Featherweight Monitors with Bacon Bits

David F. Bacon
IBM T.J. Watson Research Center

Abstract

Language-supported synchronization is a source of serious performance problems in Java programs. Even for single threaded programs the overhead of synchronization in compiled Java can be as high as 45%. I address this problem with a new language-level locking algorithm suitable for both uniprocessor and multiprocessor environments. On a Pentium uniprocessor, in the most common case the lock-and-unlock overhead for a Java synchronized method is a mere 6 machine cycles when a synchronous thread scheduler is used, or 15 machine cycles when an asynchronous thread scheduler is used.

1 Introduction

Monitors [6] are a language-level construct for providing mutually exclusive access to shared data structures in a multi-threaded environment. However, the overhead required by the necessary locking has generally restricted their use to relatively "heavy-weight" objects.

Recently, their incorporation in Java [4] has led to renewed interest in monitors, because of both their prevalence and their associated performance problems. Java uses monitor semantics derived from Mesa [13]. In Java, the methods of an object may be declared synchronized meaning that the object must be locked for the duration of the method's execution. Ideally, every Java class would be thread-safe, but because of the overhead of locking, methods are only optionally synchronized.

Making synchronization optional allows the programmer to make the tradeoff between writing safe, re-usable code and achieving high performance. Unfortunately, as with the optional virtual methods of C++, a choice in favor of high performance may subsequently manifest itself in bugs that are very difficult to find. Ultimately, such optional language features lower the level of the language.

In this paper, I describe a locking algorithm for high-level language monitors, and in particular for Java monitors, that not only can substantially improve the performance of existing Java programs, but is fast enough that it would be feasible to make all public methods of an object synchronized.

The locking algorithm has the following desirable characteristsics:

- acquiring or releasing an initial lock (the most common case) only takes 4 instructions on the Intel Pentium processor;
- acquiring or releasing a shallowly nested lock (the second most common case) only takes 10 or 11 instructions, respectively, on the Pentium;
- only one word of storage per object is required for the lock, except in uncommon cases when an eight-word "fat lock" is allocated;
- locks are granted in the order in which they were requested;
- except for a rarely used global lock for short critical sections, no spin-locking is required;
- the global lock can be arbitrarily partitioned, making the algorithm fully scalable;
- the algorithm can be applied to a uniprocessor with synchronous or asynchronous thread scheduling, or to a multiprocessor that is strongly or weakly ordered — and in each case will yield an implementation with minimal cost in the most common case.

The algorithm derives its power from the use of a "Bacon bit" in the lock word, which acts as an atomically settable virtual head of the queue.

Author's address: H1-J21 Hawthorne, IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, telephone (914) 784-7811, tie line 863/7811, fax (914) 784-6576, email dfb@watson.ibm.com.

The goal was a locking algorithm with very low overhead for single-threaded programs, but also with excellent performance in the presence of multithreading and contention. These parameters are appropriate to a Java server or to a client that is running windowing or network code that is likely to involve multiple threads of control.

I believe that my locking algorithm is the best one for these design parameters, but there are other design parameters that would dictate a different locking algorithm. Many Java applications and applets are entirely single-threaded. For an environment in which such applications predominate, solutions are possible which have lower locking overhead than Bacon bit locks. However, they will be subject to significant performance degradation in the presence of multi-threading and contention.

2 The Problem

While there is fairly universal agreement that locking is one of if not the most serious performance problem for Java, it is important to get some idea of the magnitude of the problem in a quantitative manner.

Table 1 describes the benchmarks used in this paper [19]. They are all medium-sized programs and are heavily skewed towards compiler-like applications. However, they do exhibit a wide variety of programming styles.

Table 1 gives the size of each benchmark in terms of kilobytes `class` files, excluding the `class` files of any of the standard Java libraries used. The dynamic statistics from test cases include the number of locks acquired, the number of objects that were locked (not the total number of objects), the average number of lock operations per object, and the average number of lock operations per millisecond. The latter number should be a good indicator of the potential performance impact of locking on the benchmark.

Note: data for the `jgl` *benchmark was not collected due to disk space limitations. I have ordered a bigger disk.*

Figure 1 shows the overhead for locking in a prototype Java *compiler* (not an interpreter) developed at IBM for the AIX platform [7]. Using compiled Java is a better yardstick of the importance of locking because it does not include the overhead that is present in some current Java interpreters, including Sun's, which are still quite inefficient. Over time, interpreted Java code should more and more closely approximate compiled Java code in performance.

Figure 1 shows the time to run the programs with synchronization turned on ("Base") and with synchronization disabled ("NoSync"). The "NoSync" measurements actually understate the potential performance impact of locking, because it was obtained using a runtime flag. Each lock and unlock operation is still performing a procedure call, which returns immediately upon checking a "locking disabled" flag.

The median overhead due to locking is 14%, but is as high as 45% for the `javalex` benchmark. Note that these are all single-threaded benchmarks! The locking overhead is mostly due to the use of the Java libraries, which are thread-safe because they could potentially be used by multi-threaded applications. The performance improvement in "NoSync" is almost directly proportional to the number of lock operations per millisecond in Table 1.

2.1 Characterization of Locking

In order to properly optimize Java's locking performance, one must know what the most common cases are. I assume that the order of frequency of different locking scenarios would be as follows, with each scenario about an order of magnitude less common than the one preceding it:

1. locking an unlocked object.
2. locking an object already locked by the current thread a small number of times (shallowly nested locking).
3. locking an object already locked by the current thread many times (deeply nested locking).
4. attempting to lock an object already locked by another thread, for which no other threads are waiting.
5. attempting to lock an object already locked by another thread, for which other threads are already waiting.

However, these assumptions should also be validated experimentally. Figure 2 shows the frequency of the various possibilities. Because the benchmarks were single-threaded, the last two scenarios were not measured. *The*

Program	Size(K)	Description	Lock Ops	Objs	Locks/Obj	Locks/ms
trans	504	IBM High Performance Java compiler	877783	49353	18	154
javac	499	Sun's Java source to class file compiler	722789	9686	75	185
jgl	204	ObjectSpace Java Generic Library 1.1				
jacorb	308	Java Object Request Broker 0.5	733345	22101	33	105
jobe	151	Java Obfuscator 1.0	76091	7842	10	17
toba	78	U. of Arizona Java to C translator 12/96	2026520	13760	147	356
javalex	89	Lexical parser-generator for Java	1825580	9657	189	630
jax	159	Scanner Generator	411164	2808	146	50
javacup	101	Java Constructor of Useful Parsers 0.9e	336588	14400	23	224

Table 1: Java benchmarks used in this paper and overall locking activity.

