June 15, 2000
RT0369
Computer Science 13 pages

Research Report

A Single-Atomic Algorithm for Spin-Suspend Locking

Tamiya Onodera

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

Limited Distribution Notice
This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalities).

A Single-Atomic Algorithm for Spin-Suspend Locking

Abstract

Locks for multithreaded applications are commonly implemented by augmenting suspend locking with spin locking. This allows locking and unlocking operations in the absence of contention to be performed as highly as spin locking, involving only a few instructions in the user space. However, as far as we know, all the existing algorithms are double-atomic, requiring two atomic operations, one in locking and the other in unlocking, in the uncontended path.

We propose a novel, single-atomic spin-suspend algorithm. It centers on a generalized technique for converting a spin-wait loop into a suspend-wait loop, while the timing dependency cleverly created upon the failing atomic operation guarantees that any suspend-waiting thread is eventually signaled. We also present an intriguing variation for multiprocessor systems with relaxed memory models, which does not require an additional memory barrier.

1 Introduction

Building applications with threads has already been a common practice, supported by almost all the commercial operating systems through their thread libraries. Such a multithreaded application inevitably involves sharing various data among threads, and needs *locks* to synchronize access to them.

Two basic algorithms for locks are *spin* and *suspend* locking. In either algorithm, a thread attempts to acquire a lock by testing the availability and establishing the ownership. In modern architectures, the thread relies on a hardware primitive, such as `test_and_set` and `compare_and_swap`, to atomically perform the availability test and the ownership establishment.

Assume that the thread fails in the availability test. In spin locking, the thread repeatedly performs the same steps until it eventually succeeds in lock acquisition. In suspend locking, the thread suspends execution, and relinquishes control of processor. Since the failing test of availability and thread suspension must be done atomically, a suspend locking is implemented tightly coupled with the operating system's scheduler.

While spin locks are dominantly used in multiprocessor operating systems, suspend locks are normally preferred in multithreaded applications. Suspend locking for user applications can be efficiently realized by hybridizing it with spin locking. That is, a thread first attempts to acquire a lock with spin locking, but only spins at most a certain number of times. Only when it continually fails in all the spins, the thread calls an operating system's service for suspending itself.[1] Thus, as long as no contention occurs, the thread can acquire the lock with a few instructions in the user space; it does not have to make any expensive system calls.

This *spin-suspend* locking is first suggested by Ousterhout [21], and now widely employed in thread libraries, although the details vary substantially. The upper limit on how many times precursory spins are attempted is called *spin threshold.* While the threshold must obviously be one on an uniprocessor system, it is determined by a *spinning strategy* on a multiprocessor system [14, 17].

[1]The service may be provided specifically for the thread library, heavily depending on its implementation details.

Machines	Execution Times (msec)			Ratios to Base	
	Base	Write	Cmpswp	Write	Cmpswp
RS/6000 Model 43P	19006	19510	36975	1.03	1.95
RS/6000 Model F50	15007	15413	33261	1.03	2.22

Table 1: Relative costs of atomic operations: The `base` program iterates 64 million times over a loop computing the inner product of two three-dimentional vectors of the floating-point type. The `write` inserts an extra write operation into the loop body, while the `cmpswp` program adds an extra compare-and-swap operation instead of the write. The compare-and-swap is implemented on the PowerPC architecture with the instruction pair of load and store with reservation. Each execution time represents the median of eleven runs. The RS/6000 Model 43P contains a 133-MHz PowerPC 604e processor, while F50 contains four 332-MHz PowerPC 604e processors. The PowerPC 604 processor includes a Branch Processing Unit, two Single-Cycle Integer Units, a Multi-Cycle Integer Unit, a Load/Store Unit, and a Floating-Point Unit [12].

The primary concern about locking algorithms is the performance in the absence of contention, since experience shows that contention is rare. Basically, the performance in the uncontended path correlates with the number of atomic operations required, since the path is very short and atomic operations are heavy. In the absence of contention, some algorithms require only one atomic operation, one in locking but none in unlocking, whereas others need two atomic operations, one in locking and the other in unlocking. We call the former *single-atomic* and the latter *double-atomic*.

