably, a static analysis of bytecode could conservatively "pair up" `monitorenter` and `monitorexit` instructions in most cases, allowing subsequent execution to assume perfect nesting. We would expect this analysis to succeed most of the time on bytecode resulting from translation of Java source code. In general, however, the problem is undecidable, and it would be incorrect to reject bytecode for which the analysis fails, because such bytecode is still legal in the sense that it passes the bytecode verifier [22]. To further complicate the picture, JNI, the Java Native Interface, grants native code access to synchronization in the form of lock and unlock operations, so even if all bytecode can be shown to be structured, a fall-back mechanism would still be needed for synchronization by native code.

Finally, the `wait` and `notify[All]` operations have no direct representation at the bytecode level, but instead are implemented as native methods of the top-most class (`java.lang.Object`).

3 RELATED WORK

There is a large general literature on synchronization primitives and their implementation. Dijkstra [10] and Lamport [20] present subtle algorithms that achieve mutual exclusion assuming only that individual reads and writes are atomic. Fortunately, modern architectures provide composite instructions such as compare-and-swap that read and write a memory location in a single atomic step, greatly simplifying the mutual exclusion problem and eliminating the need for such subtlety. We shall refer to the composite atomic instructions simply as "atomic instructions." Weakly consistent memory models, which allow different processors to observe memory operations as occurring in different orders, may require the use of memory barrier instructions to reintroduce consistency, whether using individual reads and writes or atomic instructions.

In its broad structure, our meta-lock resembles the MCS-lock of Mellor-Crummey and Scott [24]. The MCS-lock uses an atomic swap for lock acquisition and an atomic compare-and-swap (CAS) for lock release in much the same way as does our meta-lock algorithm. The MCS-lock has many of the same properties as our meta-lock, including starvation freedom and FIFO access to the lock. However, the details are quite different and, in particular, contention results in busy-waiting. Also, space-efficiency is not as extreme a concern in the context of their work. Magnusson *et al.* [23] survey locking schemes related to MCS-locks, and describe two new locking schemes that offer different performance tradeoffs.

Brinch Hansen [12] and Hoare [14] coined the term *monitor*, and provided the nomenclature used by the Java language. There are several ways, however, in which the monitors in the Java language differ from the original: any object may be used as a monitor, monitors may be entered recursively, and monitors provide a single implicit, rather than possibly several explicit, "condition variables" [13]. Birrell gives an excellent tutorial on programming with "standard" synchronization primitives [6]. As we have discussed, the synchronization primitives of the Java platform are difficult to implement in a manner that is both time- and space-efficient. Scalability to multiprocessor systems is another important concern. We shall now discuss some previous implementations focusing on the approach they take to trading off these concerns.

The original JDK™ implementation of the Java virtual machine [22] provides a space-efficient monitor implementation, but one that is not particularly time-efficient or scalable. This design requires that each object has a unique identifier valid over its lifetime. The actual implementation uses an *object table* whose entries are called *handles*, providing an extra level of indirection to facilitate object compaction. The handle address of an object remains constant during the object's lifetime, and can therefore serve as a unique identifier. A global table called the *monitor cache* maps object handle addresses to monitor structures that can be used to perform the actual synchronization operations. When a thread synchronizes on an object, it first ensures that the monitor cache maps the object to a monitor, creating and installing the monitor in the table if necessary. This approach has no fixed per-object space cost, using only space proportional to the number of entries in the monitor cache. However, it is fairly time-inefficient, since each synchronization operation must first do (at least) a table lookup to locate the monitor associated with the object. Further, it is not very scalable. The monitor cache is a global data structure that is accessed concurrently. To make this concurrent access safe, the monitor cache is protected by a lock. Thus, all synchronization operations obtain a single lock, an obvious source of contention. The monitor cache locking also adds some cost in the uncontended case.

Bacon *et al.* [4] propose a clever scheme motivated by many of the same concerns as ours. In this design, 24 bits of each object header are devoted to locking. One bit indicates whether the object has a *thin* or *fat* lock. If it has a thin lock, then all necessary locking information is contained in the remaining 23 bits. If it has a fat lock, then the remaining 23 bits are an index into an array of pointers to fat lock structures that, much like monitors, hold the necessary data for the synchronization operations. Thin locks are used as long as the lock is uncontended, and is not recursively locked more times than can be represented in a count field of the thin lock; if either condition is violated, the lock representation is "inflated." This design does well at optimizing the expected common case of uncontended locking and unlocking. Locking requires a small number of instructions, with only one atomic instruction in the fast path, and unlocking requires no atomic instruction. However, thin locks leave some issues to be addressed:

- *Contention.* When there is contention on a lock for the first time, all threads that do not acquire the lock must *busy-wait* (a.k.a. spin) until the lock is released. Unbounded busy-waiting is generally undesirable but, as the authors point out, busy-waiting is done at most once in the lifetime of a given object. Still, it would not be hard to construct a program that does contended locking on many short-lived objects, causing a great deal of busy-waiting.
- *Lack of deflation.* Lock *deflation* is not discussed; once a lock becomes fat, it remains fat. While this is not an issue for most programs, one could imagine a long-lived program in which many long-lived objects are locked with contention at some point in their lifetimes. Absent some form of deflation, such a program would consume a large amount of memory for fat locks.
- *Space consumption.* Thin locks use 24 bits. This is a significant overhead since the average object size for most programs is quite small [9].

Joy and Steele [16] describe a synchronization scheme that uses atomic instructions to arbitrate locking. A per-thread cache of the last locked object allows recursive locking to be optimized. This scheme does not address notification or waiting operations, and uses busy-waiting to resolve contention. Joy [15] and Joy and Van Hoff [17] describe schemes in which locked objects are enlarged to contain monitor structures, and are given altered virtual function tables whose entries for the "lock" operation assume that the object already contains monitor information.

In other related work, monitor implementations have been proposed that exploit cooperative thread scheduling [19] or make special provision for faster execution of single-threaded programs [25, 27]. As we have already noted, we assume preemptive scheduling, and desire algorithms that work well for both single- and multithreaded programs, so we shall not discuss these restricted schemes further.

In [5], Bak describes how an early version of Sun's Java HotSpot™ JVM locks objects by replacing an object header word with a pointer to an external lock structure, displacing the original contents of the header word into the lock structure. Two low-order bits in the header word encode its format. Java HotSpot stack allocates lock structures for efficiency. Furthermore, this stack allocation allows fast recursive locking by enabling efficient verification that an object is locked by the current thread: if the lock structure address is sufficiently close to the current stack pointer to guarantee membership in the same thread stack, the current thread must be the lock owner. Personal communication from Zhang [31] describes how a more recent version of Java HotSpot uses atomic instructions to arbitrate locking between threads, and to inflate contended locks to full monitors without busy-waiting.

We borrow Java HotSpot's idea of displacing a header word and using some bits of the word to encode its format and the lock state. We use a different allocation scheme that supports non-block-structured synchronization.

4 OUR ALGORITHM

In any concurrent environment, there must be some protocol observed by threads as they manipulate the *synchronization data* of objects, that is, the data structures that manage synchronization operations. Typically, the protocol specifies when a thread may access or manipulate an object's synchronization data. For example, the thin-locks approach relies on an invariant whereby only the thread owning the lock on an object may modify that object's synchronization data. Other approaches, like ours, allow any thread to update this information. The key to our approach is a time- and

```
typedef struct execenv {
    Thread        thread;               /* ExecEnv is a subtype of Thread.     */
    mutex_t       metaLockMutex;        /* Used by slow-path meta-lock/unlock. */
    condvar_t     metaLockCondvar;      /* To wait for meta-lock hand-off.     */
    bool_t        gotMetaLockSlow;      /* Wait for predecessor to give bits.  */
    bool_t        bitsForGrab;          /* Wait for successor to grab bits.    */
    BitField      metaLockBits;         /* Space to get/give releaseBits.      */
    ExecEnv      *succEE;               /* Next thread to get the meta-lock.   */
    mutex_t       monitorLockMutex;     /* Used by slow-path lock/unlock.      */
    condvar_t     monitorLockCondvar;   /* To wait for monitor acquisition.    */
    bool_t        isWaitingForNotify;   /* Am waiting for notification.        */
    ... other fields ...
} ExecEnv;
```

Fig. 1. Per-thread (execution environment) fields used for synchronization.

