TO-Lock: Removing Lock Overhead Using the Owners' Temporal Locality

Takeshi Ogasawara
Tokyo Research Laboratory
IBM Japan
takeshi@jp.ibm.com

Hideaki Komatsu
Tokyo Research Laboratory
IBM Japan
komatsu@jp.ibm.com

Toshio Nakatani
Tokyo Research Laboratory
IBM Japan
nakatani@jp.ibm.com

Abstract

The performance of locking is critical, as programming languages with built-in thread support come into wide use. Many techniques for optimizing Java monitors have been proposed, based on the observation that the locks are rarely contended for in many applications. However, the problem of the performance degradation in SMP environments caused by necessary serializations of the processors' execution has not been addressed for shared objects.

We propose a new algorithm for this problem. It uses simple instructions to acquire the lock by exploiting the owner locality for objects even if the ownership has migrated among the threads. Our algorithm is particularly effective for SMP environments because we can remove the overhead of the serialization caused by complex atomic operations for uncontended locks by allowing the lock operation and the code protected by the lock to be executed in parallel. We verified the safety of the algorithm by using a software tool, Spin. The experimental results of our benchmarking on an SMP machine using Intel Xeon processors showed that our algorithm can significantly improve the performance by 83% on average compared to the case using a complex atomic instruction.

1. Introduction

Multithreaded programming is becoming popular as programming languages with built-in thread support are coming into wide use. By using these languages, programmers can easily exploit the thread-level parallelism of programs in the multiprocessor environments.

Synchronization among threads is necessary to protect shared resources. Because of the built-in thread support of the language, all library routines must assume multithreading and perform locks before accessing data that is modifiable by multiple threads. Thus the synchronization operation is pervasive in many Java applications.

The efficient implementation of locks is critical for Java programs that frequently perform lock operations. In fact, we observed that many benchmark programs need a lock for every 15 to 30 heap accesses. Prior research also shows that Java programs frequently use locks and the overhead of locks degrades the performance of the programs [6, 3, 26, 7, 21]. It also shows that for many applications the locks are rarely contended for among multiple threads. Therefore, as the most frequent case, uncontended locks should be supported as efficient as possible with minimal memory.

Bacon et al. [6] focused on this characteristic of rare contention and proposed a very fast lock, called the thin lock. A thin lock performs an atomic compare-and-swap operation on the lock word, which is a single word of memory space within each lock target (an object in Java). This operation is usually implemented by a special atomic instruction. For example, we can use `cmpxchg` on the IA-32 architecture [15] and the IA-64 architecture [17] and a pair of one load-linked instruction and one store-conditional instruction (`ll/sc`) on the POWER architecture [13]. Many lock algorithms, both those derived from thin locks [26, 10, 21] and others [3, 7], use such an atomic operation for the lock acquisition.

The problem with these lock algorithms is the latency of the lock operation caused by serializing the processor execution with the compare-and-swap operation in an SMP environment. In these algorithms, any operations that follow the lock acquisition have to wait until the atomic instruction actually stores the thread ID into the lock word and the thread ID is visible to the other processors. The simple instructions such as stores and loads are optimized in the modern processors. But the special atomic instructions are not as efficient as the simple instructions since they are complex and use some special hardware resources. The compare-and-swap operation has to wait for all of the in-flight operations including slow operations such as memory accesses to commit on the out-of-order execution processors. Therefore, for each lock acquisition the processor typically has to stall for many CPU cycles, during which it could exe-

©2004 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

cute tens or hundreds of simple instructions.

Kawachiya et al. [21] addressed this problem by introducing the idea of object reservation. They observed that if an object is first locked by a particular thread, then the object tends to be locked again by the same thread. For any reserved object, the thread does not require any compare-and-swap operation to lock the object. If a thread is not the reserving thread when it tries to lock an object, then the reservation of the object is cancelled and any subsequent lock for the object requires a compare-and-swap operation.

The biggest problem with this approach lies in the assumption that the majority of the lock requests for a given object come from a single thread. This is certainly effective for single-threaded programs or multithreaded programs having few or no *shared* resources. However, there are many real-world applications that tend to access some shared objects with multiple threads, and in such cases it acts in the same fashion as previous approaches that use compare-and-swap operations.

Considering the sequence of lock operations for a shared object, the compare-and-swap approach supposes that the owner of the object can change for every successive lock. However, the granularity of the code protected by the lock is not so fine in practice for most shared objects and the owners rarely change. In fact, we observed that, for more than 99% of the lock operations for the shared objects, the thread that performs the lock operation was the same as the thread that last released the object. We call this ratio *the owner locality*.

In this paper, we propose a novel locking algorithm, the *tentative-ownership lock* or *TO-lock*, which exploits the high owner locality. The algorithm does not require any expensive atomic instruction as long as the same thread locks an object. When a thread first locks an object, the thread indicates its ownership of the object by storing its thread ID. As long as that thread remains as the owner of the object, the lock is very lightweight, requiring only simple instructions (three memory loads, three tests, and two memory stores). These simple instructions do not stall the processor. When a thread does not own an object and attempts to acquire the lock for the object, it performs a compare-and-swap operation to claim ownership of the object.

The most useful feature of this algorithm is that the lock operation and the other non-lock operations can be executed independently in the uncontended case. If all of the three tests of the algorithm succeed for a lock operation, the success ensures that the ownership of the object has not been moved to other threads since the thread last released the object. Therefore, it is safe to continue executing the other non-lock operations in parallel. This is similar to the well-known optimization for the recursive lock, in which the same thread acquires the same lock again without releasing the lock. For the recursive lock, the actual lock operation using the special atomic instructions is usually skipped and the nesting level is maintained instead.

For mutual exclusion, we validated the algorithm by using a standard software tool, Spin [12]. Spin generates all the possible cases for an algorithm and checks the assertions at selected program points for each case. We also considered the pros and cons of real processors for implementing our algorithm on some different processor architectures.

In summary, the contributions of our paper are the following:

- We designed a lock operation using only simple instructions, which can be performed independently of other non-lock operations in most cases.
- We validated the algorithm by using Spin.
- We show the algorithm significantly improved the performance of the code performing lock operations by 83% on average for a real-world owner locality scenario on Intel Xeon processors.