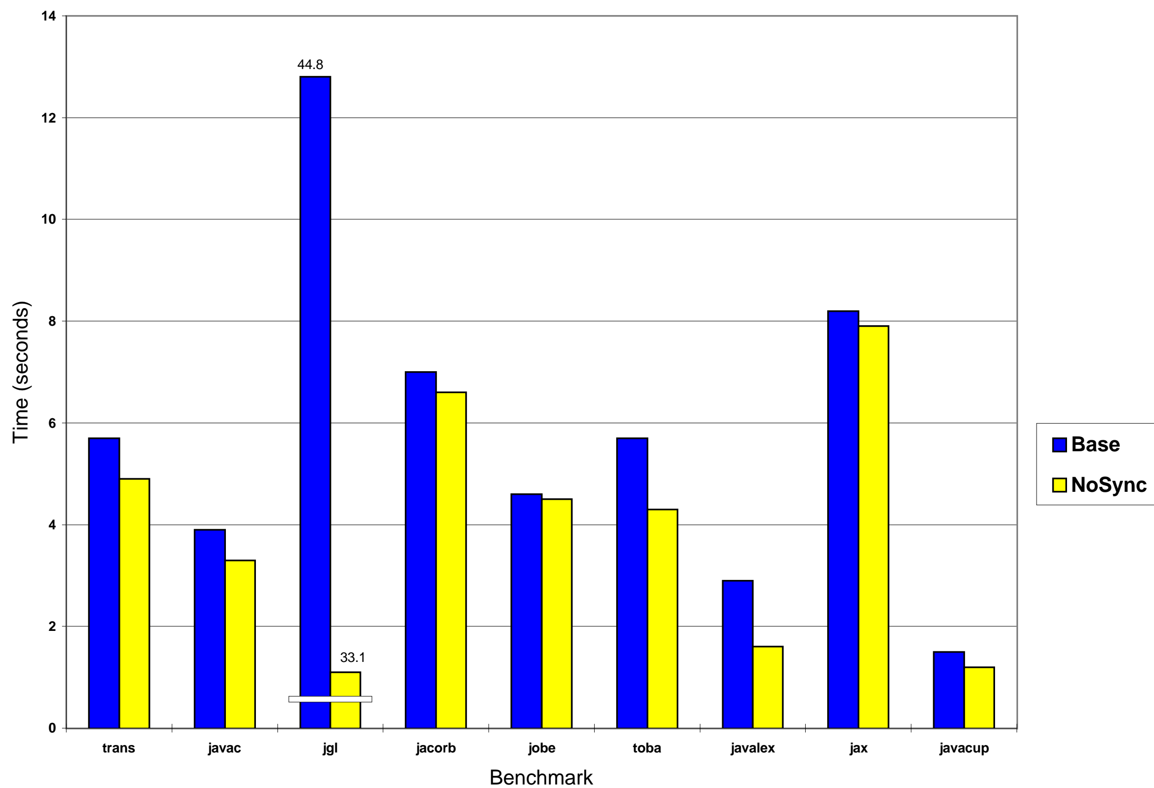

Figure 1: Locking overhead in compiled Java code. Note that the `jgl` benchmark has been rescaled to fit the graph.

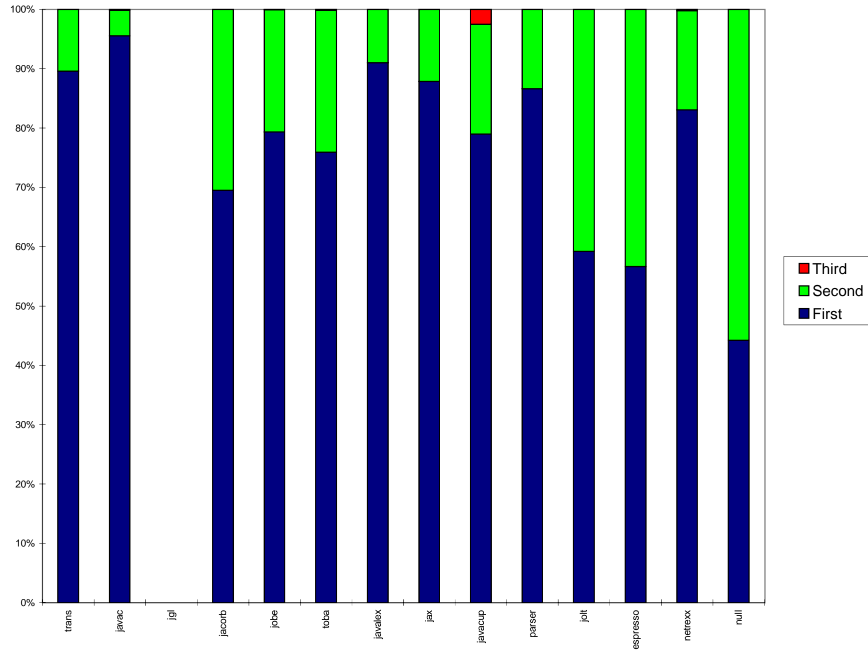

Figure 2: Depth of lock nesting by benchmark. Most lock operations are performed on objects that are not locked (they are the "First" lock on the object). Of the remaining lock operations, the vast majority are "Second" locks.

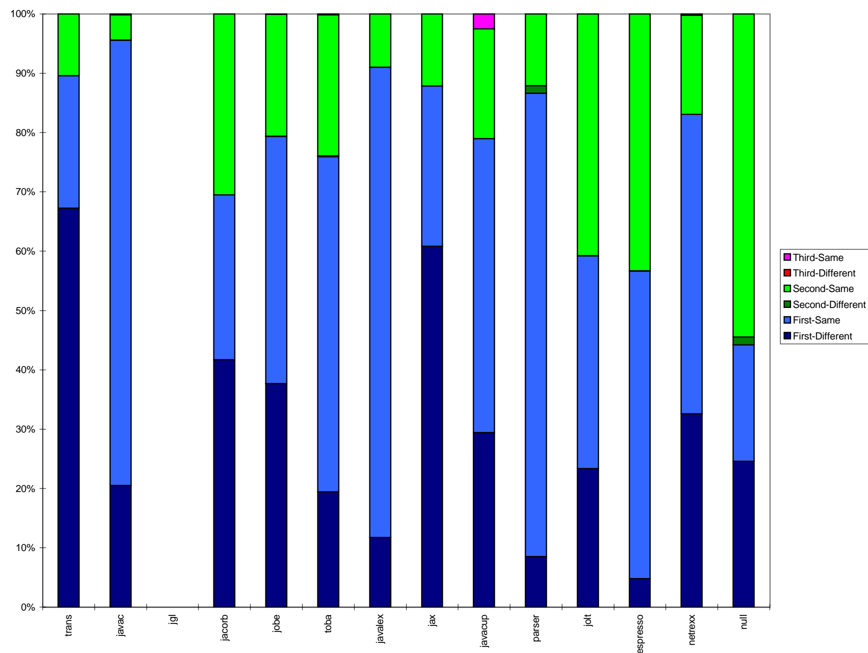

Figure 3: Frequency of re-locking by nesting level. This graph breaks down the lock operations shown in Figure 2. "Same" indicates that the lock operation was performed upon the same object as the previous lock operation; "Different" indicates that the object was different.

final paper will include measurements of WOM, IBM's Web Object Manager that was deployed to manage the Olympic Games web site. WOM is a database-oriented Java program that is heavily multi-threaded.

