Java Locks: Analysis and Acceleration

by

Kiyokuni Kawachiya

A dissertation
submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

in the Graduate School of Media and Governance
at Keio University

Java Locks: Analysis and Acceleration

Kiyokuni Kawachiya

Abstract

This dissertation presents several techniques to accelerate Java locks based on the analyses of their behavior from various viewpoints.

Ten years have passed since Java was announced, and now the language has come to be used in various fields including large-scale Web services. To make such applications practical, performance improvements for the Java execution environments are indispensable. In Java, the language directly supports parallel processing, and lock operations are executed very frequently. Their overhead is considerable. For example, one report said that 1/5 of the execution time was consumed by lock processing in an early Java environment. Therefore, accelerating Java locks has been very important to make Java environments run faster.

From the analysis of locks in various Java applications, we discovered that most contentions are temporary in Java. To exploit this *contention transience*, this dissertation presents a new method for Java locks, in which a lock can be acquired with just one atomic operation in the absence of contention. The method, called *tasuki lock*, increased the performance of Java application by up to 13.1%, and is used by IBM's production Java virtual machines.

Next, uncontended Java locks were also analyzed, and it turned out that for many objects, the lock is acquired primarily by a specific thread. By exploiting the *thread locality*, this dissertation proposes a novel technique of reserving locks for threads. With the *reservation lock*, the reservation-owner thread can acquire a lock without any atomic operations, and the performance was further improved by up to 53%.

This dissertation also presents an improvement of the reservation lock, where the reservation is not canceled and the owner thread can always acquire its locks very quickly. In this *asymmetric lock*, we observed a performance improvement of up to 7.9%. A unique methodology for introducing a new feature into Java locks is also used in this algorithm.

The main contributions of this dissertation are: the findings about new characteristics of Java locks, the proposals of new lock methods that exploit these characteristics, and their implementation and evaluation on a state-of-the-art Java system.

Javaのロック処理の分析と高速化

河内谷 清久仁

要旨

本論文は，Java ロックの挙動分析とそれに基づくいくつかの高速化手法の提案・評価を示すものである．

発表以来 10 年が経ち，Java 言語はウェブサービスをはじめとする様々な分野で使われるようになった．これらのアプリケーションを実用的なものにするためには，Java 処理系自体の性能向上が必須であった．Java では言語自身が並列処理をサポートしており，スレッド間の排他制御のためのロック操作が頻繁に行われる．そのオーバーヘッドはかなり大きく，初期の処理系では，実行時間の５分の１がロック処理に占められていたという報告もあるほどである．つまり，Java 処理系の性能向上に，Java ロックの高速化は避けて通れない課題である．

様々な Java アプリケーションの分析から我々は，Java ではロックの衝突が持続することは稀で，多くは一時的なものであるという「衝突の一時性」を発見した．本論文ではまず，この性質を活用する Java ロック手法「たすきロック」を提案する．これにより，衝突が無い状態では不可分命令一つでロックを獲得できるようになり，アプリケーションの実行性能が最大 13.1%向上した．この手法は現在でも，IBM の商用 Java 処理系で用いられている．

次に，衝突が起こらないロックの挙動についても分析を行った結果，Java では多くのオブジェクトがそれぞれ特定のスレッドからのみロックされているという「スレッド局所性」が見られることを発見した．この性質を活用するため，本論文では特定のスレッドに予約を与えるロック手法を提案する．この「予約ロック」方式で，予約スレッドは不可分命令すら用いずにロック処理を行えるようになり，実行性能は最大でさらに 53%向上した．

予約ロックでは，予約スレッド以外がロックを試みた場合，まず予約の解除が行われる．本論文ではさらに，この点を改善し特定のスレッドが常に高速なロック処理を行えるような非対称性を備えたロック手法についても提案する．この「非対称ロック」手法では，最大 7.9%の性能向上を確認した．また同時に，Java ロックの新しい実装アプローチについても示している．

本論文の主な貢献としては，ロック処理の挙動分析から Java ロックの特徴的な性質をいくつか発見したこと，それらの性質を利用する新しいロック手法について提案したこと，さらにそれらを実装し評価を行い，性能向上を確認したこと，の３点があげられる．

Contents

List of Figures

List of Tables

Acknowledgements

It is a great pleasure to thank the people who made this dissertation possible.

First of all, I would like to express my great appreciation to Professor Hideyuki Tokuda, the main advisor of this dissertation, for his suggestions and support. Without his continuous help and advice, I could not have completed the dissertation. I also learned from him the true form of research while I was working on joint research on continuous media processing.

I would like to say a special thanks to the thesis committee members, Professor Koichi Furukawa, Professor Yasushi Kiyoki, and Professor Tatsuya Hagino. They provided a lot of their valuable time for the examination. Their keen comments and advice gave me fresh perspectives to refine the dissertation.

I am grateful to Michiko Nitta, the secretary of Professor Tokuda, for her encouragement and sophisticated administrative support, such as arranging meetings with the busy professors.

This dissertation is based on several research projects I did in IBM Tokyo Research Laboratory. Therefore, I am deeply indebted to my colleagues in the company.

Tamiya Onodera and Akira Koseki worked with me on improving the performance of Java locks for several years. David Bacon and Chet Murthy gave us valuable information on thin locks and comments on our lock algorithms. Toshio Nakatani, my manager for eight years, gave me various kinds of help for the research on Java locks, and encouraged me to collect it into this dissertation.

The members of the Systems Group, formerly called the Network Computing Platform Group, in IBM Tokyo Research Laboratory also gave me valuable comments. They always discussed with each other how to improve the performance of Java. In particular, I could not implement and evaluate various lock algorithms without the state-of-the-art Java virtual machines, whose JIT compilers were developed by the group.

I also wish to thank the director of our laboratory, Kazushi Kuse, who gave me support to write this dissertation. Shannon Jacobs also greatly helped me to improve the quality of this dissertation by his exhaustive proofreading.

I would like to express my special appreciation to my parents for their help in bringing me up and for giving me support for my education. Finally, my deepest appreciation goes to my family, Sachiko, Yoichi, and Asumi Kawachiya, for their continuous support and encouragement.

November 30, 2005
Kiyokuni Kawachiya

Chapter 1

Introduction

1.1 Background

Since Java [41] was announced in 1995 [23], the language has penetrated into various fields. Now Java can be regarded as one of the most popular programming languages together with C and C++. On the server side, it is the standard for constructing Web services [108] using J2EE [106]. Also on the client side, it is used in development environments such as Eclipse [34], and more recently, Java-based rich client applications such as Lotus Workplace Client [52] are appearing, too. To make such applications practical, the performance of the Java execution environments has been a very important issue.

One special feature of Java is that it has built-in support for parallel processing. Concretely, this is realized by creating multiple *threads*, which are executed independently and asynchronously in the Java execution environment. For the mechanism of thread synchronization and mutual exclusion, Java uses the high-level concept called *monitors* [20, 21, 46], which are associated with objects. At most only one thread can enter a monitor simultaneously, by acquiring the associated object's *lock*. When a thread attempts to enter a monitor being entered by another thread, the second thread is blocked until the first thread exits from the monitor and releases the object's lock. In Java, entering and exiting from a monitor are implicitly specified by declaring a method or a block as `synchronized`, rather than by specifying the lock acquisition and release explicitly.

Because of the built-in support for multi-threaded programming, Java programs, especially its libraries, must be written as thread-safe, so methods that handle data which may be shared among threads must be declared as `synchronized`. As a result, Java applications perform a significant number of lock operations. It was reported that 19% of the total execution time was used for thread synchronization in an early version of a Java execution environment [6].

Obviously, the lock operations are not the real task a user wants to perform, so reducing the lock overhead is important to improve the performance of a Java execution environment.

1.2 Research Goal

This research aims at accelerating the performance of Java locks, which is very important to improve the overall performance of Java applications as explained above.

As a side effect of Java being designed to execute machine-independent *bytecode* on a *Java virtual machine* (JVM), its performance was very poor in early days when compared to languages whose programs are compiled into native instructions. However, mainly because of the evolution of just-in-time (JIT) compilation technologies [57, 58, 91, 102, 103], the execution speed of Java application has been considerably improved.

One approach to reduce the lock overhead is to eliminate unnecessary locks at the time of the JIT compilation, for example, by using escape analysis [4, 16, 17, 24, 25, 96, 97, 118] or analysis of nested invocations [101]. However, since Java is a dynamic language which allows class loading during execution, it is intrinsically difficult to eliminate all unnecessary locks at the times of JIT compilation. As will be shown in the following chapters[1], many lock operations still remain even if a JIT compiler with various lock elimination techniques is used. In actuality, when a JIT compiler accelerates other code portions that were executed by an interpreter, the overhead of lock operations becomes relatively conspicuous. This implies that, to reduce the overhead of Java locks, it is necessary to improve their essential performance by contriving better lock algorithms.

In early versions of Java virtual machines, Java locks were naively implemented by directly using synchronization mechanisms provided by the underlying operating systems, such as mutex. This is a straightforward way, but very expensive since a system call is issued for every lock operation. To improve this situation, various research has already been done on how to implement low-cost Java locks [3, 11, 28, 88]. In principle, this research attempts to improve performance by optimizing *common cases*, which are found from the analysis of lock behavior in real Java programs.

In their seminal work, Bacon et al. exploited the observation that Java locks are mostly not contended, and proposed an excellent optimization for Java locks, called *thin locks* [11]. The algorithm allows a lock to be acquired and released with a few machine instructions in the uncontended case. When contention occurs, the lock is converted to *inflated* mode, where the OS-provided heavyweight synchronization structure is used for the first time.

The research goal of this thesis is to design new algorithms for Java locks and ultimately accelerate the lock performance. For that purpose, we exhaustively measured the behavior of Java locks in existing Java applications from various viewpoints, and newly discovered several unique characteristics of Java locks. We then designed several new algorithms for Java locks that exploit these characteristics [65, 66, 67, 88, 89]. We also showed the effectiveness of the proposed lock algorithms by implementing them on production Java virtual machines and running several practical Java applications.

[1] Examples will later be shown in Tables 5.6 and 6.2.

1.3 Thesis Statement

We require a fast algorithm for Java locks which reduces the overhead of synchronization and improves the performance of Java applications. In this thesis, contentions in Java locks were analyzed first, and it was discovered that most contentions are transient. We thus first propose a new method for Java locks that utilizes the *contention transience* [88]. In this method, a heavyweight synchronization mechanism is used only while the lock is contended. Once the contention is cleared, the lock returns, or *deflates*, to lightweight mode, where the lock can be acquired with only one atomic operation. The method, called *tasuki lock*, is used in IBM's production Java virtual machines.

Next, uncontended Java locks were also investigated, focusing on the relationships of each object and the threads acquiring the object's lock. It turned out that for many objects, the lock is primarily acquired by a specific thread, even in multi-threaded Java programs. By exploiting the *thread locality*, we propose a novel technique of *reserving* locks for threads [65, 66]. In the *reservation lock* algorithm, a thread can acquire a lock without any atomic operations if the lock is reserved for the thread. Otherwise, it cancels the reservation and falls back to a conventional lock algorithm.

We also show an improvement of this reservation lock, where the cancellation is eliminated and a specific thread can always acquire the lock very quickly [67, 89]. This *asymmetric lock* algorithm is based on unique methodology of replacing the atomic operation of the conventional algorithm with an *asymmetric spin lock*.

This thesis, thus, contributes to the following:

Discovery of two unknown characteristics of Java locks:
We discovered the contention transience and thread locality of Java locks by analyzing their behavior in real Java applications from various viewpoints.

Proposal of new concepts and actual algorithms for Java locks:
To exploit the contention transience and thread locality, we proposed new concepts of deflation and reservation. We also designed new algorithms for Java locks that incorporated the concepts. The algorithms are precisely designed to utilize previous ideas while reducing the overhead in the common cases.

Implementations and evaluations in actual environments:
We implemented the new algorithms in a state-of-the-art Java system, and showed actual performance improvements in real Java applications as well as in micro-benchmarks.

Consequently, this thesis contributes to improving the performance of Java applications by accelerating the processing of Java locks based on deep analysis of their behavior in real Java applications.

1.4 Thesis Configuration

The remaining chapters of this thesis are organized as follows.

Chapter 2 describes the mechanism of Java locks and introduces several major techniques to accelerate their performance prior to our research. Chapter 3 then defines the problems to be solved to improve the performance of Java locks and discusses possible approaches.

The following three chapters show our analyses and accelerations of Java locks. Chapter 4 describes the tasuki lock, which is our first enhancement based on the finding of contention transience. Chapter 5 explains the second important finding of thread locality, and proposes an enhancement by using the reservation lock. Chapter 6 shows an improvement of the reservation lock, the asymmetric lock, introducing a new methodology to construct Java locks from a primitive spin lock.

Chapter 7 discusses related work from various viewpoints, and Chapter 8 concludes this thesis.

Chapter 2

Java Locks

2.1 Introduction

This chapter first explains the mechanism of thread synchronization in Java and describes Java locks in detail, which is a core component of the mechanism. Then several major acceleration techniques for Java locks that have been done prior to our research will be introduced, showing their implementations.

2.2 Overview of Java Locks

One unique feature of Java is that it has built-in support for parallel processing. In Java, *threads* are provided as a first class primitive in the language, unlike C, where threads are provided as an external library. Therefore, Java applications can utilize threads very easily, just by extending the `Thread` class and implementing its `run` method[1], and starting its instance by invoking the `start` method [7]. The new thread starts execution from the `run` method.

Figure 2.1 shows an example of thread programming in Java. The program is executed from the `main` method at line 27. First, two counters, c_1 and c_2 (lines 29–30), and three threads, A, B, and C (lines 33–35), are created as shown in Figure 2.2. The threads are started by the invocations of the `start` method at line 38, then execute the `run` method (line 19) independently. In the method, a counter specified at the thread creation time is incremented 10,000,000 times. After starting the three threads, the main thread waits for their termination (line 41) and exits after printing the values of the two counters (lines 44–45).

The threads are executed independently and asynchronously on a *Java virtual machine* (JVM). To avoid the situation that multiple threads increment the same counter simul-

[1]There is another way, where a `Thread` instance is created while specifying a class which implements a `Runnable` interface and its `run` method, and is started through the `start` method.

```
 1 : // Simple counter
 2 : class MyCounter {
 3 :     int value = 0;
 4 :
 5 :     synchronized int add(int i) { // Guarded by Java locks
 6 :         value = value + i;
 7 :         return value;
 8 :     }
 9 : }
10 :
11 : // Simple thread
12 : class MyThread extends Thread {
13 :     MyCounter counter;
14 :
15 :     MyThread(MyCounter c) { // Constructor
16 :         counter = c;
17 :     }
18 :
19 :     public void run() { // Entry point of the thread
20 :         for (int i = 0; i < 10000000; i++)
21 :             counter.add(1);
22 :     }
23 : }
24 :
25 : // Test driver
26 : class MyExample {
27 :     public static void main(String[] args) throws InterruptedException {
28 :         // Create counters
29 :         MyCounter c1 = new MyCounter();
30 :         MyCounter c2 = new MyCounter();
31 :
32 :         // Create threads with specified counters
33 :         MyThread thA = new MyThread(c1);
34 :         MyThread thB = new MyThread(c1);
35 :         MyThread thC = new MyThread(c2);
36 :
37 :         // Start the threads
38 :         thA.start(); thB.start(); thC.start();
39 :
40 :         // Wait for the threads' termination
41 :         thA.join();  thB.join();  thC.join();
42 :
43 :         // Show the counters
44 :         System.out.println("C1 is " + c1.value);
45 :         System.out.println("C2 is " + c2.value);
46 :     }
47 : }
```

Figure 2.1: Example of thread programming in Java

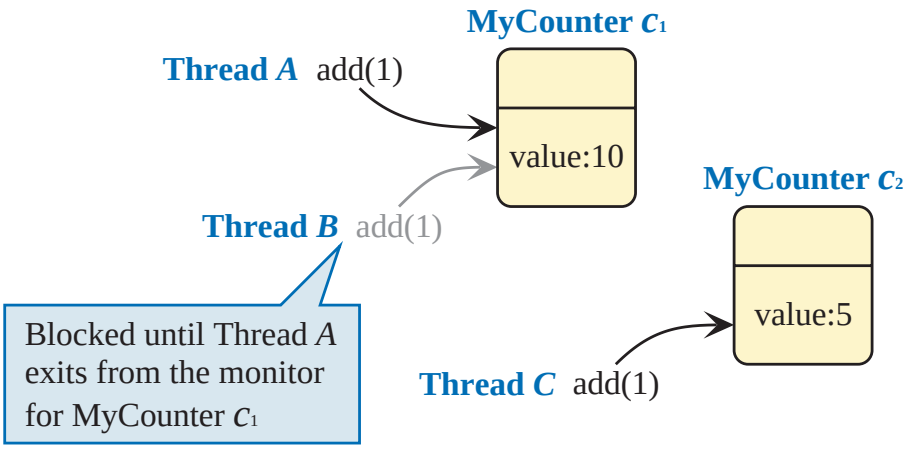

Figure 2.2: Threads and objects in the example

taneously and the results become inconsistent, some mechanism of mutual exclusion (or thread synchronization) should be supported. For that purpose, Java provides a mutual exclusion mechanism based on the concept of *monitors* [20, 21, 46], which are associated with objects [7].

At most only one thread can enter an object's monitor simultaneously. When another thread is in the monitor, a thread trying to enter is forced to wait by the JVM until the first thread exits from the monitor. In Java, a thread is permitted to enter a monitor recursively. In this case, a waiting thread can enter the monitor only when it has been exited the same number of times as it was entered.

In Java, the monitor operations are specified by declaring a method or a block as `synchronized`, rather than by specifying the enter and exit independently. If the specified method is an instance method, its receiver object, `this`, is used for a monitor. If the method is a class method (static method), its class object is used for a monitor. When a block is declared as `synchronized`, an object which will be used for a monitor is specified at the same time.

In the example of Figure 2.1, the `add` method of the `MyCounter` class is declared as `synchronized`. Therefore, for each `MyCounter` object, at most one thread can execute the `add` method simultaneously, so the counter's consistency is preserved. Note that the `add` methods can be executed for counters c_1 and c_2 simultaneously by different threads, because the monitor is prepared independently for each object,

To realize this monitor behavior, the JVM uses a lock mechanism, *Java lock*, associated with each object. The lock is *acquired* by a thread when it enters the object's monitor, *held* while the thread is in the monitor and executing the `synchronized` method, and *released* when the thread exits from the monitor. If a thread attempts to acquire a lock held by another thread, the second thread is blocked until the first thread releases the lock.

Table 2.1: Investigation of Java locks in SPECjvm98

Program	Number of lock acquisitions
_202_jess	14,646,978
_201_compress	28,895
_209_db	162,117,521
_222_mpegaudio	27,168
_228_jack	38,570,415
_213_javac	47,062,772
_227_mtrt	3,522,926

In Java programs, lock acquisitions and releases are performed very frequently. For example, in the example of Figure 2.1, 30,000,000 acquisitions and releases are done because each of the three threads invokes the `add` method 10,000,000 times. Similar situations are also observed in more realistic Java programs. Table 2.1 summarizes the number of lock acquisitions performed during the execution of seven programs[2] from the SPECjvm98 benchmark [100]. From the table, we can see that many lock operations are performed, especially in _209_db, _213_javac, and _228_jack.

One reason why lock operations are performed so frequently is that Java programs, especially its libraries, must be designed as *thread-safe* so that they run correctly even in a multi-threaded environment. For that reason, the methods for objects that *may* be shared among threads *must* be declared as `synchronized`, which leads to frequent lock operations [45]. For example, in Figure 2.1, locks for counter c_2 are actually unnecessary because it is incremented only by Thread C. However lock operations are also performed for that counter because the `add` method is declared as `synchronized` to make the class `MyCounter` thread-safe.

Another reason is that, in Java, lock operations can be specified very easily, by just declaring a method as `synchronized`. Therefore, programmers tend to use the declaration without considering deeply, even for methods that need not be `synchronized`. For example, in early versions of the Java libraries, we observed many unnecessary recursive locks which were caused by invoking a `synchronized` method from another `synchronized` method for the same object.

[2]Each program was run in the application mode, specifying the problem size as 100% and the number of executions as three, on the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1 [53], while turning off its JIT compiler to avoid lock elimination. See Table 5.1 for the description of each program.

2.3 Implementation of Java Locks

As shown in the previous section, lock operations are performed very frequently in Java, so the implementation of Java locks significantly affects the performance of Java programs. It was reported that 19% of the total execution time was consumed by thread synchronization in an early version of Java [6]. As a result, tremendous efforts have been devoted to optimizing Java locks.

Before showing the history of the improvement, this section explains how Java locks are implemented by using a naive example, which directly uses a synchronization mechanism provided by an operating system.

2.3.1 A Monitor Implementation for Java Locks

The OS-level synchronization mechanism for Java locks must provide an event notification mechanism among threads as well as the basic mutual exclusion, because Java locks support notification through the `wait` and `notify`(`All`) methods [41].

Figure 2.3 shows an example implementation of `monitor` for Java locks in pseudo-C code, which combines a mutual exclusion primitive (`os_mutex`) and a condition notification primitive (`os_cond`) provided by an operating system[3]. Here, we assume that the `os_mutex` mechanism is implemented by the operating system in cooperation with its thread scheduler, so the thread is suspended if it tries to acquire the `os_mutex` being held by another thread. We also assume that `os_mutex` supports recursive acquisition.

In the code, a thread[4] can enter a monitor by calling the `monitor_enter` function (line 6) with the address of the `monitor` structure (lines 1–4). At most one thread can enter the monitor simultaneously. When another thread is in the monitor, the thread calling `monitor_enter` is forced to wait until the already entered thread exits from the monitor by calling the `monitor_exit` function (line 11). However, it is possible for a thread to enter the same monitor recursively.

In addition, a thread can wait for an event notification by calling the `monitor_wait` function (line 17) while it is in the monitor. The thread implicitly exits from the monitor when it is suspended for waiting. The waiting thread will become runnable again when another thread calls the `monitor_notify` function (line 23) or the `monitor_notify_all` function (line 29) while entering the monitor. As their names imply, the former function notifies only one thread, while the latter notifies all threads waiting on the monitor. Note that the notified thread must implicitly enter the monitor again to resume execution. The notification functions do nothing if no thread is waiting on the monitor. No race hazard

[3]Operating systems which support threads usually provide functions corresponding to these primitives. For example, in a *Pthreads* [85] environment, `pthread_mutex` and `pthread_cond` can be used almost as they are defined.

[4]Note that threads in this subsection are *not* Java threads, but lower-level threads provided by the underlying operating system, which are used to construct Java threads.

```
 1 : typedef struct monitor {
 2 :     struct os_mutex mutex;
 3 :     struct os_cond  cond;
 4 : } monitor_t;
 5 :
 6 : int monitor_enter(monitor_t *mon) {
 7 :     os_mutex_lock(&mon->mutex); // block if the mutex is held by another thread
 8 :     return SUCCESS;             //         recursive monitor_enter is permitted
 9 : }
10 :
11 : int monitor_exit(monitor_t *mon) {
12 :     if (!os_mutex_owned(&mon->mutex)) return ILLEGAL_STATE;
13 :     os_mutex_unlock(&mon->mutex);
14 :     return SUCCESS;
15 : }
16 :
17 : int monitor_wait(monitor_t *mon) {
18 :     if (!os_mutex_owned(&mon->mutex)) return ILLEGAL_STATE;
19 :     os_cond_wait(&mon->cond, &mon->mutex); // wait until signaled
20 :     return SUCCESS;
21 : }
22 :
23 : int monitor_notify(monitor_t *mon) {
24 :     if (!os_mutex_owned(&mon->mutex)) return ILLEGAL_STATE;
25 :     os_cond_signal(&mon->cond); // restart a waiting thread
26 :     return SUCCESS;
27 : }
28 :
29 : int monitor_notify_all(monitor_t *mon) {
30 :     if (!os_mutex_owned(&mon->mutex)) return ILLEGAL_STATE;
31 :     os_cond_broadcast(&mon->cond); // restart all waiting threads
32 :     return SUCCESS;
33 : }
```

Figure 2.3: Monitor implementation by OS primitives

occurs here since both the `monitor_wait` and `monitor_notify` functions can be called only in the monitor.

If functions other than `monitor_enter` are called without entering the monitor, an `ILLEGAL_STATE` error is returned (lines 12, 18, 24, and 30).

In the remaining part of this thesis, these `monitor` functions will be used as the base synchronization primitives provided by the underlying operating system.

2.3.2 A Naive Implementation of Java Locks

The most naive implementation of Java locks is derived by directly using the `monitor` mechanism shown above. According to the Java specification, any object can be used for

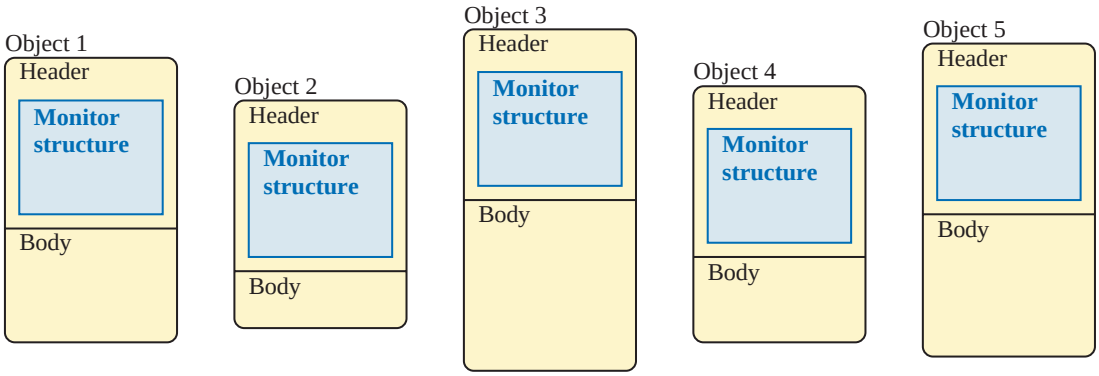

Figure 2.4: Data structure for the naive implementation

```
 1 : // Object header contains the OS-provided monitor structure
 2 : typedef struct object {
 3 :     //    :
 4 :     monitor_t mon; // initialized when the object is created
 5 :     //    :
 6 : } Object;
 7 :
 8 : // Current thread acquires the object's lock
 9 : int Java_lock_acquire(Object *obj) {
10 :     return monitor_enter(&obj->mon);
11 : }
12 :
13 : // Current thread releases the object's lock
14 : int Java_lock_release(Object *obj) {
15 :     return monitor_exit(&obj->mon);
16 : }
17 :
18 : // Current thread blocks until notified, the lock is released until woken up
19 : int Java_lock_wait(Object *obj) {
20 :     return monitor_wait(&obj->mon);
21 : }
22 :
23 : // Notify a waiting thread
24 : int Java_lock_notify(Object *obj) {
25 :     return monitor_notify(&obj->mon);
26 : }
27 :
28 : // Notify all waiting threads
29 : int Java_lock_notify_all(Object *obj) {
30 :     return monitor_notify_all(&obj->mon);
31 : }
```

Figure 2.5: Naive implementation of Java locks

a lock by specifying it as a target of a `synchronized` method or block. Therefore, this implementation prepares the `monitor` structure in every object's header[5], as shown in Figure 2.4. A similar approach was actually used in very first versions of the Kaffe [63] virtual machine.

Figure 2.5 shows the implementation in pseudo-C code. For executing a `synchronized` method, the JVM calls the `Java_lock_acquire` function (line 9) while specifying the address of the target object. This function calls `monitor_enter` with the address of the monitor structure in the object's header. The current thread is blocked if another thread is holding the object's lock.

When `Java_lock_acquire` returns, it means that the object's lock has been acquired by the current thread. The JVM then executes the code in the `synchronized` method, such as lines 6–7 of the `add` method in Figure 2.1. When it finishes, the JVM calls the `Java_lock_release` function (line 14) to release the object's lock. This function calls the `monitor_exit` function just as above.

Java supports event notification among threads through the `wait`, `notify`, and `notifyAll` methods of the `Object` class. Therefore, Java lock implementation must also provide functions for them. In the naive implementation of Figure 2.5, the `Java_lock_wait` (line 19), `Java_lock_notify` (line 24), and `Java_lock_notify_all` (line 29) are the functions for that purpose. The JVM calls these functions when corresponding Java methods are invoked, and then the corresponding `monitor` functions are called.

In functions other than `Java_lock_acquire`, if the current thread does not hold the specified object's lock, the `ILLEGAL_STATE` error is returned[6]. In this case, the JVM will generate and throw an `IllegalMonitorStateException` as defined in the Java specification [41].

The code shown here is a very naive example to explain how Java locks are implemented, so the performance is not good in many ways. The improvement of Java locks can be rephrased as to enhance the data structures for locks associated with objects and the five functions, `Java_lock_acquire`, `Java_lock_release`, `Java_lock_wait`, `Java_lock_notify`, and `Java_lock_notify_all`.

2.4 History of Java Lock Improvements

This section introduces several major techniques created to improve the performance of Java locks. In principle, these improvements are done by optimizing *common cases*, which are found from analyses of lock behavior in real Java programs.

[5]Usually, the internal structure of Java object consists of a *header* and a *body*. The body contains field values of the object, while the header contains meta-level information for the object, such as its class, various flags, information for GC, and data for lock processing.

[6]This occurs, for example, when the `Object.wait` method is invoked outside of the appropriate `synchronized` block.

Table 2.2: Number of objects used for locks in SPECjvm98

Program	Objects created	Objects used for locks	Ratio
_202_jess	23,999,733	21,278	0.089%
_201_compress	18,586	2,135	11.5%
_209_db	9,883,475	66,592	0.674%
_222_mpegaudio	26,456	1,620	6.12%
_228_jack	19,334,735	1,635,497	8.46%
_213_javac	19,140,558	1,192,734	6.23%
_227_mtrt	21,622,389	3,020	0.014%

2.4.1 Monitor Table

In the naive implementation example of Section 2.3.2, every object contains the monitor structure for lock processing in its header, as shown in Figure 2.4. This is because any object can be used for a lock in Java. However, in real Java programs, only a few objects are actually used as the target of lock operations. Table 2.2 shows the ratio of objects used for locks in the SPECjvm98 benchmark programs, measured in the same condition as in Table 2.1. From the table, it is revealed that at most only 10% of the total number of objects are actually used for lock processing.

The object's header is a very precious estate. From the viewpoint of memory usage, it is ineffective to prepare monitor structures for all objects, most of which are not used for locks. Therefore, a method was developed to allocate monitors separately from objects, and maintain the mapping from objects to monitors by a global *monitor table*.

Figure 2.6 illustrates the data structure of this method. The object's header contains no field for the lock, and the association between objects and monitor structures is maintained by a newly added table. Notice that no monitor structures are associated with Objects 1 and 4, since the structure is allocated and associated when an object is first used for a lock.

Figure 2.7 shows the implementation of the monitor table method in pseudo-C code, where the sidelines indicate the portions modified or added to the naive implementation shown in Figure 2.5. In the new code, each lock function first calls the `lookup_monitor` function (line 9) to get the monitor structure associated with the target object. This function first searches the table (line 12), and if the monitor structure is not found, it creates a new structure for the object and registers it into the table (lines 14–15).

Although this method improved the memory utilization efficiency because no lock-related field is necessary in an object's header, it forces each lock function to look up the monitor table. Since the table is a global resource in the JVM, some mutual exclusion is necessary to access it, which means that another lock must be acquired (line 11) for processing Java locks. Therefore, the method suffered from slow performance and bad

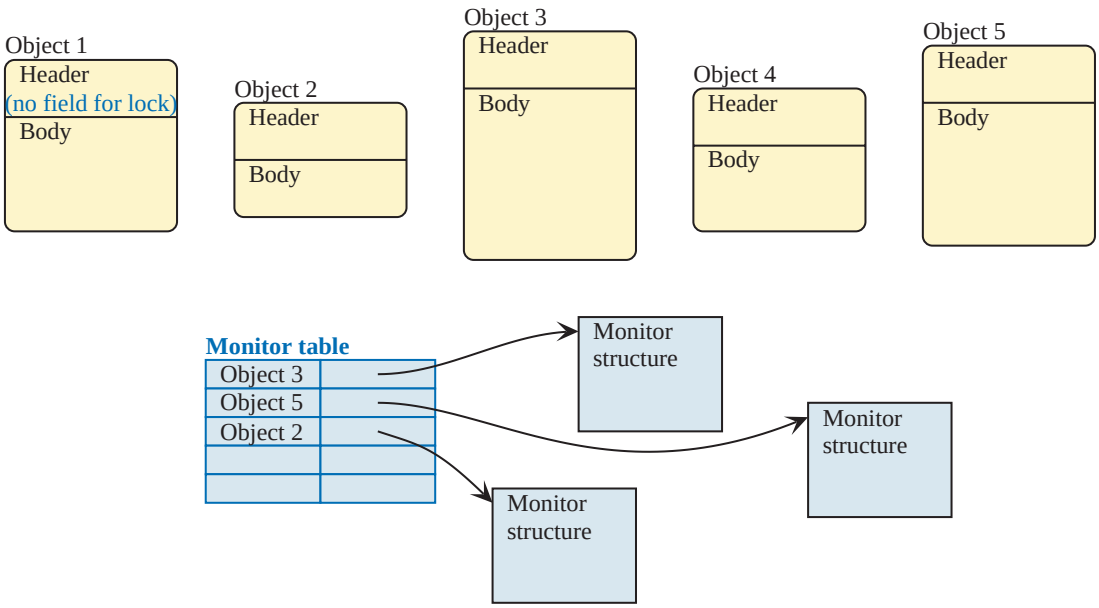

Figure 2.6: Data structure of a monitor table

```
    1 : // Object header does not contain a monitor structure
    2 : typedef struct object {
    3 :     //    :
|   4 :     // no field for lock
    5 :     //    :
    6 : } Object;
    7 :
    8 : // Returns a monitor associated with the object
|   9 : monitor_t *lookup_monitor(Object *obj) {
|  10 :     monitor_t *mon;
|  11 :     Global_lock_acquire();
|  12 :     mon = Search_a_monitor_from_the_table(obj);
|  13 :     if (mon == NULL) { // if not found
|  14 :         mon = Create_a_monitor();
|  15 :         Register_a_monitor_to_the_table(obj, mon);
|  16 :     }
|  17 :     Global_lock_release();
|  18 :     return mon;
|  19 : }
   20 :
   21 : int Java_lock_acquire(Object *obj) {
|  22 :     monitor_t *mon = lookup_monitor(obj);
   23 :     return monitor_enter(mon);
   24 : }
   25 :
   26 : int Java_lock_release(Object *obj) {
|  27 :     monitor_t *mon = lookup_monitor(obj);
   28 :     return monitor_exit(mon);
   29 : }
   30 :
   31 : // Java_lock_wait()/notify()/notify_all() are implemented in the same manner
```

Figure 2.7: Java locks using a monitor table

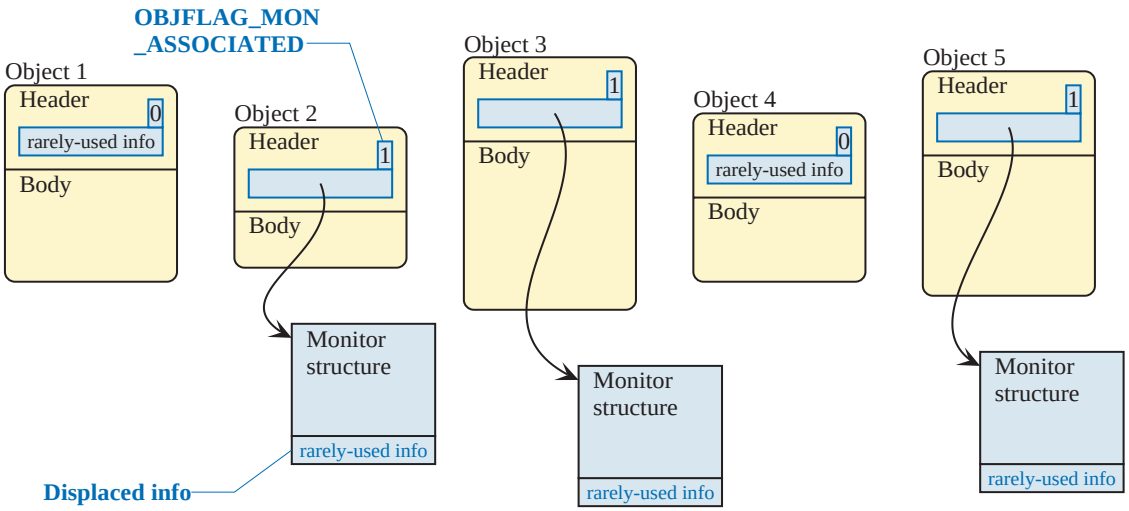

Figure 2.8: Data structure for the direct pointing method

scalability [3, 43].

The early versions of Java virtual machines from Sun, for example JDK 1.1.6, used a variation of this monitor table method, with an optimization technique called a *monitor cache* [123]. This technique reduced the number of accesses to the global monitor table by preparing small per-thread caches to hold the recent results of `lookup_monitor` by each thread.

2.4.2 Direct Pointing by Data Displacement

One technique to eliminate the table reference overhead is a *direct pointing* method by using header word displacement [87]. When an object is first used for a lock, the method directly stores a reference to a monitor structure into a rarely used field of the object's header, displacing the original value of the field into the monitor structure.

Figure 2.8 illustrates the data structure for this method. The global monitor table in Figure 2.6 has been eliminated, and the monitor structures are directly pointed at from the objects' headers. A one-bit flag named `OBJFLAG_MON_ASSOCIATED` is prepared in the object's header to indicate whether or not the monitor structure is ready for the object. For Objects 1 and 4, the flag is not set and monitor structures have not been prepared, since the objects have not yet been used for locks.

Figure 2.9 shows the implementation in pseudo-C code, where the sidelines indicate the portions modified from the monitor table code in Figure 2.7. To get the corresponding monitor structure from an object, the `get_monitor` function (lines 18–34) is prepared by replacing the `lookup_monitor` function. By using this new function, when an object is first used for a lock, a monitor structure is created and its address is stored in the `rarely_used_info` field in the object's header while setting the `OBJFLAG_MON_ASSOCIATED` flag

```
    1 : // Object header contains a rarely_used_info, which may point to a monitor
    2 : typedef struct object {
    3 :     //    :
|   4 :     volatile unsigned int flag; // 1 bit used to show the next field's usage
|   5 :     int rarely_used_info;       // may contain a pointer to a monitor struct
    6 :     //    :
    7 : } Object;
    8 :
    9 : // Now the monitor structure has an extra field to hold the displaced data
   10 : typedef struct monitor {
   11 :     //    :
   12 :     // original monitor fields
   13 :     //    :
|  14 :     int displaced_info; // additional field to store the displaced info
   15 : } monitor_t;
   16 :
   17 : // Returns a monitor for the object
|  18 : monitor_t *get_monitor(Object *obj) {
|  19 :     if (obj->flag & OBJFLAG_MON_ASSOCIATED)
|  20 :         return (monitor_t *)obj->rarely_used_info; // fast path
|  21 :     // create a monitor and store its address to the object's header
   22 :     monitor_t *mon;
   23 :     Global_lock_acquire();
|  24 :     if (obj->flag & OBJFLAG_MON_ASSOCIATED)
|  25 :         mon = (monitor_t *)obj->rarely_used_info;
|  26 :     else {
   27 :         mon = Create_a_monitor();
|  28 :         mon->displaced_info = obj->rarely_used_info;
|  29 :         obj->rarely_used_info = (int)mon;
|  30 :         obj->flag |= OBJFLAG_MON_ASSOCIATED;
|  31 :     }
   32 :     Global_lock_release();
   33 :     return mon;
   34 : }
   35 :
   36 : int Java_lock_acquire(Object *obj) {
   37 :     monitor_t *mon = get_monitor(obj);
   38 :     return monitor_enter(mon);
   39 : }
   40 :
   41 : int Java_lock_release(Object *obj) {
   42 :     monitor_t *mon = get_monitor(obj);
   43 :     return monitor_exit(mon);
   44 : }
   45 :
   46 : // Java_lock_wait()/notify()/notify_all() are implemented in the same manner
```

Figure 2.9: Java locks using the direct pointing method

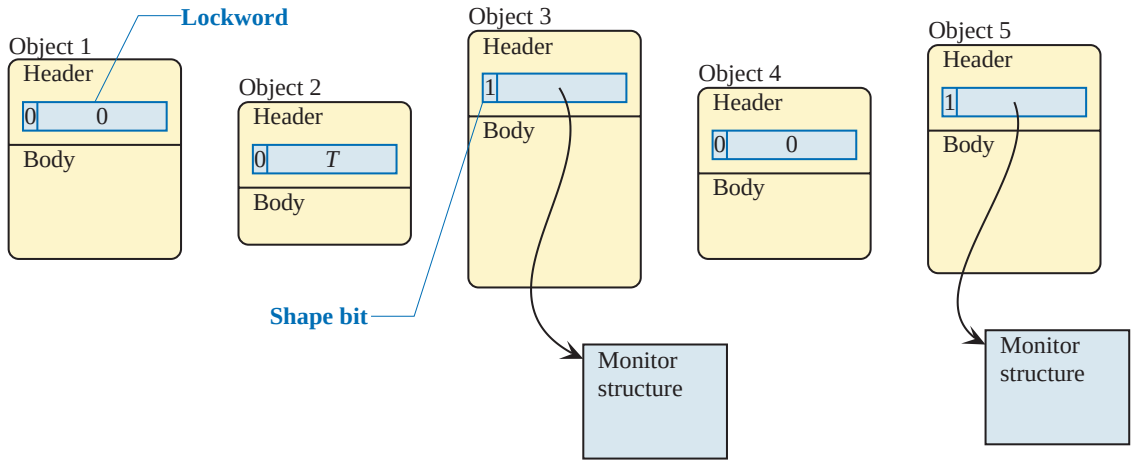

Figure 2.10: Data structure for thin lock

(lines 29–30). Just before this, the data which was originally stored in the field is moved to the `displaced_info` field added to the `monitor` structure (line 28). As for the data displaced from the header to the monitor structure, one candidate is a hash code for the object, which is a value unique to each object returned by an `Object.hashCode` method.

These monitor preparation processes must be done while acquiring a global lock (line 23). However, once done, the monitor can be got from the object very quickly (lines 19–20), without referring to a table or acquiring the global lock.

The direct pointing method improved the scalability since it does not require a global monitor table. However, this could not improve the lock performance too much, since OS-provided heavyweight monitors were still used in all cases.

2.4.3 Thin Lock

All of the lock methods explained above use a synchronization mechanism provided by the underlying operating systems, such as the one shown in Section 2.3.1, for the lock processing itself. Therefore, the overhead of Java locks was significant because OS functions are called for every lock operation. In 1998, Bacon, Konuru, Murthy, and Serrano proposed an excellent lock algorithm called *thin lock*[7] [11], exploiting the observation that most locks are not contended in Java.

As shown in Figure 2.10, the thin lock reserves one word[8] in each object's header, the *lockword*. The lockword contains a mode flag called the *shape bit*, which distinguishes

[7]In the original paper, they used the plural form, *thin locks*. However in this thesis, we mainly use the singular form, to express the thin lock algorithm itself.

[8]Actually, the whole 32 bits are not necessary for the algorithm. They reserved a 24-bit field where remaining 8 bits are used for miscellaneous flags that are not directly related to lock operations. In addition, they did not reserve the field by increasing the size of an object. In the base JVM they used,

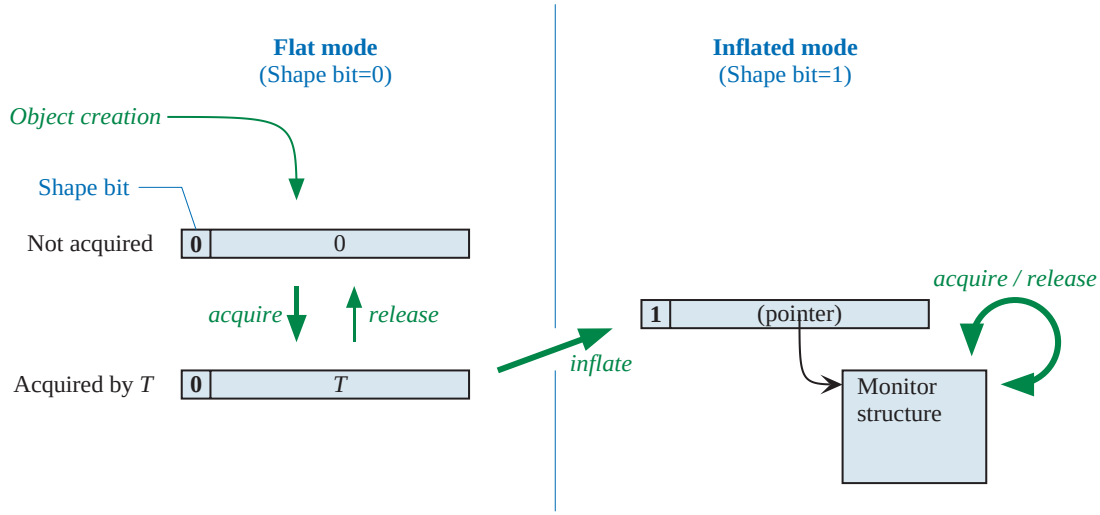

Figure 2.11: Lockword state transitions of thin lock

between the following two modes. When the shape bit is zero, the lockword is in the *flat* mode. Otherwise, it is in the *inflated* mode. This *bimodal use* of the lockword is the most important feature of the thin lock.

