Diplomarbeit

Implementation of native threads and locks in the CACAO Java Virtual Machine

ausgeführt am

Institut für Computersprachen
Arbeitsbereich für Programmiersprachen und Übersetzerbau
der Technischen Universität Wien

unter der Anleitung von

Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Krall

durch

Stefan Ring
Bachackergasse 35
2380 Perchtoldsdorf

25. November 2008

Abstract

One of Java's most prominent features, the ability to create and synchronize multiple threads of execution, requires quite elaborate support from a Java Virtual Machine (JVM) implementation. This work describes the major steps in making the CACAO JVM provide full support of this fundamental feature. First, it discusses steps needed to implement Java's memory model. Next, the choice of an algorithm for Java monitors and possible future improvements are presented. A few experimental variants are evaluated across a selection of hardware architectures. Additionally, CACAO's type checking algorithm is replaced by a method better suited for use in parallel Java programs. The implications regarding performance as well as memory requirements are assessed.

Kurzfassung

Eines der wichtigsten Leistungsmerkmale von Java, die Fähigkeit zur Ausführung und Synchronisation paralleler Threads, benötigt recht umfangreiche Unterstützung durch die Implementierung einer Java Virtual Machine (JVM). Diese Arbeit beschreibt die wesentlichen Schritte, die benötigt werden, um die CACAO JVM um diese Fähigkeit zu erweitern. Zunächst werden die notwendigen Schritte für die Umsetzung des Java Memory Model erörtert. Weiters wird die Wahl eines Algorithmus für Java Monitore sowie mögliche zukünftige Erweiterungsmöglichkeiten präsentiert. Außerdem werden einige experimentelle Varianten auf verschiedenen Hardwarearchitekturen verglichen. Zusätzlich wird der von CACAO zur Typüberprüfung verwendete Algorithmus durch eine Methode ersetzt, die besser zur Verwendung in parallelen Javaprogrammen geeignet ist, sowie deren Auswirkungen auf Laufzeit und Speicheranforderungen überprüft.

Contents

Chapter 1

Introduction

1.1 Motivation

The Java platform was one of the first mainstream programming environments offering thread support in the standard run-time library as well as a well-defined memory model and synchronization primitives built directly into the core language.

This work describes the implementation of native thread support for CACAO. While setting up the parallel execution of threads is quite straight forward and well supported on all modern platforms via POSIX threads or Win32 threads, providing efficient locking primitives poses one of the most challenging implementation problems.

1.2 Java platform features

Java supports the notion of a monitor [15]—every object can act as a monitor to all its synchronized methods. In addition, synchronized code blocks can be defined which act like "anonymous" methods of a monitor object. Java monitors also have a *wait/notify* mechanism which can be used to implement all sorts of cooperation patterns among threads such as the well-known producer/consumer pattern.

Aiming for wide cross-platform portability, Java also offers a well-defined memory model. Although the C and C++ languages are available across an impressive range of platforms, they don't offer any guarantees regarding the temporal behavior of memory in parallel programs. This can lead to very different results on different underlying hardware architectures.

1.3 CACAO

The CACAO Java Virtual Machine (JVM) was first created as as fast Just-In-Time compiler for Java bytecode on the 64-bit Alpha architecture in 1996. In the beginning, there was just enough run-time support to run simple console applications. The JVM was limited in all regards, running only on a single CPU architecture on a single operating system (Linux), and including minimal green threads support. Since then it has been greatly expanded both in terms of feature completeness in order to move closer to the JVM Specification [22] as well as support of multiple CPU architectures and operating systems—CACAO has been running on Linux, Tru64, IRIX and Mac OS X on widely varying CPU architectures like Alpha, MIPS (32 and 64-bit), PowerPC (32-bit and 64-bit), ARM (32-bit), SPARC (64-bit) and S390 as well as Intel x86 and x86_64. Recently, it has become possible to use CACAO as the JVM inside Sun's OpenJDK.

Chapter 2

Fundamentals

2.1 Mutual Exclusion

The mutual exclusion problem is best described in [9]:

> The mutual exclusion problem arises in a domain wherein each participating process executes, in strict cyclic order, program regions labeled *remainder*, *acquire*, *critical section*, and *release*. A solution to the mutual exclusion problem consists of code for the `acquire()` and `release()` operations. These operations are required to guarantee that once a process successfully completes an `acquire()` operation, no other process will complete an `acquire()` operation before the first process invokes a `release()` operation. Solutions to the mutual exclusion problem are often referred to as locks.

Essentially, a lock ensures that at most one thread can execute a *critical section* at any given time.

2.2 Synchronization Primitives

The problem of mutual exclusion has been studied for decades, and many solutions have been developed. In early solutions [19], it was considered important that only read and write operations be used to make the implementation independent of the availability of atomic hardware instructions. The appearance of multiprocessor machines fostered specialized hardware support in the form of atomic instructions. Popular variants are the *compare-and-swap* instruction or the *Load-Locked/Store-Conditional (LL/SC)* pair of instructions. All major processor architectures of today provide at least one of these choices.

2.2.1 Compare-and-swap

The *compare-and-swap* operation implemented by some architectures (*x86*, SPARC) can be described by pseudo-code like that in Figure 2.1. It takes a memory address, an old value and a new value. First, it compares the value at the memory address against the old value. If they match, the new value is written into the memory location. Otherwise, it remains untouched. Critically, an indication is returned if the memory location has been changed. All this happens atomically.

```
bool compare_and_swap(size_t *ptr, size_t old, size_t new)
{
    <atomically> {
        if (*ptr == old) {
            *ptr = new;
            return true;
        } else
            return false;
    }
}
```

Figure 2.1: Compare-and-swap

2.2.2 Load-Locked/Store-Conditional

On architectures which do not provide *compare-and-swap* as a single machine instruction, it can be synthesized using the *load-locked* and *store-conditional* instruction pair. The former loads the contents from a memory location and sets a reservation flag in the processor. The latter behaves like a normal store if this flag is still set. Otherwise, it does nothing. The reservation flag is guaranteed to be cleared when any processor does a store to the reserved memory location. By running *load-locked* followed by *store-conditional* in a tight loop until the store succeeds, as shown in figure 2.2, the semantics of *compare-and-swap* can be established.

It is important to make sure that this loop will not run forever, causing an unpleasant live-lock. It will eventually terminate because the loop will only repeat if the reservation flag has been cleared by some other processor issuing a store. But among a set of processors competing against each other for the same memory location, the only way for this to happen would be a successful *store-conditional* by another processor, thus terminating its respective loop. Therefore, no processor will go on indefinitely. This works only because the conditional store is the only store instruction inside the loop. Any additional store could potentially disrupt the reservation flag and hence lead to a livelock.

```
bool compare_and_swap(size_t *ptr, size_t old, size_t new)
{
    do {
        val = LL(ptr);
        if (val != old)
            return false;
        if (SC(ptr, new))
            return true;
    } while (1);
}
```

Figure 2.2: Compare-and-swap synthesized using LL/SC

2.2.3 Spin-lock

The *compare-and-swap* operation can readily be used for the implementation of a simple spin-lock protecting some critical section. Such an implementation requires one memory word with the semantics that if it is zero, the lock is currently free and the next inclined thread may enter. A thread which does not observe the value zero has to wait and retry until it does. Listing 2.3 illustrates the exact sequence of operations a thread has to execute in order to ensure that at most one thread can execute code inside the critical section.

Obviously, this simple spin-lock algorithm is not suitable as a general solution to the mutual exclusion problem. Unbounded spinning is a tremendous waste of energy; a spinning process should be able to suspend itself until the spinlock variable becomes available again, thus freeing up valuable processor resources. A hybrid approach composed of a spin-lock with a limited number of spin iterations and a suspend lock can be a very useful combination in many cases [28].

2.2.4 The ABA Problem

The *compare-and-swap* operation is very powerful and can be used in a wide variety of ways to build synchronization algorithms. Alas, there are some cases where a simple *CAS* is not enough and allows erratical behavior. One very common source of error is called the ABA problem. It is frequently encountered in the construction of lock-free data structures such as linked lists [24] or stacks.

```
void spin_enter(spin_t *s)
{
    /* Continuously try to lock s by setting it to 1. */
    while (!compare_and_swap(s, 0, 1))
        /* do nothing */;
}

void spin_leave(spin_t *s)
{
    /* Just set s to 0 to denote that the lock is now free. */
    *s = 0;
}

<...>
spin_enter(critical);
<critical section>
spin_leave(critical);
<...>
```

Figure 2.3: Spin-lock implementation

When applying *compare-and-swap* to construct an algorithm, it is easy to fall into the trap of mentally equating the successful execution of a *compare-and-swap* with the fact that the underlying value has not changed. In the majority of cases, the *compare-and-swap* will indeed only succeed if the value has not been touched. When a thread tries to execute `CAS(&x, A, Y)`, the *CAS* will usually succeed only if `x` "still" contains the value `A` at the time the *CAS* operation is executed. It is very well possible, however, that another thread changes the value of `x` to `B` and then back to `A` before the execution of *CAS*. If `x` is a pointer, the values referenced by it might have changed, from the first thread's point of view, without the thread having any chance of detecting this circumstance.

The ABA problem often arises in situations where heap-allocated nodes are linked together using pointers which are changed by *compare-and-swap*. Deleted nodes may be subsequently reused, leading to the problem. Specifically, the problem arises when the semantic value of the variable to be changed depends on other memory locations and is used as an operand in the comparison. In the spin-lock algorithm in listing 2.3, the only value which is used as a comparison operand is 0. Its meaning is always "the lock is free", independent of any other memory locations. Therefore, the spin-lock algorithm does not suffer from the ABA problem.

Possible solutions to the ABA problem include pointers tagged with a counter (leading to problems with wrap-around) or hardware instructions which perform atomic *compare-and-swap* on two values simultaneously (such as *cmpxchg16b* on *x86_64* processors[1]). Another possibility would be to use hazard pointers [25], where each thread is required to maintain a (usually short) list of pointers which it assumes to be valid. The easiest solution would often be the use of a garbage collector.

2.3 POSIX facilities

The POSIX standard for threads[30] describes several mechanisms for thread synchronization—semaphores, mutexes and condition variables.

A semaphore is one of the oldest synchronization primitives and used extensively in classic synchronization algorithms. In Unix, semaphores are of interest primarily because they can be used inside of signal handlers.

A mutex can provide a thread with exclusive access to shared data. If a thread tries to acquire a mutex while it is held by another thread, its execution will be suspended until the mutex becomes available. Mutexes are usually quite cheap—the dominant cost being one compare-and-swap operation for lock and unlock respectively.

If additional flexibility is needed, a mutex can be paired with a condition variable. One ore more shared state variables are usually associated with this conglomerate for communication between different threads. Condition variables are very versatile and can be used to construct other synchronization mechanisms, such as a semaphore or a Java monitor.

As an example showing the expressiveness of condition variables, the implementation of a semaphore is shown in figure 2.4. Both P and V operations lock the mutex upon entering and unlock it when leaving. The P operation then waits until the value becomes positive and decrements it by one. This does not necessarily set it back to 0 because many other threads may have incremented the value before the thread blocked in the P operation has had a chance to reacquire the mutex. The V operation increments the value and potentially signals a waiting thread if the value has become positive as a result.

The same can be implemented in Java, following almost the same design (figure 2.5).

[1]On some processors which support the *LL/SC* instruction pair, such a *double-compare-and-swap operation* could be synthesized, but because of granularity issues the two operand addresses would need to be severely restricted, just as they are in *cmpxchg16b*: the operands must be adjacent and the pair must be properly aligned on a 16 byte boundary.

```c
typedef struct {
    pthread_mutex_t m;
    pthread_cond_t c;
    int val;
} semaphore;

void sem_P(semaphore *s)
{
    pthread_mutex_lock(&s->m);
    while (s->val == 0)
        pthread_cond_wait(&s->c, &s->m);
    s->val--;
    pthread_mutex_unlock(&s->m);
}

void sem_V(semaphore *s)
{
    pthread_mutex_lock(&s->m);
    if (s->val++ == 0)
        pthread_cond_signal(&s->c);
    pthread_mutex_unlock(&s->m);
}
```

Figure 2.4: Semaphore implementation

```java
class semaphore {
    private int val;
    public synchronized void P() {
        while (v == 0)
            try {
                wait();
            } catch (InterruptedException e) { }
        v--;
    }
    public synchronized void V() {
        if (v == 0)
            notify();
        v++;
    }
}
```

Figure 2.5: Semaphore implementation in Java

2.4 Memory Models

2.4.1 Introduction

Any system which can run multiple processes in parallel, allowing those processes access to the same memory variables, needs to assert some properties regarding temporal memory behavior which those parallel processes can rely on. The Java Memory Model (JMM) defines a set of such properties.

In Java, the term "variable" can be any of an instance field, a static field or an array element. Local (stack) variables can never be accessed by any other thread than the one owning the stack. The Java memory model is not concerned with local accesses, consequently. Also, the JMM does not apply to any memory accesses a JVM has to perform internally, such as finding the address of a virtual function or determining the result of a `CHECKCAST` instruction.

Memory models affect different layers of a system. The nomenclature tends to differ somewhat between these levels. I will stick with the names "thread" and "variable". The individual layers are:

CPU Processors usually interact with memory through various caches and write buffers. From the CPU's point of view, the memory model is concerned with physical memory locations.

OS The operating system must make sure that a thread's view of (virtual) memory stays consistent when it is migrated between CPUs or when memory mappings are changed.

C compiler The C compiler does not care about multiple threads and can manipulate the code freely as long as reads and writes from one thread appear to the thread itself to happen in program order.

Garbage collector When the GC moves objects around in memory, this has to be completely transparent to the Java program. A relaxed hardware model could lead to variables transiently changing values. The GC must prevent such effects.

JIT compiler The JIT compiler is similar to the C compiler and can perform a wide variety of optimizations. It has to care about the rules of the JMM when it comes to `MONITORENTER`, `MONITOREXIT` and access to `volatile` and `final` variables.

Java compiler The Java-to-bytecode compiler does not perform a lot of optimization typically. It must carefully conserve ordering constraints imposed by `synchronized` blocks and accesses to `volatile` variables.

The semantics dictated by the Java Memory Model (JMM) have to be preserved across all these levels. A JVM does not have to care about the Java compiler level because it never deals with Java source code and operates only on pre-built class files. So CACAO only has to ensure that the bytecode it executes can only observe memory behavior consistent with the JMM rules.

Memory models deal with the general problem of variables written by one set of threads and read by another set. In many cases, these sets will consist of only one thread each. The memory model is concerned with the question what possible outcomes a particular read can produce, given an overview over all threads running on a system.

For single-threaded programs, the answer to this question is simple; a read from a variable should always see the value that was last written to that particular variable. If there has not been such a write, Java requires that the variable's initial value be read. All variables must have an initial value in Java.

