Regular Paper

Efficient Lock Algorithm for Shared Objects in SMP Environments

Takeshi Ogasawara,[†,††] Hideaki Komatsu,[†]
and Toshio Nakatani[†]

We propose a new algorithm that is effective for objects that are shared among threads but are not contended for in SMP environments. We can remove the overhead of the serialization between lock and other non-lock operations and avoid the latency of complex atomic operations in most cases. We established the safety of the algorithm by using a software tool called Spin. The experimental results from our benchmarking on an SMP machine using Intel Xeon processors revealed that our algorithm could significantly improve efficiency by 80% on average compared to using 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 multiprocessor environments.

Synchronization among threads is necessary to protect shared resources. All library routines must assume multithreading and perform locks before accessing data that can be modified by multiple threads. Synchronization is thus pervasive in many Java applications.

It is critical for Java programs to efficiently implement locks because they frequently perform lock operations. We observed that many benchmark programs need a lock for every 15 to 30 heap accesses.

There are two problems with prior research that used compare-and-swap operations to acquire locks [1)~5)] in an SMP environment. Because of these, the processor typically has to stall for many CPU cycles for each lock acquisition, during which it could execute tens or hundreds of simple instructions.

The first problem is the need to serialize processor execution. Any operations in these algorithms that follow lock acquisition have to wait until the acquisition has been completed to read consistent data. This partly counters the advantage of out-of-order execution or relaxed memory consistency models, which can hide the latency of loads. The second problem is the inefficiency of special atomic instructions. Simple instructions such as stores and loads are optimized in modern processors. However, the special atomic instructions are not as efficient as simple instructions, since they are complex and use various special hardware resources.

Kawachiya, et al. [5)] solved these problems by introducing the idea of object reservation. They observed that if an object is first locked by a particular thread, the object then tends to be locked again by the same thread. For any reserved object, that thread can avoid the cost of serialization and special atomic instructions. 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 lock operation.

The biggest problem with this approach lies in the assumption that the majority of lock requests for a given object originate 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 a number of shared objects with multiple threads, and in such cases this degrades to the same level of performance as that of the previous approaches using compare-and-swap operations.

Considering the sequence of lock operations for a shared object, the compare-and-swap approach presumes that the owner of the object can change for every successive lock. However, the granularity of the code protected by the lock is not especially fine in practice for most shared

† IBM Tokyo Research Laboratory
†† Applied Computer Science Course, Interfaculty Initiative in Information Studies, Graduate School of Interdisciplinary Information Studies, The University of Tokyo

A preliminary version of this paper appeared in the Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques. (PACT 2004).

objects and the owners rarely change. In fact, we observed that for 76% of the lock operations for shared objects, the thread that performed lock operation was the same as the thread that last released the object. We called this ratio *owner locality*.

We propose a novel locking algorithm in this paper, called the *tentative-ownership lock* or *TO-lock*, which exploits the high owner locality. The algorithm still *tentatively owns* the lock when it releases it. It does not incur the cost of serialization or expensive atomic instructions that prior research required as long as the same thread locks an object. When a thread does not own an object and attempts to acquire the lock for the object, it performs compare-and-swap operation to claim ownership of the object.

The most useful feature of this algorithm is that lock and other non-lock operations can be independently executed in uncontended cases. This is similar to the well known optimization for recursive locks, in which the same thread acquires the same lock again without releasing it. The actual lock operation for the recursive lock using special atomic instructions is usually skipped and the nesting level is maintained instead.

We validated the algorithm to ensure its safety and liveness properties by using a standard software tool called Spin, which generates all the possible cases for an algorithm and checks the assertions at selected program points for each. We will also explain a special technique to implement our algorithm on real processors. The existing memory ordering models did not support our requirements.

In summary, the contributions of our paper are as follows:

- 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 demonstrated that the algorithm significantly improved the performance of code performing lock operations by 80% on average for a real-world owner-locality scenario on Intel Xeon processors.

The rest of this paper is organized as follows. We first discuss related work. We then present an empirical study of lock behavior by using benchmark programs and real-world server applications. We next explain our algorithm. We then describe how we verified our algorithm by using Spin. Finally, we investigate the pros and cons of real processors to implement our algorithm and evaluate it.