The measurements do show that locking unlocked objects is indeed far more common than any other case: no less than 57% of locks obtained by any application were for unlocked objects; the median is 79%.

Nesting of locks in general is very shallow: all the benchmark but one performed more than 99% of their lock operations on unlocked or singly locked objects. Only one benchmark, `javalex`, nested locks more than 4 deep: it performed nested locking up to a depth of 77 locks.

What these measurements tell us is that in most cases only a few bits need to be allocated for the lock nesting count. This observation is crucial to the design of Bacon bit locks.

Figure 3 presents the same data as Figure 2 but with additional detail: at each locking level, lock operations are divided into those that were performed upon the same object as the previous locking operation, and those that were not. There are two important observations that can be made from the data in Figure 3. First, maintaining a pointer to the most recently used lock (a one-element cache) will be very effective: a median hit rate of 74% for these benchmarks.

The second observation is that the second lock is almost always to the same object as the previous locking operation (a "cache hit"). This indicates that most nested locking occurs because one synchronized method of a class calls another synchronized method of the same class. By simply cloning synchronized methods to create a non-synchronized version for internal use by the class, it is very likely that almost all of the nested locking could be eliminated. In this event, a single bit would suffice in most cases to record the nesting depth ("first" or "deep").

3 Issues and Solutions

The structure of a Bacon bit lock is shown in Figure 4. It is contained inside of a Java object, which includes a pointer to the class structure, instance data, and possibly other fields not shown.

The lock word is composed of a pointer to (or identifier of) the thread that owns the lock, a Bacon bit, and a short count field. Throughout this paper, the thread pointer field is 26 bits and the short count field is 5 bits.

There are a number of aspects of the Java monitor semantics which introduce complications that make high-performance locking difficult. We will discuss each of these issues, and how they contribute to the design of the lock word shown in Figure 4.

3.1 Queuing

The first complication is due to queueing. If contention for objects is extremely rare, then it is acceptable for a thread to spin on a lock when it fails to acquire it [1]. In this case, no locking queues need to be maintained because the "queues" are implicit in the scheduling system.

While spinning simplifies locking considerably, it violates the design criterion that locking perform well in the presence of contention. If a thread is waiting for a lock, then on a uniprocessor it will be resumed on every cycle of the thread scheduler, only to immediately yield control of the processor to the next waiting thread. On a multiprocessor, the thread will continuously issue failing atomic operations onto the shared bus or the interconnect. In either case, performance will be degraded as the number of blocked threads increases.

The problems due to spinning can be addressed by enqueueing threads for an object when it is already locked. However, this complicates the unlock operation, which must check whether any threads are waiting for the object, and if so "hand off" the lock to one of the waiting threads instead of releasing it. Unlocking is further complicated by a potential race condition between the check for an empty queue of waiting threads, and an enqueue operation by another running thread.

Accessing the queue also presents a problem. An extra word can be allocated in the object which points to the queue or is null if no threads are waiting for the object. However, this wastes an extra word in each object. On the other hand, on the assumption that there is no contention for most objects, the queue can be accessed via a hash table, but this requires that the unlock operation perform a hash table lookup.

The solution to this problem is to introduce a "Bacon bit" into the lock. The Bacon bit is 1 if there are threads enqueued for the object (or if unstructured locking is

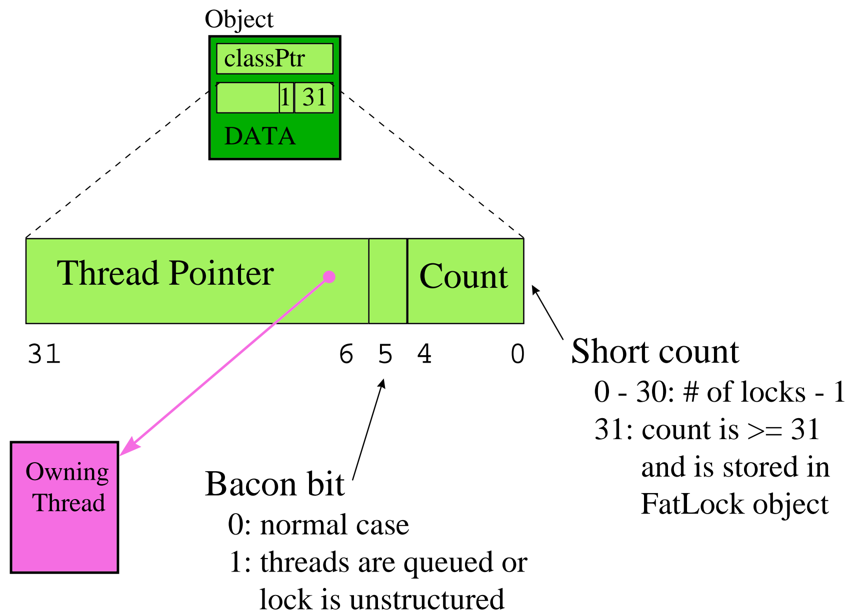

Figure 4: Structure of the lock word in a Bacon bit lock (the `lockWord` field of Figure 5).

performed – see Section 3.4), and 0 otherwise. The Bacon bit is only changed when the modifying thread has obtained a global lock, and it must be changed using a compare-and-swap operation.

By forcing any modification to a Bacon bit to occur while the global lock is held, the race condition at unlocking is avoided. By using only a single bit inside the lock word to encode the presence of waiting threads, the check for waiting threads is fast and yet the queue can be placed in a hash table. Note that the global lock is *not* acquired when the object is locked for the first time. In this situation, the Bacon bit is written to but not modified (it is 0 before and after the lock acquisition).

3.2 Nested Locking

In Java, `synchronized` method invocations by a thread may be nested. In that event, the lock is not released until the outermost `synchronized` method exits. In practice, this requires that each lock keep a counter for the number of times it has been locked.