The meanings of lockword in each mode and state transitions among them are shown in Figure 2.11, where the thickness of arrows indicates the relative cost of the transition. As long as contention does not occur, the lockword is in the flat mode. In this mode, the lock can be acquired by setting the current thread's identifier to the lockword with a `compare_and_swap`[9], which is an *atomic operation* to perform a comparison and store of memory data, as shown in Figure 2.12.

The monitor structure is first prepared when a thread attempts to acquire a lock held by another thread, that is, when the lock is contended. As in the direct pointing method in Section 2.4.2, the address of the monitor is stored in the lockword. At this time, the shape bit is set and the lockword becomes to the inflated mode. This mode transition is called *inflation*. In the inflated mode, lock operations are performed by calling the `monitor` functions, as in other lock methods explained above.

In the case of Figure 2.10, the lockwords of Objects 3 and 5 are already inflated, and lock operations for these objects are performed by using the monitor structures pointed at through the lockwords. In contrast, the lockword of Object 2 is in flat mode, even though it is currently held by Thread T. This is because no contention has yet occurred on the object, so lock operations can be performed very quickly by using `compare_and_swap`.

---

an object's header included a field for storing the object's hash code. A compression technique allows the field to be reduced to two bits, making 24 bits free and available to their algorithm. The technique was independently invented by them and Agesen [2].

[9]Since the swapped original value is not used in our definition, the operation should have been called `compare_and_store`.

```
1 : int compare_and_swap(volatile unsigned int *addr,
2 :                      unsigned int oldval, unsigned int newval) {
3 :     // The following is performed atomically
4 :     if (*addr == oldval) { *addr = newval; return SUCCESS; }
5 :     else return FAILURE;
6 : }
```

Figure 2.12: Semantics of `compare_and_swap` as used in this thesis

Figure 2.13 shows the implementation in pseudo-C code, where the sidelines indicate the portions added for the thin lock. Note that the support of shallowly nested lock acquisitions is omitted from this code[10] to simplify the explanation.

The lockword is initially set to zero, which means that it is in the flat mode and not acquired. The `Java_lock_acquire` function first attempts to set the current thread's identifier, which can be got by `thread_id`, to the lockword by using `compare_and_swap` (lines 12–13). This is the fastest, and most common case in this method. If the attempt fails, it means that the flat-mode lock is being held by another thread or the lockword is already inflated.

In the former case, the lockword is first inflated, which is performed by acquiring the flat-mode lock in a busy-wait loop (lines 15–20), and then storing the address of the created monitor in the lockword while setting its shape bit (line 30). If the lockword is in the inflated mode, the lock acquisition is performed by the `monitor_enter` function (line 23) as in other lock methods. At this time, the address of monitor structure can easily be extracted from the lockword (line 22), without referring to a table or acquiring a global lock, as in the direct pointing method.

The `Java_lock_release` function first checks the mode of the lockword (line 35). The flat-mode lock can be released by simply storing zero into the lockword (line 37), while the inflated-mode lock is released by calling the `monitor_exit` function (line 41).

The `Java_lock_wait` function requires an event notification mechanism which is provided through the monitor structure. Therefore if the lock is in the flat mode, the function first prepares the monitor structure by forcing the inflation (lines 46–49).

By using the thin lock, OS-provided heavyweight monitors became unnecessary in most lock operations, so the performance of Java applications was greatly improved. The original paper [11] reported a maximum speedup of 1.7 in realistic applications.

This subsection has given an overview of the thin lock from the viewpoint of the history of Java lock improvements. We will revisit the thin lock in Chapter 4 as the base algorithm for our Java lock accelerations, showing its philosophy, implementation techniques, and issues.

[10] The actual thin lock allocates 8 bits of the lockword in flat mode for counting the number of recursive acquisitions.

```
    1 : // Object header contains a word for lock
    2 : typedef struct object {
    3 :     //    :
|   4 :     volatile unsigned int lockword;
    5 :     //    :
    6 : } Object;
    7 :
|   8 : #define SHAPE_BIT 0x80000000
    9 :
   10 : int Java_lock_acquire(Object *obj) {
   11 :     // flat lock path
|  12 :     if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS)
|  13 :         return SUCCESS;
   14 :     // inflation loop
|  15 :     while ((obj->lockword & SHAPE_BIT) == 0) {
|  16 :         if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS) {
|  17 :             inflate(obj); return SUCCESS;
|  18 :         }
|  19 :         thread_yield();
|  20 :     }
   21 :     // inflated lock path
   22 :     monitor_t *mon = (monitor_t *)(obj->lockword & ~SHAPE_BIT);
   23 :     return monitor_enter(mon);
   24 : }
   25 :
   26 : // Inflate the object's lockword, which is held by current thread
|  27 : void inflate(Object *obj) {
|  28 :     monitor_t *mon = Create_a_monitor(); // assume that the MSB is 0
|  29 :     monitor_enter(mon);
|  30 :     obj->lockword = SHAPE_BIT | (unsigned int)mon; // set the shape bit
|  31 : }
   32 :
   33 : int Java_lock_release(Object *obj) {
   34 :     unsigned int lw = obj->lockword;
|  35 :     if ((lw & SHAPE_BIT) == 0) { // flat lock path
|  36 :         if (lw != thread_id()) return ILLEGAL_STATE;
|  37 :         obj->lockword = 0; return SUCCESS;
|  38 :     }
   39 :     // inflated lock path
   40 :     monitor_t *mon = (monitor_t *)(lw & ~SHAPE_BIT);
   41 :     return monitor_exit(mon);
   42 : }
   43 :
```

Figure 2.13: Thin lock (simplified version) (1 of 2)

```
   44 : int Java_lock_wait(Object *obj) {
   45 :     unsigned int lw = obj->lockword;
|  46 :     if ((lw & SHAPE_BIT) == 0) { // flat mode
|  47 :         if (lw != thread_id()) return ILLEGAL_STATE;
|  48 :         inflate(obj); // force the inflation
|  49 :     }
   50 :     // execute the wait using the monitor structure
   51 :     monitor_t *mon = (monitor_t *)(obj->lockword & ~SHAPE_BIT);
   52 :     return monitor_wait(mon);
   53 : }
   54 :
   55 : // Java_lock_notify()/notify_all() are implemented in the same manner
```

Figure 2.13: Thin lock (simplified version) (2 of 2)

2.5 Summary

This chapter described the basic usage and several major implementations of Java locks. To implement Java locks, we must prepare the data structures for the locks associated with the objects and the five functions: `Java_lock_acquire`, `Java_lock_release`, `Java_lock_wait`, `Java_lock_notify`, and `Java_lock_notify_all`.

Among the many implementations of Java locks proposed so far, the thin lock by Bacon et al. was an epoch-making method, by which OS-provided heavyweight monitors became unnecessary for most lock operations.

Chapter 3

Problem Definition

3.1 Introduction

In Chapter 2, we showed that fast lock algorithms are one of the key components to improve the performance of Java applications, and explained several major techniques to accelerate Java locks. However, we think there is still room for improvement. The goal of our research is to accelerate Java locks to the greatest extent theoretically possible, following upon the efforts for acceleration shown in the previous chapter.

In this chapter, we first analyze the approaches of prior lock improvements by positioning them into a chart. We then define problems to be solved in this thesis for achieving the ultimate goal. Possible directions for the solution are also briefly mentioned.

3.2 Problems to Solve

Through the research shown in the previous chapter, Java locks have become considerably faster than those in the early Java implementations. We noticed that each of these research projects was done by first discovering a characteristic of Java locks from the analysis of real applications, and then reducing the cost in that typical-use pattern, which is the most *common case* related to the characteristic. Figure 3.1 positioned the characteristics utilized in prior lock algorithms in a chart. All of the objects created for a Java application are illustrated in the figure since locks are associated with objects in Java.

The first characteristic of Java locks is that only a few objects are actually used for locks, although any objects can theoretically be used. The monitor table method introduced in Section 2.4.1 utilized this characteristic to reduce the memory consumption. In this method, monitor structures are not prepared for objects that are not used for locks, which is the most common case. When an object is first used for lock, a monitor structure is prepared and associated with the object.

The direct pointing method in Section 2.4.2 is considered to be a variation of utilizing

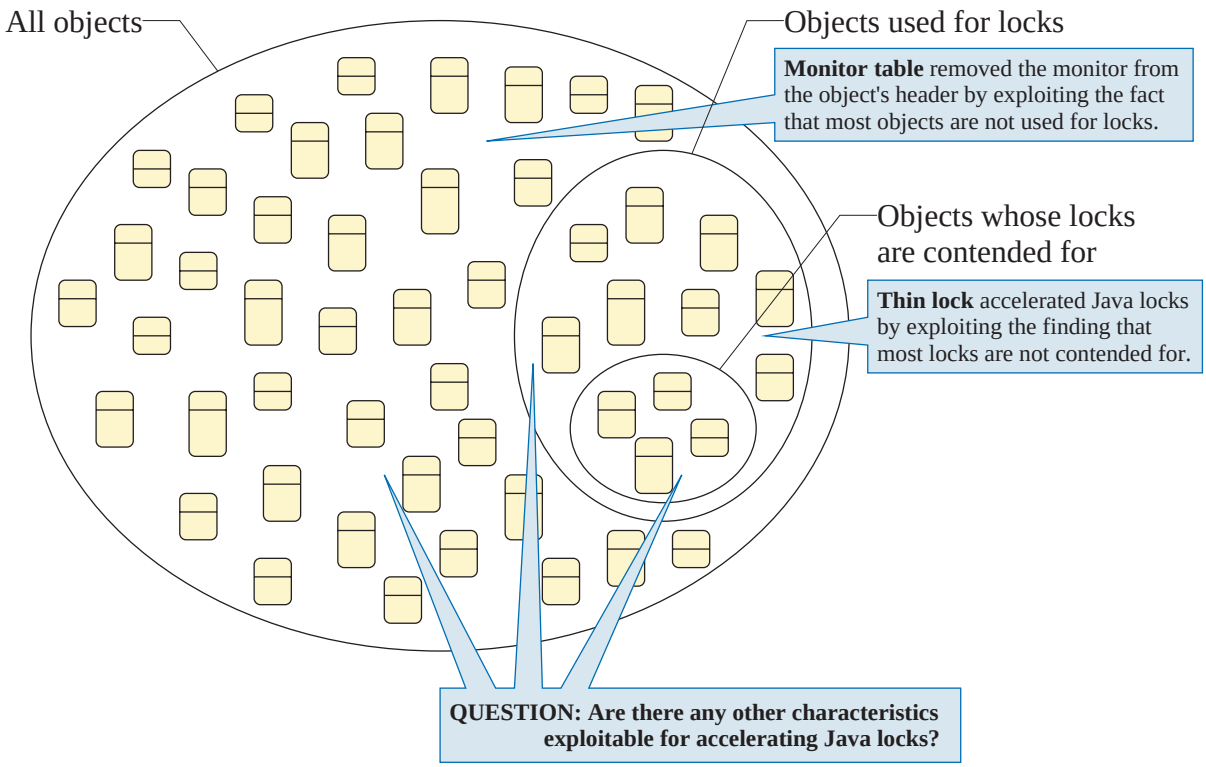

Figure 3.1: Characteristics of Java locks and lock methods exploiting them

this characteristic, which also tried to improve the speed and scalability by pointing at the prepared monitor directly from the object's header.

The second characteristic shown in Figure 3.1 is that contention between threads never occurs for most of the objects used for locks. This fact implies that the heavyweight monitor structure which supports thread suspension is unnecessary for the most common case. The thin lock explained in Section 2.4.3 accelerated this case by using a simple `compare_and_swap` until contention actually occurs for the object.

With the thin lock, the performance of Java locks was drastically improved. However, we think there is still room for further improvement. To show an example, we modified a JVM to do nothing at the time of lock processing[1]. When `_209_db` in the SPECjvm98 benchmark [100] was executed by the modified JVM on a Linux PC containing dual 500 MHz Pentium III processors, its execution time was reduced by about 12% compared to the original JVM, which uses a lock mechanism based on the thin lock. That is, even with thin locks there still exist cases with large lock overheads.

The goal of our research in this thesis is to accelerate Java locks to the highest de-

[1]Of course, since necessary mutual exclusions are also not performed, Java applications may run incorrectly on the modified JVM. The example shown here is a single thread application, so it ran correctly.

gree possible by removing the remaining overhead. The first problem which should be addressed for that purpose is the same as in previous research:

Problem 1: Discovering new characteristics of Java locks.

One possible starting point for this is the investigation of the area not yet analyzed in Figure 3.1, such as the behavior of the contended objects.

However, the finding of new characteristics does not directly lead to the lock acceleration. To achieve this final target, it is necessary to design a lock algorithm which can accelerate the most common cases with the newly found characteristic. Therefore, we must also solve the more challenging problem of:

Problem 2: Designing new lock methods which can exploit the characteristics, and evaluating them.

The construction of a lock method must be done with the greatest care. As well as the correctness as a lock, attention must also be paid to avoid degrading the performance. Therefore, the third problem to be focused on is:

Problem 3: Showing a technique to construct Java locks more easily.

Specifically, a construction technique that can exploit the new characteristic while incorporating existing acceleration methods is required.

3.3 Approach

As a concrete approach to solve the problems, we consider here possible improvements for the thin lock.

First of all, in the thin lock, once a lock is inflated, it is processed after that by a slow method using a monitor structure. This is because it is not very easy to implement *deflation*, a transition from the inflated mode which uses the monitor to the flat mode which uses `compare_and_swap`. Another reason is that they considered it better to keep contended objects in the inflated mode, because they assume such objects tend to be contended for continuously. One possible starting point for our research is the confirmation of this assumption by investigating the behavior of the contended objects in Figure 3.1.

In addition, the thin lock implementation uses a busy-wait loop for inflating an object's lock (lines 15–20 in Figure 2.13). It should also be checked that this will not cause performance degradation. These topics will be examined in the next chapter.

By using the thin lock, the cost of lock acquisition has been reduced to one `compare_and_swap` operation. However, such an atomic operation is usually very costly compared to simple memory access operations. If we can acquire a lock without any atomic operations for common cases, the cost of locks is further reduced. To pursue the possibility, the area

of non-contended objects in Figure 3.1 should also be investigated to find any exploitable characteristics. This topic will be examined in Chapter 5.

As for the lock construction techniques, we will show a method of adding a new feature to the existing lock method in Chapter 5. In addition, in Chapter 6, we will attempt a new approach of replacing the `compare_and_swap`, which is one primitive for constructing Java locks, with another spin lock.

3.4 Summary

In this chapter, we discussed the problems to be solved in this thesis. To achieve the goal of the thesis, accelerating Java locks to the highest degree possible, we must solve the problems of (1) finding new characteristics of Java locks that are (2) exploitable by new lock algorithms to reduce the lock overhead. The new algorithms should (3) coexist with existing accelerating methods, while making the best use of the new characteristics and improving the performance.

To solve the problems, we will further analyze the behavior of Java locks by first focusing on the contended locks in Figure 3.1. The area of non-contended locks in the figure will also be analyzed later.

Chapter 4

Tasuki Lock

4.1 Introduction

In this chapter, we will introduce our first improvement of Java locks, named the *tasuki*[1] *lock* [88]. The tasuki lock supports the deflation of a lockword to exploit the newly-identified characteristic of Java locks that most contentions are temporary.

We will first revisit the thin lock, and identify potential issues in its bimodal use of the lockword. Next, we analyze the synchronization behavior of real multi-threaded Java programs, especially focusing on contended objects. The result indicates that busy-wait and absence of deflation can be problematic in terms of robustness and performance. Based on the analysis, we propose a new algorithm, tasuki lock, that allows both inflation without busy-wait and deflation, but still maintains an almost maximum level of performance in the absence of contention. Finally, we evaluate an implementation of our algorithm in IBM's production Java virtual machine. The measurements show not only increased robustness but also performance improvements of up to 13.1% in server-oriented benchmarks.

The remaining sections in this chapter are organized as follows. Section 4.2 introduces the concept of spin and suspend locks, and Java locks using a bimodal lockword. Section 4.3 revisits the base bimodal lock algorithm, which is a simplified version of thin lock. Section 4.4 presents measurements for multi-threaded Java applications, especially focusing on the contended locks. Section 4.5 describes our bimodal lock algorithm in detail, and Section 4.6 shows performance results of an implementation of our algorithm. Finally, Section 4.7 gives the summary of this chapter.

[1] In Japan, a *tasuki* is worn for tucking up sleeves, resulting in the shape of the letter 'x' on the back. As we will see later in Figure 4.9, the most important characteristic of our algorithm is that the shape is formed by the write/read dependency arrows of a lock field and a status bit.

```
 1 : int spin_acquire(volatile unsigned int *lock) {
 2 :     while (compare_and_swap(lock, 0, thread_id()) != SUCCESS)
 3 :         continue; // spin until success
 4 :     return SUCCESS;
 5 : }
 6 :
 7 : int spin_release(volatile unsigned int *lock) {
 8 :     *lock = 0;
 9 :     return SUCCESS;
10 : }
```

Figure 4.1: Simple spin lock using `compare_and_swap`

4.2 Java Locks with Bimodal Fields

Even before the birth of Java, the field of thread synchronization (or process synchronization) had been studied both long and deeply. As a result, most modern computer systems provide two classes of locks for system programmers, *spin* and *suspend* locks. A spin lock is realized with a memory word, by repeatedly performing a test (spinning) against the word with such *atomic* primitives as `test_and_set`, `fetch_and_add`, or `compare_and_swap`. Figure 4.1 shows an example of a spin lock using `compare_and_swap`[2]. A thread can acquire the spin lock by setting its identifier to the memory word pointed at by `lock`.

On the other hand, suspend locks, typical instances of which are semaphores [29] and monitors [20, 21, 46], support the suspension of threads when they cannot acquire locks. Therefore, the suspend locks must be integrated with an operating system's scheduler, and thus implemented within the kernel space, and so are heavier than the spin locks. The monitor code in Figure 2.3 is one example of suspend locks.

A well-known optimization of suspend locks, first suggested by Ousterhout, is to combine the technique with spin locks in the user space [90]. In acquiring this *spin-suspend lock*, a thread first tries to acquire the spin lock, but only a small number of times (possibly only once). If it fails to grab the spin lock, the thread then attempts to obtain the actual suspend lock, and goes down to the kernel space. As a result, when there is no contention, synchronization requires just one or a few atomic primitives to be executed in the user space, and is therefore very fast.

In Java, locks are performed by specifying objects. The main concern here is how associations between objects and locks are realized. A simple way is to maintain them in a hash table that maps an object's address to its associated lock as in the monitor table method described in Section 2.4.1. This approach is space-efficient, since it does not require any working memory in an object's header. However, the runtime overhead is prohibitive, because the monitor table is a shared resource and every access must be synchronized.

As briefly explained in Section 2.4.3, Bacon et al. proposed an efficient lock algorithm

[2]The semantics of the `compare_and_swap` function was shown in Figure 2.12.

for Java, called *thin lock* [11], which optimizes common cases by combining spin locks and suspend locks as in spin-suspend locks. However, what is really intriguing is that the algorithm reserves a word (actually a 24-bit field) in an object's header and makes *bimodal use* of the single field. This was the beginning of *bimodal locks* for Java.

Initially, the field is in the spin-lock mode (flat mode), and remains in this mode as long as contention does not occur. When contention is detected, the field is put into the suspend-lock mode (inflated mode), and the reference to a monitor structure for the suspend lock is stored into the field. In this way, the algorithm achieves the highest possible performance in the absence of contention, which they found is the most common case in Java locks.

Actually, their lock algorithm requires an atomic `compare_and_swap` operation only in acquiring the lock, not in releasing the lock, whereas many other algorithms perform two atomic operations, one in acquiring and the other in releasing [3]. This is why we say that the algorithm achieves the highest performance.

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

4.3 The Base Algorithm and Its Issues

This section describes the base lock algorithm for Java we try to improve. It is a simplified version of the thin lock [11], an overview of which was given in Section 2.4.3. We simplify their original algorithm in order to concentrate on those portions relevant to mode transitions. We start by revisiting the thin lock, derive a simplified version, and discuss issues inherent in bimodal locks.

4.3.1 Thin Lock Revisited

The thin lock was the first bimodal lock algorithm to allow a highly efficient implementation of Java locks. It was deployed across many IBM versions of the JDK, including the IBM Developer Kits, Java Technology Edition [53], for AIX, Windows, Linux, OS/2, and OS/390. The important characteristics are:

Space It assumes that a 24-bit field (*lockword*) is available in each object's header for implementing locks, and makes bimodal use of the lockword, with one mode for spin lock and the other for suspend lock. These modes are distinguished by a *shape bit* in the lockword.

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

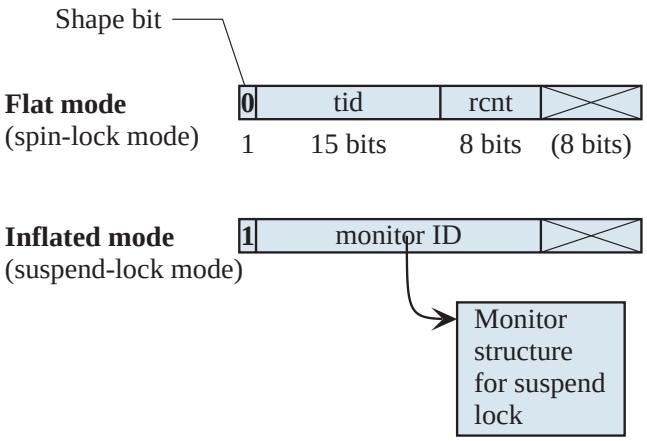

Figure 4.2: Bimodal use of a lockword in the actual implementation of thin lock

Software Environment It assumes that the underlying layer provides suspend locks with the full range of Java synchronization semantics. The (heavyweight) monitor code shown in Section 2.3.1 is an example of such suspend locks.

Hardware Support As necessary hardware support, it only assumes the existence of a `compare_and_swap` operation executable in user mode. Almost all modern processors, including Pentium [55] and PowerPC [49, 79], support the operation as a primitive instruction or as a sequence of instructions.

Like spin-suspend locks, the thin lock optimizes common, uncontended, cases by performing hardware-supported atomic operations against a lockword. However, it optimizes not only the most common case — that of acquiring a lock which is not currently acquired (outermost acquisition) — but also the second most common case — that of acquiring a lock which was already acquired by the same thread a small number of times (shallow recursive acquisition). It does so by carefully engineering the structure of its lockword, specifically by using 8 bits of lockword in the spin-lock mode as a *recursion count*.

Figure 4.2 shows the structure of 24-bit lockword actually used in the thin lock. In the flat mode where the shape bit is 0, the 15-bit `tid` field is used to hold the thread identifier currently acquiring the spin lock, and the 8-bit `rcnt` field is used to hold the count of recursive spin-lock acquisitions. The rightmost 8 bits are used for miscellaneous flags that are not directly related to lock operations.

4.3.2 The Base Algorithm

The base algorithm we use in this chapter is a simplified version of the original thin lock in two respects:

- Extension of the lockword to make it a full word.

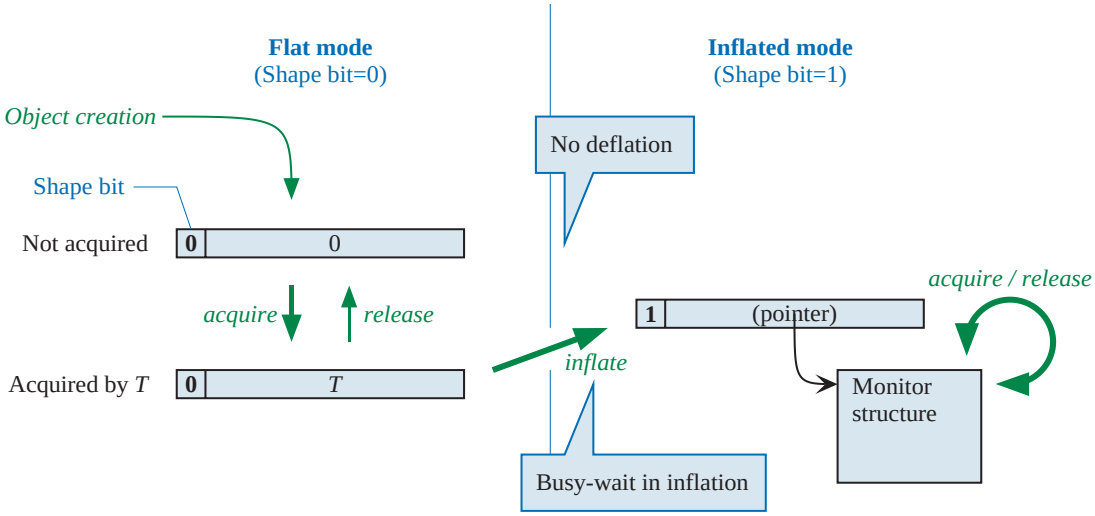

Figure 4.3: Lockword structure and its state transitions of the base algorithm

- Omission of the optimization for shallowly nested lock acquisitions.

Except in these two respects, the base algorithm shares all of the important characteristics of the thin lock. Note that both of the simplifications are purely for the sake of explanation. Our actual implementation, which we will describe in Section 4.5, uses the same number of bits, 24 bits, for the lockword, and performs the same level of optimization for shallowly nested lock acquisitions.

We now present the base lock algorithm. First, the lockword has one of two structures: a *flat* structure for spin-lock mode and an *inflated* structure for suspend-lock mode. These structures are distinguished by the shape bit. Figure 4.3 reshows the structure of the lockword and its state transitions. The left part illustrates the lockword structure in flat mode. The value of the shape bit is 0, and the remaining bits hold a thread identifier. The value of the lockword is 0 if no thread is holding the lock. Otherwise, the value is the identifier of the thread currently holding the lock.

On the other hand, the inflated structure contains a shape bit with a value of 1 and a monitor pointer (or monitor identifier in the actual implementation), as in the right part of Figure 4.3. Notice that, when the lockword of an object is in the inflated mode, a thread may or may not be holding the lock. That depends on the internal status of the monitor structure referred to through the inflated lockword.

When an object is being created, the lockword is initialized to 0, which indicates that it is in the flat mode and has not been acquired. Because of the lockword structure, the thread and monitor identifier must be less than 2^{31} in the base algorithm[3].

[3]In addition, the value 0 is not permitted as the thread identifier since it means that the lock is not acquired. In the actual implementation, whose lockword structure is shown in Figure 4.2, the thread identifier must be less than 2^{15} and the monitor identifier must be less than 2^{23}.

```
 1 : // Object header contains a word for lock
 2 : typedef struct object {
 3 :     //    :
 4 :     volatile unsigned int lockword;
 5 :     //    :
 6 : } Object;
 7 :
 8 : #define SHAPE_BIT 0x80000000
 9 :
10 : int Java_lock_acquire(Object *obj) {
11 :     // flat lock path
12 :     if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS)
13 :         return SUCCESS;
14 :     // inflation loop
15 :     while ((obj->lockword & SHAPE_BIT) == 0) {
16 :         if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS) {
17 :             inflate(obj); return SUCCESS;
18 :         }
19 :         thread_yield();
20 :     }
21 :     // inflated lock path
22 :     monitor_t *mon = (monitor_t *)(obj->lockword & ~SHAPE_BIT);
23 :     return monitor_enter(mon);
24 : }
25 :
26 : // Inflate the object's lockword, which is held by current thread
27 : void inflate(Object *obj) {
28 :     monitor_t *mon = Create_a_monitor(); // assume that the MSB is 0
29 :     monitor_enter(mon);
30 :     obj->lockword = SHAPE_BIT | (unsigned int)mon; // set the shape bit
31 : }
32 :
33 : int Java_lock_release(Object *obj) {
34 :     unsigned int lw = obj->lockword;
35 :     if ((lw & SHAPE_BIT) == 0) { // flat lock path
36 :         if (lw != thread_id()) return ILLEGAL_STATE;
37 :         obj->lockword = 0; return SUCCESS;
38 :     }
39 :     // inflated lock path
40 :     monitor_t *mon = (monitor_t *)(lw & ~SHAPE_BIT);
41 :     return monitor_exit(mon);
42 : }
43 :
44 : // Java_lock_wait()/notify()/notify_all() are omitted
```

Figure 4.4: Base lock algorithm: a simplified version of the thin lock

Lock Acquisition

Figure 4.4 shows the base algorithm, which is actually the same as that shown in Figure 2.13. In acquiring an object's lock, the thread first attempts to acquire a flat lock by using a `compare_and_swap` operation (line 12). If the `compare_and_swap` succeeds, the lockword was 0 (in the flat mode and not acquired), and becomes the current thread's identifier (acquired by the current thread). This is the fastest, and most common path in acquiring the lock.

If the `compare_and_swap` fails, there are two possibilities. In one case, the lockword is already in the inflated mode. The conditional in the `while` loop (line 15) immediately fails, and the current thread attempts to enter the object's monitor (line 23). In the other case, the lockword is in the flat mode but the object's lock is being held by some other thread. This means that *flat lock contention* is occurring. The thread then enters the *inflation loop*, which will be explained below.

Lock Release

In releasing an object's lock, the current thread first looks at the lockword to determine if it is in the flat mode (line 35). If so, the current thread releases the flat lock by simply storing 0 into the lockword (line 37). This is the fastest path in releasing the lock. Otherwise, it takes the object's monitor out of the lockword, and exits from the monitor (line 41).

Notice that the lock release in the flat mode does not require any atomic operations. This is because an important discipline is imposed on the algorithm that only the thread currently holding an object's lock can modify the lockword, except for the initial acquisition by `compare_and_swap`. Hence, the algorithm achieves the highest performance in the absence of contention: one `compare_and_swap` in acquiring a lock, and a one bit-test followed by one assignment in releasing the lock.

Inflation

When a thread detects flat lock contention, the thread enters the inflation loop (lines 15–20) in the `Java_lock_acquire` function. In the loop, it performs the same `compare_and_swap` operation in order to acquire the flat lock. The inflating thread needs to do so before modifying the lockword, because of the above-mentioned discipline.

If the `compare_and_swap` succeeds, the current thread calls the `inflate` function (lines 27–31). This function creates a monitor for the object, enters the monitor, and stores into the lockword the monitor's identifier (address in this simplified implementation) with a shape bit whose value is 1. If the `compare_and_swap` fails, the thread needs to busy-wait until it detects that some other thread has completed inflation or until it succeeds in the `compare_and_swap` in the inflation loop.

Deflation

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

Actually, deflation is difficult to realize in this algorithm. For instance, storing 0 in the lockword just before calling `monitor_exit` in the `Java_lock_release` function (line 41) does not work. This actually results in a loss of the correct lock behavior. What may happen is that, while one thread acquires the suspend lock via the inflated lock path in the `Java_lock_acquire` function (line 23), another thread could simultaneously succeed in the initial `compare_and_swap` (line 12).

4.3.3 Inherent Issues

Inflation and deflation are unique to bimodal locks. The thin lock inflates the lockwords by busy-waiting, and never deflate them, as we have seen in the base algorithm. We summarize and consider the reasons and justifications given by the authors of the thin lock [11] for their approach to mode transition.

First, they do not deflate lockwords for two reasons. One is the *locality of contention*, which states that, if contention occurs once for an object, it is likely to occur again for that object. This implies that, if we perform deflation, it will likely cause lockwords to thrash between the flat and inflated modes. The other reason is that the absence of deflation considerably simplifies the implementation.

However, the locality of contention was not verified. The real programs they measured were single-threaded applications, which obviously resulted in no contention[4]. As we will see, our measurements of real multi-threaded programs show that locality of contention does not exist in many more cases than one might expect. On the other hand, the second reason is understandable. As we explained above, deflation is quite difficult to accomplish in this algorithm.

Next, they deemed busy-wait for inflation to be acceptable for two reasons. First, the cost is paid only once because of the absence of deflation. Second, the cost is amortized over the lifetime of an object, because of the locality of contention, which implies that the object should stay in the inflated mode. However, as we have just mentioned, their decision not to deflate may not necessarily be the right one in terms of performance. In addition, locality of contention does not necessarily exist.

They also mention that standard back-off techniques like those proposed by Anderson [5] can be used to ameliorate the cost and the negative effect of busy-waiting. However, Anderson's techniques are considered and evaluated in situations in which spin locks are useful, namely, where the critical section is small or where no other process or task is ready

[4]This is not nonsensical, because the most important contribution of the thin lock is that it removed the performance tax Java imposes even on single-threaded applications.

to run. Lock activities in Java are so diverse that we do not think that these techniques are straightforwardly applicable to Java.

It is possible to eliminate the busy-wait or yield of control in the inflation loop, by using an atomic operation in the `Java_lock_release` function. However, the performance in the absence of contention will no longer be as high as in the base algorithm. Since atomic operations are usually very costly, we consider that such algorithms belong to a different class of lock algorithms in terms of their performance characteristics.

Finally, it is worth noting that yielding of control in the inflation loop (line 19) is not necessarily easy to implement correctly, because of *priority inversion* [71, 84, 111]. This refers to the phenomenon where a higher-priority thread must wait for the processing of a lower-priority thread. For instance, the situation arises when a high-priority thread attempts to acquire an object's lock that has been acquired by a low-priority thread. The algorithm causes the high-priority thread to yield control, which is likely to result in the thread being rescheduled and exhausting the processor.

To solve the priority inversion problem, the implementation of thin lock assumed a minimal level of support from the underlying operating systems, specifically *decay-usage scheduling*. However, even this minimal level of support is not provided in some systems, while priority decay is too slow in others. In the end, it turns out that each platform requires its own hacks to get around the problems.

4.4 Analysis of Contentions in Java Locks

To evaluate approaches to mode transitions, we measured the synchronization activities of multi-threaded Java programs in IBM JDK 1.1.6 for the AIX operating system. In particular, we were interested in the locality of contention and the durations for which the object's lock were acquired. The former is related to deflation, and the latter to busy-wait for inflation.

We made some additions to the JDK code to log various synchronization events for measurements. We ran all the test programs under AIX 4.1.5 on an unloaded RISC System/6000 Model 43P containing a 166 MHz PowerPC 604ev with 128 megabytes of main memory, and took timing measurements by using the PowerPC's time base facility [49], whose precision is about 60 ns on this machine. The JIT compiler was enabled for all of these measurements.

Table 4.1 summarizes the programs we measured, which consist of two client applications, eSuite [77] and HotJava [104], and two server-oriented benchmarks, Ibench and Amplace[5] [121].

The Ibench program implements the business logic of the transactions as specified in the TPC-C Benchmark Specification [113]. The benchmark creates client terminals as Java threads, gets them to generate transactions against warehouse data, and reports the

[5]We thank Gaku Yamamoto for making the Amplace benchmark available to us.

Table 4.1: Descriptions of programs measured

Program	Description
eSuite	An office suite from Lotus. Release 1.0. Open the desktop and read 62 pages of a presentation file.
HotJava	A Web browser from Sun. Version 1.1.5. Open an HTML page containing ten 40-KB animated-GIF images.
Ibench	Implements a TPC-C like business logic program. Creates m terminals and gets them to generate transactions concurrently. A value of $m = 4$ was used for the measurements.
Amplace	Implements interactions among agents in a multi-agent server for electronic commerce. Creates m shop agents and n consumer agents, and causes the consumer agents to interact with the shop agents. The values of $m = 10$ and $n = 80$ were used for the measurements.

throughput in transactions per minute. Actually, this program is a very early version of IBM's pBOB (Portable Business Object Benchmark) [12].

The Amplace (Aglets marketplace) program implements interactions among agents in a massive multi-agent system, and is built on top of a middleware system for electronic commerce. The middleware was created with IBM's Java-based mobile agents, Aglets [54], and was actually used in a real commercial service on the Internet[6]. The benchmark creates shop agents and consumer agents, causes each consumer agent to perform searches, and reports the system's throughput in searches per second. In each search, the consumer agent queries all shops for recommended items, and waits for all of the shop agents to return results.

4.4.1 Flat and Fat Sections

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

Figure 4.5 shows an example of flat and fat sections for three objects. Note that these sections are distinguished by the existence of *contention* (or wait states). The flat section does not mean that the object's lock has not been acquired. The lock may be acquired even in a flat section as long as there is no contention. The fat section does not mean that

[6]Provided at `http://www.tabican.ne.jp/`.

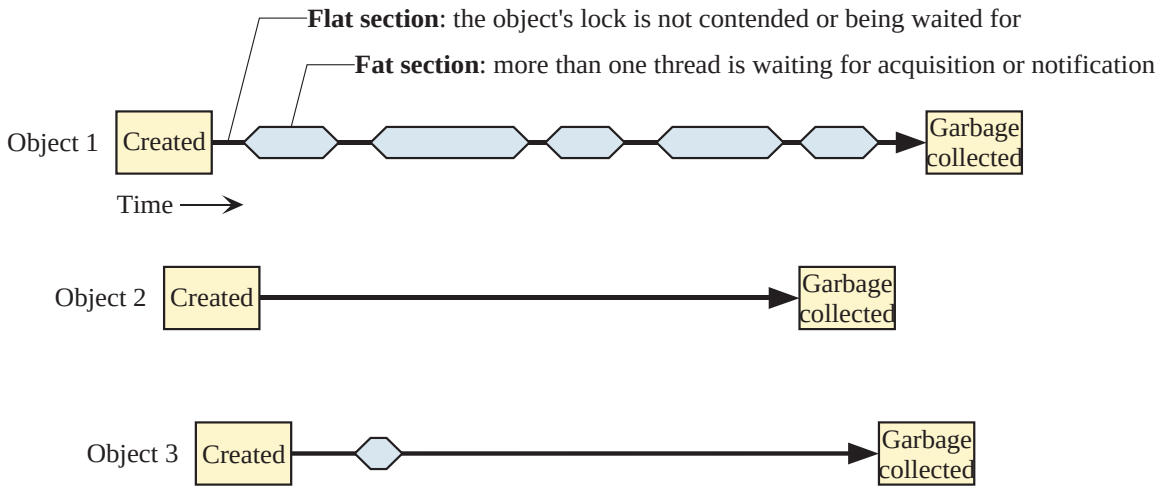

Figure 4.5: Image of flat and fat sections

the lock is being acquired by only one thread. In a fat section, the lock may be acquired by multiple threads one after another.

As long as an object is in a flat section, the object is not involved in any suspend-lock operations. Thus, in terms of performance, objects should be in the flat mode in flat sections. On the other hand, an object in a fat section is actually involved in suspend-lock operations. Objects must therefore be in the inflated mode in fat sections. An object having at least one fat section is said to be *heavily synchronized*. In Figure 4.5, Objects 1 and 3 are heavily synchronized, and are inflated in the base algorithm.

Table 4.2 summarizes the runtime statistics of the programs measured[7]. As shown in the table, we measured the number of threads created, the maximum number of threads that exist simultaneously, the number of objects created, the number of objects that are synchronized, and the total number of synchronization operations. In addition, we include in Table 4.2 the number of objects that are heavily synchronized, and the total number of synchronization operations for those objects. As we see in the table, less than 1.6% of the synchronized objects are heavily synchronized. For this point, the assumption of thin lock is correct. However, these heavily synchronized objects are involved in more than 32 times as many synchronization operations on average. This suggests that heavily synchronized (contended) objects should be analyzed more deeply.

[7]For Amplace, numbers were taken from a run of the program that was partial due to the size of the internal buffer for logging, but which was long enough to allow discussion of the benchmark's runtime characteristics.

Table 4.2: Overall synchronization statistics

Program	Threads created	Threads existing simultaneously	Objects created
eSuite	31	28	228,298
HotJava	22	16	67,642
Ibench	6	6	752,445
Amplace	207	174	465,936

Program	Objects created	Objects sync'd	SyncOps for sync'd objects	SyncOps per sync'd object
eSuite	228,298	18,120 (7.94%)	1,664,978	91.9
HotJava	67,642	6,735 (9.96%)	1,130,991	167.9
Ibench	752,445	97,957 (13.0%)	5,925,847	60.5
Amplace	465,936	61,432 (13.2%)	1,756,650	28.6

Program	Objects sync'd	Objects heavily sync'd	SyncOps for heavily sync'd objects	SyncOps per heavily sync'd object
eSuite	18,120	115 (0.63%)	20,148	175.2
HotJava	6,735	39 (0.58%)	257,394	6,599.9
Ibench	97,957	114 (0.12%)	1,770,146	15,527.6
Amplace	61,432	955 (1.55%)	226,320	237.0

4.4.2 Locality of Contention

The locality of contention is an assumption of thin lock that the contended object is almost always contended. Using our terminology, we can rephrase the locality of contention as follows: fat sections are long, or flat sections following fat sections are short. Here, time is represented by the number of synchronization operations. We verify these claims in this subsection.

Consider the example shown in Figure 4.5. Locality of contention exists for Objects 1 and 2. However, this is not the case for Object 3, for which the contention occurs only in a short period. Actually, it is better to deflate the lock for Object 3 after the short fat section is finished. If such objects are a large fraction of the heavily synchronized objects, deflation can become very important.

For verifying the locality of contention in real multi-threaded programs, we first divided the heavily synchronized objects into two groups. Objects in the *nowait* group are only involved in mutual exclusion by monitor enter and exit, while objects in the *wait* group are also involved in event notification by `wait` and `notify(All)`.

Table 4.3: Locality of contention

Program	Group	Objects heavily sync'd	SyncOps	Fat sections per object	SyncOps per fat section	SyncOps per flat section
eSuite	*nowait*	14	10,575	1.21	24.41	327.74
	wait	101	9,573	4.91	12.07	6.08
HotJava	*nowait*	27	236,884	4.67	35.45	1519.07
	wait	12	20,510	15.22	16.60	92.35
Ibench	*nowait*	114	1,770,146	12.91	81.47	1040.50
	wait	0	0	—	—	—
Amplace	*nowait*	51	206,240	12.29	25.39	281.54
	wait	904	20,080	2.91	4.31	2.60

Table 4.3 shows the lengths of fat and flat sections for each group of objects. Clearly, we observe locality of contention in the wait group. However, we do not see any such tendency in the other group. On the contrary, the flat sections are more than 10 times longer than the fat sections, which means that most contentions are temporary for this group. This *contention transience* suggests that deflation should be performed for objects in the nowait group.

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

4.4.3 Durations of Lock Acquisition

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

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

Table 4.4 shows the results. As we see, in comparison with the average length, there are a few sections that are unusually long. This suggests that busy-wait is potentially dangerous. Furthermore, notice that if we perform deflation as recommended above, busy-wait for inflation is likely to be attempted many more times.

Table 4.4: Durations of locked sections

Program	Locked sections	Average duration	Longest duration	Durations ≥10 ms (ratio)
eSuite	821,676	0.245 ms	41.74 s	1,359 (0.17%)
HotJava	555,605	1.735 ms	5.52 s	2,767 (0.50%)
Ibench	2,960,314	0.131 ms	8.42 s	6,138 (0.21%)
Amplace	849,377	0.618 ms	24.31 s	2,474 (0.29%)

4.5 Our Lock Algorithm

As described in the previous section, our assumption in improving thin lock is that most locks are not contended heavily in Java, so it is better to *deflate* the lockword after the (temporary) contention goes away. In this section, we describe a new algorithm, *tasuki lock*, that allows both inflation without busy-wait and deflation, but still maintains an almost maximum level of performance in the absence of contention. In particular, the algorithm does not require an atomic operation in releasing locks.

4.5.1 Tasuki Lock Algorithm

Tasuki lock requires one additional bit in an object's header. The bit is set when flat lock contention occurs, and is therefore named the *FLC bit*. An important requirement is that the FLC bit of an object belongs to a different word from the lockword, since the bit is set by a contending thread without holding the corresponding flat lock.

Figure 4.6 shows the data structure for tasuki lock. In addition to the FLC bit, the tasuki lock uses a monitor table to maintain the association between objects and monitors. In the figure, the monitor for Object 5 is registered in the table but not referred to from the object's lockword. This is the situation when the lockword was deflated. Compare it with the data structure of the thin lock shown in Figure 2.10.

Figure 4.7 shows our lock algorithm, where the sidelines indicate the portions added or modified from the base code (Figure 4.4). As in the base code, this is a simplified version for explanation. Our actual implementation also uses the 24-bit lockword shown in Figure 4.2, and performs optimizations for shallow recursive acquisitions, which are the same as for the thin lock.

The first thing to note is that, if a thread fails to grab the flat lock in the inflation loop, it waits on a monitor (line 24), so it does not busy-wait at all.

