September 25, 1998
RT0269
Computer Science 15 pages

Research Report

A Study of Locking Objects with Bimodal Fields

Tamiya Onodera and Kiyokuni Kawachiya

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Study of Locking Objects with Bimodal Fields

Tamiya Onodera
Kiyokuni Kawachiya
IBM Research, Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken 242-8502 Japan

Abstract

Object locking can be efficiently implemented by bimodal use of a field reserved in an object. The field is used as a lightweight lock in one mode, while it holds the reference to a heavyweight lock in the other mode. A bimodal object-locking algorithm recently proposed for Java achieves the highest performance in the absence of contention, and is still fast enough when contention occurs.

However, mode transitions inherent in bimodal locking have not yet been fully considered. The above algorithm requires busy-wait in the transition from the light mode (inflation), and does not make the reverse transition (deflation) at all.

We propose a new algorithm that allows both inflation without busy-wait and deflation, but still maintains an almost maximum level of performance in the absence of contention. We also present statistics on the synchronization behavior of real multithreaded Java programs, which indicates that busy-wait in inflation and absence of deflation can be problematic. An initial implementation of our algorithm shows promising results in terms of increased robustness and improved performance.

1 Introduction

The Java programming language [3] has built-in support for multithreaded programming. To synchronize threads, Java adopts a high-level mechanism called monitors, and provides language constructs of `synchronized` statements and methods, where locks are associated with objects.

The field of locking has been studied both long and deeply. As a result, most modern computer systems provide two classes of locks for system programmers, *spin* and *suspend* locks. A spin lock is realized with a memory word, and by repeatedly performing against the word such atomic primitives as `test_and_set`, `fetch_and_add`, and `compare_and_swap`. On the other hand, suspend locks, typical instances of which are semaphores and monitors, must inevitably be integrated with an operating system's scheduler, and thus implemented within the kernel space.

A well-known optimization of a suspend lock, first suggested by Ousterhout [11], is to combine it with a spin lock in the user space. In acquiring this *hybrid suspend lock*, a thread first tries the spin lock, but only a small number of times (possibly only once). If it fails to grab the spin lock, the thread then attempts to obtain the actual suspend lock, and goes down to the kernel space. As a result, when there is no contention, synchronization requires just one or a few atomic primitives to be executed in the user space, and is therefore very fast.

Locking of objects is done with conventional locks. The main concern here is how associations between objects and locks are realized. A simple way is to maintain them in a hash table that maps an object's address to its associated lock. This approach is space-efficient, since it does not require any working memory in an object. However, the runtime overhead is prohibitive, because the hash table is a shared resource and every access must be synchronized.

Recently, Bacon, Konuru, Murthy, and Serrano [2] proposed a locking algorithm for Java, called *thin locks*, which optimizes common cases by combining spin locking and suspend locking as in hybrid suspend locking. What is really intriguing is that the algorithm reserves a 24-bit field in an object and makes bimodal use of the single field. This was the beginning of *bimodal locking*.

Initially, the field is in the spin locking mode, and remains in this mode as long as contention does not occur. When contention is detected, the field is put

into the suspend locking mode, and the reference to a suspend lock is stored into the field. In this way, the algorithm achieves the highest performance in the absence of contention, while it is still fast enough in the presence of contention.

However, mode transitions inherent in bimodal locking have not yet been fully considered. Transitions from and to the spin locking mode are called *inflation* and *deflation*, respectively. Their algorithm requires contending threads to busy-wait in inflation, and does not perform deflation at all.

In this paper we propose a new algorithm that allows both inflation without busy-wait and deflation, but still maintains an almost maximum level of performance in the absence of contention. We also measure the synchronization behavior of real multithreaded Java programs, which indicates that busy-wait and absence of deflation can be problematic. Finally, we evaluate an implementation of our algorithm in IBM's version of JDK 1.1.6 for the AIX operating system. The initial results are promising in terms of increased robustness and improved performance.

The remaining sections are organized as follows. Section 2 presents the base bimodal locking algorithm, which is a simplified version of thin locks. Section 3 describes our bimodal locking algorithm in detail. Section 4 gives measurements for multithreaded Java applications, and Section 5 shows performance results of an implementation of our algorithm. Section 6 discusses related work, and Section 7 presents our conclusions.

2 The Base Locking Algorithm and Related Issues

The base locking algorithm is a simplified version of *thin locks* by Bacon, Konuru, Murthy, and Serrano [2]. We simplify the original algorithm to concentrate on those portions relevant to mode transitions. We start with an overview of thin locks, derive a simplified version, and discuss issues inherent in bimodal locking.

2.1 Thin Locks

Thin locks [2] was the first bimodal locking algorithm to allow a very efficient implementation of monitors in Java. It is deployed across many IBM versions of JDK, including JDK 1.1.6 OS/2, JDK 1.1.6 AIX, JDK 1.1.4 for NetworkStation 1000, and JDK 1.1.4 OS/390.[1] The important characteristics are:

Space It assumes that a 24-bit field is available in each object for locking,[2] and makes bimodal use of the lock field, with one mode for spin locking and the other for suspend locking.

Speed It takes a few machine instructions in locking and unlocking in the absence of contention, while it still achieves a better performance in the presence of contention than the original JDK.

Software Environment It assumes that the underlying layer provides suspend locks with the full range of Java synchronization semantics, namely, (heavyweight) monitors.

Hardware Support It assumes the existence of a compare-and-swap operation.