The rest of this paper is organized as follows. We first show an empirical study of the lock behavior by using benchmark programs and real-world server applications. We then explain our algorithm. Next we describe the verification of our algorithm by using Spin and investigate the pros and cons of real processors for implementing our algorithm. Finally we discuss related work and conclude the paper.

2. A study of lock behavior

This section presents our analysis of lock behavior. We first consider the frequency of lock operations in Java programs to emphasize the importance of efficient lock operations. Next we show the owner locality for these programs.

We used SPECjvm98 [27], SPECjbb2000 [28], and VolanoMark version 2.5.0.9 [29] as well-known benchmark programs. As the real-world programs, we used IBM WebSphere Application Server [14] and used Trade3 and 22 benchmarks of the Web primitives for Web services [24] on the server, which simulates the typical behaviors of the server.

To avoid profiling the system during the warming-up and start-up time, we measured the fifth run for SPECjvm98 and the time measuring phase for SPECjbb2000. We measured VolanoMark, Trade3, and the Web primitives benchmarks after the applications started up. We configured SPECjbb2000 as running two warehouses. For Trade3 and the Web primitives, we configured the benchmark driver as 10 clients sending 1,000 messages. We ran these programs on a Windows XP machine, which has two physical processors. We modified the IBM Java Just-In-Time compiler to insert trace code that profiled every synchronization into the compiled code.

Figure 1. The ratio of lock operations to heap accesses (higher bars show more frequent locks)

2.1. Lock frequency

Since the lock operation synchronizes memory accesses and modern processors usually have memory-dedicated execution pipelines, we can regard the lock operations as stalling the memory pipelines. Therefore we measured the ratio of lock operations to memory accesses, in particular the heap accesses. These profiled heap accesses include accesses to instance variables, class variables, array elements[1], and the internal fields such as array sizes and tables for virtual methods [2].

Figure 1 shows this ratio for each benchmark. For some typical programs, a lock operation is performed for every 15 to 30 heap accesses. On average a lock operation is performed for every 55 heap accesses.

In addition, modern processors have multiple memory units, which can process multiple memory accesses in parallel. Due to these parallel executions of memory accesses, the ratio will double on processors having two memory units and quadruple for four memory units.

Therefore the performance of locking is critical for the programs that perform lock operations with such high frequencies, considering that a single compare-and-swap operation stalls many CPU cycles, during which modern processors could execute tens or hundreds of simple instructions [3].

1 Array elements copied by System.arraycopy are not included.

2 The compiler enables an optimization for instance variables, which caches instance variables in local variables to promote them to registers if possible. Therefore it was possible to optimize some heap accesses.

3 In our micro benchmarking, the `lock` prefix for compare-and-swap operations, which ensures the atomicity in SMP environments, adds more than 150 cycles on Intel Xeon processors. For a different architecture, IA-64, a document [19] shows that the VLIW pipeline consisting of 6 ALU's and 4 memory units is stalled for at least ten cycles by a compare-and-swap instruction on the Itanium2.

2.2. Owner locality

We focus on *shared* objects. *Shared* objects are the objects that are actually locked by more than one thread during the measurement. A *non-shared* object is an object that is locked by only a single thread. Non-shared objects can be non-escaping objects, which are visible only to one thread but not to the other threads. Since lock operations for these non-escaping objects could be removed by some other techniques using the knowledge of their non-escaping characteristic, for example by allocating these objects in a special thread-local heap [9], we do not further discuss them in this paper.

The first bars in Figure 2 show the ratio of the number of lock operations for shared objects relative to the total lock operations. The taller bars indicate that those threads most frequently perform lock operations for shared objects. The shorter bars show that those threads most frequently perform lock operations for non-shared objects.

The ratio of lock operations for shared objects ranges up to 92.6% and is 41.0% on average, excluding SPECjbb2000 and SPECjvm98. Since SPECjbb2000 and SPECjvm98 rarely perform lock operations for shared objects, these benchmarks are mostly single threaded.

The second bars in Figure 2 and the first bars in Figure 3 show the owner locality. The owner locality in Figure 2 is the ratio of the total number of lock operations, whose locks are acquired multiple times consecutively by the same thread, to that of the total lock operations. The owner locality in Figure 3 is the ratio of the same number to the count of lock operations for shared objects. The taller bars show that the owners for those objects rarely change during all of the lock operations.

We focus on the owner locality for shared objects in Figure 3 while Figure 2 shows the ratio to the *total* lock operations. In Figure 3, the owner locality is up to 86.4% and