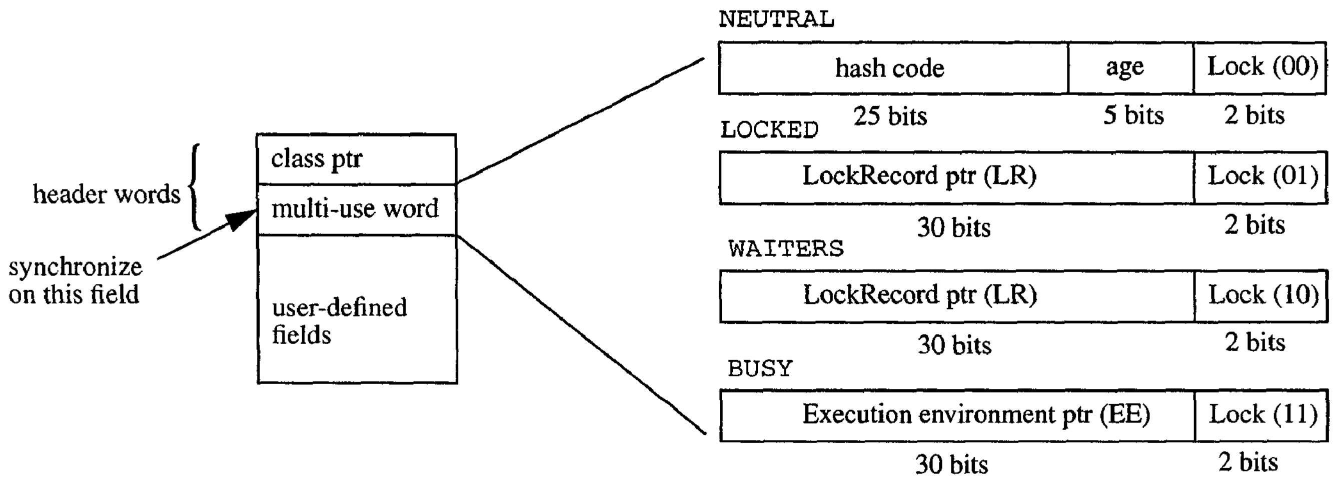

Fig. 2. Object layout and possible states for an object's multi-use word.

space-efficient meta-lock that protects the synchronization data of each object. The typical pattern for synchronization operations in our system is:

1. Obtain the object's meta-lock to ensure exclusive access to the object's synchronization data.
2. Manipulate the synchronization data of that object. This operation should be fast.
3. Release or hand off the object's meta-lock.

Meta-locks play a similar role as the auxiliary spin-locks seen in many implementations of POSIX threads [7, 21]. These spin-locks provide brief exclusive access to the synchronization state maintained in records representing programmer-level mutexes and condition variables.

We shall use the term "monitor-lock" to denote the lock abstraction exported by the Java virtual machine to avoid confusion with the meta-lock used in its implementation. Because meta-lock acquisition occurs in FIFO order, the above pattern allows a number of fairness policies at the monitor level. The rest of this section describes our algorithm in detail: Section 4.1 presents the data structures involved, Section 4.2 the meta-lock algorithm, Section 4.3 the implementation of the monitor-level lock and unlock operations, and Section 4.4 the implementation of wait and notify.

4.1 Data structures

Synchronization involves three entities: threads, objects, and lock records. We describe the relevant parts of the data structures that implement them and how they interact during synchronization.

Threads. We call the data structure that holds thread-specific state an *execution environment* (EE). Since EEs and threads correspond one to one, EE addresses are well-suited as unique thread identifiers. Figure 1 shows the fields in EEs that the synchronization code uses. The steady-state (and initial) value is FALSE for the boolean fields and NULL for the pointer fields. The (POSIX-style) mutexes and condition variables in the EEs are used to avoid busy-waiting when contention requires threads to wait for their turn to lock an object, while other fields serve to exchange information between threads synchronizing on the same object.

Objects. Figure 2 shows the object layout in our JVM. Because every object may potentially be used for synchronization, it is critical to minimize the per-object space overhead. Objects have two-word headers. The first word points to the object's class. Only the second word is used for synchronization, but it serves other purposes as well, such as holding a hash code[2] and garbage-collector

2. E.G.C.JVM uses a handle-less copying memory system, so object or handle addresses cannot be used as hash codes.

```
typedef struct LockRecord_s LockRecord;
struct LockRecord_s {
  ExecEnv *owner;        /* Owner thread.         */
  int      lockCount;    /* # recursive locks.    */
  BitField storedBits;   /* Hash and age.         */
  LockRecord *queue;     /* Lock queue on obj.    */
  LockRecord *nextFree;  /* Free-list.            */
};
```

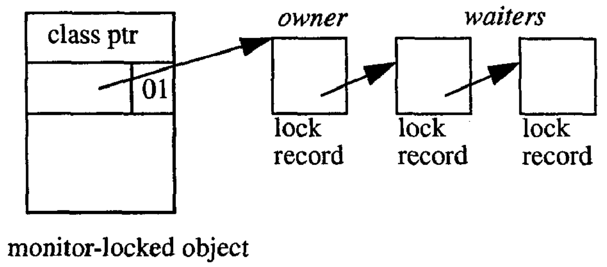

Fig. 3. A lock record and how they are chained out of the multi-use word of an object.

age information, so we call it the *multi-use* word. We employ a *header word displacement* technique invented by our colleague Lars Bak [5]. The two least significant bits in the multi-use word, called the *lock bits*, hold the *lock state*, which serve as a format indicator and simultaneously encode meta-lock information for the object. Figure 2 also shows the four possible lock states and their formats. Objects are created in the NEUTRAL state (in fact, the majority of objects never leave this state), remain in this state as long as no thread synchronizes on them, and return to NEUTRAL once synchronization ceases. A monitor-locked object is in the LOCKED state. The high 30 bits of the multi-use word hold a pointer to synchronization data (a *lock record*, see below) that indicates which thread owns the monitor-lock and also stores the displaced hash and age information. The WAITERS state is entered when a thread releases the monitor-lock while other threads are waiting to acquire the lock or to be notified: the object is no longer monitor-locked, but the remainder of the multi-use word still points to a lock record. The fourth and final state, BUSY, indicates that the object is meta-locked. In this case, the high part of the multi-use word contains the EE of the thread that has the meta-lock *or* the EE of a thread attempting to acquire the meta-lock, as will be explained in Section 4.2

Lock records. Most synchronization data is kept not in objects but in LockRecords. A lock record represents a thread for the purpose of synchronization on a particular object. Figure 3 shows the fields of a lock record: the owner thread, the number of times the thread has locked the object (recall that monitor-locks are recursive), a field for the displaced hash and age information, a queue field for linking the lock records of all threads that synchronize on a given object, and a free-list field for linking lock records when they are not in use (see Section 5.1). Figure 3 also shows an object with three lock records on its lock queue. In this example, the state is LOCKED, so one of the lock records belongs to a thread that holds the monitor-lock. In our implementation, new lock records are appended to the end of the queue (FIFO order) and stay in order, except that when a thread acquires the monitor-lock, it moves its lock record to the front so that the first lock record of a locked object always belongs to the thread that holds the monitor-lock.

4.2 Meta-locking: exclusive access to synchronization data

An object's meta-lock protects its synchronization data, which we can now define precisely as comprising the multi-use word, including the lock queue pointer (when there is one) and the lock records in that queue. For example, if a thread wants to place a lock record in the queue to wait for its turn to acquire the monitor-lock, it meta-locks the object to gain exclusive access to the queue, appends its lock record, and then releases the meta-lock. Similarly, to read or write the hash code of an object, meta-locking must be done (though it is possible to optimize reads of immutable fields, like the hash code, most of the time).

The meta-lock mechanism provides two routines for higher-level locking to use: getMetaLock() acquires the meta-lock and returns the previous (non-BUSY) value of the multi-use word, while releaseMetaLock() releases the meta-lock and sets the multi-use word of the object to a value appropriate to the new state of the synchronization data. This new value may be any non-BUSY value; the operation might release the lock and restore the displaced multi-use word bits and the NEUTRAL lock state, or it might change the queue pointer to point to a new lock record, or it might leave the multi-use word unchanged. In any case, call this new multi-use word value the *release bits* of the operation.

Figure 5 shows the flow chart for obtaining and releasing an object's meta-lock. There is a slow path for each operation when contention occurs. For reference, the source code for these operations can be found in the appendix.

A thread attempts to gain the meta-lock by using an atomic swap operation to replace the object's multi-use word with a word consisting of a reference to the thread's EE and the low-order bits representing the BUSY state. If the word returned by the swap operation has low-order bits in any state other than BUSY, the thread has acquired the meta-lock and may proceed. However, if the returned word's low-order bits indicate the object is BUSY, then some other thread holds the meta-lock, so the current thread follows the slow path for meta-lock acquisition. As Figure 4 shows, the threads contending for the meta-lock are totally ordered by the order in which the swap instructions occurred. The first thread in this order knows, since it acquired the meta-lock, that it has no pre-

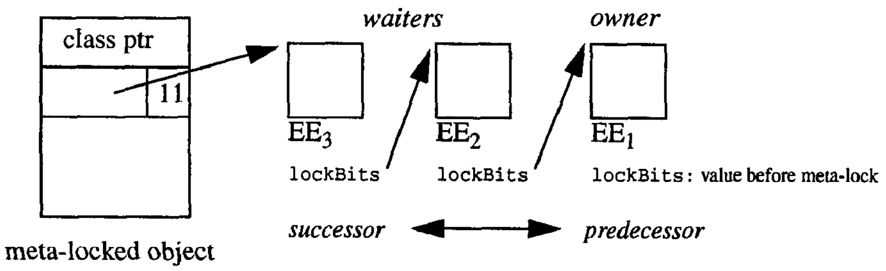

Fig. 4. Threads contending for a meta-lock.