As with queues, there is a potential tradeoff between keeping the count in the object for fast access, and keeping it in an auxiliary structure accessed via a hash table so as to minimize object size. Once again, the solution is to steal some bits from the thread pointer. We will mandate that thread objects be allocated on an address boundary that is a power of two — for the purposes of this paper, a 64-byte boundary. This leaves 5 bits for the nested locking count, plus one bit for the Bacon bit.

When the 5-bit count overflows, an auxiliary lock data structure will be allocated or found, and a full count kept there. I call this auxiliary data structure a *fat lock*.

In some situations, threads will be provided by the underlying operating system and it will not be possible to mandate the storage boundaries for threads. This does not significantly compromise the algorithm; the details of the necessary adaption are in Section 4.6.

3.3 Thread Kill Operations

Another issue that must be addressed is how to handle the semantics of the thread stop() operation, which kills a thread. When a thread is killed, all of its locks must be released.

When a Java thread is killed, a ThreadDeath error is thrown. The Java semantics dictate that when an exception causes a synchronized method to terminate, then that method must release its lock. Therefore, the call stack implicitly encodes the list of objects locked by the thread, and a separate list of locked objects does not need to be maintained. This observation is crucial to the Bacon bit locking algorithm.

3.4 Unstructured Locking

The final issue that complicates locking is the presence of unstructured locking operations. While the Java semantics specify that all locks owned by a thread must be released when it is killed, the semantics of the Java Virtual Machine allow for unstructured locking which must be tracked.

One source of unstructured locking is via the Java Native Interface (JNI). This allows Java objects to be locked and unlocked directly from non-Java code.

Unstructured locking is also generated by synchronized() blocks. While synchronized() blocks in Java source code are syntactically nested, they are expressed as bytecodes in Java class files. The Java verifier does not verify that the bytecodes enforce the nesting of the source language. Therefore, it is possible for the **monitorexit** bytecode to throw an IllegalMonitorStateException.

Whatever the source of unstructured locking, unstructured locks must be tracked to ensure that they are properly released at thread termination. This is accomplished by inflating the lock into a "fat lock" when an unstructured locking operation is performed upon the object, and setting the Bacon bit. The Bacon bit therefore does double duty, indicating that either there are threads queued to lock the object or else that unstructured locking has been performed upon the object.

In many cases it is possible to determine that the Java bytecodes representing a synchronized block *are* properly nested, in which case it is not necessary to use fat locks. This is discussed more fully in Section 6.2.

4 The Locking Algorithm

Section 3 described how individual problems associated with language-level monitors are addressed in Bacon bit locks. In this section, I describe the algorithm that is developed from the combination of these solutions, and show how the synergy yields extremely efficient lock and unlock operations.

Note that there is some additional mechanism required to handle the wait(), notify(), and notifyAll() operations that is not included in this paper due to space limitations. For the most part the implementation of these in conjunction with Bacon bit locks is straightforward.

The data definitions for the `BaconBitLock` class and the code for the `monitorEnter()` and `monitorExit()` routines is shown in Figure 5. The implementation language is C++, although type casts have been omitted in the interest of brevity.

The locking is implemented using the `AtomicWord` class, which is a single-word object with the operations `read()`, `write()`, `add()`, and `cmpAndSwap()`.

The `cmpAndSwap()` operation has the following semantics: the value of the atomic word is compared to the first argument; if they are equal, the value of the atomic word is changed to the second argument, and the operation returns true; if they are not equal, the atomic word remains unchanged and the operation returns false.

The `cmpAndSwap()` operation can be directly implemented with the Intel `CMPXCHG` instruction [9] or with the Power PC load-and-reserve and store-conditional instructions [15].

The `add()` operation atomically adds a value to the atomic word. In general it can be implemented with compare-and-swap; on the Intel architectures it can be implemented with an immediate to memory `ADD` instruction.

A Bacon bit lock consists of a single `lockWord` of type `AtomicWord`; masks are used to extract the various fields of the `lockWord` (lines 3–7 of Figure 5).

4.1 Locking

An object is locked with the `monitorEnter()` function. The `lockWord` of an unlocked object is always zero, so

```
 1 class BaconBitLock {
 2  private:
 3   AtomicWord lockWord;
 4   static const int ThreadMask = ~63,
 5                    BaconBit   =  32,
 6                    CountMask  =  31,
 7                    CountFull  =  30;
 8
 9  public:
10   inline void monitorEnter() {
11     if (lockWord.cmpAndSwap(0, thread))
12       return;
13
14     int lock = lockWord.read();
15     int count = lock & CountMask;
16     Thread* owner = lock & ThreadMask;
17
18     if (owner != thread)
19       enqueueForLock();
20     else if (count >= CountFull)
21       deeplyNestedEnter();
22     else
23       lockWord.add(1);
24   }
25
26
27   inline void monitorExit() {
28     if (lockWord.cmpAndSwap(thread, 0))
29       return;
30
31     int lock = lockWord.read();
32     int count = lock & CountMask;
33     Thread* owner = lock & ThreadMask;
34
35     if (thread != owner)
36       thread->illegalMonitorState();
37     else if (count == 0)
38       dequeueNextLocker();
39     else if (count == CountMask)
40       deeplyNestedExit();
41     else
42       lockWord.add(-1);
43   }
44 };
```

Figure 5: Inline portions of the monitor enter and exit operations.

the initial attempt to grab the lock is a single compare-and-swap call that attempts to replace zero with the pointer to the currently running thread (line 11). If the compare-and-swap succeeds, the lock has been obtained and the `monitorEnter()` function terminates successfully.

If the compare-and-swap fails, there are two possibilities: either the current thread has already locked the monitor, or another thread has locked it. The `lockWord` is read and the count and thread pointer values are masked out of it (lines 14–16). If another thread has locked the monitor, the current thread enqueues for the lock (lines 18–19). The enqueueing operation will be discussed later.

If the current thread has already locked the monitor, the nesting count must be incremented. If the count has overflowed or is about to overflow, a call is made to the function `deeplyNestedEnter()`, which looks up or creates a "fat lock" structure with the full-length count (lines 20–21).

If the short count is not full, the `lockWord` is atomically incremented and the `monitorEnter()` function returns successfully (line 23).

4.2 Unlocking

An object is unlocked with the `monitorExit()` function (lines 27–43 of Figure 5). Performing the unlock operation quickly is actually much more difficult than performing the lock operation quickly, because the unlocking thread must check that it owns the lock, that the number of locks is 1, that there are no threads waiting for the object, and that no unstructured locking operations have been performed on the object. Furthermore, if this operation can not be performed atomically, there is a potential race condition created by concurrent lockers.

