KTH Computer Science
and Communication

Evaluating and improving biased locking in the HotSpot virtual machine

MARCUS LARSSON

marcular@kth.se

Master's Thesis at CSC for Oracle Svenska AB

Date:	June 26, 2014
Subject:	Computer Science
Supervisor:	Mads Dam
Examiner:	Johan Håstad

TRITA xxx yyyy-nn

Abstract

Biased locking is an optimization to enable very fast synchronization in the Java virtual machine. It is based on the observation that many times objects are not shared between threads, and could therefore avoid the slower and otherwise necessary atomic instructions for synchronization.

In this thesis, the biased locking algorithm and some of its variations are investigated. The biased locking implementation in the HotSpot virtual machine is evaluated and analysed. Standard and common benchmark suites, as well as microbenchmarks are used to evaluate the implementation. Although occasionally negative, the results suggest overall positive effects from biased locking, both on new and old hardware.

A prototype of an improvement to the algorithm is implemented in HotSpot. The prototype is evaluated and analysed using microbenchmarks. Results indicate performance improvement in certain scenarios using the prototype, suggesting that the improvement might be useful.

Referat

Utvärdering och förbättring av partisk låsning i den virtuella maskinen HotSpot

Partiska lås (eng. *biased locking*) är en optimering för att möjliggöra mycket snabb synkronisering i virtuella maskiner för Java. Den baseras på observationen att objekt många gånger inte delas mellan trådar, och kan därför undvika de långsammare och annars nödvändiga atomiska instruktionerna för synkronisering.

I denna rapport undersöks algoritmen för partisk låsning, samt några av dess variationer. Implementationen av partiska lås i den virtuella maskinen HotSpot utvärderas och analyseras. För att utvärdera implementationen används vanliga och standardiserade prestandatester, samt även egna mikroprestandatester. Resultaten visar ibland negativa effekter av partiska lås, men överlag är effekterna positiva, både på ny och äldre hårdvara.

En prototyp på en förbättring av algoritmen implementeras i HotSpot. Prototypen utvärderas och analyseras med hjälp av mikroprestandatester. Resultaten från dessa tester påvisar förbättringar i prestanda för prototypen i specifika scenarion, vilket tyder på att förbättringen kan vara duglig.

Contents

Chapter 1

Introduction

The Java programming language provides built-in support for multi-threaded programming. More specifically, any method or block of statements can be synchronized, using the `synchronized` keyword, potentially turning any object into a *monitor* [4]. These constructs are used to provide safety and correctness in multi-threaded applications and libraries. Naturally, this requires a certain amount of overhead in the Java Virtual Machine (JVM), as some locking mechanism is required for every object. Much work in JVM development has been devoted to finding efficient solutions to this problem. The locks (Java locks) need to have small memory footprints, and impose low latencies with low contention but still allow for maximum throughput when heavily contended.

In this project *biased locking* in the HotSpot JVM is evaluated and analysed, and improvements to the implementation are investigated.

1.1 Background

There are generally two different types of locks, *spin-locks* and *suspend-locks* [9,12]. Threads locking a spin-lock will loop until the lock is successfully acquired. Spin-locks can be simple, require at minimum only a word of memory, and perform well with low contention [9, 10]. The downside to these locks is, however, that when contention rises the overall performance suffers. Much time will be spent looping to acquire the lock, and less time will be spent on actual work.

Suspend-locks are a bit more complicated because they involve thread scheduling, typically via operating system (OS) functions [10, 12]. Instead of spinning, a thread waiting on a lock yields its time-slice to another thread to hopefully make better use of the CPU. The thread scheduling and OS involvement complicates the suspend-lock, and along with the required context switches this makes suspend-locks slower than the simple spin-lock when there is little contention [12]. Under heavy contention, however, suspend-locks perform better, since less CPU time is spent just waiting on the locks [9, 10]. However, if the lock is acquired only for very short durations a better approach can actually be to use a spin-lock, because the overhead of context-switching might outweigh that of wasting some CPU cycles waiting on the lock, depending on the OS [10].

An optimization to suspend-locks involves combining it with a spin-lock, initially spinning a limited number of times before taking the slow path of suspending the thread [12, 13]. This hybridization allows for low latency with low contention, and high efficiency when contention rises. Typically this approach is used to implement the locks for the monitors required in Java.

1.1.1 Java locks

Due to the fact that any object can be used as a monitor in Java [8], all objects must have some sort locking data structure associated with it. Every object can be used as a lock in a `synchronized` statement, guaranteeing *mutual exclusion* [9] in the corresponding critical sections (synchronized blocks or methods) [8]. The monitor interface also requires all objects to support the `wait()`, `notify()` and `notifyAll()` methods, allowing threads to wait on a monitor object until notified [4]. Ideally, this should also work as the hybrid spin-suspend lock described above for efficiency and low latency.

Regardless of the type of lock used to implement the monitor functionality, some supporting monitor data structure for each object is required. Eagerly allocating these monitors for every object is not good from a memory usage perspective. One possibility is to keep a global auxiliary map from objects to their monitors, allocating and mapping them only when necessary [2,12]. In other words, the object monitor is allocated whenever a `synchronized` block or method is executed, alternatively when `wait()` or `notify()` methods are called. If no objects are ever synchronized upon, the memory requirement is simply that of the overhead for the global map. This map itself will, however, need to be synchronized with locks or similar, impeding the performance, and when many objects require synchronization the map might require considerable memory overhead [1, 2].

A perhaps better and simpler approach is to keep a lock word in the object header. To avoid adding an additional word of overhead for every object, this can possibly (if it exists) be integrated with the metadata for objects, such as class and garbage collector (GC) information [2, 12]. The result is either a distinct lock word, or a multi-purpose word in the object header, where some bits are reserved for the locking mechanism. The latter scheme is used both for JRockit and HotSpot JVMs [10,16]. Information about the object monitor associated with the object can then simply be stored in this word. With this technique, the monitor data structures can be allocated only when required, and the mapping from object to its monitor is simply a pointer contained in the *lock field* of this multi-purpose word [2, 12].

1.1.2 Thin and fat locks

Because observations revealed that most objects in Java are actually never contended, the concept of *thin* locks was introduced [1, 2, 5, 10, 12]. Rather than allocating the monitor data structure as soon as an object is locked, a lightweight locking mechanism is used. This involves using the lock field like in a spin-lock, acquiring the lock with an atomic instruction such as compare-and-swap (CAS). Not contended objects will therefore only require the lock field, rather than the heavyweight monitor data structure (*fat* lock). Thin locking does, however, not support the operations of `wait()` or `notify()` which require an actual monitor [1,2,5,10,12]. Also, whenever the lock is contended, it should fall back to a suspend-lock mechanism, again requiring a fat lock. In these cases, the thin lock is *inflated*, during which a monitor structure is allocated and the lock field is updated to contain a pointer to this monitor instead [1, 2, 5, 10, 12]. To distinguish between thin and fat locks, a bit in the lock field is used to determine its mode, and the lock is referred to as *bimodal* [12].

Furthermore, Java locks need to be *reentrant* [8], meaning that a thread needs to be able to acquire the same lock recursively [9]. This puts some additional constraints on the lock, as it needs to contain information on lock ownership and recursion count. Ownership information is necessary for the locking thread in the recursive case to be able to identify that it already holds the lock. Recursion count is required for the lock to become released only when the number of unlock calls matches that of the number of preceding lock calls. In the case of an object monitor being used, this information is easily stored in the supporting data structure [2, 12]. For thin locks this information must somehow be encoded in the lock field. The perhaps most straightforward solution to this is to split the lock field into two subfields, one containing a thread identifier for the owner and the other to contain a recursion counter [2]. Since the field is limited in size the recursion count supported for thin locks will naturally be limited. In the case of an overflow one can, however, simply inflate the lock and fall back to the fat lock mechanism [2].

Another solution for recursive thin locks involves the use of *lock records* on the stack of threads acquiring a lock [1, 10, 16]. Whenever a thread attempts to acquire a lock, a lock record is allocated on the thread's stack where information about the lock is stored. This information can include whether the lock acquisition was recursive or not, and hence the recursion count is kept implicitly by the number of lock records on the stack [10, 16]. When a thread acquires a lock for the first time, it will update the lock field, and keep information about the locked object in the lock record. During recursive lock and unlock attempts, the corresponding lock records will indicate that the locking was recursive, and the object will remain locked until the initial lock is released. Information of all the locks a given thread has acquired will then be kept in the set of lock records of that thread. This set needs to be ordered in the same order the locks were acquired to ensure proper structured acquire and release sequences [10].

Variations and improvements of the thin lock include the *tasuki-lock* [12] and *meta-lock* [1]. The basic idea remains the same in these algorithms, and the differences lie mainly in implementation specifics. The tasuki-lock improves the inflation scheme, and allows the locks to be deflated easily [12]. The meta-lock builds upon thin locking but uses lock records to implement the fat locks, rather than the separate monitor structures.