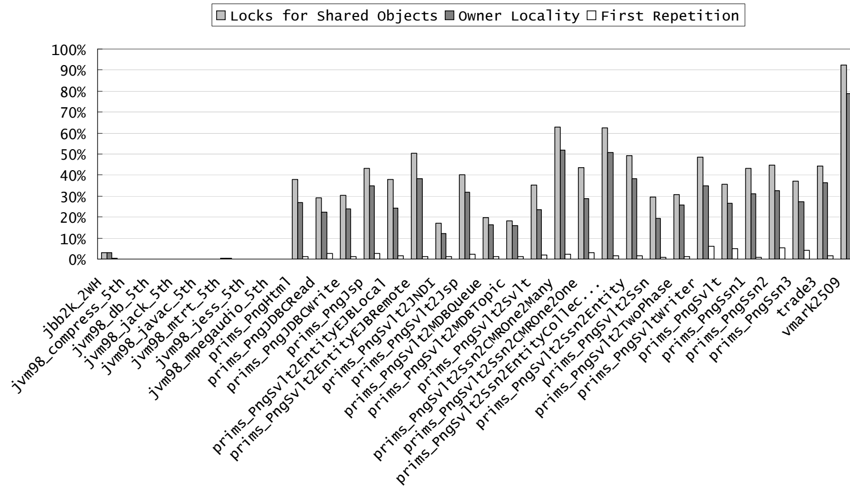

Figure 2. The owner locality for the total lock operations

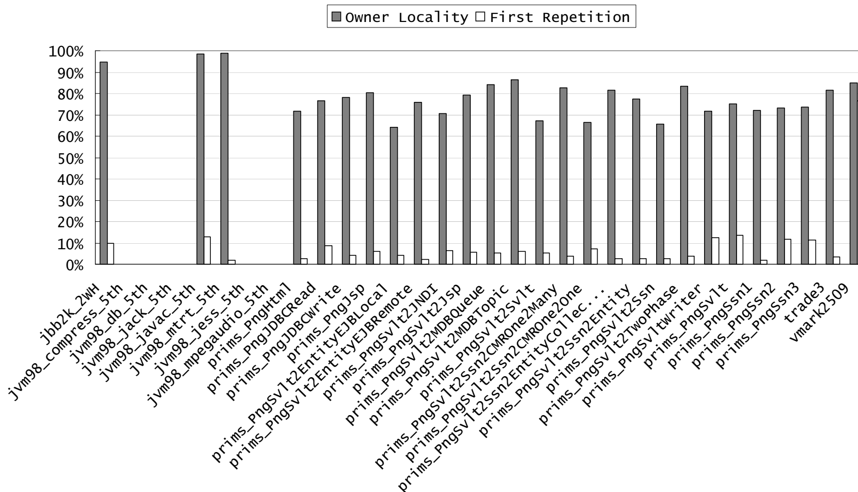

Figure 3. The owner locality for shared objects

76.0% on average of the numbers of lock operations for shared objects.

The third bars in Figure 2 and the second bars in Figure 3 show the ratios of the first repetitions [21]. The first repetition is a special sequence of lock operations for a given object, which starts from the first lock and continues until a different thread from the first locker actually locks the object. The taller bars show when the first locker continues locking without other threads' locks.

In Figure 3, we should interpret the result for VolanoMark carefully. In VolanoMark, the first repetition ratio is high, but this does not mean the first repetition ratio is high for most of the lock operations in the benchmark. On the contrary, a specific class is causing this result. The lock operations for objects of the class `java/io/BufferedOutputStream` dominate 70.0% of the lock operations for shared objects, and these particular objects are mostly non-shared, resulting in a first repetition ratio that reaches 99.99% for these objects.

The first repetition ratio is up to 13.5% and only 5.9% on average, excluding VolanoMark. In contrast, excluding VolanoMark does not affect the owner locality, which is still up to 86.4% and 75.6% on average. We ignored the high owner locality for some of the SPECjbb and SPECjvm98 benchmarks in Figure 3, since the ratio of lock operations for the shared objects is very low, as shown in Figure 2.

Thus the owner locality for shared objects is observed in real-world applications, even though the first repetition is rarely observed.

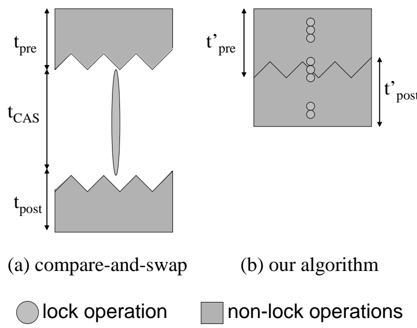

Figure 4. The improved critical path of the code

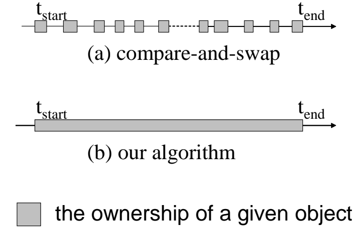

Figure 5. The extension of the ownership of an object

3. TO-lock

This section explains the algorithm of the *tentative ownership lock* or *TO-lock*. We first give an overview of the characteristics of our algorithm. Next we explain the variables used in the algorithm. We then cover the algorithm step by step.

3.1. Overview

As shown in the previous section, we observed that the objects shared by multiple threads are almost always synchronized successively by the same thread many times once the thread acquires their locks. We use this temporal locality of threads to remove the atomic operations from the most frequent case of the locking process.

Suppose that a lock algorithm can ensure that the ownership of a given object has not migrated since the thread claimed it instead of performing the compare-and-swap operation. In that case, we can skip the compare-and-swap operation as long as we can inhibit the migration of ownership.

More importantly, we may perform both such a lock algorithm and the other non-locking operations in parallel. This means that we can shorten the critical path of the code. In the semantics of the lock operations implemented by compare-and-swap, we may not perform the following non-lock operations, in particular the memory accesses, during the lock operation. Therefore, such a no-compare-and-swap lock algorithm can increase the parallelism of the machine instructions.

Figure 4 shows this increased parallelism from our approach. The X-axis shows the execution units that work in parallel and the Y-axis shows their progress on the time scale. The t_{pre}, t_{CAS}, and t_{post} times are the times for the critical paths of the code segments before, during, and after the compare-and-swap operation, respectively. Figure 4b shows that the critical path length is improved by performing the simple instructions for the lock operation and the other non-lock operations in parallel, compared to the case using the compare-and-swap in Figure 4a.

There are two key ideas in our algorithm to exploit the temporal locality of threads. The first one is the extension of the duration for which threads claim the ownership of objects, as shown in Figure 5. In Figure 5, t_{start} and t_{end} denote the start and end times of some sequence of lock operations for a given object. Since the same thread locks the object from t_{start} to t_{end}, there is temporal locality of the thread locks in this sequence. Our algorithm merges these occurrences of the ownership (Figure 5a) into a single occurrence by storing the thread ID (*tentative ownership*) into the object (Figure 5b).

The second idea is the use of *temporal locality tests*. During the period of extended ownership, threads only go through the temporal locality tests in most cases. Algorithm 1 shows how the temporal locality tests are performed. Since the temporal locality tests use only simple instructions, the most frequent case of the algorithm can be performed very quickly. The flags that are set between the tests ensure mutual exclusion while using

```
 1: if tid = myid then
 2:   tel_1 ⇐ 1
 3:   if tid = myid then
 4:     tel_2 ⇐ 1
 5:     if tid = myid then
 6:       {The thread is still within the critical section
          since no migration of the ownership has occurred.}
 7:     else
 8:       {The ownership has migrated.}
 9:     end if
10:   else
11:     {The ownership has migrated.}
12:   end if
13: else
14:   {The ownership has migrated.}
15: end if
```

Algorithm 1: The temporal locality tests

simple instructions. The small circles denote these instructions in Figure 4b. If all of these tests succeed, the thread has successfully extended the ownership of the object.