The second thing to note is that the `Java_lock_release` function, which is responsible for notifying a thread waiting in the inflation loop, first tests the FLC bit (`OBJFLAG_FLAT_CONTENDED`) *outside* the critical region, without entering a monitor. This means that, in the absence of contention, the additional overhead is only one bit test (line 48). Hence,

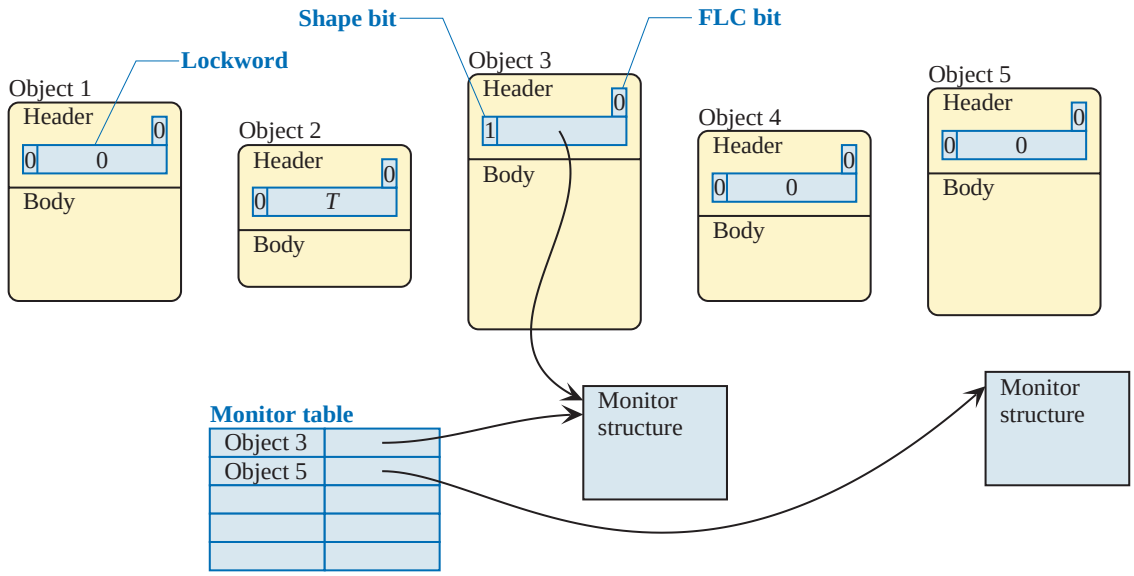

Figure 4.6: Data structure for tasuki lock

an almost maximum level of performance is maintained. We explain in Section 4.5.2 why this unsafe bit-test does not cause a race hazard.

The third thing to note is that the algorithm conditionally deflates an object's lockword at lines 59–60. The necessary condition is that nobody is waiting on the object. In addition, as long as the necessary condition is satisfied, tasuki lock allows selective deflation, which is the purpose of the second condition in line 59, `Better_to_deflate`. We can implement various *deflation policies* in this function. For instance, we can deflate lockwords on the basis of dynamic or static profiling information. We explain in Section 4.5.3 why deflation is so simple to realize in our algorithm, and does not cause problems such as those described in Section 4.3.2.

The last thing to note is the way in which monitors are used. In inflating an object's lockword, all the code related to inflation is basically protected by the corresponding *inflation monitor*, which is entered at lines 17, 50, and 70. The noteworthy exception is the unsafe test of the FLC bit (line 48). Interestingly, the inflation monitor is simply the same as the suspend lock whose reference is eventually stored in the lockword. As we will show in Section 4.5.4, tasuki lock ensures that the monitor's dual roles do not interfere with each other.

To understand the duality a bit better, consider the case in which a thread acquires an object's lock already in the inflated mode. The thread fails in the initial `compare_and_swap` in the `Java_lock_acquire` function (line 13). It then looks up and enters the object's inflation monitor (line 17). The only remaining thing is to fail in the conditional expression of the `while` loop (line 19). Notice that the object's monitor is entered after the `while` loop in the base algorithm (line 23 in Figure 4.4), but it has already been

```
    1 : // Object header contains a lockword and an FLC bit
    2 : typedef struct object {
    3 :     //    :
|   4 :     volatile unsigned int flag; // 1 bit is used as an FLC bit
    5 :     volatile unsigned int lockword;
    6 :     //    :
    7 : } Object;
    8 :
    9 : #define SHAPE_BIT 0x80000000
   10 :
   11 : int Java_lock_acquire(Object *obj) {
   12 :     // flat lock path
   13 :     if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS)
   14 :         return SUCCESS;
   15 :     // inflated lock and inflation path
|  16 :     monitor_t *mon = obtain_monitor(obj);
|  17 :     monitor_enter(mon);
   18 :     // inflation loop
   19 :     while ((obj->lockword & SHAPE_BIT) == 0) {
|  20 :         obj->flag |= OBJFLAG_FLAT_CONTENDED; // set the FLC bit
   21 :         if (compare_and_swap(&obj->lockword, 0, thread_id()) == SUCCESS)
   22 :             inflate(obj, mon);
|  23 :         else
|  24 :             monitor_wait(mon); // wait on the inflation monitor
   25 :     }
|  26 :     return SUCCESS;
   27 : }
   28 :
   29 : // Returns a monitor associated with the object
|  30 : monitor_t *obtain_monitor(Object *obj) {
|  31 :     unsigned int lw = obj->lockword;
|  32 :     if ((lw & SHAPE_BIT) != 0) return (monitor_t *)(lw & ~SHAPE_BIT);
|  33 :     return lookup_monitor(obj); // search for a monitor in the table,
|  34 : }                               // or create/register a new monitor
   35 :
   36 : // Inflate the object's lockword, which is held by current thread
   37 : void inflate(Object *obj, monitor_t *mon) { // mon must also be entered
|  38 :     obj->flag &= ~OBJFLAG_FLAT_CONTENDED; // reset the FLC bit
|  39 :     monitor_notify_all(mon);
   40 :     obj->lockword = SHAPE_BIT | (unsigned int)mon; // set the shape bit
   41 : }
   42 :
```

Figure 4.7: Our lock algorithm: tasuki lock (1 of 2)

```
   43 : int Java_lock_release(Object *obj) {
   44 :     unsigned int lw = obj->lockword;
   45 :     if ((lw & SHAPE_BIT) == 0) { // flat lock path
   46 :         if (lw != thread_id()) return ILLEGAL_STATE;
   47 :         obj->lockword = 0;
|  48 :         if (obj->flag & OBJFLAG_FLAT_CONTENDED) { // test the FLC bit,
|  49 :             monitor_t *mon = obtain_monitor(obj); //      the only overhead
|  50 :             monitor_enter(mon);
|  51 :             if (obj->flag & OBJFLAG_FLAT_CONTENDED) monitor_notify(mon);
|  52 :             monitor_exit(mon);
|  53 :         }
   54 :         return SUCCESS;
   55 :     }
   56 :     // inflated lock path, deflate if possible
   57 :     monitor_t *mon = (monitor_t *)(lw & ~SHAPE_BIT);
   58 :     if (!monitor_being_entered_by_me(mon)) return ILLEGAL_STATE;
|  59 :     if (!monitor_being_waited(mon) && Better_to_deflate(obj))
|  60 :         obj->lockword = 0; // deflation
   61 :     return monitor_exit(mon);
   62 : }
   63 :
   64 : int Java_lock_wait(Object *obj) {
   65 :     unsigned int lw = obj->lockword;
   66 :     if ((lw & SHAPE_BIT) == 0) { // flat mode
   67 :         if (lw != thread_id()) return ILLEGAL_STATE;
   68 :         // force the inflation
|  69 :         monitor_t *mon = obtain_monitor(obj);
|  70 :         monitor_enter(mon);
   71 :         inflate(obj, mon);
   72 :     }
   73 :     // execute the wait using the monitor structure
   74 :     monitor_t *mon = (monitor_t *)(obj->lockword & ~SHAPE_BIT);
   75 :     return monitor_wait(mon); // wait on the Java object
   76 : }
   77 :
   78 : int Java_lock_notify(Object *obj) {
   79 :     unsigned int lw = obj->lockword;
   80 :     if ((lw & SHAPE_BIT) == 0) { // flat mode
   81 :         if (lw != thread_id()) return ILLEGAL_STATE;
   82 :         return SUCCESS; // no one should be waiting, no need to inflate
   83 :     }
   84 :     // execute the notify using the monitor structure
   85 :     monitor_t *mon = (monitor_t *)(lw & ~SHAPE_BIT);
   86 :     return monitor_notify(mon);
   87 : }
   88 :
   89 : // Java_lock_notify_all() is implemented in the same manner
```

Figure 4.7: Our lock algorithm: tasuki lock (2 of 2)

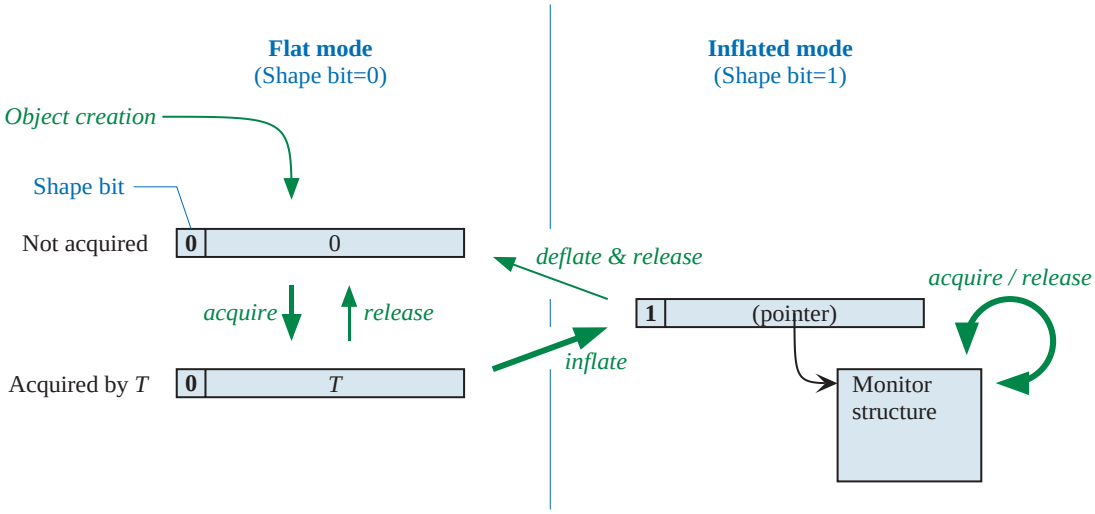

Figure 4.8: Lockword state transitions of tasuki lock

entered as the inflation monitor in our algorithm.

Finally, we briefly describe the `lookup_monitor` function used at line 33. The code is not shown in Figure 4.7. We assume that there exists an underlying *monitor table* that maintains associations between objects and their monitors. Given an object, the function searches the monitor table, and returns the object's monitor, creating a monitor if necessary. Notice that deflation simply means the mode transition of a lockword. It does not imply the removal of the corresponding association from the table. The table and `lookup_monitor` can be basically same as those in the monitor table code shown in Figure 2.7. However, unlike the old code, the table is accessed only in the inflation path. Since it is not used in the usual, flat or already-inflated, cases, the scalability is not degraded.

Figure 4.8 illustrates the transitions of a lockword in tasuki lock. Compared to the base algorithm (Figure 4.3), a new path is added for the deflation.

4.5.2 Testing an FLC Bit

This subsection explains why testing an FLC bit outside the critical region at line 48 of `Java_lock_release` does not cause a race hazard. Specifically, we show that no thread continues to wait forever at line 24 in the inflation loop without receiving any notification. We start with the following two properties, which can be obtained immediately from the code in Figure 4.7.

Property 4.1 *An object's shape bit is set only in the* `inflate` *function, and cleared only at line 60 in the* `Java_lock_release` *function. In both cases, the object's monitor is entered.*

Property 4.2 *An object's FLC bit is set only in the inflation loop, and cleared only in the* `inflate` *function. In both cases, the object's monitor is entered.*

We then prove the following crucial property, which states that the failing `compare_and_swap` has an important implication. There are subtle issues related to this property on a multiprocessor system, which we will consider in Section 4.5.5.

Property 4.3 *If a thread T fails in the* `compare_and_swap` *against an object in the inflation loop (line 21), there is always some other thread that subsequently tests the object's FLC bit at line 48 of the* `Java_lock_release` *function.*

Proof. Let t_1 be the time at which T fails in the `compare_and_swap`. From Property 4.1, the object's shape bit remains the same at t_1 as when T finds the `while` loop's conditional true. That is, the lockword is in the flat mode at t_1.

The failure then implies that some other thread U holds the object's flat lock at t_1. In other words, U does not yet execute the store of 0 into the lockword, which is at line 47 of the `Java_lock_release` function, at t_1. Hence, U will unsafely test the FLC bit at line 48 after t_1. □

Figure 4.9 illustrates Property 4.3, where the number beside each box is the line number of the code shown in Figure 4.7. Using this figure, the property can be informally stated as:

Because $t_1 < t_2$, therefore $t_0 < t_3$.

The key point of the property is that the shape of *tasuki* (Figure 4.10) is formed by the two arrows at the center, and thus our algorithm was named *tasuki lock*[8].

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

Property 4.4 *If a thread T waits on an object's monitor M in the inflation loop, there is always a thread that subsequently calls the* `inflate` *function against the object or executes a* `compare_and_swap` *against the object in the inflation loop.*

Proof. Let t_1 be the time at which T performs the `compare_and_swap` in the inflation loop that fails and causes T to wait on M. From Property 4.3, there exists some other thread U that will unsafely test the corresponding object's FLC bit at some time t_3 after t_1.

We then perform a two-case analysis based on whether U finds the FLC bit set or cleared at t_3. Consider the simpler case, in which U finds the FLC bit cleared at t_3. From Property 4.2, the clearance of the FLC bit implies that a third thread V has called `inflate` before t_3. Since T already entered M at t_1, V called `inflate` after T started

[8]There is another reason for the name. In tasuki lock, the monitor has dual roles as will be shown in Section 4.5.4. A monitor used for protecting the inflation code in the flat mode (line 17) is *passed* to the inflated mode as a suspend-lock (line 22), like the *tasuki* baton in an Ekiden [119] race in Japan.

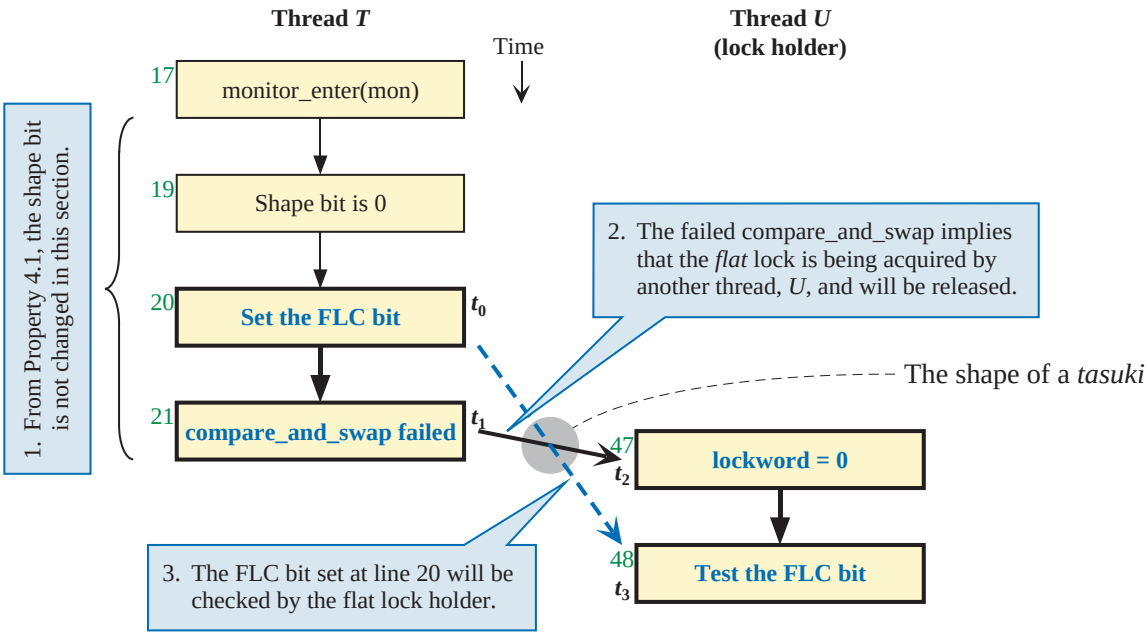

Figure 4.9: The key property of tasuki lock (heart of *tasuki*)

Figure 4.10: Japanese traditional *tasuki*

waiting on M and implicitly exited from M at line 24. Thus, the property holds in this case.

Next, consider the case in which U finds the FLC bit set at t_3. The thread U then succeeds in the unsafe test and continues to execute test-and-notify, which is properly protected by M. The properly protected test at line 51 may fail or succeed. If it fails, obviously, the FLC bit was cleared. By the same reasoning as above, we can deduce that the property holds.

If the protected test succeeds, the thread U notifies M, and wakes up one of the waiting threads, which may or may not be identical to T. Again, notice that, since T already entered M at t_1, U calls `monitor_notify` after T started waiting on M and implicitly exited from M. The woken-up thread W eventually resumes the execution and reaches the `while` loop's conditional at line 19.

The thread W may find the conditional false or true. If the conditional is false, some thread has set the shape bit, which implies, from Property 4.1, that the `inflate` has been called by the thread. If the conditional is true, the thread W continues to execute the body of the inflation loop, which means it executes `compare_and_swap` at line 21. Thus, we have shown that the property holds in both cases. □

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

Thus, a practical consequence of Property 4.4 is that, if a thread is waiting on an object's monitor in the inflation loop, there is always a thread that subsequently calls the `inflate` function. Since the function wakes up all the threads waiting on the inflation monitor (line 39), this means that every thread waiting on the inflation monitor at the call is eventually woken up[9].

4.5.3 Deflating a Lockword

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

Property 4.5 *No thread ever acquires an object's flat lock when some other thread holds the object's suspend lock. Similarly, no thread ever acquires an object's suspend lock when some other thread holds the object's flat lock.*

[9]Because of deflation, the woken-up thread might wait on the inflation monitor again, by passing the `while` loop's conditional at line 19.

Proof. In order for a thread to acquire an object's flat lock, the thread needs to succeed in the first `compare_and_swap` in the `Java_lock_acquire` function (line 13). However, it never succeeds as long as some other thread holds the suspend lock, since the shape bit is set in that situation.

Similarly, in order for a thread to acquire an object's suspend lock, the thread T needs to enter the object's monitor *and* fail in the `while` loop's conditional (line 19). The conditional never fails as long as some other thread holds the object's flat lock, where the shape bit is cleared. □

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

4.5.4 Monitors' Dual Roles

The immediate concern about using a monitor both for protecting the inflation code and for a suspend-lock of a Java object is that these two roles may interfere with each other. More specifically, one concern is that notifying an inflation monitor (lines 39 and 51) might wake up a thread waiting on the Java object (line 75), and another is that notifying a Java object (line 86) might wake up a thread waiting on the inflation monitor (line 24).

Actually, tasuki lock ensures that neither case occurs, since the functions for waiting on and notifying a Java object, `Java_lock_wait` and `Java_lock_notify`, are properly defined.

First, as in the base algorithm, the `Java_lock_wait` function (lines 64–76) forces inflation of the lockword if the lockword is in the flat mode[10]. This is done under appropriate protection by the object's inflation monitor acquired at line 70. In addition, the `Java_lock_release` function suppresses deflation as long as a thread is waiting on a Java object (line 59). Combining the two, we obtain the following property:

Property 4.6 *If some thread is waiting on a Java object, the lockword is in the inflated mode.*

Second, the `Java_lock_notify` function performs one of the following two actions. If the object's lockword is in the flat mode, the function simply ignores the notification request (line 82), since the contrapositive of Property 4.6 states that no thread is waiting on the Java object in this mode. Otherwise, the function notifies the corresponding suspend lock (line 86). This implementation immediately yields the following property:

Property 4.7 *If a thread notifies a Java object, the lockword is in the inflated mode.*

[10]Notice that the thread which calls `Java_lock_wait` in the flat mode must already hold the flat lock.

Third, tasuki lock notifies an inflation monitor in two places, one in the `Java_lock_release` function (line 51) and the other in the `inflate` function (line 39). Notice that both places are protected by the monitor. A thread of control reaches the former only when the FLC bit is set, and the latter only when the shape bit is cleared. Each of the conditions holds if and only if the lockword is in the flat mode. We thus have the following property:

Property 4.8 *If a thread notifies an inflation monitor, the lockword is in the flat mode.*

Finally, tasuki lock causes a thread to wait on an inflation monitor M in one place (line 24), that is, in the inflation loop that is entered only when the shape bit is cleared. In addition, calling `inflate` against M wakes up all the waiting threads on M. This implies the following property:

Property 4.9 *If a thread is waiting on an inflation monitor, the lockword is in the flat mode.*

We are now ready to conclude the subsection. From Property 4.8 and the contrapositive of Property 4.6, we can infer that it is impossible to notify an inflation monitor to wake up a thread waiting on the Java object. Similarly, from Property 4.7 and the contrapositive of Property 4.9, we can infer that it is impossible to notify a Java object to wake up a thread waiting on the inflation monitor.

4.5.5 Multiprocessor Considerations

In general, special care must be taken in implementing a lock algorithm on a multiprocessor system supporting a relaxed memory model [1, 27]. For instance, when implemented on a PowerPC multiprocessor system, whose memory model is weakly consistent [49, 79], the store of 0 in the lockword at line 47 of the `Java_lock_release` function must be preceded by a `sync` instruction, to ensure that any stores associated with the shared resource are visible to other processors.

This is also the case for the base algorithm. Notice, however, that the store does not have to be followed by a `sync`. Although a thread trying to acquire the same object's lock may see the stale value and thus fail in the `compare_and_swap`, this simply results in a few more iterations in the busy-wait loop.

Our algorithm imposes more stringent requirements. As we have seen above, the proof of Property 4.3 relies on the implication that, if T fails in the `compare_and_swap`, U has not yet executed the store of 0 in the lockword at line 47 and will test the FLC bit at line 48. However, the implication does not necessarily hold in a relaxed memory model. To ensure Property 4.3, which can be stated as "Because $t_1 < t_2$, therefore $t_0 < t_3$." in Figure 4.9, we must ensure the sequences of instructions preserve $t_0 < t_1$ and $t_2 < t_3$. We may thus need to issue additional instructions.

Actually, on a PowerPC multiprocessor system, the store must also be followed by `sync` to ensure $t_2 < t_3$. The failure of the `compare_and_swap` then implies that U has not yet started executing the next instruction of the `sync` instruction, testing the FLC bit (line 48). Thus, Property 4.3 still holds, although the additional instruction slows down the performance in the absence of contention.

Besides the store, the same care must be taken with the setting of the FLC bit in the inflation loop (line 20) to ensure $t_0 < t_1$, and also with the clearance of the FLC bit in the `inflate` function (line 38). However, both are in the inflation path, and do not affect the performance of the uncontended case.

4.6 Performance Results

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

We adopted a deflation policy as suggested by the measurements in Section 4.4. That is, we deflate the lockword of an object if the object belongs to the *nowait* group and if a fat section of the object has ended. We thus check these two conditions in `Better_to_deflate` at line 59 in Figure 4.7.

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

We measured the performance of two versions of JDKs on the same machine with the same configuration as in Section 4.4, and took a median of ten runs for each of the micro- and macro-benchmarks. Neither of the JDKs included code for logging. The JIT compiler was enabled for all the measurements.

4.6.1 Micro-Benchmarks

Table 4.5 summarizes the three micro-benchmarks we used. The `LongLocker` benchmark test is intended to determine the effect of inflation without busy-wait, and Figure 4.11 shows the results. As we see, as the number of concurrent threads increases, the performance of the original JDK deteriorates greatly, while our JDK maintains a constant performance.

Table 4.5: Descriptions of micro-benchmarks

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

The `FlatFat` benchmark test is intended to determine the effect of deflation. Figure 4.12 shows the results for $n = 50$ and $m = 100{,}000$. As we see, once inflation has occurred, the original JDK no longer performs as well in subsequent flat sections as in the first flat section, while our JDK maintains a constant performance in all the flat sections. We also obtained similar results for other cases such as $n = 10$, 20, or 80. On the other hand, we observed that the performance in the fat section slightly degraded in our algorithm. This is primarily because of the overhead of checking deflatability in the `Java_lock_release` function (line 59 of Figure 4.7).

The `Thrashing` benchmark test was written so that each time one thread acquires an object's lock it ends up in contention with the other thread. The result for $m = 2{,}000$ is that, while the original JDK takes 1,396 msec to complete, our JDK takes 1,438 msec, inflating and deflating the object exactly two thousand times.

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

We suspect that this is mainly because of the underlying monitor table that we use to maintain associations between objects and monitors. We simply enabled the implementation of the monitor cache [123] in the Sun JDK 1.1.6, which includes an optimization for small per-thread monitor caches. A thread first looks up its own small monitor cache for an association. When doing so, it does not have to hold any lock. Thus, as long as a hit occurs in the per-thread cache, obtaining an object's monitor is almost as efficient in the flat mode as in the inflated mode.

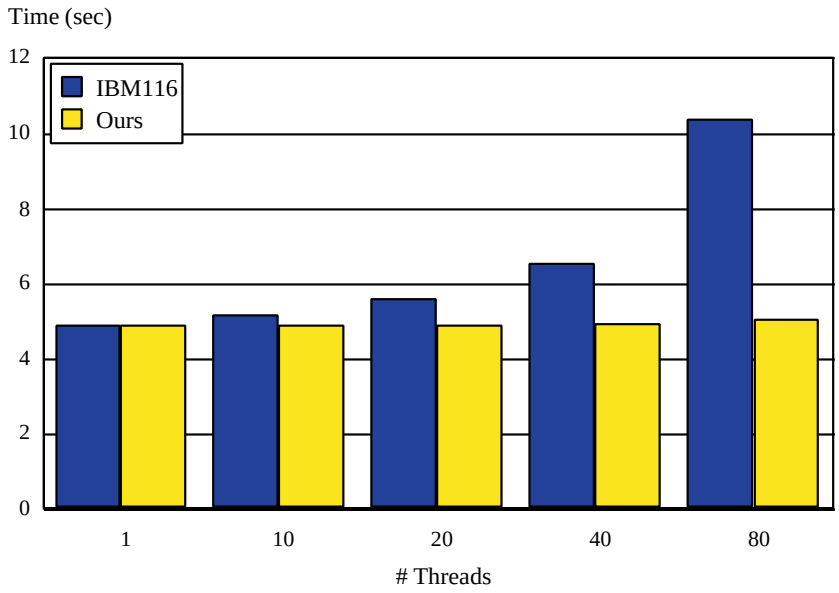

Figure 4.11: Performance of the `LongLocker` benchmark

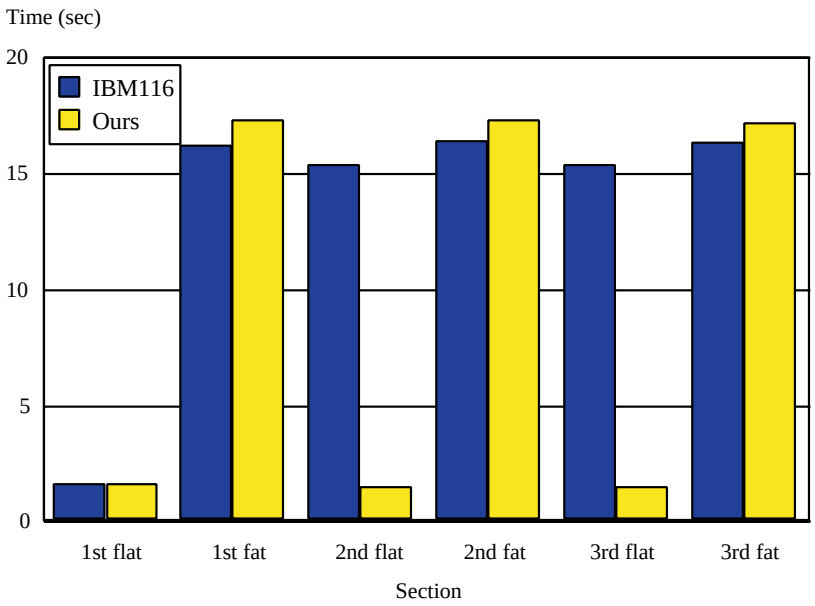

Figure 4.12: Performance of the `FlatFat` benchmark

Table 4.6: Inflation and deflation dynamics of macro-benchmarks

	IBM116			
Program	SyncOps	Inflations	Deflations	SyncOps in inflated mode (ratio)
eSuite	1,664,978	115	0	0.84%
HotJava	1,130,991	39	0	19.5%
Ibench	5,925,847	114	0	26.3%
Amplace	1,756,650	955	0	10.7%

	Ours			
Program	SyncOps	Inflations	Deflations	SyncOps in inflated mode (ratio)
eSuite	1,664,366	138	127	0.52%
HotJava	1,047,248	165	157	2.01%
Ibench	5,917,825	1,255	1,255	1.82%
Amplace	1,535,618	1,661	1,476	1.39%

4.6.2 Macro-Benchmarks

For macro-benchmarking, we used the same set of Java programs as in Section 4.4, whose processing is described in Table 4.1. We first measured the inflation and deflation dynamics of these programs in our JDK together with the dynamics in the original JDK. Table 4.6 shows the results[11]. Notice that we collected these dynamics by using another set of two versions of JDK, in which the logging facilities were enabled.

These results show that significantly more synchronizations occur in the flat mode in our JDK. That is to say, our algorithm substantially reduces the synchronization overhead. However, the extent to which the reduction in the synchronization overhead is reflected in the overall performance varies from application to application, depending on several factors, especially the amount of the execution time an application spends on synchronization.

For the client programs of eSuite and HotJava, we could not see any consistent differences between the original JDK and ours. These are interactive applications, and the execution times significantly vary from run to run. On the other hand, we did see differences for the server-oriented programs. The throughput is improved from 10,920.12 to 12,163.52 transactions per minute in Ibench, and from 1.5286 to 1.6076 searches per second in Amplace. That is to say, we observed improvements of 11.4% and 5.16%,

[11]The figures for the dynamics do not necessarily coincide with those in the measurements given in Section 4.4. This is primarily because of the different amounts of code inserted for recording different kinds of synchronization events.

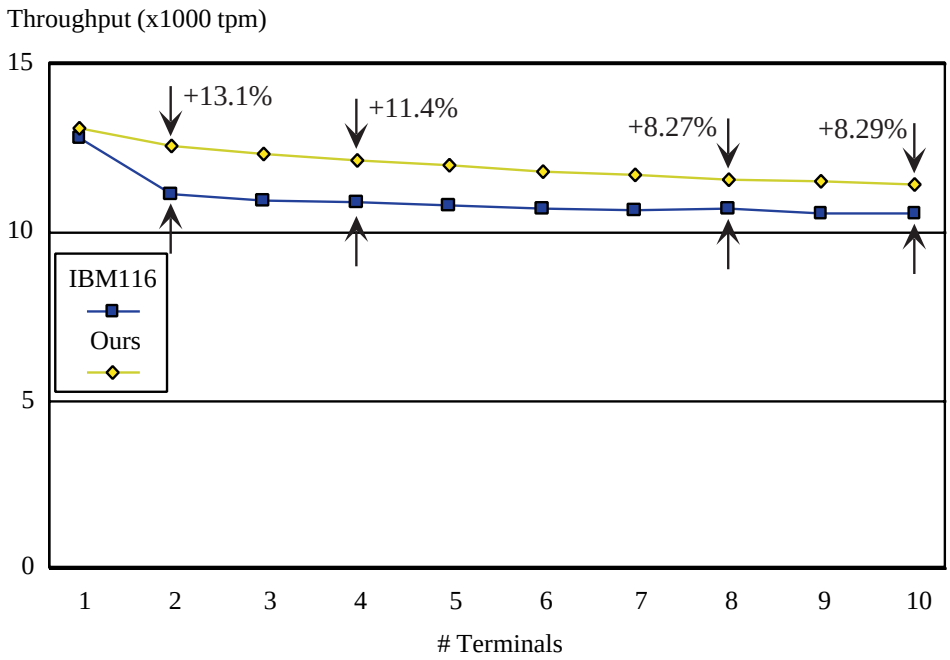

Figure 4.13: Performance of the Ibench benchmark for various numbers of terminals

respectively.

We also ran Ibench while varying the number of terminals, m, from 1 to 10. As Figure 4.13 shows, our lock algorithm achieved the maximal improvement of 13.1% at $m = 2$, and the minimal improvement of 8.27% at $m = 8$. Furthermore, it is worth noting that the throughput drops at $m = 2$ much less sharply in our JDK.

4.7 Summary

Thin lock was the first bimodal lock algorithm for Java, and made a significant contribution to accelerating Java locks. For uncontended objects, the lock can be acquired with only one `compare_and_swap`, and released by simply storing 0. However, once a lock is contended, it is inflated by busy-waiting, and never deflated.

By analyzing the lock behavior of contended objects in real multi-threaded Java applications, we discovered that most contentions are temporary, especially for objects in the *nowait* group. To exploit the *contention transience*, we defined a new bimodal lock algorithm, called tasuki lock, that allows both inflation without busy-wait and deflation. The algorithm maintains an almost maximal level of performance in the absence of contention. Two intriguing points in tasuki lock are the way in which the flat lock contention bits are manipulated, and the dual roles of monitors.

We evaluated an implementation of tasuki lock in IBM's production JDK. The results

of micro-benchmarks show that, in comparison with the original JDK, our algorithm achieved a constant performance even in the presence of a long-time lock holder, and recovers the highest performance in the absence of contention even after inflation has occurred. In addition, they suggest that thrashing of inflation and deflation is not a concern.

Although the aim of removing busy-wait for inflation is to avoid disasters, that of deflation is to improve performance. Our measurements have shown improvements up to 13.1% in server-oriented benchmarks. Currently, our tasuki lock is used in all of IBM's production Java developer kits [53], including versions for AIX, Windows[12], Linux, and z/OS [51].

To summarize, the research shown in this chapter contributes the following results:

- Finding of the *contention transience*
  We analyzed contended Java locks deeply and found contention transience in the *nowait* group objects.

- Proposal of *tasuki lock*
  We proposed a new bimodal lock algorithm, tasuki lock, that supports deflation, and inflation without a busy-wait loop.

- Implementation and measurement
  We implemented the algorithm in a production JDK, and confirmed the increased robustness and performance improvement of up to 13.1%.

[12]The Windows version cannot be downloaded as a stand alone package, but it is included in IBM's Java-based products, such as WebSphere Application Server [50].

Chapter 5

Reservation Lock

5.1 Introduction

Java locks have been improved so as to be acquired and released with only a few machine instructions in their common case, the absence of contention, as a result of various research efforts to accelerate Java locks such as thin lock [11], tasuki lock [88], and others [3, 28]. However, in all of these algorithms, the instruction sequence inevitably contains one or more compound *atomic operations* such as `compare_and_swap`. Considering that atomic operations are especially expensive in modern architectures, they are becoming the major overhead factor in Java locks.

The atomic operations are very effective in the situation where multiple threads acquire a lock symmetrically. However, this is not the best solution when there is an asymmetry in the lock acquisitions. If an object's lock is frequently acquired by a specific thread, the lock's cost may be further reduced by giving a certain precedence to that thread, while shifting costs to other threads. One obvious example is single-threaded programs[1], in which locks must have been performed by only one thread. Even for multi-threaded programs, there may be opportunities to exploit such asymmetries.

From this viewpoint, in this chapter we propose our second improvement of Java locks, named *reservation lock* [65, 66]. It also follows the principle of optimizing common cases. The observation exploited is the biased distribution of lockers called *thread locality*, as described above. The key idea is to allow a lock to be *reserved* for a thread. The reservation-owner thread can perform the lock processing without atomic operations, so the lock overhead is minimized. If another thread attempts to acquire the reserved lock, the reservation must first be canceled, and the lock processing falls back to an existing

[1]Strictly speaking, Java programs cannot be single-threaded because the Java virtual machine itself normally creates internal helper threads. Even if there is no other thread, lock operations cannot be completely omitted because some error situations which cause `IllegalMonitorStateException` must be checked to comply with the Java specifications [41]. Furthermore, the case must also be supported that a single-threaded program can become multi-threaded by dynamically creating threads.

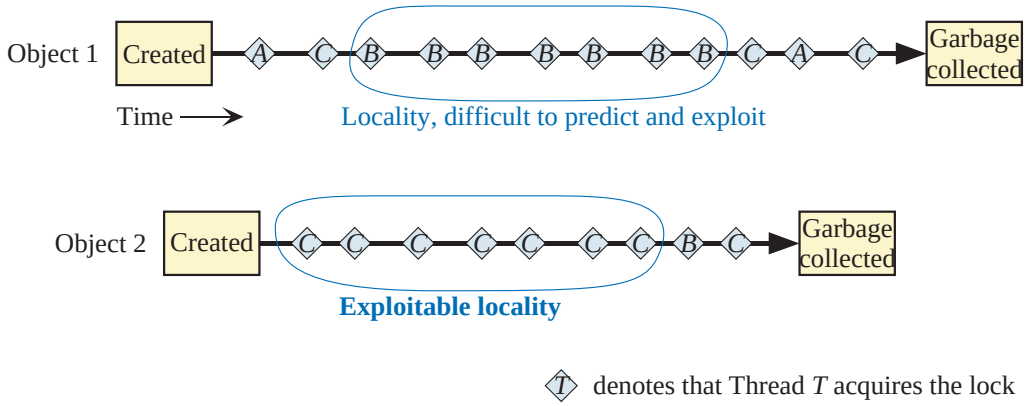

Figure 5.1: General thread locality and exploitable thread locality

algorithm.

As we will see later, the reservation lock can be built on any existing lock algorithm, as long as it uses a lockword (or a field) in the object's header and has one available bit to represent the reservation status. When the *LRV bit* is set, the meaning of the lockword is defined by our reservation lock, while when the bit is not set, the meaning is defined by the underlying algorithm. We have evaluated an implementation of the reservation lock in IBM's production virtual machine and its JIT compiler. The results show that it achieved performance improvements up to 53% in real Java programs.

The rest of this chapter is organized as follows. Section 5.2 shows the thread locality of locks in real Java programs. Section 5.3 describes the algorithm of reservation lock, and Section 5.4 discusses its correctness, characteristics, and several variations. Section 5.5 presents performance results from micro- and macroscopic viewpoints, and discusses some possible extensions. Finally, Section 5.6 summarizes this chapter.

5.2 Thread Locality of Java Locks

The reservation lock we are trying to define in this chapter accelerates Java locks by *reserving* each lock for a thread so it can be quickly processed. To make this idea effective, there must exist a locality such that each object's lock is frequently acquired by a specific thread, for which the lock is to be reserved. This section studies this *thread locality* of Java locks.

The thread locality of a lock is defined in terms of the *lock sequence*, the sequence of threads (in temporal order) that acquire the lock. The general form of thread locality is stated as follows. For a given lock, if its lock sequence contains a very long repetition of a specific thread, the lock is said to exhibit thread locality, while the specific thread is said to be the *dominant locker*.

However, the general form of thread locality is not easy to exploit, since we consider

Table 5.1: Benchmark programs

Program	Multi-threaded?	Description
SPECjvm98		Run each program 3 times in the application mode
_202_jess	No	Expert shell system solving a set of puzzles
_201_compress	No	LZW compression and decompression
_209_db	No	Perform database functions on memory resident DB
_222_mpegaudio	No	Decode MP3 audio files
_228_jack	No	Parser generator generating itself
_213_javac	No	Java source-to-bytecode compiler from JDK 1.0.2
_227_mtrt	Yes	Two-threaded ray tracer
SPECjbb2000	Yes	Simulate the operations of a TPC-C like business logic, run for 8 warehouses
Volano Server	Yes	Chat room simulator
Volano Client	Yes	Chat client, creating 200 connections and sending 100 messages per connection

adaptive optimization of locks at runtime, rather than static optimization using off-line profiles. Figure 5.1 illustrates example lock sequences for two objects. Obviously, Object 1's lock exhibits thread locality to Thread B, but it is very hard for the runtime system to cheaply determine this while the lock sequence is being constructed, and give the lock precedence to B. On the contrary, as for Object 2 in the figure, it is rather easy to give some precedence to Thread C, which was first to acquire the object's lock. Thus, a stronger form of thread locality is considered for exploitability, which is described as follows. For a given lock, if the lock sequence *starts* with a very long repetition of a specific thread, the lock is said to show *exploitable thread locality*. When the lock exhibits exploitable thread locality, the initial locker is the dominant locker.

To investigate how many objects show exploitable thread locality in real programs, we gathered lock statistics using an instrumented version of the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1 [53]. We measured ten Java programs listed in Table 5.1 — the seven programs of the SPECjvm98 [100] each of which was executed three times successively in the application mode, the SPECjbb2000 [99] for eight warehouses, and the server and client programs of the Volano Mark [114]. Among these programs, _227_mtrt, SPECjbb2000, and the Volano Mark programs are multi-threaded programs. The purpose of this investigation is to understand the inherent behavior of Java locks, so we ran these programs with the JIT compiler *disabled*, since some locks would otherwise be optimized away by compiler optimizations.

The focus in our measurements is *the first repetition* in the lock sequence for each lock. This is the beginning subsequence consisting only of the initial locker. If the first

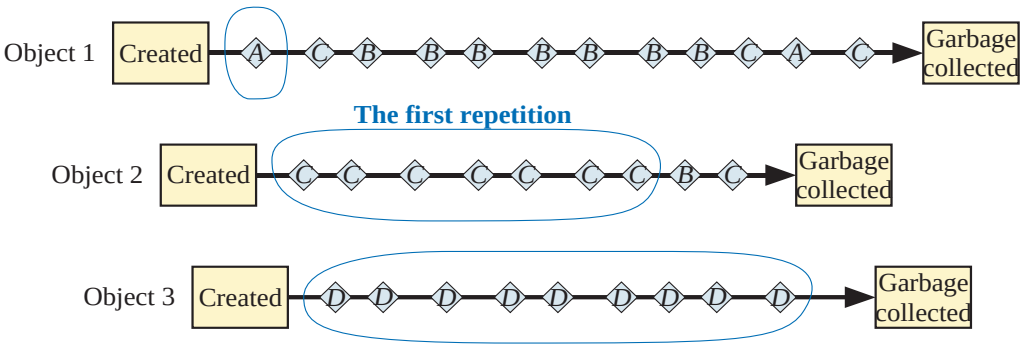

Figure 5.2: Example lock sequences and their first repetitions

repetition of a lock is very long, the lock shows exploitable thread locality. Figure 5.2 shows example lock sequences and their first repetitions for three objects. As shown in this example, the length of the first repetition may be one, which is counted as a first repetition containing only one acquisition. Also, the initial locker may appear again after the first repetition, which is not counted as a first repetition. In this example, the ratio of lock acquisitions in the first repetitions to the total lock acquisitions is calculated as $17/30 = 56.7\%$.

Table 5.2 presents the actual results, which shows the total number of lock acquisitions and the ratios of lock acquisitions in the first repetitions. The results shown here are for the complete execution of each program, including lock acquisitions during the program startup and shutdown. As shown in the table, the vast majority of lock acquisitions are performed by the initial lockers. Even for multi-threaded programs, more than 75% of the lock operations were performed by the initial lockers in the first repetitions. Thus, we can draw the conclusion that a significant number of objects exhibit exploitable thread locality, and their locks thus may be accelerated if they are reserved for their initial lockers.

Notice that the ratios in the last column are not 100% even for single-threaded programs, since the virtual machine implicitly creates helper threads for internal tasks such as finalization. We also note that the initial locker of an object is not necessarily the creator of the object. This actually happens in the Volano Mark programs, where a single thread is dedicated to creating objects and passing them to worker threads that actually use the objects. This suggests that locks should be reserved for their initial lockers, rather than their creators.

In addition, Table 5.3 gives reference data from the viewpoint of the number of objects in the same measurement. The table indicates that only 40% at most of the synchronized objects' locks are actually used by multiple threads. In other words, more than half of synchronized objects' locks are used only by single threads, so are worth being reserved for those threads.

[2]The total number of lock acquisitions and created objects for SPECjbb2000 varies depending on the execution speed.