1.1.3 Biased locking

A further optimization of Java locks is based on the observation that many locks exhibit what is called *thread locality* [11]. This means that the lock's locking sequence contains very long repetitions of acquires and releases by a specific thread, referred to as the lock's *dominant thread* [11].

As Java is a programming language which supports multi-threaded programming, many libraries are written to support deployment in concurrent applications. This means that those libraries will contain the necessary synchronization to guarantee safety and correctness in all use cases. However, when used only by a single thread these synchronizations will only add overhead. Even with thin locks, the synchronization adds the necessity of costly atomic instructions such as CAS every time the lock is acquired. Biased locking attempts to exploit the thread locality of locks in situations such as this, allowing a lock to bias itself to the first thread to acquire the lock [6, 10, 11]. When biased, the lock provides an ultra fast path for the biased thread, requiring only a few non-atomic instructions to acquire or release the lock [6, 10, 11, 16]. If the lock is ever acquired by a different thread than the bias owner, the bias will need to be *revoked*, and the lock falls back to use the underlying locking mechanism (thin locking for example). Schemes similar to biased locking include the so called *lock reservation* [11] and *lazy unlocking* [10] techniques. The idea of these techniques is the same, enabling an ultra fast path for dominant threads, and they differ mainly in implementation details.

To distinguish biased locks from non-biased, an additional bit in the lock field is used [6, 11]. Initially, the locks are in the biased state, but without a thread assigned to them. The first time a thread acquires the lock, the thread will notice that no thread is yet assigned and simply bias the lock to itself [6]. This initial biasing operation requires the use of CAS (or similar) to avoid race conditions between concurrent initial acquires of different threads. Once the lock is biased, acquiring or releasing the lock is simply a matter of reading the lock field to confirm the lock is still biased towards the thread. The reacquiring process might involve storing the recursion count in the lock field, which can be done with a simple store instruction [11].

It is also possible to, like before, ignore explicit recursion count for biased locks, and leave no indication in the lock field whether or not the lock is taken [10, 16]. In this case, once the lock is biased it can be considered always locked by the biased thread.This makes the acquire and release operations on a biased lock completely store free, and recursion count is kept implicitly by the lock records.

Revocation

When a biased lock is acquired by a different thread than that of the biased thread, the bias must be revoked. Since the biased thread might acquire or release the lock at any time, and because it has no obligations to use atomic instructions when doing so, the biased thread must be suspended in order to safely make modifications to the lock [6, 10, 11, 16]. Once the biased thread is stopped, the lock can safely be revoked, possibly involving walking the owning thread's stack to find and modify all of its lock records. Alternatively, if the biased thread no longer exists, the lock can simply be revoked or rebiased directly. If the revoker finds that the lock is held, it converts the biased lock into a thin lock by modifying the lock field and the corresponding lock records to look as if the thin locking mechanism had been used from the start [10]. If the lock is found *not* held by the bias owner, there is an alternative for the revoker to simply rebias the lock to itself. This effectively transfers the bias to the new thread, removing the previous bias of the lock. Such *rebiasing* is good for *producer-consumer* patterns, allowing the locks to be handed over from thread to thread and continue exploiting thread locality.

Revocation is a costly operation because it requires thread suspension and possibly stack walking to find and modify lock records. In comparison to the thin locking mechanism, or even the heavy-weight monitor mechanisms, the revocation process is much slower [6, 16].

Suspending biased threads can be done with OS signals, but this adds requirements on the underlying OS for the JVM. Furthermore, many I/O operations do not work properly if interrupted by signals, and signalling is therefore usually avoided [6]. Because JVMs are garbage collected environments, they typically provide a way to stop the world (STW). This means that all the Java threads are suspended at well known states, *safepoints*, for example to allow the GC to run safely [16]. This mechanism of suspending threads at safepoints is a good method to suspend threads for revocation purposes as well. When carefully designed, safepoints can guarantee that no thread is in the process of acquiring or releasing a lock during STW-time, which could otherwise complicate the revocation process [14]. A disadvantage of safepoint revocation is that many JVMs only support global safepoints, which means that all threads will be suspended when a safepoint is requested [14, 16]. Since revocations only require suspending the biased thread, this does not scale very well and increases the cost of revocations even further.

1.2 Problem

1.2.1 Motivation

New CPU architectures reduce the relative latencies of atomic instructions compared to regular instructions [7]. These latencies were the main reason to why biased locking was originally introduced. In other words, the benefit of using biased locking is directly proportional to the latency of these instructions. If the latencies are reduced, so is the need for biased locking. For biased locking to be viable, the latency must justify the necessary increased code complexity, as well as the risk and cost of the more expensive revocations.

The HotSpot JVM currently implements biased locking. The bias revocation process requires a global safepoint, suspending *all* threads in the JVM rather than involved threads only. New heuristics for biased locking suggest a learning or sampling phase for the locks that could potentially improve overall performance by reducing the number of revocations [14, 15].

1.2.2 Statement

Is biased locking needed on modern hardware, and if so, how can the revocation mechanism and heuristics for biased locking be improved in the HotSpot virtual machine?

1.2.3 Goal

The goal of the project is to evaluate, by benchmarking, biased locking in the HotSpot JVM, comparing the performance to that of some alternative lock implementations. This includes HotSpot without biased locking and JRockit with its similar lazy unlocking scheme.

Improvements of the biased locking algorithm are studied, implemented and evaluated. Benchmarks are used as a basis to determine if improvements are successful or not.

Chapter 2

Previous research

2.1 Bulk operations

To mitigate the cost of revocations, a technique involving *bulk rebias and revocation* has been developed. This basically allows the JVM to rebias or revoke not just the bias on a certain object's lock, but on all locks on objects for a specific data type [16]. Different classes will typically have different use cases, and some objects are more likely to be shared between threads than others [16]. The use pattern of certain data types can prove to be problematic for biased locking schemes. Producer-consumer queues are typically always contended, for example. In these scenarios it is beneficial to detect the pattern at a data type level, revoking or rebiasing all objects for that type in a single pass, rather than revoking each individual object when its thread ownership changes. In other words, heuristics are added to the biased locking algorithm, estimating the cost of bias revocations per data type and rebiasing all locks of that data type when a certain threshold is reached [16]. This bulk rebias operation resets the biased thread for each unlocked object of the given data type, allowing the next acquiring thread to bias the lock to itself without the need for revocation. During this operation locked objects can either be revoked, or left biased. The heuristics might also decide to perform a so called bulk revocation operation, completely forbidding biased locking to be used on all objects of the class.

Implementing this scheme in the naive way, one would be required to scan the whole heap for objects in order to rebias unlocked objects. A perhaps better way to implement bulk rebias operations is to introduce an *epoch* field for objects [16]. Adding this field to the class metadata, and also for each object (as part of the lock field for example), the bias of a given lock is only valid if the epoch numbers in the class metadata and object match. The invalidation then simply consists of increasing the epoch field of the class, iterating over all threads to update the epoch field of currently locked objects as well. Objects that were not updated will then have an invalid epoch number, and will thus be rebiasable. The epoch field is limited in size. There are occasions when this field wraps around, possibly causing some invalidated biases to become valid again. This is luckily not a problem for correctness, it is only a performance issue as locks are unintentionally rebiased. In the worst case this causes revocations. Experiments by Detlefs and Russell [16] indicate the impact on performance due to epoch wrapping is negligible. Also, as a countermeasure the GC can normalize the lock field to the initial unbiased but rebiasable state when encountering objects that have an invalid epoch, further reducing the impact of epoch wrapping.

To allow bulk revocation for classes, one can similarly add a field to the class metadata specifying if biasing is allowed or not. When acquiring a lock, this field will have to be checked to determine whether to use biased locking or just thin locking [16]. If a prototype object header is kept in the class metadata, biasing can be disabled in the lock field of this prototype, and new objects of that class will retain the class biasing policy [16]. Naturally, when bulk revocation occurs all currently biased locked objects of the class must be revoked, after which the bias bit in the prototype can be set to 0, implicitly revoking the remaining objects of the class.

2.2 Adaptive biased locking with learning

A different improvement to biased locking, attempting to reduce the need for revocations, involves introducing a learning phase for locks. Instead of biasing the locks when first acquired, locks are biased after a certain number of acquires by a single thread [14, 15]. Acquiring the lock in the learning phase works just like with thin locking, using atomic instructions. During the learning phase, the field that would normally contain the biased thread reference instead contains a bias candidate. Once the candidate has acquired the lock sufficiently many times without conflict, the lock is biased to the candidate.