Like hybrid suspend locking, thin locks optimize common cases by performing hardware-supported atomic operations against a lock field. However, they optimize not only the most common case — that of locking an unlocked object — but also the second most common case — that of locking an object already locked by the same thread a small number of times. They do so by carefully engineering the structure of a lock field or, specifically, by using as *entry counts* 8 bits in a lock field in the spin locking mode.

2.2 The Base Algorithm

We made two major simplifications to the original algorithm:

- Extension of the lock field to make it a full word.
- Omission of optimization for shallowly nested locking.

[1] As of October 9, 1998

[2] They did not reserve the field by increasing the size of an object. The base Java virtual machine adopts the handles-in-headers object structure, and an object's header includes a field for storing the object's hash code. They invented a technique that allows the field to be reduced to two bits, and made 24 bits free and available to their algorithm.

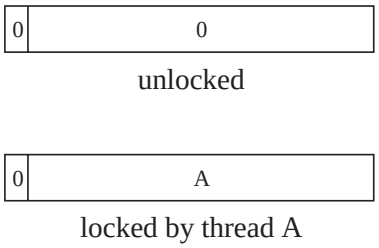

Figure 1: Flat lock structure

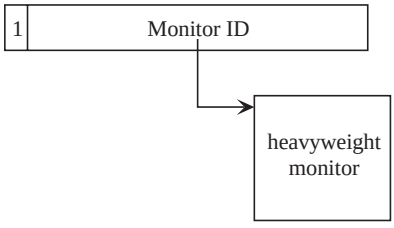

Figure 2: Inflated lock structure

Except in these two respects, the base algorithm shares all the abovementioned important characteristics of thin locks.

We now present the base locking algorithm. First, the lock word takes one of two structures - a *flat* structure for spin locking and an *inflated* structure for suspend locking. These structures are distinguished by one bit, called the *shape bit*.

The flat structure is shown in Figure 1. The value of the shape bit is 0, and the remaining bits hold a thread identifier. The value of the lock field is 0 if nobody owns the object. Otherwise, the value is the identifier of the owning thread. On the other hand, the inflated structure contains a shape bit with a value of 1 and a monitor identifier, as in Figure 2. Notice that, when the lock word of an object is in the inflated mode, the object may or may not enter the monitor.

When an object is being created, the lock word is initialized to 0, which indicates that it is unlocked in the flat mode. The compare-and-swap operation takes three arguments, and atomically performs the task represented by the pseudo-C code in Figure 3.

2.2.1 Locking

Figure 4 shows the base algorithm. In locking an object, it first attempts to grab a flat lock by using a compare-and-swap operation. If the compare-and-swap succeeds, the lock word was 0 (unlocked in the flat mode), and becomes the current thread's identifier (locked by the current thread).

If the compare-and-swap fails, there are two possibilities. In one case, the lock word is already in the inflated mode. The conditional in the `while` loop immediately fails, and the current thread enters the object's monitor. In the other case, the lock word is in the flat mode but the object has been locked by some other thread. Namely, *flat lock contention* happens. The thread then enters the inflation loop, which we will explain in Section 2.2.3.

2.2.2 Unlocking

In unlocking an object, the current thread first tests the lock word to determine whether it is in the flat mode. If so, the current thread unlocks the flat lock by simply storing 0 into the lock word. Otherwise, it takes the object's monitor out of the lock word, and exits from the monitor.

Notice that unlocking in the flat mode does not require any atomic operation. This is because an important discipline is imposed on the algorithm that only the thread locking an object can modify the lock word, except for the initial compare-and-swap in the `lock` function. Hence, the algorithm achieves the highest performance in the absence of contention: one compare-and-swap in locking, and one bit-test followed by one assignment in unlocking.

2.2.3 Inflation

When a thread detects flat lock contention, the thread enters the inflation loop in the `lock` function. In the loop, it performs the same compare-and-swap operation in order to grab the flat lock. The inflating thread needs to acquire the flat lock before modifying the lock word, because of the abovementioned discipline.

If the compare-and-swap succeeds, the current thread calls the `inflate` function. It creates a monitor for the object, enters the monitor, and stores in the lock word the monitor's identifier with a shape bit

```
int compare_and_swap(Word* word,Word old,Word new){
 if (*word==old){
  *word=new; return 1; /* succeed */
 } else
  return 0;  /* fail */
}
```

Figure 3: Semantics of compare-and-swap

```
void lock(Object* obj){
 /* flat lock path */
 if (compare_and_swap(&obj->lock,0,thread_id()))
   return;

 /* inflation loop */
 while ((obj->lock & SHAPE_BIT)==0) {
  if (compare_and_swap(&obj->lock,0,thread_id())){
   inflate(obj);
   return;
  }
  yield();
 }

 /* inflated lock path */
 monitor_enter(obj->lock & ~SHAPE_BIT);
 return;
}

void unlock(Object* obj){
 if ((obj->lock & SHAPE_BIT)==0) /* flat lock path */
   obj->lock=0;
 else  /* inflated lock path */
   monitor_exit(obj->lock & ~SHAPE_BIT);
}

void inflate(Object* obj){
 MonitorId mon = create_monitor(obj);
 monitor_enter(mon);
 obj->lock = (Word)mon | SHAPE_BIT;
}
```

Figure 4: Base locking algorithm: a simplified version of thin locks

whose value is 1. If it fails, the thread needs to perform busy-wait until it detects that some other thread has completed inflation or it succeeds in the compare-and-swap in the inflation loop.

2.2.4 Deflation

Like the original algorithm, the base algorithm does not attempt *deflation* at all. Once the lock word has inflated, all subsequent attempts to lock the object take the inflated lock path, even if contention no longer occurs.