2. Related Work

Many techniques of reducing the overhead of synchronization have been proposed for Java. Bacon, et al. [1] proposed the *thin lock*, which only requires one atomic operation to acquire and release it. The thin lock does not use any multi-word monitor structures in most cases. Onodera, et al. [3] proposed the *Tasuki lock* to fix the thin lock's problems with unbounded busy-waiting by using a *flc (flat lock contention)* bit. For techniques that always use a multi-word monitor structure, Agesen, et al. [2] proposed the *meta lock*. However, the meta lock requires an extra atomic operation when the lock is released and therefore performs two atomic operations for each synchronization operation. Dice [4] also proposed the *relaxed lock*, which only performs one atomic operation. These techniques require serialization and one or two atomic operations in their best cases.

Addressing the problems of cost for serialization and complex atomic operations in SMP environments, Kawachiya, et al. [5] proposed *lock reservation*. Bacon, et al. [6] 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. [7] also presented an idea for *global bits* that indicate whether or not objects have escaped. Onodera, et al. [8] proposed a reservation-based spin lock, in which non-reserving threads cannot reserve locks but perform a spin lock instead of canceling the reservation.

In comparison to lock reservation, our approach has advantages when locking shared objects, i.e., every thread benefits from an efficient lock using temporal locality, and a non-owner can efficiently change ownership using 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 and the previous owners would incur penalties: the new owner would suffer from the cost of the suspend and resume operations, and the previous owner would suffer from the overhead of being suspended and

then resumed. The cost of suspend and resume operations is usually very large.

In the absence of contention for the lock, Lamport [9] presented *a fast mutual exclusion algorithm*, which has constant time complexity using only simple instructions. Lamport's algorithm performs five memory accesses for entering and two for leaving critical sections. However, the code within the critical sections cannot start executing until the third memory access has been completed, since the memory access has *acquire ordering semantics* for the critical section. In terms of memory efficiency, Lamport's method requires memory space ☆ proportional to the number of threads. It also uses two variables that can have process IDs for each lock. Our algorithm, in contrast, allows the lock algorithm and the critical section code to be executed in parallel and only requires two bits and the thread ID field for each lock (or object).

Andersen, et al. [10] surveyed 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 on fast mutual exclusion algorithms that not only use a fixed number of simple instructions, but also allow the execution of critical sections in parallel while only requiring a fixed memory space.

3. Study of Lock Behavior

This section presents our analysis of lock behavior. We first consider the frequency of lock operations in Java programs to clarify the importance of efficient lock operations. We next demonstrate that owner locality for shared objects can be observed in real-world applications.

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

3.1 Lock Frequency

Since lock operations synchronize memory accesses and modern processors usually have

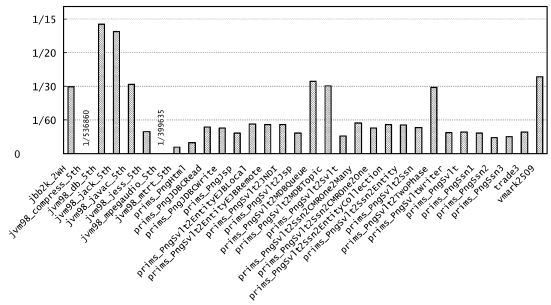

Fig. 1 Ratio of lock operations to heap accesses (higher bars indicate more frequent locks).

memory-dedicated execution pipelines, lock operations can be regarded to stall the memory pipelines. We therefore 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 ☆☆, and internal fields such as array sizes and tables for virtual methods.

Figure 1 shows this ratio for each benchmark. The 1/N labels on the Y-axis signify that N heap accesses are performed for every lock operation. On average, a lock operation is performed for every 55 heap accesses. Locking performance is therefore critical for programs that perform lock operations with such high frequency, considering that a single lock acquisition stalls many CPU cycles.

3.2 Owner Locality

We focused on *shared* objects, which are actually locked by more than one thread during the measurement. A *non-shared* object is an object that is only locked by a single thread. Non-shared objects can be non-escaping objects, which are only visible to one thread but not to other threads. Since the lock operations for these non-escaping objects could be removed by various other techniques using the knowledge about their non-escaping characteristics, e.g., by allocating these objects in a special thread-local heap [7], we have not discussed them further in this paper.