When two or more threads operate on the same variable, the definition of "the last value written" becomes blurry. In general, because of a great variety of contributing factors, like caches, write buffers, various latencies, compiler reorderings and so on, it cannot always be decided if a particular write happens before or after another.

It should be noted that implementations are free to perform the actual memory accesses, at the hardware level, in any order as long as this fact cannot be observed by the program. Such reorderings are in fact very commonly performed by both processors and compilers. In the single-thread case, there is ample room for such reorderings to occur. A large amount of outstanding reads is likely to better utilize the underlying caching system. Reads can be executed far in advance, as long as they respect dependencies imposed by possibly conflicting writes. Similarly, writes can be deferred for a long time; they usually spend some time in a write buffer before they finally reach their physical memory destination.

Similarly, the JMM is not concerned with the actual sequence of writes reaching physical memory. Only behavior detectable by the Java program is regulated.

2.4.2 Hardware memory models

Similar to the single-threaded program, a system with only one CPU (core) can perform memory operations in any order as long as they appear to the program to be in the correct order. As soon as more processors are involved, the processor manufacturer has to specify what memory behavior can be expected. The processor memory model affects the individual read and write machine instructions in the same way the JMM affects bytecode instructions.

Modern CPUs often support various types of memory. For example, uncached and cached memory or write-combining memory for the graphics framebuffer. For Java, only cached memory is of interest; a machine's RAM is usually represented by this type of memory. In general, processors communicate with one or more caches private to each processor. These caches are constantly synchronized with one another and with main memory, so a write performed by one CPU will eventually be seen by every CPU in the system. The order in which this happens, is not necessarily always very intuitive.

The prevalence of *x86* and *x86_64* architectures in today's computing landscape may lead one to the conclusion that most current CPUs do not reorder memory accesses extensively. The memory model of these architectures is quite strict and does not allow many reorderings commonly found in other, arguably rarer architectures. When a program developed for such a strict memory model is ported to an architecture with a more relaxed model, subtle and very hard to find concurrency problems are likely to appear. The example shown in figure 2.6 will behave in the intuitively expected way on *x86* and *x86_64* but not on most other architectures.

In the following examples, `x` and `y` denote shared memory variables while `a`, `b` and `p` are local variables visible to the respective thread only. The shared variables always start out with the initial values 0 or `null`.

Thread 1	Thread 2
`a = y;`	`x = 1;`
`b = x;`	`y = 1;`

`a` = 1 and `b` = 0 is not possible on x86 CPUs.

Figure 2.6: No surprising behavior on *x86* processors

If thread 1 can see the value 1 written to `y`, it will also see the previously written value 1 in `x`. On *x86* processors, reads are never scheduled ahead of earlier reads, and writes are never scheduled ahead of earlier writes.

Thread 1	Thread 2
`y = 1;`	`x = 1;`
`a = x;`	`b = y;`

`a` = `b` = 0 is possible.

Figure 2.7: Common result on current CPUs

However, the result shown in figure 2.7 is very common across almost all processors. On most processors, writes go to a write buffer, and reads can be scheduled ahead of them. Thus, the reads in both threads may still see the previous values of `x` and `y`. In a strictly sequential model, at least one thread would have to see the value written by the other thread. If this is required[2], the processor's ordering must be explicitly restricted by a memory barrier instruction, as shown in figure 2.8. Note that the barrier is necessary on both sides because reordering on either side can result in `a` = `b` = 0.

Thread 1	Thread 2
`y = 1;`	`x = 1;`
`<MEMORY BARRIER>`	`<MEMORY BARRIER>`
`a = x;`	`b = y;`

`a` = `b` = 0 is *not* possible.

Figure 2.8: Use of a memory barrier

The most aggressive form of memory reordering is presently performed by the Alpha processor on which the behavior shown in figure 2.9 might be observed.

Initially, `p.z` = 0.

Thread 1	Thread 2
`b = -1;`	`p.z = 1;`
`a = x;`	`<MEMORY BARRIER>`
`if (a)`	`x = p;`
`    b = a.z;`	

`b` can be 0.

Figure 2.9: Surprising result on an Alpha processor

Thread 2 writes to the variable `p.z` and subsequently publishes a reference to this variable in the shared variable `x`. The memory barrier is supposed to make sure that the object referenced by `x` has been fully initialized before another thread can see the reference. Thread 1 dereferences the pointer to read the value supposedly written by thread 2. Even with the enforced write ordering dictated by the memory barrier, thread 1 can still see the previous value 0, even if it seems obvious that it needs to determine the value of `p` first.

Because the hardware memory models differ significantly across the CPU architectures supported by CACAO, a different set of memory barriers needs to be developed for each architecture. The rules governing access to shared memory locations with mixed word sizes also differ significantly across ar-

[2]It is often desirable to require this property in algorithms for mutual exclusion, as will be seen in the description of tasuki lock and KKO lock in section 3.3.1.

chitectures. Fortunately, such mixed-size accesses can not be expressed in Java.

2.4.3 Sequential consistency

There is a model which disallows many memory reorderings and exhibits very intuitive behavior. It is called *sequential consistency* [20, 1].

> **Definition:** [A multiprocessor system is *sequentially consistent* if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Informally, in a sequentially consistent system, it appears as if only one memory operation (read or write of a shared variable) can be executing at any one time. The results of a write becomes visible to all threads immediately. Instructions in each specific thread are executed in program order.

While sequential consistency is a convenient model, it disallows lots of useful performance optimizations. Typical multiprocessor machines implement at least some relaxations in order to improve performance. These relaxations can be disabled selectively by the use of barrier instructions in appropriate places. However, sequential consistency can *not* always be achieved this way[5].

2.4.4 JSR 133

The specification of a memory model for the Java programming language has proved to be a more complicated undertaking than initially anticipated. As a result, the original specification was difficult to understand and had undesirable consequences. Popular implementations of the JVM violated the rules of the JMM mostly because of its difficult interpretation as well as run-time penalties that strict adherence would have caused. Because of the unclear specification, JVM implementations have always employed all sorts of optimizations that did not respect the specification. This led to a revised specification, aiming for better optimization potential and improved clarity and intuitivity.

Improved areas of the new specification include

- More intuitive semantics of programs.
- More efficient implementation on popular hardware.
- `final` fields cannot appear to change.

- New semantics of `volatile` variables.
- A concise set of formal rules.

2.4.5 The Java Memory Model

The Java memory model[23, 13] describes a very relaxed memory model. This choice was made to give compiler and hardware enough freedom to achieve good performance.

The JMM also describes a *happens-before* relation that forms a partial order over a program's synchronization actions (*lock*, *unlock*, `Thread.join`, ...). It then goes on to call a program *properly synchronized* if all its *conflicting memory actions* are ordered by the *happens-before* relation (or program order). A *memory action* is a read or write to a shared variable; two memory actions are conflicting if at least one of them is a write. A pair of *conflicting memory actions* is called a *data race* if there is no ordering imposed by *happens-before* (or program order). The execution of a properly synchronized program will appear to be sequentially consistent.

So in order to be able to efficiently reason about the temporal behavior of a program, programmers should strive to make them properly synchronized. But even in the presence of data races—that is, *not* properly synchronized—, Java programs must not produce values "out of thin air". This property is established in the JMM through a voluminous body of theoretical constructs. In a reasonable JVM implementation on real hardware, "out of thin air" values will not occur as long as the appropriate memory barriers are used for synchronization actions. On the other hand, a heavily optimizing compiler might be restricted by this rule.

Areas affected by the JMM

synchronized The Java language allows a program to enter an object's monitor with the `synchronized` keyword. A `MONITORENTER` operation is performed upon entering a synchronized code block while a matching `MONITOR-EXIT` is performed upon leaving. `synchronized` blocks can be established implicitly, by attaching the `synchronized` attribute to a method's definition, or explicitly, by enclosing a sequence of statements inside a `synchronized` block. In the latter case, the object to lock is specified explicitly, in the former it is always the `this` reference or, in the case of static methods, the class object. An *unlock* action *happens-before* all subsequent *lock* actions *on the same object.*

volatile Access to `volatile` variables is guarded by its own set of special rules in the JMM. The intent of `volatile` is similar to the C language. It provides much stronger visibility guarantees, however. `volatile` writes behave very similar to memory barriers; in fact the semantics are even a bit stronger.

All `volatile` reads and writes need to observe behavior consistent with sequential consistency. Because of this, there is a total order over all `volatile` actions (reads and writes). A write to a `volatile` variable *happens-before* all subsequent reads of the same variable, according to this total order.

final The semantics of `final` variables are easily understood—they can never change[3]. Yet, under the old memory model, it has been possible for threads to observe different values over the lifetime of `final` variables.

The problem with `final` reference fields is that the referenced object must be observed as fully constructed. While this is clearly the case from the initializing thread's viewpoint, it is possible for another thread to observe the reference in the `final` field before the memory contents it references have become completely visible to this thread.

The revised JMM demands that `final` variables can not appear to change, as long as the objects containing them are properly constructed. That means that object's constructors cannot leak their `this` reference to other threads.

java.lang.String The Java class `java.lang.String` class implements immutable strings. To ensure that they actually appear to be immutable, the JVM implementation must not allow any Java code to see uninitialized contents through references to newly created strings. This is very similar to the `final` field problem. However, since string creation is an operation internal to the JVM, there is no way for Java code (or bytecode) to thwart this behavior.

2.4.6 Surprising effects of the JMM

The following examples show some effects of the Java memory model that may be surprising to an unaware programmer.

In the sequential consistency model, the combinations $(0, 0)$, $(1, 1)$ and $(0, 1)$ for $(\texttt{a}, \texttt{b})$ can be expected as a result of the operations in figure 2.10—if both instructions in thread 1 execute before the instructions in thread 2, the result

[3]It is possible to change a `final` variable through the reflection API, although such changes need not always have the desired effect because of compiler optimizations.

Thread 1	Thread 2
`a = y;`	`x = 1;`
`b = x;`	`y = 1;`

Any combination of 0 and 1 is allowed for `a` and `b`.

Figure 2.10: Effects of reordering reads or writes

will be $(0, 0)$. Conversely, if both instructions in thread 2 execute before the instructions in thread 1, the result will be $(1, 1)$. With interleaved execution, $(0, 1)$ may occur if the two reads in thread 1 are scheduled between the two writes.

The result $(1, 0)$ is also allowed in the JMM, however. An implementation is free to execute both reads in any order—just reversing the two read instructions in thread 1 could easily produce this result. Similarly, the writes in thread 2 can be reordered freely.

```
int x;
```

Thread 1	Thread 2
`a = x;`	`x = 1;`
	`x = 2;`

`a` may be 0, 1 or 2.

Figure 2.11: Atomic values

In example 2.11, it might seem obvious that only 0, 1 or 2 can show up in `a`. Specifically, `a` must not contain anything else, like 42. The JMM does not allow such creation of values "out of thin air", mostly for security reasons.

```
long x;
```

Thread 1	Thread 2
`a = x;`	`x = 1;`
	`x = -1;`

`a` may take on surprising values.

Figure 2.12: 64 bit values

One exception to this rule is the non-atomic treatments of 64 bit values. For performance reasons, a JVM is allowed to split up the reads and writes to such a value. On 32 bit implementations, common outcomes for `a` in 2.12, apart from the expected 0, 1 and -1, would be 4294967295 (`0xffffffff`) or -4294967296 (`0xffffffff00000000`).

So far, the executed statements were just plain, unconditional read or write instructions. The following example shows how compiler optimizations might produce surprising results.

Thread 1	Thread 2
`a = x;`	`b = y;`
`if (a == 1)`	`if (b == 1)`
`    y = 1;`	`    x = 1;`
	`else`
	`    x = 1;`

`a` = `b` = 1 is legal.

Figure 2.13: Compiler optimizations

In example 2.13, the compiler might detect that `x = 1` is always executed and can therefore be scheduled before any other instruction. The JMM allows such optimizations.

Implementation consequences

A JVM implementation unaware of the JMM would allow the Java program to observe memory behavior consistent with the memory model of the hardware that the JVM is running on. This is acceptable only as long as the hardware memory model is more restrictive than the JMM. Fortunately, this is often the case, at least in the absence of `synchronized`, `volatile` and `final`.

These synchronization features often need to be supported by memory barriers in generated code or in C code. Doug Lea's JSR-133 cookbook[21] lays out in detail which kinds of barriers are needed on various architectures. The amount of barriers can appear a bit overwhelming, though, as on many architectures, most of them are unnecessary. The types of barriers are also described in the cookbook. In short, the JVM can use barriers like described in the following to implement the individual language features.

synchronized A *LoadLoad* and a *LoadStore* barrier is required after a `MONITORENTER`. On many platforms, this is achieved only by issuing a full barrier (*StoreLoad*). The *x86* does not need a barrier because the *CAS* used for entering the monitor acts as a full barrier. Additionally, *LoadLoad* and *StoreStore* are always no-ops on *x86*.

Before `MONITOREXIT`, *StoreLoad* and *StoreStore* barriers are needed. On most platforms, this is a full barrier. In tasuki lock, there is a full barrier in the unlock path anyway, so only the *StoreStore* is necessary.

volatile A *StoreStore* barrier is needed before each `volatile` store and a *StoreLoad* thereafter. A `volatile` load needs exactly the same barriers as a `MONITORENTER`.

32 bit architectures need to make sure that `volatile` 64 bit values (longs and doubles) are treated atomically.

final If a class has any `final` fields, then its constructor must issue a *StoreStore* barrier before returning. On architectures where data dependency does not order loads (only Alpha), a *LoadLoad* is required before each load of a `final` field.

The simplest and most conservative sufficient approach would be to place a full barrier after each `MONITORENTER` and before and after each `MONITOREXIT` and `volatile` access.

Additionally, the Alpha requires *LoadLoad* barriers before accessing internal fields of strings or arrays (like the length) and *StoreStore* barriers when creating such objects.

2.5 The Bytecode Verifier

The Java bytecode verifier needs to be able to statically prove, among other tasks, that all bytecode instructions are properly typed for their arguments, otherwise it rejects the code in question. In a very similar way, it *could* also be used to prove the pairing of `MONITORENTER` and `MONITOREXIT` instructions but this is not required. In fact, Java bytecode may issue these monitor instructions quite arbitrarily. However, because most existing Java classes have been produced by Java compilers, such arbitrary locking sequences are rare. Hotspot is optimized to take advantage of proper pairing and allocates lock records on the stack if proper pairing can be proved statically. In fact, it does not even compile methods which lack proper nesting but executes them in interpreter mode instead[31].

One restriction to the arbitrary execution of monitor instructions applies: when a stack frame becomes deactivated ("is left"), possibly because of an exception, the JVM is free to leave (release) all monitors acquired in that stack frame. In bytecode compiled from Java source code, there are always `MONITOREXIT` instructions to this effect anyway; many exception handlers exist just for this reason. In the absence of such measures, the JVM can make use of this freedom granted by the JVM specification. This is what the Hotspot interpreter does.