Actually, deflation is difficult to realize in the algorithm. For instance, storing 0 in the lock word just before calling `monitor_exit` in the `unlock` function does not work. What may happen is that, while one thread acquires the suspend lock at the inflated lock path in the `lock` function, another thread simultaneously succeeds in the initial compare-and-swap.

2.3 Inherent Issues

Inflation and deflation are unique to bimodal locking. Thin locks inflate the lock words by busy-waiting, and never deflate them, as we have seen in the base algorithm. We summarize and consider the reasons and justifications given by the authors of [2] for their approach to mode transition.

They deem busy-wait in inflation to be acceptable for two reasons. First, the cost is paid once because of the absence of deflation. Second, the cost is amortized over the lifetime of an object, assuming *locality of contention*, which states that, if contention occurs once for an object, it is likely to occur again for that object. Furthermore, they mention that standard backoff techniques like those proposed by Anderson [1] can be used to ameliorate the cost and the negative effect of busy-waiting.

However, the above reasons rely on their choice of no deflation, which may not necessarily be a right thing to do. In addition, Anderson's techniques are considered and evaluated in situations in which spin locking is useful, namely, where the critical section is small or where no other process or task is ready to run. Locking activities in Java are so diverse that we do not think that these techniques are straightforwardly applicable to Java.

They do not deflate lock words, for two reasons. The first of these is locality of contention. The choice prevents lock words from thrashing between the flat and inflated modes. The second is that the absence of deflation considerably simplifies the implementation.

However, locality of contention is not verified. The real programs they measured were single-threaded applications, which obviously resulted in no contention.[3] As we will see, our measurements of real multithreaded programs show that locality of contention does not exist in many more cases than one might expect. On the other hand, the second reason is understandable. As we explained above, deflation is quite difficult to realize in the algorithm.

3 Our Locking Algorithm

We describe a new locking algorithm that allows both inflation without busy-wait and deflation, but still maintains an almost maximum level of performance in the absence of contention.

The algorithm requires one additional bit in an object. The bit is set when flat lock contention occurs, and hence named the *flc* bit. An important requirement is that the flc bit of an object belongs to a different word from the lock word, since the bit is set by a contending thread without holding a flat lock.

Figure 5 shows our locking algorithm. The first thing to note is that, if a thread fails to grab the flat lock in the inflation loop, it waits on a monitor; it does not busy-wait at all.

The second thing to note is that the unlock function, which is responsible for notifying a thread waiting in the inflation loop, first tests the flc bit *outside* the critical region. This means that, in the absence of contention, the additional overhead is only one bit-test. Hence, an almost maximum level of performance is maintained. We explain in Section 3.1 why this unsafe bit-test does not cause a race hazard.

The third thing to note is that the algorithm conditionally deflates an object's lock word at Line 36. The necessary condition is that nobody is waiting on the object. Furthermore, as long as the necessary condition is satisfied, our algorithm allows selective defla-

[3]This is not nonsensical, because the most important contribution of thin locks is that it removes the performance tax Java imposes on single-threaded applications.

```
 1 void lock(Object* obj){
 2  /* flat lock path */
 3  if (compare_and_swap(&obj->lock,0,thread_id()))
 4   return;
 5
 6  /* inflated lock & inflation path */
 7  MonitorId mon=obtain_monitor(obj);
 8  monitor_enter(mon);
 9  /* inflation loop */
10  while ((obj->lock & SHAPE_BIT)==0){
11   set_flc_bit(obj);
12   if (compare_and_swap(&obj->lock,0,thread_id()))
13    inflate(obj, mon);
14   else
15    monitor_wait(mon);
16  }
17 }
18
19 void unlock(Object* obj){
20  /* flat lock path */
21  if ((obj->lock & SHAPE_BIT)==0){
22   obj->lock=0;
23   if (test_flc_bit(obj)){/*the only overhead*/
24    MonitorId mon=obtain_monitor(obj);
25    monitor_enter(mon);
26    if (test_flc_bit(obj))
27     monitor_notify(mon);
28    monitor_exit(mon);
29   }
30   return;
31  }
32  /* inflated lock path */
33  Word lockword=obj->lock;
34  if (no thread waiting on obj)
35   if (better to deflate)
36    obj->lock=0; /* deflation*/
37  monitor_exit(lockword & ~SHAPE_BIT);
38 }
39
40 void inflate(Object* obj, MonitorId mon){
41  clear_flc_bit(obj);
42  monitor_notify_all(mon);
43  obj->lock = (Word)mon | SHAPE_BIT;
44 }
45
46 MonitorId obtain_monitor(Object* obj){
47  Word lockword=obj->lock;
48  MonitorId mon;
49  if (lockword & SHAPE_BIT)
50    mon=lockword & ~SHAPE_BIT;
51  else
52    mon=lookup_monitor(obj);
53  return mon;
54 }
```

Figure 5: Our locking algorithm

tion, which is the purpose of the condition in Line 35. For instance, we can deflate lock words on the basis of dynamic or static profiling information. Again, we explain in Section 3.2 why deflation is so simple to realize in our algorithm, and does not cause problems such as those described in Section 2.2.4.

The last thing to note is the way in which monitors are used. In inflating an object's lock word, all the code related to inflation is basically protected by the corresponding monitor; a notable exception is the unsafe test of the flc bit. Interestingly, the monitor is simply the same as the suspend lock whose reference is eventually stored in the lock word. As we will show in Section 3.3, our algorithm ensures that a monitor's dual roles do not interfere with each other.

To understand the duality a bit better, consider the case in which a thread locks an object already in the inflated mode. The thread fails in the initial compare-and-swap in the `lock` function. It then looks up and enters the object's inflation monitor. The only remaining thing is to fail in the conditional expression of the `while` loop. Notice that the object's monitor is entered after the `while` loop in the base algorithm; it is already entered as the inflation monitor in our algorithm.