The most straightforward implementation of this scheme would be to keep a counter in the lock field to keep track of how many times the candidate has subsequently acquired the lock (disregarding recursive acquires) [14, 15]. Naturally, if someone other than the candidate acquires the lock during the learning phase, the counter has to be reset and the candidate adjusted. Alternatively, biasing the object could then be forbidden completely, since the lock did not exhibit thread locality [14]. The learning phase can reduce the amount of revocations required at the cost of increasing the number of CAS instructions before locks are biased [14, 15].

An idea to avoid the requirement of the additional space in the lock field for the counter involves *random counting* [14]. Instead of actually counting up to N acquires, one can bias the lock with probability $\frac{1}{N}$ for every attempt. On average, this will require N attempts to bias the lock. Since implementing and using a random number generator might both complicate and slow down the synchronization code, a simplified version of random counting can instead be used, where each thread has a local counter for successful acquire attempts made [14]. Threads will then bias a lock after reaching the threshold for successful acquire attempts. This would result in a $\frac{1}{N}$ bias probability on average, even though it is not truly random [14]. Another alternative for random counting involves using probabilistic branching when supported by the architecture.

2.3 Fixed bias locking

A way to eliminate the need for revocations completely involves biasing the lock to a thread but still allow non-biased threads to acquire the lock [13, 17]. In other words, a lock will have a fixed biased thread that never (or rarely) changes, and still allow non-biased threads to acquire the lock without revoking the bias first. Biased threads acquiring the lock will have to make sure no other thread holds the lock before entering the critical section. As the whole purpose of biased locking is to avoid the use of atomic read-modify-write (RMW) instructions, the locking algorithm must use only regular read and write instructions. Efficient examples of such algorithms include *Dekker's algorithm* and *Peterson's algorithm*, both of which are solutions to the mutual exclusion problem for two threads [9, 13, 17].

Carefully adopted, the above locking techniques can be used to provide mutual exclusion between the biased thread and all the other threads, while still allowing the biased thread to acquire and release the lock using only regular read and write instructions. Depending on the implementation, *memory barrier* instructions are possibly necessary to guarantee that the read and write instructions are not reordered by the CPU [13,17]. Non-biased threads acquiring the lock will first have to qualify to compete with the biased thread (become the other thread for the 2-thread lock). As there is no requirement for non-biased threads to acquire the lock without atomic RMW instructions, these threads can simply use a regular RMW-based lock to guarantee mutual exclusion among non-biased threads.

The *Spin-by-KKO-lock* [13] is a realization of this type of lock proposed by Onodera et al. Conceptually this lock requires three fields: one identifying the biased thread, one identifying the other thread (non-biased locking thread) and one indicating whether or not the lock is held by the biased thread. As before, the bias is acquired with a CAS on the biased thread identifier field. Once biased, the biased thread can acquire and release the lock by setting the status field to its appropriate status using regular write instructions. When acquiring the lock the biased thread also makes sure that the `other` field is empty (0) before entering the critical section. Non-biased threads acquire the lock with a CAS on the `other` and status fields, expecting them both to be 0 and writing their identifier into the `other` field. If any of the acquire attempts fail, the algorithm falls back to the underlying lock algorithm (inflating the lock for example). The implementation of this lock actually only requires a single word to contain the three fields [13]. This allows a CAS instruction to work on the multiple fields without the need for multiple word CAS instructions. To avoid overwriting the `other` field when a biased thread acquires the lock, the status bit is written using 8-bit write instructions that only overlaps part of the biased thread field and not the `other` field.

Chapter 3

Existing HotSpot implementation

3.1 Java objects and locks

In the HotSpot JVM, each object has a header consisting of two words. One word is used to identify the class of the object, and the other word, called the *mark word*, is used for hashcode computations, synchronization and garbage collection. The mark word contains different information depending on its two least significant bits. This word is used for biased, thin and fat locks alike.

Figure 3.1 and 3.2 describe the different layouts of the mark word for 64-bit and 32-bit architectures, respectively. Due to implementation specifics in hashcode computations, the hashcode field requires at most 31 bits, and hence the 64-bit mark word leaves some bits unused. The *Age* and *CMS* are bits used by the GC, and are unrelated to the locking implementation.

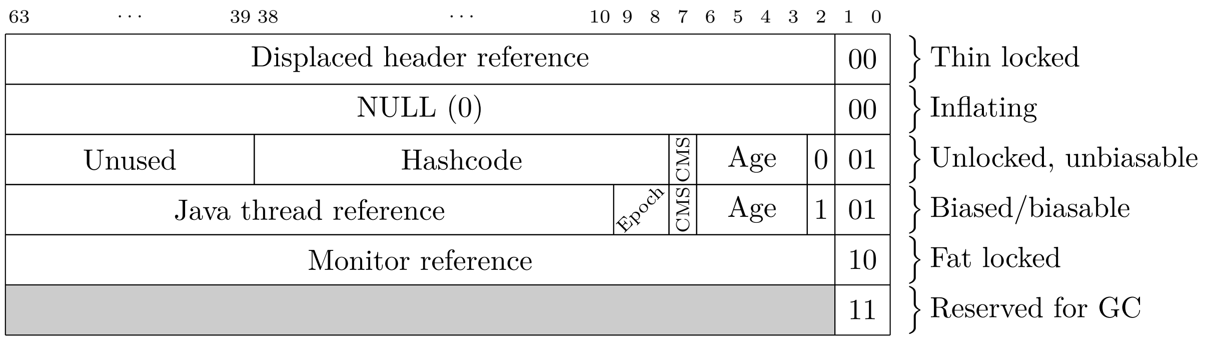

Figure 3.1. The different states and layouts of the 64-bit mark word in HotSpot.

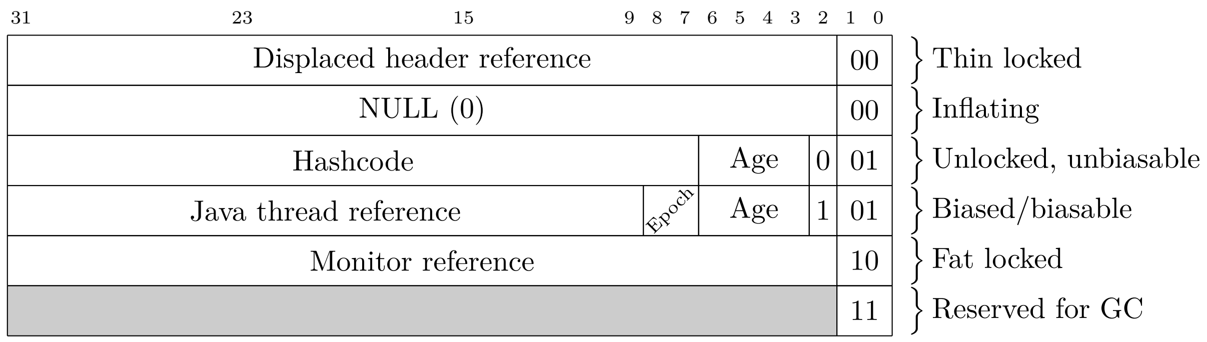

Figure 3.2. The different states and layouts of the 32-bit mark word in HotSpot.

3.2 Thin locks

Thin locks are implemented in HotSpot using displaced headers, so called *stack locks*. When a thread acquires a thin lock, it will copy the mark word to the thread's lock record and then CAS the header to the thin locked state, leaving a pointer to the lock record in the object's mark word. The two least significant bits in the mark word will be set to 00 to indicate it is thin locked, and that the rest of the word contains a pointer to the original object's mark word. Figure 3.3 shows the state of the object and lock record after a thin lock has been acquired. During recursive acquires, the thread will notice that it already owns the lock by checking that the pointer is pointing somewhere into the thread's stack. The lock record is in this case set to `NULL` (0), indicating that it is recursive. Unlock attempts will check the lock record, and if it is `NULL` the release operation simply returns, as the lock is still held recursively.

Lock records are allocated on the stack, either explicitly during interpreted execution or implicitly when compiled. The lock records contain a location where a *displaced mark word* can be placed, and also a pointer to the object being locked. The object pointer is initialized by the interpreter or compiler when allocated, and the displaced mark word is initialized whenever the lock is thin or fat locked.

3.3 Inflation

When a thread tries to acquire a thin lock already held by some other thread (the CAS fails), the non-owner thread will attempt to inflate the lock. During inflation the heavyweight object monitor structure is allocated and initialized for the object. The inflating thread will attempt to inflate the lock in a loop until it is successful, or until some other thread has successfully inflated the lock. Inflation also occurs if `wait` or `notify` is called, even on objects that are not locked.