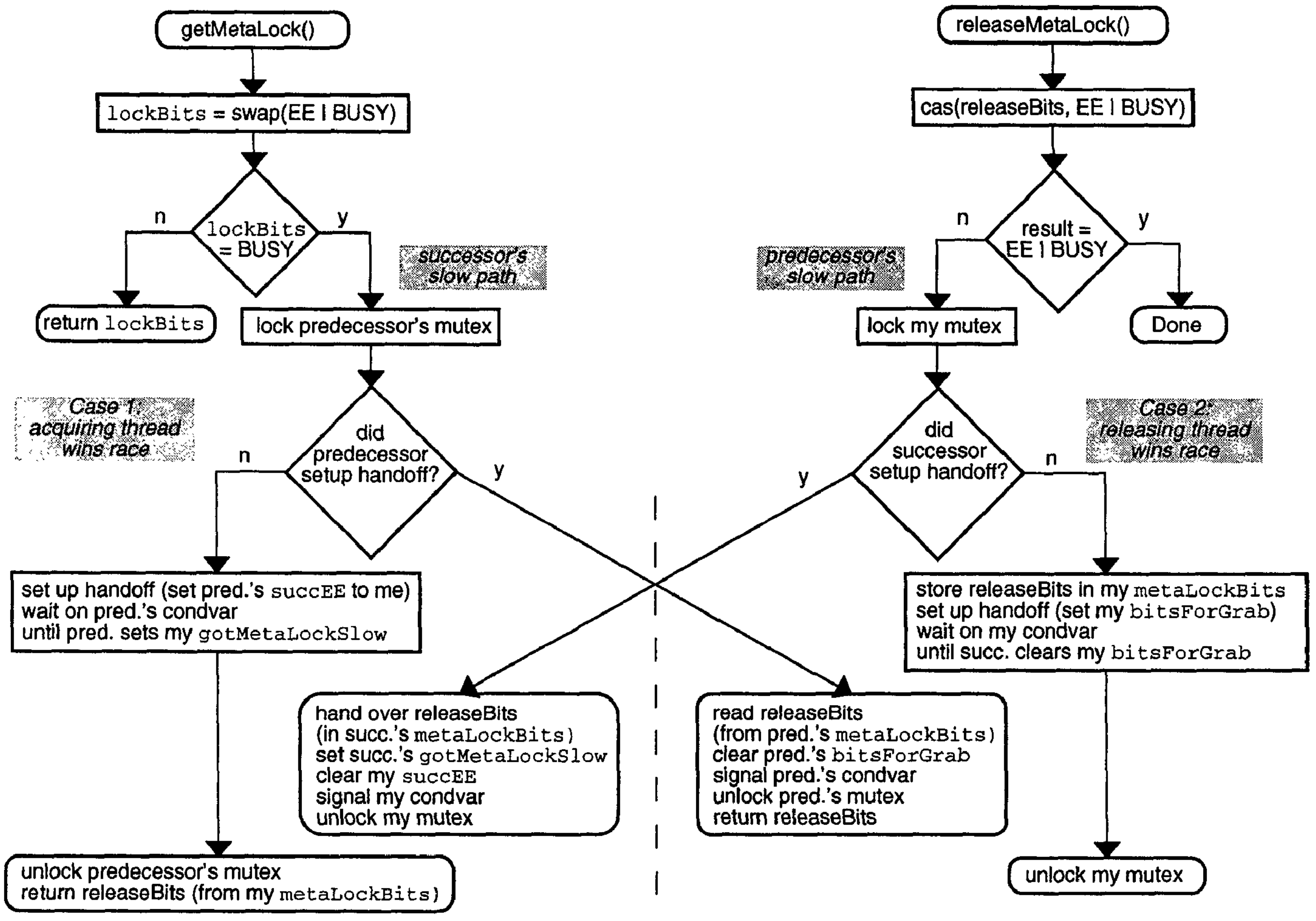
getMetaLock()
lockBits = swap(EE | BUSY)
lockBits = BUSY
n
y
return lockBits
lock predecessor's mutex
successor's slow path
Case 1: acquiring thread wins race
did predecessor setup handoff?
set up handoff (set pred.'s succEE to me)
wait on pred.'s condvar
until pred. sets my gotMetaLockSlow
unlock predecessor's mutex
return releaseBits (from my metaLockBits)
hand over releaseBits
(in succ.'s metaLockBits)
set succ.'s gotMetaLockSlow
clear my succEE
signal my condvar
unlock my mutex
releaseMetaLock()
cas(releaseBits, EE | BUSY)
result = EE | BUSY
predecessor's slow path
lock my mutex
Done
did successor setup handoff?
Case 2: releasing thread wins race
store releaseBits in my metaLockBits
set up handoff (set my bitsForGrab)
wait on my condvar
until succ. clears my bitsForGrab
unlock my mutex
read releaseBits
(from pred.'s metaLockBits)
clear pred.'s bitsForGrab
signal pred.'s condvar
unlock pred.'s mutex
return releaseBits

Fig. 5. Flow chart for acquiring and releasing a meta-lock.

decessor, and every other thread knows its predecessor from the EE in the lockBits word read by the swap instruction.

To accomplish the meta-lock release, a thread uses an atomic compare-and-swap (CAS) operation to atomically compare the current contents of the object's multi-use word with what it had written there (i.e., its EE and the BUSY state) and, if it is still the same, write the release bits. If the comparison fails, then some other thread has attempted to obtain the meta-lock and is now waiting for its turn. In this case, the releasing thread will "hand off" the meta-lock to the next thread in the order induced by the swap operations, by taking the slow path for meta-lock release. The releasing thread must also hand off the release bits to the acquiring thread, so the acquiring thread will obtain the bits that would have been written had there been no contention for the meta-lock.

The main complication in the slow-path meta-lock hand-off is that each thread in the atomic swap total order knows the identity of its predecessor, but not of its successor. Because of this asymmetry, the hand-off from a predecessor to its successor synchronizes using state in the predecessor's EE. As we saw in Figure 1, that state includes a mutex and condition variable pair metaLockMutex/metaLockCondvar, a field to record the successor's EE, and several booleans used to coordinate the transfer of the value of the release bits. The mutex is used to ensure that the threads participating in the hand-off update the other fields in the correct order.

The condition variable is used to block whichever thread enters the hand-off first until the other thread is ready to complete the transaction.

The hand-off protocol proceeds in one of two ways, as shown in the two cases highlighted in Figure 5. The predecessor thread releasing the meta-lock and the successor thread attempting to acquire it "race" to acquire the predecessor thread's mutex. If the successor thread wins the race, it will change the predecessor's `succEE` field from the default value of `NULL` to the address of its own `EE`. If the predecessor thread wins the race, it will set it's own `bitsForGrab` field to `TRUE`. Thus, each thread may determine whether it won the race by noting whether the competitor has made the corresponding change.

Case 1: successor (acquiring) thread wins race. When the successor thread obtains the mutex and its predecessor's `bitsForGrab` field is `FALSE`, it knows it acquired the mutex first. In this event, it updates the predecessors's `succEE`, and waits for the predecessor to complete the transaction. When the predecessor acquires the mutex, it notes from the non-`NULL` value in its `succEE` field that the successor went first, and therefore completes the meta-lock hand-off by setting the successor's `metaLockBits` to the release bits, setting `gotMetaLockSlow` to indicate that those bits are valid, and waking the successor by signalling `metaLockCondvar`. Finally, the predecessor releases the mutex, allowing the successor to continue, having acquired the meta-lock.

Case 2: predecessor (releasing) thread wins race. Here the predecessor thread determines that it acquired the mutex first by noting that its `succEE` field is still `NULL`. It does not know the identity of its successor, but it knows that the successor knows its identity. Thus, it sets the `metaLockBits` field of its `EE` to the proper release bits value, and sets the `bitsForGrab` field to `TRUE` to indicate that those bits are valid, and waits for the successor to read the bits (releasing the mutex in the process). The successor thread obtains the mutex, sees that its predecessor's `bitsForGrab` is `TRUE`, and thus realizes that it has acquired the mutex second, and that the release bits are available in its predecessor's `metaLockBits` field. It copies those bits, resets the predecessor's `bitsForGrab` to the default value of `FALSE` to indicate that the hand-off is complete, signals its predecessor's condition variable to inform it of that fact, and, finally, releases the mutex.

The meta-lock protocol guarantees that threads obtain the meta-lock in the order determined by the execution of the atomic swap operations. A thread need only wait for threads ahead of it in the swap order, so if no thread blocks indefinitely while holding the meta-lock, all threads attempting to acquire the meta-lock will eventually succeed. An informal proof that the meta-lock guarantees mutual exclusion and freedom from lockout can be found in our Sun Labs Technical Report [1].