Table 5.2: Exploitable thread locality of Java locks

Program	Number of lock acquisitions	Ratio of acquisitions in the first repetitions
SPECjvm98		
_202_jess	14,646,978	99.993%
_201_compress	28,895	97.211%
_209_db	162,117,521	99.9998%
_222_mpegaudio	27,168	98.108%
_228_jack	38,570,415	99.998%
_213_javac	47,062,772	99.974%
_227_mtrt	3,522,926	99.557%
SPECjbb2000[2]	102,282,147	79.392%
Volano Server	7,244,208	75.983%
Volano Client	10,419,671	84.270%

Table 5.3: Investigation of synchronized objects

Program	Objects created	Objects sync'd	Objects sync'd by multiple threads	
SPECjvm98				
_202_jess	23,999,733	21,278	187	(0.878%)
_201_compress	18,586	2,135	127	(5.948%)
_209_db	9,883,475	66,592	52	(0.078%)
_222_mpegaudio	26,456	1,620	91	(5.617%)
_228_jack	19,334,735	1,635,497	144	(0.0088%)
_213_javac	19,140,558	1,192,734	1,760	(0.148%)
_227_mtrt	21,622,389	3,020	114	(3.775%)
SPECjbb2000[2]	18,511,021	2,077,210	176,318	(8.488%)
Volano Server	719,245	7,279	1,888	(25.94%)
Volano Client	3,957,166	4,102	1,640	(39.98%)

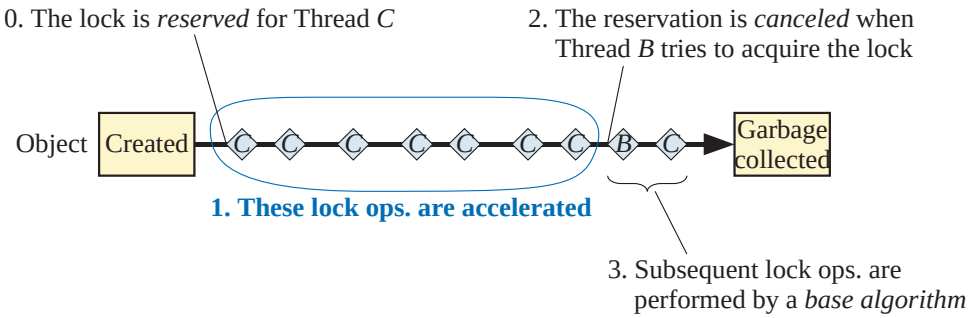

Figure 5.3: Example lock sequence of reservation lock

5.3 The Proposal of Reservation Lock

The experiments in the previous section revealed that for many Java objects their locks are dominantly used by the first locker thread. Therefore, it is confirmed that by giving proper precedence to the first locker, we can accelerate the common case, where the lock is successively used by that thread. This section presents a new lock algorithm called *reservation lock*, which exploits the thread locality,

The key idea of this algorithm is to reserve locks for threads. When a thread attempts to acquire an object's lock, one of the following actions is taken in accordance with the lock's reservation status:

1. If the object's lock is reserved for the thread, the runtime system allows the thread to acquire the lock with a few instructions involving no atomic operations.

2. If the object's lock is reserved for another thread, the runtime system *cancels* the reservation, and falls back to a conventional algorithm for further processing.

3. If the object's lock is *not* reserved, or the reservation was already canceled, the runtime system uses a conventional algorithm.

The overall performance of Java locks is accelerated if most lock operations are processed in the case 1 situation. Figure 5.3 illustrates the behavior of reservation lock when it is applied to Object 2 in Figure 5.2. In this example, the object's lock is reserved for Thread C at its first acquisition, and is accelerated until the second thread, B, attempts to acquire the lock.

As an interesting idea in the reservation-lock implementation, it should be pointed out that the algorithm leaves the processing of the not-reserved case to a conventional, existing lock algorithm (*base algorithm*). By using this approach, the implementation can concentrate on the case where the lock is reserved, while utilizing the results from previous research for the not-reserved case. This idea is similar to that of thin lock, where the OS-provided monitor mechanism is used after a contention once occurs for the object.

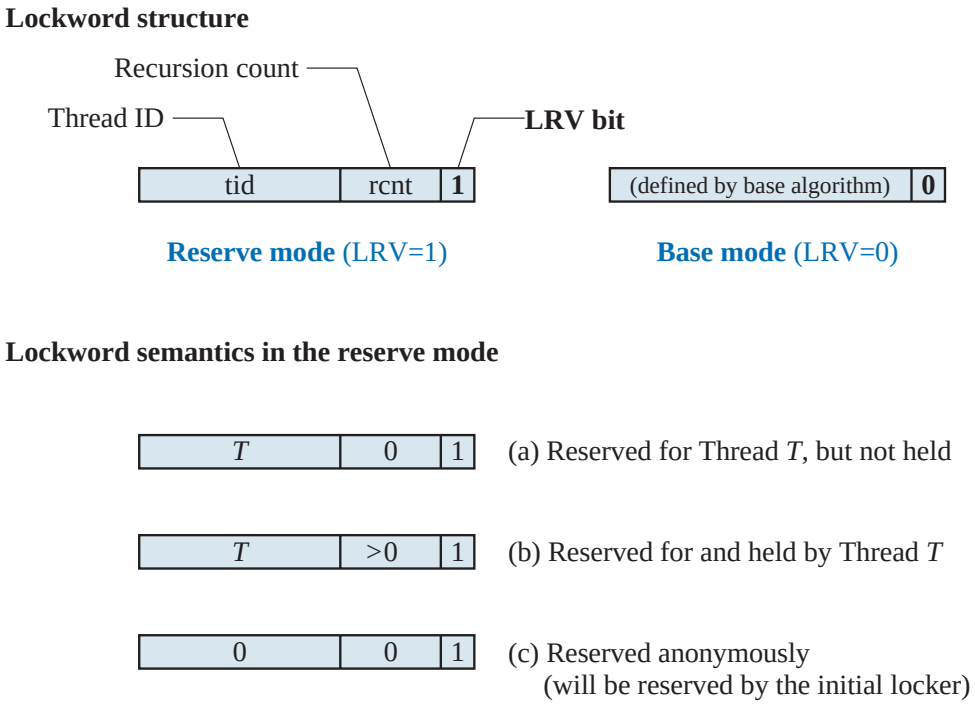

Figure 5.4: Lockword structure and semantics

5.3.1 Data Structure

Our reservation lock can be built on any existing lock algorithm, as long as it uses a *lockword*, a word[3] in the object's header for lock processing, and allows one bit to be available in the lockword. The bit is used for representing the lock's reservation status, and hence named the *LRV* (Lock ReserVed) bit. When the LRV bit is set, the lockword is in the *reserve* mode, and the structure is defined by our algorithm. When the bit is not set, the lockword is in the *base* mode, and the structure is defined by the underlying algorithm that the runtime system falls back to after canceling the reservation.

Figure 5.4 shows the structure of the lockword. When the LRV bit is set, the lockword is in the reserve mode, and is further divided into the thread identifier (`tid`) field and the recursion count (`rcnt`) field. The `tid` field contains an identifier of the *owner thread*, for which the lock is reserved, while the `rcnt` field holds the lock recursion level.

When the `rcnt` field is zero (Figure 5.4(a)), the lock is reserved but not held by any thread. When the field is non-zero (Figure 5.4(b)), the lock is held by the owner thread. As we will see later, the owner thread can acquire the lock by simply incrementing the `rcnt` field, without any atomic operations.

[3]Actually, we don't need the whole 32 bits of the word, and could put other information in the word unrelated to the lock. However, for the sake of explanation, we assume that the whole word is used for lock processing. See Figure 5.9 for the lockword structure used in the actual implementation.

The `rcnt` field is also intended for *recursive lock acquisition*, which is fairly common in Java. The owner thread acquires the lock recursively by incrementing the `rcnt` field, in just the same manner as it initially acquires the lock. We must maintain the recursion count of a lock since Java does not allow a thread to release a lock more times than it acquires the lock. The virtual machine must detect such an illegal state and raise an instance of `IllegalMonitorStateException` [76].

When an object is created, the lock is *anonymously reserved*. That is, the lockword is in the reserve mode, but not reserved for or held by any particular thread (Figure 5.4(c)). This is because the thread for which the lock should be reserved is normally not known at the time of creation. In general, a *reservation policy* determines when and for which thread a lock is reserved. Since we base our algorithm on exploitable thread locality from the previous section, we use the *initial-locker policy* in our algorithm. That is, when an object's lock is acquired for the first time by a thread, we reserve the object's lock for that thread by confirming the anonymous reservation and make it *specifically reserved*.

When the reservation is canceled, the LRV bit is cleared, and the lockword is put in the base mode. The structure is completely defined by the base algorithm. As we will see later, canceling a reservation is the most challenging part of our algorithm, requiring the owner thread to be suspended. The cancellation replaces the lockword in the reserve mode with the corresponding state in the base algorithm. In the reservation lock system, once a lockword is converted to the base mode, it never returns to the reserve mode.

Figure 5.5 depicts the state transitions of the lockword in our algorithm, where the cancellation is shown as *unreserve* arrows. In the figure, the left half shows new states managed by the reservation lock, while the right half is handled by the base algorithm. The thickness of each arrow implies the relative cost of the transition. The next subsection will describe the necessary processing for each transition.

5.3.2 Algorithm

Figure 5.6 shows the algorithm for reservation lock in pseudo-C code, where the functions whose names start with `Base_lock_` are Java lock functions of the base algorithm. We assume that each of the thread-manipulating functions (`thread_suspend`, `thread_resume`, `thread_get_context`, and `thread_set_context`) does nothing and just returns `FAIL` if the target thread does not exist or the specified thread identifier is zero. We also assume that the `thread_suspend` function can be called multiple times for a thread, where the target thread will be resumed after `thread_resume` is called the same number of times[4].

For readability, the code shown here is slightly different from the actual code. For instance, the condition checks in the beginning of the `Java_lock_acquire` and `Java_lock_release` functions are merged into two checks in the actual code. Also, the `Base_`-

[4]Note that the `thread_suspend` and `thread_resume` functions are unrelated to the deprecated Java methods `suspend` and `resume` in the `java.lang.Thread` class.

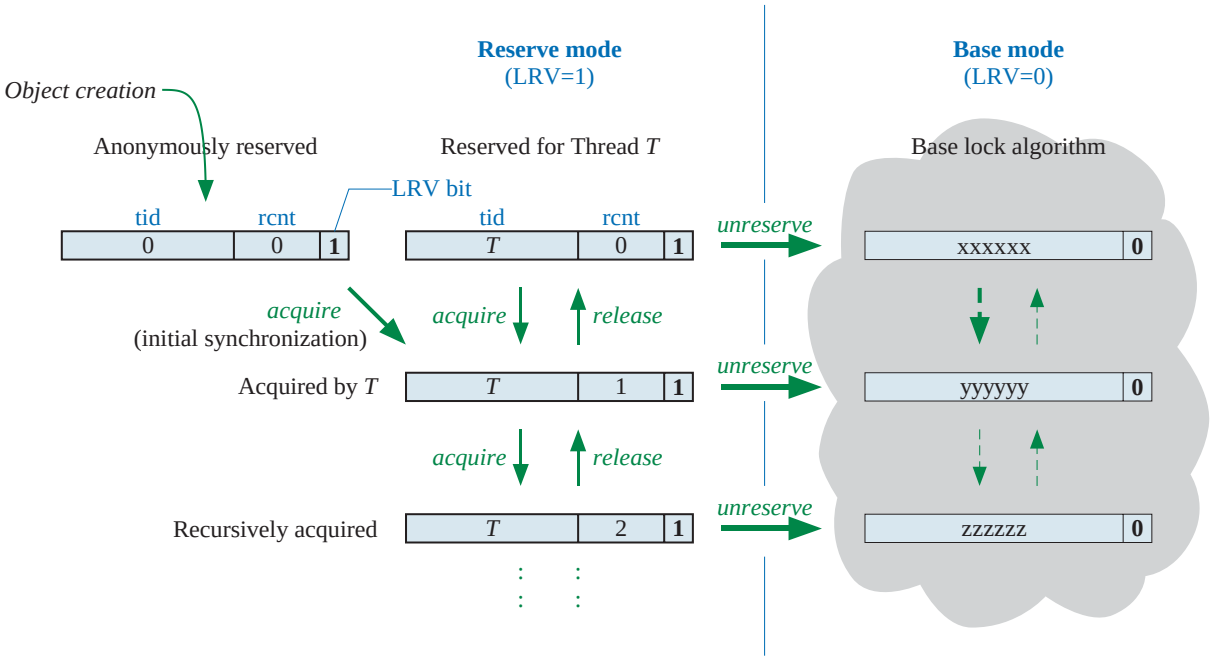

Figure 5.5: Lock state transitions of reservation lock

`lock_acquire` and `Base_lock_release` functions are tightly coupled with the `Java_lock_acquire` and `Java_lock_release` functions, respectively.

Lock Acquisition

A thread attempting to acquire an object's lock calls the `Java_lock_acquire` function, where it reads the lockword, and performs four checks to see if it is not in a special state (lines 23–26). If it passes all of the checks, then the lock is in the most common state where the thread owns the lock's reservation. It completes the lock acquisition by simply incrementing the `rcnt` field (line 28). Thus, if the lock is reserved for the thread, it can be acquired very quickly by simple memory operations with no atomic operations.

There are three special cases where the quick acquisition cannot be performed. First, when the lockword is not in the reserve mode (line 23), the thread executes the corresponding function of the base algorithm, `Base_lock_acquire` (line 43). Second, when the lock is anonymously reserved (line 24), the function attempts to make it *specifically reserved* by using `compare_and_swap` (line 35). Third, when the lock is reserved for another thread (line 25), the thread calls the `unreserve` function to cancel the reservation (line 40), and falls back to the base algorithm. This third special case also results when the thread owns the reservation but the recursion count has reached the maximum value (line 26). We will discuss the reservation cancellation in detail later.

```
 1 : // Object header contains a lockword
 2 : typedef struct object {
 3 :     //    :
 4 :     volatile lockword_t lockword;
 5 :     //    :
 6 : } Object;
 7 :
 8 : // Lockword structure in each object header
 9 : typedef struct lockword {     // [tid:rcnt:R]
10 :     unsigned int tid     : N; // Thread ID of the owner
11 :     unsigned int rcnt    : M; // Recursion count
12 :     unsigned int reserve : 1; // LRV bit
13 : } lockword_t;
14 :
15 : int Java_lock_acquire(Object *obj) {
16 :     lockword_t l1, l2;
17 :     unsigned int myTID = thread_id();
18 :
19 :   retry_acquire:
20 :     l1 = obj->lockword;   // read the lockword   ----------------------(1)
21 :                                                                  //   A
22 :     // check special cases                                       //   |
23 :     if (l1.reserve == 0)     goto base_acquire;                  //   |
24 :     if (l1.tid == 0)         goto make_specific;                 // unsafe
25 :     if (l1.tid != myTID)     goto unreserve_and_base;            // region
26 :     if (l1.rcnt == RCNT_MAX) goto unreserve_and_base;            //   |
27 :                                                                  //   |
28 :     // reserved for me, and rcnt < RCNT_MAX                      //   |
29 :     l2 = l1; l2.rcnt++;                                         //   V
30 :     obj->lockword = l2;   // write the lockword  ----------------------(2)
31 :     return SUCCESS;
32 :
33 :   make_specific:
34 :     l2 = l1; l2.tid = myTID; l2.rcnt = 1;
35 :     if (compare_and_swap(&obj->lockword, l1, l2) != SUCCESS)
36 :         goto retry_acquire;
37 :     return SUCCESS;
38 :
39 :   unreserve_and_base:
40 :     unreserve(obj, l1.tid, myTID);
41 :
42 :   base_acquire:
43 :     return Base_lock_acquire(obj);
44 : }
45 :
```

Figure 5.6: Algorithm of reservation lock (1 of 3)

```
46 : int Java_lock_release(Object *obj) {
47 :     lockword_t l1, l2;
48 :     unsigned int myTID = thread_id();
49 :
50 :   retry_release:
51 :     l1 = obj->lockword;   // read the lockword   ----------------------(1)
52 :                                                                    //   A
53 :     // check special cases                                         //   |
54 :     if (l1.reserve == 0)     goto base_release;                    //   |
55 :     if (l1.tid != myTID)     goto illegal_state;                   // unsafe
56 :     if (l1.rcnt == 0)        goto illegal_state;                   // region
57 :                                                                    //   |
58 :     // reserved for and held by me                                 //   |
59 :     l2 = l1; l2.rcnt--;                                            //   V
60 :     obj->lockword = l2;   // write the lockword  ----------------------(2)
61 :     return SUCCESS;
62 :
63 :   illegal_state:
64 :     return ILLEGAL_STATE;
65 :
66 :   base_release:
67 :     return Base_lock_release(obj);
68 : }
69 :
70 : // Unreserve an object's lock
71 : void unreserve(Object *obj, unsigned int ownerTID, unsigned int myTID) {
72 :     lockword_t l1, l2;
73 :     struct Context context;
74 :
75 :     if (ownerTID == myTID) ownerTID = 0;
76 :     thread_suspend(ownerTID);
77 :
78 :   retry_unreserve:
79 :     l1 = obj->lockword;
80 :     if (l1.reserve == 0) goto already_unreserved;
81 :     l2 = Base_equivalent_lockword(l1);
82 :     if (compare_and_swap(&obj->lockword, l1, l2) != SUCCESS)
83 :         goto retry_unreserve;
84 :
85 :     // modify the owner's context if it's in an unsafe region
86 :     if (thread_get_context(ownerTID, &context) == SUCCESS) {
87 :         if (In_unsafe_region(context.pc)) { // if (1)<NextPC<=(2)
88 :             context.pc = Get_retry_point(context.pc); // get the corresponding
89 :             thread_set_context(ownerTID, &context);   //           retry point
90 :         }
91 :     }
92 :
93 :   already_unreserved:
94 :     thread_resume(ownerTID);
95 : }
96 :
```

Figure 5.6: Algorithm of reservation lock (2 of 3)

```
 97 : int Java_lock_wait(Object *obj) {
 98 :     lockword_t l1 = obj->lockword;
 99 :     unsigned int myTID = thread_id();
100 :
101 :     if (l1.reserve == 1) {
102 :         if (l1.tid != myTID || l1.rcnt == 0) return ILLEGAL_STATE;
103 :         unreserve(obj, l1.tid, myTID); // actually, l1.tid == myTID
104 :     }
105 :     return Base_lock_wait(obj);
106 : }
107 :
108 : int Java_lock_notify(Object *obj) {
109 :     lockword_t l1 = obj->lockword;
110 :     unsigned int myTID = thread_id();
111 :
112 :     if (l1.reserve == 1) {
113 :         if (l1.tid != myTID || l1.rcnt == 0) return ILLEGAL_STATE;
114 :         return SUCCESS; // no one should be waiting, no need to unreserve
115 :     }
116 :     return Base_lock_notify(obj);
117 : }
118 :
119 : // Java_lock_notify_all() is implemented in the same manner
```

Figure 5.6: Algorithm of reservation lock (3 of 3)

Lock Release

Similarly, a thread attempting to release an object's lock calls the `Java_lock_release` function, where it first reads the lockword, and performs three checks to see if it is not in a special state (lines 54–56). If it passes all the checks, then the function finishes the lock release by simply decrementing the `rcnt` field (line 60). This is the fastest and most common case in releasing the lock.

There is only one *legal* special case in the `Java_lock_release` function. That is, when the lockword is not in the reserve mode (line 54), the function invokes the corresponding function in the base algorithm, `Base_lock_release` (line 67). The other two checks in lines 55 and 56 are for detecting illegal states in the reserve mode, where the lock is not currently held by the thread. For these cases, the algorithm returns the `ILLEGAL_STATE` error (line 64), and the JVM will raise an instance of `IllegalMonitorStateException` in compliance with the Java language specification [41, 76].

We should note that the `unreserve` function need not be called in releasing the lock, unlike in acquiring the lock. This is because it can never happen in our algorithm that a currently-acquired lock is reserved for another thread.

Reservation Cancellation

We now explain cancellation of a reservation, the most complicated part of our algorithm, which the `unreserve` function (lines 71–95) is responsible for. Basically, a thread calls this function when the thread attempts to acquire a lock which is reserved for another thread[5]. The function atomically replaces the lockword in the reserve mode with the equivalent state in the base algorithm (line 81), by using `compare_and_swap` (line 82) to prevent multiple non-owner threads from succeeding in the replacement.

However, since the reservation-owner thread can access the lockword with non-atomic memory operations, special care must be taken to cancel the reservation when the owner is in the middle of the accelerated lock processing, more specifically, when it is in one of the *unsafe regions* which are between the read (1) and write (2) of the lockword in the `Java_lock_acquire` (lines 20–30) and `Java_lock_release` (lines 51–60) functions. If the owner thread stays in the unsafe region when the `unreserve` function replaces the lockword, the lockword will be overwritten by the thread at the end of the region, which ends with an inconsistent status of the lockword.

To avoid such data race conditions, the `unreserve` function first suspends the owner thread (line 76) before replacing the lockword. When the lockword is successfully replaced using the atomic operation, the function then obtains the execution context of the suspended owner thread (line 86) to see whether the thread is in one of the unsafe regions. If it is suspended inside an unsafe region, the function modifies the program counter of the thread with the address of the corresponding *retry point* (line 19 or 50). After guaranteeing that the suspended thread is outside of unsafe regions by using this technique, the function resumes the thread (line 94).

This cancellation mechanism is the key of our reservation lock. Notice that each unsafe region is carefully designed to have the following important properties:

Property 5.1 *The unsafe region is formed as a continuous sequence of instructions.*

Property 5.2 *The unsafe region contains no instruction that causes side effects.*

Property 5.1 makes it easy to check if the suspended thread is inside the region. Property 5.2 makes it possible to move the thread's processing to the retry point from an arbitrary point in the region. Figures 5.7 and 5.8 illustrate the operations of `Java_lock_acquire` and `Java_lock_release` functions with their unsafe regions, respectively. The number beside each box corresponds to the line number in the algorithm of Figure 5.6.

In our cancellation algorithm, if the suspended thread is in an unsafe region, the thread is always moved to the retry point even if it is processing another object's lock. However, this does not cause any inconsistency because the region is restartable without any side effect by the Property 5.2 stated above. This also does not cause any additional

[5]The `unreserve` function is also called when the `rcnt` is about to overflow or when the `Java_lock_wait` function is called.

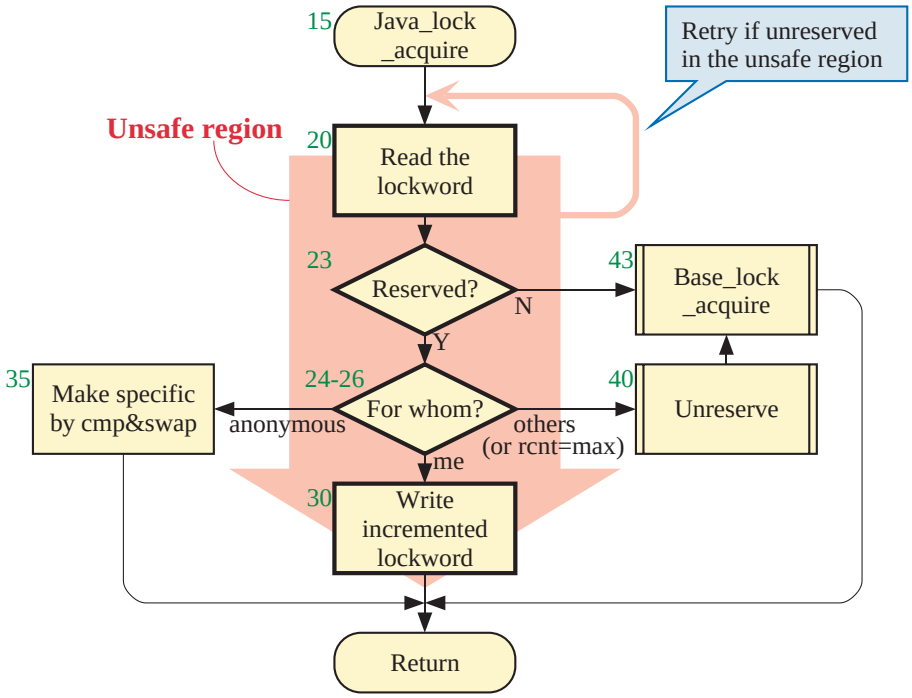

Figure 5.7: Flow of `Java_lock_acquire`, and its unsafe region

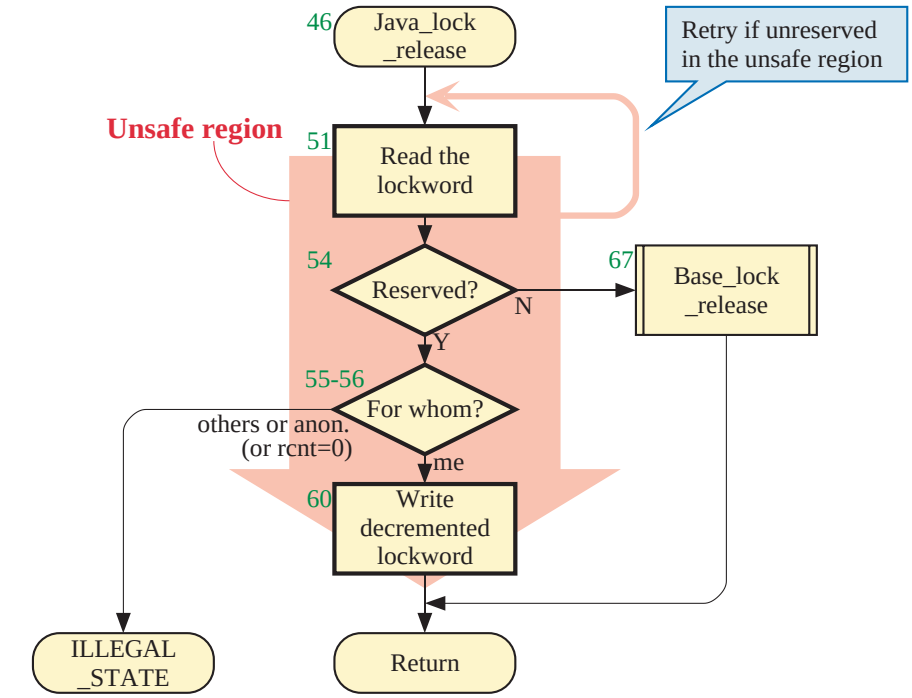

Figure 5.8: Flow of `Java_lock_release`, and its unsafe region

noticeable performance degradation, because it seldom happens that a thread is inside an unsafe region, since the regions are very short.

Wait and Notification

Finally, we briefly explain the support for the `Java_lock_wait` and `Java_lock_notify` functions. Since the reservation lock itself does not support the event notification mechanism, `Java_lock_wait` enforces the reservation cancellation (line 103) to process the wait function using the base algorithm, `Base_lock_wait` (line 105).

In contrast, the `Java_lock_notify` function simply returns (line 114) without forcing the cancellation if the lock is in the reserve mode, since this status means no thread has called `Java_lock_wait` for the lock. If the lock is in the base mode, the `Base_lock_notify` function is called (line 116). The `Java_lock_notify_all` function is implemented in the same manner.

This approach is basically same as that used in the tasuki lock, which forces inflation in `Java_lock_wait` but does not in `Java_lock_notify`, as shown in Section 4.5.4.

5.4 Discussion

This section discusses the reservation lock algorithm described in the previous section from various viewpoints.

5.4.1 Correctness

First of all, the correctness of our algorithm should be discussed. As we have shown, a thread does not have to execute any atomic operation in acquiring and releasing a lock when it owns the reservation. In other words, the owner thread can read-modify-write the lockword without atomic operations. Thus, when a different thread attempts to change the lockword between the read and the write (i.e. in an unsafe region), special care must be taken to prevent the modification from being lost. Otherwise, the lock state would become inconsistent.

When a thread does not own a lock's reservation, our algorithm requires the thread call the `unreserve` function, where the non-owner thread modifies the lockword after suspending the owner thread. When the owner thread is suspended in the middle of an unsafe region, it is forced to restart the unsafe region, and will detect that it no longer has the reservation. This prevents the thread from continuing its execution based on the no-longer-valid assumption that it still owns the reservation.

The owner thread may have already completed the computation and ceased to exist when another thread attempts to cancel a reservation. Although the `unreserve` must also

handle this case properly[6], there is no risk of a data race condition involving the owner thread.

More than one thread may simultaneously try to make an anonymous reservation specific (line 35) or try to convert the lockword in the reserve mode to the base mode (line 82). However, it is guaranteed that only one thread eventually succeeds, since atomic operations (`compare_and_swap`) are used in both cases.

Once the reservation is canceled, the lockword will never be reserved again. Thus, after the cancellation, our algorithm behaves in exactly the same manner as the base algorithm, and the correctness is ensured by the correctness of the base algorithm.

In certain sequences of thread scheduling, it may happen that a thread tries to modify a lockword in reserve mode even after the reservation is canceled. However, in such situations, making the anonymous reservation specific (line 35) and converting the lockword to the base mode (line 82) do not succeed because they are performed by using `compare_and_swap`. Writing to a lockword under reservation (lines 30 and 60) is performed by a simple store instruction, but this cannot be executed after the cancellation, since it forces the owner thread to be outside of the unsafe region.

5.4.2 Performance Characteristics

Next, we discuss the performance characteristics of the reservation lock. Our algorithm is strongly expected to reduce the synchronization overhead when the reservation succeeds, since the owner thread can acquire and release the lock by simply reading and writing the lockword without any atomic operations.

When a lock is not reserved, our algorithm falls back to the base algorithm with almost no additional overhead. It simply requires two additional checks for the reservation status, one in the `Java_lock_acquire` function (line 23) and the other in the `Java_lock_release` function (line 54). However, depending on the details of the base algorithm, we can completely eliminate the additional overhead. That is, if the base algorithm starts the lock acquisition and the lock release by testing one or more bits in the lockword, we may be able to merge the additional checks of our algorithm into the testing. Actually, this is the case in our implementation that we will present in Section 5.5. No additional computation is necessary for the base method, since the lockwords are never re-reserved once the reservation is canceled.

The greatest concern in terms of performance is reservation cancellation in the `unreserve` function, which relies on expensive system calls such as `thread_suspend` and `thread_get_context`. However, since we do not reserve locks repeatedly, the cancellation occurs at most once during the lifetime of an object. As we will show in the next section,

[6] As described at the first part of Section 5.3.2, we assume that thread-manipulating functions do nothing and just return `FAIL` if the target thread does not exist. This makes our `unreserve` implementation work properly even if the owner thread is terminated. In addition, it does not cause any error even if the terminated thread's identifier is reused.

the ratios of cancellations to lock acquisitions are less than 0.05% in actual lock-intensive programs. Thus, we believe that performance loss from cancellation does not offset the performance gain by reservation success except for artificially created pathological benchmarks[7].

As mentioned above, in our algorithm, once a lock's reservation is canceled, subsequent lock operations are performed by the base algorithm, and the lock never returns to the reserve mode. This is for the following reasons:

1. The investigation in the previous section shows that there is enough exploitable thread locality even without re-reservation.

2. Once a reservation is canceled, it is usually useless to re-reserve it. Moreover, the re-reservation may cause more cancellations, which degrade performance.

3. The algorithm to support the re-reservation was expected to be too complicated to achieve the performance goals.

5.4.3 Unsafe Regions

If a thread always acquires and releases an object's lock by calling the runtime functions `Java_lock_acquire` and `Java_lock_release`, respectively, we have only two unsafe regions in the virtual machine. The `In_unsafe_region` function (line 87) only has to perform two range checks, which is easy to implement.

However, the JIT compiler may inline the synchronization operations into the generated code. This can result in *many* unsafe regions in the virtual machine, which we must register in a data structure with the corresponding retry addresses. Given a program counter, the `In_unsafe_region` function searches the data structure to see if the program counter points to any unsafe region.

Alternatively, we could use the *designated code sequence* approach by Bershad et al. [15]. When the JIT compiler inlines a lock operation, it can embed with each unsafe region some *landmark code pattern*, which is a special code pattern not generated for other purposes. Since the code sequence for lock processing is basically fixed, the distance to the landmark can be known from each instruction in the unsafe region. The `In_unsafe_region` function searches for the landmark at the address anticipated from the instruction at the program counter. If the landmark code pattern is successfully found, it means that the thread is suspended inside an unsafe region.

Whatever techniques are used, we need to obtain the program counter of a suspended thread by invoking an appropriate system call, which is expensive in most operating systems. Thus, it is desirable to reduce the number of calls to the `thread_get_context` function at line 86. If the virtual machine provides a fast way to see if the thread is in

[7] We will revisit this issue in the next chapter.

a module of JIT-compiled code, we can reduce the number of calls by creating unsafe regions only in the compiled code.

Some virtual machines allow us to cheaply determine, without the program counter, whether a thread has been suspended within the module of compiled code. For instance, the virtual machine we use in Section 5.5 maintains a thread local variable for the thread's *execution mode*. The variable takes values such as `EXECUTING_COMPILED_CODE`, `COMPILING`, and `INTERPRETING`. We can thus know if the thread is in the module of compiled code by simply checking the current value of the thread local variable.

On the other hand, we can confine unsafe regions to the module of compiled code as follows. In general, we can convert an unsafe region into a safe region by modifying the lockword with a `compare_and_swap` even in the reserve mode (lines 30 and 60). Although we should not make such conversions for frequently executed unsafe regions since it sacrifices the gain by reservation, it is reasonable to convert the unsafe regions in the Java bytecode interpreter and other performance insensitive components.

Putting these two approaches together, we can, in our virtual machine, use the following sequence in the `unreserve` function.

```
   85 :     // modify the owner's context if it's in an unsafe region
|  85a:   if (get_exec_mode(ownerTID) == EXECUTING_COMPILED_CODE) {
   86 :     if (thread_get_context(ownerTID, &context) == SUCCESS) {
   87 :         if (In_unsafe_region(context.pc)) { // if (1)<NextPC<=(2)
   88 :             context.pc = Get_retry_point(context.pc); // get the corresponding
   89 :             thread_set_context(ownerTID, &context);   //           retry point
   90 :         }
|  91 :   } }
```

The quick check in line 85a is expected to filter out many uninteresting cases, such as the thread is in the interpreter or is waiting for some event, resulting in many fewer calls to the `thread_get_context`.

5.4.4 Reservation Cancellation

The essential property in the reservation cancellation is to prevent the owner thread from changing the lockword while another thread is canceling the reservation. As long as this property is satisfied, we could implement the `unreserve` function in different ways. We show two variations of the function here.

First, there may be a case where functions such as `thread_suspend` and `thread_get_context` are not available in the underlying operating system. In this case, we could use *signals* as provided in Unix operating systems. In this variation, the non-owner thread just requests the cancellation, while the owner thread actually does the cancellation by itself. More concretely, the non-owner thread sends a signal to the owner thread, and waits until the latter has completed the processing. In the signal handler, the owner thread converts the lockword to the base mode, checks with the saved program counter to

see if it has been interrupted in an unsafe region, and, if so, modifies the program counter to the corresponding retry address.

Second, more efficient implementations may become possible by utilizing a processor's special operations. As an example, we could exploit *predicated stores*[8], which are available, for instance, on Intel's IA-64 processors [56]. We dedicate one predicate register in the JVM process to reservation lock. We initialize it to `TRUE` before reading the lockword in `Java_lock_acquire` (line 20) and `Java_lock_release` (line 51), while we write into the lockword in the reserve mode (lines 30 and 60) with a predicated store qualified by the predicate register. In the `unreserve` function, we always set the predicate register of the owner thread to `FALSE`. This prevents the owner thread from changing the lockword inconsistently[9].

5.4.5 Multiprocessor Considerations

The Java language specification [41] describes the Java memory model in Chapter 17. According to the rules about the interaction of locks and variables, we cannot move before a lock acquisition the load operations that follow the acquisition or move after a lock release the store operations that precede the release. Therefore, when we implement the reservation lock on a multiprocessor system with a relaxed memory model [1], we need to issue appropriate types of memory barriers in the functions for lock acquisition and release.

Practically speaking, we believe that these memory barriers are unnecessary in the reserve mode, since no other thread can be trying to execute the same critical region. We can take care of the necessary synchronizations when the reservation is canceled and while the owner thread is suspended.

Finally, we note that Pugh pointed out flaws in the Java memory model [94], and that revisions have been discussed under Java Specification Request (JSR) 133 [60], which was recently adopted in JDK 5.0 [8, 78, 105].

5.5 Performance Results

This section evaluates the effectiveness of the reservation lock with the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1 [53]. The JDK is same as that used

[8]In the IA-64, most operations can be qualified by a one-bit predicate register to indicate whether it is actually executed or not. The execution of a predicated store consists of checking the predicate register and conditionally performing the store, and cannot be interrupted in the middle. For example, an instruction "`(p1) st4 [r35]=r37`" executes an operation same as "`if (p1) *r35 = r37`" in C. That is, the content of register `r37` is stored to a memory pointed by `r35`, *only when* the predicate register `p1` is `TRUE`.

[9]Hudson et al. proposed a similar technique for object allocation, which utilizes dedicated predicate registers set and reset by the context switcher [48].

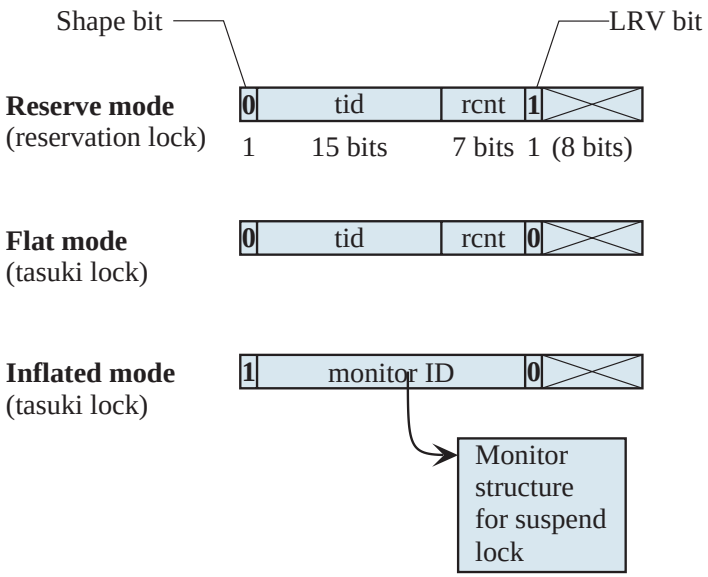

Figure 5.9: Actual structure of lockword in the reservation lock implementation

in the investigation in Section 5.2, but this time we enabled its JIT compiler [58, 102, 103] with appropriate modifications for the reservation lock.

5.5.1 Actual Implementation

As the base algorithm combined with our reservation lock, we used the *tasuki lock* [88], as used in the developer kit. As described in Chapter 4, by using the tasuki lock, a thread can acquire a lock with a `compare_and_swap` and release the lock with a simple memory store operation if the lock is not contended among threads (*flat* mode). When contention happens, the lock is temporarily converted to the *inflated* mode, where lock operations are performed by using a heavyweight monitor which supports the thread suspension. The lock returns to the flat mode after the contention ceases.

Although our reservation lock can be built upon any algorithm, tasuki lock is a very natural fit since the lockword structure in the flat mode is almost the same as the structure in the reserve mode. This allows lock operations to be highly efficient in terms of both space and time. As shown in Figure 4.2, the actual implementation of tasuki lock uses a 24-bit lockword, where the 15-bit `tid` and 8-bit `rcnt` fields are already defined in its flat mode. We squeezed in the LRV bit for reservation lock by reducing the `rcnt` field to 7 bits. Figure 5.9 shows the actual lockword used in the implementation, where the lockword is used *tri-modally*.

Since the `tid` and `rcnt` fields are used in both lock methods, we could easily implement the `Base_equivalent_lockword`, which is a function to generate the equivalent lockword for the base mode at the time of reservation cancellation (line 81), by just removing the

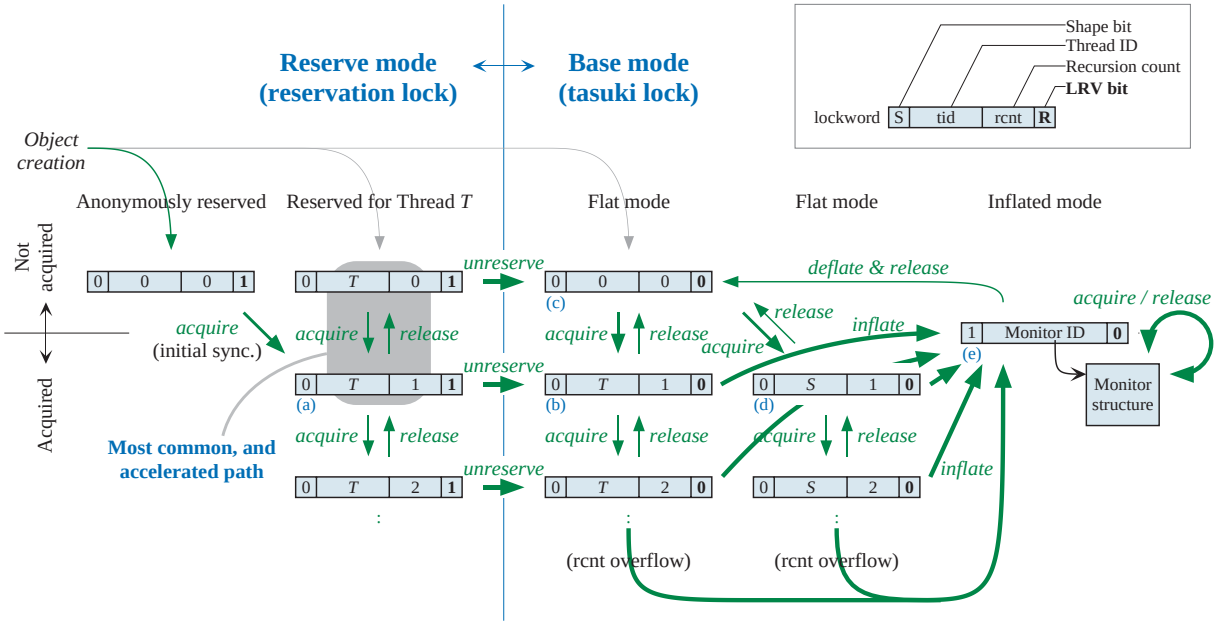

Figure 5.10: All lock state transitions when reservation is coupled with tasuki lock

reservation information as follows[10]:

```
1 : lockword_t Base_equivalent_lockword(lockword_t l1) {
2 :     lockword_t l2 = l1;
3 :     l2.reserve = 0;                // clear the LRV bit
4 :     if (l1.rcnt == 0) l2.tid = 0; // clear the reservation owner info
5 :     return l2;                     // return the new lockword for tasuki lock
6 : }
```

Figure 5.10 illustrates all of the state transitions when the reservation lock is coupled with the tasuki lock. In the figure, the thickness of each arrow informally represents the cost of the transition. While a lock is reserved for Thread T (i.e. the lock is used only by T), it is processed in the reserve mode, which is the left half of the figure. When the second thread, S, attempts to acquire the lock, the reservation is canceled and subsequent lock operations are processed by tasuki lock, which is the right half of the figure.

For example, if S attempts the acquisition while T is holding the lock in reserve mode, the lockword is converted from (a)→(b) by the cancellation. When T releases the lock, the content of the lockword changes along the path of (b)→(c)→(d)→(e), and the inflated

[10]In the original implementation of tasuki lock, an outermost lock acquisition just sets the thread identifier to the `tid` field and leaves the `rcnt` field as 0. We slightly modified the implementation to also set the field to 1 at the outermost acquisition, to improve affinity with the reservation lock, where the `rcnt` is set to 1 at the outermost acquisition. The modified tasuki lock code is omitted here since it is trivial.

lock is acquired by Thread S. Since tasuki lock supports the deflation of the lockword, the lockword will be converted from (e)→(c) if appropriate conditions are satisfied when S releases the lock.

We took a simple approach to implementing checks for unsafe regions. Our virtual machine includes two sets of implementations of `Java_lock_acquire` and `Java_lock_release` functions, one pair in the module of the JIT runtime code and the other pair in the module of the interpreter. The JIT-runtime version is called from but not inlined into the JIT generated code. The interpreter version, written in C, is called from the interpreter, and implemented without unsafe regions as described in Section 5.4.3. This means we only have two unsafe regions in our virtual machine. To make the performance comparison exact, we disabled inlining of the lock acquisition and release code in the original virtual machine.

In order to comply with the Java memory model, we inserted the `lfence` and `sfence` instructions of Pentium 4 [55] into the functions for lock acquisition and release, respectively.

We ran all of the benchmark programs under Windows 2000 SP2 on an unloaded IBM IntelliStation M Pro containing two 1.7 GHz Pentium 4 Xeon processors with 1,024 megabytes of main memory.

5.5.2 Micro-Benchmarks

First, the basic performance of reservation lock was measured by using the following two micro-benchmarks. We confirmed in both tests that the relevant methods were JIT-compiled to native code, and that the synchronization operations within the methods were not optimized away by the JIT compiler. We also verified that garbage collection did not occur during the measurements.

PrimitiveTest

The first micro-benchmark, `PrimitiveTest`, is intended for measuring the cost of synchronization, that is, of acquiring and releasing a lock, in different lock states. We measured the following two cases in three lock states — reserved, not reserved (flat), and inflated:

- *Outermost*: Acquire and release a lock using a synchronized block n times, and measure the elapsed time.
- *Recursive*: Perform the same measurement inside another synchronized block.

To identify the cost of acquiring and releasing a lock in each state, we created a special virtual machine that does nothing on lock acquisition or release, and calculated the differences between the times of the normal and special virtual machines.