Finally, we mention the `lookup_monitor` function; the code is not shown in Figure 5. We assume that there exists an underlying hash table that maintains associations between objects and their monitors. Given an object, the function searches the hash table, and returns the object's monitor, after creating a monitor if necessary. Notice that deflation purely means the mode transition of a lock word; it does not imply the removal of the corresponding association from the table.

3.1 Testing an Flc Bit

We explain why testing an flc bit outside the critical region does not cause a race hazard. Specifically, we show that no thread continues to wait forever in the inflation loop without receiving any notification. We start with the following two properties, which can be obtained immediately from the code in Figure 5.

Property 3.1 *An object's shape bit is set only in the* `inflate` *function, and cleared only at Line 36 in the* `unlock` *function. In both cases, the object's monitor is entered.*

Property 3.2 *An object's flc bit is set only in the inflation loop, and cleared only in the* `inflate` *function. In both cases, the object's monitor is entered.*

We then prove the following crucial property, which states that the failing compare-and-swap has an important implication. There are subtle issues related to this property on a multiprocessor system, which we will consider in Section 3.4.

Property 3.3 *If a thread T fails in the compare-and-swap against an object in the inflation loop, there is always some other thread that subsequently tests the object's flc bit at Line 23 of the* `unlock` *function.*

Proof. From Property 3.1, the object's shape bit remains the same when T fails in the compare-and-swap at t_1 as when T finds the `while` loop's conditional `true`. That is, the lock word is in the flat mode at t_1.

The failure then implies that some other thread U holds the object's flat lock at t_1. In other words, U does not execute the store of 0 into the lock word at Line 22 of the `unlock` function. Hence, U unsafely tests the flc bit after t_1. □

The following lengthy property is all that we can theoretically state about what happens after a thread waits in the inflation loop.

Property 3.4 *If a thread T waits on an object's monitor M in the inflation loop, there is always a thread that subsequently calls the* `inflate` *function against the object or executes the compare-and-swap against the object in the inflation loop.*

Proof. Let t_1 be the time at which T performs the compare-and-swap in the inflation loop that fails and causes T to wait on M. From Property 3.3, there exists some other thread U that unsafely tests the corresponding object's flc bit at some time t_2 after t_1.

We then perform a two-case analysis based on whether U finds the flc bit set or cleared at t_2. Consider the simpler case, in which U finds the flc bit cleared at t_2. From Property 3.2, the clearance of the flc bit implies that a third thread calls `inflate` before t_2. Since T already enters M at t_1, V calls `inflate` after T waits on M and implicitly exits from M. Thus, the property holds in this case.

Next, consider the case in which U finds the flc bit set at t_2. The thread U then succeeds in the unsafe test and continues to execute test-and-notify, which is properly protected by M. The properly protected test may fail or succeed. If it fails, obviously, the flc bit is cleared. By the same reasoning as above, we can induce that the property holds.

If the test succeeds, the thread U notifies M, and wakes up one of the waiting threads, which may or may not be identical to T. Again, notice that, since T already enters M at t_1, U calls `notify` after T waits on M and implicitly exits from M. The woken-up thread W eventually resumes the execution and reaches the `while` loop's conditional. W may find the conditional `false` or `true`. If the conditional is `false`, some thread sets the shape bit, which implies, from Property 3.1, that the thread has called `inflate`. If the conditional is `true`, the thread W continues to execute the body of the inflation loop; that is, it executes compare-and-swap. Thus, we have shown in both cases that the property holds. □

In theory, the execution of the compare-and-swap in the inflation loop repeatedly fails and is retried forever. Our algorithm lacks fairness, like the base algorithm. However, we can state that, in practice, a thread eventually succeeds in the compare-and-swap and calls the `inflate` function.

Thus, a practical consequence of Property 3.4 is that, if a thread waits on an object's monitor in the inflation loop, there is always a thread that subsequently calls the `inflate` function. Since the function wakes up all the threads waiting on the inflation monitor, this means that every thread waiting on the inflation monitor at the call is eventually woken up.[4]

3.2 Deflating a Lock Word

We show that our deflation is safe. Consider a general locking sequence in which a thread T locks an object, executes the code in the critical section, and unlocks the object. Let us consider that T acquires the object's lock when T returns from the `lock` function, and releases the lock when T calls (rather than completes) the `unlock` function. Let us also consider that T holds the lock between the two times. We can then state the safety of our deflation as follows.

Property 3.5 *No thread ever acquires an object's flat lock when some other thread holds the object's suspend lock. Similarly, no thread ever acquires an object's suspend lock when some other thread holds the object's flat lock.*

Proof. In order for a thread to acquire an object's flat lock, the thread needs to succeed in the first compare-and-swap in the `lock` function. However, it never succeeds as long as some other thread holds the suspend lock.

Similarly, in order for a thread to acquire an object's suspend lock, the thread T needs to enter the object's monitor *and* fail in the `while` loop's conditional. The conditional never fails as long as some other thread holds the object's flat lock. □

Indeed, what makes our deflation simple and safe is that the shape bit is always tested after the monitor (re-)enters. Notice that deflation of an object does not imply that the association between the object and its suspend lock is instantaneously removed from the underlying hash table. Thus, it is safe even if the lock word of an object becomes deflated in the middle of the `obtain_monitor` function.