However, because of the encoding of the `lockWord`, all of these conditions are checked by the single compare-and-swap operation that implements the most common case (line 28). It checks if the `lockWord` is equal to the current thread pointer, and if so, atomically changes the `lockWord` to zero, and returns.

If the `lockWord` is equal to the current thread pointer, then the current thread owns the lock, the Bacon bit is zero, and the short count is zero. If the Bacon bit is zero, there are no threads waiting for the object and no

unstructured locking has been performed on the object. If the short count is zero, then the number of locks is one (the short count is the number of locks minus one). Therefore, all conditions necessary for the unlock operation are atomically tested in a single compare-and-swap operation.

If the compare-and-swap operation fails, then the less common cases must be handled. The `lockWord` is read and the lock owner and count fields are masked out (lines 31–33).

First, a check is made to ensure that the current thread really owns the lock; if not, an exception is thrown and the operation aborts (lines 35–36).

Provided that the current thread does own the lock, the count is examined. If the count is zero, then the Bacon bit must be one (because otherwise the compare-and-swap operation would have succeeded). Therefore, a function is called that either dequeues the next waiting thread and gives it the lock, or else handles the unstructured locking case (lines 37–38).

If the short count is all ones, then the count has overflowed and a function is called to handle the long count in the fat lock object (lines 39–40). Finally, if the count is neither zero nor all ones, the short count is atomically decremented, and the function returns (line 42).

4.3 Queueing

The fast lock and unlock operations are made possible by the way in which queueing is handled. I therefore discuss this operation in some detail.

I describe an implementation of Bacon bit locks in which the fat lock objects that hold the queues of waiting threads are accessed through a global hash table, accessed via the built-in hash value associated with every Java object. Other implementations are possible: for instance, the pointer to the fat lock could be stored in the object (a space-time tradeoff). However, since access to fat locks appears to be rare, the performance penalty of hashing is acceptable.

The data structures of the locking algorithm prior to an enqueue operation are shown in Figure 6. The hash table is a private data structure of the global `lockTable` object. The `lockTable` also includes a `globalLock` field, which is the global lock that must be obtained before any queue manipulations are performed, and a `quickCells` pointer, which points to a list of pre-allocated fat lock objects.

The `globalLock` is a simple (non-nested) compare-and-swap spin-lock. It is the only spin-lock in the system. Therefore, all operations on the `lockTable` and the associated fat locks and queues must be short, non-blocking critical sections. Note that on a uniprocessor, spinning is accomplished by repeatedly testing the lock and yielding the processor; spinning in the true sense only occurs on a multiprocessor.

The fundamental invariant of Bacon bit locks is that the Bacon bit is never changed without holding the global lock. This requirement prevents race conditions in the unlock operation.

To understand the enqueueing operation, refer to Figure 6. Thread_1 has already locked Object_8, and Thread_2 is the currently running thread. There are two pre-allocated fat lock objects, and the hash table entry for Object_8 is empty.

When Thread_2 attempts to lock Object_8, it begins by executing the `monitorEnter()` function of Figure 5. However, both the compare-and-swap and the check for lock ownership fail, so the `enqueueForLock()` function is called (Figure 8).

The first thing the `enqueueForLock()` function does is to lock the global spin lock (line 2). It then enters an infinite loop that begins by copying the value of the object's `lockWord`, checks that it is non-zero, and attempts to set its Bacon bit with a compare-and-swap operation (lines 3–7). Note that the Bacon bit can *not* be set with an atomic bit set operation because this might result in a `lockWord` that was zero except for its Bacon bit, which is an inconsistent state.

If the compare-and-swap operation succeeds, the `inflate()` function is called, which either returns the fat lock for the object if it already exists, or "inflates" the lock by allocating a new fat lock from the quick cell list and inserting it into the hash table (line 8). In Figure 6 there is no fat lock for Object_8, so a new one is allocated.

The `enqueueForLock()` function then appends the currently running thread to the end of the list of waiting threads of the fat lock, unlocks the global lock, and suspends itself (lines 10–12). Eventually, when the owner of the lock releases the lock and dequeues the waiting locker, the locker will resume execution at line 13 and will now own the lock.

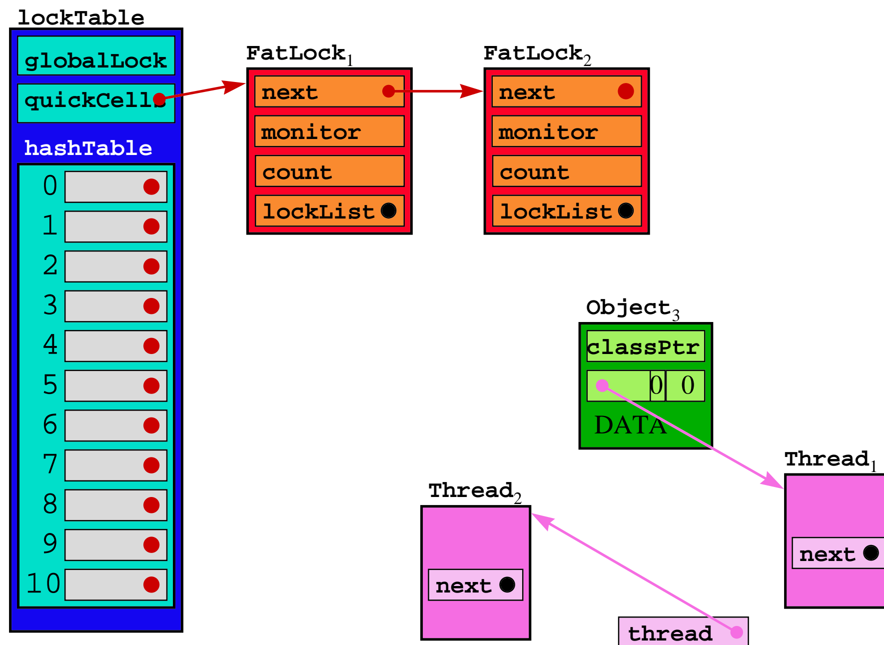

Figure 6: Locking data structures before `Thread`$_2$ attempts to lock `Object`$_8$, which is already locked by `Thread`$_1$. `thread` points to the currently running thread. The bullet "•" represents a null pointer.

The result after line 12 has been executed is shown in Figure 7. Thread$_2$ has placed itself on the `lockList` of the FatLock$_1$, the fat lock associated with Object$_8$. Since Thread$_2$ has suspended itself, Thread$_1$ is now the current (running) thread.