Table 5.4: Synchronization costs in reservation lock

	Is recursive?	
Lockword state	Outermost	Recursive
Reserved	61.4 nsec	61.4 nsec
Not reserved (flat)	229.5 nsec	61.4 nsec
Inflated	335.5 nsec	155.8 nsec
Flat in original	228.9 nsec	62.2 nsec
Inflated in original	330.3 nsec	150.0 nsec

Table 5.5: Costs of lock state transitions

State transition	Time
Anonymous-to-specific	89.0 nsec
Reserved-to-base (faster case)	6,741 nsec
Reserved-to-base (slower case)	18,986 nsec

Table 5.4 shows the results. For comparison, the table also contains the numbers for the original tasuki lock algorithm without reservation. When the reservation succeeds, we dramatically reduced the cost of the outermost synchronization by more than 70%. Although the profit from each successful reservation is small, we believe that it can accumulate to result in visible performance improvements because there are many lock-intensive Java programs [32], as shown in Section 5.2. On the other hand, after the reservation is canceled, the cost of the synchronization is almost the same as in the original algorithm.

TransitionTest

The second micro-benchmark, `TransitionTest` measures the cost of transitions of lock states unique to reservation lock. We created a total of n objects, and forced them to make the following two transitions.

- *Anonymous-to-specific*: Acquire and release the lock of each object, making the anonymously reserved lock specifically reserved.
- *Reserved-to-base*: Cancel the lock reservation for each object by creating another thread and having this second thread acquire and release the lock.

To calculate the cost of each transition, we took the differences from the times of lock acquisition and release without the transitions.

Table 5.5 presents the results. These numbers show the cost of the transitions, and the time of lock acquisition itself is not included. Since we found that the cost of cancellation

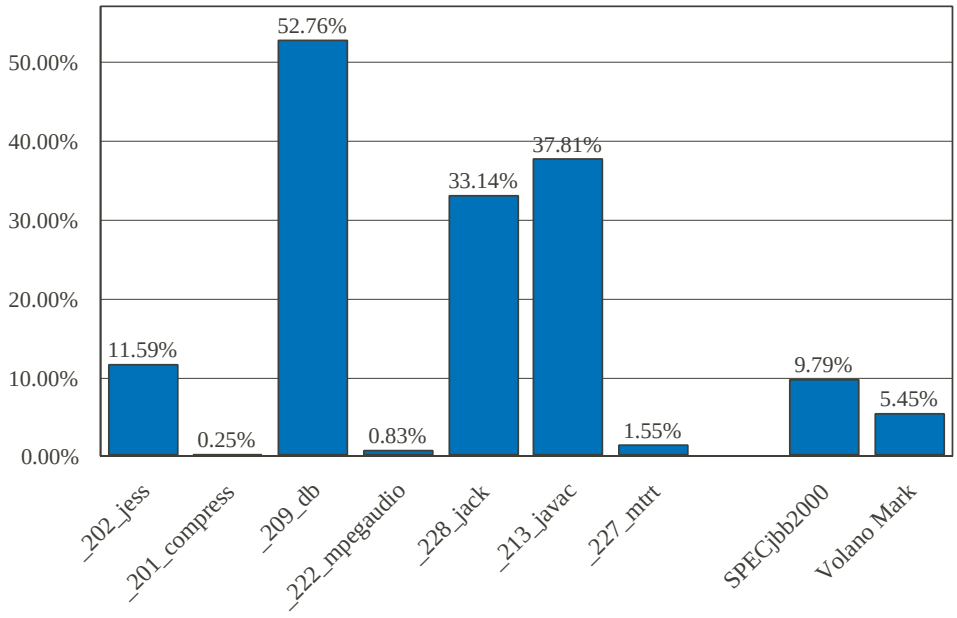

Figure 5.11: Performance improvements by reservation lock

heavily depends on whether or not `thread_get_context` (line 86 in Figure 5.6) is actually executed, we show two cases for the cancellation in the table, a faster case in which the function is not executed and a slower case in which the function is executed.

As the table shows, the cost of making an anonymous reservation specific is very small and negligible, while the cost of reservation cancellation is very large, as expected. The cost of cancellation is 20 to 60 times larger than the cost of outermost synchronization in the inflated mode. One reason for this is that getting a thread context (by calling `GetThreadContext`) is very slow in Windows. It is important to continue to pursue a better implementation for reducing the cancellation cost.

5.5.3 Macro-Benchmarks

We now show the performance improvement in real programs. We measured the performance of the same set of programs as in our investigation in Section 5.2. The overview of each benchmark was listed in Table 5.1. We ran each program several times with two virtual machines, one with the original tasuki-lock algorithm and the other with reservation lock, and compared the best scores. We took the measurements with the JIT compiler enabled including its various lock elimination techniques.

Figure 5.11 shows the results, which are the ratios of the performance improvements by reservation lock compared to the original tasuki lock. The reservation lock improved the performance of all programs except for `_201_compress` and `_222_mpegaudio`, both of

Table 5.6: Reservation lock statistics in the benchmarks

Program	Number of lock acquisitions	Ratio of accelerated lock acquisitions	Ratio of reservation cancellations
SPECjvm98			
_202_jess	14,585,409	99.289%	0.00125%
_201_compress	29,150	31.547%	0.419%
_209_db	162,079,177	99.963%	0.0000296%
_222_mpegaudio	27,480	35.837%	0.313%
_228_jack	35,207,339	91.947%	0.000395%
_213_javac	43,510,883	99.402%	0.00403%
_227_mtrt	3,523,262	99.035%	0.00284%
SPECjbb2000[11]	335,718,621	58.544%	0.0535%
Volano Server	6,862,014	79.755%	0.0248%
Volano Client	10,381,000	84.333%	0.0138%

which perform very few lock operations. We observed especially significant improvements of more than 30% in _209_db, _228_jack, and _213_javac. As a result, reservation lock improved the geometric mean of the SPECjvm98 programs by 18.13%. In addition, we observed improvements of 5 to 10% even in the multi-threaded programs, SPECjbb2000 and the Volano Mark.

Table 5.6 shows lock statistics in the actual environment, which we measured separately. As the table shows, even when the JIT compiler is enabled, many lock operations are performed. The table also shows the ratios of lock acquisitions accelerated by our implementation of reservation lock. Note that these numbers do not include synchronizations performed inside the interpreter or performed recursively in the compiled code, even if the reservations were successful. Because of this, most of the lock acquisitions were not accelerated in _201_compress and _222_mpegaudio, since they were not in hot methods and were executed by the interpreter rather than compiled by the JIT. For other, lock-intensive programs, more than 58% of the lock acquisitions were accelerated by reservation lock.

As already mentioned in Section 5.2, Volano Mark programs cannot be accelerated if we adopt the *creator* reservation policy, where a lock is reserved for the thread that creates the object. As a reference, we also measured the lock statistics as in Table 5.6 while implementing this policy. The ratios of accelerated lock acquisitions were significantly reduced to 0.543% in Volano Server and to 0.126% in Volano Client.

[11] Again, the total number of locks for SPECjbb2000 is not very meaningful because it varies with the execution speed.

5.5.4 Possible Extensions

As the results of the micro- and macro-benchmarks show, the implementation of reservation lock significantly improves performance if the reservation succeeds, while it maintains comparable performance if the reservation fails. The only problem is a relatively high cost in canceling a reservation, which occurs when a thread acquires an object's lock reserved for another thread. However, as Table 5.6 shows, canceling a reservation rarely happens in real programs. Although the locks are initially put in the reserve mode in our implementation, less than 0.05% of lock acquisitions caused reservations to be canceled in the lock-intensive benchmarks.

There might be pathological programs in which reservations are canceled more frequently. It may be important to lower the cost of a cancellation, and to reduce the number of cancellations by refining the reservation policy.

For example, if dynamic profiles of cancellations uncover that reservations are frequently canceled for objects of specific classes or created at specific execution points, we should initially put them into the base mode. Also, we may be able to predict which thread is likely to initially acquire an object's lock, using dynamic profiles or static analysis. The gray arrows at the object creation in Figure 5.10 indicate such cases as reserving (or not reserving) the object's lock in advance.

Finally, if we can reduce the cost of a cancellation, it could become worthwhile to pursue an algorithm allowing repeated reservations.

5.6 Summary

We have presented a new lock algorithm, reservation lock, which optimizes Java locks by exploiting thread locality.

The algorithm allows locks to be reserved for threads, and runs in either reserve mode or base mode. When a thread attempts to acquire a lock, it can do so extremely quickly without any atomic operations if the lock is reserved for that thread. If the lock is reserved for another thread, it cancels the reservation and falls back to the base mode.

We have defined thread locality of locks, which means that the lock sequence of a lock contains a very long repetition of a specific thread, and confirmed that the vast majority of Java locks exhibit thread locality.

We have evaluated an implementation of reservation lock in IBM's production virtual machine and JIT compiler. The results of micro-benchmarks show that we could reduce the lock overhead by more than 70% when the reservation succeeded. The results of macro-benchmarks show that reservation lock accelerated more than 58% of the lock acquisitions, and achieved up to 53% performance improvements in real Java applications.

To summarize, the research shown in this chapter contributes the following results:

- Discovery of the *thread locality* of Java locks
  We found that many objects' locks are repeatedly acquired only by the same thread specific to each object, even in multi-threaded programs.

- Proposal of *reservation lock*
  To exploit the above-mentioned thread locality, we proposed an asymmetric lock method that reduces the lock cost by giving precedence of *reservation* to a specific thread.

- Implementation and measurement
  We showed an implementation of reservation lock which can be easily combined with existing lock methods with minimum overhead, and observed performance improvements with actual Java programs on a state-of-the-art Java environment.

Chapter 6

Asymmetric Lock

6.1 Introduction

The reservation lock shown in the previous chapter is a novel optimization for Java locks, which does not require any atomic operations in common cases [65, 66]. We observed that most Java locks exhibit *thread locality*, that is, each lock tends to be dominantly acquired and released by a specific thread. Attempting to exploit this observation, we *reserve* a lock for such a dominant thread, or let the dominant thread be the *owner* of the lock. The owner thread of a lock can acquire and release the lock without any atomic operations, resulting in a significantly higher performance on a reservation hit.

However, when a non-owner thread attempts to acquire the lock, the algorithm requires the non-owner thread to *cancel* the reservation by suspending the owner thread, which incurs a significant performance penalty. Although few reservations were canceled in the benchmarks we measured, this weakens the robustness and allows for pathological behaviors.

Although it is true that atomic operations such as `compare_and_swap` (CAS) are usually much heavier than other general instructions, their relative costs differ among processors or system configurations. Therefore, in the combination of a processor whose atomic operations are relatively cheap and an operating system whose thread-manipulating functions are heavy, there may be situations such that the overhead of reservation cancellation becomes visible even in actual Java programs. In addition, in some scientific programs which strictly maintain the mutual exclusion among threads by themselves without depending on standard Java libraries, there may be cases where the ratio of cancellations increases even if the program is written in Java.

Based on these viewpoints, in this chapter, we propose a new reservation-based lock algorithm for Java which does not require reservation cancellation [67, 89]. We derive the algorithm in two steps.

First, applying the concept of reservation to spin lock for the first time, we develop a new algorithm, called the *asymmetric spin lock*. Interestingly, this new algorithm is a

hybridization of a CAS-based spin lock and a Dekker-style spin lock. Taking on the flavor of Dekker's Algorithm [30], the new algorithm allows the owner thread of a lock to acquire and release the spin lock with a small constant number of read and write instructions, requiring no compound atomic operations.

Second, observing that the conventional spin lock is subsumed in a widely-used algorithm for Java locks, we attempt to replace the spin lock with the asymmetric spin lock. This results in a new reservation-based algorithm for Java lock which possesses the properties we desire. While the owner thread of a lock can acquire and release the lock without any atomic operations, a non-owner thread does not have to cancel the reservation, thus avoiding the need to suspend the owner thread.

We have evaluated our new Java lock algorithm, *asymmetric lock* [67, 89], in IBM's production Java virtual machine and JIT compiler. The results of micro-benchmarks show that the new algorithm achieves high performance, close to our previous algorithm on a reservation hit, while it removes the anomalous behavior the previous algorithm exhibits on a reservation miss. Furthermore, since our new algorithm does not cancel reservations at all, it allows programs to run with more reservation hits. For macro-benchmarks, while the new algorithm achieved comparable speedups in the SPECjvm98 benchmarks, it even improved the performance of two scientific programs for which the previous algorithm actually caused degradation.

The remaining sections in this chapter are organized as follows. Section 6.2 quickly revisits the reservation lock. Section 6.3 describes the concept and implementation of our asymmetric spin lock, and Section 6.4 gives several explanations of and extensions to the new spin lock. Section 6.5 constructs a new reservation-based Java lock algorithm by embedding the asymmetric spin lock into a widely-used algorithm for Java lock. Section 6.6 presents experimental results of the constructed asymmetric lock, and compares it with other lock methods. Finally, Section 6.7 gives the summary of this chapter.

6.2 Reservation Lock Revisited

In this section, we review our previous algorithm, reservation lock [65, 66], focusing on the points necessary for introducing the new algorithm. See Chapter 5 for the full details of reservation lock.

The reservation lock is constructed on top of an existing algorithm, by making one bit available in the lockword of each object's header. The bit is used to represent the lock reservation status, and hence called the *LRV bit*. When the LRV bit is set, the lockword is in the *reserve* mode, and the structure is defined by the algorithm. When the bit is not set, the lockword is in the *base* mode, and the structure is defined by the existing base algorithm.

Figure 6.1 summarizes the lockword state transitions in the reservation lock, where thick arrows indicate that atomic operations are used for the transitions. A lockword is

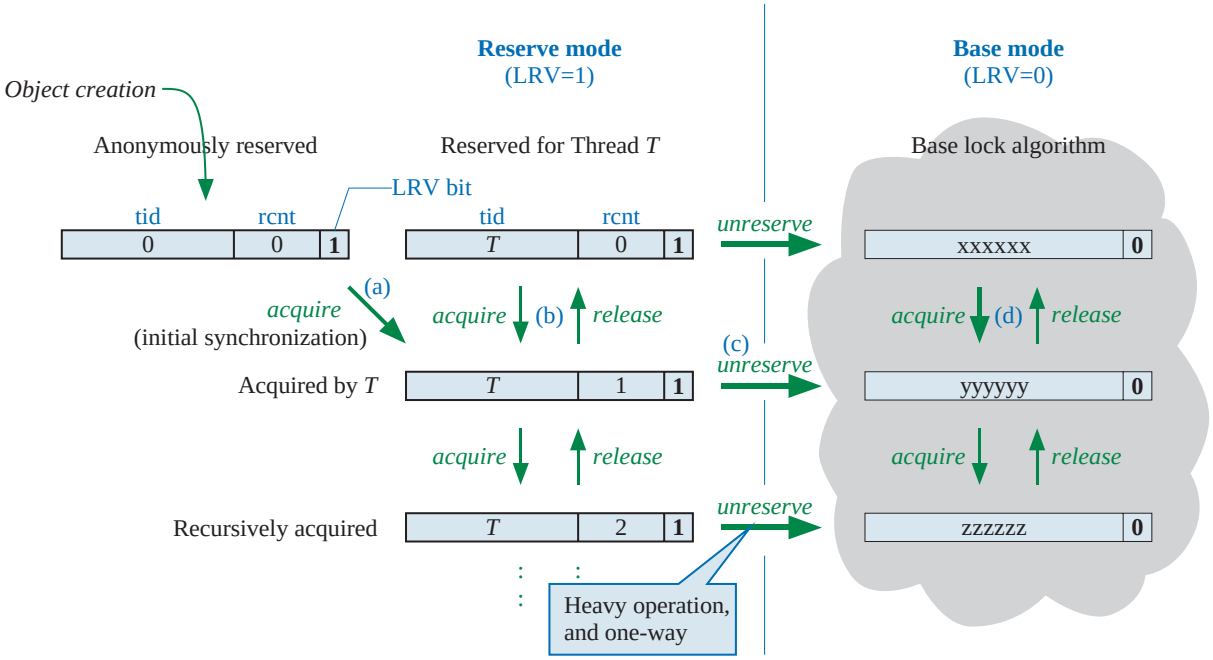

Figure 6.1: Lockword structure and state transitions of the reservation lock

initialized to be the anonymously-reserved state. The first thread attempting to acquire the lock makes the reservation specific by setting its identifier to the `tid` field with an atomic operation (Figure 6.1(a)). By this operation, the lock is reserved for the thread, and the owner thread can successively acquire and release the lock by simply incrementing and decrementing the `rcnt` field, respectively (Figure 6.1(b)). Since no atomic operations are necessary[1] in the common cases, the algorithm attained higher performance in the synchronization-intensive Java programs we measured.

When a non-owner thread attempts to acquire the reserved lock, the thread first *cancels* the reservation (Figure 6.1(c)), and then falls back to the base algorithm. Canceling a reservation is the most crucial and trickiest part of the algorithm. In doing so, the non-owner thread first suspends the owner thread, then replaces the lockword in the reserve mode with the equivalent state in the base mode. To avoid data race, the non-owner thread forces the suspended owner thread to be outside of *unsafe regions* by controlling the saved context, and allows the owner thread to resume execution. Once the reservation is canceled, subsequent lock operations are performed by the base algorithm using atomic operations (Figure 6.1(d)).

[1]Strictly speaking, the method utilizes the feature that memory read and write operations to a word are atomic, in other words, it cannot be observed that the word is partially modified. However, in this thesis, we use the term *atomic operations* for complex compound instructions that perform memory-reading, testing or modifying, and writing without being interrupted, such as `compare_and_swap`, `test_and_set`, and `fetch_and_add`.

While few reservations were canceled in the benchmarks we measured (Table 5.6), canceling a reservation requires expensive system calls and incurs a very large overhead. This weakens the robustness of the algorithm and makes it subject to pathological behavior. Another issue is that the cancellation is a one-way transition. Once a lock's reservation has been canceled, even the original owner thread can no longer acquire the lock quickly so the thread locality may not be fully exploited.

One reason for these weaknesses is that the algorithm exploits the cancellation transition for delegating complicated operations, such as contention management, to the full-function base algorithm. This chapter also reconsiders such approaches for constructing Java locks, and tries to remove the barrier of reservation cancellation by directly introducing the asymmetricity into a full-function Java lock method.

More concretely, while the previous approach defines an extension layer to an existing algorithm, thus making it applicable for most of the existing algorithms, our new approach takes a particular class of the existing algorithms, and replaces the spin locks used in those algorithms with new, reservation-based spin locks. In this *asymmetric spin lock*, the reservation persists even when a non-owner thread acquires the lock. Therefore, the troublesome reservation cancellation never occurs in the new Java lock which includes this spin lock.

6.3 The Proposal of Asymmetric Spin Lock

This section describes the idea of the asymmetric spin lock, which is a core component of the new reservation-based Java lock. We first review the generic spin lock, then present an asymmetric algorithm based on the reservation.

6.3.1 Generic Spin Lock

The spin lock is a primitive used to implement complex synchronization operations. Modern processors provide atomic read-modify-write instructions such as `compare_and_swap` and `test_and_set`, in order to facilitate creating locks in software. Thus, spin locks are commonly implemented with such atomic instructions.

Figure 6.2 presents one of the simplest of such spin locks[2], showing again the semantics of the `compare_and_swap` (CAS). When some thread holds the lock, the lockword contains the thread's identifier. Otherwise, the value is zero. To acquire a lock, a thread calls `spin_acquire`, which tries to change the value from zero to the thread's identifier by calling `spin_try_acquire`. This process is performed atomically by `compare_and_swap`, and iterated until it succeeds (i.e. spin). To release the acquired spin lock, the thread

[2]This is basically same as the code shown in Figure 4.1, but the `spin_try_acquire` function is separated and the `spin_is_acquired` function is added for future use in this chapter. Error checking is also added to the `spin_release` function.

```
 1 : int compare_and_swap(volatile unsigned int *addr,
 2 :                      unsigned int oldval, unsigned int newval) {
 3 :     // The following is performed atomically
 4 :     if (*addr == oldval) { *addr = newval; return SUCCESS; }
 5 :     else return FAILURE;
 6 : }
 7 :
 8 : int spin_acquire(volatile unsigned int *lock) {
 9 :     while (spin_try_acquire(lock) != SUCCESS)
10 :         continue; // spin until success
11 :     return SUCCESS;
12 : }
13 :
14 : // Try to acquire the spin lock
15 : int spin_try_acquire(volatile unsigned int *lock) {
16 :     return compare_and_swap(lock, 0, thread_id());
17 : }
18 :
19 : int spin_release(volatile unsigned int *lock) {
20 :     if (*lock != thread_id()) return ILLEGAL_STATE;
21 :     *lock = 0;
22 :     return SUCCESS;
23 : }
24 :
25 : // Check if current thread holds the spin lock
26 : int spin_is_acquired(volatile unsigned int *lock) {
27 :     if (*lock == thread_id()) return SUCCESS;
28 :     else return FAILURE;
29 : }
```

Figure 6.2: Semantics of `compare_and_swap`, and a simple CAS-based spin lock

calls `spin_release`, which resets the lockword to zero to enable another spinning thread to acquire the lock. The `spin_is_acquired` function returns `SUCCESS` if the lock is held by the current thread.

The algorithm in Figure 6.2 usually has to be augmented for practical use. For instance, many algorithms in use in the real world support recursive locking. Beyond that, they typically incorporate optimizations such as spin-on-read or exponential back-off [5] for scalability. We omit all of these extensions and optimizations for the sake of simplifying the explanation. Also notice that the field does not necessarily have to be one word long. It can be a byte or a short word, as long as the processor architecture allows `compare_and_swap` to act upon it.

6.3.2 Asymmetric Spin Lock

Next, we construct a new algorithm for spin lock, called *asymmetric spin lock*, by applying lock reservation to spin lock for the first time. The new algorithm allows a specific thread

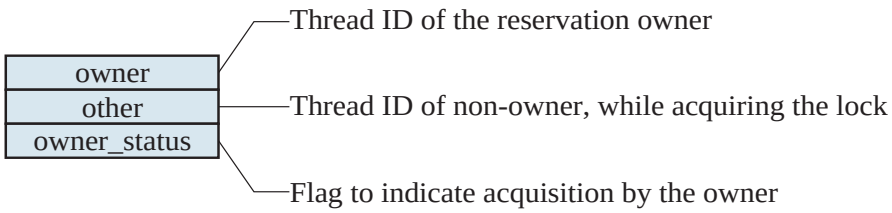

Figure 6.3: Data structure of the asymmetric spin lock

(the reservation owner) to process the lock without any atomic operations.

As shown in Figure 6.3, the data structure for our asymmetric spin lock consists of three fields, `owner` and `other` to hold thread identifiers, and `owner_status` to indicate the acquisition status of the owner thread. The lock is either in the *not-yet-reserved* state or the *reserved* state. When the lock is in the not-yet-reserved state, the `owner` field is zero. When it is in the reserved state, the field holds the identifier of the reservation-owner thread of the lock. When the lock is held by the owner, the `owner_status` field is set to one, and the `other` field is zero or will sooner or later be set to zero. When the spin lock is held by a non-owner thread, the `other` field contains the identifier of the thread, and the `owner_status` field is zero or will sooner or later be set to zero.

Figure 6.4 summarizes the state transitions of the asymmetric spin lock, where thick arrows indicate that atomic operations are used for these transitions. Like the reservation lock for Java in Section 6.2, the algorithm uses the first-acquirer reservation policy. Thus, the state of a lock is initialized to be the not-yet-reserved state, and is changed into the reserved state by the first thread which attempts to acquire the lock, by setting its identifier to the `owner` field (Figure 6.4(a)). In addition, we note that, once the lock has been reserved for a thread, the reservation is never canceled or switched to another thread, and it continues to be reserved for the same owner thread.

Once the lock is reserved, the owner thread attempts to acquire the spin lock by storing one to the `owner_status` field (Figure 6.4(b)), while any non-owner thread attempts to acquire the lock by setting its identifier to the `other` field (Figure 6.4(c)). An atomic operation is used for the latter attempt, since it may be performed by multiple non-owner threads.

These two types of acquisition attempts may occur at the same time. To solve the data race, each thread reads the other field (i.e. `other` or `owner_status`) after it wrote to its relevant field. If zero is read back, the spin lock was successfully acquired by the thread. If not, the thread resets the written field to zero, and spins for acquisition (Figure 6.4(d), (e)).

The lock can be released by simply writing zero to the field the thread modified to acquire the lock. Notice that the `owner_status` field is modified only by the owner thread, while the `other` field is modified by non-owner threads.

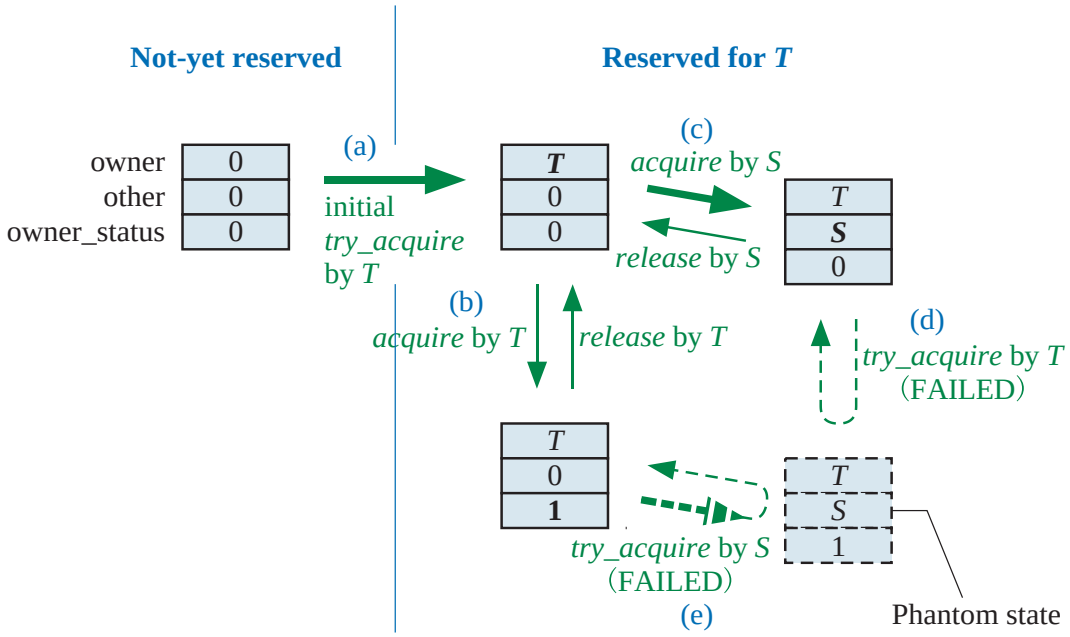

Figure 6.4: State transitions of the asymmetric spin lock

Figure 6.5 shows this algorithm of asymmetric spin lock in pseudo-C code. The `spin_try_acquire` function (lines 13–41) does the essential task, where lines 17–21 reserve a lock for the first acquirer (Figure 6.4(a)), and lines 22–30 acquire the lock for the owner thread (Figure 6.4(b), (d)), while lines 31–40 acquire the lock for non-owner threads (Figure 6.4(c), (e)).

Let us first consider the case when a thread attempts to acquire the lock in the not-yet-reserved state (lines 17–21). It performs the `compare_and_swap` in order to change the lock state from the initial state to the reserved state. If it does not succeed, its attempt has failed (line 19), and some other thread must have succeeded in the state transition. Otherwise, the thread proceeds to the case for the owner thread.

We next consider the case when the owner thread attempts to acquire the lock (lines 22–30). It first sets the `owner_status` field, and then checks the `other` field. If the `other` field is zero, it implies that no race has occurred, and the owner has successfully acquired the lock (line 25). Otherwise, the owner's attempt has failed because another non-owner thread holds or is just acquiring the lock, and the owner resets the `owner_status` field (line 27).

Finally, we consider the case when a non-owner thread attempts to acquire the lock (lines 31–40). It first tries to atomically change the `other` field from zero to its thread identifier with a `compare_and_swap`. If it failed, the non-owner's attempt to acquire the lock failed (line 33) because another non-owner thread holds or is just acquiring the lock. Otherwise, the non-owner proceeds to check the `owner_status` field. If the field is zero, it

```
 1 : typedef struct asym_spin {
 2 :     volatile unsigned int owner; // thread ID of the reservation owner
 3 :     volatile unsigned int other; // thread ID of non-owner, acquiring the lock
 4 :     volatile int owner_status;   // flag to indicate acquisition by the owner
 5 : } asym_spin_t;
 6 :
 7 : int spin_acquire(volatile asym_spin_t *lock) {
 8 :     while (spin_try_acquire(lock) != SUCCESS)
 9 :         continue; // spin until success
10 :     return SUCCESS;
11 : }
12 :
13 : int spin_try_acquire(volatile asym_spin_t *lock) {
14 :     unsigned int owner = lock->owner;
15 :     unsigned int myTID = thread_id();
16 :
17 :     if (owner == 0) { // attempt in the initial state
18 :         if (compare_and_swap(&lock->owner, 0, myTID) != SUCCESS)
19 :             return FAILURE;
20 :         owner = myTID; // falls through if reservation succeeded
21 :     }
22 :     if (owner == myTID) { // attempt by the owner thread
23 :         lock->owner_status = 1;
24 :         if (lock->other == 0)
25 :             return SUCCESS;
26 :         else {
27 :             lock->owner_status = 0;
28 :             return FAILURE;
29 :         }
30 :     }
31 :     else { // attempt by a non-owner thread
32 :         if (compare_and_swap(&lock->other, 0, myTID) != SUCCESS)
33 :             return FAILURE;
34 :         if (lock->owner_status == 0)
35 :             return SUCCESS;
36 :         else {
37 :             lock->other = 0;
38 :             return FAILURE;
39 :         }
40 :     }
41 : }
42 :
```

Figure 6.5: Algorithm of the asymmetric spin lock (1 of 2)

```
43 : int spin_release(volatile asym_spin_t *lock) {
44 :     unsigned int myTID = thread_id();
45 :
46 :     if (lock->owner == myTID) {
47 :         if (lock->owner_status == 0) return ILLEGAL_STATE;
48 :         lock->owner_status = 0;
49 :     } else {
50 :         if (lock->other != myTID) return ILLEGAL_STATE;
51 :         lock->other = 0;
52 :     }
53 :     return SUCCESS;
54 : }
55 :
56 : int spin_is_acquired(volatile asym_spin_t *lock) {
57 :     unsigned int myTID = thread_id();
58 :
59 :     if (lock->owner == myTID) {
60 :         if (lock->owner_status == 1) return SUCCESS;
61 :     } else {
62 :         if (lock->other == myTID) return SUCCESS;
63 :     }
64 :     return FAILURE;
65 : }
```

Figure 6.5: Algorithm of the asymmetric spin lock (2 of 2)

implies that no race has occurred, and the non-owner has successfully acquired the lock (line 35). Otherwise, the non-owner's attempt has failed because the owner thread holds or is just acquiring the lock, and the non-owner resets the `other` field (line 37).

The `spin_acquire` function (lines 7–11) is implemented by continuously calling (i.e. spinning) the `spin_try_acquire` function until it succeeds, which is same as in the generic spin lock shown in Figure 6.2. The `spin_release` function (lines 43–54) first reads the `owner` field. If the current thread is the owner of the lock, it stores zero to the `owner_status` field, otherwise it stores zero to the `other` field. The `ILLEGAL_STATE` error is returned if the lock is not held by the current thread.

It is quickly known from the code in Figure 6.5 that the `owner_status` field is only modified by the owner thread, while the `other` field is only modified by a non-owner thread which successfully acquired the lock by `compare_and_swap`. Thus, the writes to these two fields must not interfere with each other.

The performance of the asymmetric spin lock in the absence of contention can be summarized as follows. The owner thread can acquire and release the lock with three reads (lines 14, 24, and 46) and two writes (lines 23 and 48), requiring no atomic read-modify-write instructions, while the non-owner thread requires one atomic read-modify-write instruction (line 32), in addition to three reads (lines 14, 34, and 46) and one write (line 51). Since the atomic operation is unnecessary for the reservation owner, performance can be improved if most of the lock operations are performed by that thread.

6.4 Discussion and Extension

In this section, we discuss the asymmetric spin lock introduced in the previous section from various viewpoints, and show several extensions.

6.4.1 Dekker's Algorithm

It was of academic interest to design algorithms for spin locks using only read and write instructions [29, 30, 70, 93]. The most famous one among these is *Dekker's Algorithm* introduced by Dijkstra [30], which is a mutual exclusion mechanism for two threads (or processes) to execute a critical section. In this algorithm, a flag for mutual exclusion is prepared for each of the two threads, and initially cleared. To execute the critical section, each thread first sets its flag and then checks the other's flag. If it is set, the thread resets its flag and retries the process. Otherwise, the thread executes the critical section and clears its flag in exiting from the critical section.

Figure 6.6 shows a spin lock based on Dekker's Algorithm. The two-element `status` array in the `dekker_spin_t` structure is used for the flags. Notice that this is a *two-thread algorithm*. The algorithm assumes that only two threads, whose identifiers are zero and one, are involved. We also assume that the `myTID` and `anotherTID` variables are appropriately set for each thread in advance.

Each of the two threads attempts to acquire the lock by setting its own status element (line 12), and then checking the other thread's status element (line 13). If the other thread's element is not set, the thread has successfully acquired the lock. Otherwise, the thread has failed in acquisition, and resets its own status element (line 16).

We can generalize the algorithm for n threads by expanding the `status` array to n and checking the other $(n-1)$ threads' status before executing the critical section. However, the number of memory operations required becomes proportional to n. Thus, as the number of threads increases, the performance becomes worse than the CAS-based spin lock shown in Figure 6.2. As far as we know, all the algorithms composed from read and write instructions suffer from this problem, including those in [29, 30, 70, 93].

It is important to observe that our asymmetric spin lock in Figure 6.5 hybridizes the CAS-based lock in Figure 6.2 and the Dekker-style lock in Figure 6.6. The non-owner threads first compete with each other in the preliminary round and the winner advances to the final round, while the owner thread is seeded to the final round. The preliminary round is performed through the CAS-based lock (line 32 of Figure 6.5). The final round, involving exactly two threads, is done using the Dekker-style lock (lines 23–29 for the owner, and lines 32 and 34–39 for the winner of the preliminary round).

The asymmetric spin lock solved the above-mentioned scaling issue by introducing asymmetricity and atomic operations into Dekker's Algorithm. However, only the owner thread can process the lock without atomic operations. We believe that this algorithm is still very useful, at least for Java locks, where many locks are dominantly acquired by specific threads.

```
 1 : typedef struct dekker_spin {
 2 :     volatile int status[2]; // flags for the two threads, initialized as {0,0}
 3 : } dekker_spin_t;
 4 :
 5 : int spin_acquire(volatile dekker_spin_t *lock) {
 6 :     while (spin_try_acquire(lock) != SUCCESS)
 7 :         continue; // spin until success
 8 :     return SUCCESS;
 9 : }
10 :
11 : int spin_try_acquire(volatile dekker_spin_t *lock) {
12 :     lock->status[myTID] = 1;           // set my flag, and
13 :     if (lock->status[anotherTID] == 0) // check another's flag
14 :         return SUCCESS;
15 :     else {
16 :         lock->status[myTID] = 0;
17 :         return FAILURE;
18 :     }
19 : }
20 :
21 : int spin_release(volatile dekker_spin_t *lock) {
22 :     if (lock->status[myTID] == 0) return ILLEGAL_STATE;
23 :     lock->status[myTID] = 0; // clear my flag
24 :     return SUCCESS;
25 : }
```

Figure 6.6: Dekker-style spin lock for two threads

6.4.2 Correctness

Next, we discuss the correctness of the new algorithm.

In our asymmetric spin lock, each lock's reservation owner is decided by setting the `owner` field when the lock is first acquired. Because this step is performed by `compare_and_swap` (line 18 of Figure 6.5), the `owner` field cannot be overwritten once it is set. The `owner_status` field is modified only by the owner thread after it is decided, so no race hazard can occur for this field.

On the other hand, the `other` field can be modified by multiple non-owner threads. However, setting the thread's identifier to acquire the lock is also performed by `compare_and_swap` (line 32), so it never overwrites any other thread's identifier. Resetting the `other` field to release the lock (line 51) or to avoid a contention (line 37) is performed only by the thread which has successfully stored its identifier to the `other` field, so racing never occurs here.

It is guaranteed by Dekker's Algorithm that the asymmetric spin lock provides mutual exclusion between the owner and non-owner, and does not deadlock. However, the implementation shown in Figure 6.5 *may* livelock in exactly the same manner as the Dekker-style lock. That is, when the owner and a non-owner thread simultaneously attempt to acquire

the lock, they may continuously fail. More concretely, the owner may continuously execute lines 23, 24, 27, and 28 (Figure 6.4(d)) in the `spin_try_acquire` function, while the non-owner may continuously execute lines 32, 34, 37, and 38 (Figure 6.4(e)).

We could resolve the livelock with a technique used in Dekker's Algorithm [30], by introducing a field to indicate which of the owner and a non-owner takes precedence on contention. However the technique was not included in our asymmetric spin lock because it increases memory accesses. We could also ameliorate the issue with back-offs on contention as proposed by Anderson [5]. Also, we note that the one-word asymmetric spin lock we will soon present in Section 6.4.5 does not cause livelock. In addition, when a spin lock is embedded into a bimodal Java lock (as will be described in Section 6.5), once the `spin_try_acquire` function has failed, then the spin lock is usually taken over by a suspend-lock instead of spinning to acquire the lock. Therefore, livelock does not occur in Java locks even without modifying the implementation.

6.4.3 Multiprocessor Considerations

While our algorithm does not require any atomic operations on a reservation hit, it does rely on the program order execution of writes and reads. More concretely, in Figure 6.5, the owner thread must execute the write to the `owner_status` field (line 23) and the read from the `other` field (line 24) in that order[3].

Multiprocessor architectures with *relaxed memory consistency models* [1, 27] do not necessarily guarantee the program order of memory operations, but provide hardware instructions to force the program order. Modern architectures include dedicated instructions called *memory barriers* or *fences*, while older architectures rely on atomic read-modify-write instructions to achieve the same effect.

Some architectures allow the program order to be preserved with *software techniques*, although the details significantly vary depending on processor architectures and even processor implementations. For instance, for the IBM 370, we can preserve the order of `write into X` and `read from Y` (`X` and `Y` are different memory locations) with the sequence of `write into X`, `read from X`, and `read from Y` [1]. In general, software techniques are cheaper than hardware instructions.

Whether the program order is preserved with hardware instructions or software techniques, it must be noted that the asymmetric spin lock provides performance gains on those multiprocessor systems where the cost of the atomic read-modify-write instruction (which it removes) is higher than the cost of program-order enforcement (which it introduces).

Finally, we note that while enforcing the program order obviously affects some optimizations such as speculation, using the atomic read-modify-write instruction also does so in the same manner.

[3] Actually, the `compare_and_swap` against the `other` field (line 32) and the read from the `owner_status` field (line 34) must also be executed in that order by a non-owner thread, but usually this is assured because the atomic operation implies a memory-ordering barrier.

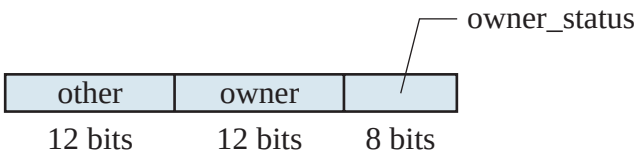

Figure 6.7: Data layout in the one-word variation

6.4.4 Optimizations

If we have a `compare_and_swap` that can act upon both the `other` and `owner_status` fields simultaneously, the execution path by a non-owner thread could be made much simpler, since this instruction allows the operation on `other` at line 32 and the check of `owner_status` at line 34 to be combined.

Similarly, if we have a `compare_and_swap` that can act upon both the `owner` and `owner_status` fields simultaneously, the execution path in the not-yet-reserved state could be made simpler, since a thread could directly change the lock state from the not-yet-reserved state to the state actually reserved for and acquired by the thread itself at line 18.

These optimizations do not necessarily require the underlying processor architecture to support a special instruction such as `double_compare_and_swap` [42] (`DCAS`, or `CAS2` [82]). We could achieve the same effect by co-locating multiple fields in a single memory region upon which the usual `compare_and_swap` could act. The technique is used in the one-word variation of the asymmetric spin lock we will describe below.

6.4.5 One-Word Embodiment

The asymmetric spin lock explained to this point allocated each of the three fields in a separate word[4], mainly to simplify the explanation. Although the three-word version is simple and clearly conveys the key ideas and the essential properties of the asymmetric spin lock, it is not as space efficient as to construct a Java lock on the data structure. In this section, we show a variation where the three fields are packed into one word, in order to be easily embedded into a Java lock.

The one-word embodiment assumes that `compare_and_swap` and atomic read are possible for 32-bit data, and that atomic writes are also possible for 8-bit and 16-bit data without affecting the other data. The algorithm uses a 32-bit word with the data layout shown in Figure 6.7, where the minimal size of atomic write, 8 bits, is assigned to `owner_status` and the remaining 24 bits are divided into the `owner` and `other` fields

Figure 6.8 shows the algorithm of the one-word variation in pseudo-C code, where the sidelines indicate the portions optimized from the original asymmetric spin lock in Figure 6.5. Note that we assume Little Endian addressing here. As we mentioned earlier,

[4]Unless otherwise stated, we assume that a word is 32 bits long.

```
    1 : #define OWNER_MASK 0x000fff00
    2 : #define OTHER_MASK 0xfff00000
    3 :
    4 : // The following definitions assume Little Endian addressing
    5 : #define OWNER_STATUS_BYTE(lock) (((char  *)lock)[0])
    6 : #define OTHER_SHORT(lock)       (((short *)lock)[1])
    7 :
    8 : // We assume that thread_id() returns values shifted to fit into
    9 : // the owner field, between 0x00000100 and 0x000fff00
   10 : #define OWNER_TO_OTHER(tid)     ((tid)<<12)
   11 :
   12 : int spin_acquire(volatile unsigned int *lock) {
   13 :     while (spin_try_acquire(lock) != SUCCESS)
   14 :         continue; // spin until success
   15 :     return SUCCESS;
   16 : }
   17 :
   18 : int spin_try_acquire(volatile unsigned int *lock) {
   19 :     unsigned int l = *lock;
   20 :     unsigned int myTID = thread_id();
   21 :
   22 :     if ((l & OWNER_MASK) == 0) { // attempt in the initial state
|  23 :         unsigned int locked = myTID | 1;
|  24 :         return compare_and_swap(lock, 0, locked);
   25 :     }
   26 :     else if ((l & OWNER_MASK) == myTID) { // attempt by the owner thread
   27 :         OWNER_STATUS_BYTE(lock) = 1;
   28 :         if ((*lock & OTHER_MASK) == 0)
   29 :             return SUCCESS;
   30 :         else {
   31 :             OWNER_STATUS_BYTE(lock) = 0;
   32 :             return FAILURE;
   33 :         }
   34 :     }
   35 :     else { // attempt by a non-owner thread
|  36 :         unsigned int unlocked = l & OWNER_MASK;
|  37 :         unsigned int locked = unlocked | OWNER_TO_OTHER(myTID);
|  38 :         return compare_and_swap(lock, unlocked, locked);
   39 :     }
   40 : }
   41 :
```

Figure 6.8: Algorithm of the one-word variation (1 of 2)

```
42 : int spin_release(volatile unsigned int *lock) {
43 :     unsigned int l = *lock;
44 :     unsigned int myTID = thread_id();
45 :
46 :     if ((l & OWNER_MASK) == myTID) {
47 :         if (OWNER_STATUS_BYTE(&l) == 0) return ILLEGAL_STATE;
48 :         OWNER_STATUS_BYTE(lock) = 0;
49 :     } else {
50 :         if ((l & OTHER_MASK) != OWNER_TO_OTHER(myTID)) return ILLEGAL_STATE;
51 :         unsigned int unlocked = l & ~OTHER_MASK;
52 :         OTHER_SHORT(lock) = OTHER_SHORT(&unlocked);
53 :     }
54 :     return SUCCESS;
55 : }
56 :
57 : int spin_is_acquired(volatile unsigned int *lock) {
58 :     unsigned int l = *lock;
59 :     unsigned int myTID = thread_id();
60 :
61 :     if ((l & OWNER_MASK) == myTID) {
62 :         if (OWNER_STATUS_BYTE(&l) == 1) return SUCCESS;
63 :     } else {
64 :         if ((l & OTHER_MASK) == OWNER_TO_OTHER(myTID)) return SUCCESS;
65 :     }
66 :     return FAILURE;
67 : }
```

Figure 6.8: Algorithm of the one-word variation (2 of 2)

the writes to the `owner_status` and `other` fields must not interfere with each other. Thus, the one-word variation sets and resets the `owner_status` field with the 8-bit write (lines 27, 31, and 48), while it clears the `other` field using the 16-bit write in the `spin_release` function (line 52). Note that involving the upper four bits of the `owner` field in the 16-bit write for the `other` field does not cause any problem. This is because the value of the `owner` field is never changed once the lock has been set to the reserved state.

The `compare_and_swap` operations on the 32-bit data are used for making the lock to the reserved state (line 24) and for the acquisition by non-owner threads (line 38). Since this makes it possible to check multiple fields atomically, the two optimizations shown in the last subsection are also applied to the algorithm.