3.3 Monitors' Dual Roles

The immediate concern about using a monitor both for protecting the inflation code and suspend-locking a Java object is that these two roles may interfere with each other. More specifically, one concern is that notifying an inflation monitor might wake up a thread waiting on the Java object, and another is that notifying a Java object might wake up a thread waiting on the inflation monitor.

Actually, our locking algorithm ensures that neither case occurs, if we properly define the internal functions for waiting on and notifying a Java object, `wait_object` and `notify_object`. First, the `wait_object` function forces inflation of the lock word if the lock word is in the flat mode.[5] This is done under appropriate protection by the object's inflation monitor. In addition, the `unlock` function suppresses deflation as long as a thread is waiting on a Java object. Combining the two, we obtain the following property:

[4] Because of deflation, the woken-up thread might wait on the inflation monitor again.

[5] Notice that the thread which calls `wait_object` in the flat mode already holds the flat lock.

Property 3.6 *If some thread is waiting on a Java object, the lock word is in the inflated mode.*

Second, the `notify_object` function performs one of the following two actions. If the object's lock word is in the flat mode, the function simply ignores the notification request, since the contrapositive of Property 3.6 states that no thread is waiting on a Java object in this mode. Otherwise, the function notifies the corresponding suspend lock. This implementation immediately yields the following property:

Property 3.7 *If a thread notifies a Java object, the lock word is in the inflated mode.*

Third, our locking algorithm notifies an inflation monitor in two places, one in the `unlock` function and the other in the `inflate` function. Notice that both places are protected by the monitor. A thread of control reaches the former only when the flc bit is set, and the latter only when the shape bit is cleared. Each of the conditions holds if and only if the lock word is in the flat mode. We thus have the following property:

Property 3.8 *If a thread notifies an inflation monitor, the lock word is in the flat mode.*

Finally, our locking algorithm causes a thread to wait on an inflation monitor M in one place, that is, in the inflation loop that is entered only when the shape bit is cleared. Furthermore, calling `inflate` against M wakes up all the waiting threads on M. This gives the following property:

Property 3.9 *If a thread is waiting on an inflation monitor, the lock word is in the flat mode.*

We are now ready to conclude the section. From Property 3.8 and the contrapositive of Property 3.6 we can infer that it is impossible for notifying an inflation monitor to wake up a thread waiting on the Java object. Similarly, from Property 3.7 and the contrapositive of Property 3.9, we can infer that it is impossible for notifying a Java object to wake up a thread waiting on the inflation monitor.

3.4 MP Issues

In general, special care must be taken in implementing a locking algorithm on a multiprocessor system. This is the case for the base algorithm. For instance, when implemented on a PowerPC multiprocessor system, which is weakly ordered, the store of 0 in the lock word at Line 22 of the `unlock` function must be preceded by the `sync` instruction, to ensure that any stores associated with the shared resource are visible to other processors. As a result, the performance in the absence of contention is not as high as in a uniprocessor system.

Notice, however, that the store does not have to be followed by `sync`. Although a thread trying to lock the same object may see the stale value and thus fail in the compare-and-swap, this simply results in a few more iterations in the busy-wait loop.

Our algorithm imposes more stringent requirements. As we have seen above, the proof of Property 3.3 relies on the implication that, if T fails in the compare-and-swap, U has not yet executed the store of 0 in the lock word at Line 22.

However, the implication is not necessarily true in a multiprocessor system, whether it is strongly ordered or weakly ordered. Thus, we need to issue an additional instruction to ensure that Property 3.3 remains valid. More specifically, on a PowerPC multiprocessor system, the store must also be followed by `sync`. The failure of the compare-and-swap then implies that U has not yet started executing the next instruction of the `sync` instruction. Thus, Property 3.3 still holds, although the additional instruction slows down the performance in the absence of contention.

Besides the store, the same care must be taken with the setting of the flc bit in the inflation loop and the clearance of the flc bit in the `inflate` function. However, both are in the inflation path, and do not affect the performance in the absence of contention.

4 Measurements

To evaluate approaches to mode transitions, we have measured the synchronization activities of multithreaded Java programs in IBM JDK 1.1.4 for the AIX operating system. In particular, we are interested in the locality of contention and the durations for which objects are locked. The former is related to deflation, and the latter to busy-wait in inflation.

We made some additions to the JDK code to log var-

ious synchronization events for measurements. We ran all the programs under AIX 4.1.5 on an unloaded RISC System/6000 Model 43P containing a 166-MHz PowerPC 604ev with 128 megabytes of main memory, and took timing measurements by using the PowerPC's time base facility, whose precision is about 60 ns on our machine. We disabled the JIT compiler for these measurements.

We consider that the lifetime of each object consists of *fat* and *flat* sections, one alternating with the other. An object is said to be in a fat section either when a thread is waiting on the object or when a thread has attempted to lock the object but has not yet acquired the lock. Otherwise, an object is said to be in a flat section.

As long as an object is in a flat section, the object is not involved in any suspend locking operations. Thus, in terms of performance, objects should be in the flat mode in flat sections. On the other hand, an object in a fat section is really involved in suspend locking operations. Objects should therefore be in the inflated mode in fat sections. An object having at least one fat section is said to be *heavily synchronized*.

Tables 1 and 2 summarize the programs we measured and their runtime statistics, respectively. We measured the number of threads created, the maximum number of threads that simultaneously exist, the number of objects created, the number of objects that are synchronized, and the total number of synchronization operations.

In addition, we include in Table 2 the number of objects that are heavily synchronized, and the total number of synchronization operations by those objects. As we see in the table, less than 1% of synchronized objects are heavily synchronized, but these objects are involved in more than 18 times as many synchronization operations on average.