The hand-off protocol is somewhat complicated, but it avoids pitfalls that occur if simple busy-waiting is used. If n acquiring threads busy-wait for the meta-lock when there is contention, then even with fair thread scheduling there can be a delay for the next thread to acquire the lock proportional to n thread time slices. If thread scheduling isn't fair at all, then the delay may be unbounded. On the other hand, bounded busy-waiting (using the hand-off protocol if still locked) can be useful. A scheme where an acquiring thread yields when it detects meta-lock contention is much simpler, but works only if the operating system's yield call guarantees that all other threads will run before the acquiring thread is run again. Otherwise the thread holding the meta-lock might starve.

Armed with our meta-lock, we now proceed to implement the monitor operations: lock, unlock, wait, and notify. Because the meta-lock arbitrates access among contending threads, we can implement monitor operations using a number of different data structures and offer a variety of semantics [8]. We have chosen an implementation that uses a simple linked list of lock records and gives equal preference to awakened waiters and newly arrived threads.

4.3 Locking and unlocking objects

Acquiring and releasing the monitor-lock of an object corresponds to entering and exiting the object's monitor. Figure 6 shows the fast-path implementation for these operations. Most commonly, the object being monitor-locked is either unlocked—in a `NEUTRAL` or `WAITERS` state—or being locked recursively. In these cases, the locking thread simply updates the object's synchronization data; it need not interact with other threads. Likewise, when unlocking an object, there are two cases that involve no interaction with other threads: the unlocking thread has recursively locked the object, in which case it simply decrements the lock count; or there are no other threads attempting to acquire a singly-held lock, in which case it restores the displaced multi-use word value, which has the `NEUTRAL` lock state.

The remaining cases involve threads contending for the monitor-lock; see Figure 7. Much as in meta-lock hand-off, we use a per-thread mutex and condition variable to coordinate acquiring and releasing threads. When a thread attempts to acquire a monitor-lock but finds it locked, it suspends on a condition variable in its own `EE`, waiting to be signalled by a lock-releasing thread that it should re-attempt the acquisition. When the acquiring thread receives this signal, it repeats the lock-acquisition slow path: it acquires the object's meta-lock and checks the object's lock state. If the state is now different from `LOCKED`, it adjusts the synchronization data to indicate that it holds the monitor-lock and releases the meta-lock; if the state is `LOCKED`, the thread releases the meta-lock and waits again.

To release a contended monitor-lock, a thread first obtains the meta-lock. Then it removes its own lock record from the queue. Subsequently, it calls `wakeupEE()` to find the first thread on the lock queue that is waiting to acquire the lock. If there is such a thread, it is signalled. Then, the releasing thread performs a meta-lock release to write out the shortened lock queue and set the lock state to `WAITERS`. (We have elided code that optimizes away redundant signalling on waiting threads.) Thus, at the monitor-lock level, unlike the meta-lock level, we do not use a hand-off: the releasing thread does not give the monitor-lock to a waiting thread but merely invites the waiting thread to re-attempt the acquisition.

4.4 Waiting and notifying

Figure 8 shows the remaining two monitor operations: wait and notify. The Java language specification requires that the thread performing them must hold the object's monitor-lock, otherwise the operations throw an exception. A thread waits by acquiring the meta-lock, setting the `isWaitingForNotify` field in its EE, and releasing the monitor-lock and meta-lock (i.e., setting the lock state to `WAITERS`). It then waits until a notification operation makes it a potential lock contender again, and some monitor-lock release operation signals it to actively contend, or until some amount of time specified in the wait operation has elapsed. A notifying thread similarly acquires the meta-lock. Then it walks the

```
bool_t monitorEnter(ExecEnv *ee, Object *obj) {
  BitField r     = getMetaLock(ee, obj);
  LockState state = lockState(r);
  if (state == NEUTRAL) {
    /* Establish locking by this thread. */
    LockRecord *lr = allocLockRecord(ee);
    lr->storedBits = r;
    releaseMetaLock(ee, obj, lr | LOCKED);
  } else if (state == LOCKED) {
    LockRecord *ownerLR = lockRecord(r);
    if (ownerLR->owner == ee) {
      /* Recursive locking. */
      ownerLR->lockCount++;
      releaseMetaLock(ee, obj, r);
    } else {
      LockRecord *lr = allocLockRecord(ee);
      ownerLR->queue = appendToQueue(ownerLR->queue,
                                     lr);
      monitorEnterSlow(ee, obj, r);
    }
  } else if (state == WAITERS) {
    /* obj is unlocked but has threads waiting
       for notification. */
    LockRecord *lr = allocLockRecord(ee);
    LockRecord *firstWaiterLR = lockRecord(r);
    lr->queue      = firstWaiterLR;
    lr->storedBits = firstWaiterLR->storedBits;
    releaseMetaLock(ee, obj, lr | LOCKED);
  }
  return TRUE;
}
```

```
bool_t monitorExit(ExecEnv *ee, Object *obj) {
  BitField    r       = getMetaLock(ee, obj);
  LockRecord *ownerLR = lockRecord(r);
  LockState   state   = lockState(r);
  if (state == LOCKED && ownerLR->owner == ee) {
    assert(ownerLR->lockCount >= 1);
    if (ownerLR->lockCount == 1) {
      /* Last release: will not have lock
         after this operation. */
      if (ownerLR->queue == NULL) {
        /* No-one waiting. */
        assert(lockState(ownerLR->storedBits)
               == NEUTRAL);
        releaseMetaLock(ee, obj,
                        ownerLR->storedBits);
      } else {
        /* There is a queue. Release
           with wakeup call. */
        ownerLR->queue->storedBits =
                            ownerLR->storedBits;
        monitorExitSlow(ee, obj, ownerLR->queue);
        ownerLR->queue = NULL;
      }
      recycleLockRecord(ee, ownerLR);
    } else {
      /* Still has lock after this. */
      ownerLR->lockCount--;
      releaseMetaLock(ee, obj, r);
    }
  } else {
    releaseMetaLock(ee, obj, r);
    throwIllegalMonitorStateException();
    return FALSE;
  }
  return TRUE;
}
```

Fig. 6. Fast paths for monitor-lock operations.

```
void monitorEnterSlow(ExecEnv *ee, Object *obj,
                      BitField r) {
  LockRecord *lr;
  while (lockState(r) == LOCKED) {
    mutexLock(&ee->monitorLockMutex);
    releaseMetaLock(ee, obj, r);
    condvarWait(&ee->monitorLockCondvar,
                &ee->monitorLockMutex);
    mutexUnlock(&ee->monitorLockMutex);
    r = getMetaLock(ee, obj);
  }
  assert(lockState(r) == WAITERS);
  lr = moveMyLRToFront(ee, lockRecord(r));
  releaseMetaLock(ee, obj, lr | LOCKED);
}
```

```
void monitorExitSlow(ExecEnv *ee, Object *obj,
                     LockRecord *lr) {
  ExecEnv *wakeEE = wakeupEE(lr);
  if (wakeEE) {
    mutexLock(&wakeEE->monitorLockMutex);
    releaseMetaLock(ee, obj, lr | WAITERS);
    condvarSignal(&wakeEE->monitorLockCondvar);
    mutexUnlock(&wakeEE->monitorLockMutex);
  } else {
    releaseMetaLock(ee, obj, lr | WAITERS);
  }
}
```

Fig. 7. Slow paths for monitor-lock operations.

queue of lock records, looking for threads waiting for notification—ones whose `isWaitingForNotify` field is TRUE—and resetting this boolean to indicate that they have been notified. The `notify()` operation finds the first such thread and resets its boolean; `notifyAll()` traverses the entire lock queue. Finally, the notifying thread releases the meta-lock.

Since some styles of concurrent programming result in a high frequency of notifications, our implementation has further optimized the notify code (the optimization is not shown in the figure). The idea is that a simple read of the multi-use word most of the time suffices to grab the root of the lock queue. If the read fetches a word in LOCKED state, the notifying thread can verify that it holds the monitor-lock and walk the queue without holding the meta-lock. The correctness of this optimization relies on two properties: a new thread waiting for a notify cannot appear in the queue (because the notifying thread holds the monitor-lock), and other threads that join the queue do so at the end (so the queue is never disconnected). See also Section 5.3 for an alternative implementation in which notify does no queue walking.