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

☆ A Boolean variable $b[i]$ used in Algorithm 2 [9] usually consumes one byte in actual processors due to the atomicity of stores.

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

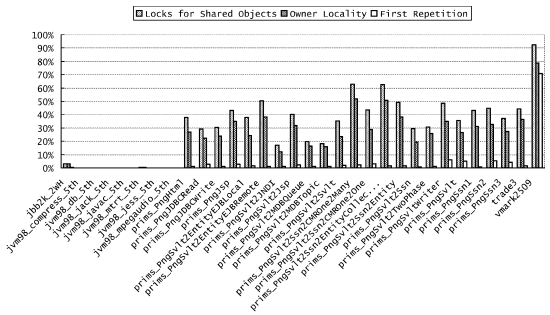

Fig. 2 Owner locality relative to all lock operations.

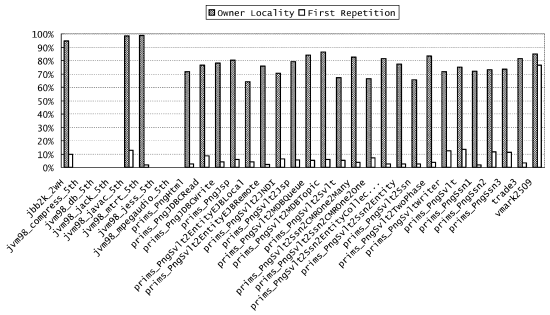

Fig. 3 Owner locality for shared objects.

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 Fig. 2 and the first bars in **Fig. 3** show owner locality. The owner locality in Fig. 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 Fig. 3 is the ratio of the same number to the count of lock operations for shared objects. The taller bars indicate that the owners of those objects rarely change during all the lock operations.

We focused on owner locality for shared objects in Fig. 3 while Fig. 2 shows the ratio to the *total* number of lock operations. The owner locality is up to 86.4% and 76.0% on average for the numbers of lock operations for shared objects in Fig. 3.

The third bars in Fig. 2 and the second bars in Fig. 3 show the ratios of the first repetitions [5]. 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 indicate when the first locker continues locking without the other threads' locks.

The first repetition is up to 13.5% and only 5.9% on average, excluding VolanoMark ☆. In contrast, excluding VolanoMark does not affect 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 Fig. 3, since the ratio of lock operations for the shared objects is very low, as can be seen in Fig. 2.

4. TO-lock

This section explains the algorithm for the *tentative ownership lock* or *TO-lock*. We enhanced it [11] by eliminating a flag and simplifying the handling of lock collisions. We first explain the variables used in the algorithm and then cover it step-by-step. Finally, we explain the memory ordering that the algorithm requires.

4.1 Variables

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

4.1.1 Tid

The tid field holds the identifier of the thread that currently has the lock or that last released it. We use this variable to keep track of the temporal locality of the threads that acquire the lock for the object.

The thread checks the tid several times during the locking process. If the tid reveals another thread's ID, then the thread attempts to replace the tid with its own ID.

4.1.2 Tel1 and tel2

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

A pair consisting of a *tel1* flag and a *tel2* flag indicates 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, 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*. The *tel1* flag prevents other threads from changing the

☆ We should interpret the results for VolanoMark carefully. The lock operations for objects of a specific class dominate 70% of the lock operations for shared objects, and these objects are mostly non-shared.