As a side effect of the above optimizations, the one-word variation cannot cause the livelock mentioned in Section 6.4.2. That is, the owner and a non-owner thread never continuously fail. If the owner thread takes the path to return `FAILURE` at line 32, it implies that the non-owner thread has succeeded in the `compare_and_swap` at line 38, meaning that the non-owner thread has succeeded in the acquisition of the lock.

Finally, Figure 6.9 illustrates the state transitions of the lockword in the one-word variation.

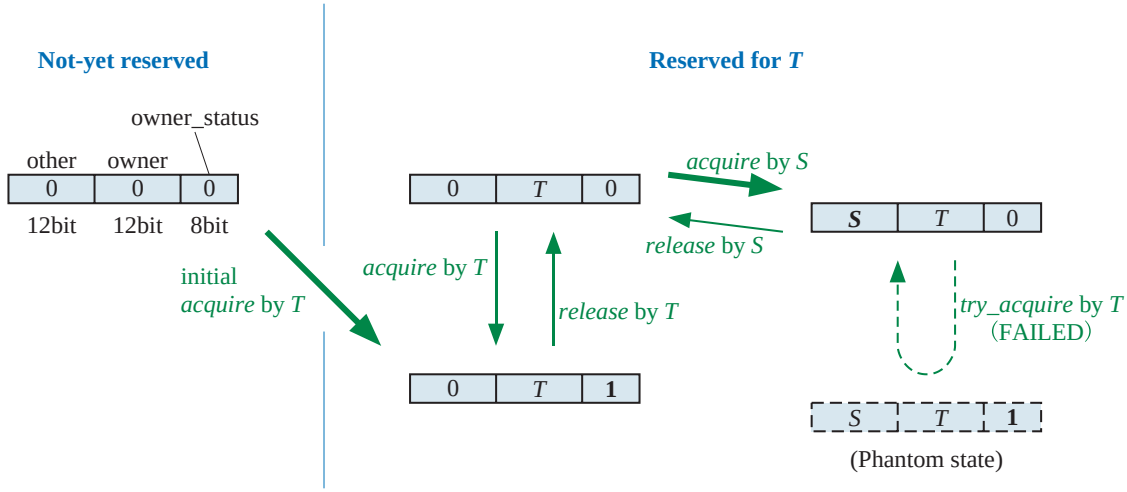

Figure 6.9: Lockword state transitions of the one-word variation

6.5 Optimizing Java Lock with the Asymmetric Spin Lock

By using the one-word asymmetric spin lock derived above, this section describes a technique to implement a new reservation-based Java lock which is free from the overhead of reservation cancellation.

6.5.1 Java Lock and Spin Lock

As shown in Chapter 5, it is known that many Java locks exhibit *thread locality*, which means that each lock is dominantly used by a specific thread [65, 66]. Therefore, we may be able to accelerate Java locks by utilizing the asymmetric spin lock, while giving the precedence (reservation) to the specific thread which will dominantly use the lock. In addition, since the reservation is not canceled in the asymmetric spin lock, we can remove the step of reservation cancellation, which is a potential problem for the reservation lock proposed in the previous chapter.

However, it is not realistic to directly use a spin lock as a Java lock. Synchronization in Java is based on monitors [20, 21, 46], and the critical sections guarded by Java locks may be long, such as the example shown in Table 4.4 of Chapter 4. Since a spin lock requires the thread to busy-wait when the lock is not acquired, it is not suitable for long critical sections. Clearly, *suspend locks*, which require the thread to relinquish control of the processor, are much preferred for Java locks.

A widely employed optimization of a suspend lock is hybridizing it with a spin lock [90] as a *spin-suspend lock*. When a thread attempting to acquire a spin-suspend lock finds that it is not available, it busy-waits or spins but does so only a certain number of

times. After the thread has repeatedly failed in all of the spins, it relinquishes control of the processor.

Space-efficiency and time-efficiency are equally important in Java locks. In general, we can make lock operations on objects faster by implementing them with bits in the headers. However, the real estate in an object's header is very precious, and thus the number of bits dedicated to lock processing must be minimized.

A *bimodal lock* [11, 38, 88] is a space-efficient implementation of the spin-suspend lock intended for use in Java. This method reserves only one field (a lockword) in each object's header, using the field in two modes. In the *flat* mode, the bimodal lock acts as a spin lock. When the lock is held by a thread, the lockword contains the thread's identifier. Otherwise, the value of the lockword is zero. In the *inflated* mode, the bimodal lock behaves as a suspend lock, and the lockword contains a reference to the suspend lock. These two modes are distinguished by one bit in the lockword, called the *shape bit*.

The bimodal lock is initialized to be in the flat mode, and remains in that mode as long as no contention occurs. Thus, the bimodal lock is as efficient in the uncontended case as the spin lock. When the lock is contended, a structure for suspend lock is prepared, and the bimodal lock is *inflated* and put into the inflated mode.

One instance of the bimodal locks for Java is the tasuki lock [88] explained in Chapter 4. As in its base algorithm, thin lock [11], tasuki lock uses the `compare_and_swap` as the spin lock primitive in the flat mode. A new reservation-based Java lock can be implemented by replacing this part with the asymmetric spin lock.

6.5.2 Tasuki Lock Revisited

Before showing the new algorithm, we briefly review the tasuki lock here, focusing on how the spin lock is used in it.

Figure 6.10 shows the algorithm of the tasuki lock, which is basically same as that shown in Figure 4.7 except for the portions indicated by sidelines, which are modified or added to make the internal spin lock replaceable.

The `Java_lock_acquire` function is called to acquire a Java lock. This function first attempts to acquire the spin lock by using `spin_try_acquire` (line 14). Successfully acquiring the spin lock means successfully acquiring the Java lock. If the spin lock cannot be acquired, the function then tries to acquire the suspend lock by using `monitor_enter` (line 18). In this case, if the lockword is in the flat mode, it is converted to the inflated mode by calling the `inflate` function.

The `Java_lock_release` function is called to release a Java lock. This function first tests the lockword to determine whether it is in the flat mode. If so, it releases the spin lock by using `spin_release` (line 48), with additional processing if the lock is contended. Otherwise, it releases the suspend lock by using `monitor_exit` (line 62). Just before exiting from the monitor, the function checks if the lockword can return to the flat mode, and calls the `deflate` function if it is possible.

```
    1 : // Object header contains a lockword and an FLC bit
    2 : typedef struct object {
    3 :     //    :
    4 :     volatile unsigned int flag; // 1 bit is used as an FLC bit
    5 :     volatile unsigned int lockword;
    6 :     //    :
    7 : } Object;
    8 :
    9 : #define SHAPE_BIT 0x80000000
|  10 : #define MON_MASK (~SHAPE_BIT)
   11 :
   12 : int Java_lock_acquire(Object *obj) {
   13 :     // flat lock path
|  14 :     if (spin_try_acquire(&obj->lockword) == SUCCESS)
   15 :         return SUCCESS;
   16 :     // inflated lock and inflation path
   17 :     monitor_t *mon = obtain_monitor(obj);
   18 :     monitor_enter(mon);
   19 :     // inflation loop
   20 :     while ((obj->lockword & SHAPE_BIT) == 0) {
   21 :         obj->flag |= OBJFLAG_FLAT_CONTENDED; // set the FLC bit
|  22 :         if (spin_try_acquire(&obj->lockword) == SUCCESS)
   23 :             inflate(obj, mon);
   24 :         else
   25 :             monitor_wait(mon); // wait on the inflation monitor
   26 :     }
   27 :     return SUCCESS;
   28 : }
   29 :
   30 : // Returns a monitor associated with the object
   31 : monitor_t *obtain_monitor(Object *obj) {
   32 :     unsigned int lw = obj->lockword;
|  33 :     if ((lw & SHAPE_BIT) != 0) return (monitor_t *)(lw & MON_MASK);
   34 :     return lookup_monitor(obj); // search for a monitor in the table,
   35 : }                               // or create/register a new monitor
   36 :
   37 : // Inflate the object's lockword, which is held by current thread
   38 : void inflate(Object *obj, monitor_t *mon) { // mon must also be entered
   39 :     obj->flag &= ~OBJFLAG_FLAT_CONTENDED; // reset the FLC bit
   40 :     monitor_notify_all(mon);
   41 :     obj->lockword = SHAPE_BIT | (unsigned int)mon; // set the shape bit
   42 : }
   43 :
```

Figure 6.10: Review of the tasuki lock algorithm (1 of 2)

```
   44 : int Java_lock_release(Object *obj) {
   45 :     unsigned int lw = obj->lockword;
   46 :     if ((lw & SHAPE_BIT) == 0) { // flat lock path
|  47 :         if (spin_is_acquired(&obj->lockword) != SUCCESS) return ILLEGAL_STATE;
|  48 :         spin_release(&obj->lockword);
   49 :         if (obj->flag & OBJFLAG_FLAT_CONTENDED) { // test the FLC bit,
   50 :             monitor_t *mon = obtain_monitor(obj); //      the only overhead
   51 :             monitor_enter(mon);
   52 :             if (obj->flag & OBJFLAG_FLAT_CONTENDED) monitor_notify(mon);
   53 :             monitor_exit(mon);
   54 :         }
   55 :         return SUCCESS;
   56 :     }
   57 :     // inflated lock path, deflate if possible
|  58 :     monitor_t *mon = (monitor_t *)(lw & MON_MASK);
   59 :     if (!monitor_being_entered_by_me(mon)) return ILLEGAL_STATE;
   60 :     if (!monitor_being_waited(mon) && Better_to_deflate(obj))
|  61 :         deflate(obj, mon);
   62 :     return monitor_exit(mon);
   63 : }
   64 :
   65 : // Deflate the object's lockword and release the spin lock
|  66 : void deflate(Object *obj, monitor_t *mon) {
|  67 :     obj->lockword = 0; // reset the shape bit, while releasing the spin lock
|  68 : }
   69 :
   70 : int Java_lock_wait(Object *obj) {
   71 :     unsigned int lw = obj->lockword;
   72 :     if ((lw & SHAPE_BIT) == 0) { // flat mode
|  73 :         if (spin_is_acquired(&obj->lockword) != SUCCESS) return ILLEGAL_STATE;
   74 :         // force the inflation
   75 :         monitor_t *mon = obtain_monitor(obj);
   76 :         monitor_enter(mon);
   77 :         inflate(obj, mon);
   78 :     }
   79 :     // execute the wait using the monitor structure
|  80 :     monitor_t *mon = (monitor_t *)(obj->lockword & MON_MASK);
   81 :     return monitor_wait(mon); // wait on the Java object
   82 : }
   83 :
   84 : // Java_lock_notify()/notify_all() are implemented in the same manner
```

Figure 6.10: Review of the tasuki lock algorithm (2 of 2)

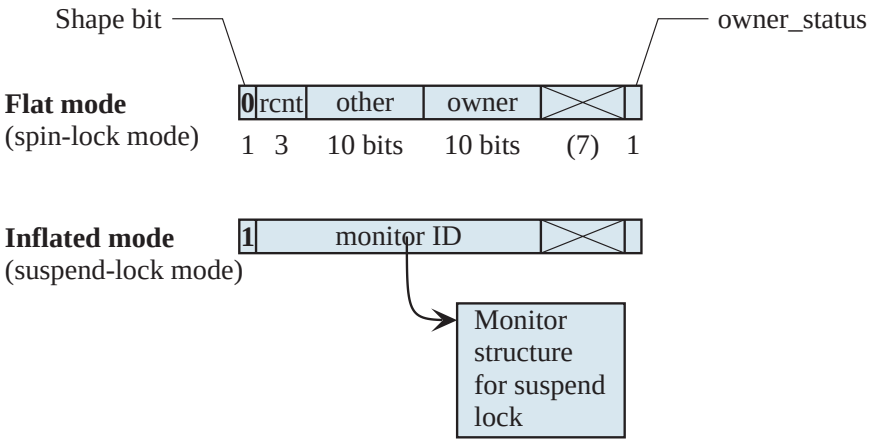

Figure 6.11: Lockword structure of the asymmetric tasuki lock

For the mode transitions, tasuki lock uses a flag in another word of the object's header named the FLC bit (`OBJFLAG_FLAT_CONTENDED` in the code), and a notification mechanism utilizing the suspend lock. Full details of this algorithm were shown in Chapter 4.

6.5.3 Asymmetric Tasuki Lock

The original tasuki lock shown in Chapter 4 uses the CAS-based spin lock shown in Figure 6.2 for the `spin_try_acquire`, `spin_release`, and `spin_is_acquired` functions in the code. We must note that the shape bit is implicitly tested in acquiring the spin lock, which fails if the bit is set. In addition to the `inflate` function, the `deflate` function is newly defined to separate the lockword-conversion function.

A new Java lock can be constructed by modifying the lockword structure and replacing these five functions with the version for the asymmetric spin lock. The resulting *asymmetric tasuki lock* behaves like the asymmetric spin lock in the flat mode. Thus, this new reservation-based Java lock does not require any atomic operations on a reservation hit *and* does not have to suspend the owner on a reservation miss.

For the replacement, the one-word variation of asymmetric spin lock presented in Section 6.4.5 is used, while modifying the data layout as shown in Figure 6.11 to include the shape bit and recursion count field[5]. Using this data layout, the asymmetric spin lock is slightly modified in order to embed it into the tasuki lock as shown in Figure 6.12. The sidelines indicate the portions modified from the corresponding functions in Figures 6.8 and 6.10. To simplify the explanation, the `rcnt` handling is omitted from this code. In the real code used for the evaluation in next section, the 3-bit field is used appropriately[6]

[5]Since only 10 bits are assigned for the `owner` and `other` fields in this layout, the number of threads that can simultaneously exist in the system is limited to 1,023. It is possible to expand the thread fields to 15 bits by using the unused 7 bits for `owner` and moving the `rcnt` field to another word.

```
    1 : // Now the monitor structure has an extra field to hold the spin lock's owner
    2 : typedef struct monitor {
    3 :     //    :
    4 :     // original monitor fields
    5 :     //    :
|   6 :     unsigned int spin_owner; // additional field to store the owner's TID
    7 : } monitor_t;
    8 :
    9 : #define SHAPE_BIT  0x80000000
|  10 : #define MON_MASK   0x7fffff00
|  11 : #define OWNER_MASK 0x0003ff00
|  12 : #define OTHER_MASK 0x0ffc0000
   13 :
|  14 : #define SHAPE_OWNER_MASK (SHAPE_BIT | OWNER_MASK)
|  15 : #define SHAPE_OTHER_MASK (SHAPE_BIT | OTHER_MASK)
   16 :
   17 : #define OWNER_STATUS_BYTE(lock) (((char  *)lock)[0])
   18 : #define OTHER_SHORT(lock)       (((short *)lock)[1])
|  19 : #define OWNER_TO_OTHER(tid)     ((tid)<<10)
   20 :
   21 : int spin_try_acquire(volatile unsigned int *lock) {
   22 :     unsigned int l = *lock;
   23 :     unsigned int myTID = thread_id();
   24 :
|  25 :     if ((l & SHAPE_OWNER_MASK) == 0) { // attempt in the initial state
   26 :         unsigned int locked = myTID | 1;
   27 :         return compare_and_swap(lock, 0, locked);
   28 :     }
|  29 :     else if ((l & SHAPE_OWNER_MASK) == myTID) { // attempt by the owner thread
   30 :         OWNER_STATUS_BYTE(lock) = 1;
|  31 :         if ((*lock & SHAPE_OTHER_MASK) == 0)
   32 :             return SUCCESS;
   33 :         else {
   34 :             OWNER_STATUS_BYTE(lock) = 0;
   35 :             return FAILURE;
   36 :         }
   37 :     }
   38 :     else { // attempt by a non-owner thread
   39 :         unsigned int unlocked = l & OWNER_MASK;
   40 :         unsigned int locked = unlocked | OWNER_TO_OTHER(myTID);
   41 :         return compare_and_swap(lock, unlocked, locked);
|  42 :                         // will fail if the shape bit is set
   43 :     }
   44 : }
   45 :
   46 : // spin_release()/spin_is_acquired() are not changed
   47 :
```

Figure 6.12: Asymmetric spin lock modified for use in the tasuki lock (1 of 2)

```
   48 : // We assume that mon fits into the MON_MASK bits of a lockword
   49 : void inflate(Object *obj, monitor_t *mon) {
   50 :     obj->flag &= ~OBJFLAG_FLAT_CONTENDED; // reset the FLC bit
   51 :     monitor_notify_all(mon);
|  52 :     mon->spin_owner = obj->lockword & OWNER_MASK; // save the owner info
|  53 :     if (mon->spin_owner == thread_id()) {
|  54 :         obj->lockword = SHAPE_BIT | (unsigned int)mon;
|  55 :     } else while (1) { // inflate while preserving the owner_status bit
|  56 :         unsigned int l = obj->lockword;
|  57 :         unsigned int inflated = SHAPE_BIT | (unsigned int)mon | (l & 1);
|  58 :         if (compare_and_swap(&obj->lockword, l, inflated) == SUCCESS) return;
|  59 :     }
   60 : }
   61 :
   62 : void deflate(Object *obj, monitor_t *mon) {
|  63 :     if (mon->spin_owner == thread_id()) {
|  64 :         obj->lockword = mon->spin_owner; // restore the owner info
|  65 :     } else while (1) { // deflate while preserving the owner_status bit
|  66 :         unsigned int l = obj->lockword;
|  67 :         unsigned int deflated = mon->spin_owner | (l & 1);
|  68 :         if (compare_and_swap(&obj->lockword, l, deflated) == SUCCESS) return;
|  69 :     }
   70 : }
```

Figure 6.12: Asymmetric spin lock modified for use in the tasuki lock (2 of 2)

to hold the count of recursive spin-lock acquisitions up to seven.

As described above, the `spin_try_acquire` function must always fail in the inflated mode. Therefore, the function is modified to check the shape bit in the availability checks (lines 25, 29, 31, and 41). This modification is unnecessary for `spin_release`, since the function is called only in the flat mode.

When modifying the `inflate` and `deflate` functions, the following three points must be taken into account. First, the `owner_status` field should be preserved in the transition. Therefore, `compare_and_swap` is used if the transition is performed by a non-owner thread (lines 58 and 68). However, this does not cause performance degradation, since the transition functions are not called frequently. Second, the owner thread should not be changed in the transition. Therefore, the information about the owner is saved in `inflate` (line 52) and restored to the lockword in `deflate` (lines 64 and 67). Third, the `deflate` function not only clears the shape bit but also releases the spin lock. Therefore, the `owner_status` field is cleared if the lock is held by the owner (line 64), otherwise the `other` field is cleared (line 67).

[6]The `rcnt` field is modified by writing to the leftmost 8 bits of the lockword to avoid overwriting the `owner_status` field, which may be modified by the owner thread even while the lock is held by a non-owner thread. Note that unlike the case of reservation lock in Chapter 5, the `rcnt` field remains zero in the outermost acquisition, which is same as in the original tasuki lock.

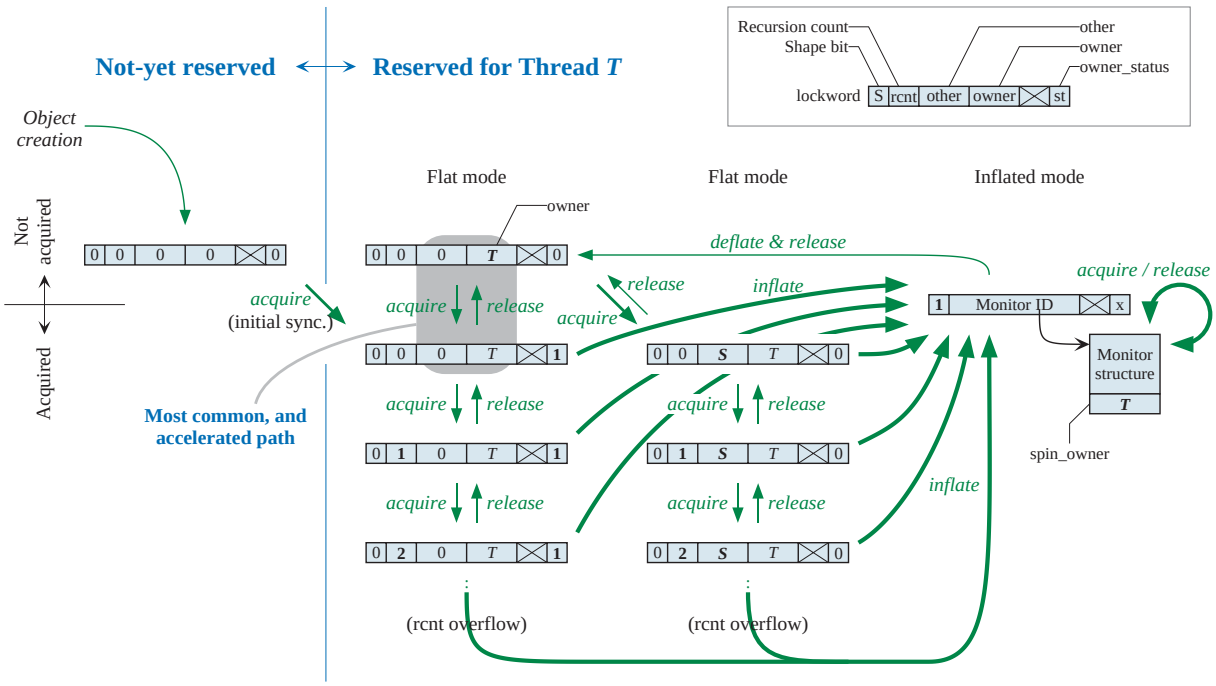

Figure 6.13: All lock state transitions of the asymmetric tasuki lock

By using the modified asymmetric spin lock with the tasuki lock shown in Figure 6.10, a new reservation-based Java lock was created which is free from the overhead of reservation cancellation while the owner thread can acquire the lock very quickly, without any atomic operations.

Figure 6.13 shows the complete state transitions of the *asymmetric tasuki lock*, where the thickness of each arrow informally represents the cost of the transition. This figure also illustrates the `rcnt` processing which is omitted in the pseudo-code in Figure 6.10. The flat mode acquisition by the reservation owner T is accelerated. Since the algorithm removed the mode change of reservation cancellation, the accelerated path can still be used for T even after the second thread S appears.

6.6 Performance Results

In this section, we evaluate the effectiveness of the new *asymmetric lock* with the IBM Developer Kit for Windows, Java Technology Edition, Version 1.4.0 [53]. The virtual machine in the developer kit contains a synchronization subsystem based on the tasuki lock [88]. We implemented both the new reservation-based algorithm and the original reservation lock shown in Chapter 5, and compared the two algorithms, using the tasuki lock in the original virtual machine as the base algorithm. The JIT compiler [58, 102, 103] included in the JVM was also modified to support the new algorithms.

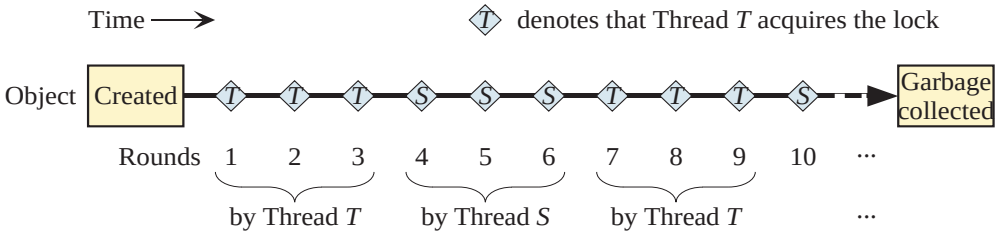

Figure 6.14: Lock sequence for the micro-benchmark

The new reservation-based Java lock is implemented by the method of replacing the spin lock in the tasuki lock with the one-word asymmetric spin lock as explained in Section 6.5.3. Although it was omitted in the explanation, the actual implementation utilizes the `rcnt` field appropriately to allow shallow recursive acquisitions in the flat mode. The implementation of tasuki lock and reservation lock are almost the same as described in Chapters 4 and 5, respectively. These also include the support for shallow recursive acquisitions.

All of the benchmark programs were run under Windows XP Professional Edition with SP1 on an unloaded IBM IntelliStation M Pro with two 933 MHz Pentium III processors and 512 megabytes of RAM.

6.6.1 Micro-Benchmarks

First, we show the results of a micro-benchmark which clearly highlights important behavioral characteristics of the three Java lock algorithms — the tasuki lock, reservation lock, and asymmetric lock. The benchmark creates two threads T and S, which alternately perform three rounds of computation. In each round, T or S executes the following loop:

```
1 : for (int i = 0; i < MAX; i++) {
2 :     synchronized (a[i]) { counter++; }
3 : }
```

Here the variable `a` holds a `MAX`-element array of objects. Notice that, when the loop is executed for the first time, each of the `MAX` locks is acquired for the first time. Figure 6.14 shows the lock sequence of each object in the micro-benchmark.

Figure 6.15 shows the execution times of the three algorithms for the initial few rounds of execution, relative to that of the base algorithm in the first round. For each of the rounds, the thread which executes the round is shown under the round number.

Let us first consider the performance of the base algorithm, tasuki lock. Although two threads are involved, the lock is not contended at all, since the two threads run alternately in the test. Thus, the base algorithm shows the same uncontended performance in all of the rounds. As mentioned in Section 6.5.2, tasuki lock allows a lock to be acquired and

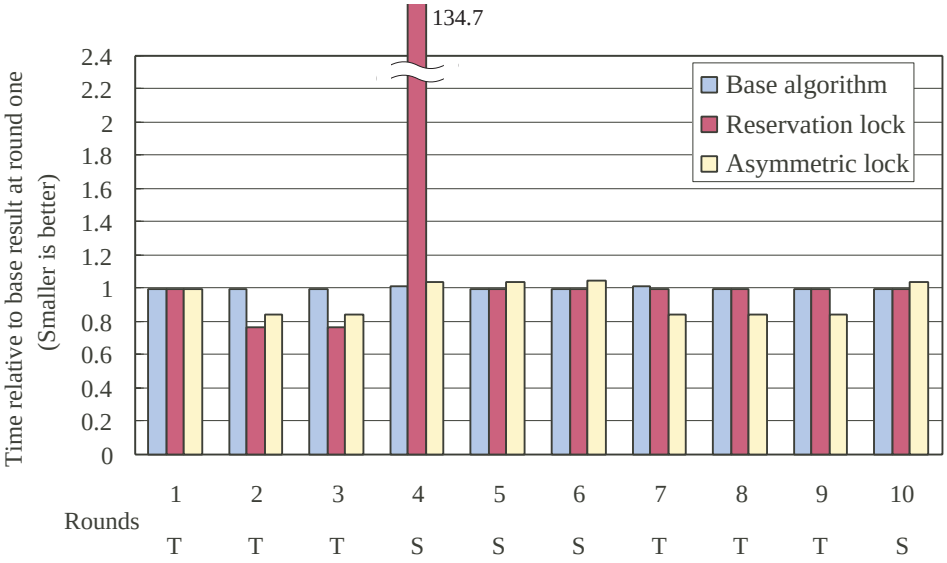

Figure 6.15: Micro-benchmark results

released in the absence of contention with a few machine instructions containing only one `compare_and_swap`.

Next, we consider the performance of the original reservation lock. As we mentioned in Section 6.2, the algorithm adopts the reservation policy of first acquirer. Thus, a thread attempting to acquire the lock for the first time, T in this case, reserves the lock for itself with a `compare_and_swap` (Figure 6.1(a)). This happens in the first round in Figure 6.15, so it took almost same time as the base algorithm.

Since the lock is now reserved for T, the performance in the second and third rounds is improved, since the thread can now acquire and release the lock without any atomic operations (Figure 6.1(b)). This is the power of the reservation lock, and the performance is improved if this situation continues long enough.

However, when S comes into play in the fourth round, the pathological behavior of the reservation lock manifests itself. In this round, S suspends T and cancels the reservation (Figure 6.1(c)), making the performance in this round more than 100 times worse than the performance of the base algorithm. In addition, after the fourth round, the reservation lock algorithm never exhibits the same level of high performance even for T's rounds as for the second and third rounds, since it has fallen back to tasuki lock (Figure 6.1(d)) in the fourth round and does not reserve the lock again. Therefore, the times for the following rounds are same as the base algorithm.

Now, we consider the performance of the new asymmetric lock algorithm. The same explanation applies for the first three rounds. In particular, we can observe a level of high performance close to the original reservation lock on a reservation hit as shown in the

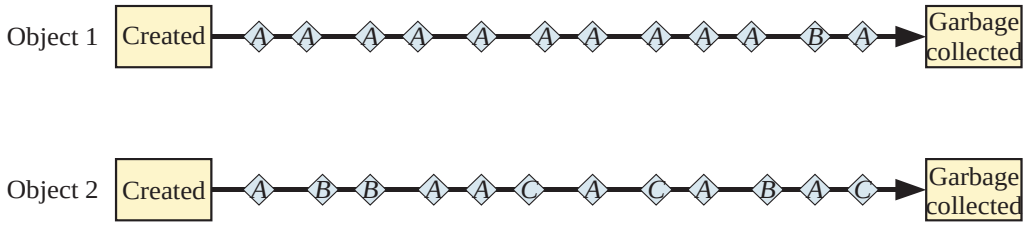

Figure 6.16: Two types of lock sequences

second and third rounds. Little degradation is observed in these rounds because of the additional cost to check the `other` field.

The subsequent rounds show two significant advantages of the new algorithm over the original algorithm. First, the new algorithm does not show any pathology in the fourth round, since it does not suspend the owner. Second, the algorithm shows the same level of high performance in all of T's rounds, since it does not cancel the reservation and the lock continues to be reserved for T.

Finally, the performance of the new algorithm in S's rounds is similar to the base algorithm. This is because the algorithm is a bimodal lock and the performance in the uncontended path is almost the same as the performance of the asymmetric spin lock. In each of S's rounds, the non-owner path of the spin lock is followed, which contains one `compare_and_swap` as the dominant cost. We believe that the slight degradation compared to the base algorithm comes from the additional cost to check the `owner_status` field.

From the result of the micro-benchmark, the characteristics of the two reservation-based Java lock algorithms are summarized as follows.

The original reservation lock is the most suitable algorithm for programs where each object's lock is used by a specific thread, because most lock operations are handled as in the second (or third) round in Figure 6.15. Object 1 in Figure 6.16 shows an example lock sequence for such a program. However, for programs where locks are mainly used for mutual exclusion among multiple threads, the algorithm may not be able to achieve much performance improvement, since most locks are processed as in the cases after the fifth round in Figure 6.15. Object 2 in Figure 6.16 shows an example lock sequence for that kind of program. In addition, if many objects' locks are used for such mutual exclusion, the performance may even be harmed by the cancellation overhead of the fourth round.

The new asymmetric lock can also accelerate programs of the Object 1 type, but it is not as fast as the reservation lock. However, the new algorithm can even accelerate programs in the Object 2 group if many lock operations are performed as in the seventh round.

Based on the above qualitative analysis, the next subsection shows quantitative results using more realistic Java programs, as well as their lock statistics.

Table 6.1: Benchmark programs

Program	Description
SPECjvm98	Run each program 3 times in the application mode
_202_jess	Expert shell system solving a set of puzzles
_201_compress	LZW compression and decompression
_209_db	Perform database functions on memory resident DB
_222_mpegaudio	Decode MP3 audio files
_228_jack	Parser generator generating itself
_213_javac	Java source-to-bytecode compiler from JDK 1.0.2
_227_mtrt	Two-threaded ray tracer
SPLASH-2	Multi-threaded scientific benchmark programs ported to Java
Water	An evaluator of forces and potentials in water molecules
Barnes	A simulator of an N-body system using the Barnes-Hut method

6.6.2 Macro-Benchmarks

We now show the results of more realistic applications, seven programs from SPECjvm98 [100] and two scientific applications from the SPLASH-2 benchmark set [120]. We ran each of the SPECjvm98 benchmarks in the application mode, specifying the problem size as 100% and the number of executions as three. The scientific applications are Water and Barnes which were written in Java by the authors of [97][7]. We chose them as possible candidates where locks are mainly used for mutual exclusion. Table 6.1 shows descriptions of these programs. Among these programs, _227_mtrt, Water, and Barnes are multi-threaded programs.

Table 6.2 shows the synchronization statistics of the benchmark programs. We collected these statistics by running a special version of the virtual machine and JIT compiler which counts the events of our interest. As the table shows, even when JIT compiler is enabled, locks are very frequently acquired (and released) in all of the programs except _201_compress and _222_mpegaudio. This happens even in single-threaded programs, which is one of the unique characteristics of Java.

For each program, the table also shows the ratios of reservation hits of the two algorithms. We compute the ratio as the number of *outermost* acquisitions by the owner thread divided by the total number of lock acquisitions. That is, we exclude recursive acquisitions by the owner thread, since they are already optimized in the base algorithm and performed without any atomic operations. In other words, the number becomes the ratio of lock acquisitions that can be accelerated by each algorithm.

As shown here, both of the reservation-based algorithms can speed up the vast major-

[7]We thank Alexandru Salcianu and Martin Rinard for making their benchmark programs available to us.

Table 6.2: Lock statistics of macro-benchmarks

Program	Objects sync'd	Number of lock acquisitions	Reservation lock		Asymmetric lock
			Ratio of reservation hits	Number of cancel-lations	Ratio of reservation hits
SPECjvm98					
_202_jess	12,800	14,977,053	99.353%	187	99.356%
_201_compress	2,462	35,382	85.764%	127	86.868%
_209_db	66,800	170,834,005	99.982%	52	99.982%
_222_mpegaudio	2,111	31,201	88.327%	91	89.028%
_228_jack	538,631	46,972,114	95.822%	144	95.859%
_213_javac	133,448	43,820,079	98.662%	1,760	99.676%
_227_mtrt	3,358	3,528,225	99.451%	114	99.548%
SPLASH-2					
Water	858,230	4,326,541	43.668%	6,022	44.342%
Barnes	216,459	2,064,200	25.245%	78,819	34.076%

ity of the lock acquisitions in the SPECjvm98 benchmarks, while they can still accelerate many though not most of the acquisitions in the scientific applications. Also, as expected from the behavioral characteristics, the hit ratios are always higher for the measured benchmarks in the new asymmetric lock than in the original reservation lock. The difference is largest in Barnes.

As described in the previous subsection, the original reservation lock may suffer if many cancellations occur. To check this, for the algorithm, the table also shows how many cancellations of reservations occur in each program. The cancellations rarely happen in the SPECjvm98 programs[8]. The number is slightly larger in _213_javac, but it is still only about 1.3% of the number of synchronized objects. Since the number of lock acquisitions are also large in the program, we expect the penalty of cancellation to be relatively small.

On the contrary, the cancellations are frequent in the two scientific applications. Especially in Barnes, 36% of synchronized objects' lock reservations were canceled. We suspect that frequent cancellations and the lower hit ratios just mentioned above resulted because these two programs were translated from the C versions, and do not rely on Java's standard library as much as the other programs.

Figure 6.17 shows the speedup ratios of the reservation-based algorithms to the base algorithm[9]. We took the best times from repeated runs.

[8] A few cancellations occur even in single-threaded programs because internal helper threads created by the virtual machine acquire some locks.

[9] The values shown in Figure 6.17 are slightly different from those in our previous papers [67, 89], where the *reduction ratios* of execution time were actually calculated. Here, I recalculated with the same data to show the *speedup ratios* correctly.

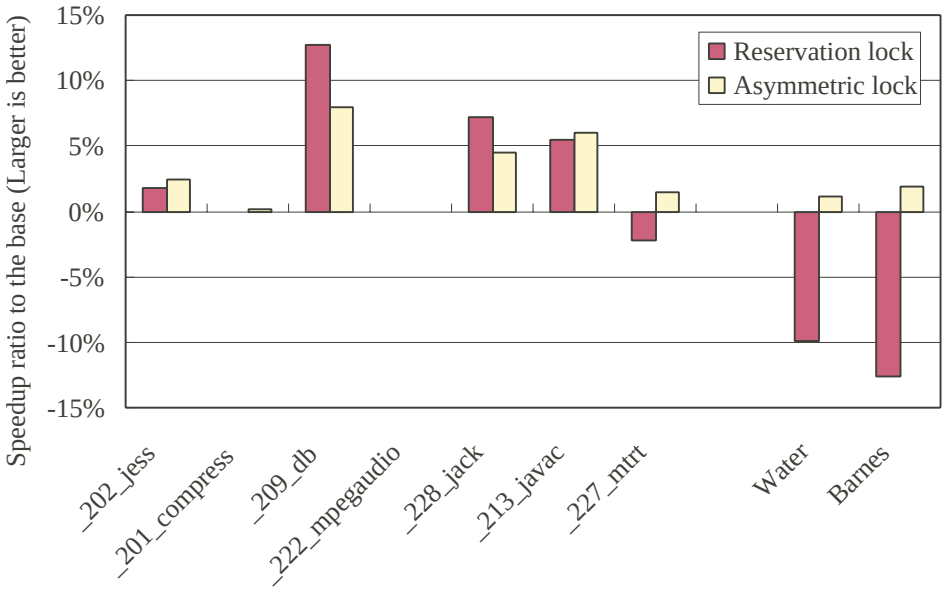

Figure 6.17: Macro-benchmark results

Let us first consider the performance of SPECjvm98. The original reservation lock showed a maximum speedup of 12.7% in `_209_db` but slows down in `_227_mtrt`. The new asymmetric lock showed a maximum speedup of 7.9% in `_209_db`, no slowdown for `_227_mtrt`, and greater speedups in `_202_jess` and `_213_javac` than the reservation lock. Neither significantly improved `_201_compress` and `_222_mpegaudio`, since these programs involve very few synchronization operations and are little affected by the performance of the Java locks.

Considering the most effective program, `_209_db`, its execution time was reduced by 11.3% and 7.4% from the base algorithm by reservation lock and asymmetric lock, respectively. Since the lock overhead in the execution time of this program was about 11.9%[10], it is roughly estimated that more than 90% of the overhead was removed by reservation lock for this case. In contrast, asymmetric lock can remove only about 60% of the overhead, but it also removes the penalty of cancellation, which will be described below.

We now turn to the performance of the two scientific programs. The original reservation lock degraded these two programs. In particular, it causes a 12.6% of slowdown in Barnes. This shows that canceling reservations by suspending the owner threads could have a significantly negative impact even on macro-benchmarks. The new asymmetric lock, on the other hand, did not exhibit such anomalous behavior because cancellation is unnecessary, and it even improved the performance of these scientific applications.

[10]The number was measured on the same processors by using a special JVM modified to actually do nothing during lock processing.

We should note that the results shown here are slightly different from those in Figure 5.11 in the previous chapter. Our considered opinion is that this is mainly because the processors and Java environments used for the measurements are different. That is, on a processor where the cost of `compare_and_swap` is not so high, the merit of reservation lock by removing these operations becomes relatively small, and the overhead of reservation cancellation may become dominant. This becomes conspicuous in some scientific programs which strictly maintain mutual exclusion among threads by themselves, without depending on standard Java libraries.

Actually, the motivation for developing the new reservation-based Java lock is that we found such cases in real programs. As shown in the macro-benchmark results, the new asymmetric lock successfully remedied the anomalous behavior and improved the performance for the combination of such a processor and benchmarks.

6.7 Summary

In this chapter, we have presented a new reservation-based algorithm for Java lock. Like the previous reservation lock, it reserves locks for specific threads, allowing the owner thread of a lock to acquire and release the lock without any atomic operations. Thus, the new algorithm attains a similar level of high performance on a reservation hit. Furthermore, unlike the previous algorithm, it does not require a non-owner thread to suspend the owner in order to cancel the reservation. Thus, the algorithm no longer suffers from the anomalous behavior the original algorithm exhibits on a reservation miss. Also, it allows programs to run with more reservation hits, since it does not cancel reservations.

We obtained the algorithm by replacing a CAS-based spin lock in a bimodal lock with our new spin lock, called the asymmetric spin lock. We devised the asymmetric spin lock by applying lock reservation to spin lock for the first time, and by hybridizing a CAS-based lock and a Dekker-style lock.

We have evaluated an implementation of the new algorithm in IBM's production virtual machine and JIT compiler. The results of micro-benchmarks show that, compared to the original reservation lock algorithm, the new asymmetric lock algorithm achieves a similar level of high performance on a reservation hit, shows no anomaly on a reservation miss, and also yields more reservation hits. For macro-benchmarks, while the new algorithm achieved up to 7.9% speedups in the SPECjvm98 benchmarks, it even improved the performance of two scientific programs which the previous algorithm degraded by more than 10%.

To summarize, the research shown in this chapter contributes the following results:

- A new algorithm for spin lock
  We devised a novel optimization for spin lock based on lock reservation. The *asymmetric spin lock* allows the owner thread of a lock to acquire and release the lock

with a few read and write instructions, while it does not cause a non-owner thread to suspend the owner thread.

- A new method of introducing a new feature to Java lock
  We showed that a new Java lock can be constructed by *replacing a spin lock* of bimodal Java lock algorithms.

- Evaluation on a production virtual machine and JIT compiler
  By using the new method with the asymmetric spin lock, we developed a new reservation-based Java lock, *asymmetric lock*, on a production Java virtual machine and JIT compiler, and measured the performance on a multiprocessor system, using an industry-standard benchmark set and scientific programs.

Chapter 7

Related Work

7.1 Introduction

The topic of locks has been addressed by a significant number of papers and textbooks, even before the appearance of Java. This chapter introduces this research while categorizing it from several viewpoints. Our three approaches to Java locks are also put into context by comparing them with these alternatives.

First, in Section 7.2, we revisit general lock techniques without limiting the area to Java. Next, Section 7.3 picks up several lock methods without atomic operations, comparing them with our reservation-based locks. Following two sections deal with research on Java locks. Section 7.4 covers various algorithms for Java locks and compares them with ours. Section 7.5 discusses another approach to accelerate Java locks which was not treated in this thesis, the elimination of locks. Finally, Section 7.6 gives the summary of this chapter.

7.2 Generic Locks

As already described several times, the locks for synchronizing threads (or processes) can be divided into two categories: *spin* and *suspend* locks. A spin lock is usually implemented with *atomic* operations such as `compare_and_swap`, `test_and_set`, or `fetch_and_add`, by continuously performing the operation until the lock is acquired. On the other hand, suspend locks, typical instances of which are semaphores [29] and monitors [20, 21, 46], are integrated with an operating system's scheduler. They support the suspension of threads and do not waste CPU resource when they cannot acquire locks. However, since the suspend locks are implemented in kernel space to cooperate with the scheduler, they are usually heavier than spin locks.

Anderson studied the performance of various algorithms for spin locks on multiprocessor systems, and discussed optimizations such as *spin on read* and *exponential back-off*

[5]. Mellor-Crummey and Scott proposed a sophisticated spin-lock algorithm, named *MCS locks* from their initials, that performs efficiently in shared-memory multiprocessors of arbitrary size [80]. The key to the algorithm is for every processor to spin only on separated locally accessible locations, and for some other processor to terminate the spin with a single remote write operation. Although the algorithm constructs a queue of waiting processors in the presence of contention, it is a spin-lock algorithm, and neither hybrid nor bimodal.

Basically, our work on Java locks is orthogonal to theirs, and an interesting future research direction would be to use their algorithm for acquiring a lock in the flat mode, or in the spin-lock mode, in our algorithms.

The idea of hybridizing spin locks and suspend locks as *spin-suspend locks* was first suggested by Ousterhout [90], leading to subsequent work on the *spinning strategy*. Karlin, Li, Manasse, and Owicki empirically studied seven spinning strategies based on the measured lock-waiting-time distributions and elapsed times [64], while Lim and Agarwal derived static methods that attain or approach optimal performance using knowledge about the likely wait-time characteristics of different synchronization types [75]. Based on the concept of spin-suspend locks, recent versions of the Linux operating system provide a fast synchronization mechanism named *futex* [36], which exposes the spin lock to the user level. In the uncontended case, a futex lock can be acquired by an atomic increment operation and released by an atomic decrement operation, without using system calls.

As mentioned in Chapter 2, bimodal locks for Java, first proposed by using the thin lock [11], are space-efficient hybridizations of spin locks and suspend locks. The findings for spin-suspend locks are thus applicable to bimodal locks, which includes all three of our algorithms as well as the original thin lock. Actually, Dimpsey, Arora, and Kuiper proposed a three-tier spin/yield/block solution for Java locks for multiprocessor environments, where a thread that failed to acquire a lock in the inflated mode spins then yields for a while before blocking [31]. BEA's JRockit JVM [13] also adopts a similar spinning mechanism for multiprocessor to reduce the context-switching cost and negative cache-effects [98].

As for the memory consumption, the space overhead has never been a concern in the research on spin-suspend locks. In other words, it is totally acceptable to add one word to a suspend lock structure for spin lock. However, for Java locks, the idea of the bimodal use of a single field in the thin lock was a painful one to reach, because the bits in an object's header are extremely precious, and increasing the header size is simply prohibitive.

In the area of locks for objects, Mesa [71, 81] provides language support for *monitoring* of large numbers of small objects. A *monitored record* is a normal Mesa record, except that it implicitly includes a field for a monitor lock. It is of a predefined type named `MONITORLOCK`, which can be thought of as the following record:

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

The size is one word (16 bits at that time) which suggests packing. In our terminology,

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

Although some Java implementations, notably earlier versions of Kaffe [63], take a similar approach as shown in Section 2.3.2, the space overhead is prohibitive, since Java permits recursive lock acquisition and allows any arbitrary object to be involved in long-term synchronization.