4.1 Locality of Contention

Using our terminology, we can rephrase the locality of contention as follows: fat sections are long, or flat sections following fat sections are short. Here, time is represented by the number of synchronization operations. We verify these claims in this section.

We first divide heavily synchronized objects into two groups. Objects in the *nowait* group are only involved in mutual exclusion by monitor enter and exit, while objects in the *wait* group are also involved in long-term synchronization by `wait` and `notify(All)`.

Table 3 shows the lengths of fat and flat sections for each group of objects. Clearly, we observe locality of contention in the wait group. However, we do not see any such tendency in the other group. This suggests that deflation should be performed for objects in the nowait group.

The average number of fat sections per object in the eSuite program is interesting. All the heavily synchronized objects in the nowait group have only one fat section. We suspect that contentions were accidental for these objects.

4.2 Durations of Locking

Busy-wait begins to have a negative effect on performance when a thread holds an object's lock for a long time, keeping other threads in the inflation loop. We therefore measured the lengths for *locked* sections.

An object's locked section starts when a thread acquires the object's lock, and ends when the thread releases the lock. Notice that the thread may implicitly release the lock to wait on the object, and implicitly acquire the lock after returning from the wait. When a garbage collection is invoked within a locked section, we subtract the time spent in the collection from the length of the locked section, since the JDK's garbage collector is stop-the-world.

Table 4 shows the results. As we see, in comparison with the average length, there are a few sections that are unusually long. This suggests that busy wait is potentially dangerous. Furthermore, notice that if we perform deflation as recommended above, busy-wait in inflation is likely to be attempted many more times.

5 Performance Results

In this section, we evaluate an implementation of our locking algorithm in JDK 1.1.6 for IBM's AIX operating system. We based the implementation on that of thin locks contained in the original JDK. Thus, we use a 24-bit lock field, and include optimization for shallowly nested locking.

We adopted a deflation policy as suggested by the measurements in the previous section; that is, we de-

Table 1: Description of programs measured

Program	Description
eSuite	An office suite from Lotus. Version 1.0. Open the desktop and read 62 pages of a presentation file.
HotJava	A Web browser from Sun. Version 1.0. Open an HTML page containing ten 40-KB animated-GIF images.
Jigsaw	An HTTP server from W3C. Version 2.0 Beta2. Let it serve requests from ten clients simultaneously. Each client receives two hundred 5-KB files.
Ibench	A Java application that implements a business logic. Create twelve threads and let them perform transactions.

Table 2: Overall synchronization statistics

Program	Threads created	Threads that exist simulta -neously	Objects created	Objects sync'd	SyncOps by sync'd objects	SyncOps per sync'd object	Objects heavily sync'd	SyncOps by heavily sync'd objects	SyncOps per heavily sync'd object
eSuite	35	27	214820	22771	2195829	96.4	116	56535	487.4
HotJava	21	21	80951	13191	1652768	125.3	33	380770	11538.5
Jigsaw	71	71	346995	39269	1854118	47.2	209	424324	2030.3
Ibench	14	14	599963	79007	6092938	77.1	109	903542	8289.4

Table 3: Locality of contention

Program	Group	Objects	SyncOps	Fat sections per object	SyncOps per fat section	SyncOps per flat section
eSuite	nowait	14	6596	1.00	28.36	250.64
	wait	102	49939	4.51	101.11	5.59
HotJava	nowait	22	170370	5.14	26.44	1319.53
	wait	11	210400	27.09	657.26	49.58
Jigsaw	nowait	141	387382	5.06	38.53	466.58
	wait	68	36942	65.78	6.56	1.62
Ibench	nowait	109	903542	18.53	72.68	296.74
	wait	0	0	—	—	—

Table 4: Durations of locked sections

Program	Locked sections	Average duration	Longest duration	Durations $\geq$10 ms (ratio)
eSuite	895349	0.146 ms	32.13 s	953 (0.11%)
HotJava	709970	0.625 ms	3.46 s	3178 (0.45%)
Jigsaw	669851	0.316 ms	1.23 s	5042 (0.75%)
Ibench	2986946	0.519 ms	25.64 s	13062 (0.44%)

flate the lock word of an object if the object belongs to the *nowait* group and if a fat section of the object ends. We thus check these two conditions at the location corresponding to Line 35 in Figure 5.

The check is realized as follows. First, in order to determine whether an object belongs to the *nowait* group we add a counter to the object's monitor, which is incremented when the `wait_object` function is called. Notice that this incurs virtually no execution overhead, since only the execution of `wait_object` is affected. Second, we determine when an object's fat section ends by checking whether both the entry and waiting queues of the object's monitor are empty. In most cases, the information needed for checking this is already included in the underlying layer's internal structures. Thus, if these structures are available and accessible, checking the second condition does not involve any extra overhead.

We measured the performance of two versions of JDKs on the same machine with the same configuration as in the previous section, and took a median of ten runs for each measurement. Table 5 summarizes the three micro-benchmarks we used.

The `LongLocker` benchmark test is intended to determine the effect of inflation without busy-wait, and Figure 6 shows the results. As we see, as the number of concurrent threads increases, the performance of the original JDK badly deteriorates, while our JDK maintains a constant performance.

The `FlatFat` benchmark test is intended to determine the effect of deflation. Figure 7 shows the results for $n = 40$ and $m = 40000$. As we see, once inflation has occurred, the original JDK no longer performs as well in subsequent flat sections as in the first flat section, while our JDK maintains a constant performance in all the flat sections. We also obtained similar results for other cases such as $n = 10,20,80$.