In this paper we propose a single-atomic algorithm for spin-suspend locking. To our best knowledge, no such algorithms have been known so far, while there are fairly straight-forward single-atomic algorithms for spin locking. We believe that single-atomic algorithms are getting increasingly important for two reasons. The first one is related to the application trend. It could be argued that contention-free locks are rarely a bottleneck in applications. However, as more and more applications and libraries are becoming multithread-safe, many more locks are getting involved in a single program, which may collectively have an impact on performance. For instance, as reported in [2], a Java source-to-bytecode compiler executes 765,000 lock operations per second on a high-performance virtual machine.

The second one is related to the architecture trend. In general, executing an atomic operation puts significant constraints on hardware optimizations performed by today's dynamic scheduling processors and memory systems, whether it is realized as a single instruction or by the code sequence consisting of the load-linked and store-conditional instructions. The negative effects by these constraints are getting larger as the hardware optimizations are more aggressive, and as the system contains more processors.

Table 1 summarizes the results of a simple attempt to quantify this trend. We measured three artificial programs, `base`, `write`, and `cmpswp`, on both one-way and four-way machines. The `base` program iterates over a loop which computes the inner product of two vectors. The `write` program has one write operation artificially inserted into the loop, while the `cmpswp` program adds a compare-and-swap operation instead of the write operation. As shown in the figure, although the processor could effectively hide the cost of one additional write operation, the extra atomic operation in the loop has greatly increased the execution time; it more than doubles on the 4-way machine.

The rest of the paper is organized as follows. Section 2 introduces a single-atomic spin locking, together with the impact of memory models on locking algorithms. Section 3 presents our single-atomic spin-suspend locking

```
void spin_lock(int *lock){
  while (compare_and_swap(lock, UNLOCKED, LOCKED)==0)
     ;
}

void spin_unlock(int *lock){
  *lock=UNLOCKED;
}
```

Figure 1: Single-atomic spin locking

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

Figure 2: Semantics of compare-and-swap

algorithm. Section 4 deals with issues encountered when our algorithm is written in multiprocessor systems. Section 5 shows preliminary results from an implementation. Section 6 discusses related work, and Section 7 presents our conclusions.

2 Single-Atomic Spin Locking

Single-atomic algorithms are well-known for spin-locking. Basically, the algorithms involve one word for representing whether or not the lock is available. They then attempt to acquire a spin lock by executing an atomic operation for changing the word from the unlocked state to the locked state, but release the lock simply by writing into the word the value for the unlocked state.

Figure 1 shows an example. It uses as an atomic operation a version of compare-and-swap, which atomically performs the task represented by the pseudo-C code in Figure 2. Actually, we could use different atomic operations, and apply optimizations such as spin on read and exponential backoff [3].

Behind this algorithm we can see an important discipline for writing a single-atomic algorithm, which is called the *nonlockers' discipline*. Assume that a thread needs to acquire a lock to access a set of shared data. Obviously, this implies that, while a thread is holding the lock, the thread is allowed to modify the shared data without any atomic operation, and no other threads are allowed to modify them in any way.

Extending this to a lock's data structure, the nonlockers' discipline states that, given a field of a lock, any thread that is not holding the lock does not modify the field. If the discipline is obeyed for a lock field, the lock-holding thread can then modify the field with a simple write.

Before we conclude this section, let us mention about a complication which arises in implementing a locking algorithm on a multiprocessor system supporting a relaxed memory model [1]. Memory models of advanced commercial architectures, including IBM PowerPC, Digital Alpha, and Sparc V9 RMO, are so relaxed that they

```
typedef struct {
  int       lock;     /* initially, UNLOCKED */
  int       wcount;   /* initially, 0 */
  mutex_t   mutex;
  condvar_t condvar;
} tasuki_t;
```

Figure 3: Data structure

no longer guarantee the program-order execution of read and write operations.

While these models allow aggressive hardware optimizations to hide read and write latency, they pose a challenge to writing a locking algorithm. For instance, assume that, using the above algorithm, a thread running on a processor acquires a lock, performs memory operations to the corresponding shared data, and releases the lock. The problem is that the write operation for the release might be made visible to other processors before the memory operations to the shared data.