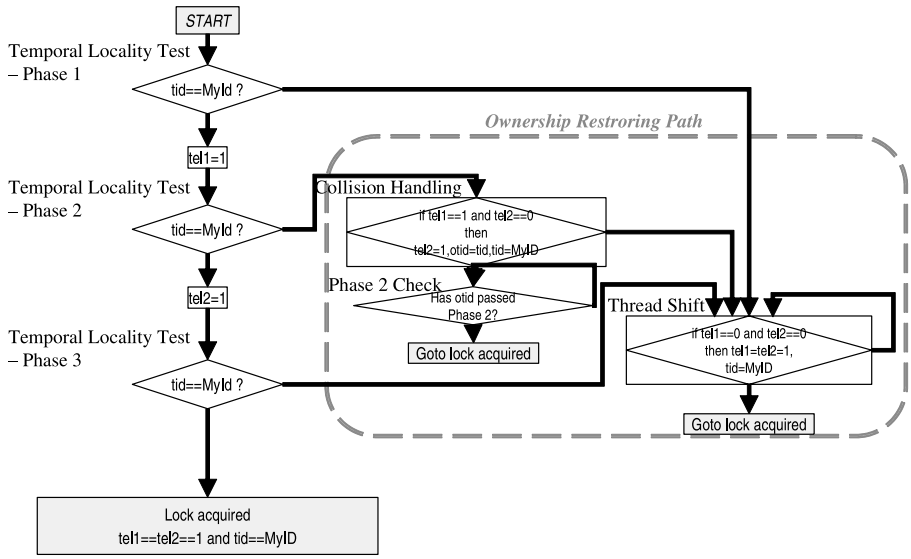

Fig. 4 Flow chart for TO-lock algorithm.

tid to restore the temporal locality of threads while the current thread is executing the algorithm's fast path. We will explain this change in tid in Section 4.2.4.1. We need the *tel2* flag to show that the lock has been acquired. 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.

4.2 Locking Algorithm

This section explains the algorithm. **Figure 4** is a flow chart for the algorithm, which consists of three components: (1) *temporal locality tests*, (2) *thread shift*, and (3) *collision handling*. We describe each of these steps in detail in the following subsections. The diamonds within rectangles in Fig. 4 denote atomic operations. For the conditions specified within diamonds, the algorithm proceeds downward if the condition is true and rightward otherwise.

4.2.1 Temporal Locality Test - Phase 1

The thread first checks if the tid is still the same as its own ID. If this test succeeds, the thread sets the *tel1* flag to inhibit a *thread shift*, i.e., 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 on thread shift are explained in Section 4.2.4.1.

4.2.2 Temporal Locality Test - Phase 2

The thread again checks to ensure the tid is still the same as its own ID. This test is required to ensure a thread shift did not occur in the period between the load and the store of Phase 1. If the test succeeds, the thread sets the *tel2* flag to inhibit *collision handling*. If the tid does not match, the thread proceeds to the step for collision handling. The details on this step are explained in Section 4.2.4.2.

4.2.3 Temporal Locality Test - Phase 3

The thread again checks to ensure the tid is still the same as its own ID. This test is required to ensure that no other threads have succeeded in collision handling in the period between the load and the store of Phase 2. If the test succeeds, 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.

4.2.4 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 then tries to reclaim the ownership of the object.

4.2.4.1 Thread Shift

This step restores the temporal locality of threads. The thread attempts to place its own

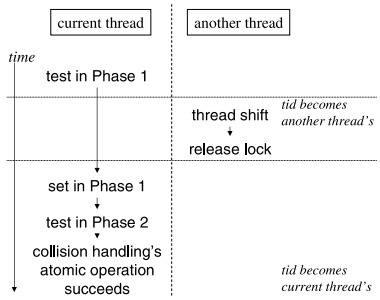

Fig. 5 Atomic operation in collision handling performs another form of thread shift.

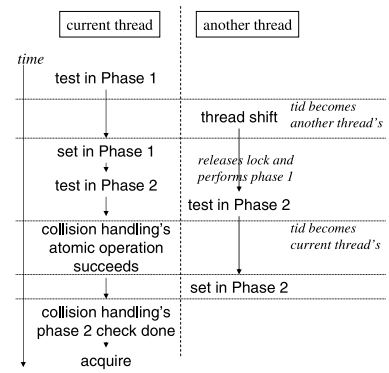

Fig. 6 Phase 2 check inhibits another thread from setting tel2 after the current thread releases the lock.

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 4.1.2, the cleared *tel1* and *tel2* flags show that the object is not currently locked by any thread.

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

If multiple threads perform thread shift at the same time, only one thread succeeds. The other threads fail and performs a spin-locking loop, similar to other compare-and-swap-based algorithms [1),3),12)].