Chapter 3

Java Locks

3.1 Introduction

Java's multi-threaded nature requires that the language provide a mechanism for synchronization among threads. The Java platform provides such a mechanism based on the monitor concept [15]. Any Java object can serve as a monitor at any time. Because the JVM cannot know in advance which objects will be used as monitors—and most objects never will [4]—it should not impose an undue space overhead for objects which never take part in synchronization operations. The exact amount of overhead considered undue has varied in different contexts, but for use in CACAO, a locking algorithm may allocate a full word in the object header. This overhead is generally accepted as tolerable, and the lockword can often be shared with the garbage collector in order to mitigate the memory waste.

Primarily because Java has had built-in support for multi-threading from the beginning, most Java libraries are written with multi-thread safety in mind and therefore need to make use of the synchronization features. This results in very frequent execution of locking operations, demanding a time-efficient implementation.

This chapter presents various locking algorithms which have been considered for CACAO. The "Related Work" section will describe them briefly. Full details can be found in the respective referenced papers. The "Implementation" section describes only those implementation details which cannot be found in these papers. In particular, recursive locking and the implementation of *wait/notify* are often omitted.

In the following two chapters, threads will often be said to "lock an object", "take a lock", "lock a monitor" or "enter a monitor". These are all considered the same, namely a Java thread performing a `MONITORENTER` operation on a Java object.

3.2 Thread models

There are two flavors of multi-threading: “green” and native threading. With “green” threads, all of a program’s threads executes in a single OS-level thread. Switching from thread A to B is done by storing all of A’s registers, including its stack pointer and return address, in A’s thread structure and restoring the same set of registers from B’s thread structure, returning to the just-restored return address. Caller-saved (temporary) registers can be omitted if the switching is done by calling a C function. Blocking system calls need to be emulated with non-blocking equivalents because they would block the whole program instead of just a single thread. Thread switches are triggered by blocking operations, including entering or waiting on Java monitors, or by some sort of periodic real-time interrupt. Because everything is controlled by the program itself, interruptions from the periodic timer—the only source of non-deterministic thread switching—can be temporarily suppressed and handled at convenient moments. While this ability does not come for free, it might allow other, more expensive, provisions to be omitted. Because everything runs in a single thread (and more importantly, on a single processor), expensive atomic instructions and memory barriers are not necessary.

The “green” model was very popular among early implementations of the Java virtual machine because both OS-level synchronization and atomic hardware instructions were very expensive and because SMP machines were the exception rather than the norm. On single CPU systems, better performance could be obtained by handling the threads in the JVM itself, bypassing many expensive mode transitions between user mode and kernel mode, that would have been required in a kernel-based approach.

In the native model, each program thread runs in its own OS thread. Today, the most important consequence of this for Java is that it enables Java code to run on more than one processor simultaneously. It is also much cleaner than the “green” model which has always been more somewhat of a kludge with lots of OS- and architecture specific parts. As a downside, however, preemptive thread scheduling and simultaneous execution on multiple processors mandates the use of expensive mechanisms like atomic instructions.

Today, with Moore’s law applying more to the number of cores than to processor clock speeds, the native model clearly needs to be utilized in order to make current processors’ full power available to Java programs.

OS provided locking primitives were very slow on most platforms because they involved switching to kernel mode. The Java lock optimizations discussed in this chapter work around this by implementing the fast path in user space, resorting to kernel support only when a thread has to be stopped (because of lock contention). The same technique has been used later to speed up OS-provided mutexes on Linux[11].

3.3 Related Work

The topic of locking for Java has been studied long and deeply. As a result, locking implementations for Java have improved considerably over the last few years. Early implementations of the Java virtual machine allocated monitors for objects on demand and set up an association between the object and its monitor by means of a global hash table called the monitor cache [18]. This kind of implementation was quite inefficient but acceptable when using "green" threads. It was found to be completely unsuitable for native threads, though, because access to the monitor cache itself had to be synchronized.

3.3.1 Locking algorithms for native threads

The following algorithms all rely on the observation that for the majority of lock operations in Java, there is no contention. Most locks are either only ever acquired by a single thread or by one thread at a time. Recursive locking is considered but only to a limited depth. While recursive locking happens frequently in Java, large recursion counts are rarely encountered. All algorithms make use of atomic instructions which are available on every major general purpose CPU architecture or can be synthesized by a short instruction sequence. Since these are relatively expensive, it is customary to use them as a basis for performance characterization.

Thin Lock

The thin lock algorithm was developed by Bacon et al. It requires an underlying general Java lock implementation as a fall-back scenario for when the optimized fast path cannot cope with a specific usage pattern (like very deeply nested locking or heavy lock contention). It was argued that using a full word in the object header would be an unacceptable space-time trade-off. The thin lock algorithm needs 24 bits which the authors could salvage by changing the encoding of other header fields without increasing the size of the object header.

The 24 bits in the header, which will subsequently be called the "lockword", have two different modes: flat mode and inflated mode. A single bit, the shape bit, that is also part of the lockword, indicates the current mode of the lockword. In flat mode, the shape bit is 0 and the remaining bits contain a recursion counter and a thread id. In inflated mode, the shape bit is 1, and the rest contains a monitor id that points to a larger memory structure containing the full monitor object as required by the underlying locking algorithm. When the lock is free, all bits are zero. Threads trying to acquire the lock can therefore use `CAS(&lockword, 0, <tid>|1)` to declare themselves owner of the lock. Because the most common operation is locking an unlocked object, this operation will succeed in the majority of cases. Only if it fails, further processing is necessary. In this case, the thread will enter a spin-loop called the inflation loop and try to install a lockword with the shape bit set. This inflated mode lockword points to a (possibly newly allocated) monitor that the thread enters just before installing it. When a thread releases the lock in flat mode, it just stores 0 in the lockword and is done. The cost in the common case without contention is therefore one atomic instruction for locking and zero—just a simple store—for unlocking.

There are two problems, however, with this approach. First, once an object is inflated, it can never go back to flat mode. Second, it can potentially cause a thread to spin forever, when another thread holds a lock on a non-inflated object. These problems were justified with ease of implementation and *locality of contention* which states that if there is contention for a specific object, most subsequent locking operations performed on this object will also experience contention. For the single-threaded programs used for evaluation, these were reasonable assumptions but the authors never verified them.

Meta-Lock

Agesen et al. developed another algorithm called the meta-lock [3]. It does not require an additional full word in the object header but only two available bits. They are used to encode one of four possible states: *neutral*, *busy*, *locked*, *waiters*. If these two bits can be made available without increasing the object size, it is very space-efficient. The downside is that it requires at least two atomic instructions for locking and two for unlocking in the base algorithm, and one for each in an optimized variant with an additional third bit in the header.

The lock algorithm works by first storing a pointer to a thread-local data structure called the execution environment in the lockword. The state for the lockword is set to *busy*. This happens with an atomic swap operation so that the state of the previous lockword can be inspected. In the author's implementation, the rest of the lockword contains the object's hash value.

Threads which want to read the hash value of an object will thus also have to execute the expensive locking protocol. The execution environment contains a number of auxiliary fields and, most importantly, an OS lock on which contending threads can block. After the execution environment is installed, the locking thread can freely update any number of fields because other threads will detect the *busy* state and will block on the locking thread's execution environment. The locking thread will transition to *locked* state and install a pointer to a lock record, concluding the locking sequence. When a lock is released, the first action is again marking the lockword as *busy* and installing a pointer to the execution environment. After that, all remaining necessary actions, like waking other threads and recycling the lock record, are performed. In essence, the algorithm uses a "lock for the lock", hence the name meta-lock.

Because of the high cost of atomic instructions, this algorithm is not very interesting with regard to inclusion in CACAO where a less space-efficient algorithm is considered a better choice.

Relaxed-Lock

The relaxed-lock[8] algorithm by Dice is well rounded in terms of performance and space requirements. It requires a full word in the header which can be shared with a hash code, so the object size needs not be expanded. The atomic instruction count is comparable to thin lock. For uncontended locking, it performs one atomic instruction and for unlocking a simple store followed by a memory barrier.

Unfortunately, the algorithm is rather complicated in terms of code complexity and subtlety as well as required infrastructure. Each thread manages a pool of available lock records from which it obtains one when it wishes to enter a monitor. It then tries to insert a pointer to the lock record into the lockword via *CAS*. If this succeeds, the monitor is locked. In terms of thin-lock, the locked object is always inflated as soon as it is locked. There is no "thin" locked state. The advantage is that no spinning is needed. In contrast to spin-lock, inflated objects don't have to stay this way. The relaxed-lock algorithm uses eager deflation where it tries to restore a locked object's original state as soon as possible (during unlocking). Lock records can change their owner if a thread tries to lock an object while it is being deflated by another thread. This could lead to an undue lock record hoarding by one thread and needs to be counteracted. Another problem is that lock records might be leaked. Because of this, a list of potentially leaked lock records needs to be maintained and periodically checked and cleared, preferably at garbage collection time.

Because of these necessary maintenance actions at times other than locking or unlocking, it cannot be included as cleanly in CACAO as would be desirable. A variation of the algorithm has been attempted which does not migrate lock records between threads but it suffered from the ABA problem and grew more complicated than originally expected. It has been replaced with tasuki lock.

Tasuki Lock

Onodera proposed an extension of the thin-lock algorithm, called tasuki lock [26] ("tasuki" is not the name of any involved person). It addresses two shortcomings of the thin-lock: spinning and inability to deflate. To make this work, the algorithm has two additional requirements. First, tasuki lock needs an external facility like a hash table to maintain associations between Java objects and lock records. Second, it needs an additional bit in the object header, called the FLC (fat lock contention) bit. Critically, this bit cannot be part of the lockword.

As soon as the initial *CAS* fails, the object's associated lock record is looked up (or allocated). The failing thread then blocks itself on this lock record. This modification removes the spinning from the thin-lock algorithm. Before a thread blocks itself, it sets the FLC bit so that the thread holding the lock will eventually become aware of the blocked thread and unblock it. Because everything is protected by the associated lock record and because inflating is relatively cheap, the lock can be deflated any time it is held (thereby releasing it) by just writing the initial value 0 into the lockword.

The SableVM modification

The problem with the FLC bit is that it must be allocated in a word separate from the lockword. Such a word is not always easily available. Gagnon and Hendren encountered this problem during the implementation of SableVM[12]. They modified the algorithm so that the FLC bit can be allocated in the thread structure where the allocation of an extra word does not raise space concerns.

KKO Lock

In [17], Kawachiya, Koseki and Onodera observed that most Java objects, are only ever locked by a single thread over their entire lifetime, if they are locked at all. In the absence of other threads, it should not be necessary to use expensive atomic instructions for repeated locking and unlocking of an object by the same thread. Their proposed solution is an algorithm that reserves an object's lock to the first thread which accesses it. If another thread tries to lock the same object, it has to cancel the reservation first. This happens by sending a signal to the owner thread, thus stopping its execution, and examining the stopped thread's instruction pointer. In most cases, the lock can simply be unreserved while the owner thread is stopped. Occasionally, though, the thread is in an *unsafe region* where the instruction pointer must be adjusted so that the *unsafe region* is restarted when the thread continues execution.

Reservation cancellation is a very expensive operation in this algorithm, which led the authors to reconsider the problem. In [27], they devised a new locking algorithm, called KKO lock, after the author's initials. It combines a Dekker-style lock[10] and the tasuki lock algorithm.

Dekker's algorithm can ensure mutual exclusion for an arbitrary but fixed number of participating threads. It does not require combined atomic instructions like *compare-and-swap*. In KKO lock, only 2 participants are allowed. For this size, Dekker's algorithm is very simple. It requires 2 boolean flags, one for each thread, which are both initially 0. A thread trying to enter the critical section first sets its own flag to 1, then verifies that the other thread's flag is (still) 0. If this is the case, the thread has successfully entered the critical section. Otherwise, it has to reset its own flag and try again. The thread leaves the critical section by just resetting its flag to 0. On multiprocessor systems, a memory barrier must be inserted between setting a flag and reading the other one.

KKO lock needs to make sure that only 2 threads perform Dekker's algorithm. The thread which has reserved a lock is always one of the two participants. The other thread is decided by use of a *compare-and-swap* instruction. The cost of KKO lock for the thread holding a reservation is thus 1 memory barrier at entry and 1 memory barrier at exit (because of the FLC bit). For a thread without a reservation, it is 1 CAS at entry, just like in the tasuki lock.

Quickly Reacquirable Locks

In [9], Dice et al. discuss a strategy for completely getting rid of expensive hardware synchronization instructions in the "reserved" path. They specifically exploit the *SPARC* TSO memory model[33] to prevent a read from being reordered before a write by using mixed word sizes to access the same memory word. The same technique is also applicable to the *x86* and *x86_64* architectures.

Biased Locking

Unfortunately, this “soft” memory barrier is just as expensive as a real barrier on many machines. It seems that the only way to really speed up locks is the elimination of all synchronization in the reserved path. Hotspot uses a scheme described in [31] which is very similar to Kawachiya’s “Lock reservation”. However, instead of signals, they can use safepointing[2] for lock revocation which is already built into Hotspot. Additionally, because the cost of revocation is very large, Hotspot keeps track of the number of revocations for each class and disables reservation for classes with large counts. When this happens, it also revokes all class instances at once. Because lock records are allocated on the stack in Hotspot, lock revocation also entails walking and manipulating the stacks of all threads.

Chapter 4

Implementation in CACAO

4.1 Java Locks

4.1.1 Overview

For a fully functional implementation of Java locks, the first step is to implement all the functionality based on platform functionality, specifically the pthreads functions for mutexes and condition variables. All existing incarnations of thin locks can only operate on top of an underlying implementation of Java monitors, absorbing common fast-path calls but passing through calls when the full functionality cannot be provided by the thin-lock algorithm. Simple uses of `MONITORENTER` and `MONITOREXIT` can be handled by the thin-lock code but calls to `wait()` will require a more sophisticated mechanism.

In the following list, I will use Java notation only for better readability. Some terms have to be translated to pthreads nomenclature if it is to be understood in terms of pthreads functionality.

Java	pthreads
monitor	mutex
`notify()`	`pthread_cond_signal()`
`notifyAll()`	`pthread_cond_broadcast()`
`wait()`	`pthread_cond_wait()`

The Java model is quite similar to the basic usage pattern of pthreads mutexes and condition variables in the following ways.

- Before a thread can suspend itself, it has to enter the monitor.
- Calling `wait()` will release the monitor and atomically add the thread to the waiting set for this object.