7.3 Locks without Atomic Operations

In Chapters 5 and 6, we examined methods to accelerate Java locks by removing atomic operations[1] from their common cases. Actually, there are several methods that can perform locks without atomic operations.

Early work on locks focused on achieving mutual exclusion using only the atomicities of read and write instructions, because no complex atomic operations were available. Numerous algorithms were proposed including those of Dekker [30], Dijkstra [29], Peterson [93], and Lamport [70]. However, the number of read and write instructions required in these algorithms is proportional to the number of threads (or processes), preventing them from being used in practical systems. Furthermore, academic interest in such algorithms quickly dwindled with the widespread adoption of hardware support for complex atomic operations such as `compare_and_swap` (CAS).

As discussed in Section 6.4.1, our asymmetric spin lock can be considered as a revitalization of Dekker's idea [30], by dividing threads into the two categories of *reservation owner* and *others*. For the others group, a thread is first chosen by using an atomic operation as a *representative* to compete with the reservation owner. By hybridizing the CAS-based lock and the Dekker-style lock, our algorithm only requires a few read and write instructions on a reservation hit, at the cost of forcing all of the non-owner threads to perform an atomic `compare_and_swap` instruction against a single field.

By using a similar idea, Frigo, Leiserson, and Randall implemented a *work-stealing* mechanism in the Cilk-5 multi-threaded language [37], which is an extension of C for parallel processing. Their *THE protocol*, by which a *thief* thread steals work from a *worker* thread, gives the worker thread a priority (or reservation) which makes it possible for the thread to check the status without using heavy atomic operations in common cases. Compared to their protocol, one unique feature of our asymmetric spin lock is that it utilizes the action of setting a non-owner thread's identifier to the `other` field by

[1] We should note that again, the *atomic operations* mentioned in this thesis refer to complex compound instructions that perform memory-reading, testing or modifying, and writing without being interrupted, such as `compare_and_swap`, `test_and_set`, and `fetch_and_add`.

`compare_and_swap` as the first step of a Dekker-style lock. We also mention that the lock is packed into one word and applied to a more generic situation, Java locks, than the limited situation of work-stealing.

Bershad, Redell, and Ellis proposed a unique lock algorithm that closely cooperates with the operating system's scheduler [15]. In this method, when a thread is preempted inside a critical section of lock processing, it will be forcibly restarted from the entry point of the section by the scheduler. The *restartable atomic sequence* proposed by them resembles the unsafe regions of our reservation lock algorithm, which are also restartable. Actually, as discussed in Section 5.4.3, their idea of marking each atomic sequence with a *designated code sequence* can be used in our reservation cancellation system. By extending Bershad's idea, Johnson and Harathi proposed *interruptible critical sections*, which support the modification of multiple data objects [62].

The idea of *restartable critical sections* has been successfully used in the area of real-time processing [112]. However, replacing Java locks with such an optimistic synchronization model is not so easy, because the synchronized sections of Java programs are not originally written as restartable and are sometimes too long to log the activities inside them, as measured in Section 4.4.3. By limiting the target to short `monitored` sections, Welc, Jagannathan, and Hosking introduced the optimistic synchronization named *transactional monitors* into Java [115].

When a Java virtual machine uses its own user-level threading mechanism instead of OS-provided threads, we can use these scheduler-based techniques to implement Java locks. Actually, both the CACAO [68] and LaTTe [122] Java systems implement locks without atomic operations by inhibiting thread switches inside the critical sections. However, the scheduler-based locks are only effective on a uniprocessor system, so cannot exploit the benefits of multiprocessing. Moreover in Java, they may cause starvation when a foreign function is called through the Java Native Interface [74], and the foreign function attempts to acquire a system-level lock. On the contrary, our reservation-based locks work properly on a multiprocessor system and under the system-level, preemptive scheduler. The lock is accelerated only for the reservation owner, but we already showed that this still has much merit in Java because of the thread locality.

The communities of database systems and distributed file systems invented many optimization techniques based on access locality [110], which is similar to our thread locality. Kung and Robinson proposed an *optimistic concurrency control* for database systems, which speculatively executes critical regions without acquiring locks and commits the changes if there is no contention [69]. Rajwar and Goodman proposed a technique to implement a similar idea at the micro-architectural level [95]. The transactional monitor [115] mentioned above can also be considered as adapting the optimistic concurrency control to Java.

Microsoft's CIFS distributed file system includes a file-lock mechanism called *opportunistic locks* or oplocks [33, 73]. When a client is granted an exclusive oplock for a file, it can cache the file's data for better performance. If another client attempts to open the

file, the server sends the client holding the oplock an *oplock break* request to return the cached data. This resembles reservation cancellation in our reservation lock algorithm, although the granularity differs greatly.

7.4 Acceleration of Java Locks

Java created renewed interest in algorithms for locks, because very frequent and ubiquitous synchronizations caused a significant performance penalty in early virtual machines [6, 43]. Many techniques have since been proposed for optimizing locks in Java, which can be divided into two categories, runtime techniques and compile-time techniques. The runtime work attempts to make lock operations cheaper, while the compiler approaches attempt to eliminate lock operations.

All three of the lock methods proposed in this thesis are categorized as runtime techniques. We have already covered the significant runtime techniques in Chapter 2, but we briefly revisit them here while adding other research to clarify the positions of our methods.

In early versions of Java virtual machines, OS-provided monitor mechanisms were used directly for implementing Java locks, so their cost was very high [6, 43, 87, 123]. Bacon et al. disrupted this situation with their thin lock, based on the observation that most locks are not contended in Java and a heavyweight monitor which supports thread suspension is unnecessary in many cases [11]. The thin lock was the first bimodal lock algorithm for Java, where a lockword in each object's header was used for both the spin-lock and suspend-lock modes. We have already fully described that algorithm in Section 4.3.

Their work inspired us to deeply consider issues inherent in bimodal locks. In the thin lock, once a lock is inflated by contention, it is processed by using a suspend lock after that. We discovered that many contentions are temporary in Java, and proposed the tasuki lock [88], which was presented in Chapter 4. Allocating one additional bit in an object's header to represent the contention status, the method allowed for *deflation* while also removing the busy-wait from the inflation process. Importantly, no atomic operations were added in the uncontended path. The tasuki lock is now widely used in IBM's production Java virtual machines [53]. Gagnon and Hendren also implemented a variation of tasuki lock in the SableVM [38]. They put the contention bit in the thread structure, instead of in the object's header, to reduce the per-object memory overhead. Whaley's Joeq VM also uses the tasuki lock with Gagnon's extension [117].

As similar research which accelerated the uncontended case, Agesen et al. proposed another algorithm for Java lock, called *meta lock* [3]. While it is highly space-efficient since it only uses two bits of a word in each object's header, it requires at least two atomic operations in acquiring and releasing a lock. Thus, it is not as time-efficient as thin lock and tasuki lock, which require only one `compare_and_swap` for these operations. Furthermore, when contention happens, the meta lock requires a full word (32 bits), and

moves the 30 remaining bits to an out-of-line data structure. Thus, frequently accessed information such as a class pointer cannot be stored in the remaining bits. Dice proposed a modified version of meta lock named *relaxed lock*, where atomic operations are used only at lock acquisitions. However, its uncontended execution path seems to be not as optimized as thin lock and tasuki lock.

There are few public descriptions about how locks are implemented in production Java execution environments other than IBM's [53], where (an optimized version of) tasuki lock is used. Sun's HotSpot JVM [107] claims that it provides "ultra-fast, constant-time performance for all uncontended synchronizations" [109]. This is considered to be an optimization similar to our tasuki lock, so that heavyweight OS monitors are not necessary for uncontended cases. Actually, the meta lock mentioned above seems to be used, at least in an early version of the HotSpot JVM [91]. BEA's JRockit JVM [13] seems to adopt the concept of thin lock with the fat lock extension of multiprocessor spinning [14], as mentioned in Section 7.2. They claim that it requires only one atomic operation at lock acquisition and no atomic operations at lock release [98].

Although the details are significantly different, these fast algorithms including our tasuki lock allow a lock to be acquired and released with a small number of instructions if it is not contended. However, the instruction sequences of each algorithm still contain one or two atomic operations, which have been becoming relatively more and more expensive in modern multiprocessor architectures. Our reservation lock [65, 66] shown in Chapter 5 is an attempt to remove the last-remaining overhead of atomic operations. This makes it possible for a lock to be *reserved* for a thread to exploit the observation that most locks are not only uncontended in Java, but also dominantly acquired by a specific thread. We note that Bacon and Fink independently proposed a similar idea of eliminating atomic operations in Java locks, where they gave the creator of an object preference about the lock [10].

As explained in Section 6.2, while the reservation lock significantly reduces the synchronization overhead on a reservation hit, it must suspend the owner on a reservation miss. A similar idea of reservation was proposed by Gomes et al. [40]. Their *speculative lock* algorithm also requires a heavyweight fix-up operation to be performed on speculation failure. Our asymmetric lock [67, 89] in Chapter 6 is positioned as another implementation of the reservation concept, but it eliminated the reservation cancellation with a unique approach. It first introduced an asymmetricity to a spin lock, which is a core primitive of all of the lock implementations, and then showed that a new reservation-based lock can be constructed by putting the *asymmetric spin lock* into the bimodal lock algorithm.

Recently, Ogasawara et al. proposed a Java lock also based on the concept of reservation, named *TO-Lock* [86]. Their algorithm is interesting because it supports the *migration* of the reservation ownership instead of canceling it. Although they showed only micro-benchmark results for the reserved path, ownership migration is a possible direction for the reservation-based locks if its cost is really small.

To further accelerate Java locks, investigation of real applications is indispensable. We

believe that the various lock statistics shown in this thesis, such as Tables 4.2, 4.3, 4.4, 5.2, 5.3, 5.6, and 6.2, are useful for that purpose. As for more exhaustive investigation, Dufour, Driesen, Hendren, and Verbrugge carefully studied the dynamic characteristics of many kinds of Java applications using various metrics [32]. For synchronization, they defined the following five metrics: average number of lock acquisitions per execution of 1,000 bytecodes, locality of lock acquisitions (percentage of locks responsible for 90% of the lock acquisitions), percentage of contended lock acquisitions, average number of contended lock acquisitions per execution of 1,000 bytecodes, and locality of contended lock acquisitions. Their measurement results are useful to understand the characteristics of Java applications and to design new Java locks.

7.5 Elimination of Java Locks

Another approach to improve the performance of Java locks is to eliminate the locks altogether rather than to reduce the cost of the locks, as explained in the previous section. As already shown in Table 5.3 of Chapter 5, Java programs contain many objects each of whose locks is processed by a specific thread. Especially for these objects, there is a chance of performance improvement by completely eliminating their lock operations, even beyond the savings of reservation-based locks.

The most common elimination technique in Java is to discover objects accessible only by their creator threads by using *escape analysis* [92], and to eliminate all lock operations for such non-escaping objects [4, 16, 17, 24, 25, 96, 97, 118]. This technique is effective in static offline Java compilation environments, where entire programs can be analyzed at compilation time. However, Java is essentially a dynamic language which allows class loading during execution, and there are limits to escape analysis in such an environment. When applying escape analysis to Java, many objects must conservatively be judged as escaping, and their locks cannot be optimized away.

For example, in one of the papers [17], only a little improvement was observed by lock elimination for the `_209_db` program of SPECjvm98 [100], even though it is a single-threaded lock-intensive program [32]. As another example, the JVM used in the experiments of Chapter 5 performs lock elimination in its JIT compiler [58, 102, 103] using dynamic escape analysis based on [118], but the total number of locks was not significantly reduced from Table 5.2 (no optimization by JIT) to Table 5.6 (with optimization by JIT). Whaley proposed *partial method compilation* for improving the effectiveness of escape analysis for a dynamic language [116].

In our reservation-based locks, all lock operations to non-escaping objects are accelerated because such objects' locks will be acquired only by the creator threads and their reservations are never canceled. In addition, lock operations to escaping objects are also accelerated if they are performed only by a specific thread. Typical examples of such objects are found in the Volano Mark programs [114], where objects created by a single

thread are passed to other worker threads, as already described in Section 5.2. These objects are escaping from the creator thread, but their locks can still be accelerated by giving the reservation to their first acquirers, because each lock is processed only by a specific worker thread.

The reservation-based lock approach has another merit in that it is applicable to interpreter environments, since they do not need analysis before execution. However, all lock processing cannot be totally removed from the lock algorithm, since reservation checks and recursion counter management must be done. Therefore, it is desirable to first eliminate locks by escape analysis and accelerate (a part of) the remaining locks by reservation.

There are several techniques to eliminate *recursive* locks. For example, when a `synchronized` method is inlined into another `synchronized` method, the JIT compiler can eliminate the inner locks if it detects that the receiver objects of these methods are always identical [101]. A similar optimization is possible in writing Java programs. Useless inner locks can be removed by not invoking a synchronized method from another synchronized method for the same object. This technique is widely used in recent Java core class libraries [45].

It was pointed out that shallowly nested lock acquisitions occur frequently in Java [11]. To exploit this situation, the actual implementation of thin lock included the `rcnt` (recursion count) field in the lockword. Shallow recursive locks can be acquired by just incrementing the field. The implementations of our three methods also have a similar mechanism to accelerate the shallow recursive locks. In addition, the reservation-based locks can accelerate the outermost lock acquisitions by removing the atomic operations from the fast paths.

If a program's behavior is correctly understood, unnecessary synchronizations can be eliminated during the programming [45]. To support this, several methods have been proposed to detect data races in Java programs using a type system [19, 35]. However, it seems to be difficult to apply these methods to general-purpose Java class libraries.

Hovemeyer, Pugh, and Spacco investigated a unique approach for reducing Java locks by recognizing special *idioms* in Java programs and converting them into code utilizing atomic machine instructions such as `compare_and_swap` [47]. Another idea of *exposing* such atomic instructions to Java programs has been discussed under Java Specification Request (JSR) 166 [61], and was recently adopted in JDK 5.0 [105] as a `java.util.concurrent`(`.atomic`) package, which provides several atomic-operation methods such as `compareAndSet`. This package finally made it possible to write *lock-free* algorithms [44] in Java [39].

Bacon attempted to eliminate all of the synchronization overhead from single-threaded executions [9]. As long as the system creates and runs only one thread, nothing is done for lock acquisition and release. When the running program attempts to create a second thread, the system scans the stack frames and properly recovers the lock states. Muller also briefly mentioned a similar idea [83]. Ruf proposed whole-program analysis to de-

termine if the program does not create a second thread [96]. Unfortunately, these ideas cannot be used in most of the commercial virtual machines, since they always create a couple of *helper threads*, besides the main thread, at start-up time.

Compared to these lock elimination techniques, our reservation-based locks can be used without modifying the Java applications, and are effective even in the dynamic Java environment. As shown in the results of Sections 5.5.3 and 6.6.2, the new lock methods can accelerate almost all of the locks in single-threaded programs and can even accelerate many of the locks in multi-threaded programs.

7.6 Summary

This chapter categorized related research into four areas: generic locks, locks without atomic operations, acceleration of Java locks, and elimination of Java locks. In each section, we introduced major research efforts in each category, and positioned our research in this thesis among them.

Figure 7.1 summarizes the evolution of locks explained in this chapter, while also positioning our three methods within it.

All of our Java lock algorithms included in this thesis incorporate the concept of spin-suspend locks, while using the lockword in each object's header in multiple modes. The bimodal use of lockword was first introduced in thin lock. The tasuki lock shown in Chapter 4 can be positioned as an extension of the thin lock, supporting deflation of lockwords, to exploit the contention transience in Java locks.

Locks without atomic operations were originally designed in an older period when processors did not have modern atomic operations. However, the reservation lock in Chapter 5 and the asymmetric lock in Chapter 6 revitalized these ideas by utilizing the thread locality of Java locks.

Since Java is a dynamic language, many lock operations remain even in the environment of lock elimination by JIT compilers. Our reservation-based locks can accelerate the remaining locks and improve the overall performance.

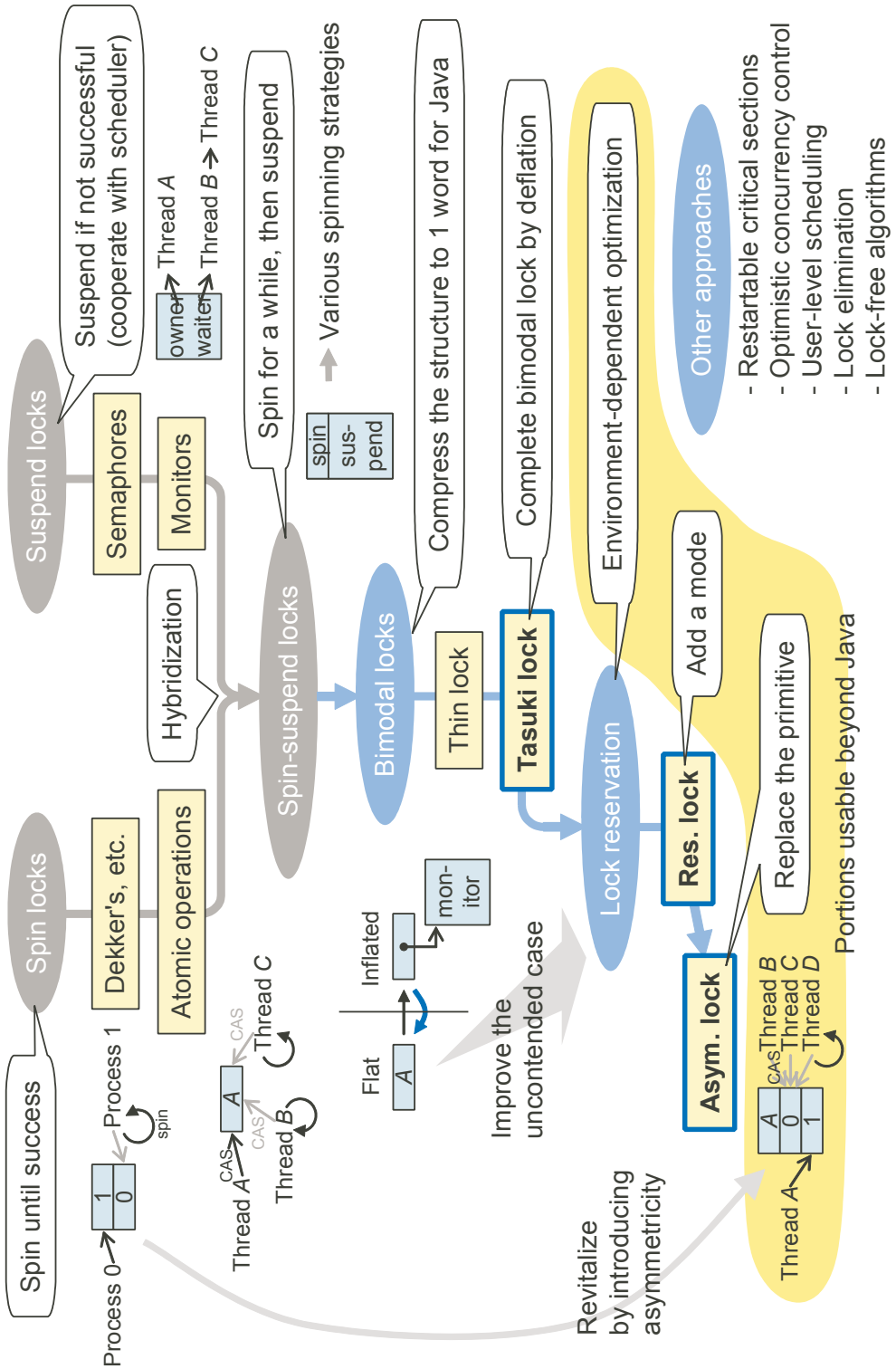

Figure 7.1: Evolution of locks, and positioning of our research

Chapter 8

Conclusion

8.1 Summary

In this thesis, we have described three techniques to accelerate Java locks, the tasuki lock, the reservation lock, and the asymmetric lock, each of which incorporated or improved the previous ideas.

At the problem definition in Chapter 3, we showed that the acceleration of Java locks has been done by first discovering the characteristics of Java locks from the analysis of real applications, and by reducing the cost in the typical-use patterns, which is the most *common case* derived from the characteristics. We then defined problems to be solved for further accelerating Java locks, as follows:

Problem 1: Discover new characteristics of Java locks.

Problem 2: Design new lock methods exploiting these characteristics, and evaluate them.

Problem 3: Show techniques to construct Java locks more easily.

As a solution of Problem 1, we first paid attention to the behavior of contended locks, and discovered the *contention transience*, that is most contentions are transient in Java. We also investigated the behavior of uncontended locks, and found the *thread locality* of Java locks, which means that a lock is primarily used by a specific thread in many cases.

As a solution of Problem 2, this thesis proposed three lock methods exploiting these characteristics. Figure 8.1 positions the characteristics of the Java locks utilized in these methods, along with those in the prior methods shown in Figure 3.1.

The tasuki lock proposed in Chapter 4 is a technique to exploit the contention transience, in which a lock can be *deflated* to lightweight mode once the contention goes away. The reservation lock in Chapter 5 enables each lock to be *reserved* for a specific thread to exploit the thread locality, where the reservation-owner thread can acquire the lock without any atomic operations. The asymmetric lock in Chapter 6 is a variation of the

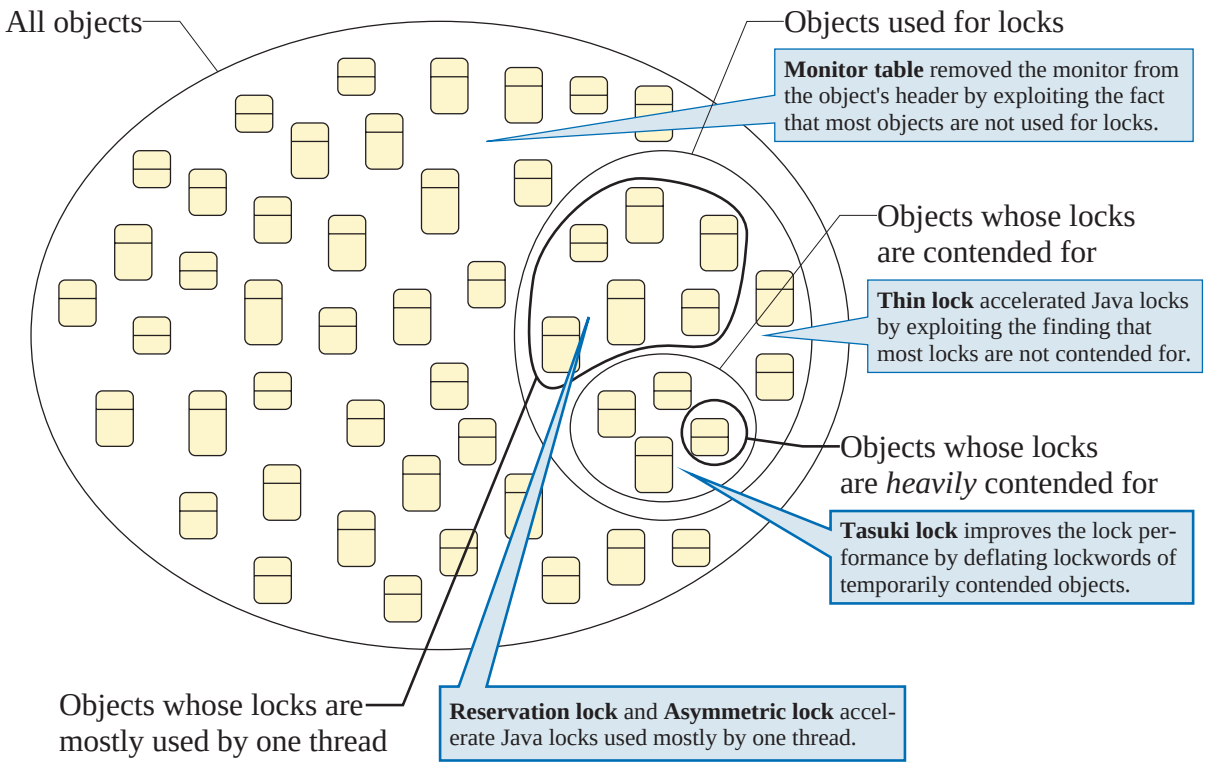

Figure 8.1: Revealed characteristics of Java locks and our lock methods exploiting them

Table 8.1: Target area and key characteristic of each lock method

	Frequent ← Uncontended locks: Used by one thread	Uncontended locks: Used by multiple threads	→ Rare Contended locks: Not-heavily contended	Contended locks: Heavily contended
(Thin lock)	**xxx**		xx	
Tasuki lock	xxx		**xxx**	xx
Res. lock	**xxxxx**	xx	xxx	xx
Asym. lock	**xxxx**	**xxx**	xxx	xx

The number of 'x's indicates the relative performance (more 'x's is faster).

Key characteristic
- Do not use monitors until contended.
- Return to the fast mode after contention.
- Accelerate locks until the 2nd thread appears.
- Always accelerate locks for specific threads.

reservation lock, but it enables a specific thread to always acquire the lock very quickly by eliminating the reservation cancellation step. Table 8.1 summarizes the target area and key characteristic of each lock method, including the thin lock. In the table, the number of 'x's indicates the relative performance of each lock method to the specified lock type, and bold '**x**'s indicate the lock types accelerated by the method.

The proposed three lock methods were actually implemented on IBM's production Java virtual machines, and their performance was measured. Although the measurement environments differ from each other, we observed performance improvements of real Java

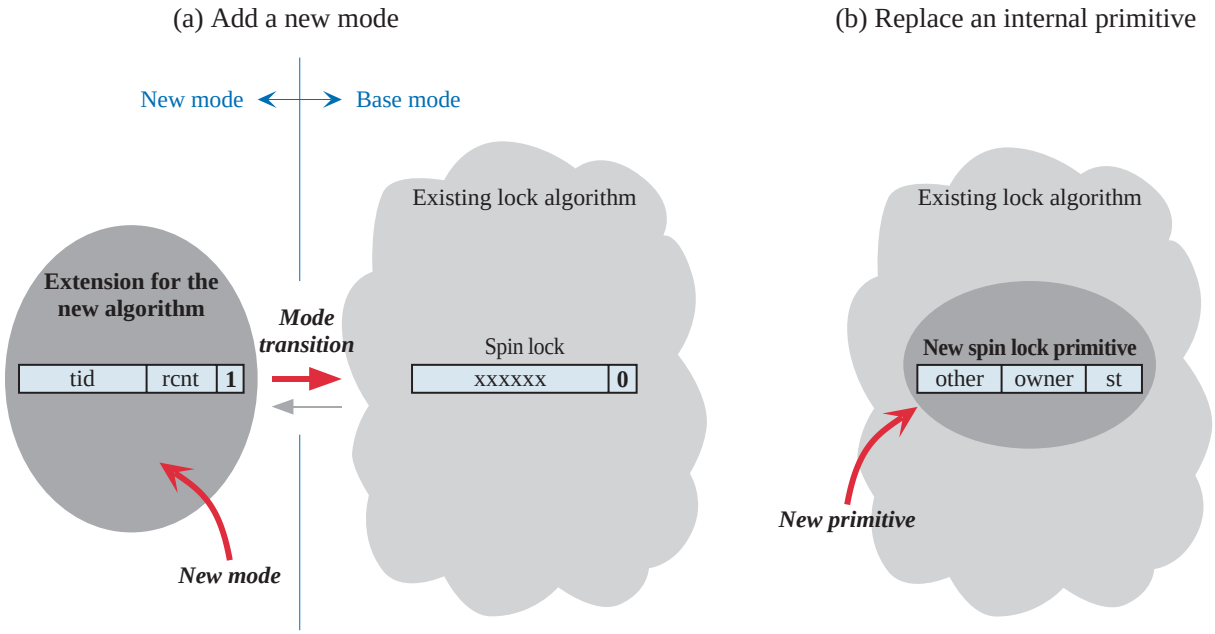

Figure 8.2: Two methods to extend existing lock algorithms

applications by up to 13.1% for the tasuki lock, and additional improvements by up to 53% and 7.9% for the reservation lock and the asymmetric lock, respectively.

As a solution of Problem 3, we showed in Chapter 5 a method to incorporate the concept of reservation while still taking advantage of the merits of the existing lock algorithm, by adding a mode flag (the LRV bit) to the lockword. In addition, in Chapter 6, we constructed an asymmetric lock algorithm through a unique methodology of replacing the core atomic operation of conventional algorithms with an asymmetric spin lock. Figure 8.2 illustrates these two methods to extend existing lock algorithms, where (a) corresponds to the reservation lock and (b) corresponds to the asymmetric lock.

8.2 Contribution

The primary contribution of this thesis is that it provided three new algorithms for Java locks to accelerate the performance of Java applications. The contributions of each research method were already described in Chapters 4 to 6, but we summarize them here to clarify the contributions of this thesis.

The first contribution is based on the discovery of two unknown characteristics of Java locks. We discovered the contention transience and thread locality of Java locks by analyzing their behavior in real Java applications from various viewpoints.

The second contribution is the proposal of new concepts and actual algorithms for Java locks. To exploit the contention transience and thread locality, we proposed new

```
1 : // Data structure
2 : struct {
3 :   volatile int owner_status;   // flag for frequent locker's acquisition
4 :   volatile unsigned int other; // ID of an infrequent locker, acquiring
5 :   int data;                    // shared data to be accessed exclusively
6 : } shared;
```

```
1 : // Code for a frequent locker
2 : spin_retry:
3 : shared.owner_status = 1;
4 : if (shared.other != 0) {
5 :     shared.owner_status = 0;
6 :     goto spin_retry;
7 : }
8 : // (access to shared.data)
9 : shared.owner_status = 0;
```

```
1 : // Code for infrequent lockers
2 : spin_retry:
3 : if (compare_and_swap(&shared.other,0,myID)
4 :           != SUCCESS) goto spin_retry;
5 : if (shared.owner_status != 0) {
6 :     shared.other = 0; goto spin_retry;
7 : }
8 : // (access to shared.data)
9 : shared.other = 0;
```

Figure 8.3: Generic usage pattern of the asymmetric spin lock

concepts of deflation and reservation. We also designed new algorithms for Java locks that incorporated the concepts. The algorithms are precisely designed to utilize previous ideas while reducing the overhead in the common cases.

The third contribution is implementations and evaluations in actual environments. We implemented the new algorithms in a state-of-the-art Java system, and showed actual performance improvements in real Java applications as well as in micro-benchmarks.

To summarize, this thesis directly contributes to improving the performance of Java applications by accelerating the processing of Java locks based on deep analysis of their behavior in real Java applications. It was reported that the market for Java application servers has grown to 1.1 billion U.S. dollars in 2002 [26]. We believe that our research into improving Java performance has contributed to the rapid spread of Java by making it possible to process complicated business logic with practical execution times.

In this thesis, we proved that locks can be further accelerated by utilizing characteristics specific to the target environment, Java. As a more general contribution, we believe that such an *environment-dependent optimization* approach is widely applicable to various research areas in computer science, especially in the situations where generic acceleration techniques have already reached their maximum levels. We also believe that the two lock construction methodologies shown in Chapters 5 and 6 and summarized in Figure 8.2 are usable beyond the world of Java and can lead to faster lock algorithms.

In addition, the asymmetric spin lock shown in Sections 6.3 and 6.4 are generally applicable to situations where the lock frequency is biased to a specific locker. Figure 8.3 shows pseudo-code to access such shared data using the asymmetric spin lock. One possible example of such situations is a thread scheduler on multiprocessor systems where a scheduling queue is prepared for each processor and is occasionally peeked at by other processors.

8.3 Future Work

The asymmetric tasuki lock shown in Chapter 6 is a lock algorithm which exploits all the characteristics of Java locks shown in Figure 8.1, including our two findings. In addition, it also eliminated the overhead of reservation cancellation, which was a potential problem in the original reservation lock. However, it is not necessarily the final form of Java locks. Research should continue to improve Java locks.

One possible future research direction is the examination of the *reservation policies*. Both of our reservation lock and asymmetric lock implementations adopted the *first-acquirer policy*, where a lock is reserved for the thread that attempts to acquire it for the first time. However, alternative policies may further improve the performance.

For example, in the case of our original reservation lock, if we can predict the categories of objects whose lock will be used by multiple threads, it becomes possible not to reserve those locks from the beginning, since their reservations will be canceled anyway. The performance degradation of the scientific applications observed in Section 6.6.2 may be remedied by this approach. In addition, for the asymmetric lock, if we can know which thread performs lock operations most frequently for each object in advance, through some profiling or code analysis, it would become possible to reserve the object's lock for that thread rather than its first acquirer.

As for research on the lock algorithm itself, it would be worthwhile to examine a technique for transferring or re-acquiring the lock reservation, or a technique for switching between multiple algorithms on the basis of the runtime statistics or execution environment.

Another interesting approach is to manipulate the objects' lockwords during the garbage collection. For example, it would be worth considering resetting the reservation status to the not-yet-reserved state, which would allow the lock to be reserved for an appropriate thread again.

The costs of atomic operations and memory barriers differ among processors or system configurations. One research candidate would involve dynamically tuning the lock algorithm at runtime by taking into account these costs. Since Java is a language executed by a virtual machine, such an adaptive feature can be rather easily implemented, for example, into its dynamic JIT compiler [58, 102, 103].

Such profile-based adaptive optimization and reconfiguration is considered to be one of the hottest research areas. Another possible approach in this area would be to change the processing of contended locks by dynamically estimating the remaining time of the synchronized section being executed.

Incorporating real-time features into Java has been discussed for a long time under the first Java Specification Request (JSR 1) [59], and is recently becoming realistic since Sun unveiled an implementation of the RTSJ (Real-Time Specification for Java) [18] in June, 2005 [22]. There are new opportunities to apply various real-time synchronization techniques [84, 111, 112] to such real-time Java environments.

Java has flourished for ten years since its birth, and has now come to be recognized as a practical programming language. During this period, the environments surrounding Java have changed a lot. For example, JDK 5.0 [105], which is the latest release from Sun, adopted a new memory model, JSR 133 [60, 72], where the memory access ordering rules are redefined strictly [8, 78]. Therefore, future research on Java locks must take into account, or possibly exploit, the new specification. In addition, Java is now being used to construct large-scale commercial applications such as Web services [108] using J2EE [106]. Investigation of the lock behavior and improvement of the lock techniques, including thread management, in such emerging environments, are also considered to be possible future work.

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, Vol. 29, No. 12, pp. 66–76, 1996.

[2] Ole Agesen. Space and Time-Efficient Hashing of Garbage Collected Objects. *Theory and Practice of Object Systems*, Vol. 5, No. 2, pp. 119–124, 1999.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An Efficient Meta-lock for Implementing Ubiquitous Synchronization. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 207–222, 1999.

[4] Jonathan Aldrich, Craig Chambers, Emin G. Sirer, and Susan Eggers. Static Analyses for Eliminating Unnecessary Synchronization from Java Programs. In *Proceedings of the 6th International Static Analysis Symposium (SAS '99)*, pp. 19–38, 1999.

[5] Thomas E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 1, pp. 6–16, 1990.

[6] Eric Armstrong. HotSpot: A New Breed of Virtual Machine, 1998. `http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html`.

[7] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[8] Calvin Austin. J2SE 5.0 in a Nutshell, 2004. `http://java.sun.com/developer/technicalArticles/releases/j2se15/`.

[9] David F. Bacon. Fast and Effective Optimization of Statically Typed Object-Oriented Languages. Ph.D. Dissertation UCB/CSD-98-1017, University of California, 1997.

[10] David F. Bacon and Stephen Fink. Method and Apparatus to Provide Concurrency Control over Objects without Atomic Operations on Non-Shared Objects. United States Patent, US 6,772,153 B1, 2004 (Filed Aug. 11, 2000).

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

[12] Sandra J. Baylor, Murthy V. Devarakonda, Stephen J. Fink, Eugene Gluzberg, Michael H. Kalantar, Prakash Muttineni, Eric Barsness, Rajiv Arora, Robert T. Dimpsey, and Steven J. Munroe. Java Server Benchmarks. *IBM Systems Journal*, Vol. 39, No. 1, pp. 57–81, 2000.

[13] BEA Systems. BEA JRockit. `http://dev2dev.bea.com/jrockit/`.

[14] BEA Systems. BEA WebLogic JRockit: Java for the Enterprise, Technical White Paper. `http://www.bea.com/content/news_events/white_papers/BEA_JRockit_wp.pdf`.

[15] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast Mutual Exclusion for Uniprocessors. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pp. 223–233, 1992.

[16] Bruno Blanchet. Escape Analysis for Object-Oriented Languages: Application to Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 20–34, 1999.

[17] Jeff Bogda and Urs Hölzle. Removing Unnecessary Synchronization in Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 35–46, 1999.

[18] Gregory Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, David Hardin, and Mark Turnbull. *The Real-Time Specification for Java*. Addison Wesley, 2000.

[19] Chandrasekhar Boyapati and Martin Rinard. A Parameterized Type System for Race-Free Java Programs. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pp. 56–69, 2001.

[20] Per Brinch Hansen. *Operating System Principles*. Prentice Hall, 1983.

[21] Peter A. Buhr, Michel Fortier, and Michael H. Coffin. Monitor Classification. *ACM Computing Surveys*, Vol. 27, No. 1, pp. 63–107, 1995.

[22] Builder UK. Sun Unveils First Real-Time Java Implementation, 2005. `http://uk.builder.com/programming/java/0,39026606,39254201,00.htm`.

[23] Jon Byous. Java Technology: The Early Years. `http://java.sun.com/features/1998/05/birthday.html`.

[24] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack Allocation and Synchronization Optimizations for Java Using Escape Analysis. *ACM Transactions on Programming Languages and Systems*, Vol. 25, No. 6, pp. 876–910, 2003.

[25] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape Analysis for Java. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 1–19, 1999.

[26] CNET News.com. IBM Pulls Away in App Server Race, 2003. `http://news.com.com/2100-1012_3-1000046.html`.

[27] David E. Culler, Jaswinder P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[28] David Dice. Implementing Fast Java Monitors with Relaxed-Locks. In *Proceedings of USENIX Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 79–90, 2001.

[29] Edsger W. Dijkstra. Solution of a Problem in Concurrent Programming and Control. *Communications of the ACM*, Vol. 8, No. 9, p. 569, 1965.

[30] Edsger W. Dijkstra. Co-operating Sequential Processes. In Francois Genuys, editor, *Programming Languages*, pp. 43–112. Academic Press, New York, 1968.

[31] Robert T. Dimpsey, Rajiv Arora, and Kean Kuiper. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, Vol. 39, No. 1, pp. 151–174, 2000.

[32] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic Metrics for Java. In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pp. 149–168, 2003.

[33] Robert Eckstein, David Collier-Brown, and Peter Kelly. *Using Samba*. O'Reilly, 1999.

[34] Eclipse.org. Eclipse.org Main Page. `http://www.eclipse.org/`.

[35] Cormac Flanagan and Stephen N. Freund. Type-Based Race Detection for Java. In *Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pp. 219–232, 2000.

[36] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, pp. 479–495, 2002.

[37] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*, pp. 212–223, 1998.

[38] Etienne M. Gagnon and Laurie J. Hendren. SableVM: A Research Framework for the Efficient Execution of Java Bytecode. In *Proceedings of USENIX Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 27–39, 2001.

[39] Brian Goetz. Java Theory and Practice: Going Atomic. `http://www.ibm.com/developerworks/java/library/j-jtp11234/`.

[40] Benedict A. Gomes, Lars Bak, and David P. Stoutamire. Method and Apparatus for Speculatively Locking Objects in an Object-Based System. United States Patent, US 6,487,652 B1, 2002.

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

[42] Michael Greenwald and David Cheriton. The Synergy Between Non-blocking Synchronization and Operating System Structure. In *Proceedings of the 2nd USENIX Symposium on Operating System Design and Implementation (OSDI '96)*, pp. 123–136, 1996.

[43] Weiming Gu, Nancy A. Burns, Michael T. Collins, and Wai Yee Peter Wong. The Evolution of a High-Performing Java Virtual Machine. *IBM Systems Journal*, Vol. 39, No. 1, pp. 135–150, 2000.

[44] Tim Harris, Keir Fraser, Ian Pratt, and Chris Purcell. Practical Lock-Free Data Structures, 2005. `http://www.cl.cam.ac.uk/Research/SRG/netos/lock-free/`.

[45] Allan Heydon and Marc Najork. Performance Limitations of the Java Core Libraries. In *Proceedings of the ACM 1999 Java Grande Conference*, pp. 363–373, 1999.

[46] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, Vol. 17, No. 10, pp. 549–557, 1974.

[47] David Hovemeyer, William Pugh, and Jaime Spacco. Atomic Instructions in Java. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*, pp. 133–154, 2002.

[48] Richard L. Hudson, J. Elliot B. Moss, Sreenivas Subramoney, and Weldon Washburn. Cycles to Recycle: Garbage Collection on the IA-64. In *Proceedings of the 2nd ACM International Symposium on Memory Management (ISMM '00)*, pp. 101–110, 2000.

[49] IBM Corporation. *PowerPC Virtual Environment Architecture (Book II).* `http://www.ibm.com/developerworks/eserver/library/es-archguide-v2.html`.

[50] IBM Corporation. WebSphere Application Server. `http://www.ibm.com/software/webservers/appserv/was/`.

[51] IBM Corporation. z/OS. `http://www.ibm.com/servers/eserver/zseries/zos/`.

[52] IBM Corporation. IBM Workplace Client Technology: Delivering the Rich Client Experience, 2004. `http://www.lotus.com/products/product5.nsf/wdocs/7231aae345edcabf85256e890071b1ad`.

[53] IBM developerWorks. IBM developer kits. `http://www.ibm.com/developerworks/java/jdk/`.

[54] IBM Tokyo Research Laboratory. Aglets. `http://www.trl.ibm.com/aglets/`.

[55] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manuals.* `http://developer.intel.com/design/pentium4/manuals/index_new.htm`.

[56] Intel Corporation. *Intel Itanium Architecture Software Developer's Manuals.* `http://developer.intel.com/design/itanium/manuals/iiasdmanual.htm`.

[57] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proceedings of the ACM 1999 Java Grande Conference*, pp. 119–128, 1999.

[58] Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler. In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pp. 187–204, 2003.

[59] Java Community Process. JSR 1: Real-Time Specification for Java. `http://jcp.org/jsr/detail/1.jsp`.

[60] Java Community Process. JSR 133: Java Memory Model and Thread Specification Revision. `http://jcp.org/jsr/detail/133.jsp`.

[61] Java Community Process. JSR 166: Concurrency Utilities. `http://jcp.org/jsr/detail/166.jsp`.

[62] Theodore Johnson and Krishna Harathi. Interruptible Critical Sections. Technical Report TR94007, University of Florida, 1994.

[63] Kaffe.org. Kaffe.org. `http://www.kaffe.org/`.

[64] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical Studies of Competitive Spinning for A Shared-Memory Multiprocessor. In *Proceedings of the 13th Annual ACM Symposium on Operating Systems Principles (SOSP '91)*, pp. 41–55, 1991.

[65] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations. In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pp. 130–141, 2002.

[66] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Accelerating Java Locks by Utilizing Their Thread Locality. *IPSJ Transactions on Programming*, Vol. 44, No. SIG 15 (PRO 19), pp. 13–23, 2003. in Japanese.

[67] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Asymmetric Spin Lock and Its Application to Java. *IPSJ Transactions on Programming*, Vol. 45, No. SIG 5 (PRO 21), pp. 62–76, 2004. in Japanese.

[68] Andreas Krall and Mark Probst. Monitors and Exceptions: How to Implement Java Efficiently. In *Proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, pp. 15–24, 1998.

[69] H. T. Kung and John. T. Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database System*, Vol. 6, No. 2, pp. 213–226, 1981.

[70] Leslie Lamport. A Fast Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 1–11, 1987.

[71] Butler W. Lampson and David D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, Vol. 23, No. 2, pp. 105–117, 1980.

[72] Doug Lea and JMM mailing list. The JSR-133 Cookbook for Compiler Writers. `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[73] Paul Leach and Dan Perry. CIFS: A Common Internet File System, 1996. `http://www.microsoft.com/mind/1196/cifs.asp`.

[74] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison Wesley, 1999.

[75] Beng-Hong Lim and Anant Agarwal. Waiting Algorithms for Synchronization in Large-Scale Multiprocessors. *ACM Transactions on Computer Systems*, Vol. 11, No. 3, pp. 253–294, 1993.

[76] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[77] Lotus Development Corporation. eSuite DevSite, 1999. `http://esuite.lotus.com/`.

[78] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pp. 378–391, 2005.

[79] Cathy May, Ed Shilha, Rick Simpson, and Hank Warren. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[80] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, Vol. 9, No. 1, pp. 21–65, 1991.

[81] James G. Mitchell, William Maybury, and Richard Sweet. Mesa Language Manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

[82] Motorola Inc. *M68040 User's Manual*. `http://e-www.motorola.com/files/32bit/doc/ref_manual/MC68040UM.pdf`.