4.2.4.2 Collision Handling

If another thread changed the tid using thread shift in the period between the test and the setting of Phase 1 performed by the current thread, the current thread enters this step. Since locks are rarely contended for and the period is usually very short, this step rarely occurs. Assume that another thread not only acquired but also released the lock within the same period (see **Fig. 5**). Without this step, the *tel1* flag set by the current thread is never cleared and no threads can acquire the lock by thread shift. Atomic operation in the step resolves this situation.

Atomic operation by the current thread, however, also succeeds if another thread acquired, released the lock, and entered Phase 2 to acquire the lock but has not yet set *tel2* within the same period as previously explained (see **Fig. 6**). In this case, if the current thread releases the lock acquired by atomic operation and then another thread sets *tel2*, a situation in which *tel2* is set but *tel1* is not occurs. The algorithm does not allow for this situation.

To avoid this, the *phase 2 check* that follows atomic operation ensures that the current thread completes lock acquisition after another thread that was captured by atomic operation (denoted by otid in Fig. 4) leaves Phase 2. The phase 2 check can be implemented by obtaining the current instruction address for otid and comparing it with the registered code regions for Phase 2. For other implementations, we can use a flag in the thread structure, which indicates whether or not a thread is within Phase 2, by adding an overhead to maintain the flag in Phase 2.

4.2.5 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 using a simple store operation.

4.3 Memory Ordering

The stores to the *tel1* and *tel2* flags and the succeeding load of *tid* are ordered for the temporal locality tests. 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 non-locking operations.

When releasing the object, we ensure *release ordering semantics* to clear the two flags to release the object. Clearing the flags using release ordering semantics ensures that any updates to the memory when the object is being locked will become visible to the other processors before the object is released.

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 specifications in the design of distributed systems. We used Spin as an exhaustive verifier, which rigorously proved the validity of user-specified correctness requirements.

5.1 Safety

We checked three safety properties for our algorithm, mutual exclusion, avoidance of illegal states of variables, and liveness.

5.1.1 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 Spin's *assertion* feature. We tested the following assertions at points in the program where a thread acquires the lock, i.e., (1) the tid is the thread's own ID, (2) the *tel1* and *tel2* flags are set, and (3) only a single thread acquires the lock by using multiple threads. All the threads lock a single shared object, and each thread repeats our algorithm infinitely. We only ran up to five threads because of the limited physical memory of the machine we used for the verification.

5.1.2 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 4.2.4.2.

We confirmed that this situation never occurred by using Spin's *linear time temporal logic formulae* feature. Using this, we could verify that this correctness requirement holds for every step of all the traces.

5.2 Liveness

We confirmed the liveness properties: If threads are trying to acquire a lock, one of them will eventually acquire it. We used Spin's feature of the linear time temporal logic formulae.

We also performed another experiment to verify that each thread that was going to acquire the lock would eventually acquire it. We examined the number of lock acquisitions for each thread for this experiment. We confirmed that every thread acquired the lock a specified number of times. The compare-and-swap operation in the ownership-restoring path of our algorithm ensures that these threads can change ownership as in other compare-and-swap-based algorithms [1),3),5),13)].

6. Experimental Results

This section presents an evaluation of our algorithm. We first explain how we performed the experiment. We next show how we implemented the algorithm on real-world processors. We then present the results of the experiment.

6.1 Methodology

We wrote a program that executes in a loop, containing a code segment and single lock operation, using a hand-optimized IA-32 assembly code. The code segment consists of repetitions of a *simple sequence*, four loads, and the store of their total, which simulates heap accesses in server programs. We determined this ratio between the loads and stores based on the observation that loads were about 80% of the total memory operations in the server programs analyzed in Section 3. We ran our experiments on an SMP machine, i.e., an IBM IntelliStation Z-Pro with two physical Intel Xeon 3.06-GHz processors.