In order to inflate a lock that is thin locked, the inflating thread will first CAS the mark word to `INFLATING` (0), after which it will read the displaced mark word and

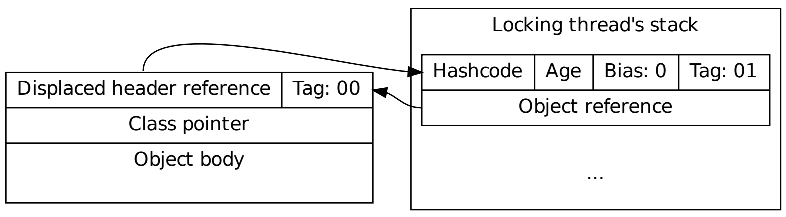

Figure 3.3. Example of a thin locked object.

copy it to the new object monitor structure. Threads that encounter an `INFLATING` mark word will wait for the inflating thread to complete the inflation action. This includes the thread that currently holds the lock, making it unable to release the lock until the inflation is complete. The use of this temporary inflation state is necessary, because otherwise the inflating thread might read an outdated version of the mark word. By putting the lock in the inflating state, the inflating thread guarantees that the displaced mark word cannot change, and it is safe to read and copy it.

Depending on the state of the lock in each iteration of the inflation loop, the following actions are taken:

- The lock is already inflated (tag is 10)
  Some other thread successfully inflated the lock. Break out of the loop.

- The lock is being inflated (mark word is `INFLATING`)
  Some other thread is currently inflating the lock. Wait until inflated.

- The lock is thin locked (tag is 00)
  Allocate an ObjectMonitor, then CAS `INFLATING` to the mark word. If the CAS fails, deallocate the monitor and retry the inflation loop. If the CAS succeeds, install the monitor by setting up the appropriate fields, copying the displaced mark word to the object monitor, and finally store the monitor pointer in the mark word (replacing the `INFLATING`).

- The lock is unlocked (tag is 01)
  Allocate an ObjectMonitor, set it up, and try to CAS its reference into the mark word. On failure, deallocate the monitor and retry the inflation loop, otherwise break out of the loop.

Once the lock is inflated any acquiring threads will simply use the underlying monitor mechanism to acquire the lock. An example of a fat lock can be seen in Figure 3.4. HotSpot deflates idle (unused) monitors during STW-time to allow no

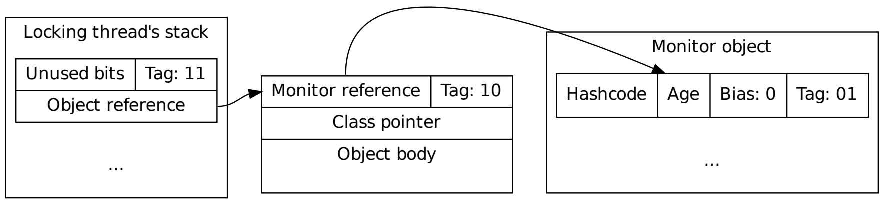

Figure 3.4. Example of a fat locked object.

longer contended locks to fall back to thin or biased locking again. This is naturally safe since no threads can acquire or release the locks during STW-time.

3.4 Biased locking

Building upon the thin and fat locking mechanisms described in previous sections, HotSpot supports store free biased locking with bulk rebias and revocation [16]. This feature can be toggled with JVM arguments, and is enabled by default. A bit is used in the unlocked state of the mark word to indicate whether or not biased locking is used or disallowed for the object. This is the third least significant bit as can be seen in Figure 3.1 and 3.2. If the bit is 0, the object is truly unlocked and biasing is not allowed on the object. If it is `1`, the lock can be in one of the following states:

Anonymously biased
The thread pointer is `NULL` (0) meaning that no thread has yet biased the lock. The first thread to acquire the lock will notice this and bias the lock to itself with an atomic CAS instruction. This is the initial state for objects of classes that allow biased locking.

Rebiasable
The epoch field of the biased lock is invalid (does not match the epoch of the class). The next thread to acquire the lock will notice this and bias the lock to itself with an atomic CAS instruction. During bulk rebias operations, not held biased locks are put in this state to allow quick rebiasing.

Biased
The thread pointer is *not* `NULL` and the epoch is valid, meaning that some thread might currently be holding the lock.

Due to the fact that biased locking requires the use of the hashcode field for the biased thread identifier, biased locking cannot be used for hashed objects. Hashcode computations on objects that allow biasing will first revoke any (valid or invalid) bias, and then CAS the computed hashcode into the mark word. This only applies to the *identity hashcode* though, which is what the `hashcode()` method on the `Object` class will compute. Hashcode computations for classes that override the `hashcode()` method will not require an identity hashcode (unless explicitly using the object hashcode of course), and can still use the biased locking mechanism.

HotSpot keeps a prototype mark word in the class metadata for every loaded class. Whether or not biased locking is allowed for the class is determined by the bias bit in this prototype. Also, the current epoch for the class is kept in the prototype. This means new objects can simply copy the prototype and use it without any further modification. During bulk rebias operations the prototype's epoch is updated, and bulk revocation will change the prototype into the unbiasable state (setting the bias bit to 0).

A biased lock is acquired by inserting the thread pointer in the mark word using an atomic CAS instruction. Naturally, a prerequisite for biasing a lock is that it is either anonymously biased or rebiasable (a lock can only be biased to one thread at a time). Once the lock is biased, recursive locking and unlocking require only a read of the object header and the class prototype to verify the bias has not been revoked. Lock records for biased locks will contain the pointer to the locked object, but are otherwise uninitialized. An example of a biased lock can be seen in Figure 3.5. The unused memory location for the displaced mark word is still required in case the bias is revoked, during which the lock is converted into a thin lock.

Revocations in HotSpot are performed using global safepoints. During revocation the revoker will iterate over the bias owning thread's lock records in order to conclude whether or not the object is currently locked. If the lock is found to be held by the biased thread, the lock records are modified to look as if thin locking was used. If the lock was currently not held, and depending on what caused the revocation, the object is either disallowed to use biased locking or rebiased to the revoking thread.

Even when biased locking is enabled, the feature is disabled the first four seconds after the JVM has started due to poor performance when used at start-up time. This means that the prototype mark words will during this time have their bias bits set to 0, disallowing biasing of objects instantiated during this time. After the four second period, the bias bit is set to 1 in all prototype mark words and thus allowing biased locking to be used on new objects.

During GC, object mark words are sometimes normalized to the prototype mark word. This is done to reduce the number of object headers that must be preserved during GC. Object headers that are hashed, locked or have biasing locally disabled (bias bit set to 0 when the prototype has its set to 1) will be preserved through GC, but not headers that are biased but currently not held by its biased thread. This means that the bias of unlocked biased objects are possibly forgotten at each GC (depending on the GC), forcing these locks to the anonymously biased state.

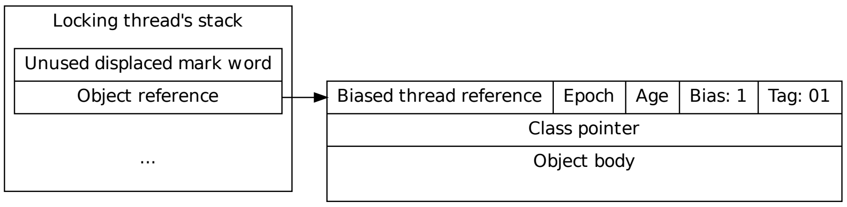

Figure 3.5. Example of a bias-locked object.

To acquire a lock when biasing is enabled, the following steps are taken:

1. Test the bias bit of the object
   If it is 0, the lock is not biased, and the thin locking algorithm should be used.
2. Test the bias bit of the object's class
   Check that the bias bit is set in the prototype mark word for the object's class. If it is not, biasing is globally disallowed on objects of this class and the bias bit of the object should be reset and thin locking used instead.
3. Verify the epoch
   Check that the epoch in the object's mark word matches that of the prototype mark word. If not, the bias has expired and the lock is rebiasable. In this case the locking thread can simply try to rebias the lock with an atomic CAS.
4. Check owner
   Compare the biased thread identifier to the locking thread's identifier. If they match, the locking thread is currently holding the lock and can safely return. If they do not match, the lock is assumed to be anonymously biased and the locking thread should try to acquire the bias with an atomic CAS. On failure, revoke the bias (possibly involving a safepoint) and fall back to thin locking. On success, the locking thread is the biased owner of the lock and can return.

After the initial check, the last three checks can be realized with only one conditional jump by first loading the prototype mark word, bitwise-or it with the locking thread's identifier and then `XOR` the result with the object's mark word. If the result is 0 this means that the class allows biasing, the epoch is valid and that the locking thread is currently the biased owner. Otherwise, if the result is not 0, the locking thread must investigate which of the bits that differed to know which of the three cases from above that was encountered to be able to proceed.

The release sequence when biased locking is enabled is simply to check whether the bias bit is set or not. If it is not set the thin locking algorithm is used. If it is set, the unlocking thread must be the bias owner, because the lock can not have been rebiased to a different thread while the lock was held. With a set bias bit, the unlocking thread will simply do nothing. Whether or not a given thread holds a certain lock depends on the lock records on the thread's stack, and no additional book-keeping is required.