Normally, such a processor architecture provides one or more special hardware instructions, called *memory barriers*, for enforcing program ordering. Thus, we could resolve the abovementioned problem by issuing a memory barrier just before the lock release.

3 Our Locking Algorithm

We describe a novel, single-atomic algorithm for spin-suspend locking, called *tasuki lock*.[2] Figure 3 shows the data structure, while Figure 4 shows the `tasuki_lock` and `tasuki_unlock` functions which are placed in the user space. As seen in Figure 4, the `lock` field is used much like the simple-atomic spin locking in the previous section.

The figure also shows that the `unlock` function now includes a test of the `wcount` field, which indicates whether or not the lock-holding thread needs to take some action on lock release for other threads having fallen back to suspend-locking. As long as no contention occurs, the test fails, and the `tasuki_resume` function is never called. Furthermore, notice that this is just a simple test; it is not in a critical section. Thus, our algorithm is single-atomic, only adding one extra test in the uncontended path in comparison with the fastest spin-locking algorithm.

Figure 5 describes the steps taken when tasuki lock falls back to suspend locking. Here we use as primitives mutex variables and condition variables as defined by the POSIX threads interface [13]. Notice, however, that we simply use them for the illustrative purpose. Actually, these two functions should be directly implemented in the kernel space in a customized manner, rather than built upon a thread library.

As shown in Figure 5, the `tasuki_suspend` function contains the `while` loop in which the calling thread attempts to acquire the spin lock. However, this is not a *spin-wait* loop, but a *suspend-wait* loop. Obviously, the algorithm must guarantee that any thread waiting at Line 21 will eventually be unblocked.

For this purpose, the algorithm keeps track of the number of currently suspend-waiting threads in the `wcount` field. The counter is incremented and decremented under the mutex's protection. Thus, it is possible to correctly know whether or not some thread is suspend-waiting, by checking the counter under the same protection. However,

[2]In Japan, *tasuki* are worn for tucking up sleeves, resulting in the shape of the letter 'x' on the back. As we will see later, the most important characteristic of our algorithm is that the shape is formed by the write/read dependency arrows of two fields in the data structure.

```
 1 void tasuki_lock(tasuki_t *tsk){
 2   if (compare_and_swap(&tsk->lock, UNLOCKED, LOCKED))
 3     return; /* ok */
 4   tasuki_suspend(tsk);
 5 }
 6
 7 void tasuki_unlock(tasuki_t *tsk){
 8   tsk->lock = UNLOCKED;
 9   if (tsk->wcount)
10     tasuki_resume(tsk);
11 }
```

Figure 4: Tasuki locking algorithm – the user-space code

```
12 void tasuki_suspend(tasuki_t *tsk){
13   mutex_lock(&tsk->mutex);
14   while (1){
15     tsk->wcount++;
16     if (compare_and_swap(&tsk->lock, UNLOCKED, LOCKED)){
17       tsk->wcount--;
18       break;
19     }
20     else
21       condvar_wait(&tsk->condvar, &tsk->mutex);
22   }
23   mutex_unlock(&tsk->mutex);
24 }
25
26 void tasuki_resume(tasuki_t *tsk){
27   mutex_lock(&tsk->mutex);
28   if (tsk->wcount>0){
29     tsk->wcount--;
30     condvar_signal(&tsk->condvar);
31   }
32   mutex_unlock(&tsk->mutex);
33 }
```

Figure 5: Tasuki locking algorithm – the kernel-space code

the `unlock` function checks the counter without any protection, and may read a wrong value of the counter. Nonetheless, our algorithm provides the abovementioned guarantee, as we soon show.

We first prove the following 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 deal with in Section 4.

Property 3.1 *If a thread T suspend-waits at time t, there is always some other thread that signals after t.*
Proof. Let t_1 be the time at which T performs the compare-and-swap in the loop that fails and causes T to suspend-wait. This implies that some thread S holds the lock at t_1. Let s_1 be the time at which S executes the lock release at Line 8. Obviously, t_1 before s_1. Since T increments the counter before t_1, and S tests the counter after s_1 in the `tasuki_unlock` function, the test returns `true`. S thus calls the `tasuki_resume` function.