The `Thrashing` benchmark test was written so that each time one thread locks an object, it ends up with a contention with the other thread. The result for $m = 2000$ is that, while the original JDK takes 1966.4 msec to complete, our JDK takes 1977.6 msec, inflating and deflating the object exactly two thousand times.

The rate of more than one cycle of inflation and deflation per msec is extremely high if we consider that the scheduling quantum is of the order of ten milliseconds in our machine. Nevertheless, our JDK performs as well as the original JDK, which means that thrashing does not pose a serious problem in our algorithm.

We suspect that this is mainly because of the underlying hash table that we use to maintain associations between objects and monitors. We simply enabled the implementation of the monitor cache in the Sun JDK 1.1.6, which includes an optimization of small per-thread monitor caches. A thread first looks up its own small monitor cache for an association. In doing so, it does not have to hold any lock. Thus, as long as a hit occurs in the per-thread cache, obtaining an object's monitor is almost as efficient in the flat mode as in the inflated mode.

Finally, Table 6 shows the dynamics of our algorithm in the real Java programs mentioned in the previous section[6]. As we can see, significantly more synchronizations occur in the flat mode in our locking algorithm.

While the removal of busy-wait in inflation is purely aimed at increasing robustness or avoiding disasters, deflation is aimed at improving performance. Obviously, the effect of deflation on overall performance depends on several factors, particularly, how much time an application spends on synchronization. All of eSuite, HotJava, and Jigsaw perform many I/O operations, and the elapsed times of these programs thus unpredictably varied from one run to another. As a result, we could not see any consistent differences between the original and our JDKs for these programs.

On the other hand, Ibench is computation-intensive, and gives scores in terms of transactions per minute. Here we observed that the scores are improved by 1.77%. Although this may look small, a speedup of 1%–2% in the interpreter mode often results in a speedup of 10%–20% in the JIT compilation mode. With this expectation, we are currently working on modifying the JIT compiler to accommodate our locking algorithm.

[6] The figures for the dynamics do not necessarily coincide with those in the measurements given in the previous section. This is primarily because of the different amounts of code inserted for recording different kinds of synchronization events.

Program	Description
LongLocker n	One thread locks an object for a long computation (about 5 sec), while $n-1$ threads attempt to lock the same object.
FlatFat n m	The flat section, where one thread executes a small synchronized block $n \times m$ times, alternates with the fat section, where n threads concurrently execute the same block m times.
Thrashing m	Each of two threads iterates over a small synchronized block m times in such a way that each iteration forces contention to occur and disappear.

Table 5: Micro-benchmarks

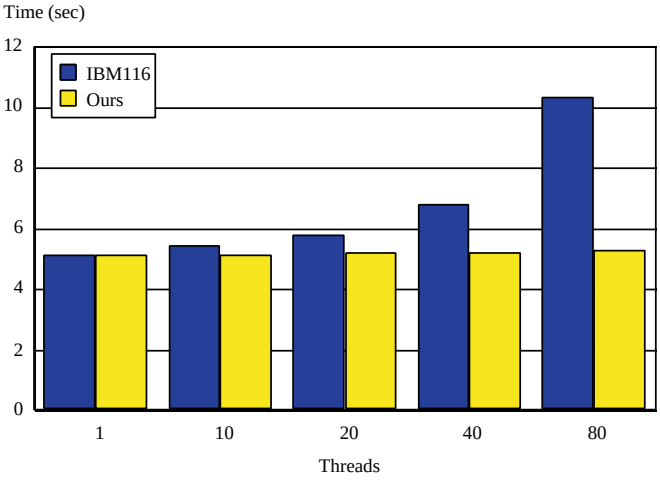

Figure 6: Performance of the `LongLocker` benchmark

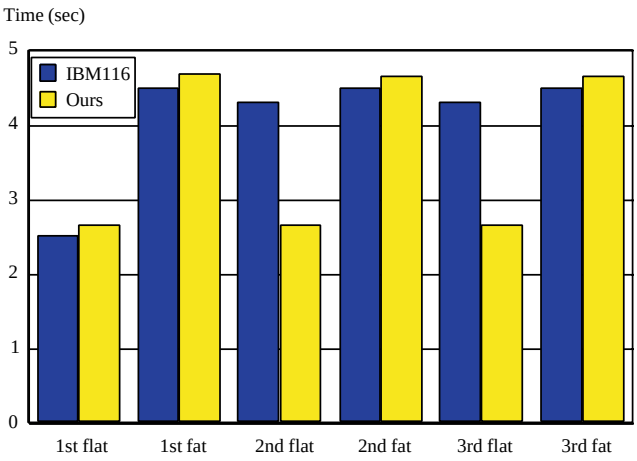

Figure 7: Performance of the `FlatFat` benchmark

Table 6: Inflation and deflation dynamics of various Java programs

Program	IBM116				Ours			
	SyncOps	Infla -tion	Defla -tion	SyncOps in inflated mode (ratio)	SyncOps	Infla -tion	Defla -tion	SyncOps in inflated mode (ratio)
eSuite	1642879	118	0	0.78%	1643217	137	126	0.39%
HotJava	813954	23	0	11.4%	853686	70	63	1.65%
Jigsaw	1612467	140	0	27.0%	1616936	659	592	6.54%
Ibench	16852849	199	0	26.4%	17030564	3963	3963	2.27%

6 Related Work

A significant number of papers and text books deal with the topic of locks. Here we mainly concentrate on locks for objects.

Mesa [9, 6] provides language support for *monitoring* of large numbers of small objects. A *monitored record* is a normal Mesa record, except that it implicitly includes a field for a monitor lock. It is of a predefined type named `MONITORLOCK`, which can be thought of as the following record [9]:

```
RECORD [locked: BOOLEAN, queue: QUEUE]
```

The size is one word (16 bits at that time) [6], which suggests packing.