- Before the `wait()` call can return, the monitor will be reacquired. The returning thread will have to contend for it just like it would in an ordinary call to monitorenter. This means that other unrelated threads may be able to acquire the monitor between its release by the notifying thread and the lock by the woken-up thread. This also means, that the woken-up thread can continue only after the notifying thread has released the monitor.
- `notify()` wakes up one thread waiting on the monitor, and there is no way to influence which thread it will be. There is also a function, `notifyAll()`, waking up all threads.
- The `wait()` method may return spuriously—that is without any thread having called `notify()` on the monitor. Therefore, in almost all cases, a loop has to be used to repeatedly check if the program is in a state that allows the waiting thread to continue.
- Calls to `notify()` must not be lost. If thread t enters a monitor and, while inside the monitor, issues n `notify()` calls, and there are N threads waiting on the monitor, then the number of threads woken up will be at least n if $n \leq N$ or N otherwise. That is, at least one thread will wake up in response to each `notify()` call as long as the monitor's waiting set is non-empty.

  This is an important property that could easily be overlooked, leading to difficult-to-explain program hangs. Because there is no guarantee as to when threads waking up from a `wait()` will be scheduled, they may not yet have removed themselves from the waiting set, leading to the situation that a subsequent `notify()` call would try to notify an already-notified thread.
- In order to send a `notify()` call, a thread needs to first enter the monitor. This is not strictly required by pthreads but is recommended for predictable scheduling. Application logic will, in most cases, automatically lead to a program structure which follows this recommendation. In Java, it is mandatory, and not adhering to the rule will result in an `IllegalMonitorStateException`.

Considering that these two APIs are so similar, it would seem practical to implement Java monitors in a straightforward way as just a pair of a mutex and a condition variable. Things are a bit more complicated than this, however. `Thread.interrupt()` can wake up any thread waiting on a monitor, yet it is not possible to wake up a specific thread blocked on a condition variable. The only way to implement this would be to wake up all threads, but the overhead caused by this strategy, especially in situations with many threads waiting on a monitor, would not be justifiable. This situation calls for the use of separate condition variables for each waiting thread. Such an approach has to track threads waiting on a given monitor by using a list or some similar data structure. This gives the algorithm freedom of choice when it comes to notifying a single thread—naturally, threads with the highest priority should be chosen first. However, since thread priorities in general are very poorly implemented on CACAO's main platform, Linux, thread priorities are currently ignored in CACAO.

4.1.2 Supporting data structures

In CACAO, a `threadobject` structure is allocated for every run-time thread. Listing x shows parts of this structure. Only fields relevant to the Java lock implementation are shown.

```
struct threadobject {
   uintptr_t           thinlock; /* pre-computed thin lock value */
   s4                  index;    /* thread index, starting at 1  */

   /* these are used for the wait/notify implementation          */
   mutex_t             waitmutex;
   pthread_cond_t      waitcond;
   bool                interrupted;
   bool                signaled;

    /* for the sable tasuki lock extension */
   bool                flc_bit;
   struct threadobject *flc_list; /* FLC list head for this thread*/
   struct threadobject *flc_next; /* next pointer for FLC list    */
   java_object_t       *flc_object;
   mutex_t             flc_lock;
   pthread_cond_t      flc_cond;

   <...>
};
```

`thinlock` contains the value that this thread will try to enter into an object's lockword initially. It is a bit field consisting of three sub-fields—recursion count, thread index and shape bit. Because a zero value for the recursion field means a nested locking level of 1, and because this is a thin lock—the shape bit is not set—the `thinlock` value is formed by just shifting the thread id left by the size of the recursion field plus one, the size of the shape bit field. `index` is a short identifier for the thread as the thread id provided by the OS would be too large for the bit field in the thin lock. `waitmutex` and `waitcond` are used for the *wait/notify* implementation, as are the 2 remaining flags. `interrupted` is set by `Thread.interrupt()`, while `signaled` is set by `Object.notify()`.

The other data structure of central importance is `lock_record_t`. It is shown in its entirety in listing y. Lock records are associated with Java objects on demand, during inflation.

```
struct lock_record_t {
   java_object_t       *object; /* object protected by this monitor*/
   struct threadobject *owner;  /* current owner of this monitor   */
   int                  count;  /* recursive lock count            */
   mutex_t              mutex;  /* heavy OS lock                   */
   list_t              *waiters /* list of threads waiting         */
};
```

`object` points to the Java object that is currently protected by the lock record. Its value is only set at initialization time; it can only change when a lock record is recycled. It's mostly needed for hash table management and cleanup. `owner` denotes the thread that is currently holding the lock record. This means that it has locked the associated object or that it is waiting on it. `count` is the number of nested locks by the thread currently owning the lock record. `mutex` is a OS-level mutex used for blocking on the lock record itself. `waiters` is a linked list of threads waiting on the object.

None of these fields may be changed by any other thread than the one holding the lock record's mutex. In fact, `object` and `owner` never change. They are initialized when the lock record is first associated with an object. The lock record will stay in memory until after its associated object has been reclaimed by the garbage collector.

4.1.3 monitorenter/monitorexit

With these structures in place, the *monitorenter* and *monitorexit* operations can be implemented almost trivially. These operations are only invoked for objects which have already been inflated. The fast paths and inflation itself are handled by the tasuki lock algorithm described in [26].

```
void slow_monitor_enter(java_object_t *o)
    uintptr_t      lockword = o->lockword;
    lock_record_t *lr = GET_FAT_LOCK(lockword);

    /* Check for recursive locking */
    if (lr->owner == t) {
        lr->count++;
        return;
    }

    mutex_lock(&lr->mutex);
    lr->owner = t;
}
```

`slow_monitor_enter` only needs to handle the special case of recursive locking by the same thread. For this case, incrementing the recursion counter is the only thing to do. If the lock record is owned by another thread, the pthreads implementation is used to lock its `mutex` member. When this call succeeds, the only thing left is to set the owner of the lock record. This will be used later for distinguishing the recursive locking case and also for deciding if `IllegalMonitorStateException` needs to be thrown.

```
void slow_monitor_exit(java_object_t *o)
{
    uintptr_t      lockword = o->lockword;
    lock_record_t *lr = GET_FAT_LOCK(lockword);

    if (lr->owner != t) {
        throw_illegalmonitorstateexception();
        return;
    }

    if (lr->count != 0) {
        /* Recursive lock. Just decrement the counter, it will
           still be locked. */
        lr->count--;
        return;
    }

    /* unlock this lock record */
    mutex_unlock(&(lr->mutex));
}
```

First, `slow_monitor_exit` needs to check if the lock record is actually owned by the current thread. If not, `IllegalMonitorStateException` is thrown. Next, like in the locking function, the recursive locking case needs to be handled separately. Only the recursion counter needs to be decremented. In the non-recursive case (the outermost *monitorexit*), the lock record's `mutex` is unlocked by the pthreads implementation.

4.1.4 wait/notify

wait

The *wait/notify* mechanism is quite independent from the fast locking algorithm but it needs access to some of the same data fields. Slightly condensed versions of the *wait* and *notify* implementations in CACAO are shown in listings 4.1 and z2 respectively.

The function `lock_monitor_wait()` takes 4 arguments. In addition to the ones provided by the Java method `notify()` (`millis` and `nanos`), it needs

```
void lock_monitor_wait(threadobject *t, java_object_t *o,
                       s8 millis, s4 nanos) {
    uintptr_t      lockword = o->lockword;
    lock_record_t *lr;
    int            lockcount;

    if (IS_FAT_LOCK(lockword)) {
        lr = GET_FAT_LOCK(lockword);
        if (!check_illegalmonitorstate(t, lr)) return;
    }
    else {
        if (LOCK_WORD_WITHOUT_COUNT(lockword) != t->thinlock) {
            throw_illegalmonitorstateexception();
            return;
        }
        /* inflate lock */
        lr = lock_hashtable_get(t, o);
        lock_record_enter(t, lr);
        lock_inflate(t, o, lr);
        notify_flc_waiters(t, o);
    }

    /* add a new entry to the waiters list */
    lock_record_add_waiter(lr, t);

    /* remember the old lock count */
    lockcount = lr->count;

    /* unlock this lock record */
    lr->count = 0;
    mutex_unlock(&lr->mutex);

    /* sleep until notified/interrupted/timed out */
    threads_wait_with_timeout_relative(t, millis, nanos);

    /* re-enter the monitor */
    mutex_lock(&lr->mutex);
    lr->owner = t;

    /* remove entry from the waiters list */
    lock_record_remove_waiter(lr, t);

    /* restore the previous lock count */
    lr->count = lockcount;

    if (t->signaled)
        t->signaled = false;
    else check_interrupted_state(t);
}
```

Figure 4.1: wait

pointers to the `threadobject` of the currently executing thread (`t`) and the Java object which *wait* is called on (`o`). First, the lock needs to be inflated if it is not in fat lock mode already. The sequence of operations for inflation is defined by the tasuki lock algorithm. After the inflation, the threads on the FLC waiting list need to be woken up so that they can migrate from the per-thread lock to the newly installed lock record. When the lock record is unlocked as part of the wait operation, one of them will become eligible for acquiring it while the current thread is sleeping.

After that, the current thread enters itself into the lock record's list of waiters, resets its recursion count, preserving the current value, and unlocks the lock record. All these operations can be done safely without any further synchronization because only threads holding the lock may access the involved fields. After unlocking the lock record, the thread blocks itself on its own `waitcond` condition variable.

It returns from this blocked state only when another thread *notifies* this object (via `notify()`) or *interrupts* it (via `Thread.interrupt()`) or when the timeout has elapsed. After waking up, first the lock record needs to be reacquired. After that, the lock record is returned to the state it was in before by restoring the recursion count and owner and removing the current thread from the list of waiters. Finally, if the wake-up was caused by a `notify()` call, indicated by a set `signaled` field, it is reset and the *wait* operation returns normally. Otherwise, it might have been *interrupted.* The helper function `check_interrupted_state` inspects the interruption state, resets it and throws an exception if necessary. The thread might also have been signaled and interrupted; in this case, the interruption flag remains set for subsequent inspection.

notify

The *notify* implementation is pretty straightforward. Because it is used for both *notify* and *notifyAll*, it takes an additional argument `one` specifying whether to notify all waiters or just one. Like *wait*, the *notify* implementation has to check if the monitor is actually owned by the calling thread. The lockword may be in thin-locked state. In this case, there can be no list of waiters because the first thread calling `wait()` would have inflated it.

After that, the list of waiters is traversed. Waiting threads remove themselves from the list after they have been signaled, but before that, they have to reacquire both the `waitmutex` and the lock record. Because they may not yet have had a chance to do so, some threads in the `waiters` list may have their `signaled` flag set already. These threads must be skipped; otherwise the *notify* would get lost.

```
void lock_monitor_notify(threadobject *t, java_object_t *o,
                         bool one)
{
    uintptr_t      lockword = o->lockword;
    lock_record_t *lr;
    list_t        *l;
    lock_waiter_t *w;
    threadobject  *waitingthread;

    if (IS_FAT_LOCK(lockword)) {
        lr = GET_FAT_LOCK(lockword);
        if (!check_illegalmonitorstate(t, lr)) return;
    } else {
        if (LOCK_WORD_WITHOUT_COUNT(lockword) != t->thinlock)
            throw_illegalmonitorstateexception();
        /* No thread can wait on a thin lock, so there's nothing
           to do. */
        return;
    }

    l = lr->waiters;
    for (w = list_first(l); w != NULL; w = list_next(l, w)) {
        waitingthread = w->thread;

        /* Skip threads which have already been notified.
           They will remove themselves from the list. */
        if (waitingthread->signaled) continue;

        mutex_lock(&(waitingthread->waitmutex));
        pthread_cond_signal(&(waitingthread->waitcond));
        waitingthread->signaled = true;
        mutex_unlock(&(waitingthread->waitmutex));

        if (one) break;
    }
}
```

Figure 4.2: notify

4.1.5 Implementation of the SableVM modification

For the initial implementation of tasuki lock in CACAO, an additional word had to be added to the object header just for the FLC bit. This rather wasteful use of object header estate has since been remedied with the SableVM modification which allows the FLC bit to be moved to the `threadobject` structure.

The SableVM modification removes the inflation loop from the lock algorithm. In thin lock and tasuki lock, all threads contending for a thin-locked object try to inflate it. The modified lock algorithm just enters the thread in the FLC waiting list and waits until the unlocking thread inflates the lock. The unlock algorithm inflates all locks in its FLC waiting list at once when it observes a set FLC bit. The list contains tuples (t, o) where t is a thread and o is the object it is trying to lock. Every thread necessarily holds all the locks on objects in its FLC list. This fact makes inflation a simple matter of storing a pointer to a lock record into the lockword. Additionally, a hash table is no longer required as in tasuki lock.

`sable_flc_waiting` (figure 4.3) is called when a thread has tried to thin-lock an object but has not succeeded. Its main purpose is avoiding busy waiting by blocking the thread on a condition variable. It takes as parameters the `lockword` observed when thin-locking failed and the already known parameters `t` and `o`, containing the current thread's `threadobject` and the object which is to be locked.

First, the index of the thread holding the lock is extracted from the lockword and looked up in the global thread list (lines 7–8). It might not be found in rare cases, when the other thread has already ended. In this case, the function just returns. The algorithms correctness does not depend on any amount of blocking. In fact, the function could be removed entirely, turning the whole locking mechanism into an implementation of thin-lock.