If there has been a break in the temporal locality of the thread's use of the object, the *thread shift* attempts to restore the locality. If any collisions occurred during the temporal locality tests, the *collision notification* and the *flag reset* reset the algorithm. These processes use atomic operations.

In the following subsections, we first explain the variables used by the algorithm and then describe the algorithm itself in detail.

3.2. Variables

The algorithm uses a variable, *tid*, and three 1-bit flags for each object.

3.2.1. Tid The tid field holds the identifier of the thread that currently has the lock or that last released the lock. We use this variable to keep track of the temporal locality of threads that acquire the lock for the object. The thin lock and its variants also require a tid. The difference between their tid and ours is whether or not a thread clears the tid when it releases the lock. In our algorithm, a thread does not clear the tid at the release of the lock. Therefore a thread sees its own thread ID in the tid as long as no other threads have locked the object since the last time. If a thread sees another thread's ID in the tid, it indicates that there has been a break in the temporal locality of the thread's use of the object.

During the locking process, the thread checks the tid several times. When a thread attempts to lock an object, it does not change the tid as long as the tid shows its own ID. If the tid shows another thread's ID, then the thread attempts to replace the tid with its own ID.

The size of this variable must be large enough to express the maximum number of threads. For example, the maximum number of threads is 32767 in IBM's 32-bit Java virtual machine [6].

3.2.2. Tel1, tel2, and fail There are three 1-bit flags, *tel1*, *tel2*, and *fail*. These flags are used to indicate the status of the object.

A pair consisting of a *tel1* flag and a *tel2* flag shows the progress of the locking process. These flags are set by the thread that started the locking process for the object. Once a thread sets both flags for an object, the lock acquisition will succeed for that thread, and other threads cannot acquire the lock for the object. A thread first sets *tel1* and then *tel2*. Before setting each, the thread checks if certain conditions have been met. Therefore, the threads can observe which step of the locking process has been reached for a given object by using these flags. These flags are cleared by the thread that acquired the lock when that thread releases the object.

The *fail* flag is used when there is a collision and more than one thread is attempting to lock the same object. In that situation, any thread that detects the situation will attempt to set the *fail* flag.

3.3. Locking algorithm

This section explains the algorithm. Figure 6 shows the flow chart of the algorithm. The algorithm consists of five components: (1) *temporal locality tests*, (2) *collision notification*, (3) *thread shift*, (4) *flag reset*, and (5) *phase-2 check*. We describe each of these steps in detail in the following subsections. In Figure 6, a diamond enclosed within a rectangle denotes an atomic operation.

3.3.1. Temporal locality test - phase 1 First the thread checks if the tid is still the same as its own ID. If the test succeeds, the thread sets the *tel1* flag to inhibit a *thread shift*, an attempt by some other thread to change the tid. If the tid does not match, the thread proceeds to the step for thread shift. The details of the thread shift are explained in Section 3.3.5.

3.3.2. Temporal locality test - phase 2 The thread checks again to make sure the tid is still the same as its own ID. This test is required to make sure a thread shift did not occur since the load of the tid through the store that sets *tel1* in Phase 1. If the test succeeds, the thread sets the *tel2* flag to inhibit *flag reset*. If the tid does not match, the thread proceeds to the step for *collision notification*. The details of collision notification are explained in Section 3.3.4.

3.3.3. Temporal locality test - phase 3 The thread checks again to make sure the tid is still the same as its own ID. This test is required to make sure that no other threads have succeeded in a flag reset since the second load of tid until the store that sets *tel2* in Phase 2. If the test succeeds, the lock acquisition has succeeded. Otherwise, the thread proceeds to the step for thread shift.

As the thread completes Phases 1 through 3, it prevents other threads from proceeding through these same phases. When Phase 3 has succeeded, the *tel1* and *tel2* flags are set and the *tid* matches its own ID. This status remains until the object is released.

3.3.4. Collision notification If the thread enters this step, it has already set the *tel1* flag in Phase 1 but failed to proceed to Phase 3. Therefore the *tel1* flag should be cleared before aborting the current locking process. However, the thread cannot simply clear the *tel1* flag in this step.

This scenario analysis shows why the thread cannot simply clear *tel1*. Assume that there are three threads that attempt to acquire the lock for the same object. Before the

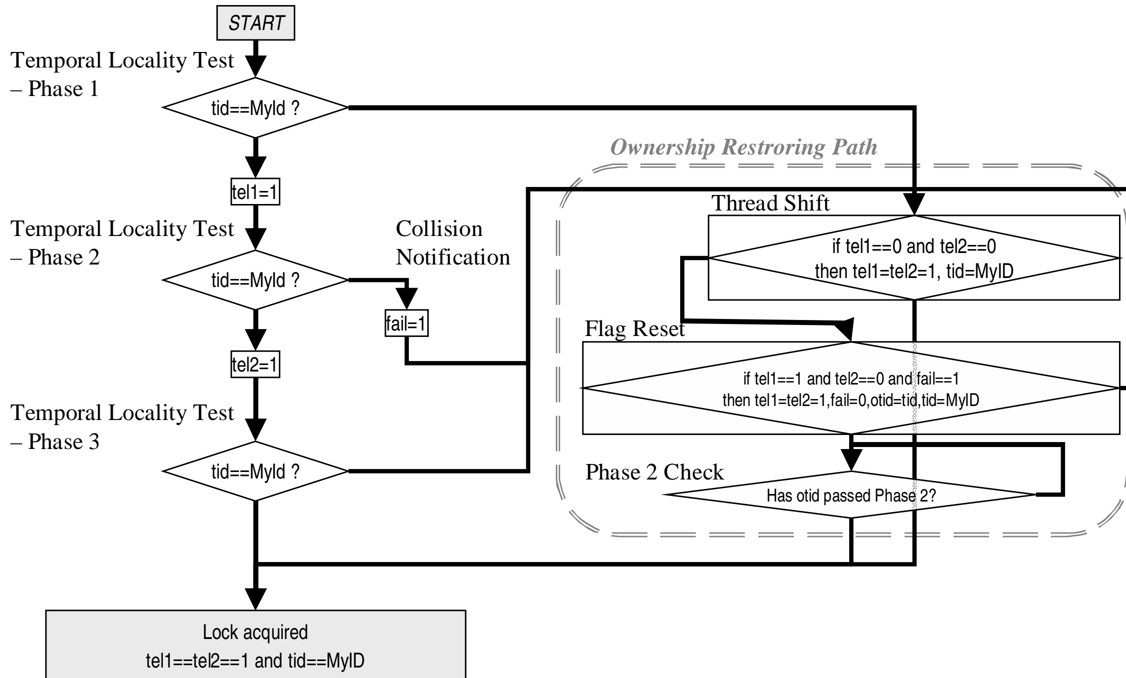

Figure 6. The flow chart of the TO-lock algorithm.

first thread sets *tel1* in Phase 1, the second thread changes the tid for a thread shift and enters Phase 3 but has not yet set *tel2*. Meanwhile, the third thread initiates a thread shift. The second thread will detect the thread shift and proceed to collision notification. However, if the second thread clears *tel1*, the third thread will see that the *tel1* flag inhibiting the thread shift is not set. Therefore we cannot clear *tel1* at this point.