5 EXTENSIONS TO THE BASIC ALGORITHM

In this section, we discuss extensions to our algorithm, related to management of lock records and optimization of cases where we may safely avoid meta-locking because the change in the object's lock state requires only one word to be updated. We also demonstrate the flexibility of our approach and outline how to implement it on hardware that does not provide atomic CAS or SWAP operations.

```
void monitorWait(ExecEnv *ee, Object *obj,
                 java_long millis) {
  BitField    r       = getMetaLock(ee, obj);
  LockRecord *ownerLR = lockRecord(r);
  LockState   state   = lockState(r);
  if (state == LOCKED && ownerLR->owner == ee) {
    mutexLock(&ee->monitorLockMutex);
    ee->isWaitingForNotify = TRUE;
    monitorExitSlow(ee, obj, ownerLR);
    if (millis == TIMEOUT_INFINITY)
      condvarWait(&ee->monitorLockCondvar,
                  &ee->monitorLockMutex);
    else
      condvarTimedWait(&ee->monitorLockCondvar,
                  &ee->monitorLockMutex, millis);
    ee->isWaitingForNotify = FALSE;
    mutexUnlock(&ee->monitorLockMutex);
    r = getMetaLock(ee, obj);
    monitorEnterSlow(ee, obj, r);
  } else {
    releaseMetaLock(ee, obj, r);
    throwIllegalMonitorStateException();
  }
}
```

```
void notifyOneOrAll(ExecEnv *ee, Object *obj,
                    bool_t one) {
  BitField    r       = getMetaLock(ee, obj);
  LockRecord *ownerLR = lockRecord(r);
  LockState   state   = lockState(r);
  if (state == LOCKED && ownerLR->owner == ee) {
    LockRecord *q = ownerLR->queue;
    while (q) {
      if (q->owner->isWaitingForNotify) {
        q->owner->isWaitingForNotify = FALSE;
        if (one) break;
      }
      q = q->queue;
    }
    releaseMetaLock(ee, obj, r);
  } else {
    releaseMetaLock(ee, obj, r);
    throwIllegalMonitorStateException();
  }
}

void monitorNotify(ExecEnv *ee, Object *obj) {
  notifyOneOrAll(ee, obj, TRUE);
}

void monitorNotifyAll(ExecEnv *ee, Object *obj) {
  notifyOneOrAll(ee, obj, FALSE);
}
```

Fig. 8. Wait and notify code.

5.1 Lock record allocation

As we discussed in Section 3, each of the locking schemes we know about, including the present one, at least occasionally allocates data structures related to locking. This section discusses how those data structures are allocated and deallocated. The original JDK implementation allocates monitors globally, causing serialization of monitor cache operations and resulting scalability bottlenecks. Periodically, unused monitors are reclaimed. The thin locks scheme globally allocates "fat locks," which remain allocated for the lifetime of the associated object [4].

In our scheme, lock records are the unit of allocation. Each thread has a set of lock records for its exclusive use, linked together in a free list. Lock records on a thread's free list have as many fields as possible preinitialized: the owner field points to the owning thread, the count fields contains 1, which is the proper count when locks are first acquired, and the queue field contains NULL because uncontended locking is most frequent (separation of the free-list link and the queue link, see Figure 3, allows the queue field to be preset to NULL). Lock record allocation is optimized to avoid any test for an empty free list; instead, an attempt to dereference a NULL pointer generates a signal. The signal handler recognizes the situation, refills the thread's lock record free list, and retries the operation. Threads start with 8 free lock records and add an exponentially increasing number each time they exhaust the free list.

When a thread unlocks an object, the lock record used by the thread to accomplish the locking is returned to the thread's free list. In our current implementation, the set of lock records allocated to a given thread only grows; there is no provision for removing lock records from a thread's free list if the thread briefly locks many objects, but usually locks few. This is not so bad; the "high water mark" of allocated lock records is limited by the product of the thread stack size and the maximum lexical nesting depth of synchronized statements (at least for bytecode created by compiling Java language source code, as discussed in Section 2.2). If we wished to add a mechanism to return lock records on free lists to the global memory pool, it would be a simple matter to do so as part of garbage collection, as long as we can guarantee that no thread is accessing the lock record free list during garbage collection. Our system has a general mechanism for restricting when garbage collection occurs that can be used to provide this guarantee.

5.2 Extra fast locking and unlocking of uncontended objects

We can optimize the algorithm further in the case of uncontended objects. This optimization fuses the meta-lock and monitor-lock operations into a single step. With this optimization, a thread attempting to lock an object reads the object's multi-use word. If the object's lock state is NEUTRAL, then an "extra fast" path is tried. The thread copies the hash and age bits into a fresh lock record and builds a new multi-use value containing the lock record address and the LOCKED state. A CAS instruction is then used to atomically change the multi-use word to the new value if it has not changed since it was read. If the CAS succeeds, then the object is locked; otherwise, the normal meta-locking protocol is used. With this optimization, the extra fast path for locking uses one atomic instruction rather than the two needed for meta-locking and meta-unlocking and the total number of instructions is smaller (15 SPARC instructions).

A similar extra fast path for unlocking is slightly more complicated. When the extra fast locking path succeeds, the only lock record in its queue is that of the locking thread; the queue field of that lock record is NULL. Another thread may add a lock record to the queue, changing this queue field at any time. So the extra fast unlocking path must atomically change the multi-use word of the object back to its original contents, but only if the queue field of the first lock record remains NULL. Unfortunately, this "double-compare-and-swap" operation is not supported in most architectures (though it is not completely unheard of; see [11]). To get around this, we add a new constraint to the slow path. We require that lock records be allocated with eight-byte alignment, so that three bits are zero in the address of a lock record. In the LOCKED state, this extra bit is used to summarize the state of the queue field of the first lock record: we maintain the invariant that when the bit is 0, the queue field is NULL. If the bit is 1, the queue may be non-NULL. Thus, the first thread to enqueue a lock record after the initial one is required to set this bit when releasing the meta-lock. Once this invariant may be assumed, we can construct an extra fast unlock path: check the locking depth, decrementing it and returning if it is greater than one. Otherwise, construct the expected current value of the multi-use word (pointer to same lock record, queue field bit still clear, LOCKED state), and the desired new value (original multi-use bits, NEUTRAL state), and perform a CAS instruction to write the new value if the current value is still the expected value. If no other thread has enqueued a lock record, then the CAS succeeds and the object is unlocked; otherwise, we revert to the normal meta-locking protocol. If recursive locking were found to be very rare, this proposal could be extended to also summarize the lock count in the extra bit, so that a zero bit observed by an unlocking thread implied both a NULL queue field and a lock count of 1, eliminating the explicit test for recursion.

The instruction count of the extra fast unlocking sequence is similar to that of extra fast locking, and both use a single atomic instruction. The thin locks scheme uses no atomic instruction in unlocking, but, as we have discussed, pays for that lack with the possibility of unbounded busy-waiting. We feel that in many situations the trade-offs made in our algorithm will be more desirable.

5.3 Flexibility

One of the main advantages we have claimed for the meta-locking approach is flexibility. This flexibility results from the fact that we place few constraints on the nature of the data structures protected by the meta-lock. Specifically, it enables separation of mechanism and policy, to allow implementation of a variety of monitor semantics.

We have tested this flexibility claim to some extent in an attempt to address two potential shortcomings of our simple linked-list data structure: lack of fairness and long searches through queues. First, consider fairness. Motivated by Buhr *et al.*, who classify and compare a spectrum of monitor "styles" that offer different trade-offs between performance and fairness [8], we programmed a version that gives preference to awakened waiters (so-called "priority non-blocking monitors"). To provide this preference, we replaced the single queue with three queues, holding entering, waiting, and awakened threads, respectively. Now it is possible to find and give preference to awakened waiters without searching. Similarly, `notify()` can execute in constant time, by moving the first thread from the waiting queue to the awakened queue. Second, consider contention. To allow threads to append lock records to queues without having to search to the end of the queue, a search which could become costly if queues get long, we kept head and tail pointers for each of the three queues. Tail pointers also allow `notifyAll()` to run in constant time, regardless of the number of threads waiting, via list concatenation. While this alternative implementation was straightforward, performance turned out to be inferior to our single-queue system because greater fairness incurs a higher context switch rate and the three-queue data structures were more heavyweight.

5.4 Hardware without SWAP or CAS

The meta-lock algorithm relies on two "exotic" atomic operations: SWAP and CAS. First, note that SWAP is easily simulated using CAS: repeatedly read the memory location and CAS until success. The CAS operation, or some other sufficiently powerful primitive such as "load-locked/store-conditional," seem to be available on most modern architectures, including mainstream Intel, UltraSPARC™, PowerPC, and Alpha microprocessors.

The JVM in which we implemented our synchronization must run on the previous generation of SPARC processors, which has SWAP but does not have CAS. While correctness cannot be compromised, it was deemed acceptable to trade away some performance and scalability on this older hardware. We first dropped the extra fast synchronization optimization because it relies directly on CAS. Next, we modified `getMetaLock()` to use a test-and-set protocol (where "set" means swapping out the locked value 1 and "test" means obtaining a non-locked value). When the test fails, the thread yields and optionally sleeps (using exponential back-off as in [2]). The corresponding `releaseMetaLock()` operation simply stores back the release bit pattern, which of course must be different from the locked value 1.