[83] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles Consel. Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code. In *Proceedings of the 3rd USENIX Conference on Object-Oriented Technologies and Systems (COOTS '97)*, pp. 1–20, 1997.

[84] Tatsuo Nakajima, Takuro Kitayama, Hiroshi Arakawa, and Hideyuki Tokuda. Integrated Management of Priority Inversion in Real-Time Mach. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS '93)*, pp. 120–130, 1993.

[85] Bradford Nichols, Dick Buttlar, and Jacqueline P. Farrell. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*. O'Reilly, 1996.

[86] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. TO-Lock: Removing Lock Overhead Using the Owners' Temporal Locality. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT '04)*, pp. 255–266, 2004.

[87] Tamiya Onodera. A Simple and Space-Efficient Monitor Optimization for Java. Research Report RT0259, IBM, 1998.

[88] Tamiya Onodera and Kiyokuni Kawachiya. A Study of Locking Objects with Bimodal Fields. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 223–237, 1999.

[89] Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. Lock Reservation for Java Reconsidered. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, pp. 560–584, 2004.

[90] John K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS '82)*, pp. 22–30, 1982.

[91] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot Server Compiler. In *Proceedings of USENIX Java Virtual Machine Research and Technology Symposium (JVM '01)*, pp. 1–12, 2001.

[92] Young G. Park and Benjamin Goldberg. Escape Analysis on Lists. In *Proceedings of ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)*, pp. 116–127, 1992.

[93] Gary L. Peterson. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, Vol. 12, No. 3, pp. 115–116, 1981.

[94] William Pugh. Fixing the Java Memory Model. In *Proceedings of the ACM 1999 Java Grande Conference*, pp. 89–98, 1999.

[95] Ravi Rajwar and James R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th International Symposium on Microarchitecture (MICRO-34)*, pp. 294–305, 2001.

[96] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pp. 208–218, 2000.

[97] Alexandru Salcianu and Martin Rinard. Pointer and Escape Analysis for Multithreaded Programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP '01)*, pp. 12–23, 2001.

[98] Kumar Shiv and Joakim Dahlstedt. The Performance Architecture of the BEA JRockit Virtual Machine for the Java Platform (JVM Machine). Presentation in JavaOne '04, 2004.
`http://developers.sun.com/learning/javaoneonline/2004/corej2se/TS-1904.pdf`.

[99] Standard Performance Evaluation Corporation. SPEC JBB2000.
`http://www.spec.org/osg/jbb2000/`.

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

[101] Mark Stoodley and Vijay Sundaresan. Automatically Reducing Repetitive Synchronization with a Just-in-Time Compiler for Java. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization (CGO '05)*, pp. 27–36, 2005.

[102] Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Evolution of a Java Just-In-Time Compiler for IA-32 Platforms. *IBM Journal of Research and Development*, Vol. 48, No. 5/6, pp. 767–795, 2004.

[103] Toshio Suganuma, Takeshi Ogasawara, Mikio Takeuchi, Toshiaki Yasue, Motohiro Kawahito, Kazuaki Ishizaki, Hideaki Komatsu, and Toshio Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, Vol. 39, No. 1, pp. 175–193, 2000.

[104] Sun Microsystems, Inc. HotJava Product Family.
`http://java.sun.com/products/hotjava/`.

[105] Sun Microsystems, Inc. J2SE 5.0.
http://java.sun.com/j2se/1.5.0/.

[106] Sun Microsystems, Inc. Java 2 Platform, Enterprise Edition (J2EE).
http://java.sun.com/j2ee/.

[107] Sun Microsystems, Inc. Java HotSpot Technology.
http://java.sun.com/products/hotspot/.

[108] Sun Microsystems, Inc. Java Technology and Web Services.
http://java.sun.com/webservices/.

[109] Sun Microsystems, Inc. The Java HotSpot Virtual Machine, V1.4.1 White Paper.
http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html.

[110] Alexander Thomasian. Concurrency Control: Methods, Performance, and Analysis. *ACM Computing Surveys*, Vol. 30, No. 1, pp. 70–119, 1998.

[111] Hideyuki Tokuda, Clifford W. Mercer, Yutaka Ishikawa, and Thomas E. Marchok. Priority Inversions in Real-Time Communication. In *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS '89)*, pp. 348–359, 1989.

[112] Hideyuki Tokuda and Tatsuo Nakajima. Evaluation of Real-Time Synchronization in Real-Time Mach. In *Proceedings of USENIX 2nd Mach Symposium*, pp. 213–221, 1991.

[113] Transaction Processing Performance Council. TPC-C Benchmark Specification.
http://www.tpc.org/tpcc/.

[114] Volano LLC. Volano Benchmarks.
http://www.volano.com/benchmarks.html.

[115] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Transactional Monitors for Concurrent Objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, pp. 519–542, 2004.

[116] John Whaley. Partial Method Compilation using Dynamic Profile Information. In *Proceedings of the 16th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '01)*, pp. 166–179, 2001.

[117] John Whaley. Joeq: A Virtual Machine and Compiler Infrastructure. In *Proceedings of ACM SIGPLAN 2003 Workshop on Interpreters, Virtual Machines and Emulators (IVME '03)*, pp. 58–66, 2003.

[118] John Whaley and Martin Rinard. Compositional Pointer and Escape Analysis for Java Programs. In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 187–206, 1999.

[119] Wikipedia. Ekiden.
`http://en.wikipedia.org/wiki/Ekiden`.

[120] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pp. 24–36, 1995.

[121] Gaku Yamamoto and Yuhichi Nakamura. Architecture and Performance Evaluation of a Massive Multi-Agent System. In *Proceedings of the 3rd Annual Conference on Autonomous Agents*, pp. 319–325, 1999.

[122] Byung-Sun Yang, Junpyo Lee, Jinpyo Park, Soo-Mook Moon, Kemal Ebcioğlu, and Erik Altman. Lightweight Monitor for Java VM. *ACM SIGARCH Computer Architecture News*, Vol. 27, No. 1, pp. 35–38, 1999.

[123] Frank Yellin and Tim Lindholm. Java Runtime Internals. Presentation in JavaOne '96, 1996.
`http://java.sun.com/javaone/javaone96/pres/Runtime.pdf`.

List of Publications

原著論文

A1. 河内谷 清久仁, 古関 聰, 小野寺 民也. “非対称なスピンロックの提案とその Java への応用,” 情報処理学会論文誌：プログラミング, Vol. 45, No. SIG 5 (PRO 21), pp. 62–76 (2004/05).

Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. “Asymmetric Spin Lock and its Application to Java,” In *IPSJ Transactions on Programming*, Vol. 45, No. SIG 5 (PRO 21), pp. 62–76, in Japanese (2004/05).

A2. 河内谷 清久仁, 古関 聰, 小野寺 民也. “スレッド局所性を利用した Java ロックの高速化,” 情報処理学会論文誌：プログラミング, Vol. 44, No. SIG 15 (PRO 19), pp. 13–23 (2003/11).
※ 情報処理学会平成 16 年度論文賞受賞.

Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. “Accelerating Java Locks by Utilizing Their Thread Locality,” In *IPSJ Transactions on Programming*, Vol. 44, No. SIG 15 (PRO 19), pp. 13–23, in Japanese (2003/11).
∗ Received IPSJ Best Paper Award.

A3. 河内谷 清久仁, 石川 浩. “携帯情報ブラウジングのための入力デバイス「NaviPoint」,” 情報処理学会論文誌, Vol. 39, No. 5, pp. 1431–1439 (1998/05).

Kiyokuni Kawachiya and Hiroshi Ishikawa. “NaviPoint: An Input Device for Mobile Information Browsing,” In *Journal of IPSJ*, Vol. 39, No. 5, pp. 1431–1439, in Japanese (1998/05).

A4. 河内谷 清久仁. “マルチメディア処理の動的 QoS 制御のためのフレームワーク,” 電子情報通信学会和文論文誌, Vol. J80-B-I, No. 6, pp. 465–471 (1997/06).

Kiyokuni Kawachiya. “A Framework for Dynamic QoS Control of Multimedia Processing,” In *Transactions of IEICE*, Vol. J80-B-I, No. 6, pp. 465–471, in Japanese (1997/06).

国際学会発表

B1. Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. "Lock Reservation: Java Locks Can Mostly Do Without Atomic Operations," In *Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pp. 130–141 (2002/11/07).
(和題) 予約ロック：Java の同期機構における不可分命令の除去.

B2. Kiyokuni Kawachiya and Hiroshi Ishikawa. "NaviPoint: An Input Device for Mobile Information Browsing," In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems (CHI '98)*, pp. 1–8 (1998/04/21).
(和題) NaviPoint：モバイルブラウジング用入力デバイス.

B3. Kiyokuni Kawachiya, Nagatsugu Yamanouchi, and Takayuki Kushida. "VideoProxy: A Media and Protocol Converter for Internet Video," In *Proceedings of the INTERWORKING '96: Global Information Infrastructure (GII) Evolution: Interworking Issues*, Edited by S. Rao, H. Uose, and J. C. Luetchford, IOS Press, pp. 541–550 (1996/10/03).
(和題) VideoProxy：インターネットビデオのためのメディア／プロトコル変換機構.

B4. Kiyokuni Kawachiya and Hideyuki Tokuda. "Dynamic QOS Control Based on the QOS-Ticket Model," In *Proceedings of the 3rd IEEE International Conference on Multimedia Computing and Systems (ICMCS '96)*, pp. 78–85 (1996/06/19).
(和題) QOS チケットモデルにもとづく動的 QOS 制御手法.

B5. Kiyokuni Kawachiya and Hideyuki Tokuda. "Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing," In *Proceedings of the 6th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '96)*, pp. 149–156 (1996/04/26).
(和題) Q スレッド：連続メディア処理の動的 QOS 制御のための新しい実行モデル.

B6. Kiyokuni Kawachiya and Hideyuki Tokuda. "QOS-Ticket: A New Resource-Management Mechanism for Dynamic QOS Control of Multimedia," In *Proceedings of the Multimedia Japan '96*, pp. 14–21 (1996/03/18).
(和題) QOS チケット：マルチメディアの動的 QOS 制御のための新しい資源管理機構.

B7. Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, and Hideyuki Tokuda. "Evaluation of QOS-Control Servers on Real-Time Mach," In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV '95)*, pp. 123–126 (1995/04/19).
Also in *Lecture Notes in Computer Science, LNCS 1018 (NOSSDAV '95)*, Springer-Verlag GmbH, pp. 117–120 (1995).

(和題) Real-Time Mach 上での QOS 制御サーバーの評価.

B8. Kiyokuni Kawachiya, Hidehiko Wada, Shigeto Mochida, Masanobu Ogata, and Hideyuki Tokuda. "Extending Real-Time Mach for Continuous Media Applications," In *Collected Abstracts from the 4th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '93)*, pp. 55–58 (1993/11/03).

(和題) 連続メディア処理のための Real-Time Mach の拡張.

国内学会発表

C1. 河内谷 清久仁, 石川 浩. "マルチモード入力デバイス「NaviPoint」によるコンテクスト依存操作インタフェースの検討," 情報処理学会第 56 回全国大会論文集, 1B-4, pp. 4-7–4-8 (1998/03/17).

Kiyokuni Kawachiya and Hiroshi Ishikawa. "Context-Dependent Operation Interface Using NaviPoint," In *Proceedings of the 56th Annual Convention IPS Japan*, 1B-4, pp. 4-7–4-8, in Japanese (1998/03/17).

C2. 河内谷 清久仁, 森山 孝男. "PowerPC を用いたハードウェアシステム用の GDB サーバ," 情報処理学会第 55 回全国大会論文集, 6Z-4, pp. 1-234–1-235 (1997/09/26).

Kiyokuni Kawachiya and Takao Moriyama. "A GDB Server for PowerPC-Based Hardware," In *Proceedings of the 55th Annual Convention IPS Japan*, 6Z-4, pp. 1-234–1-235, in Japanese (1997/09/26).

C3. 河内谷 清久仁, 徳田 英幸. "MKng プロジェクトにおけるマルチメディア技術: 動的 QOS 制御のための資源交渉手法の提案," 情報処理学会第 55 回全国大会論文集, 2Z-4, pp. 1-169–1-170 (1997/09/24).

Kiyokuni Kawachiya and Hideyuki Tokuda. "Multimedia Technology in the MKng Project: A Resource-Negotiation Method for Dynamic QOS Control," In *Proceedings of the 55th Annual Convention IPS Japan*, 2Z-4, pp. 1-169–1-170, in Japanese (1997/09/24).

C4. 河内谷 清久仁, 石川 浩. "ハイパーメディア・ブラウジングに適した新しい入力デバイス," 情報処理学会研究報告, Vol. 97, No. 43 (97-HI-72), pp. 55–60 (1997/05/16).

Kiyokuni Kawachiya and H. Ishikawa. "A Useful Input Device for Hypermedia Browsing," In *IPSJ SIG Notes*, Vol. 97, No. 43 (97-HI-72), pp. 55–60, in Japanese (1997/05/16).

C5. 河内谷 清久仁, 石川 浩. "超小型機器の「指一本操作」のための入力機構," 情報処理学会第 54 回全国大会論文集, 4R-3, pp. 4-111–4-112 (1997/03/13).

Kiyokuni Kawachiya and Hiroshi Ishikawa. “A Single-Finger Controller for Mobile Devices,” In *Proceedings of the 54th Annual Convention IPS Japan*, 4R-3, pp. 4-111–4-112, in Japanese (1997/03/13).

C6. 河内谷 清久仁, 椎尾 一郎. “モービル情報キャッシュのためのフレームワーク,” 情報処理学会第 53 回全国大会論文集, 1B-9, pp. 1-17–1-18 (1996/09/04).
Kiyokuni Kawachiya and Itiro Siio. “A Framework for Mobile Information Cache,” In *Proceedings of the 53rd Annual Convention IPS Japan*, 1B-9, pp. 1-17–1-18, in Japanese (1996/09/04).

C7. 河内谷 清久仁, 徳田 英幸. “MKng プロジェクトにおける動的 QOS 制御サポート,” 情報処理学会第 53 回全国大会論文集, 5B-8, pp. 1-47–1-48 (1996/09/06).
Kiyokuni Kawachiya and Hideyuki Tokuda. “Dynamic QOS-Control Support in the MKng Project,” In *Proceedings of the 53rd Annual Convention IPS Japan*, 5B-8, pp. 1-47–1-48, in Japanese (1996/09/06).

C8. 河内谷 清久仁, 徳田 英幸. “Q スレッド: 連続メディアの動的 QOS 制御に適したプロセス実行モデル,” 情報処理学会第 52 回全国大会論文集, 2F-2, pp. 3-215–3-216 (1996/03/06).
Kiyokuni Kawachiya and Hideyuki Tokuda. “Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing,” In *Proceedings of the 52nd Annual Convention IPS Japan*, 2F-2, pp. 3-215–3-216, in Japanese (1996/03/06).

C9. 河内谷 清久仁, 根岸 康, 田胡 和哉. “VOD 向けのネットワーク API の提案,” 情報処理学会第 52 回全国大会論文集, 6Bb-5, pp. 1-271–1-272 (1996/03/08).
Kiyokuni Kawachiya, Yasushi Negishi, and Kazuya Tago. “A New Network API for VOD,” In *Proceedings of the 52nd Annual Convention IPS Japan*, 6Bb-5, pp. 1-271–1-272, in Japanese (1996/03/08).

C10. 河内谷 清久仁, 徳田 英幸. “「QOS チケット」モデルに基づく連続メディア処理の動的 QOS 制御,” 情報処理学会第 7 回コンピュータシステム・シンポジウム論文集, pp. 141–148 (1995/11/08).
Kiyokuni Kawachiya and Hideyuki Tokuda. “Dynamic QOS Control of Continuous-Media Processing based on the “QOS-Ticket” Model,” In *Proceedings of the 7th IPSJ Computer System Symposium*, pp. 141–148, in Japanese (1995/11/08).

C11. 河内谷 清久仁, 山内 長承, 椎尾 一郎, 串田 高幸, 川副 博. “ATM を使った VOD システムの試作,” 情報処理学会第 51 回全国大会論文集, 6F-7, pp. 1-227–1-228 (1995/09/22).
Kiyokuni Kawachiya, Nagatsugu Yamanouchi, Itiro Siio, Takayuki Kushida, and Hiroshi Kawazoe. “An Experimental VOD System over ATM Network,” In *Proceedings of the 51st Annual Convention IPS Japan*, 6F-7, pp. 1-227–1-228, in Japanese (1995/09/22).

C12. 河内谷 清久仁, 緒方 正暢, 西尾 信彦, 徳田 英幸. “連続メディアの QOS 制御のための「QOS チケット」モデル,” 情報処理学会第 50 回全国大会論文集, 1N-6, pp. 3-153–3-154 (1995/03/15).

Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, and Hideyuki Tokuda. “A “QOS-Ticket” Model for QOS Control of Continuous Media Processing,” In *Proceedings of the 50th Annual Convention IPS Japan*, 1N-6, pp. 3-153–3-154, in Japanese (1995/03/15).

C13. 河内谷 清久仁, 緒方 正暢, 西尾 信彦, 徳田 英幸. “連続メディア処理の QOS 制御のための OS サポート,” 情報処理学会第 6 回コンピュータシステム・シンポジウム論文集, pp. 119–126 (1994/11/11).

Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, and Hideyuki Tokuda. “OS Support for QOS Control of Continuous Media Processing,” In *Proceedings of the 6th IPSJ Computer System Symposium*, pp. 119–126, in Japanese (1994/11/11).

C14. 河内谷 清久仁, 緒方 正暢. “Real-Time Mach 用のオーディオドライバ,” 情報処理学会第 49 回全国大会論文集, 7R-3, pp. 3-317–3-318 (1994/09/30).

Kiyokuni Kawachiya and Masanobu Ogata. “An Audio Driver for Real-Time Mach,” In *Proceedings of the 49th Annual Convention IPS Japan*, 7R-3, pp. 3-317–3-318, in Japanese (1994/09/30).

C15. 河内谷 清久仁, 緒方 正暢, 徳田 英幸. “連続メディア処理のためのリアルタイムスレッドモデルの拡張,” 情報処理学会第 48 回全国大会論文集, 1H-1, pp. 4-17–4-18 (1994/03/23).
※ 情報処理学会第 48 回全国大会奨励賞受賞.

Kiyokuni Kawachiya, Masanobu Ogata, and Hideyuki Tokuda. “Extending Real-Time Thread Model for Continuous Media Processing,” In *Proceedings of the 48th Annual Convention IPS Japan*, 1H-1, pp. 4-17–4-18, in Japanese (1994/03/23).
∗ Received IPSJ Convention Award.

C16. 河内谷 清久仁, 緒方 正暢, 徳田 英幸. “Real-Time Mach 上での QOS 制御サーバの実験,” 情報処理学会第 47 回全国大会論文集, 4V-3, pp. 2-355–2-356 (1993/10/07).

Kiyokuni Kawachiya, Masanobu Ogata, and Hideyuki Tokuda. “A QOS-Control Server on Real-Time Mach,” In *Proceedings of the 47th Annual Convention IPS Japan*, 4V-3, pp. 2-355–2-356, in Japanese (1993/10/07).

C17. 河内谷 清久仁, 白鳥 敏幸, 山崎 秘砂. “MP UNIX におけるデバイスアクセス手法の比較,” 情報処理学会研究報告 92-OS-56 (SWoPP ’92), pp. 89–96 (1992/08/19).

Kiyokuni Kawachiya, Toshiyuki Shiratori, and Hisa Yamasaki. “Comparison of Device Access Methods in MP UNIX,” In *IPSJ SIG Notes 92-OS-56 (SWoPP ’92)*, pp. 89–96, in Japanese (1992/08/19).

C18. 河内谷 清久仁, 森山 孝男, 山崎 秘砂, 白鳥 敏幸. “TOP-1 オペレーティングシステムの構造,” 情報処理学会研究報告 90-OS-48, pp. 25–34 (1990/09/07).

Kiyokuni Kawachiya, Takao Moriyama, Hisa Yamasaki, and Toshiyuki Shiratori. “Structure of the TOP-1 Operating System,” In *IPSJ SIG Notes 90-OS-48*, pp. 25–34, in Japanese (1990/09/07).

C19. 河内谷 清久仁, 山崎 秘砂, 白鳥 敏幸, 森山 孝男, 穂積 元一. “TOP-1 オペレーティングシステム (3) メモリ管理,” 情報処理学会第 39 回全国大会論文集, 3P-6, pp. 1203–1204 (1989/10/17).

Kiyokuni Kawachiya, Hisa Yamasaki, Toshiyuki Shiratori, Takao Moriyama, and Motokazu Hozumi. “TOP-1 Operating System (3) Memory Management,” In *Proceedings of the 39th Annual Convention IPS Japan*, 3P-6, pp. 1203–1204, in Japanese (1989/10/17).

C20. 河内谷 清久仁, 前川 守, 太田 昌孝, 濱野 純. “稠密処理システム GALAXY のアドレススペースの考え方,” 情報処理学会第 33 回全国大会論文集, 3U-8, pp. 1053–1054 (1986/10/02).

Kiyokuni Kawachiya, Mamoru Maekawa, Masataka Ohta, and Jun Hamano. “Address Space of GALAXY Holonic Processing System,” In *Proceedings of the 33rd Annual Convention IPS Japan*, 3U-8, pp. 1053–1054, in Japanese (1986/10/02).

C21. 河内谷 清久仁, 前川 守, 太田 昌孝, 荒野 高志, 野口 佳一. “マルチメディアマシンのメディア構造,” 情報処理学会第 31 回全国大会論文集, 8P-2, pp. 1439–1440 (1985/09/12).

Kiyokuni Kawachiya, Mamoru Maekawa, Masataka Ohta, Takashi Arano, and Yoshikazu Noguchi. “Media Structure of Multimedia Machine,” In *Proceedings of the 31st Annual Convention IPS Japan*, 8P-2, pp. 1439–1440, in Japanese (1985/09/12).

その他（リサーチレポート，解説記事など）

D1. Kiyokuni Kawachiya and Takao Moriyama. “A Symbolic Debugger for PowerPC-Based Hardware, Using the Engineering Support Processor (ESP),” Research Report RT0212, IBM (1997/08/27).

(和題) PowerPC システム用の，エンジニアリングサポートプロセッサ（ESP）を用いたシンボリックデバッガ.

D2. Kiyokuni Kawachiya and Hideyuki Tokuda. “A Negotiation-Based Resource Management Framework for Dynamic QOS Control,” Research Report RT0192, IBM (1997/03/18).

(和題) 動的 QOS 制御のための，交渉型資源管理フレームワーク.

D3. 河内谷 清久仁. “[トピックス] NOSSDAV ’95 報告,” コンピュータソフトウェア, Vol. 12, No. 5, pp. 105–114 (1995/09).
Kiyokuni Kawachiya. “[Topics] NOSSDAV ’95 Report,” In *JSSST Computer Software*, Vol. 12, No. 5, pp. 105–114, in Japanese (1995/09).

D4. 河内谷 清久仁. “[解説] 並列計算機TOP-1 の OS,” 情報処理, Vol. 36, No. 8, pp. 734–738 (1995/08).
Kiyokuni Kawachiya. “[Explanation] Operating Systems on the TOP-1 Multiprocessor,” In *Journal of IPS Japan*, Vol. 36, No. 8, pp. 734–738, in Japanese (1995/08).

D5. Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, and Hideyuki Tokuda. “QOS Control of Continuous Media: Architecture and System Support,” Research Report RT0108, IBM (1995/04/11).
(和題) 連続メディアの QOS 制御：アーキテクチャとシステムサポート.

D6. 河内谷 清久仁. “[文献紹介] スケジューラ・アクティベーション：ユーザレベルで並列性を管理するための効果的なカーネルサポート,” 情報処理, Vol. 33, No. 11, pp. 1371–1372 (1992/11).
Kiyokuni Kawachiya. “[Book Guide] Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism,” In *Journal of IPS Japan*, Vol. 33, No. 11, pp. 1371–1372, in Japanese (1992/11).

D7. 河内谷 清久仁. “[図書紹介] UNIX 詳説 — 改訂 2 版,” 電子情報通信学会誌, Vol. 73, No. 9, p. 1019 (1990/09).
Kiyokuni Kawachiya. “[Book Review] UNIX Expatiation — 2nd. edition,” In *Journal of IEICE*, Vol. 73, No. 9, p. 1019, in Japanese (1990/09).

D8. Kiyokuni Kawachiya. “Transparent Object Management in the GALAXY Holonic Processing System,” Master’s Thesis, Department of Information Science, Faculty of Science, University of Tokyo (1987/02).
(和題) 稠密処理システム Galaxy の透過型オブジェクト管理.

D9. 河内谷 清久仁. “[翻訳] 仮想プロトコル・マシン,” UNIX 原典, AT&T ベル研究所 編集, 石田 晴久 監修, 長谷部 紀元・清水 謙多郎 翻訳. *パーソナルメディア*, pp. 323–341 (1986/11).
Kiyokuni Kawachiya. “[Japanese translation] The Virtual Protocol Machine (by M. J. Fitton, et al.),” In Japanese translation of *BSTJ: The UNIX System*, Personal Media, pp. 323–341, in Japanese (1986/11).

D10. 河内谷 清久仁, 濱野 純. “ワークステーションのオペレーティングシステム,” コンピュートロール, No. 14, 前川 守 責任編集, コロナ社, pp. 7–13 (1986/04).
Kiyokuni Kawachiya and Jun Hamano. “Operating Systems of Workstations,” In *Computrol*, No. 14, Edited by Mamoru Maekawa, pp. 7–13, in Japanese (1986/04).

共著文献

E1. 緒方 一則, 小野寺 民也, 河内谷 清久仁, 小松 秀昭, 中谷 登志男. “再現コンパイル手法を用いた Java JIT コンパイラの問題判別,” 情報処理学会論文誌：プログラミング, Vol. 46, No. SIG 14 (PRO 27), pp. 1–11 (2005/10).

Kazunori Ogata, Tamiya Onodera, Kiyokuni Kawachiya, Hideaki Komatsu, and Toshio Nakatani. “Problem Determination for a Java JIT Compiler Using Replay Compilation,” In *IPSJ Transactions on Programming*, Vol. 46, No. SIG 14 (PRO 27), pp. 1–11, in Japanese (2005/10).

E2. Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. “Evolution of a Java Just-In-Time Compiler on IA-32 Platforms,” In *IBM Journal of Research and Development*, Vol. 48, No. 5/6, pp. 767–795 (2004/11).
(和題) IA-32 プラットフォーム上の Java Just-In-Time コンパイラの進化.

E3. Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. “Lock Reservation for Java Reconsidered,” In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP '04)*, pp. 560–584 (2004/06/17).
Also in *Lecture Notes in Computer Science, LNCS 3086 (ECOOP '04)*, Edited by Martin Odersky, Springer-Verlag GmbH, pp. 559–583 (2004).
(和題) Java の予約ロックに関する再考察.

E4. Kazuaki Ishizaki, Mikio Takeuchi, Kiyokuni Kawachiya, Toshio Suganuma, Osamu Gohda, Tatsushi Inagaki, Akira Koseki, Kazunori Ogata, Motohiro Kawahito, Toshiaki Yasue, Takeshi Ogasawara, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani, “Effectiveness of Cross-Platform Optimizations for a Java Just-In-Time Compiler,” In *Proceedings of the 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pp. 187–204 (2003/10/29).
(和題) Java Just-In-Time コンパイラのクロスプラットフォーム最適化の効果.

E5. Yasushi Negishi, Kiyokuni Kawachiya, Hiroki Murata, and Kazuya Tago. “Tuplink: A Meta-middleware System for Micro-clients,” In *Journal of IPS Japan*, Vol. 41, No. 10, pp. 2881–2894 (2000/10).
(和題) Tuplink：マイクロクライアント用のメタ・ミドルウェア.

E6. Tamiya Onodera and Kiyokuni Kawachiya. “A Study of Locking Objects with Bimodal Fields,” In *Proceedings of the 14th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '99)*, pp. 223–237 (1999/11/04).

(和題) 2モードフィールドを用いるオブジェクトロック手法の研究.

E7. 根岸 康, 村田 浩樹, 奥山 健一, 鎌田 國生, 河内谷 清久仁, 田胡 和哉. “携帯端末向けの通信機構とそのアプリケーション,” 日本ソフトウェア科学会第 1 回プログラミングおよび応用のシステムに関するワークショップ (SPA '98), 12 pages (1998/03/27).

Yasushi Negishi, Hiroki Murata, Kenichi Okuyama, Kunio Kamata, Kiyokuni Kawachiya, and Kazuya Tago. “A Communication System for Mobile Micro-Clients and its Applications,” In *Proceedings of the JSSST 1st Workshop on Systems for Programming and Applications (SPA '98)*, 12 pages, in Japanese (1998/03/27).

E8. Nagatsugu Yamanouchi, Kiyokuni Kawachiya, and Takayuki Kushida. “An Experimental Dynamic QoS Control System over Internet,” In *Proceedings of the INTERWORKING '96: Global Information Infrastructure (GII) Evolution: Interworking Issues*, Edited by S. Rao, H. Uose, and J. C. Luetchford, IOS Press, pp. 368–377 (1996/10/03).

(和題) インターネット越しの動的 QoS 制御システムの実験.

E9. 山内 長承, 串田 高幸, 河内谷 清久仁. “インターネット上のビデオ転送のための優先度・締切駆動プロトコル,” 情報処理学会次世代ボーダレスメディアの新展開と標準化シンポジウム論文集, pp. 1–8 (1996/06/10).

Nagatsugu Yamanouchi, Takayuki Kushida, and Kiyokuni Kawachiya. “PDTP: Priority and Deadline Driven Transport Protocol for Video Transfer over Internet,” In *Proceedings of the IPSJ Symposium for Next Generation Borderless Media*, pp. 1–8, in Japanese (1996/06/10).

E10. 串田 高幸, 河内谷 清久仁, 山内 長承. “インターネット上においてオーディオ及びビデオを配送するプロトコル,” 情報処理学会研究報告 96-DPS-76, pp. 157–162 (1996/05/17).

Takayuki Kushida, Kiyokuni Kawachiya, and Nagatsugu Yamanouchi. “Realtime Multimedia Protocol with Deadline Driven and Prioritized Data on Packet Networks,” In *IPSJ SIG Notes 96-DPS-76*, pp. 157–162, in Japanese (1996/05/17).

E11. Yasushi Negishi, Kiyokuni Kawachiya, and Kazuya Tago. “A Portable Communication System for Video-on-Demand Applications using the Existing Infrastructure,” In *Proceedings of IEEE INFOCOM '96 — The Conference on Computer Communications*, Vol. 1, pp. 18–26 (1996/03/26).

(和題) ビデオオンデマンドアプリケーション向けの，既存インフラストラクチャを用いたポータブルな通信システム.

E12. 串田 高幸, 河内谷 清久仁, 山内 長承. “実時間でデータ転送を行なうための方式について,” 情報処理学会第 52 回全国大会論文集, 1Bb-1, pp. 1-183–1-184 (1996/03/06).

Takayuki Kushida, Kiyokuni Kawachiya, and Nagatsugu Yamanouchi. "A Method for Real-Time Data Transfer," In *Proceedings of the 52nd Annual Convention IPS Japan*, 1Bb-1, pp. 1-183–1-184, in Japanese (1996/03/06).

E13. 山内 長承, 河内谷 清久仁, 串田 高幸. "インターネット上の動画転送を意識した動的 QoS の制御," 情報処理学会研究報告 96-DPS-75, pp. 19–24 (1996/02/29).

Nagatsugu Yamanouchi, Kiyokuni Kawachiya, and Takayuki Kushida. "Dynamic QoS Control for the Internet Environment," In *IPSJ SIG Notes 96-DPS-75*, pp. 19–24, in Japanese (1996/02/29).

E14. 田胡 和哉, 根岸 康, 河内谷 清久仁. "既存の通信プロトコルを利用して実現した VOD 向け通信機構," 情報処理学会第 7 回コンピュータシステム・シンポジウム論文集, pp. 149–156 (1995/11/08).

Kazuya Tago, Yasushi Negishi, and Kiyokuni Kawachiya. "User Level Implementation of Communication System for VOD Applications," In *Proceedings of the 7th IPSJ Computer System Symposium*, pp. 149–156, in Japanese (1995/11/08).

E15. 鈴木 則久, 河内谷 清久仁, 山内 長承. "[解説] 情報ハイウェイの基盤技術／2-2 コンピュータ技術," 電子情報通信学会誌, Vol. 78, No. 4, pp. 358–363 (1995/04).

Norihisa Suzuki, Kiyokuni Kawachiya, and Nagatsugu Yamanouchi. "[Explanation] About Information Highway / 2-2 Computer Technologies," In *Journal of IEICE*, Vol. 78, No. 4, pp. 358–363, in Japanese (1995/04).

E16. 堀切 和典, 多田 征司, 河内谷 清久仁, 西尾 信彦, 徳田 英幸, 斎藤 信男. "分散マルチメディア環境のためのセキュリティ機構," 情報処理学会第 50 回全国大会論文集, 1N-7, pp. 3-155–3-156 (1995/03/15).

Kazunori Horikiri, Seiji Tada, Kiyokuni Kawachiya, Nobuhiko Nishio, Hideyuki Tokuda, and Nobuo Saito. "A Security Mechanism for Distributed Multimedia Environment," In *Proceedings of the 50th Annual Convention IPS Japan*, 1N-7, pp. 3-155–3-156, in Japanese (1995/03/15).

E17. 緒方 正暢, 西尾 信彦, 河内谷 清久仁, 徳田 英幸. "可変プロセッサ速度に対応した実時間処理の考察," 情報処理学会第 50 回全国大会論文集, 3H-2, pp. 4-231–4-232 (1995/03/16).

Masanobu Ogata, Nobuhiko Nishio, Kiyokuni Kawachiya, and Hideyuki Tokuda. "Real-time Processing for Power Managed Computers," In *Proceedings of the 50th Annual Convention IPS Japan*, 3H-2, pp. 4-231–4-232, in Japanese (1995/03/16).

E18. 和田 英彦, 河内谷 清久仁, 緒方 正暢, 西尾 信彦, 追川 修一, 徳田 英幸. "分散マルチメディア統合環境とマイクロカーネル," 電気学会情報処理研究会資料 1P-94-45, pp. 79–88 (1994/12/20).

Hidehiko Wada, Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, Shuichi Oikawa, and Hideyuki Tokuda. "Distributed Multimedia Computing Environment and Micro-kernel Architecture," In *Proceedings of the IEE-J SIG Information Processing*, 1P-94-45, pp. 79–88, in Japanese (1994/12/20).

E19. 西尾 信彦, 徳田 英幸, 河内谷 清久仁. "Conductor/Performer モデルにおける連続メディア処理のためのクラスの実現," 情報処理学会第 6 回コンピュータシステム・シンポジウム論文集, pp. 127–134 (1994/11/11).

Nobuhiko Nishio, Hideyuki Tokuda, and Kiyokuni Kawachiya. "Classes for Continuous Media Processing in Conductor/Performer Model," In *Proceedings of the 6th IPSJ Computer System Symposium*, pp. 127–134, in Japanese (1994/11/11).

E20. 緒方 正暢, 河内谷 清久仁, 西尾信彦, 徳田英幸. "モービルコンピュータを使った分散マルチメディア環境の実現," 情報処理学会第 49 回全国大会論文集, 6S-9, pp. 3-345–3-346 (1994/09/30).

Masanobu Ogata, Kiyokuni Kawachiya, Nobuhiko Nishio, and Hideyuki Tokuda. "Implementation of Distributed Multimedia Environment with Mobile Computers," In *Proceedings of the 49th Annual Convention IPS Japan*, 6S-9, pp. 3-345–3-346, in Japanese (1994/09/30).

E21. 和田 英彦, 河内谷 清久仁, 緒方 正暢, 西尾 信彦, 追川 修一, 徳田 英幸. "Keio-MMP におけるマイクロカーネルアーキテクチャ," 情報処理学会第 49 回全国大会論文集, 7R-8, pp. 3-327–3-328 (1994/09/30).

Hidehiko Wada, Kiyokuni Kawachiya, Masanobu Ogata, Nobuhiko Nishio, Shuichi Oikawa, and Hideyuki Tokuda. "Micro-kernel Architecture in the Keio-MMP Project," In *Proceedings of the 49th Annual Convention IPS Japan*, 7R-8, pp. 3-327–3-328, in Japanese (1994/09/30).

E22. 持田 茂人, 緒方 正暢, 河内谷 清久仁, 西尾 信彦, 和田 英彦, 徳田 英幸. "Real-Time Mach 3.0 における I/O サーバの構成と評価," 情報処理学会第 48 回全国大会論文集, 1H-2, pp. 4-19–4-20 (1994/03/23).

Shigeto Mochida, Masanobu Ogata, Kiyokuni Kawachiya, Nobuhiko Nishio, Hidehiko Wada, and Hideyuki Tokuda. "Evaluation of I/O Server on Real-Time Mach 3.0," In *Proceedings of the 48th Annual Convention IPS Japan*, 1H-2, pp. 4-19–4-20, in Japanese (1994/03/23).

E23. 西尾 信彦, 追川 修一, 緒方 正暢, 尾上 裕子, 河内谷 清久仁, 塩野崎 敦, 南部 明, 持田 茂人, 和田 英彦, 徳田 英幸. "マイクロカーネルによる連続メディア処理の基盤技術," 情報処理学会第 5 回コンピュータシステム・シンポジウム論文集, pp. 17–24 (1993/10/20).
※ 情報処理学会平成 6 年度山下記念研究賞受賞.

Nobuhiko Nishio, Shuichi Oikawa, Masanobu Ogata, Yuko Onoe, Kiyokuni Kawachiya, Atsushi Shionozaki, Akira Nambu, Shigeto Mochida, Hidehiko Wada, and Hideyuki Tokuda. "Fundamental Technology for Continuous Media Processing Using Micro-Kernel," In *Proceedings of the 5th IPSJ Computer System Symposium*, pp. 17–24, in Japanese (1993/10/20).
∗ Received IPSJ Yamashita Award (SIG Research Award).

E24. 緒方 正暢, 河内谷 清久仁, 徳田 英幸. "動画のフレーム間相関を利用した動的なQOS 制御の実験," 情報処理学会第 47 回全国大会論文集, 4V-4, pp. 2-357–2-358 (1993/10/07).
Masanobu Ogata, Kiyokuni Kawachiya, and Hideyuki Tokuda. "Experiments with Dynamic QOS control Using Correlation of Sequential Frames," In *Proceedings of the 47th Annual Convention IPS Japan*, 4V-4, pp. 2-357–2-358, in Japanese (1993/10/07).

E25. 穂積 元一, 白鳥 敏幸, 吉永 秀志, 大澤 暁, 山崎 秘砂, 河内谷 清久仁, 森山 孝男. "TOP-1 オペレーティングシステム (1) 基本方針," 情報処理学会第 39 回全国大会論文集, 3P-4, pp. 1199–1200 (1989/10/17).
Motokazu Hozumi, Toshiyuki Shiratori, Hideshi Yoshinaga, Gyo Ohsawa, Hisa Yamasaki, Kiyokuni Kawachiya, and Takao Moriyama. "TOP-1 Operating System (1) Basic Direction," In *Proceedings of the 39th Annual Convention IPS Japan*, 3P-4, pp. 1199–1200, in Japanese (1989/10/17).

E26. 山崎 秘砂, 森山 孝男, 河内谷 清久仁, 白鳥 敏幸, 穂積 元一. "TOP-1 オペレーティングシステム (2) プロセス・スケジューリング," 情報処理学会第 39 回全国大会論文集, 3P-5, pp. 1201–1202 (1989/10/17).
Hisa Yamasaki, Takao Moriyama, Kiyokuni Kawachiya, Toshiyuki Shiratori, and Motokazu Hozumi. "TOP-1 Operating System (2) Process Scheduling," In *Proceedings of the 39th Annual Convention IPS Japan*, 3P-5, pp. 1201–1202, in Japanese (1989/10/17).

E27. 白鳥 敏幸, 森山 孝男, 河内谷 清久仁, 山崎 秘砂, 穂積 元一. "TOP-1 オペレーティングシステム (5) デバッグ環境," 情報処理学会第 39 回全国大会論文集, 3P-8, pp. 1207–1208 (1989/10/17).
Toshiyuki Shiratori, Takao Moriyama, Kiyokuni Kawachiya, Hisa Yamasaki, and Motokazu Hozumi. "TOP-1 Operating System (5) Debugging Environment," In *Proceedings of the 39th Annual Convention IPS Japan*, 3P-8, pp. 1207–1208, in Japanese (1989/10/17).

E28. Mamoru Maekawa, Kiyokuni Kawachiya, Masataka Ohta, Jun Hamano, and Kentaro Shimizu. "Object Management and Address Space of Galaxy Holonic Processing System," Technical Report TR 86-15, Department of Information Science, Faculty of Science, University of Tokyo (1986/10).

(和題) 稠密処理システム Galaxy のオブジェクト管理とアドレス空間.

E29. 太田 昌孝, 前川 守, 河内谷 清久仁, 濱野 純. "稠密処理システム GALAXY でのオブジェクト管理," 情報処理学会第 33 回全国大会論文集, 3U-7, pp. 1051–1052 (1986/10/02).

Masataka Ohta, Mamoru Maekawa, Kiyokuni Kawachiya, and Jun Hamano. "Object Management of GALAXY Holonic Processing System," In *Proceedings of the 33rd Annual Convention IPS Japan*, 3U-7, pp. 1051–1052, in Japanese (1986/10/02).

E30. 濱野 純, 前川 守, 太田 昌孝, 河内谷 清久仁. "稠密処理システム GALAXY でのネーミング," 情報処理学会第 33 回全国大会論文集, 3U-9, pp. 1055–1056 (1986/10/02).

Jun Hamano, Mamoru Maekawa, Masataka Ohta, and Kiyokuni Kawachiya. "Naming of GALAXY Holonic Processing System," In *Proceedings of the 33rd Annual Convention IPS Japan*, 3U-9, pp. 1055–1056, in Japanese (1986/10/02).

E31. Masataka Ohta, Mamoru Maekawa, Takashi Arano, Kiyokuni Kawachiya, and Yoshikazu Noguchi. "Multimedia Information Processing Based on a General Media Model," In *Information Processing 86: Proceedings of the IFIP 10th World Computer Congress (IFIP WCC '86)*, Edited by H.-J. Kugler, Elsvier Science Publishers B. V., pp. 957–962 (1986/09).

(和題) 汎用メディアモデルに基づくマルチメディア情報処理.

E32. 前川 守, 太田 昌孝, 荒野 高志, 河内谷 清久仁, 野口 佳一. "マルチメディア・ワークステーション PIE," 情報処理学会研究報告 85-IS-008, 8 pages (1985/12/17).

Mamoru Maekawa, Masataka Ohta, Takashi Arano, Kiyokuni Kawachiya, and Yoshikazu Noguchi. "Multimedia Workstation PIE," In *IPSJ SIG Notes 85-IS-008*, 8 pages, in Japanese (1985/12/17).

E33. Masataka Ohta, Takashi Arano, Kiyokuni Kawachiya, Yoshikazu Noguchi, and Mamoru Maekawa. "Multimedia Workstation – PIE," Technical Report TR 85-17, Department of Information Science, Faculty of Science, University of Tokyo (1985/11).

(和題) マルチメディア・ワークステーション – PIE.

E34. 前川 守, 太田 昌孝, 荒野 高志, 河内谷 清久仁, 野口 佳一. "マルチメディアマシン," 情報処理学会第 31 回全国大会論文集, 8P-1, pp. 1437–1438 (1985/09/12).

Mamoru Maekawa, Masataka Ohta, Takashi Arano, Kiyokuni Kawachiya, and Yoshikazu Noguchi. "Multimedia Machine," In *Proceedings of the 31st Annual Convention IPS Japan*, 8P-3, pp. 1437–1438, in Japanese (1985/09/12).

E35. 太田 昌孝, 前川 守, 荒野 高志, 河内谷 清久仁, 野口 佳一. "マルチメディアマシンの図形メディア処理," 情報処理学会第 31 回全国大会論文集, 8P-3, pp. 1441–1442 (1985/09/12).

Masataka Ohta, Mamoru Maekawa, Takashi Arano, Kiyokuni Kawachiya, and Yoshikazu Noguchi. "Graphic Media Processing in Multimedia Machine," In *Proceedings of the 31st Annual Convention IPS Japan*, 8P-3, pp. 1441–1442, in Japanese (1985/09/12).

E36. 荒野 高志, 前川 守, 太田 昌孝, 河内谷 清久仁, 野口 佳一. "マルチメディアマシンの音メディア処理," 情報処理学会第 31 回全国大会論文集, 8P-4, pp. 1443–1444 (1985/09/12).
Takashi Arano, Mamoru Maekawa, Masataka Ohta, Kiyokuni Kawachiya, and Yoshikazu Noguchi. "Audio Media Processing in Multimedia Machine," In *Proceedings of the 31st Annual Convention IPS Japan*, 8P-4, pp. 1443–1444, in Japanese (1985/09/12).