On the other hand, the thread cannot abandon *tel1*, which has already been set. Assume that there are two threads that attempt to acquire the lock for the same object. Before the first thread sets *tel1* in Phase 1, the second thread changes the tid for a thread shift, completes the locking process, and releases the object. As explained in Section 3.2.2, the second thread clears the *tel1* and *tel2* flags at the release of the object. In this case, after the second thread clears the *tel1* flag, the first thread sets the *tel1* flag. If the second thread is no longer locking the object and the *tel1* flag is not cleared, the *tel1* flag will continue to inhibit thread shift, and no threads can acquire the lock.

The *fail* flag is used to create a special status for the object, in which the *tel1* and *fail* flags have been set but *tel2* is not set. This special status is used to safely reset the uncleared *tel1*, which inhibits any thread shift, as explained in Section 3.3.5.

The step of collision notification simply sets the *fail* flag and then proceeds to the step of thread shift.

3.3.5. Ownership restoring path If the thread proceeds to this path, there has been a break in the temporal locality of the thread's use of the object. The thread tries to reclaim the ownership of the object. Since we currently cannot find any single atomic instruction that can reclaim the ownership, this path consists of two components: *thread shift* and *flag reset*.

Thread shift This step restores the temporal locality of threads. The thread attempts to put its own thread ID into the tid. It can change the tid if and only if the *tel1* and *tel2* flags are cleared. As explained in Section 3.2.2, the cleared *tel1* and *tel2* flags show that the object is not currently locked by any thread.

Phase 1 assumes that thread shift does not modify the tid after it sets the *tel1* flag. To ensure that the thread changes the tid only if the *tel1* and *tel2* flags are cleared, it uses a compare-and-swap atomic operation. This restores the ownership of the object and acquires the lock if the atomic operation succeeds.

If more than one thread performs a thread shift at the same time, only a single thread succeeds and the other threads fail, similar to other implementations of compare-and-swap such as `lock cmpxchg` on the IA32 architecture [15] and `ll/sc` on the POWER architecture [13].

Flag reset Thread shift can fail because of the *fail* flag as well as because of contended thread shift. If the *fail* flag is set during the step of collision notification, *tel1* is always set. If the *tel1* flag is set, thread shift cannot change the tid.

This step re-initializes the flags. To ensure that the thread clears fail and *tel1* only if both flags are set but *tel2* is not set, it uses a compare-and-swap atomic operation. If the atomic operation succeeds, the thread reclaims the ownership of the object and acquires the lock. Otherwise the thread restarts the ownership-restoring path.

The algorithm does not allow for a situation in which *tel2* is set but *tel1* is not. In Figure 6, this situation is avoided by using the *phase-2 check*, which follows the step of flag reset. If some thread succeeds in a flag reset, the phase-2 check ensures that no other thread is currently in the middle of Phase 2 for the same object. Though such a situation could potentially occur, it will be quite rare, since our algorithm should be used for rarely contended locks and the situation can occur only if two lock operations are interrupted by each other during a very short period for the same object. A simple implementation of this check without extra overhead on the temporal locality tests would involve the inspection of the thread context of the interrupted thread associated with the program counter. If the lock is frequently contended for, instead of the phase-2 check, we may inflate the lock as in the thin lock. An example of how this situation occurs is as follows. First, thread A interrupts thread B that completed Phase 1 and acquired the object. Thread B sets *fail* and proceeds to the step of flag reset. Before thread B enters the flag reset stage, thread A releases the object and then proceeds to Phase 2 to acquire the object again, but does not yet set *tel2*. Without the phase-2 check, thread B can complete the flag reset and can clear *tel1* and *tel2* to release the object, and then thread A sets *tel2*.

3.3.6. Releasing algorithm The thread just clears *tel1* and *tel2* to release the object. By arranging these flags within a single memory word, most processors can clear them by a simple store operation.

3.4. Memory ordering

For the temporal locality tests, the stores to the *tel1* and *tel2* flags and the succeeding load of *tid* are ordered. However, since this memory ordering for the tests is independent of other non-locking operations, no further memory ordering is required between the tests and the non-locking operations.

When releasing the object, we ensure *release ordering semantics* [16] for clearing the two flags to release the object. Clearing the flags using the release ordering semantics ensures that any update to the memory during locking the object becomes visible to the other processors before releasing the object. However, there are no additional operations to ensure the ordering semantics on processors that do not change the memory ordering of store operations, such as the x86-based architectures including Intel IA-32 and AMD Athlon.

3.5. Verification

Here we discuss the safety and liveness of our algorithm. To verify these properties, we used Spin, which is a software tool that checks the logical consistency of a specification in distributed systems design [12]. We used Spin as an exhaustive verifier, which rigorously proves the validity of the user-specified correctness requirements.

We show the source code for Spin in Appendix A, which is a direct translation of the algorithm shown in Figure 6. There are threads that perform the algorithm in an infinite loop in the source code. Spin generates all of the possible traces of the executions of these threads.

3.5.1. Safety We check three safety properties for our algorithm, mutual exclusion, avoiding illegal states of the variables, and avoiding deadlocks.

Mutual exclusion For mutual exclusion, we must prove that any given threads i and j never enter the critical section at the same time.