When the thread is found, its `flc_mutex` is locked with the intention of modifying its `flc_list`. Then the FLC bit is set, retaining its previous value. After setting the FLC bit, the lockword is inspected again to see if it is still thin-locked by the same thread. Because this needs to happen after setting the FLC bit, a memory barrier is inserted between those two actions (line 13). It might seem strange that a memory barrier is needed inside a region protected by a mutex. It is necessary, though, because the FLC is read without acquiring the mutex, as exemplified by tasuki lock. If the lockword is not in the expected state, the FLC bit is restored and the mutex left. Otherwise, the thread can enter itself into the FLC waiting list. As shown in lines 18–21, a tuple is inserted. Then the thread blocks itself on its own condition variable. After being woken up, it traverses the list to check if it has

```c
static void sable_flc_waiting(uintptr_t lockword, threadobject *t,
                              java_object_t *o) {
    int index, old_flc;
    threadobject *t_other, *current;

    index = GET_THREAD_INDEX(lockword);
    if (!(t_other = threads_lookup_thread_id(index))) return;

    mutex_lock(&t_other->flc_lock);
    old_flc = t_other->flc_bit;
    t_other->flc_bit = true;
    MEMORY_BARRIER();
    lockword = o->lockword;

    if (IS_THIN_LOCK(lockword)
            && (GET_THREAD_INDEX(lockword) == index)) {
        /* Add tuple (t, o) to the other thread's FLC list */
        t->flc_object = o;
        t->flc_next = t_other->flc_list;
        t_other->flc_list = t;
        for (;;) {
            /* Wait until another thread sees the flc bit and
               notifies us of unlocking. */
            pthread_cond_wait(&t->flc_cond, &t_other->flc_lock);

            /* Traverse FLC list to check if we're still there */
            current = t_other->flc_list;
            while (current && current != t)
                current = current->flc_next;
            if (!current) break;
        }
        t->flc_object = NULL;
        t->flc_next = NULL;
    } else
        t_other->flc_bit = old_flc;
    mutex_unlock(&t_other->flc_lock);
}
```

Figure 4.3: SableVM modification

been removed from the list, immediately blocking again in case it has not. This is the usual loop around waiting on condition variables (or monitors in Java), protecting against spurious wake-ups. When the thread detects that it is no longer in the waiting list, it clears the list linkage fields (lines 35–36), leaves the mutex and returns. The FLC bit does not have to be reset in this case—in fact, it would be fatal to do so. The thread which has cleared the FLC list has also taken care of the FLC bit.

```
 1  static void notify_flc_waiters(threadobject *t, java_object_t *o)
 2  {
 3      threadobject *current;
 4
 5      mutex_lock(&t->flc_lock);
 6      current = t->flc_list;
 7      for (; current; current=current->flc_next) {
 8          if (current->flc_object != o) {
 9              /* Only if not already inflated */
10              uintptr_t lockword = current->flc_object->lockword;
11              if (IS_THIN_LOCK(lockword)) {
12                  lock_record_t *lr;
13                  lr = lock_record_new(t, current->flc_object);
14                  lock_record_enter(t, lr);
15                  lock_inflate(t, current->flc_object, lr);
16              }
17          }
18          pthread_cond_broadcast(&current->flc_cond);
19      }
20
21      t->flc_list = NULL;
22      t->flc_bit = false;
23      mutex_unlock(&t->flc_lock);
24  }
```

Figure 4.4: SableVM modification

`sable_flc_waiting`'s counterpart is `notify_flc_waiters` (figure 4.4). It is invoked when a thread observes a set FLC bit during monitor unlocking. It essentially iterates over its list of FLC waiters, waking up every thread on this list (line 17) and inflating all involved objects (lines 12–15). Inflation consists of allocating a new lock record, entering its mutex and then storing a reference to it in the object's lockword. The inflation is safe because the thread is guaranteed to hold all object's monitors on the list.

There are two important points to note about this code. First, the object which has just been unlocked (passed in parameter `o`), is skipped in line 8. It is the only exception to the previous statement—it is not locked by the current thread anymore—and cannot be inflated because another thread might have already acquired it. Second, all other objects on the list are inflated.

This is sensible to do because otherwise no object would ever get inflated. Also, inflating all locks and waking all threads on the list allows it to be discarded entirely, so it doesn't need to be traversed repeatedly. Importantly, it allows the FLC bit to be cleared, causing subsequent unlocking operations to take the fast path. Discarding the entire list is a very efficient operation: just setting the list head to `NULL` has the desired effect, while every thread in the list clears its linkage fields by itself.

Figure 4.5 illustrates the process of inflation. In the upper part of the picture, thread T_A holds thin locks on objects O_1 and O_2, as indicated by the zero shape bit in those objects' lockwords. In its FLC list, several other threads trying to lock those objects have accumulated. When an inflation happens, objects O_1 and O_2 are inflated—they are associated with lock records, and their shape bits are set to 1. All threads on the list are awoken, and the list is discarded. It is still shown in the lower part of the picture because the awakened threads may not be scheduled right away. The list eventually dissolves, while the threads T_B, T_C and T_D migrate to the objects' lock records. It is well possible that a new FLC list has formed in the meantime, shown here containing thread T_E waiting on O_3. This is the reason why awakened threads need to traverse the list and check if they are in it. A simple check for an empty list would not be sufficient and would cause threads to be blocked forever.

In practice, this traversal is irrelevant, performance-wise, because the list will be empty in the majority of cases. However, a pathological case has been constructed using hundreds of threads and an equal number objects. It was possible to force list traversals of lengths equaling about a fifth the number of involved objects and threads. This very rare case can be avoided by keeping track of the list tail at line 20 of `sable_flc_waiting` and skipping the list traversal if the tail has changed while the thread has been sleeping. This improvement will be included in CACAO.

Type-stable memory

The modifications to the lock algorithm make a thread which tries to lock an already thin-locked object acquire the contention mutex `flc_mutex` in the owning thread's threadobject. This change makes it necessary to allocate threadobjects in type-stable memory[14]. In practice this just means that they cannot be deallocated, or at least that deallocation has to be deferred to a safe point where no thread can possibly see a reference to the threadobject in question anymore. In CACAO, threadobjects are added to a free list when their owning threads end; the free list is consumed before allocating new threadobjects.

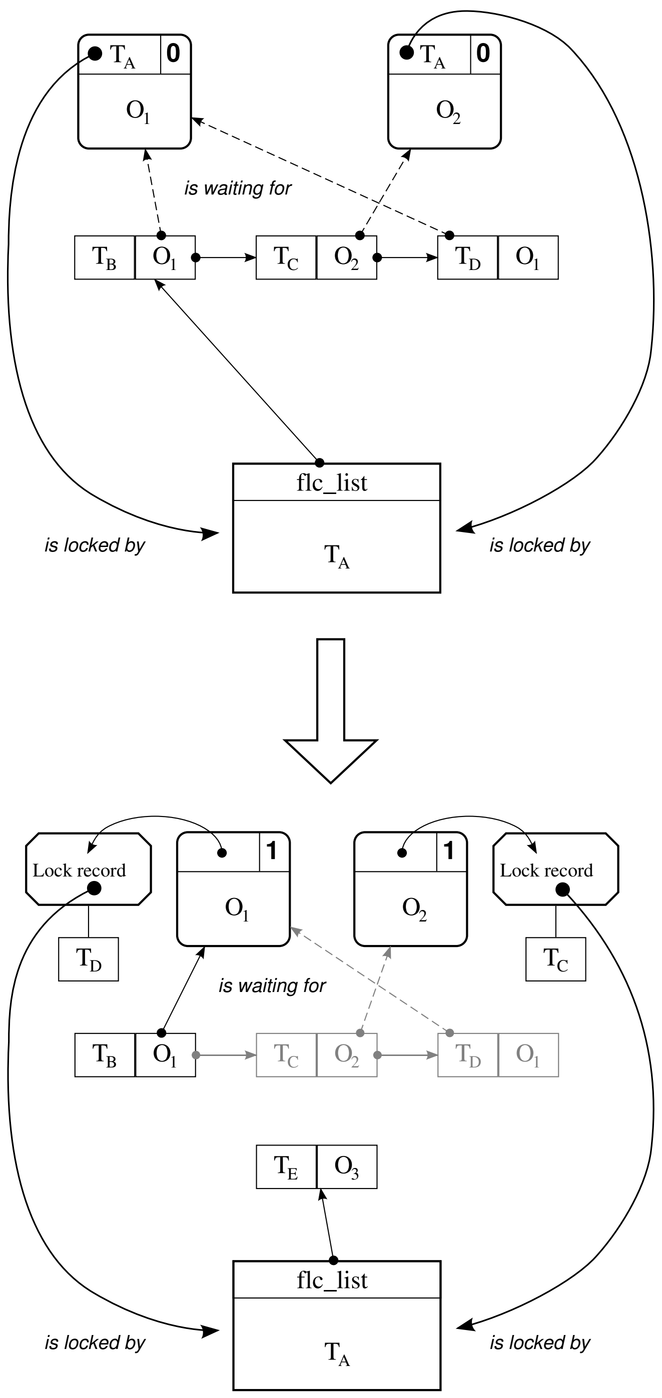


Figure 4.5: The inflation process with the SableVM modification

Similarly, lock records have to be type-stable but this is automatically guaranteed in CACAO because lock records are only removed after all references to their associated objects have disappeared. The same technique cannot be applied to threadobject cleanup, however, because references to threadobjects are stored as integer identifiers in the lockword, not as pointers.

4.1.6 KKO Lock

Only a draft implementation of the KKO locking algorithm has been attempted in CACAO, allowing a rough measurement of its performance characteristics. In their paper, the authors do not mention how they implemented the recursion count. Judging from the description in [16], they used only 10 bits for both `owner` and `other`.

Because the CACAO implementation has been done on an *x86_64* machine, there was plenty of room for the recursion count in the 64 bit lockword (figure 4.6). The `cnt` field is allocated on a 16 bit boundary which allows direct access as a 16 bit word. The reservation flag has been moved to the most significant byte which is always zero for heap addresses on current Linux implementations. This allows storing unmodified pointers to lock records in the lockword.

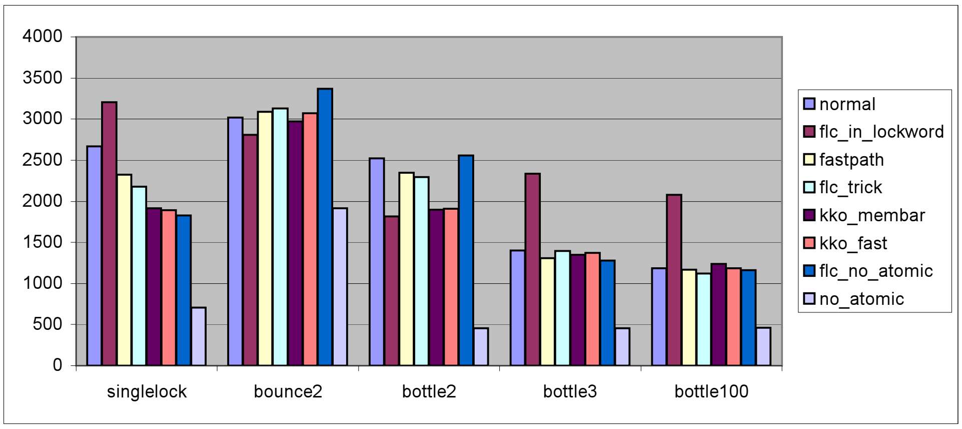
RF	unused		cnt	u	other	owner	shape
1	7	8	16	1	15	15	1

Figure 4.6: Bit allocation for KKO lock

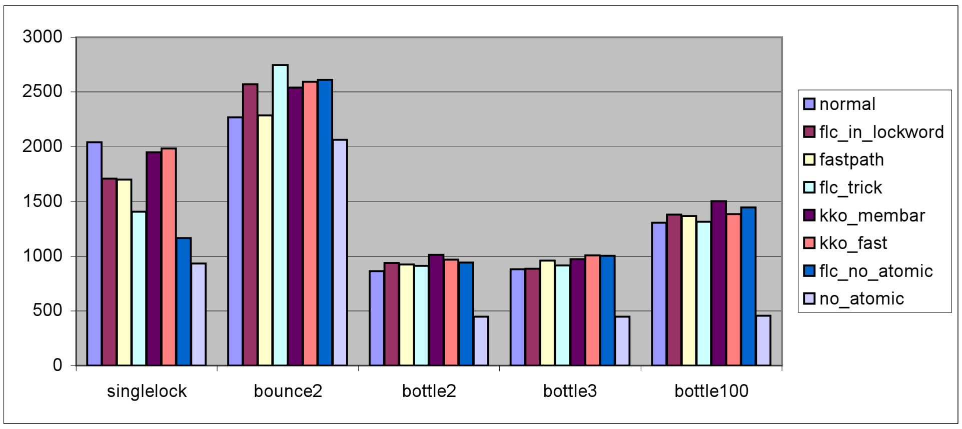
Because this algorithm does not exhibit a convincing performance benefit (shown in the next section) and because of portability concerns, KKO lock has not be implemented in the official version of CACAO.

4.1.7 Performance

It is quite ironic that the performance of a locking implementation—a building block for the implementation of parallel programs—is best measured in single-threaded programs. The locking algorithm's influence on the performance of multi-threaded programs does not differ from its influence on single-threaded programs. This is not very surprising, though, given the similar nature of all algorithms considered in this chapter. They all strive to make the common path fast, and the common path does not involve contention. Therefore, as mentioned before, the number of atomic instructions an algorithm uses in its fast path is the most important performance indicator.



(a) Xeon

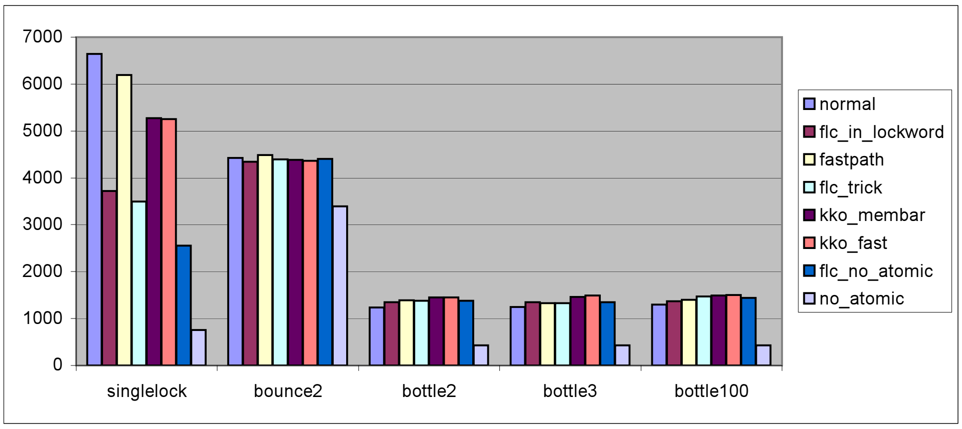


(b) Opteron

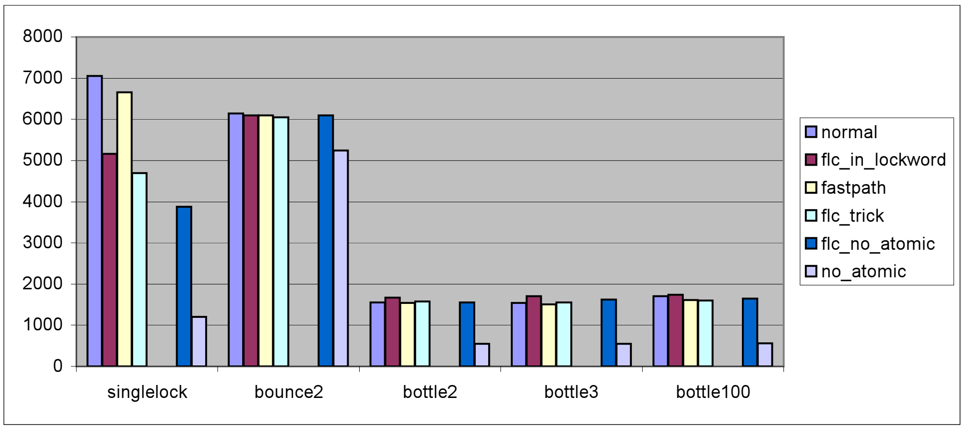
I have set out to compare several variants of the tasuki lock algorithm currently implemented in CACAO. First, the machines participating in the benchmark, the variants and the benchmark programs are described.