We measured execution times for two extreme cases with no lock contentions, changing the number of memory operations from 10 to 80. For all lock operations, we only performed phases one through three (or the fast path) of the algorithm in the first case and only performed thread shift (or compare-and-swap) in the second case. We calculated the estimated execution times for cases when the fast path was used for 86.4% and 76.0% of the total locks and compare-and-swap was used for the remaining locks. For real-world owner locality scenarios, the owner locality ranges up to 86.4% and is 76.0% on average as shown in Section 3. We can compare ours with the algorithm proposed by Kawachiya, et al. [5)] by using the results for compare-and-swap since theirs performs compare-and-swap for shared objects as explained in Section 1. We ran the program five times for all lock frequencies and used the average of the execution times.

6.2 Processor-specific Considerations

Our algorithm does not rely on memory ordering between lock and non-locking operations, though it does rely on the memory ordering within the lock operation.

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 or specific load. We therefore need some hardware support to efficiently implement the algorithm on existing processor architectures.

We used a special technique, *store forwarding avoidance*, in this experiment to control the

memory ordering for our algorithm. If a store to a memory area and a load from the same area appear in the code in that order, these operations are performed in that order for most processors, even under relaxed memory consistency models. Furthermore, the store-to-load forwarding feature [14] 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 [15]. In that case, the store becomes visible and then the data is then loaded from the memory hierarchy. We relied on this characteristic to ensure the memory ordering expected by the scalable synchronization algorithm in the relaxed consistency model.

For out-of-order execution processors, the CPU hardware can reorder independent instructions. The locking and non-locking instructions are independent of each other, and therefore the CPU can execute non-locking instructions while it executes a sequence of locking instructions. However, the hardware usually has some restrictions to perform this special memory ordering using store forwarding avoidance to perform efficiently. For example, Intel's NetBurst-based processors, such as the Xeon we used, incur penalties of replays for failed speculative loads. Therefore, compilers should schedule these instructions to avoid such penalties. We scheduled these instructions in this experiment. For in-order-execution processors such as IA-64, the compilers should carefully schedule the instructions using store forwarding avoidance to evade unnecessary stall cycles.

6.3 Results

Figure 7 plots the results of the experiment. The X-axis represents the lock frequency. The 1/N labels for the X-axis signify that N memory operations are performed for every lock operation. A large 1/N indicates a high lock frequency. The Y-axis shows the speedup in average execution time for each loop iteration, where taller is faster.

Our algorithm significantly improved the performance for each lock frequency. For a real-world owner locality scenario, or 86.4% owner locality, we achieved 2.02 times speedup on average. For another real-world owner locality scenario, or 76.0% owner locality, we achieved 1.80 times speedup on average. For both scenarios, we achieved the best speedup for 1/20 lock frequency, i.e., 2.21 times for 86.4% owner locality and 1.93 times for 76.0% owner locality.

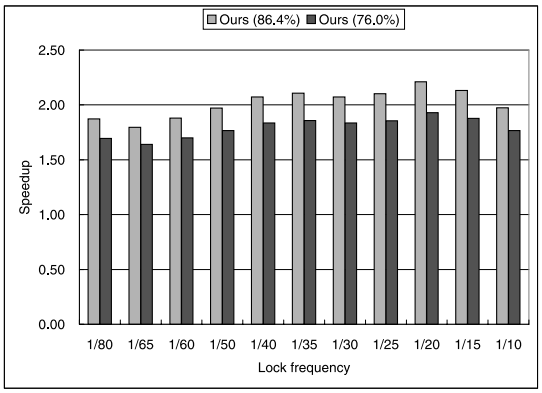

Fig. 7 Speedup in execution time per loop iteration (taller is faster).

7. Conclusion

This paper presented a novel algorithm for optimizing lock operations, which is particularly useful for Java programs that frequently perform lock operations for shared objects. It only uses simple memory operations in the absence of contention, even if these objects are actually shared and locked by multiple threads. By exploiting owner locality, 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 lock operation. We experimented with many Java programs to profile lock behaviors. The results revealed that, for real-world server-side programs, objects are shared among multiple threads and owner locality is not infringed. We verified the safety of the algorithm by using a software tool called 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 complex atomic instruction by experimenting with assembler programs that perform a typical number of memory operations within and outside the critical section. The experimental results demonstrated that our algorithm can improve the performance of the code by 80% on average for a real-world owner locality scenario on Intel Xeon processors, compared to the existing approach using atomic compare-and-swap instruction.