We confirmed that only one thread will always successfully acquires the lock with consistent values for the variables by using the *assertion* feature of Spin. We tested the following assertions at the program point where a thread acquires the lock: (1) the tid is the thread's own ID, (2) the *tel1* and *tel2* flags are set, and (3) only a single thread acquired the lock.

Illegal state of variables The algorithm does not allow for a situation in which *tel2* is set but *tel1* is not set, as explained in Section 3.3.5.

We confirmed that this situation never occurs by using the *linear time temporal logic formulae* feature of Spin[4]. Using this feature, we can verify that a correctness requirement holds at every step of all of the traces.

Deadlocks The algorithm has no deadlocks, in which threads do not execute any steps of the algorithm, because it has no paths that block threads.

3.6. Liveness

We confirmed a liveness property: if threads are trying to acquire a lock, one of them eventually acquires the lock. We used the linear time temporal logic formulae feature of Spin.

We also performed another experiment to verify that each thread that is going to acquire the lock eventually acquires the lock. For this experiment, we examined the number of lock acquisitions for each thread[5]. We confirmed that every thread acquired the lock for the specified number of

---

4 This feature is not included in the source code.

5 We limited the number of lock operations by each thread in the source code in Appendix A.

times. A compare-and-swap operation in the ownership-restoring path of our algorithm ensures that these threads can change the ownership as in other compare-and-swap-based algorithms [6, 26, 10, 21].

4. Processor-specific considerations

This section considers implementations of our algorithm on existing processors and shows experimental results by using a prototype.

4.1. Store forwarding avoidance

As explained in Section 3.4, our algorithm does not rely on the memory ordering between the lock operation and the non-locking operations, though our algorithm does rely on the memory ordering within the lock operation.

The existing memory ordering models, such as *acquire* and *release*, do not support the memory ordering that the algorithm requests, or the ordering of a specific store and a specific load. Therefore we require some hardware support for efficient implementation of the algorithm on existing processor architectures.

We use a special technique, *store forwarding avoidance (SFA)*, to control the memory ordering for our algorithm. For most processors, if a store to a memory area and a load from the same area appears in the code in that order, these operations are performed in that order even under relaxed memory consistency models. Furthermore, the store-to-load forwarding feature [20] makes it possible to ensure that these store-then-load operations are performed before the stored data becomes visible to other processors. However, this store forwarding is inhibited if the data size of the load is larger than the size of the store on some processors [18]. In that case, the store becomes visible and then the data is loaded from the memory hierarchy. We rely on this characteristic to ensure the memory ordering expected by the scalable synchronization algorithm in the relaxed consistency model. For in-order-execution processors such as IA-64, the compilers carefully schedule the instructions using the SFA to avoid unnecessary stall cycles. For out-of-order execution processors[6], the reorder buffer of the CPU will schedule independent instructions.

4.2. Experimental results

We prototyped the algorithm using hand-optimized IA-32 assembly code. The program executes in a loop containing a critical section and performs typical numbers of memory operations within and outside of the critical section. The memory operations are repetitions of a *simple sequence*: four loads and a store of their total. We decided on this ratio between the loads and stores based on the observation that loads are about 80% of the total memory operations in the server programs analyzed in Section 2. We ran our experiments on an SMP machine, an IBM IntelliStation Z-Pro with two physical Intel Xeon 3.06 GHz processors.

6 Compilers should schedule these instructions to avoid failing speculative loads on Intel's NetBurst-based processors such as Xeon since these processors have the overhead of replays for failed speculative loads.

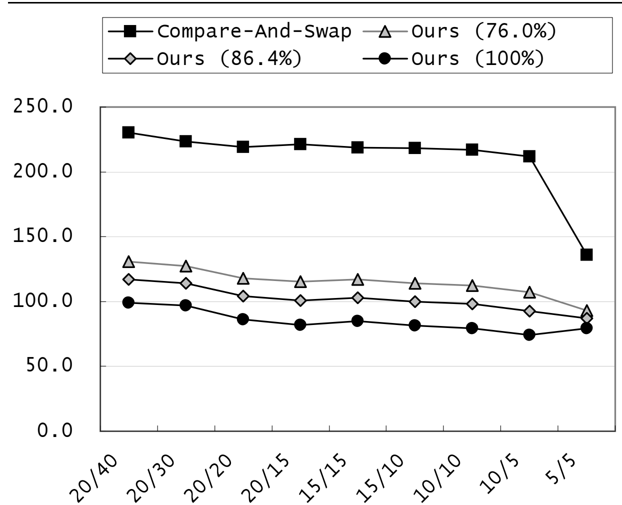

Figure 7. The execution times per iteration of the loops of simple sequences measured in CPU cycles (fewer is better)

We performed two types of experiments, using different workloads within the critical section. In the first experiment, there were only simple sequences. In the second experiment, we used Java's `Random.nextInt()` method, which has a longer critical path of computation compared to the first experiment. In both experiments, we measured execution times for a compare-and-swap-based algorithm [6] and ours, changing the number of memory operations from 5 to 40 (or one to eight simple sequences).

Figure 7 shows the results of the first experiment. The M/N labels for the X-axis denote that the number of memory operations within the critical section is M and N is for those outside of the critical section. As shown in Section 2, the owner locality ranges up to 86.4% and is 76.0% on average. Therefore, we plotted the estimated execution times for the cases that our algorithm is performed for 86.4% and 76.0% of the total locks and the compare-and-swap-based algorithm is performed for the remaining locks.

Our algorithm significantly improved the performance. We achieved 2.49 times speedup on average if we use our algorithm for all of the lock operations (100% utilization of our algorithm). For a real-world owner locality scenario,

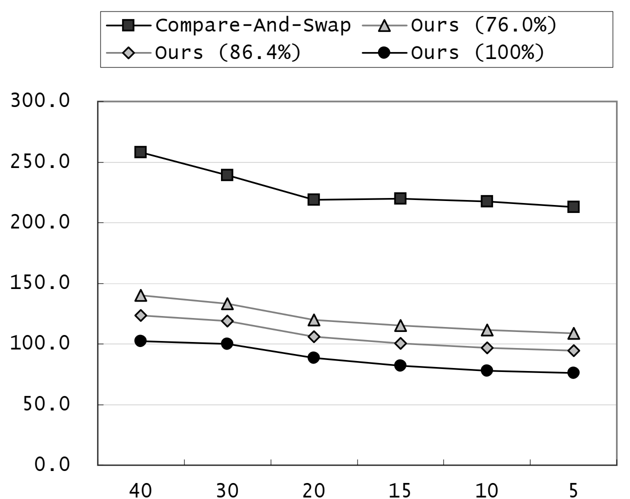

Figure 8. The execution times per iteration of the loops of `Random.nextInt()` in CPU cycles (fewer is better)

we achieved 2.07 times speedup for 86.4% owner locality (86.4% utilization of our algorithm) on average. For another real-world owner locality scenario, we achieved 1.83 times speedup for 76.0% owner locality on average.