If the compare-and-swap operation of line 7 fails, then some other thread has been concurrently updating the `lockWord`, either to set the Bacon bit or to release the lock. Therefore, the enqueueing function attempts once again to lock the object with a simple compare-and-swap operation, and if it is successful, unlocks the global lock and returns (lines 16–19).

Otherwise, the loop is repeated. On a uniprocessor, the scheduling quantum will normally be vastly longer than the time to execute an iteration of the loop, and the loop should never be executed more than twice. On a multiprocessor, a pathological case is possible in which the enqueueing thread could hold the global lock indefinitely; therefore, on a multiprocessor the loop should have a limited number of iterations, after which the global lock is temporarily released before lock acquisition is re-tried.

4.4 Dequeueing

The dequeueing operation is simpler than the enqueueing operation, because the currently running thread already owns the lock. The `dequeueNextLocker()` function (lines 23–44 of Figure 8) is called by `monitorExit()` (lines 27–43 of Figure 5) when the owning thread is ready to release the lock but the Bacon bit is set, indicating that threads are enqueued waiting to lock the object.

The dequeue function begins by locking the global lock and getting a pointer to the fat lock. If there is a fat lock, the first queued locker thread is removed (lines 24–28).

If the fat lock does not exist, or if there is no thread waiting to lock the object, then a waiting thread or threads concurrently removed themselves from the queue (most likely because of a Java `Thread::stop()` operation – see Section 3.3), and the monitor lock is released by setting the `lockWord` to zero, the global lock is released, and the function returns (lines 30–34).

Otherwise, the lock will be given to the thread `locker`,

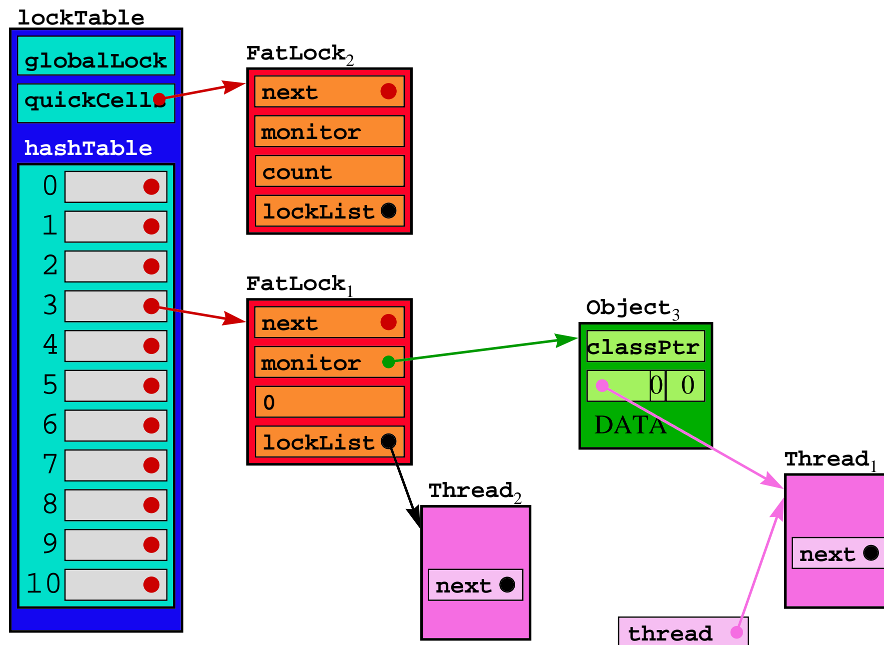

Figure 7: Locking data structures after `Thread`$_2$ has enqueued itself for `Object`$_8$ and suspended itself until `Thread`$_1$ releases the lock.

which was the first thread queued for the object. If there are still other threads waiting to lock the object, then the `lockWord` is set to the thread pointer to `locker` with the Bacon bit set (lines 36–37). If there are no other threads waiting, then the monitor is deflated (removed from the hash table and returned to the quick cell list), and the `lockWord` is set to the pointer to the `locker` thread (lines 38–41).

At this point, the lock had been given to the `locker` thread, so the `locker` is resumed, the global lock is released, and the function returns (lines 42–43).

4.5 Deeply Nested Locking

I will not describe the details of the deeply nested enter and exit functions. They are similar to the enqueue and dequeue operations in their manipulation of the fat lock objects, but are simpler because there are no manipulations of the Bacon bit.

One optimization that is worth noting is that a pointer can be kept to the most recently used deeply nested fat lock object. This pointer can then be used to avoid locking the global lock when the same object is repeatedly locked.

4.6 Adaptations

Thread Structure Alignment In some environments, it may not be possible to force the thread pointer to be aligned on anything larger than a word (4-byte) boundary. Therefore, there will only be two bits available in the `lockWord`. In this case, one bit is the Bacon bit, and the other bit is the "nested lock" bit (in other words, the short count is either 1 or "deep").

Depending on the desired space/time tradeoffs, the count can either be kept in the object or in a fat lock. If the count is kept in a fat lock, then the optimization noted above to reduce use of the global lock should definitely be employed.

In multiprocessor environments in which the compare-and-swap operation is expensive, it may also be desirable to move the count out of the `lockWord` so that the

```
 1 void BaconBitLock::enqueueForLock() {
 2   lockTable.lock();
 3   while (true) {
 4     int lock = lockWord.read();
 5     int bits = lock | BaconBit;
 6     if (lock != 0 &&
 7         lockWord.cmpAndSwap(lock, bits)) {
 8       FatLock* fat=lockTable.inflate(this);
 9
10       fat->addLocker(thread);
11       lockTable.unlock();
12       thread->suspend();
13       return;
14     }
15
16     if (lockWord.cmpAndSwap(0, thread)) {
17       lockTable.unlock();
18       return;
19     }
20   }
21 }
22
23 void BaconBitLock::dequeueNextLocker() {
24   lockTable.lock();
25   FatLock* fat = lockTable.lookup(this);
26   Thread* locker = 0;
27   if (fat)
28     locker = fat->removeLocker();
29
30   if (fat == 0 || locker == 0) {
31     lockWord.write(0);
32     lockTable.unlock();
33     return;
34   }
35
36   if (fat->moreLockers())
37     lockWord.write(locker | BaconBit);
38   else {
39     lockTable.deflate(fat);
40     lockWord.write(locker);
41   }
42   locker->resume();
43   lockTable.unlock();
44 }
```

Figure 8: Monitor enqueueing and dequeueing operations.

number of atomic operations required is minimized.

Thread Identifier Another solution to the inability to reserve bits, or to limited space in the object, is to use a thread identifier instead of a thread pointer. Many systems can not realistically support more than a few hundred threads, so a 16-bit Bacon bit lock can be allocated using 2 bits for the nested lock count, 1 bit for the Bacon bit, and 13 bits for the thread identifier (8192 possible threads).