6 PERFORMANCE

Usually, good performance is taken to mean that both memory and CPUs are used efficiently. Since different systems must make space/time trade-offs differently, we shall consider space and time costs for our synchronization algorithm separately. All our measurements were collected using the E.G.C.JVM on a lightly loaded 4-CPU 296 MHz UltraSPARC system with 2 gigabytes of RAM and Solaris™ 2.6. Some measurements were obtained by adding counters to the code. To minimize the disturbance resulting from the instrumentation, we used per-thread counters that were accumulated into global totals as threads exited.

6.1 Benchmarks

Table 1 shows the benchmarks we use to assess the performance of our synchronization code. The HelloWorld program shows how the minimal program behaves. The next seven lines show widely-known SPECjvm98 benchmarks [29]. Finally, we include a selection of multi-threaded benchmarks, some of which perform significant amounts of I/O and some of which use graphics. The "#lines" column shows the approximate lines of source code in the benchmark itself, excluding class library code. The "#threads" column shows the maximum number of active threads, excluding three system threads (finalizer, reference handler, and signal dispatcher) and a short-lived secondary finalizer thread created by the SPECjvm98 benchmarks. The volano benchmark is VolanoMark version 2.0.0 build 137 [26], and the Java Web Server can be found at http://www.sun.com/software/jwebserver/index.html. The execution times in the table are best of two runs.

6.2 Space performance

We consider separately the space costs of *used* and *unused* synchronization capability.

Cost of used synchronization capability: the cost for objects that are actually synchronized upon. From the description of our algorithm, it follows that this space cost is proportional to the number of lock records in use at any point in time. More precisely, since threads recycle lock records locally rather than globally, we report the number of lock records allocated by the global allocator during the execution of each benchmark. This higher number reflects our implementation more accurately. In the worst case, a program will synchronize on every object allocated (see Section 2.2), making

				real time, seconds	
Benchmark	Description	#lines	#threads	with extra fast	without extra fast
Hello	Hello world program	5	1	0.7	0.7
_201_compress	LZW compression and decompression	927	1	45.8	46.0
_202_jess	Version of NASA's CLIPS expert system shell	10,579	1	22.3	24.8
_209_db	Search and modify a database	1,028	1	72.1	87.9
_213_javac	Source to bytecode compiler	25,211	1	42.7	48.9
_222_mpegaudio	Decompress audio file	n/a	1	50.4	51.3
_227_mtrt	Multi-threaded image rendering	3,799	2	13.2	13.6
_228_jack	Parser generator generating itself	8,194	1	34.5	38.0
_224_richards	Five threads running multiple versions of O/S simulator	3,637	5	17.3	19.0
_233_tmix	Thread mix: sort, crc, producer-consumer, primes, etc.	8,194	14	28.8	29.1
SwingMark	Benchmark and test of swing libraries	3,998	8	51.3	51.5
volano server	"Chat server," reads and distributes messages	n/a	406	9.6	10.3
volano client	Generates work-load to stress server	n/a	402		
JWS	Java Web Server, serving 20,000 requests	200,000	63	25.7	27.3

Table 1. Characterization of benchmark programs.

the worst-case space cost of our algorithm, as well as of any other algorithm that we know of, proportional to the size of the heap. Fortunately, this number is far more pessimistic than the behavior typical programs exhibit. Table 2 shows that for our benchmarks, the total number of lock records allocated is very small, and pales in comparison with the total number of objects allocated. Moreover, at 24 bytes per lock record, even the worst program seen, the volano server, consumes 77 Kbytes for lock records, a trivial amount compared with the several Mbytes used for objects and thread stacks.

Benchmark	# objects	# objs sync'ed on	# lock records
Hello	2,076	262 (12.6%)	40
_201_compress	8,917	936 (10.5%)	40
_202_jess	7,934,141	6,545 (0.1%)	40
_209_db	3,213,429	17,123 (0.5%)	40
_213_javac	5,912,859	351,538 (5.9%)	40
_222_mpegaudio	12,009	994 (8.3%)	40
_227_mtrt	6,641,320	1,195 (0.0%)	60
_228_jack	6,841,290	506,157 (7.4%)	40
_224_richards	36,065	1,878 (5.2%)	80
_233_tmix	1,366,985	169,823 (12.4%)	140
SwingMark	2,345,281	69,245 (3.0%)	100
volano server	140,335	6,334 (4.5%)	3,280
volano client	661,199	3,643 (0.6%)	3,240
JWS	1,336,170	258,960 (19%)	540

Table 2. Objects allocated, objects synchronized on, and lock records allocated.

Cost of unused synchronization capability: the cost for objects that are never synchronized upon. This cost, in our meta-lock scheme, amounts to two bits per object. However, an alternative view is that the cost is either 0 or 1 word, since it is impractical to have objects of fractional word sizes on contemporary hardware. Put differently, *if* two spare bits can be found in objects without increasing object sizes, our locking algorithm has no space cost for objects that are not synchronized upon. Otherwise, if finding two bits requires increasing the size of objects by a full word, then a different synchronization algorithm that takes advantage of a full word of memory should (probably) be used. Thus, it can be argued, our algorithm has no space overhead for objects that are not synchronized upon.

6.3 Time performance

We study time performance of our algorithm in several ways. First, Section 6.3.1 compares the cost of synchronization in our system with that of the original JVM found in the "JDK 1.2 *Reference* Release for Solaris™." For this section, as explained below, we use synthetic benchmarks. Second, Section 6.3.2 studies the behavior of our algorithm on the more realistic programs shown in Table 1, paying special attention to contention and the efficacy of the "extra-fast" optimization. Finally Section 6.3.3 compares the cost of uncontended locking in realistic programs on several VMs. We do not compare the absolute performance of the JVMs since they differ in many other respects than the synchronization code.

6.3.1 Time performance comparison with the original JVM

In this section we compare the speed of our synchronization algorithm, using the extra fast extension, with that of the monitor cache approach in the original JVM. To measure the speed of synchronization rather than the speed of context switching provided by the underlying operating system, we use programs that primarily do

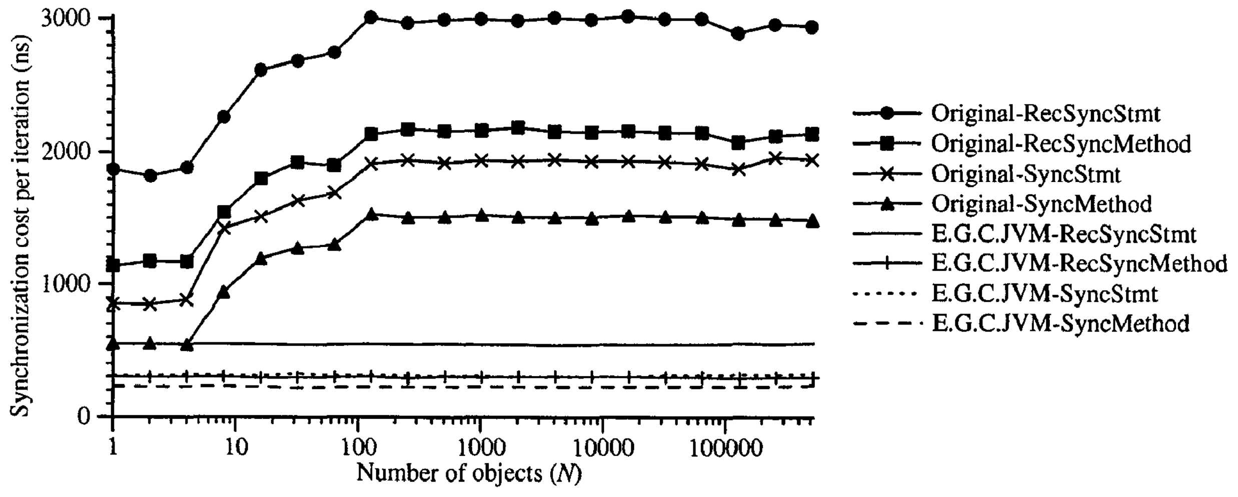

Fig. 9. Cost of synchronization.

uncontended synchronization. Section 6.3.2 shows that contention is relatively infrequent for typical Java-based programs, justifying this approach, at least in part.

Ideally, we would compare different synchronization algorithms directly by implementing them as alternatives in the same virtual machine. In our case, however, this approach was impractical. First, the implementation effort is non-trivial. Second, an algorithm added quickly to a JVM for the purpose of measuring will be at an inherent disadvantage compared with an algorithm that has been tuned with the rest of the system over a long period. Third, it may be technically impossible to keep all other factors constant, since each algorithm may take advantage of features that the other one does not use (e.g., the monitor cache works best in the presence of handles). Since E.G.C.JVM and the original JVM differ in many respects, comparing bottom-line performance does not reveal much about the two systems' synchronization code. Fortunately, a different measurement approach can give us the information we want. Consider a program P that performs synchronization. Construct the *baseline* program $\bar{P}$, which is just like P, except that all synchronization has been stripped out. The difference in execution time, $sync(P) = time(P) - time(\bar{P})$, reflects the cost of synchronization. Computing $sync(P)$ for E.G.C.JVM and the original JVM gives numbers that can be compared. The rest of this section takes this approach with simple synthetic benchmarks. Section 6.3.3 takes this approach with more realistic programs.