Figure 8 shows the results using `Random.nextInt()`. The X-axis shows the number of memory operations outside of the critical section. Our algorithm also significantly improved the performance of this method. We achieved 2.61 times speedup on average for 100% utilization of our algorithm. For real-world owner locality scenarios, we achieved 2.14 times speedup for 86.4% utilization and 1.88 times speedup for 76.0% utilization on average.

5. Related work

Many techniques for reducing the overhead of synchronization have been proposed for Java. Bacon et al. [6] proposed the *thin lock*, which requires only one atomic operation for acquiring and releasing the lock. In contrast to other techniques [30, 22, 25, 3, 7], the thin lock does not use any multi-word monitor structure in most cases. However, the thin lock has a possibility of unbounded busy-waiting in the worst case. Onodera et al. [26] proposed the *Tasuki lock* to fix this problem, which avoids busy-waiting by using the *flc (flat lock contention)* bit, while still preserving the benefit of only a single atomic operation for each synchronization. For the techniques that always use a multi-word monitor structure, Agesen et al. [3] proposed the *meta lock*. However, the meta lock requires an extra atomic operation at the release of the lock and therefore performs two atomic operations for each synchronization operation. Dice [7] also proposed the *relaxed lock*, which performs only one atomic operation. These techniques require one or two atomic operations in their best cases.

Addressing the problem of the cost for the complex atomic operations in SMP environments, Kawachiya et al. [21] proposed *lock reservation*. Bacon et al. [5] independently proposed a similar idea of eliminating atomic operations for non-shared objects. Their techniques are effective for non-escaping objects, which are not shared among multiple threads. Domani et al. [9] also presented an idea for *global bits* that indicate whether or not objects are escaped.

In comparison to lock reservation, our approach has the following advantages to lock shared objects; every thread benefits from the efficient lock using the temporal locality and a non-owner can efficiently change the ownership by a compare-and-swap operation. Lock reservation must suspend and resume a thread whenever a non-owner tries to safely change the status of a lock reserved by another thread. Therefore, if lock reservation were extended to change the owners, both the new owner and the previous owner would have to pay penalties: the new owner suffers from the cost of the suspend and resume operations, and the previous owner suffers from the overhead of being suspended and being resumed.

The cost of the suspend and resume operations is usually large, since a program has to switch to the kernel mode to change the status of a thread. In the same environment as in the experiments of this paper, each suspend-and-resume operation takes more than 10,000 cycles on average. Therefore, lock reservation requires more than 2,400 cycles for 76.0% owner locality and 1,360 cycles for 86.4% owner locality, just for the suspend and resume operations, while our approach can complete each benchmark execution including a lock operation within less than 150 cycles. Also lock reservation requires additional cycles if we include the overhead of switching a thread for the previous owner that was suspended and resumed. Thus, the lock reservation approach is much slower than ours.

In the absence of contention for the lock, Lamport [23] presented *a fast mutual exclusion algorithm*, which has constant time complexity using only simple instructions. Lamport's algorithm performs five and two memory accesses for entering and leaving the critical sections, respectively. However the code within the critical sections cannot start executing until the third memory access is completed, since the memory access has *acquire semantics*[16] for the critical section. Regarding memory efficiency, Lamport's method requires memory space[7] proportional to the number of threads. It also uses two variables that can have process IDs for each lock. In contrast, our algorithm allows

7 A Boolean variable $b[i]$ used in Algorithm 2 [23] usually consumes one byte in actual processors due to the atomicity of stores.

the lock algorithm and the critical section code to be executed in parallel and requires only three bits and the thread ID field for each lock (or object). Andersen et al. [4] survey the algorithms for shared-memory mutual exclusion in the distributed algorithm community, including fast mutual exclusion algorithms, which have a constant-time fast path in the absence of contention. To the best of our knowledge, there has been no research for fast mutual exclusion algorithms that not only use a fixed number of simple instructions but also allow the execution of the critical sections in parallel while only requiring a fixed-size memory space.

6. Conclusion

This paper has presented a novel algorithm for optimizing lock operations, particularly useful for Java programs that frequently perform lock operations for shared objects. It uses only simple memory operations in the absence of contention, even if these objects are actually shared and locked by multiple threads. By exploiting the owner locality, the lock operation and the code protected by the lock can be executed in parallel so that the protected code no longer needs to wait for the relatively long latency of the lock operation. We experimented with many Java programs to profile the lock behaviors. The results show that, for real-world server-side programs, objects are shared among multiple threads and owner locality is honored. We have verified the safety of the algorithm by using a software tool, Spin, and considered combining our algorithm with existing approaches to balance the overheads for spinning and blocking.

We also have evaluated the efficiency of our algorithm with a complex atomic instruction by experimenting with assembler programs that perform typical numbers of memory operations within and outside of the critical section. Experimental results show that our algorithm can improve the performance of the code by 83% on average for a real-world owner locality scenario on the Intel Xeon processors, compared to the existing approach using the atomic compare-and-swap instruction.

Acknowledgements

We would like to thank Akira Koseki for his useful comments on our algorithm and for guiding us towards the verification of one of the safety properties. Thanks to Kiyokuni Kawachiya and Tamiya Onodera for their feedback and encouragement, and to Toshio Suganuma for many suggestions to improve the presentation of the paper. Finally, we wish to thank the anonymous reviewers for their constructive suggestions on how to improve the paper.