The thread S then attempts to check the counter under the mutex's protection. It is only after t that S acquires the mutex, since T needs to first release the mutex at t.

We then perform a two-case analysis. If S finds the counter nonzero at Line 28, S does signal (after t). Otherwise, it means that T already got out of the loop, in which case some other thread intervened and signalled (still after t). In either case, we have the desired result. □

Then, we can obtain the following property.

Property 3.2 *If a thread T suspend-waits in the loop at time t, one or more threads continue to signal until T has been unblocked.*
Proof. By Property 3.1, some thread signals after t, which unblocks one waiting thread. If it is T, the property holds. Otherwise, the unblocked thread S retries to acquire the lock in the loop. If it fails, S suspend-waits, and Property 3.1 ensures that there is some thread which signals later. If it succeeds, S finds the counter nonzero on lock release, calls `tasuki_resume`, and signals, as long as T is suspend-waiting. By repeating this process, we have the conclusion. □

In theory, T could never be unblocked since the `condvar_signal` function might continue to choose one among the other threads. Also, even if T is unblocked, T fails again in retrying the compare-and-swap. In theory, T could continue to be unblocked and suspend-wait. Our algorithm lacks fairness. However, in practice, we could consider that T eventually succeeds in the compare-and-swap in the loop, and acquires the lock.

3.1 Discussion

The algorithm presented shows how we can convert a spin-wait loop into a suspend-wait loop. The technique simply requires one extra field in a lock structure, and one additional simple test in the uncontended path. It further suggests that, by applying this *tasuki conversion*, we may be able to transform single-atomic spin locking algorithms into spin-suspend locking algorithms without losing single-atomicity.

While we keep track of the number of the suspend-waiting threads in the `wcount` field, there is a variation which records just whether or not some thread is suspend-waiting. This variation only requires one bit rather than one field. However, the bit needs to be set inside the `while` loop and before the compare-and-swap, and that the `condvar_broadcast` needs to be called after the bit is reset in the `if` statement.

Considering that the `lock` field is only used to represent two states, it may be tempting to place the extra field or the extra bit into the unused portions of the `lock` field. However, the nonlockers' discipline states that we could no longer maintain single-atomicity.

```
int compare_and_swap_370(Word* word,Word *old,Word new){
 if (*word==*old){
   *word=new; return 1; /* succeed */
 } else {
   *old=*word; return 0;  /* fail */
 }
}
```

Figure 6: Semantics of the System/370-style of compare-and-swap

4 Multiprocessor Issues

The proof of Property 3.1 relies on the program-order execution of two pairs of memory operations; Line 8 and Line 9 in the `tasuki_unlock` function, and Line 15 and Line 16 in the `tasuki_suspend` function. However, as discussed in Section 2, the program-order execution is not necessarily guaranteed in modern multiprocessor systems. Thus, the implementation of tasuki lock on these systems requires a memory barrier to be inserted after the lock release at Line 8 and after the counter increment at Line 15.

In addition, the implementation requires a memory barrier to be inserted before the lock release for exactly the same reason as in the multiprocessor version of single-atomic spin locking. This results in two memory barriers being involved in the absence of contention.

In some multiprocessor systems, the execution of a memory barrier is very expensive. In such systems, we can no longer consider that a single-atomic algorithm with two memory barriers outperforms a double-atomic algorithm with one memory barrier in terms of the uncontended performance.

Here we show an interesting variation of tasuki lock for these systems, which requires only one memory barrier. The `lock` field can now take a third state, `UNLOCKING`, while the suspend-wait loop uses the original style of compare-and-swap as defined in IBM System/370 [11], which atomically performs the task represented by the pseudo-C code in Figure 6.[3]

Figure 7 shows the version of `tasuki_unlock` and `tasuki_suspend`; the other two functions are the same as before. The essential portions of the proof of Property 3.1 are now as follows.