Acknowledgments We would like to thank

Dr. Akira Koseki for his useful comments on our algorithm and for guiding us toward improving it and verifying of one of its safety properties. Thanks also go to Drs. Kiyokuni Kawachiya and Tamiya Onodera for their feedback and encouragement, and to Mr. Toshio Suganuma for his many suggestions to improve the presentation of the paper. Finally, we wish to thank the anonymous reviewers of IPSJ and PACT2004 for their constructive suggestions on how to improve the paper.

References

1) Bacon, D.F., Konuru, R., Murthy, C. and Serrano, M.: Thin Locks: Featherweight Synchronization for Java, *ACM SIGPLAN '98 Conf. on Programming language design and implementation*, pp.258–268, ACM Press New York, NY (1998).
2) Agesen, O., Detlefs, D., Garthwaite, A., Knippel, R., Ramakrishna, Y.S. and White, D.: An efficient meta-lock for implementing ubiquitous synchronization, *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp.207–222, ACM Press, New York, NY (1999).
3) Onodera, T. and Kawachiya, K.: A Study of Locking Objects with Bimodal Fields, *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp.223–227, ACM Press, New York, NY (1999).
4) Dice, D.: Implementing fast Java monitors with relaxed-locks, *Proc. Java VM Research and Technology Symp.*, pp.79–90, USENIX Association (2001).
5) Kawachiya, K., Koseki, A. and Onodera, T.: Lock reservation: Java locks can mostly do without atomic operations, *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages and Applications*, pp.130–141, ACM Press, New York, NY (2002).
6) Bacon, D.F. and Fink, S.J.: Method to provide concurrency control over objects without atomic operations on non-shared objects, U.S. patent (2000).
7) Domani, T., Goldstein, G., Kolodner, E.K., Lewis, E., Petrank, E. and Sheinwald, D.: Thread-Local Heaps for Java, *Intl. Symp. Memory Management*, pp.183–194, ACM Press, New York, NY (2002).
8) Onodera, T., Kawachiya, K. and Koseki, A.: Lock Reservation for Java Reconsidered, *Proc. 18th Euro. Conf. on Object-Oriented Programming*, pp.560–584, Springer-Verlag Berlin (2004).
9) Lamport, L.: A fast mutual exclusion algorithm, *ACM Trans. Comput. Syst.*, Vol.5, No.1, pp.1–11 (1987).
10) Anderson, J.H., Kim, Y.-J. and Herman, T.: Shared-memory mutual exclusion: major research trends since 1986, *Distributed Computing*, Vol.16, No.2-3, pp.75–110 (2003).
11) Ogasawara, T., Komatsu, H. and Nakatani, T.: TO-Lock: Removing Lock Overhead Using the Owners' Temporal Locality, *Proc. 13th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pp.255–266, IEEE Computer Society, Los Alamitos, CA (2004).
12) Dimpsey, R., Arora, R. and Kuiper, K.: Java server benchmarks, *IBM Systems Journal*, Vol.39, No.1, pp.151–174 (2000).
13) Gagnon, E.M. and Hendren, L.J.: SableVM: A Research Framework for the Efficient Execution of Java Bytecode, *Proc. Java VM Research and Technology Symp.*, pp.27–39, USENIX Association (2001).
14) Johnson, M.: *Superscalar Microprocessor Design*, Prentice Hall, Englewood Cliffs, NJ (1991).
15) Intel Corporation: *Intel Pentium 4 and Xeon Processor Optimization Reference Manual*, Intel Corporation, Mt. Prospect, IL (2002).

(Received April 26, 2005)
(Accepted September 14, 2006)
(Online version of this article can be found in the IPSJ Digital Courier, Vol.2, pp.759–767.)

Takeshi Ogasawara joined the IBM Tokyo Research Laboratory in 1991. His research interests include optimizations using compilers and runtime systems for parallel computers.

Hideaki Komatsu joined the IBM Tokyo Research Laboratory in 1985. His research interests include compiler optimizations for parallel architectures and loop optimizations for massively parallel computers.

Toshio Nakatani joined the IBM Tokyo Research Laboratory in 1987. His research interests include computer architecture, optimizing compilers, and algorithms for parallel computer systems.