As an overview, the various states of the mark word and the transitions between them is illustrated in Figure 3.6. To simplify, the transitions for `wait`, `notify` and `hashCode` operations are omitted. Also, depending on if the lock is held or not when it is in the biased state, different transitions are possible. For example, if the lock is not held during revocation, it simply transitions to the unlocked state. If the lock is held during revocation, it transitions to the thin locked state. Whenever `wait` or `notify` is called, the lock will always transition to the inflated state. Hashcode computations in the thin locked state requires inflation to support modification of the displaced mark word. In the unlocked case it is simply the natural transition to the unlocked but hashed state, where the computed hashcode has been inserted into the mark word. As the inflated state contains both the case when the hashcode is computed and when it is not, it will cause different deflate-transitions depending on this. Additionally, hashcode computations for the biased or biasable state will first revoke the bias, after which it will follow the same hashcode transitions as the thin locked or unlocked state.

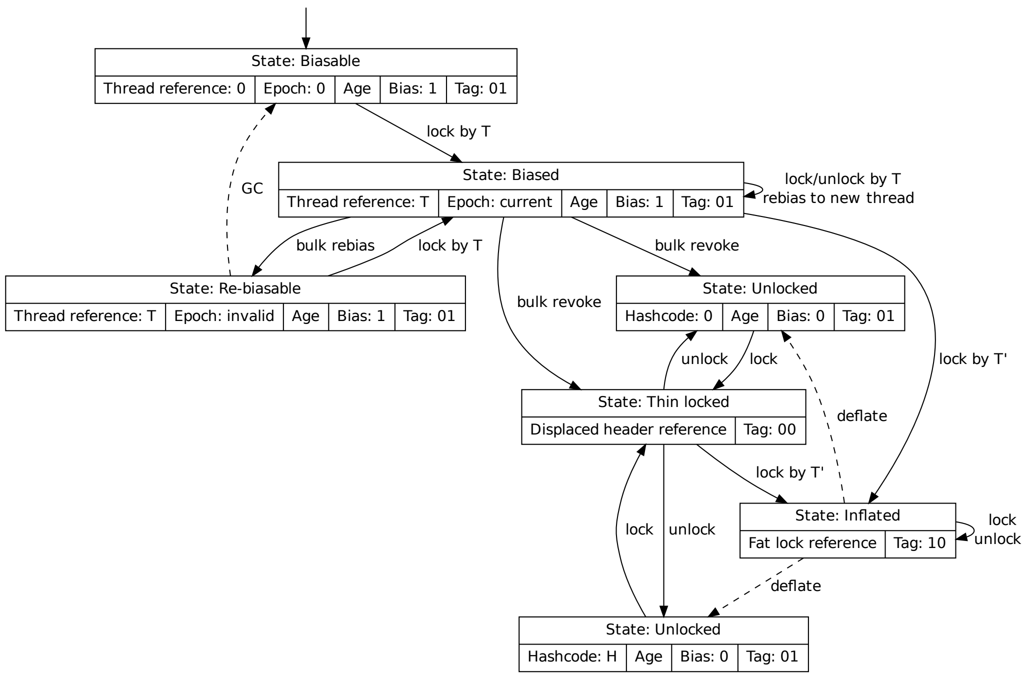

Figure 3.6. Simplified state graph of the mark word in HotSpot.

3.4.1 Heuristics

As mentioned in previous sections, heuristics are used to determine when to perform bulk rebias and revocation operations. The metadata for each class includes a counter that is incremented each time a biased lock is revoked for an object of the given class. Also a timestamp is used to indicate the last time a bulk rebias operation was performed on the class. The revocation count does not include revocations on objects that are anonymously biased or rebiasable. These revocations are very cheap, requiring only a single CAS, and should therefore not cause bulk operations on the class.

The current heuristics has two thresholds for the revocation counter: the bulk rebias threshold and the bulk revocation threshold. Initially, the heuristics will choose to revoke or rebias locks individually. Once the bulk rebias threshold is reached the bulk rebias operation is performed, revoking all biases and leaving the locks in a rebiasable state.

A time threshold (decay time) is used to reset the revocation counter if enough time has passed since the last bulk rebias operation. This means that there has not been many revocations since the last bulk rebias operation was performed, which suggests that biasing could still be beneficial for the class. Typically this is caused by a scenario where one thread has been working on objects of a certain class and then handed them all over to other threads. This behaviour can be fine, as the epoch based bulk rebias operation should be cheap enough to allow the repeated rebiasing if it does not occur too often.

If revocations keep occurring after a bulk rebias operation, within the decay time threshold, the counter will keep incrementing. Once the bulk revocation threshold is reached, a bulk revocation is performed and biased locking is no longer allowed on objects of the class. The heuristics will at this point stop tracking revocation count and decay time. See Table 3.1 for the current threshold values used.

Table 3.1. Current heuristics threshold values in HotSpot.

Threshold	**Value**
Startup delay	4s
Bulk rebias threshold	20
Bulk revoke threshold	40
Decay time	25s

3.4.2 Comparison with JRockit lazy unlocking

The implementation of biased locking in HotSpot is similar to that of lazy unlocking in JRockit, in the sense that they both use fat locks, thin locks and then biased locking or lazy unlocking on top of these locking schemes. Both JVMs have the class pointer and mark word as the metadata for each object. However, thin locking and the biasing or lazy unlocking differ in implementation. JRockit does not displace the mark word when acquiring locks. Instead, thin locks will have an identifier of the locking thread in the mark word [10]. The layout for JRockit's mark word can be seen in Figure 3.7.

The concept of lock records is also used in JRockit, although the lock record consists of only a single word as there is no need to provide space for a displaced mark word. The lock records contain pointers to the locked objects. Due to alignment restrictions on objects, the lower three bits of the lock record is always 0, and can thus contain information about how the lock was acquired (if it was fat, thin, recursive or lazy locked). This information is used to determine which unlock method to use when the lock is released.

In JRockit, the first thread to acquire a lock will attempt to use lazy unlocking on it. This is just like the bias policy in HotSpot, where the first thread biases the lock to itself. A bit in the mark word is used to distinguish between lazy locked objects and thin locked objects, called the *forbid bit*. If the forbid bit is set, lazy unlocking is not allowed on the object.

The revocation scheme is similar to HotSpot in the sense that a safepoint is required to walk the locking thread's stack. This safepoint technique, however, lets all threads other than the revoking and lock owning threads to continue execution immediately or shortly after they have been suspended.

JRockit does not support any bulk rebias or revocation operations like HotSpot does. Lazy locking can, however, be disabled at a class level by setting the forbid bit in the class metadata. New objects of this class will then have the forbid bit set during allocation and can not be lazy locked.

If it is found during revocation that the lock is not held by the reserved thread, JRockit allows the reservation to be *transferred* (effectively rebiased) to the revoking thread. Some bits in the mark word are reserved for a transfer counter, called the *transfer bits*. Every time a lazy lock reservation is transferred to another thread, this counter is incremented. This enables the possibility to track the number of transfers per object, which could be a helpful measurement for deciding whether or not to allow lazy unlocking. Once a specific transfer limit is reached, lazy unlocking is disabled on the object. The field is 7 bits wide and therefore it can track up to 127 reservation transfers. The default threshold is however set to 1, and thus lazy unlocking is forbidden on the second transfer attempt. Revocation when no transfer is possible (when the lock is held by the owner) causes lazy unlocking to be forbidden on the object immediately, and the lock is converted to a thin lock.

A notable difference between JRockit and HotSpot is that the JRockit mark word does not contain or reserve space for the hashcode. The identity hashcode is stored at a separate location. This allows lazy unlocking to be used even on objects that have their hashcode computed.

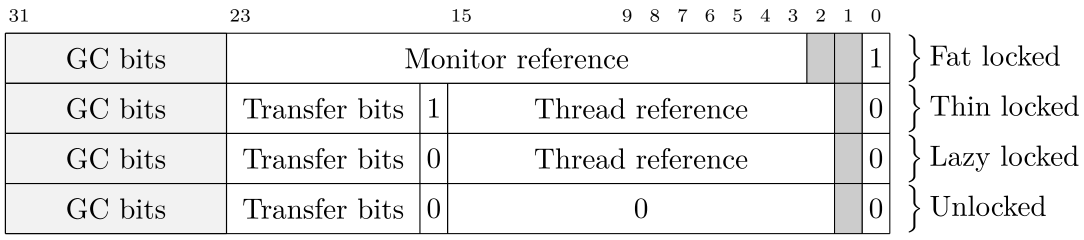

Figure 3.7. The different states and layouts of the JRockit mark word.