In our terminology, Mesa directly stores a suspend lock, not a reference to one, in the lock word. A suspend lock can be so small in Mesa for two reasons. First, recursive entry into a monitor is not allowed. Second, a suspend lock is solely used for mutual exclusion. Long-term synchronization is done through a condition variable for which an additional field must be explicitly declared.

Although some Java implementations, notably earlier versions of Kaffe [12], take a similar approach, the space overhead is prohibitive, since Java requires recursive locking, and allows the arbitrary object to be involved in long-term synchronization.

Krall and Probst [5] use a hash table for implementing monitors in a JIT-based Java Virtual Machine. The approach is space-efficient, since it does not require any dedicated field within an object. However, the time overhead is unacceptable, because it is necessary to synchronize with the hash table each time a thread enters and exits from a monitor.

IBM JDK 1.1.2 for the AIX operating system takes a hybrid approach to implementing monitors [10]. It basically uses an implementation by a hash table. However, when it detects that an object enters the monitor frequently, the IBM implementation directly stores the reference to the monitor in the object's header.

In doing so, it takes advantage of their handles-in-headers object layout. Each object's header consists of two words, one of which mainly holds the hash code and is less frequently used. The implementation stores in the word the reference to a suspend lock (monitor) by moving the displaced information into the suspend lock's structure. Thus, it does not increase the header size at all.

Bacon, Konuru, Murthy, and Serrano [2] defined the first bimodal locking algorithm, which allows a very efficient implementation of monitors in Java. We have already fully described that algorithm in Section 2. Actually, their work inspired us to deeply consider issues inherent in bimodal locking.

Ousterhout [11] first suggested hybrid suspend locking. The main issue is the strategy for determining whether and how long a competing thread should spin before suspending. Karlin, Li, Manasse, and Owicki [4] empirically studied seven spinning strategies based on the measured lock-waiting-time distributions and elapsed times, while Lim and Agarwal [7] derived static methods that attain or approach optimal performance, using knowledge about likely wait-time characteristics of different synchronization types. These results can be used in bimodal locking if the initial compare-and-swap are tried more than once.

However, the space overhead has never been an issue

in hybrid suspend locking. In other words, it is totally acceptable to add one word to a suspend lock structure for spin locking. Indisputably, the idea of bimodal use of a single field was a painful one to reach, because the bits in an object's header are extremely precious, and increasing the header size is simply prohibitive.

Mellor-Crummey and Scott [8] proposed a sophisticated spin-locking algorithm that performs efficiently in shared-memory multiprocessors of arbitrary size. The key to the algorithm is for every processor to spin only on separate locally accessible locations, and for some other processor to terminate the spin with a single remote write operation. Although the algorithm constructs a queue of waiting processors in the presence of contention, it is a spin-locking algorithm; neither hybrid nor bimodal. Our work is orthogonal to theirs, and an interesting future direction will be to use their algorithm for locking in the flat mode in our algorithm.

7 Concluding Remarks

We have defined a new bimodal locking algorithm that allows both inflation without busy-wait and deflation. The algorithm maintains an almost maximum level of performance in the absence of contention. Two intriguing points in our algorithm are the way in which the flat lock contention bits are manipulated, and the dual roles of monitors.

We have also measured real multithreaded Java applications, and shown that locality of contention does not necessarily exist, and that in some cases objects are actually locked for a long time. This suggests that we should perform deflation, and remove busy-wait from inflation.

In addition, we have evaluated an implementation of our algorithm in IBM JDK 1.1.6 for AIX. In comparison with the original JDK, the results of micro-benchmarks show that our algorithm achieves a constant performance even in the presence of a long-time lock holder, and recovers the highest performance in the absence of contention even after inflation has occurred. They also suggest that thrashing is not a concern.

Although removal of busy-wait in inflation aims at avoiding disasters, deflation aims at improving performance. Actually, we observed a speedup in a computation-intensive server application. Currently, we are working on enabling the JIT compiler to our locking algorithm, with the expectation that performance gains due to deflation become amplified in the JIT compilation mode.

Acknowledgments

We thank David Bacon and Chet Murthy for valuable information on thin locks and their comments on an earlier version of our locking algorithm. We also thank Takao Moriyama and Kazuaki Ishizaki for making available their expertise in the PowerPC architecture.

References

[1] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1(1), 6–16 (1990).

[2] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin Locks: Featherweight Synchronization for Java. *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation* (1998), pp. 258–268.

[3] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996.

[4] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. *Proceedings of the 13th Annual ACM Symposium on Operating Systems Principles* (1991), pp. 41–55.

[5] Andreas Krall and Mark Probst. Monitors and Exceptions: How to Implement Java Efficiently. *ACM 1998 Workshop on Java for High-Performance Network Computing.*

[6] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM* 23(2), 105–117 (1980).

[7] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems Systems* 11(3), 253–294 (1993).

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

[9] James G. Mithcell, William Maybury, and Richard Sweet. *Mesa Language Manual.* CSL-79-3, Xerox Palo Alto Research Center (April 1979).

[10] Tamiya Onodera. *A Simple and Space-Efficient Monitor Optimization for Java.* Research Report RT0259, IBM Research, Tokyo Research Laboratory (July 1998).

[11] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. *Proceedings of the 3rd International Conference on Distributed Computing Systems* (1982), pp. 22–30.

[12] Transvirtual Technologies, What is Kaffe OpenVM? `http://www.transvirtual.com/kaffe.html`, (current October 7, 1998).