Name	Cores	CPU	Clock	Arch.	OS/Linux kernel
Xeon	4	Xeon E5320	1.86GHz	x86_64	Fedora 5 / 2.6.20
Opteron	2x2	Opteron 270	2.0GHz	x86_64	Debian / 2.6.24
Pentium D	2	Pentium D	3.2GHz	x86_64	Fedora 9 / 2.6.25
Pentium 4	HT	Pentium 4	3.0GHz	i386	Fedora 9 / 2.6.25
VMWare	1[1]	Opteron 252	2.6GHz	x86_64	VMWare ESX / 2.6.25
Alpha	1	Alpha 21264	800MHz	alpha	Debian / 2.6.18

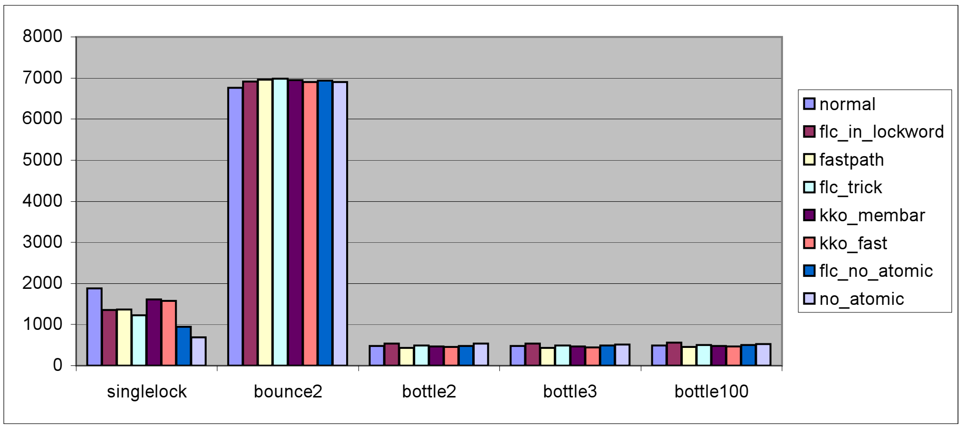
[1]The host machine has 2 CPUs but only one is allocated to the guest.



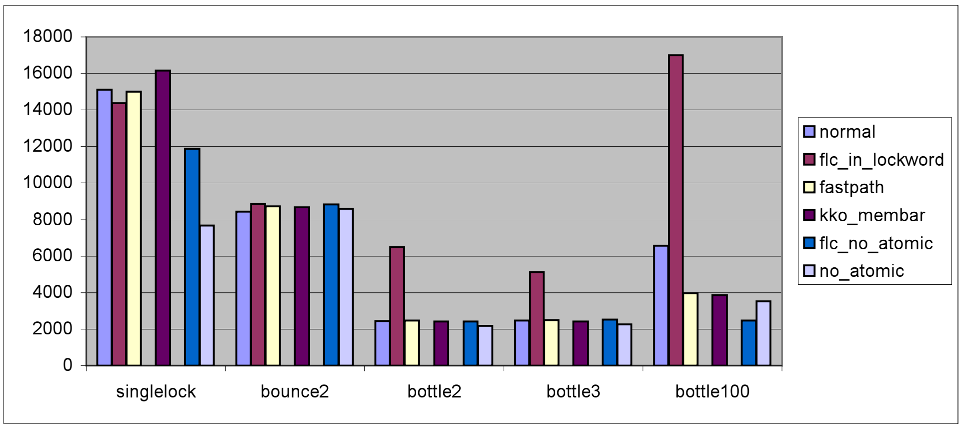
(c) Pentium D



(d) Pentium 4



(e) VMWare



(f) Alpha

Figure 4.7: Lock performance on various CPUs

The variants:

normal This is the baseline implementation of tasuki lock with SableVM extension.

flc_in_lockword The original tasuki lock in CACAO used a separate word in the object header just for storing the FLC bit. In an attempt to remove this space overhead, the first experimental step was to move the FLC bit into the lockword. This requires that the lock be released with an atomic *compare-and-swap*. In this particular implementation, a tight loop was used to retry the *compare-and-swap* until success.

fastpath The functions `lock_monitor_enter` and `lock_monitor_exit` contain quite a bit of code that is rarely executed. Unfortunately, this code increases the whole functions' register usage considerably, resulting in enlarged function prologues and epilogues which contribute a measurable performance penalty. In the **fastpath** variant, the common fast path has been duplicated in a separate function which uses the complete function only as a fallback when the fast path fails. The resulting machine code is quite dense and possibly optimal, leaving almost nothing to gain from an inlined variant. All variants except **normal** use this optimization.

flc_trick Inspired by [9] (see also "Quickly Reacquirable Locks" on page 25), this variant uses the same word tearing technique to avoid the memory barrier before reading the FLC bit in the release path.

kko_membar This variant is an implementation of the KKO lock described in [17] ("KKO Lock" on page 24). It uses a memory barrier between the write and the read of the Dekker algorithm.

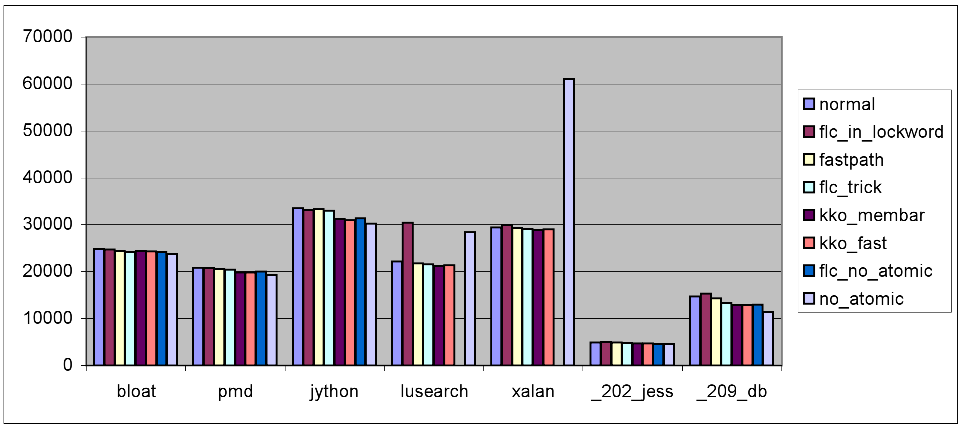


Figure 4.8: Performance of dacapo and SpecJVM98 benchmark programs

kko_fast The same as **kko_membar** but without the memory barrier. It is the same technique as described in [9] and works only on certain hardware memory models. Both KKO variants have only been implemented on 64 bit architectures.

flc_no_atomic In an attempt to measure the cost of the memory barrier before reading the FLC bit, it has been removed entirely in this variant. The algorithm does not work reliably this way but error occurrence is infrequent enough to allow the benchmark programs run correctly most of the time.

no_atomic This variant does not use atomic instructions or memory barriers at all[2] and as such acts as a "speed of light" case—it is the fastest an implementation can get[3]. Because synchronization among multiple processors cannot reliably work this way, this variant also binds the entire CACAO process to a single processor.

The benchmark programs:

singlelock A single-threaded program that acquires and releases a single lock repeatedly and does nothing else.

bounce2 Two threads work on the same lock and hand it over to one another as soon as they have acquired it. This is done by repeatedly calling *wait* and *notify*.

bottle2 An integer counter variable is protected by a lock and two threads

[2]On *x86* and *x86_64*, the *cmpxchg* instruction can be used without a *lock* prefix. In a single processor scenario, it can still be considered atomic—a context switch cannot happen in the middle of an instruction.

[3]Of course there is plenty of opportunity for advanced optimizations like lock coarsening or lock elimination.

continually increment it in a synchronized fashion.

bottle3 The same with 3 threads.

bottle100 The same with 100 threads.

Execution

All programs have been run 50 times, and the average runtime reported by the test harness was used as the result. The machines were, for the most part, only lightly loaded. The different variants have been run in a round-robin fashion so a background job of a few minutes would not skew the results significantly.

Interpretation

The results discussed here are shown in figures 4.7(a)–4.7(f).

singlelock The singlelock results bear no surprises on the Xeon. With the exception of *flc_in_lockword*, the variants are expected to perform increasingly better from left to right. *flc_in_lockword* is supposed to be slower because it executes *compare-and-swap* twice as often. The *no_atomic* variant is noticeably faster, indicating that synchronization mechanisms are still very costly on this platform.

The Opteron behaves quite differently. Interestingly, *flc_in_lockword* (with 2 *CAS* instead of 1) is just as fast as the tasuki lock algorithm (*fastpath*) with a memory barrier. The KKO “optimization” is not an improvement at all on this machine. The gap between the synchronized variants and *no_atomic* is a lot smaller than on the Xeon.

The Pentium D results look interesting. The 2-*CAS* variant is a lot faster than the one with one *CAS* and one memory barrier. It appears as if the memory barrier instruction was particularly expensive on this processor. However, the *kko_fast* variant—which contains no memory barriers at all—does not support this impression. The Pentium 4, which is also based on the Netburst architecture, exhibits the same behavior, except that the experimental KKO implementation could not be tested because it does not work on this 32 bit architecture. It is also interesting to note that both the Pentium D and the Pentium 4 can be run in either uni-processor or SMP mode (Hyperthreading on the Pentium 4). There is absolutely no difference between the two modes, performance-wise.

I have also run the benchmarks on a virtual machine inside VMWare ESX on an Opteron, and the performance is exactly the same as on the raw machine itself.

Finally, I tested the performance on an Alpha processor. There is not much difference between the individual variants. Again, KKO is not a win on this machine, and the *no_atomic* variant is only moderately faster. The variants with “soft” memory barriers have been omitted because they make no sense on the Alpha with its extremely relaxed memory model.

bounce2 This benchmark measures mostly the context-switching time. The *no_atomic* variant does so well in this benchmark because it only needs to do an intra-processor context switch. On the VMWare and Alpha, the two uni-processor machines, this does not make a difference. The VMWare benchmark shows that the virtualization layer makes context switching much more expensive than it already is.

bottle*n* The bottleneck benchmarks don’t reveal particularly interesting insights. The *no_atomic* performance is again very good because there is only one processor involved. It is interesting to note that the number of competing threads does not make any difference to the overall throughput[4]. Again, most of the variants perform equally well with the notable exception of *flc_in_lockword*, especially on the Alpha[5]. The cause for the serious slowness seems to be an unfortunate live-lock-like situation. The Alpha already needs to use a *LL/SC* loop to perform the *compare-and-swap*, and the outer loop of the *flc_in_lockword* variant seems to interfere badly with this.

In order not to give too much importance to the results of micro-benchmarks, I have also run some dacapo and SpecJVM98 benchmarks on one machine (the quad-core Xeon). The benchmarks used have a relatively high number of monitor operations. Figure 4.8 shows that the overall picture is not different from the micro-benchmark runs. The bad performance of *no_atomic* in the *lusearch* and *xalan* benchmarks comes from the fact that they profit from multiple cores but this variant can only utilize one. The bad *lusearch* performance also shows that the *loop-around-compare-and-swap* is not a good idea in programs with heavy contention.

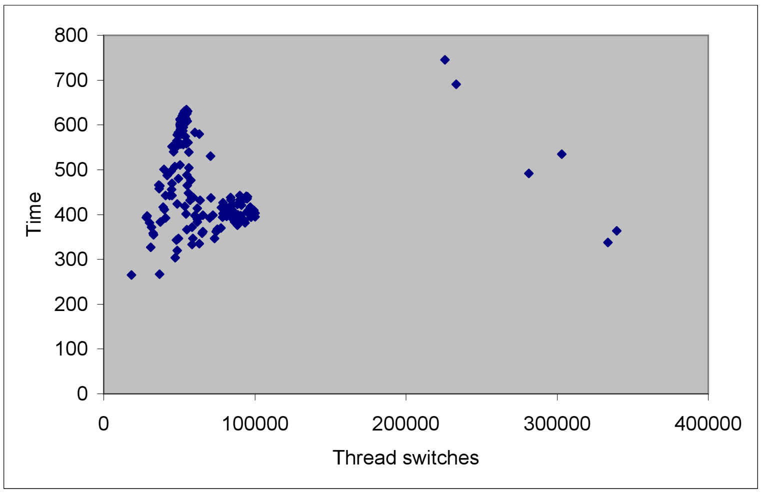


Figure 4.9: Throughput vs. context switches

Throughput vs. granularity

Measuring the performance of an algorithm for mutual exclusion is tricky and not very meaningful if there is a high degree of contention. For example, the *bottle2* benchmark would show the highest performance if only one thread ever acquired the lock. An algorithm which accomplishes this would just not be very useful. The other extreme is represented by the *bounce2* benchmark, where every thread is allowed to run only for a tiny amount of time before it has to give up the lock for its peer. Figure 4.9 shows the relationship between throughput and the number of context switches. Opaque scheduler decisions introduce a great amount of variability here. Although a higher number of context switches shows in lower performance, this relationship is less pronounced than I had expected. There are also extreme outliers which happen occasionally for no apparent reason.

4.2 Internal CACAO Synchronization

CACAO needs to manage several internal data structures which are potentially accessed by multiple threads. These are mainly a few hash tables, lists and trees. Obviously, they have to be protected against destructive concurrent access. No fancy mechanisms are at work in these areas. A thread which needs access to such a structure (both read and write access) locks it before

[4]Except on the Opteron where other programs were keeping 3 out of the 4 cores busy for the duration of the benchmark.

[5]The *bottle100* result is actually scaled so as not to distort the diagram. The actual performance was much worse.

the access and releases the lock when it is done. These locks are only held for very brief periods of time, and there is no instance of nested locking. CACAO uses the same locking mechanism for those internal locks as it uses for Java locks. Additionally, some internal threads, like the finalizer thread, make use of the *wait/notify* capability.

4.2.1 Freedom from deadlock

Figure 4.10 is the result of a tracing run which shows the nesting of locks taken by CACAO itself[6]. In terms of deadlock safety, leaf nodes, shown in gray color, are not interesting; they can never cause a deadlock. After removing the leaf nodes, the picture becomes much clearer, as shown in figure 4.11.