Combining Checks By using exclusive-or operations it is actually possible to fold the two conditionals of the `monitorEnter()` function and the three conditionals of the `monitorExit()` function into a single test. Folding the conditionals reduces the number of Pentium instructions required for each function by four, and reduces the time required by one cycle. There is the additional benefit that the branch-prediction hardware (branch target buffer or branch history table) is not polluted with rarely taken branches. All instruction counts and timings for the implementation are for the version with folded conditionals.

Bi-modal Pointer An interesting variant of the algorithm suggested by Chet Murthy involves pointing directly at the fat lock object when the Bacon bit is set or the short count is overflowed.

There are a number of advantages to this variation: (1) the hash table is not needed except for the wait() and notify() operations; (2) there is no hash table lookup to find the fat lock, so enqueue, dequeue, and deeply nested locking are much faster; (3) there is no global lock – instead there is a lock on each fat lock object; and no locking is required when a thread is the first to enqueue on an object.

The disadvantages are that nested locking is slowed down when threads are waiting for an object, and that the implementation is much more complicated.

Global Lock Partitioning On a multiprocessor or a system with significant contention, the global lock controlling access to the hash table could become a source of contention and inefficiency. However, this can easily be remedied by partitioning the hash table by hash values and assigning an individual lock and list of free fat locks to each parition.

4.7 Multiprocessing

On a multiprocessor, more expensive atomic operations must be used. On the Pentium, the `LOCK` prefix must be used with the `CMPXCHG` instruction to guarantee exclusive bus access during the atomic operation.

While the Pentium is a strongly-ordered architecture, a potentially serious performance problem exists on multiprocessors that are weakly-ordered. The Java Virtual Machine Specification specifies that "locking any lock conceptually flushes all variables from the thread's working memory, and unlocking any lock forces the writing out to memory of *all* variables that the thread has assigned" [14]. This means that, in the general case, after an object is locked the modified data in the data cache must be written back to global memory.

The Pentium Pro is an example of a weakly-ordered architecture. On a Pentium Pro multiprocessor, the `LOCK CMPXCHG` instruction must be followed by a `CPUID` instruction, which is a non-privileged instruction that triggers a cache write-back.

5 Performance

For a uniprocessor Java run-time system on the Intel Pentium that uses asynchronous thread scheduling, when the `monitorEnter()` function is hand-coded in assembly language and inlined into the body of synchronized methods, locking an unlocked monitor takes four instructions or 7.5 cycles. Unlocking a monitor that has been locked once also takes four instructions or 7.5 cycles.

Most of the time is consumed by the `CMPXCHG` instruction, which takes six cycles and can not execute concurrently with any other instructions. The final half-cycle is due to a conditional branch instruction, which can be paired with the following instruction in the code body of the method (the Pentium has two integer pipelines).

By comparison, the fastest implementation to my knowledge is Microsoft's J++, which requires approximately 40 cycles of overhead for a synchronized method, almost three times as long as the 15 cycle overhead required for Bacon bit locks.

The second most common case, namely nested locking (up to 31 locks with a 5-bit short count field), takes 10 instructions or 17 cycles on the Pentium. This includes six cycles for the failed `CMPXCHG` instruction, a three-cycle branch misprediction penalty, and three cycles to atomically increment the `lockWord`. However, the branch misprediction penalty will only occur for the second lock (the first nested lock); afterwards, the branch target buffer will correctly predict the taken branch, so the cost for locks 3 through 31 will be 14 cycles.

Unlocking a shallowly nested lock takes 11 instructions — 18 cycles to release the second lock, and 15 cycles to release locks 3–31. Therefore the total overhead for a `synchronized` method performing nested locking is 35 cycles for the second lock and 29 cycles for the others.

If the Java run-time system is using a synchronous thread scheduler, then atomic instructions do not need to be used and the `CMPXCHG` instruction can be replaced with a `CMP` and a `MOV` instruction. In this case, locking an unlocked monitor requires four instructions or three cycles on the Pentium, as does the corresponding unlock operation. This yields a total overhead for a `synchronized` method of only 6 cycles!

All of these timings assume that the current thread pointer resides in the data segment and can be loaded with a single `MOV` instruction. If the thread pointer resides in another segment (as it often does), one cycle must be added to the timings.

The total size of the inlined lock and unlock code is 25 instructions. If code space is at a premium, the shallowly nested locking code can be moved into a called procedure, reducing the size of the inlined code to 12 instructions at a cost of an extra 6 cycles for shallowly nested locking.

6 Compile-Time Improvements

This section discusses some optimizations that can be performed at translation time to reduce the number of cases in which fat locks must be used.

6.1 Reducing Nested Locking

The second most common case in the locking algorithm is shallowly nested locking. This often occurs when a synchronized method of a class calls another synchronized method of the same class. However, such *self-locking* can be easily eliminated with a simple transformation.

```
synchronized (f) {
  doSomething();
}
```

(a) A Java synchronized block

```
2     aload_2
3     monitorenter
4     ...
      ...
8     aload_2
9     monitorexit
10    return
11    aload_2
12    monitorexit
13    return

Exception table:
    From To  Target  Type
    4     8      11  any
```

(b) Generated bytecodes

Figure 9: Translation of block-structured synchronization primitive into unstructured bytecodes (from [14]).

If a synchronized method is invoked by another synchronized method of the same class or one of its subclasses, the invoked method can be cloned to create a version of the method that is not synchronized. The self-locking invokers can then be transformed to invoke the non-locking version.

In a dynamic environment such as those provided by most Java run-time systems, classes may be arbitrarily extended at some future time. In such an environment, all synchronized methods can be cloned.

6.2 Reducing Unstructured Locking

As mentioned briefly in Section 3.4, Java's synchronized() blocks are converted into bytecodes that are not guaranteed by the verifier to be properly nested. Therefore, any objects locked with synchronized() blocks must use the heavy-weight "fat locks" so that if the thread is killed the locks will be released.

However, in most cases it is relatively easy to verify that the **monitorenter** and **monitorexit** bytecodes are properly nested. Figure 9 shows a Java synchronized() block and the generated bytecode instructions. A simple data-flow analysis can be used to determine that all paths following the **monitorenter** bytecode lead to a properly nested **monitorexit** applied to the same stack frame element (in this case, element 2).

A similar analysis can be used to determine that the stack frame element in question is not modified to refer to a different object between the **monitorenter** and **monitorexit** bytecodes.