On the class level, JRockit uses heuristics to determine when to disallow lazy unlocking globally for the class, referred to as class banning. The heuristics measure the number of transfers and revocations for all instances of the class. Class banning can be triggered by two different conditions:

- More than 5 bans occur in a time span of 5s
- After three consecutive periods of time each with a duration of 3-24s, and each with more than 10 transfers

Whenever any of these conditions hold, lazy unlocking is disallowed for new instances of the class and thin locking is used instead.

Chapter 4

Evaluation of existing implementation

4.1 Method

The primary method to evaluate the implementation is by benchmarking, using different standard and common benchmark suites. The relevant results from these benchmarks include the impact of biased locking. Therefore, benchmark results with biased locking enabled are compared against results when biased locking is disabled.

To perform the actual performance measurements, each benchmark is run 10 times with biased locking enabled and then repeats this with biased locking disabled. The average result of when biased locking is enabled is then compared against the result of that when it is disabled. The relative impact on performance with biased locking is presented as the percentual change in throughput or execution time.

4.1.1 Benchmarks

The benchmarks used include some of the benchmark suites from SPEC, the DaCapo benchmark suite, and a few custom micro benchmarks developed for this project.

SPEC

The Standard Performance Evaluation Corporation[1] (SPEC) is a non-profit organization providing industry standard benchmarks. They provide a wide range of benchmarks for different purposes, but the ones used in this project are the following:

- **SPECjvm2008**
  A benchmark suite mainly exercising core functionality in Java. Consists of a couple of real life applications measuring the performance of the Java

[1]Standard Performance Evaluation Corporation, `http://www.spec.org`

Runtime Environment (JRE). The base workload is used for this benchmark in this evaluation, and no additional options are specified.

SPECjvm2008 consists of several benchmarks with different types of workloads, each of which measures the throughput of the JVM. The suite computes the composite result from all of its benchmarks, providing an overall measurement of the JVM throughput.

- **SPECjbb2013**
  A larger and more complex single benchmark for measuring performance of modern Java applications. Emulates a system representing the complete IT infrastructure of a global supermarket company. The suite is executed in composite mode (everything running in the same JVM), with no additional options specified.

  The benchmark suite produces two different results, one called the *max-jOPS* and the other *critical-jOPS*. The max-jOPS is a measurement of the maximum throughput the JVM achieves, whereas the critical-jOPS measures throughput under certain response time constraints.

DaCapo

The DaCapo benchmark suite[2] is an open source benchmarking suite for Java. The suite originates from the DaCapo research project, and attempts to benchmark the JVM and runtime with non-trivial and modern real world applications [3]. In this project version *9.12-bach* of the DaCapo benchmark suite is used. The `eclipse`, `jython` and `xalan` benchmarks are excluded because they did not work properly on all machines.

All of the included benchmarks except `luindex` and `batik` are multi-threaded and use one thread per available hardware thread on the machine. This means that the multi-threaded benchmarks will use 24 threads on a machine with 12 hyperthreading cores (two threads per core). The two benchmarks `luindex` and `batik` are, however, single threaded except for some helper threads with limited concurrency. Also, `avrora` does not scale to the available hardware threads but depends on benchmark parameters instead.

Each benchmark is invoked 10 times, every time in a new JVM with the benchmark options: "`-n 40 --scratch-directory /tmp/scratch`". These options specify that each benchmark invocation should use the specified scratch directory for temporary files, and should let the JVM run 39 iterations of the benchmark before doing the measurement iteration. With 40 iterations the classes that are unsuitable for biasing should have been banned by the heuristics, and the JVM should have reached a more stable state. The benchmark result is summarized as the average relative execution times and standard errors for each benchmark. Additionally, the geometric mean is calculated for the whole suite to summarize the results. The

[2]The DaCapo benchmark suite, `http://www.dacapobench.org`

geometric mean is used rather than the arithmetic mean because the results are normalized. Also, the geometric mean is less sensitive to occasional deviating results, which gives a fairer summary of the results.

Micro benchmarks

To measure JVM performance in specifically interesting situations and application patterns, some micro benchmarks are used. While these benchmarks do not quantify the actual performance gain or give some true measure of improvement in real world applications they do, however, give insights in where the algorithms perform well and where they do not.

To perform the actual micro benchmarking, the Java microbenchmark harness[3] (JMH) is used – a framework to build and run benchmarks for the JVM. The harness provides methods to measure performance as throughput (amongst other measures), and can avoid certain optimizations by the JVM that would otherwise interfere with the benchmarks.

The harness is invoked with arguments "`-f 10 -wi 20`". This means that each benchmark is run in a new JVM 10 times, each JVM performing 20 warm-up iterations before performing the measurement iterations. By default there are 20 measurement iterations. Each iteration is one second by default, and the harness summarizes the result as the average throughput in these iterations.

The following microbenchmarks are used:

Random-Syncs
: For 3 seconds per iteration, 10 threads repeatedly synchronizes on randomly chosen objects among 10^6 shared objects.

Handover-Same
: Two threads take turns to iterate over and synchronize on 10^4 shared objects of the same class.

Handover-Different
: Two threads take turns to iterate over and synchronize on 10^4 shared objects of 26 different classes.

Producer-Consumer
: A small producer-consumer application with 1 producer and 2 consumers. The producer allocates new objects, synchronizes on them repeatedly five times, and then pushes them to a global shared `BlockingQueue`. The consumers pop objects from this queue, and then synchronizes on them five times before throwing the objects away.

Both producer and consumer will perform some other work during synchronization to emulate a more realistic scenario. This is accomplished with the

[3]JMH, `http://openjdk.java.net/projects/code-tools/jmh/`

`BlackHole.consumeCPU()` method from JMH, with workload parameter 5 inside the synchronized block, and 100 directly after the block. As the name implies, this method will consume CPU cycles doing some irrelevant computations. These workloads are chosen to make the critical section small, and have some considerable work to do in-between synchronizations, hopefully reflecting a realistic use pattern.

Synchronized looping

As a raw measurement of biased locking performance, the benchmark in Figure 4.1 is used. The `syncloop()` method is called repeatedly during this microbenchmark, and the throughput is measured.

```java
private Object o;
public Object syncloop() {
    o = new Object();
    for (int i = 0; i < 1000; i++) {
        synchronized (o) {
            BlackHole.consumeCPU(0);
        }
        BlackHole.consumeCPU(0);
    }
    return o;
}
```

Figure 4.1. Code for the Synchronized looping benchmark.

Allocate-and-hash

Due to the unfortunate side effect of hashcode computations in HotSpot causing revocations it could be of interest to measure scenarios where the identity hashcode is requested on new objects. With biased locking enabled, the current implementation will first revoke the biasable object using a CAS, after which it will install the computed hashcode into the header using another CAS. Without biased locking only one CAS is required (to install the hashcode), and hence there should be a small performance degradation in use cases like this. These revocations are considered cheap enough not to trigger bulk operations, preventing the class from being forced to use thin locking by default which would otherwise prevent the extra CAS for new objects.

To exercise this specific case the very small and simple benchmark in Figure 4.2 can be used, simply instantiating new objects and computing their identity hashcodes.

```
private Object o;
public int allochash() {
        o = new Object();
        return o.hashCode();
}
```

Figure 4.2. Code for the Allocate-and-hash benchmark.

4.1.2 Platforms

The implementation is benchmarked on various platforms to investigate the performance of biased locking on both old and new hardware, where the latency of atomic operations (especially CAS instructions) might vary significantly. The machines used are:

AMD Seoul: 2x6-core AMD Opteron™ 4334 @ 3.1GHz, 4GB RAM, Oracle Enterprise Linux 6

Intel Core: Dual-core Intel® Xeon® 5140 @ 2.33GHz, 4GB RAM, Oracle Linux Server 5.6

Intel Nehalem: 2x6-core Intel® Xeon® X5670 @ 2.93GHz, 94GB RAM, Oracle Linux Server 5.8

Intel Haswell DT: Quad-core Intel® Haswell Desktop (DT) CPU @ 2.7GHz, 8GB RAM, Oracle Enterprise Linux 6

Intel Haswell EX: 2x14-core Intel® Haswell Server (EX) CPU @ 2.2GHz, 64GB RAM, Solaris 11

The evaluation compares the performance of biased locking on the HotSpot JVM and also lazy unlocking on the JRockit JVM. Because some of the benchmarks requires at least Java 1.7 and the JRockit JVM only supports Java 1.6, the benchmarks for JRockit are limited. Only DaCapo and SPECjvm2008 benchmarks are used in this case. The following JVM versions are used:

- Java HotSpot™ 64-Bit Server VM (build 24.51-b03, mixed mode), Java v.1.7.0u51
- Oracle JRockit® (build R28.2.7-7-155314-1.6.0_45-20130329-0641-linux-x86_64, compiled mode), Java v.1.6.0u45

4.2 Results

4.2.1 SPECjvm2008 and SPECjbb2013

The results for SPECjvm2008 and SPECjbb2013 for HotSpot are presented in Figure 4.3. As expected, the impact of biased locking is rather small in large and complex benchmark suites such as these, and hence we see quite small numbers in the results. The SPECjvm2008 results show positive gains from using biased locking on HotSpot, for all of the tested platforms. Nehalem seems to be benefiting the least, suggesting that the cost of CAS instructions are cheaper on the Nehalem platform hence reducing the benefit from biased locking. The largest improvement in these benchmarks is seen on the Haswell desktop (DT) platform, which suggests that CAS instructions on this platform are more costly.