> *Proof.* Assume that the thread T suspend-waits. Let t_1 be the time at which T performs the System/370-style of compare-and-swap that fails and causes T to suspend-wait. This implies that some thread S holds the lock at t_1. Let s_1 be the time at which the `UNLOCKING` value is visible to T. Obviously, t_1 before s_1. Because of the memory barriers, T increments the counter before t_1 and S performs the simple test of the counter after s_1. Thus, the test returns `true`. □

The `while` loop now contains spin-wait as well as suspend-wait. However, a thread T running in the loop spin-waits only while the write of `UNLOCKING` by the lock-holding thread is made visible to T, but the write of `UNLOCKED` is not yet made visible. On multiprocessor systems, this kind of spin-waiting is very preferable, since the lock will soon be released.

[3] The compare-and-swap was originally invented with this semantics in IBM System/370. It is available in many other architectures, such as IA-32, while it can also be implemented using the load-linked and store-conditional instructions.

```
void tasuki_unlock_smp(tasuki_t *tsk){
  tsk->lock = UNLOCKING;
  MEMORY_BARRIER();
  tsk->lock = UNLOCKED;

  if (tsk->wcount)
    tasuki_resume(tsk);
}

void tasuki_suspend_smp(tasuki_t *tsk){
  mutex_lock(&tsk->mutex);
  while (1){
    int unlocked=UNLOCKED;
    tsk->wcount++;
    MEMORY_BARRIER();
    if (compare_and_swap_370(&tsk->lock, &unlocked, LOCKED)){
      tsk->wcount--;
      break;
    } else if (unlocked == UNLOCKING){
      ; /* spin-wait */
    } else
      condvar_wait(&tsk->condvar, &tsk->mutex);
  }
  mutex_unlock(&tsk->mutex);
}
```

Figure 7: Variation of tasuki lock for multiprocessor systems

5 Preliminary Results

In this section, we evaluate an implementation of our locking algorithm, using a substantially large multithreaded program – a Java virtual machine [16]. Concretely, we use IBM Developer Kit and Runtime Environment for AIX, Java Technology Edition, Version 1.2.2 [10]. The virtual machine includes the platform dependent layer of synchronization facilities, called *system monitors*, which are basically used for two purposes.

First, *language-level monitors* are built on top of system monitors. Java's built-in support for multithreaded programming is based on monitors, where every Java object is (logically) associated with its own monitor and could be synchronized upon. Second, the *internal locks* inside the virtual machine are just system monitors. The virtual machine requires mutual exclusion for many different purposes, including thread management, heap management, class linking, and self-modifying bytecode, just to name a few. Each of these results in its own system monitor being allocated.

System monitors are then implemented upon a thread library provided by the underlying operating systems, the AIX Pthreads library in our virtual machine, where the mutex variables are realized using a double-atomic spin-suspend algorithm.

We implemented tasuki lock into the platform dependent layer of system monitors.[4] Starting with the algorithm as described in Figure 4 and 5, we added the support of recursive locking, and of long-term synchronization such as notifying and waiting on a system monitor.

[4]It might be preferable to directly modify the pthreads library, and implement the mutex variables with tasuki lock. However, we could not access the source code at the layer of the operating system.

Benchmarks	Method Sizes (in bytecode)	Execution Times (msec)		Improvements	
		Original	Tasuki	Absolute	Relative
`{ counter++}`	6 bytes	41797	28595	13202	31.6%
`StringBuffer.charAt`	27 bytes	43806	30528	13278	30.3%
`StringBuffer.getChars`	79 bytes	50820	35651	15169	28.8%

Table 2: Results of micro-benchmarks

We ran all the programs under AIX 4.3.3 on an unloaded RISC System/6000 Model 43P containing a 133-MHz PowerPC 604 with 128 megabytes of main memory. The JIT compiler was enabled for all of these measurements. We took a median of eleven runs for each benchmark.

It is worthwhile to note that IBM JDKes include significant optimizations of language-level monitors [4, 19], which allow most monitor operations at this level to be performed without involving system monitors. The language monitor falls to the backing system monitor only when it is being contented or being waited upon by some other thread. Thus, there have already been no low-hanging fruits for our optimization of system monitors.