To this end, we constructed a set of simple benchmarks: *SyncMethod* calls a synchronized method; *SyncStmt* executes a synchronized statement; *RecSyncMethod* calls two nested synchronized methods on an object; and *RecSyncStmt* executes a pair of nested synchronized statements on an object. The first two benchmarks measure the cost of a non-recursive lock/unlock pair, and the second two measure the sum of the costs of a non-recursive and recursive lock/unlock pair. Each benchmark completes 10 million iterations, cycling through an array of length N to select the objects to synchronize on. We let N range from 1 to 512K and plot the cost per iteration; see Figure 9. As one would expect, the meta-locking scheme delivers unchanged performance regardless of the number of objects synchronized upon whereas the monitor cache approach suffers an increasing slowdown, despite the fact that in all of these tests, no more than one object is locked at any time.

The graph also shows that the absolute cost of a synchronization operation in E.G.C.JVM is always significantly lower than in the original JVM. For example, a non-recursive synchronized method call, the most frequent form of synchronization, executes in about 220 ns on E.G.C.JVM but takes 550 ns to 1500 ns on the original JVM.

6.3.2 Behavior of our algorithm on realistic programs

Consider now the algorithm's behavior on realistic programs. We first study the "pure" form of the meta-lock algorithm, without extra fast locking and unlocking. In this case, each monitor-level synchronization operation involves a `getMetaLock()` and `releaseMetaLock()` call. The left half of Table 3 shows that the fast path is taken in all but an extremely small fraction of the cases; that is, meta-lock contention is extremely rare. We instrumented only meta-lock acquisition, since the algorithm is such that the number of fast/slow `getMetaLock()` calls equals the number of fast/slow `releaseMetaLock()` calls.

Having confirmed that the meta-locking fast paths dominate, let us study those fast paths on a typical RISC processor. Figure 10 shows the SPARC instructions that result from translating a synchronization operation of the form:

```
multiUseWord = getMetaLock(ee, obj);
newMultiUseWord =
    bodyOfSyncOp(ee, obj, multiUseWord);
releaseMetaLock(ee, obj, newMultiUseWord);
```

On entry, we assume that register `%i0` holds the address of the execution environment `ee` and `%i1` holds the address of an object `obj`. It takes seven instructions to perform the fast path `getMetaLock()`, including extracting the high 30 bits of the multi-use word into one register and the low 2 bits (the lock state) into another. The code for the body of the synchronization operation would follow. At the end, we have 4 instructions for the fast path of `releaseMetaLock()`. This gives us a total of 11 instructions for the fast paths of meta-lock acquisition and release. While a careful analysis of the cycles consumed by an optimal implementation is interesting in the context of a particular architecture, for the present purposes we shall be satisfied with considering the SPARC

```
! getMetaLock
or      %i0, 3, %l0             ! %l0 = my busy value
add     %i1, 4, %l1             ! %l1 = multi-use word address
swap    [%l1], %l0              ! Swap out busy value
and     %l0, 3, %l2             ! %l2 = meta-lock state
cmp     %l2, 3                  ! Is lock state busy?
beq     slowGetMetaLockPath
sub     %l0, %l2, %l0           ! In delay slot compute high 30 bits
                                ! Slow path gets predecessor EE in %l0
                                ! and synch operation gets lock
                                ! record pointer or age&hash in %l0
... %l2 = body of synchronization operation
! releaseMetaLock
or      %i0, 3, %l0             ! %l0 = my busy value
cas     [%l1], %l0, %l2         ! if [%l1] == %l0 then swap([%l0],%l2)
cmp     %l0, %l2                ! did we do the swap?
bne     slowReleaseMetaLockPath
! unfilled delay slot here
```

Fig. 10. Fast path for a synchronization operation wrapped in meta-lock and unlock.

	without extra fast		with extra fast	
Benchmark	**# getMetaLock**	**#getMetaLockSlow**	**# getMetaLock**	**#getMetaLockSlow**
Hello	3,054	0	1,418	0
_201_compress	22,180	3	2,269	2
_202_jess	9,619,742	19	4,171	10
_209_db	106,829,540	1	2,024	5
_213_javac	34,380,756	50	39,949	61
_222_mpegaudio	22,813	10	2,620	5
_227_mtrt	1,424,925	1	2,397	3
_228_jack	23,851,600	8	2,979	5
_224_richards	70,560	59	3,434	52
_233_tmix	8,711,428	1,612	2,183,531	1,730
SwingMark	4,062,787	2,508	465,758	1,977
volano server	9,622,570	587	209,341	542
volano client	9,495,680	6	17,625	3
JWS	1,783,691	7,800	162,586	2,743

Table 3. Frequency of meta-lock contention.

implementation representative of a typical RISC implementation. The most costly instructions are the two atomic instructions, swap in `getMetaLock()`, and cas in `releaseMetaLock()`.

Now consider the performance of the system with extra fast synchronization enabled. Recall that this optimization fuses meta-locking and monitor-locking to allow monitor-lock acquisition and release each with a single atomic instruction in uncontended cases, but in contended cases falls back to the meta-lock protocol for a total cost of three atomic instructions. If monitor-lock contention is rare, as Bacon *et al.*'s data indicate [4], this will be a net win; otherwise, it could be a loss. Table 1 shows the bottom line on extra fast synchronization for our benchmarks: no program slows down, and several speed up significantly. Comparing the left and right halves of Table 3 shows that the speedup results from a significant reduction in the number of meta-locking operations, confirming that monitor-lock contention is indeed rare. However, Table 3 also shows that, for some programs, the fall-back case is sufficiently frequent that its performance cannot be neglected. Finally, the similar number of slow meta-lock operations in the left and right halves of Table 3 implies that extra fast synchronization does not reduce contention on the meta-lock. (For completeness, we should mention that a few of the meta-lock operations that remain when using extra fast synchronization result from layers in the JVM, such as class loading and JNI, that do not use the extra fast operations.)

6.3.3 Comparison with other implementations

To provide an unsynchronized baseline for realistic programs, we implemented a tool that removes all synchronization operations from class files: the bits that declare methods synchronized are reset, and `monitorenter`/`monitorexit` operations are changed into `pop`s. For programs that are essentially single threaded, such as most of the SpecJVM98 benchmarks, the resulting program still works correctly. A few classes must be excepted from this desynchronization process, including classes involved with finalizer and weak reference handling, and `java.lang.Thread`. However, measurements indicate that almost all locking operations are eliminated: in executions that perform more than one million locking operations, less than 0.1% of the original are executed after desynchronization.

Given this tool, we can measure the absolute decrease in execution time caused by executing the desynchronized class files instead of the original class files, independent of Java platform. As in Section 6.3.1, this decrease represents the cost of locking for that execution. There were four SpecJVM benchmarks that both did at least one million locking operations and still worked correctly after desynchronization. Table 4 compares locking costs for these benchmarks, on three different platforms: E.G.C.JVM, the Java HotSpot Performance Engine for Solaris/SPARC, and the IBM Developer Kit for Windows, Java Technology Edition, Version 1.1.7, which uses the thin-locks scheme. The first line of the figure gives the number of locking operations eliminated by desynchronization for these benchmarks. We measured this on the E.G.C.JVM, and used the same class files for the Java HotSpot/SPARC measurement. We were unable to complete direct measurements of the corresponding numbers for the 1.1.7 platform, but the measurements we did perform lead us to believe that assuming the same number of eliminated lock operations as in E.G.C.JVM is fairly accurate. The remaining lines of Table 4 show, for each platform, the decrease in user time, in seconds, for these benchmarks, and the number of cycles per lock/unlock pair implied by the decrease and the number of eliminated locking pairs. (The latter number is shown in italics.) E.G.C.JVM and Java HotSpot/SPARC were run on a 2-cpu 360 Mhz UltraSPARC system running the Solaris OS, and the IBM system was run on a Dell PowerEdge 2200, with 4 300 MHz Pentium processors, under Microsoft Windows NT. Cycles per lock/unlock pair assumes that each processor runs at the listed clock speed.

	_202_jess	_209_db	_213_javac	_228_jack
lock pairs eliminated	4,819,403	53,414,248	17,197,220	11,924,230
E.G.C.JVM	1.4 (*105*)	10.6 (*71*)	4.2 (*88*)	2.6 (*78*)
Java HotSpot/SPARC	1.1 (*82*)	10.0 (*67*)	4.0 (*84*)	3.1 (*94*)
IBM 1.1.7	0.8 (*50*)	13.2 (*74*)	4.9 (*85*)	2.4 (*60*)

Table 4. Comparison of uncontended locking costs on 3 platforms.

The strongest conclusion that should be drawn from these measurements is probably that these virtuals machines are fairly similar in efficiency of uncontended locking. Note that the costs on uniprocessor systems could be lower, because of decreased cost of atomic memory instructions. Also, these measurements provide no information about performance in high-contention situations.