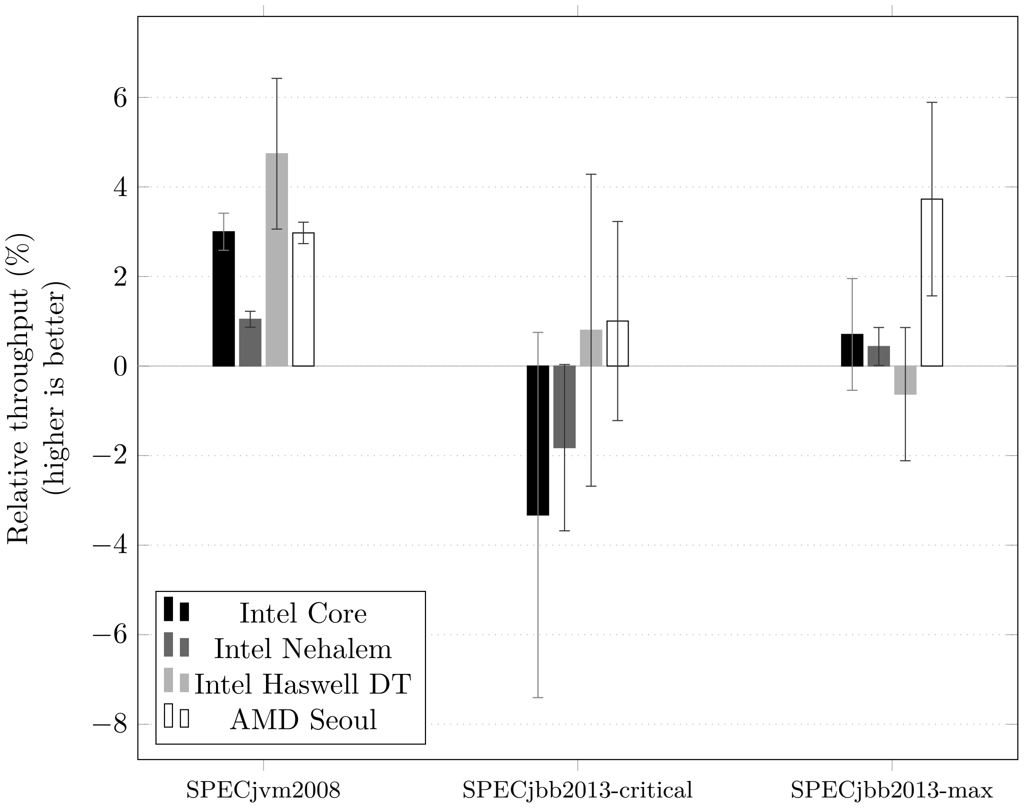

Figure 4.3. Relative change in throughput of SPECjvm2008 and SPECjbb2013 when biased locking is enabled in HotSpot.

SPECjbb2013 results are not as positive as the SPECjvm2008 results. The results varies more from run to run, as is visible from the large standard errors. An interesting aspect here is that while the *critical* score seems to decrease with biased locking on some platforms, the *max* score seems nearly unchanged and sometimes even increasing (see Section 4.1.1 for an explanation of these scores). As the critical score is a measure of throughput *and* response time, it makes sense that biased locking might actually decrease the response time while still maintaining or increasing overall and maximum throughput. When threads attempt to acquire locks that are biased to other threads, the safepoint revocation will prolong the lock acquire, and hence biased locking will decrease response time occasionally. The overall throughput might however benefit from biased locking as the average cost of acquire and release operations should hopefully decrease, which is somewhat reflected by the increases in the max scores.

It is difficult to say whether or not biased locking has an overall positive effect on the SPECjbb2013 benchmark, since the results have such great variance. The critical scores are especially varying, making it difficult to tell what effect biased locking has on these particular results. The max scores do not vary as much as the critical scores but still have significant variation, although they do lean towards a small positive effect from biased locking.

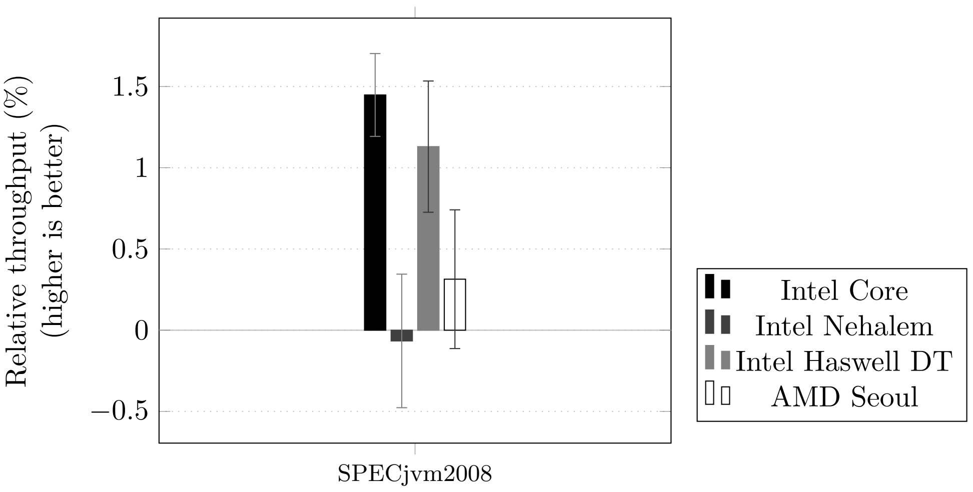

Figure 4.4. Relative throughput of SPECjvm2008 when lazy unlocking is enabled in JRockit.

Figure 4.4 reveals that, although not absolutely positive, JRockit produces similar results as HotSpot for SPECjvm2008. The most significant difference is the lower gains from lazy unlocking. A possible reason to why JRockit gains less from lazy unlocking than HotSpot with its biased locking is that JRockit does not support bulk operations. However, this could also be explained by a faster thin locking mechanism than that in HotSpot, reducing the *relative* gain from the lazy unlocking scheme.

A similar pattern can be observed in the JRockit results as in the HotSpot results regarding differences between the platforms. This confirms that the observed results depend mainly on the underlying platforms, and most likely due to the relative cost of CAS instructions on these platforms.

4.2.2 DaCapo benchmark suite

As expected, the impact of biased locking greatly depends on the application, which is reflected in the varying results from the DaCapo benchmarks (Figure 4.5). Some of the benchmarks are hardly affected at all by biased locking, but one benchmark that does stand out is `fop`, decreasing its execution time by more than 20% on the Intel Core platform. The benchmark most probably consists of frequent thread local locking, explaining the large benefit from biased locking.