5.1 Micro-benchmarks

Each micro-benchmark involves two Java threads. One thread is waiting on a Java object, while another thread calls a synchronized method against the object eight million times. Heavily depending on the particular implementation of language-level monitors in our virtual machine, the first thread effectively forces all the synchronizations by the second thread to fall back to the corresponding system monitor. In addition, no thread contends with the second thread. Thus, the benchmark exercises the uncontended path of entering and exiting from a system monitor, which is optimized by tasuki locking.

Three benchmarks call synchronized methods of different sizes; an one-expression method, `StringBuffer.charAt`, and `StringBuffer.getChars`. As Table 2 shows, our algorithm achieves improvements of around 30% in each benchmark. Furthermore, the larger the method invoked, the more the absolute improvement. Since each benchmark enters the system monitors the same number of times, this is an evidence that the single-atomic algorithm imposes fewer constraints upon hardware optimizations.

5.2 Macro-benchmark

We measured a business object benchmark for Java, called Portable BOB [5], which is expected to be included in the SPEC Server benchmark suite. The benchmark creates warehouses and client terminals, gets the terminals to generate transactions against warehouse data, and reports the throughput in transactions per second. We ran Portable BOB by varying the number of warehouses, w, from 1 to 10; the more warehouses, the more concurrency introduced. Figure 8 shows the results. As we see in the figure, the virtual machine with tasuki lock constantly outperforms the original in all the data points, with the maximal improvement of 3.37% at $w = 1$.

As we mentioned earlier, basically system monitors have not been a bottleneck in most Java applications. For instance, the frequency of system monitor entries are one or two orders of magnitude lower in the tests of the SPECjvm98 benchmark suite [22] than Portable BOB, so that the single atomic algorithm in this layer does not make as much difference for them. However, we think that server applications as represented by Portable BOB are gradually, though not at an amazing pace, making system monitors hotter. Furthermore, by applying the

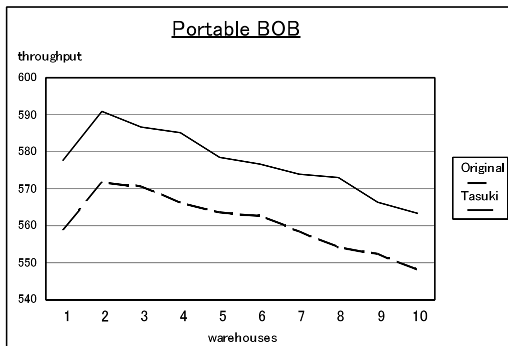

Figure 8: Performance of Portable BOB for various numbers of warehouses

single-atomic algorithm in the layer of the thread library, we could capture many more locking operations in our algorithm, not just from Java virtual machines but from other libraries and middleware systems, which we expect to result in more improvements.

6 Related Work

There are a significant number of papers and books on locks. Early work focuses on achieving mutual exclusion using atomic read and write operations, leading to numerous algorithms, including Dekker's [7], Peterson's [20], and Lamport's [15]. However, these quickly became obsolete with the prevailing hardware support of atomic read-modify-write operations.

Anderson [3] studied the performance of various algorithms for spin locking, and discussed optimizations such as spin on read and exponential backoff. Mellor-Crummey and Scott [18] 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.

Ousterhout [21] first suggested spin-suspend locking, leading to the subsequent work on the spinning strategy. Karlin, Li, Manasse, and Owicki [14] empirically studied seven spinning strategies based on the measured lock-waiting-time distributions and elapsed times, while Lim and Agarwal [17] derived static methods that attain or

approach optimal performance, using knowledge about likely wait-time characteristics of different synchronization types.

However, there is little literature which shows the full details of spin-suspend locking algorithms. Also, most commercial operating systems provide thread libraries to support multithreaded programming, and are said to use spin-suspend locking for implementing synchronization primitives such as critical sections and mutex variables. The details are not disclosed, either, but we could confirm that the algorithms used in the thread libraries of OS/2 and AIX are double-atomic.

Recently, Java [8] created renewed interest in locking algorithms, because of its prevailing use of monitors. Bacon, Konuru, Murthy, and Serrano [4] proposed a locking algorithm for Java, called *thin locks*, which reserves a 24-bit field in an object and makes bimodal use of the single field. 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. Thus, this is a spin-suspend algorithm which is very space efficient.