A. Source code

```
bit tel1, tel2, fail;
byte thread_id; /* thread ID ranges 0 to 255 */

int n_entered;
bit in_p3[3]; /*@only for SPIN@ emulates phase-2 check */

#define n_proc 2  /* number of processes try locks */

proctype lock(short myid) {
  byte r_tid;

L_start:
  r_tid=thread_id;
L_test_phase1:
  if
  :: (r_tid==myid) ->
    in_p3[myid]=1; /*@only for SPIN@  emulates phase-2 check */
    tel1=1;
L_test_phase2:  /* thread shift is inhibited */
    r_tid=thread_id;
    /* check if thread shift occurred during phase1 */
    if
    :: (r_tid==myid) ->
      /* no thread shift occurred during phase1 */
      tel2=1;
      in_p3[myid]=0;  /*@only for SPIN@ emulates phase-2 check */
L_test_phase3:    /* fail & flag reset inhibited */
      r_tid=thread_id;
      /* check if collision & flag reset occurred during phase2 */
      if
      :: (r_tid==myid) -> goto L_acquired
      :: (r_tid !=myid) -> goto L_ownership_recovering
      fi
    :: (r_tid !=myid) ->
      in_p3[myid]=0; /*@only for SPIN@ emulates phase-2 check */
      /* thread shift occurred during phase1 */
L_collision_notification: fail=1; goto L_ownership_recovering
    fi
  :: (r_tid !=myid) ->
L_ownership_recovering:
L_thread_shift: atomic {  if
      :: (tel1==0 && tel2==0) -> tel1=1; tel2=1; thread_id=myid;
                            goto L_acquired
      :: (tel1==1   ||   tel2==1) -> goto L_flag_reset
      fi
    }
L_flag_reset: atomic {  if
      :: (tel1==1 && tel2==0 && fail==1) ->
          tel2=1; fail=0; r_tid=thread_id; thread_id=myid;
          goto L_phase2_check
      :: (tel1==0   ||   tel2==1   ||   fail==0) ->
          goto L_ownership_recovering
      fi
    }
L_phase2_check: do :: (in_p3[r_tid]==0) -> break od; goto L_acquired
  fi;

L_acquired:
  /* TO-lock has been acquired */
  atomic {
  assert (thread_id==myid);
  assert (tel1==1);
  assert (tel2==1);
  assert (n_entered==0);
  }

  /* body */
  atomic { n_entered++ }
  atomic { n_entered-- }

  /* release */
  /* fail=0; */
  atomic { tel1=0; tel2=0 }
  goto L_start;
}

init {
  int i;
  /* Variables are initialized to zero in Promela:
    tel1=0;
```

```
    tel2=0;
    fail=0;
    thread_id=0;
    n_entered=0;
  */
  atomic {
    i=1;
    do
    :: (i <= n_proc) -> run lock(i); i++
    :: (i > n_proc) -> break
    od;
  }
}
```

References

[1] *Proceedings of the Java Virtual Machine Research and Technology Symposium*. USENIX Association, Apr. 2001.

[2] *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-1999)*, New York, NY, USA, 1999. ACM Press.

[3] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In ACM [2], pages 207–222.

[4] J. H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, special issue celebrating the 20th anniversary of PODC, To appear.

[5] D. F. Bacon and S. J. Fink. Method to provide concurrency control over objects without atomic operations on non-shared objects. U.S. patent, Aug. 2000.

[6] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *ACM SIGPLAN '98 Conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM Press.

[7] D. Dice. Implementing fast Java monitors with relaxed-locks. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* [1], pages 79–90.

[8] R. Dimpsey, R. Arora, and K. Kuiper. Java server benchmarks. *IBM Syst. J.*, 39(1):151–174, 2000.

[9] T. Domani, G. Goldstein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *The International Symposium on Memory Management (ISMM 2002)*, pages 183–194, New York, NY, USA, 2002. ACM Press.

[10] E. M. Gagnon and L. J. Hendren. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the Java Virtual Machine Research and Technology Symposium* [1], pages 27–39.

[11] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, Massachusetts, Aug. 1996.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[13] IBM. *The PowerPC Architecture, Second Edition*. Morgan Kaufmann Publishers, San Francisco, 1994.

[14] IBM. WebSphere Application Server, Version 5. http://www7b.boulder.ibm.com/wsdd/downloads/WASsupport.html, 2003.

[15] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual, Volume 3: System Programming Guide*. Intel Corporation, Mt. Prospect, IL, 2001.

[16] Intel Corporation. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*. Intel Corporation, Mt. Prospect, IL, 2002.

[17] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture*. Intel Corporation, Mt. Prospect, IL, 2002.

[18] Intel Corporation. *Intel Pentium 4 and Xeon Processor Optimization Reference Manual*. Intel Corporation, Mt. Prospect, IL, 2002.

[19] Intel Corporation. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*. Intel Corporation, Mt. Prospect, IL, 2003.

[20] M. Johnson. *Superscalar Microprocessor Design*. Prentice Hall Series in Innovative Technology. Prentice Hall, Englewood Cliffs, NJ, Jan. 1991.

[21] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-2002)*, pages 130–141, New York, NY, USA, 2002. ACM Press.

[22] A. Krall and M. Probst. Monitors and exceptions: How to implement Java efficiently. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, pages 15–24, New York, NY, USA, 1998. ACM Press. Also published as *Concurrency: Practice and Experience*, **10**(11–13), September 1998, CODEN CPEXEI, ISSN 1040-3108.

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

[24] T. Lau and Y. An. Running WebSphere Benchmark Sample Trade3 with Web Performance Tool (WPT). http://www7b.boulder.ibm.com/dmdd/library/techarticle/0303lau/0303lau.html, Mar. 2003.

[25] T. Onodera. A simple and space-efficient monitor optimization for Java. Technical Report RT0259, IBM Research Report, 1998.

[26] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In ACM [2], pages 223–227.

[27] The Standard Performance Evaluation Corporation (SPEC). JVM Client98 (SPECjvm98). http://www.spec.org/osg/jvm98/, 1998.

[28] The Standard Performance Evaluation Corporation (SPEC). Java Business Benchmark (SPECjbb2000). http://www.spec.org/osg/jbb2000/, 2000.

[29] Volano LLC. VolanoMark version 2.5.0.9. http://www.volano.com/benchmarks.html, 2003.

[30] F. Yellin and T. Lind. Java Business Benchmark (SPECjbb2000). http://www.spec.org/osg/jbb2000/, 2000.