7 Related Work

The MCS locks of [16] are similar to Bacon bit locks in that they only require a single atomic operation to lock and unlock an object in the most common case, and queues are used to minimize the memory system impacts of (non-local) spinning. However, MCS locks are more expensive than Bacon bit locks for the initial lock because they always require the use of a two-word lock object and therefore an extra store in the lock acquisition and an extra load in the lock release. In addition, these objects must somehow be allocated, de-allocated, and tracked for the lifetime of the lock, which will further increase the cost.

MCS locks incur the storage overhead of the lock objects for the lock owner and every waiting locker, while Bacon bit locks only require no additional storage unless there is contention, and then the only overhead is per contended object rather than per waiting thread.

However, MCS locks are faster when there is contention because the lock object points directly to the next waiter, with no need for a global lock to access the queue. MCS locks are therefore better suited to situations in which contention is the norm, as is often the case when parallel applications are built using low-level locking primitives directly. For language-level monitors, where large numbers of uncontested locks are common, I expect that Bacon bit locks will be superior.

The Bacon bit is superficially similar in function to the "syncbits" associated with the Queue On Sync Bit (QOSB) primitive proposed for the cache controllers of the Wisconsin Multicube [3].

There is a significant body of work on how to achieve mutual exclusion with only atomic read and write operations [2, 11, 12, 17, 18]. These solutions were rendered obsolete by the introduction of instructions that per-

formed compound atomic operations such as exchange, test-and-set, and compare-and-swap [8]. Such operations were later generalized to the Fetch-and-Φ primitive for multiprocessors [10], and were particularly popular on machines with butterfly-type interconnects because concurrent Fetch-and-Φ operations to the same location could be combined in the network [5].

Many microprocessors have not had compound atomic operations until relatively recently because mutual exclusion was generally considered to be the province of operating systems and parallel processors. In 1987, Lamport stated that "if the concurrent processes are being time-shared on a single processor, then mutual exclusion is easily achieved by inhibiting hardware interrupts at crucial times" [12]. He is implicitly assuming that mutual exclusion is only being used in the operating system, or that the overhead of an operating system trap is acceptable on every lock and unlock operation from user-level code.

Both Anderson [1] and Mellor-Crummey and Scott [16] provide thorough discussions of synchronization algorithms for multiprocessors and include comparative performance measurements.

8 Conclusions

I have described a novel algorithm for implementing language-level monitors that is highly optimized for the case when most locks are granted immediately. Initial lock or unlock operations only require a read of the current thread pointer and a single compare-and-swap operation.

For the most common case, invoking a synchronized method on an unlocked object, the algorithm reduces the overhead for Java's synchronized methods on a Pentium uniprocessor to only 6 machine cycles when synchronous thread scheduling is used, or 15 cycles when asynchronous thread scheduling is used. The algorithm can also be applied with minor modification to multiprocessors.

For the second most common case, invoking a synchronized method on an object already locked a small number of times by the current thread, the algorithm requires only 6 additional instructions for locking or 7 additional instructions for unlocking.

Many current Java implementations (such as the Sun JVM) use monitor enter and exit routines that require 50 to 200 instructions and are many times slower than my algorithm. Even the fastest production implementation (Microsoft's J++) is almost three times slower than Bacon bit locking with an asynchronous thread scheduler.

The algorithm requires the reservation of a single word of storage in each object. When there is contention, unstructured locking, or deeply nested locking, an additional "fat lock" is also allocated, but these situations are relatively rare. The number of fat locks created as a result of contention is no more than the number of threads.

The algorithm derives its power from an encoding of information that allows lock and unlock operations to be performed with a single compare-and-swap operation. In particular, the "Bacon bit" acts as an atomically settable virtual head of the queue of waiting objects. This prevents race conditions and avoids extra checks in the unlock operation.

I am currently implementing Bacon bit locks in conjunction with a high-performance just-in-time compiler for Java in order to fully evaluate the performance benefits and experiment with further optimizations that can be performed at translation time.

Acknowledgements

Thanks to Mauricio Serrano for alerting me to the problem, sharing his benchmarks, and providing the data in Figure 1; to Rob Strom for his help in validating the algorithm, Mark Wegman for suggesting a key improvement, Kevin Stoodley for his help with optimizations for the Pentium's branch target buffer, Pat Gallop for sharing information about his locking proposals, Chet Murthy and Ravi Konuru for help with the Java runtime system, and Alex Dupuy and Peter Sweeney for their comments on earlier drafts of this paper.

References

[1] Anderson, T. E. The performance of spin lock alternatives for shared memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (Jan. 1990), 6–16.

[2] Dijkstra, E. W. Solution of a problem in concurrent programming and control. *Commun. ACM 8*, 9 (Sept. 1965), 569.

[3] Goodman, J. R., Vernon, M. K., and Woest, P. J. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachussetts, Apr. 1989), ACM Press, New York, New York, pp. 64–75.

[4] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley, Reading, Massachussetts, 1996.

[5] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 164–189.

[6] Hoare, C. A. R. Monitors: An operating system structuring concept. *Commun. ACM 17*, 10 (Oct. 1974), 549–557.

[7] IBM. IBM high performance compiler for Java: An optimizing native code compiler for Java applications. *AIXpert* (1997). See also `http://www.alphaWorks.ibm.com/formula`.

[8] IBM Corporation. *IBM 370 Principles of Operation.*

[9] Intel. *Pentium Processor Family Developer's Manual — Volume 3: Architecture and Programming Manual*, 1995.

[10] Kruskal, C. P., Rudolph, L., and Snir, M. Efficient synchronization on multiprocessors with shared memory. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (1986), pp. 218–228.

[11] Lamport, L. The mutual exclusion problem. *J. ACM 33*, 2 (Apr. 1986), 313–348.

[12] Lamport, L. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst. 5*, 1 (Feb. 1987), 1–11.

[13] Lampson, B. W., and Redell, D. D. Experience with processes and monitors in Mesa. *Commun. ACM 23*, 2 (1980), 105–117.

[14] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, Reading, Massachussetts, 1997.

[15] May, C., Silha, E., Simpson, R., and Warren, H., Eds. *The PowerPC Architecture*, second ed. Morgan Kaufmann Publishers, San Mateo, California, 1994.

[16] Mellor-Crummey, J. M., and Scott, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst. 9*, 1 (Feb. 1991), 1–20.

[17] Peterson, G. L. A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Program. Lang. Syst. 5*, 1 (Jan. 1983), 56–65.

[18] Raynal, M. *Algorithms for Mutual Exclusion.* MIT Press Series in Scientific Computation. MIT Press, Cambridge, Massachussetts, 1986. Translated from the French by D. Beeson.

[19] Serrano, M. Java benchmark suite, 1997.