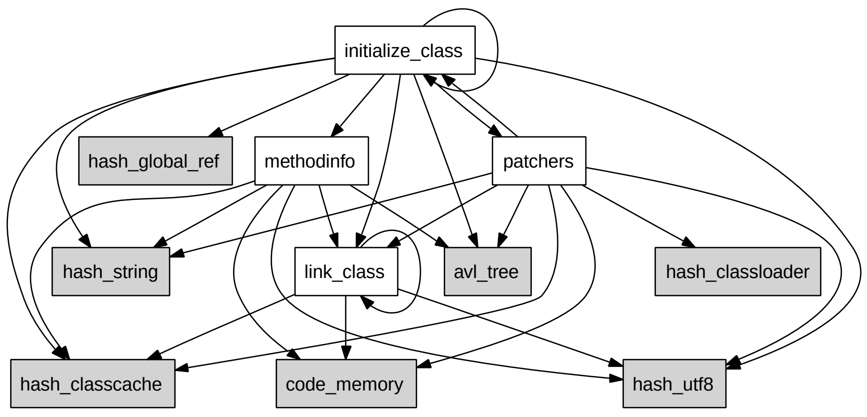


Figure 4.10: All locks taken by the CACAO JVM

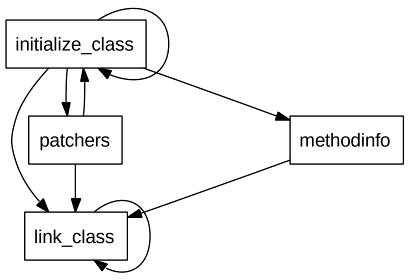


Figure 4.11: Lock graph with leaf nodes removed

Cycles in the lock dependency graph bear a potential for deadlock. A closer

[6]Because of layout problems, two additional leaf nodes have been omitted from the graph. They are used for native library loading and insubstantial to this discussion.

examination of the remaining cases shall show that no deadlock can happen.

`link_class` This lock is taken when a class is linked. It is actually not a single lock but one for each class. The cycle exists because `link_class` recursively links all superclasses before the class itself can be linked. Because the superclass relation is acyclic, this node is safe and can be treated like a leaf node. This, in turn, makes `methodinfo` a leaf node and leaves just two nodes.

`initialize_class` This lock is similar to the `link_class` case. Again, superclasses are initialized before the class itself. That's where the cycle with itself comes from.

`patchers` This is a global lock which protects a list of items to be patched on demand. The code generator adds entries to this list for a variety of reasons; in the case of interest here, a trap is added to code that accesses not-yet-initialized classes. At runtime, the trap will cause a signal handler invocation which will then traverse the list of patchers and initialize the class. Before the list traversal, the `patchers` lock is taken. Initializing the class entails locking `initialize_class`.

The only edge left to explain is the one from `initialize_class` to `patchers`. When a class is initialized, the class constructor needs to be called. This class constructor consists of Java bytecode which needs to be compiled. During code generation, patchers might need to be added. That is why the `patchers` lock is taken. The cycle cannot be dismissed yet, because one thread (inside a trap handler) might take the `patchers` lock first and then `initialize_class` while another thread might do it exactly the other way round. This can not happen, though, because `initialize_class` is again not a single lock but one for each class. But a single class will never be initialized twice. Thus, only one thread will try to acquire the `patchers` lock, and no deadlock can arise. Therefore, the internal lock structure in CACAO is free from deadlock.

4.2.2 Statistic counters

CACAO also features a fairly large number of internal counters for purpose of statistics only. These are stored in global variables for historical reasons. Incrementing such a counter is not done in a thread-safe way, so occasionally some increments might be lost. As the numbers gained from these counters rarely need to be exact, this has not yet become a problem. It would be fairly easy to use atomic increment instructions for updating those counters or to convert them into thread-local variables and accumulate them in the main thread when a thread exits.

Chapter 5

Subtype checking

The JVM provides the two bytecode instructions `CHECKCAST` and `INSTANCEOF` for determining if an object instance is a sybtype of a particular class or supports a given interface. Both bytecodes basically provide the same functionality. The difference is that `CHECKCAST` throws an exception if the subtype relation is not met while `INSTANCEOF` produces a boolean result. `CHECKCAST` operations happen quite often, so it is important to make them fast.

`CHECKCAST` and `INSTANCEOF` work not only on Java classes but also on interfaces and arrays. Because these types have always been handled in a thread-safe way in CACAO, this chapter only deals with checks against normal classes. Figure 5.1 lists execution counts for various forms of those instructions in the programs from the dacapo and the SpecJVM98 benchmark suites. The two columns labeled with "classes" contain the number of checks against normal classes, separately counting `CHECKCAST` and `INSTANCEOF`. The next two columns contain checks against interfaces. This chapter does not describe how these are handled in CACAO. Because of layout restrictions, numbers larger than 1 million have been shortened, using factor k = 1000. The last two columns contain the overall percentage of `CHECKCAST` instructions and of subtype checks against normal classes respectively. If we name the four number columns c_0, c_1, c_2 and c_3 and $s = \sum c_{0\ldots3}$, then the penultimate column results from $\frac{c_0+c_2}{s}$, while the last one contains $\frac{c_0+c_1}{s}$. For most programs, the percentage of class checks constitutes the vast majority of subtype checking operations. The "bloat" benchmark is a notable outlier in this regard, performing a huge number of interface checks. Programs with lots of class check operations, like "jess" and "db" will be particularly interesting for performance measurements in this chapter.

	classes		interfaces			
Program	`CHECKC.`	`INST.OF`	`CHECKC.`	`INST.OF`	%`CHECKC.`	% class
compress	1,947	1,376	48	2	59.1%	98.5%
jess	16,888 k	9,129 k	701,405	18	65.8%	97.4%
db	56,125 k	2,915 k	48	2	95.1%	100.0%
mpegaudio	50,947	1,387	48	2	97.3%	99.9%
jack	5,201 k	1,066 k	48	2	83.0%	100.0%
antlr	1,982 k	841,404	221,341	220,488	67.5%	86.5%
bloat	5,418 k	4,188 k	140,777 k	521,062	96.9%	6.4%
fop	4,036 k	1,653 k	72,711	17,745	71.1%	98.4%
hsqldb	26,523 k	350,444	20,046	19,829	98.6%	99.9%
jython	30,516 k	24,031 k	180,999	20,671	56.1%	99.6%
luindex	16,996 k	3,195 k	515,946	541,796	82.4%	95.0%
lusearch	13,832 k	1,380 k	148,698	148,550	90.1%	98.1%
pmd	10,789 k	5,832 k	13,139 k	4,029 k	70.8%	49.2%
xalan	33,983 k	10,497 k	3,092 k	303,769	77.4%	92.9%

Figure 5.1: Execution counts for subtype testing instructions

5.1 Tree numbering

CACAO enters all classes into a type tree when they are loaded [32]. In this tree, the nodes are numbered according to pre-order traversal. By storing the sequence number and the number of sub-nodes in each node, the result for `CHECKCAST` can be easily determined. The simple predicate `is_subtype_of` is true if and only if class S is a subtype of class T.

`is_subtype_of(S, T) := (S.baseval - T.baseval)` $\leq$ `T.diffval`

Every time a class is loaded or unloaded, the whole type tree needs to be traversed and renumbered. This interferes badly with multi-threaded programs because the 3 numbers required for the subtype test need to be read atomically. In a conservative approach, this atomicity could easily be obtained by using a monitor. Alas, the frequency of `CHECKCAST` is prohibitive for this kind of usage; monitor operations incur too great an overhead. Not only do they require at least one atomic instruction but they get translated into calls into VM code which degrades register allocator performance.

Surprisingly, the second claim turns out to be not true with CACAO's current simplistic register allocator. A modified allocator that treats `MONITORENTER` and `MONITOREXIT` instructions like function invocations, making unavailable temporary and (for leaf methods) argument registers, causes no measurable slowdown. Because this surprising effect might be attributed to the *x86_64* architecture's general register paucity, it has also been reproduced on an

Alpha processor which sports a larger number of available registers. A better register allocator would very likely expose the performance degradation, however.

In contrast, the first claim is true and measurable. Figure 5.2 shows how the addition of a single *compare-and-swap* instruction at every `CHECKCAST` and `INSTANCEOF` instruction causes a noticeable slowdown, especially in programs with many `CHECKCAST` instructions like "db".

Program	normal	with CAS	% Slowdown
compress	5.11 s	5.15 s	0.93%
jess	5.75 s	6.27 s	9.06%
db	14.48 s	16.35 s	12.87%
mpegaudio	5.30 s	5.34 s	0.61%
jack	6.75 s	6.99 s	3.58%
antlr	5.59 s	5.74 s	2.53%
bloat	29.03 s	29.23 s	0.71%
fop	4.14 s	4.21 s	1.73%
luindex	17.52 s	18.10 s	3.32%
pmd	23.73 s	24.05 s	1.39%

Figure 5.2: `CHECKCAST` and `INSTANCEOF` with atomic instructions

5.1.1 Unsafe regions

This problem can be solved by using a technique similar to the one described in [17]. The machine code for `CHECKCAST` does not have to do any synchronization operations, the burden instead lies entirely on the thread doing the tree numbering. It has to stop all other threads and inspect their program counters. If a thread is interrupted just inside an *unsafe region*, its program counter is reset to a restart point. That way, the entire operation can be retried (and likely succeed) after the renumbering is done.

This requires the `CHECKCAST` machine code to be carefully prepared for this kind of start-over usage. In particular, it cannot touch anything apart from the temporary registers. This is not too difficult to achieve in CACAO because each bytecode is translated to machine code more-or-less independently of all others. Thus, the restarting point can be placed right at the beginning of the code for the `CHECKCAST` ICMD, while the *unsafe region* needs to surround only the three loads.

The practical implementation is quite problematic, however. The Boehm garbage collector currently used by CACAO has the required functionality built in. It can stop all threads for its own purposes but it is not helpful in exposing this mechanism to the outside. The garbage collector's stop-the-world code needs to be patched for this kind of usage. Because of the maintenance burden, this approach is not favored in the CACAO implementation.

5.1.2 Signal handler limitations

Another serious problem is introduced with this approach. Signal handlers in POSIX are severely limited in what they are allowed to do. This restriction is necessary because of the asynchronous nature of signals. They can occur at any time, most notably during the execution of system library functions. Because of this, and because it is impractical for some functions to allow reentrancy, asynchronous signal handlers may only call functions from a list of explicitly allowed functions. This list does not include `pthread_mutex_lock`, for good reason.

The signal handler for checking *unsafe regions* needs to resort to some global data structure containing all currently registered *unsafe regions*. This data structure can be accessed by any thread and is therefore protected by a lock. In order to access the list of *unsafe regions*, the signal handler would need to acquire the lock. Which is precisely what a signal handler is not allowed to do.

Let's recall that the reason for this restriction is that the signal handler might be interrupting a system function that is not marked as safe for asynchronous reentrant invocation. So, if the signal handler can determine whether it has been called from JIT code or from native code, the problem is solved because if called from native code, which does not contain *unsafe regions*, the signal handler need not do any further processing at all, and if called from JIT code, it cannot be interrupting a system function and may therefore safely call the lock function to access the list of *unsafe regions*. Alas, this information is currently not available in CACAO.

This can be worked around by carefully recording the lower and upper bounds of memory regions containing JIT code and checking the instruction pointer against these but this does not work reliably on 32 bit architectures. It works on Linux *x86_64* but the method's reliability is questionable at best since there are no guarantees as to which addresses `mmap`-ed memory regions might happen to fall on.

Because the signal handler needs only read access to the data structure, the problem could also be solved by using a structure safe for reading in the presence of writes. In (purely) functional programming languages, such constructs are rather common, and a purely functional tree could be built in C with some help from the garbage collector. This would be a quite complicated undertaking, though, and might not be worth the effort. Also, portability would need to be taken into consideration; on some processors memory barriers would be required.

5.1.3 Improved renumbering algorithm

Palacz and Vitek [29] use a renumbering algorithm designed so that the tree is consistent at all times. The algorithm assigns a *low* and a *high* value to each type in the tree and allows for not-yet-numbered types, such as recently loaded classes, which have their *high* value set to 0. *Low* and *high* values for class C will be called *C.low* and *C.high*, respectively, while *C.low'* and *C.high'* denote values from a later generation (after one or more renumbering passes).

Unfortunately, the consistency provided by their algorithm does not help, because, while snapshots of the tree are in fact consistent, there is no practical way to observe such a snapshot without the use of locking constructs. The subtype test requires three distinct values[1] to operate on, and they cannot all be read at the same time. Because of this, the thread doing the test might see some values from before an update and some others from after the update or even from a much later generation.

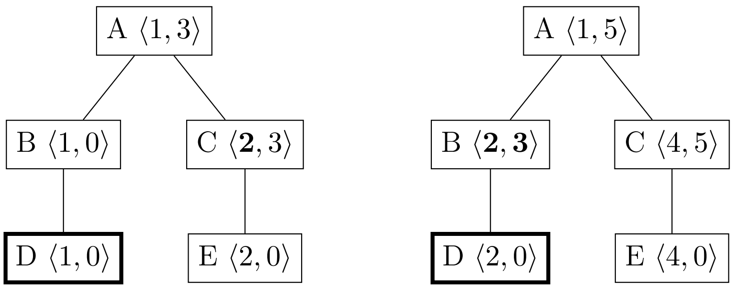


Figure 5.3: Type tree before and after renumbering

This can easily lead to a situation where the subtype test produces a wrong result. Figure 5.3 shows a simple type tree to which D has been added recently. The numbers in angle brackets denote the *low* and *high* values respectively. The left tree shows the assigned ranges before D's promotion, while the right tree shows the situation after a renumbering pass. If, during the renumbering, a thread were to find out if C is a subtype of B, it might see the value *C.low* $= 2$ (from the old generation) and the values *B.low'* $= 2$ and *B.high'* $= 3$ (from the new generation) and conclude accordingly that C is a subtype of B even though this is clearly not the case.

[1]In the paper, the authors seem to hint at a representation where two values are packed into a single word. But even then, two reads from two distinct memory addresses are required.

The same could have happened with a naïve renumbering of the tree, so Palacz's algorithm is not an improvement at all.

5.1.4 Another approach at improving the renumbering

Another approach has been discussed for CACAO. It would aim at minimizing the number of times needing a renumbering of the class tree by choosing a better distribution of the class values. However, it would still not solve the original problem of having to read three values atomically. The real problem is stopping an unsynchronized thread from producing bad results, and while it would have to be done less frequently, it would still be necessary nonetheless.

5.1.5 Embedded generation count

CACAO's original algorithm can be preserved by embedding a generation count in each `baseval` and `diffval`. The JIT code then has to verify that all three values are from the same generation. This leads to quite a bit of computational overhead, and this method's only right to exist may be that it can be implemented without any portability concerns.

5.1.6 Performance

In terms of run-time cost in the absence of class loading, the subtype test based on tree numbering needs to perform 3 loads and 2 arithmetic instructions. Notably, it does not require a branch instruction for the `INSTANCEOF` instruction. In real code, the benefit of this property is questionable, though, because in almost all cases, the `INSTANCEOF` test is followed by a conditional branch bytecode instruction anyway. Also, the more frequent `CHECKCAST` instruction does not share this advantage.