7 CONCLUSIONS AND FUTURE WORK

We have presented a meta-locking algorithm that supports a variety of higher-level locking protocols, by providing exclusive access to the data structures used in the higher-level protocol. This meta-locking algorithm has several virtues. Like the Java HotSpot system that introduced header word displacement, the meta-locking algorithm is highly space-efficient, requiring only two reserved bits in each object, and a number of lock records that is small for normal programs. It is also reasonably time-efficient in the normal case, requiring 7 instructions to acquire and 4 instructions to release an uncontended meta-lock. Each of those paths includes a single atomic instruction. Finally, it is careful to avoid pathologies when there is contention: the algorithm introduces no busy-waiting, and only very rarely allocates from global memory.

We have also presented a particular higher-level locking protocol, based on this meta-locking algorithm, for the synchronization primitives of the Java virtual machine. An optimization of this protocol gains the efficiency of avoiding meta-locking in most cases, but the ability to fall back to meta-locking in uncommon cases regularizes and simplifies the protocol.

Finally, we have implemented and validated the performance of the meta-lock in the context of a high-performance Java virtual machine. Our measurements, which include a study of several multi-threaded programs running on a 4-CPU system, indicate that the meta-lock algorithm operates with a low contention rate to ensure that the fast path strongly dominates the performance. Synthetic benchmarks designed to isolate the cost of synchronization indicate that our scheme outperforms the original monitor cache scheme by a factor of three or more. Measurements on desynchronized benchmarks show that the meta-lock-based monitor implementation in E.G.C.JVM has performance comparable to that of the monitor implementations in two other high-performance JVMs.

In the future, we may work on extending the extra fast instruction sequences to handle more cases while continuing to use the meta-lock protocol as a comfortable fall-back. For example, if measurements justify it, extra fast locking could be extended to allow a non-empty queue with lock state `WAITERS`. We are also investigating whether the high-level synchronization state could be made to influence the order in which threads acquire meta-locks. For example, it might improve efficiency if a thread attempting to release a monitor-lock could be given preferential treatment at the meta-lock level.

Acknowledgments. Lars Bak, David Dice, David Holmes, Urs Hölzle, Doug Lea, and Hong Zhang provided very useful comments on a draft of the paper. Bill Joy advised us of interesting related work. The anonymous reviewers suggested substantial improvements to the paper.

8 REFERENCES

1. Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. *An Efficient Meta-lock for Implementing Ubiquitous Synchronization.* Sun Labs Technical Report 99-76, http://www.sunlabs.com/technical-reports/1999/.

2. Tom Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems,* 1(1), p. 6-16, January 1990.

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

4. David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI),* p. 258-268, Montreal, Canada, June 1998.

5. Lars Bak, presentation on the HotSpot JVM, Panel: The New Crop Of Java Virtual Machines. In *Proc. ACM SIGPLAN '98 Conference on Object-Oriented Programming Systems, Languages, and Applications,* p. 179-182, Vancouver, Canada, October 1998.

6. Andrew Birrell. *An Introduction to Programming with Threads.* Digital Systems Research Center report no. 35, 1989.

7. David R. Butenhof. *Programming with POSIX®* Threads. Addison-Wesley Professional Computing Series, 1997.

8. Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys,* 27(1), p. 63-107, March 1995.

9. Sylvia Dieckmann and Urs Hölzle. *A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks.* Technical Report TRCS98-33, Computer Science Department, University of California, Santa Barbara, December 1998.

10. Edsgar Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM* 8(9), p. 569, August 1965.

11. Michael Greenwald and David Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96),* p. 123-136, Seattle, WA, October 1996.

12. Per Brinch Hansen. Monitors and Concurrent Pascal: a personal history. In *Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages,* p. 1-35. Published as *ACM SIGPLAN Notices* 28(3), March 1993.

13. Per Brinch Hansen. Java's Insecure Parallelism. *ACM SIGPLAN Notices* 34(4), p. 38-45, April 1999.

14. C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM* 17(10), p. 549-557, October 1974.

15. William N. Joy. System and method for space efficient object locking using global and local locks. US Patent #5,761,670.

16. William N. Joy and Guy L. Steele. System and method for space and time efficient object locking. US Patent #5,862,376.

17. William N. Joy and Arthur Van Hoff. System and method for space efficient object locking using a data subarray and pointers. US Patent #5,701,470.

18. Andreas Krall. Efficient JavaVM Just-in-Time Compilation. In *Proc. International Conference on Parallel Architectures and Compilation Techniques (PACT'98),* p. 12-18. Paris, France, October 1998.

19. Andreas Krall and Mark Probst. Monitors and Exceptions: How to implement Java efficiently. In *ACM 1998 Workshop on Java for High-Performance Computing,* p. 15-24, Palo Alto, California, March 1998.

20. Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computing System* 5(1), p. 1-11, February 1987.

21. Xavier Leroy. *The LinuxThreads library.* http://pauillac.inria.fr/~xleroy/linuxthreads/index.html, 1997.

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

23. Peter Magnusson, Anders Landin, and Erik Hagersten. *Efficient Software Synchronization on Large Cache Coherent Multiprocessors.* SICS Research Report T94:07, Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden, February 1994.

24. John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems,* 9(1), p. 21-65, 1991.

25. Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proc. of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS),* p. 1-20, Berkeley, California, June 1997.

26. John Neffinger. Which Java VM scales best? *JavaWorld,* August 1998. http://www.javaworld.com/javaworld/jw-08-1998/jw-08-volanomark.html. See also www.volano.com.

27. Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. *Toba: Java For Applications—A Way Ahead of Time (WAT) Compiler.* Technical Report, Dept. of Computer Science, University of Arizona, Tucson, 1997.

28. Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parakh, and James M. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proc. ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, p. 280-290, Montreal, Canada, June 1998.

29. SPECjvm98 Benchmarks. August 19, 1998 release. http://www.spec.org/osg/jvm98.

30. Sun Microsystems, Inc. Java 2 on-line documentation: http://java.sun.com/products/jdk/1.2/docs/api/index.html.

31. Hong Zhang, Sheng Liang, and Lars Bak. Personal communication of draft paper: An Efficient Monitor Scheme for the Java™ Virtual Machine.

Appendix: Code for Meta-locking

The C code for fast-path and slow-paths for meta-locking is included for reference.

```
BitField getMetaLock(ExecEnv *ee, Object *obj) {

  BitField busyBits = ee | BUSY;
  BitField lockBits =
    SWAP(busyBits, multiUseWordAddr(obj));
  return getLockState(lockBits) != BUSY ?
    lockBits : getMetaLockSlow(ee, lockBits);
}
```

```
void releaseMetaLock(ExecEnv *ee, Object *obj,
                     BitField releaseBits) {
  BitField busyBits = ee | BUSY;
  BitField lockBits = CAS(releaseBits, busyBits,
                          multiUseWordAddr(obj));
  if (lockBits != busyBits)
    releaseMetaLockSlow(ee, releaseBits);
}
```

Fig. 11. Fast paths for meta-lock operations.

```
BitField getMetaLockSlow(ExecEnv *ee,
                         BitField predBits) {
  BitField bits;
  ExecEnv *predEE = busyEE(predBits);
  assert(getLockState(predBits) == BUSY);
  mutexLock(&predEE->metaLockMutex);
  if (!predEE->bitsForGrab) {
    /* Won the race: */
    predEE->succEE = ee;
    do {
      condvarWait(&predEE->metaLockCondvar,
                  &predEE->metaLockMutex);
    } while (!ee->gotMetaLockSlow);
    ee->gotMetaLockSlow = FALSE;
    bits = ee->metaLockBits;
  } else {
    /* Lost the race: */
    bits = predEE->metaLockBits;
    predEE->bitsForGrab = FALSE;
    condvarSignal(&predEE->metaLockCondvar);
  }
  mutexUnlock(&predEE->metaLockMutex);
  return bits;
}
```

```
void releaseMetaLockSlow(ExecEnv *ee,
                         BitField releaseBits) {
  /* We are in a race with our successor to
     lock ee->metaLockMutex; the winner of
     the race waits for the loser. */
  mutexLock(&ee->metaLockMutex);
  if (ee->succEE) {
    /* Lost the race: */
    assert(!ee->succEE->bitsForGrab);
    assert(!ee->bitsForGrab);
    assert(!ee->succEE->gotMetaLockSlow);
    ee->succEE->metaLockBits    = releaseBits;
    ee->succEE->gotMetaLockSlow = TRUE;
    ee->succEE = NULL;
    condvarSignal(&ee->metaLockCondvar);
  } else {
    /* Won the race: */
    ee->metaLockBits = releaseBits;
    ee->bitsForGrab  = TRUE;
    do {
      condvarWait(&ee->metaLockCondvar,
                  &ee->metaLockMutex);
    } while (ee->bitsForGrab);
  }
  mutexUnlock(&ee->metaLockMutex);
}
```

Fig. 12. Slow paths for meta-lock operations.