Thin locks are also single-atomic. However, it requires spin-wait in the spin-to-suspend mode transition (inflation), and does not support the suspend-to-spin transition (deflation). Onodera and Kawachiya [19] proposed another bimodal locking algorithm, which removes spin-wait from inflation and supports deflation. Keeping the algorithm single-atomic, they converted the spin-wait into a suspend-wait by introducing an extra bit in an object to indicate whether or not contention has occurred. Indeed, this bimodal algorithm for locking Java objects led us to consider a single-atomic spin-suspend algorithm in a general context.

7 Concluding Remarks

We presented a single-atomic spin-suspend locking algorithm, called tasuki lock. Although hybridizing spin locking with suspend locking is commonly practiced, the details of such algorithms are rarely published, and, as far as we know, all the existing spin-suspend algorithms are double-atomic.

Our algorithm centers on a generalized technique of tasuki conversion for turning a spin-wait loop into a suspend-wait loop. It uses an extra field which is separate from the lock field, and is tested in the uncontended path *without* any protection. Nonetheless, the timing dependency cleverly created upon the failing compare-and-swap guarantees that the algorithm does not lose an important liveness property.

We also described an intriguing variation for multiprocessor systems with relaxed memory models. With two-step lock release, tasuki lock is now able to manage without any additional memory barrier.

Finally, we evaluated an implementation of our single-atomic algorithm in the platform dependent layer of synchronization in the IBM Developer Kit 1.2.2 for AIX. The results have shown that the virtual machine with tasuki lock achieves improvements of around 30% in the micro-benchmarks, while it constantly outperforms the original in all the data points in a server benchmark. By applying the single-atomic algorithm directly to the layer of the thread library, more locking operations could be captured in the algorithm, which is expected to result in more improvements.

Acknowledgments

We thank Cathy May for giving us the benefit of her expertise in the memory model of PowerPC architecture. We also thank Kiyokuni Kawachiya, Anthony Cocchi, and Pat Gallop for invaluable discussions on tasuki lock for multiprocessor systems.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29(12), 1996, 66–76.

[2] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y.S. Ramakrishna, and Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. *OOPSLA'99 Conference Proceedings*, 1999, 207–222.

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

[4] 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, 258–268.

[5] Sandra Johnson Baylor, Murthy Devarakonda, Stephen J. Fink, Eugene Gluzberg, Michael Kalantar, Prakash Muttineni, Eric Barsness, Rajiv Arora, Robert Dimpsey, and Steven J. Munroe. Java Server Benchmarks. *IBM Systems Journal.* 39(1), 2000, 57–81.

[6] David E. Culler and Jaswinder Pal Singh with Anoop Gupta. *Parallel Computer Architecture.* Morgan Kaufmann, 1999.

[7] Edsger W. Dijkstra. *Cooperating Sequential Processes*, 43-112. Academic Press, New York, 1968.

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

[9] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach, Second Edition.* Morgan Kaufmann, 1995.

[10] IBM Corporation. *IBM Developer Kit for AIX, Java Technology Edition.* `http://www.ibm.com/java/jdk/aix/` (current April 10, 2000).

[11] IBM Corporation. *IBM System/370 Principles of Operation.* IBM Publication GA22-7000-9, 1983.

[12] IBM Microelectronics and Motorola. *PowerPC 604: RISC Microprocessor User's Manual.* MPR604UMU-01(IBM order number), 1994.

[13] IEEE. *Information Technology – Portable Operating System Interface (POSIX) – Part 1: System Application: Program Interface (API) [C Language].* ISO/IEC 9945-1:1996.

[14] 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, 41–55.

[15] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems* 5(1), 1987, 1–11.

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

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

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

[19] Tamiya Onodera and Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. *OOPSLA'99 Conference Proceedings*, 1999, 223–237.

[20] Gary L. Peterson. A New Solution to Lamport's Concurrent Programming Problem Using Small Shared Variables. *ACM Transactions on Programming Languages and Systems* 5(1), 1983, 56–65.

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

[22] Standard Performance Evaluation Corporation. *SPEC JVM98 Benchmarks.*
`http://www.spec.org/osg/jvm98/`