5.2 Hotspot's fast subtype checking method

Hotspot uses a different method[6] for subtype checks which does not have to deal with any kind of synchronization. Once the supporting data structures associated with a particular class are in place, they remain constant over the entire lifetime of that class. This comes at the cost of some additional pointers in the class structure and slightly enlarged machine code. The average runtime cost is reduced, though, as the common path consists, in addition to a compare and branch, of only a single load instruction in contrast to the three loads and one subtraction the tree numbering method weighs in at.

Hotspot uses Cohen's algorithm[7] and its supporting data structure, the so-called display. The display for a particular class is an ordered list of all its superclasses. It is sorted in a way so that the least specialized classes appear first or that for every class, all its superclasses appear prior to the class itself. This automatically places `java.lang.Object` in the first position of every display.

In Hotspot, the fast subtype check is used for checking all Java objects, including arrays and interfaces. Historically, these have been handled differently in CACAO in a way that does not suffer from synchronization issues to begin with. The new subtype implementation in CACAO consequently replaces only the tree numbering method and leaves all array and interface checking in place.

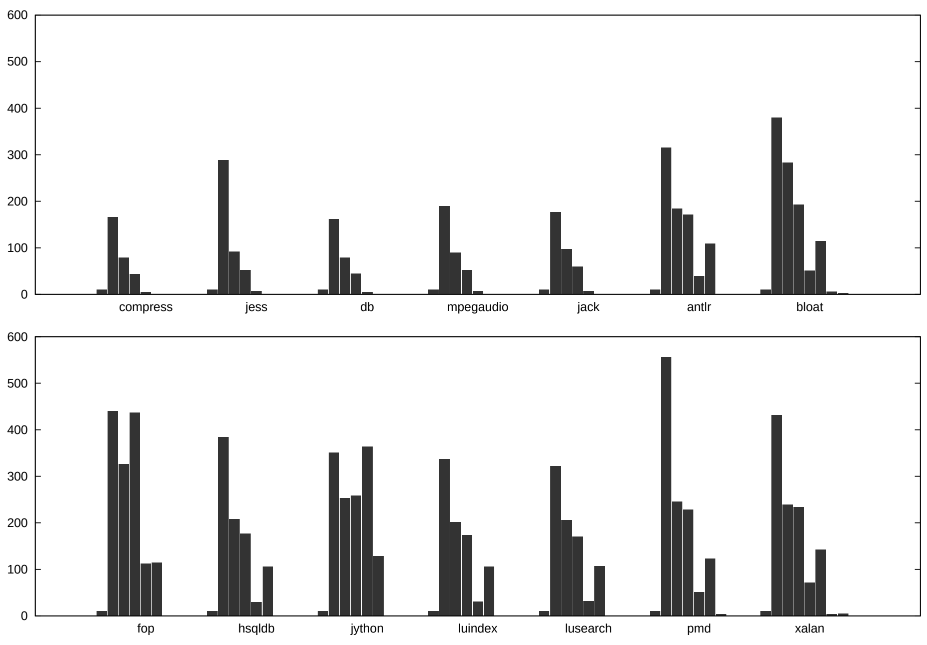


Figure 5.4: Display depth distributions

The type checking is implemented as described in [6]. In particular, the display for a class is stored in an array of fixed size inside the class run-time structure. If the display does not fit in this fixed-size array, it is stored in an overflow array allocated separately on the heap. The method's high performance depends on the fact that most displays are relatively short, so the overflow array is rarely needed. Figure 5.4 shows display length distributions for various benchmark programs. Many programs have a spike at depth 2 and use almost no classes with a depth larger than 6.

The choice of a suitable array length for storing the restricted display proves to be less straightforward than expected. Larger numbers lead to larger class metadata and smaller code. The increase in class metadata should be self-explanatory, while the decrease in code size is caused by more frequent omissions of the array scanning loop. Array scanning is only required for classes with a display depth larger than the length of the restricted display array. It is also needed for unresolved classes, but the relative frequency of those cases is not affected by max depth choice.

It should be expected that larger values would result in higher performance. In most cases, though, this effect could not be measured with any statistical significance. The reason for this is a surprisingly large number of checks against types types of depth 2. Figure 5.5 shows the relative frequencies of class checks against each display depths 2–4. Most benchmark programs tested show a significant peak at depth 2. The “jython” benchmark stands out in that it is the only one with a measurable performance increase as the max display depth is increased from 2 to 3. Other programs with a high relative frequency of checks against larger depths, like “mpegaudio” and “bloat”, do quite few class checks in general, so they do not exhibit this behavior.

Program	Depth 2	Depth 3	Depth 4	Depth 5+
compress	98.9%	0.9%	0.1%	0.1%
jess	88.0%	12.0%	0.0%	0.0%
db	100.0%	0.0%	0.0%	0.0%
mpegaudio	6.2%	39.4%	54.4%	0.0%
jack	86.6%	12.3%	1.1%	0.0%
antlr	75.0%	15.8%	3.7%	5.5%
bloat	59.0%	17.0%	16.6%	7.4%
fop	92.0%	0.5%	7.1%	0.4%
hsqldb	99.7%	0.3%	0.0%	0.0%
jython	31.7%	55.4%	0.3%	12.7%
luindex	88.0%	11.9%	0.1%	0.0%
lusearch	93.9%	6.1%	0.0%	0.0%
pmd	88.8%	10.6%	0.5%	0.0%
xalan	76.4%	3.9%	17.3%	2.5%

Figure 5.5: Display depth distribution of checked supertypes

5.2.1 Code and data size

The machine code for display-based subtype checking is slightly larger than its tree numbering equivalent. Unlike the latter, it can vary in size, though. In

Java bytecode, `CHECKCAST` and `INSTANCEOF` instructions take two arguments, S and T, where S is taken from the run-time stack while T is a constant stored in the class file. Therefore, at compile time, T is always a known type. However, it is quite possible that class T has not been loaded at that point; hence, it is not always known if type T denotes a class or an interface type.

In those cases when it is not known, the compiler generates code for both scenarios and prepends a run-time check and appropriate branching instructions. The display-based code leaves all this in place but it derives an additional benefit from a known type T. As mentioned earlier, a known T allows the array scanning code to be omitted in most cases. Furthermore, because the reference to T is really just a constant pointer, it can be embedded directly in the instruction stream as an immediate value on *x86* and *x86_64* architectures.

Program	code size (KB)		data size (KB)		instructions	% known
compress	1.3	(0.36%)	20.6	(21.15%)	76	57.9%
jess	1.5	(0.31%)	28.1	(16.97%)	155	63.9%
db	1.3	(0.35%)	19.7	(19.78%)	94	62.8%
mpegaudio	1.3	(0.20%)	24.1	(21.61%)	81	60.5%
jack	-0.3	(-0.07%)	13.9	(11.46%)	267	79.4%
all jvm98	-0.2	(-0.02%)	26.9	(12.61%)	370	77.6%
antlr	0.8	(0.09%)	40.3	(13.72%)	890	86.7%
bloat	11.4	(0.87%)	5.6	(1.27%)	1528	81.0%
fop	2.7	(0.20%)	81.9	(14.41%)	620	71.6%
hsqldb	-2.4	(-0.21%)	38.8	(11.25%)	673	80.8%
jython	5.1	(0.35%)	54.4	(4.38%)	1272	75.1%
luindex	1.2	(0.15%)	54.0	(19.37%)	312	69.9%
lusearch	1.5	(0.20%)	54.9	(19.82%)	274	66.1%
pmd	8.4	(0.71%)	57.9	(7.45%)	1031	58.3%
xalan	2.7	(0.23%)	58.7	(8.36%)	662	69.8%
all dacapo	13.7	(0.27%)	48.4	(1.17%)	5337	78.8%

Figure 5.6: Code and data size measurements

Figure 5.6 shows how the new subtype checking method affects code and data sizes. The code size column shows the increase of code size in kilobytes and the percentage of all generated code. The data size column shows the increase in class metadata, including the data used for tracking *unsafe regions*. The column "instructions" contains the number of times a `CHECKCAST` or `INSTANCEOF` instruction has been generated. The last column shows the percentage of checks against known classes. These are the cases which allow array scanning omission if the depth is not too large. These known classes of sufficiently short display depth are also the ones responsible for the negative numbers in the code size column. The new code is shorter in those cases, comprising only one load and one compare instruction.

Code size can vary slightly in multi-threaded programs due to slight timing differences. Different threads can load/resolve classes and compile method code simultaneously, causing the number of known classes to be slightly different between identical invocations.

It might be possible to reduce the data size a bit further. Some fields in the class data occupy more space than strictly needed, due to alignment. Also, some architecture specific changes are still possible. For example, on *x86_64*, where a byte value can be loaded from memory and expanded into a full register value in a single instruction, the offset field could be reduced to a single byte. On a similar note, the machine code is not at its minimum, either. With the current infrastructure, it is difficult to emit branch instructions with one-byte displacement, so most of the branch instructions use four bytes. Again, this is an optimization specific to *x86_64* (and *x86*). Other architectures may open up similar opportunities for optimization.

5.2.2 Performance

Figure 5.7 shows the run-time performance of several benchmark programs. The code for *unsafe region* checking was not enabled in the tree numbering run. The display length has been set at 4. Performance has been tested on two *x86_64* machines. On the older Pentium D, the new code performs significantly better. On the newer Xeon, the improvement is less marked. The measured times display considerable fluctuations—for some runs, not all benchmarks show a consistent improvement; on average, the display-based code performs slightly better though.

5.2.3 Conclusion

The display based method for subtype checking used in Hotspot uses data structures which have to be built only once per class and stay constant as long as the particular class is loaded, i.e. as long as they might be in use. It is therefore inherently thread-safe. Run-time performance is slightly but measurably increased in popular benchmark programs at the expense of an almost negligible increase in code size and a moderate increase in class metadata size. It does not rely on fragile, CPU-specific tricks and is thus easily portable.

CPU: Intel Pentium D 3.2GHz x86_64 UP

Program	Tree Numbering	Display-based	% Improvement
compress	5.15 s	5.11 s	0.79%
jess	6.60 s	6.30 s	4.55%
db	16.37 s	16.15 s	1.32%
mpegaudio	5.04 s	4.99 s	0.87%
jack	9.19 s	8.39 s	8.64%
antlr	6.02 s	5.77 s	4.28%
bloat	33.12 s	32.01 s	3.37%
fop	4.80 s	4.63 s	3.67%
hsqldb	9.62 s	8.43 s	12.39%
jython	45.11 s	40.46 s	10.30%
luindex	20.13 s	19.55 s	2.87%
lusearch	42.83 s	42.43 s	0.95%
pmd	27.56 s	25.21 s	8.51%
xalan	78.94 s	74.68 s	5.39%

CPU: Intel Xeon E5320 1.86GHz x86_64 quad-core

Program	Tree Numbering	Display-based	% Improvement
compress	5.13 s	5.12 s	0.09%
jess	5.82 s	5.59 s	3.81%
db	14.63 s	14.56 s	0.45%
mpegaudio	5.45 s	5.28 s	3.03%
jack	6.80 s	6.71 s	1.34%
antlr	5.70 s	5.64 s	1.20%
bloat	29.30 s	29.10 s	0.69%
fop	4.09 s	4.02 s	1.73%
hsqldb	7.04 s	6.31 s	10.39%
jython	36.40 s	34.56 s	5.04%
luindex	16.93 s	16.53 s	2.37%
lusearch	24.92 s	24.96 s	-0.17%
pmd	23.59 s	23.03 s	2.37%
xalan	57.51 s	56.73 s	1.37%

Figure 5.7: Performance of display-based subtype checking

Bibliography

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996.

[2] Ole Agesen. GC points in a threaded environment. Technical report, SMLI TR-98-70. Sun Microsystems, Mountain View, CA, USA, 1998.

[3] Ole Agesen, David Detlefs, Alex Garthwaite, Ross Knippel, Y. S. Ramakrishna, and Derek White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[4] David F. Bacon, Ravi B. Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: Featherweight synchronization for Java. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[5] Hans Boehm. Getting C++ threads right . `http://www.hpl.hp.com/personal/Hans_Boehm/misc_slides/c++threads.pdf`.

[6] Cliff Click and John Rose. Fast subtype checking in the Hotspot JVM. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 96–107, New York, NY, USA, 2002. ACM.

[7] Norman H. Cohen. Type-extension type test can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4):626–629, 1991.

[8] David Dice. Implementing fast java monitors with relaxed-locks. In *JVM'01: Proceedings of the JavaTM Virtual Machine Research and Technology Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 13–13, Berkeley, CA, USA, 2001. USENIX Association.

[9] David Dice, Mark Moir, and Bill Scherer. Quickly reacquirable locks. `http://home.comcast.net/~pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf`.

[10] Edsger W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, New York, 1968.

[11] Hubertus Franke, Rusty Russell, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Proceedings of the 2002 Ottawa Linux Summit*, pages 479–495, 2002.

[12] Etienne Gagnon. *A portable research framework for the execution of Java bytecode*. PhD thesis, McGill University, Montreal, Que., Canada, Canada, 2003. Adviser-Laurie J. Hendren.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Professional, 2005.

[14] Michael Greenwald and David R. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Operating Systems Design and Implementation*, pages 123–136, 1996.

[15] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[16] Kiyokuni Kawachiya. *Java Locks: Analysis and Acceleration*. PhD thesis, Keio University, 2005.

[17] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. *SIGPLAN Not.*, 37(11):130–141, 2002.

[18] Andreas Krall and Mark Probst. Monitors and exceptions: how to implement Java efficiently. *Concurrency: Practice and Experience*, 10(11–13):837–850, 1998.

[19] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, 1987.

[20] Leslie Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[21] Doug Lea. The JSR-133 cookbook for compiler writers . `http://gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[22] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[23] Jeremy Manson, William Pugh, and Sarita Adve. Java Specification Request 133: Java memory model and thread specification.

[24] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA '02: Proceedings of the fourteenth annual*

ACM symposium on Parallel algorithms and architectures, pages 73–82, New York, NY, USA, 2002. ACM.

[25] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, 2004.

[26] Tamiya Onodera and Kiyokuni Kawachiya. A study of locking objects with bimodal fields. *SIGPLAN Not.*, 34(10):223–237, 1999.

[27] Tamiya Onodera, Kiyokuni Kawachiya, and Akira Koseki. Lock reservation for Java reconsidered. In *ECOOP*, pages 559–583, 2004.

[28] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS '82)*, pages 22–30, 1982.

[29] Krzysztof Palacz and Jan Vitek. Java subtype tests in real-time. In *ECOOP*, pages 378–404, 2003.

[30] Standard for threads interface to POSIX. IEEE, P1003.1c, 1996.

[31] Kenneth Russell and David Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. *SIGPLAN Not.*, 41(10):263–272, 2006.

[32] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In Toby Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA'97)*, pages 142–157, Atlanta, 1997.

[33] D. Weaver and T. Germond. *The SPARC architecture manual (version 9).* PTR Prentice Hall, Englewood Cliffs, NJ 07632, USA, 1994.