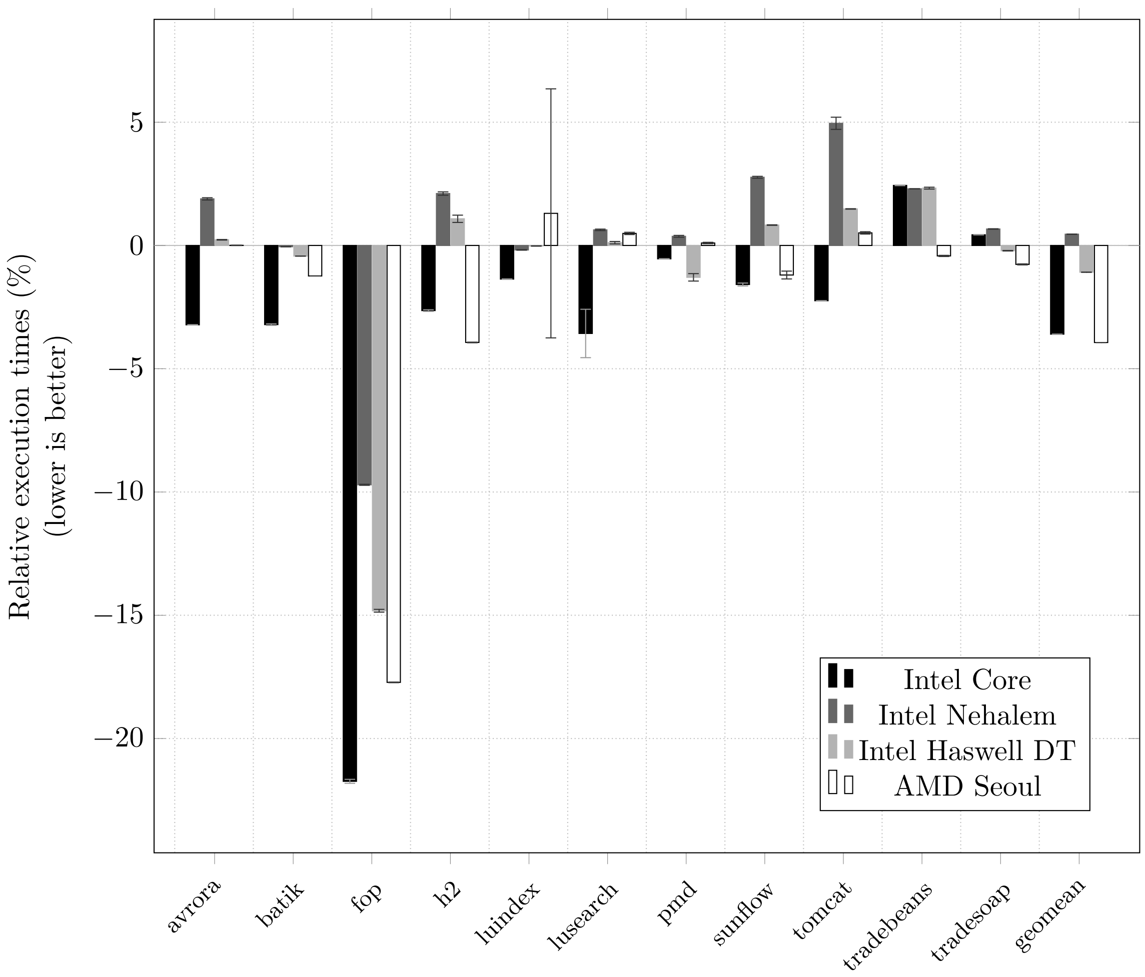

Figure 4.5. Relative execution times of individual DaCapo benchmarks and the geomean when biased locking is enabled in HotSpot.

It is also visible that some benchmarks increase their execution time with biased locking, suggesting revocations are occurring throughout the benchmarks (most visible for `h2`, `sunflow`, `tomcat`, and `tradebeans`). The geometric mean indicates that overall the performance is improved when enabling biased locking on all platforms but Intel Nehalem, which is barely affected at all.

Comparing the differences between the platforms for some benchmarks (especially clear in the `fop` benchmark, and also visible in the geometric mean), a similar pattern as in previous benchmarks can be observed. Intel Core benefits the most, followed by AMD Seoul, Intel Haswell DT and last Intel Nehalem.

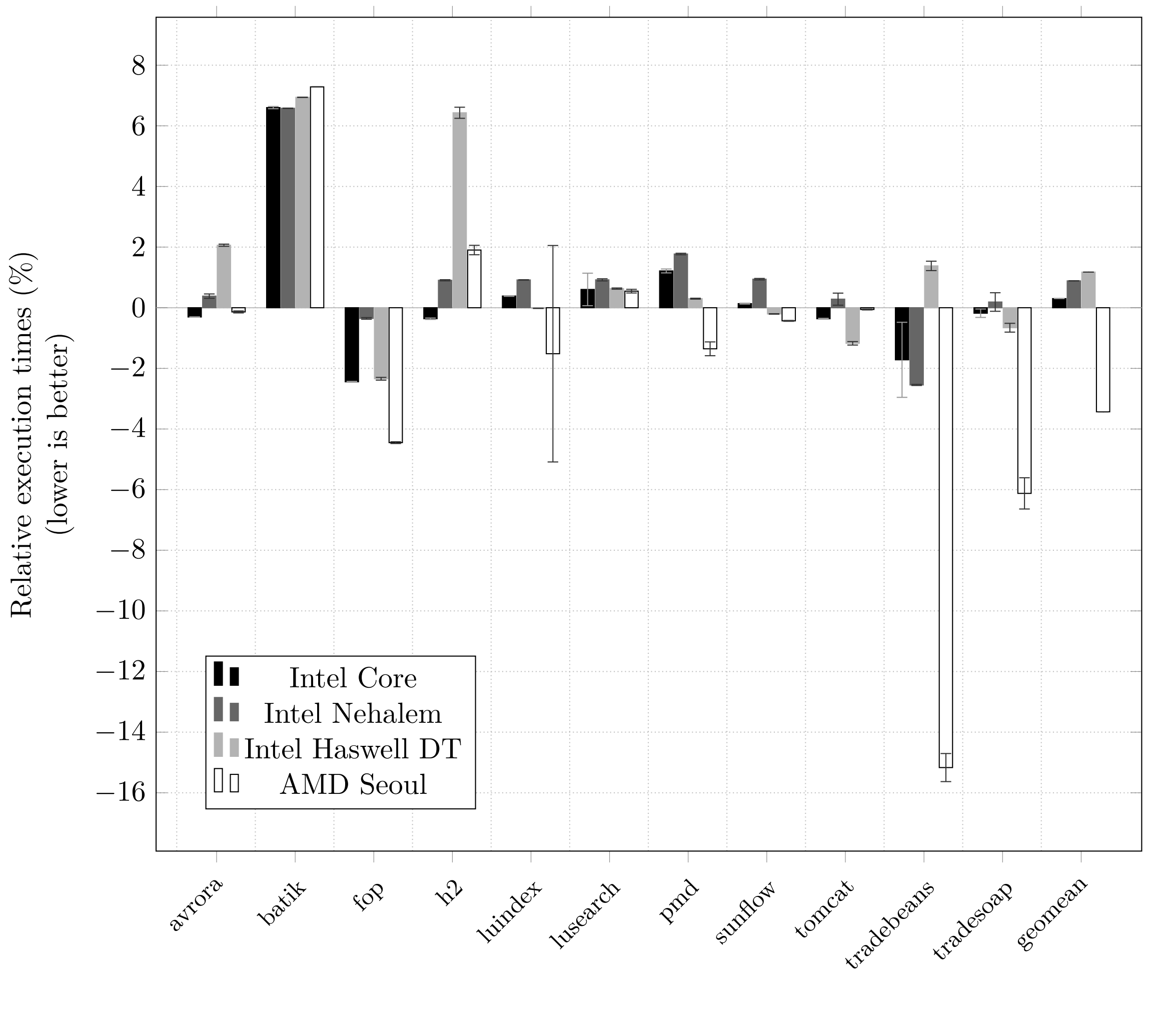

Figure 4.6. Relative execution times of individual DaCapo benchmarks and the geomean when lazy unlocking is enabled in JRockit.

In comparison, lazy unlocking for JRockit is not improving the execution times of the benchmarks to the same extent as biased locking does for HotSpot (see Figure 4.6). Many benchmarks increase their execution times by 1 - 2% with lazy unlocking. The `batik` benchmark increases its execution time more significantly, which when compared to the HotSpot results could be explained by the lack of bulk operations in JRockit.

There are some strange results for the AMD Seoul platform on the last few benchmarks (`tradebeans` especially). Seeing as these results only appear on this platform for these benchmarks and nowhere else, it seems to be the combination of the JRockit implementation and the AMD Seoul platform causing these results. Some part of the lazy unlocking implementation for JRockit in combination with these benchmarks must be especially favorable for this platform, possibly involving better cache locality, instruction scheduling or similar. A possible explanation is that the AMD platform can go from the lazy unlocked state to inflated state much faster than from thin locked to inflated. This would be visible in benchmarks with some significant contention, which is possibly the case for these benchmarks. Comparing the results for these benchmarks on the other platforms and also with the results for HotSpot, it seems very likely that these benchmarks indeed have some contention. In any case, as the results are specific for JRockit and this platform, it does not seem related to the biased locking algorithm, but rather some implementation specific detail in JRockit. As the goal of this project is to investigate the biased locking algorithm and its implementation in HotSpot, further investigating the cause for these results is out of scope for this report.

The geometric mean for JRockit shows both positive and negative effects with lazy unlocking. The impact on AMD Seoul is the most significant, and the other platforms only differ typically by a single percentage. Intel Core seems to benefit very slightly in an overall measure, but the newer platforms such as Nehalem and Haswell are decreasing their overall performance with lazy unlocking.

4.2.3 Micro benchmarks

Table 4.1 presents the benchmark results for the *synchronized looping* microbenchmark for HotSpot. The result table shows relative throughput when biased locking is enabled, both for the benchmark running single threaded, and when 10 threads are running the benchmark concurrently. The multi-threaded measurement is performed to investigate global effects that the biased locking and thin locking algorithms might have.

Table 4.1. Relative throughput with biased locking on HotSpot in a tight synchronized loop.

Platform	Single thread	10 threads
Intel Core	888%	870%
Intel Nehalem	613%	557%
Intel Haswell DT	1017%	1009%
Intel Haswell EX	626%	510%
AMD Seoul	700%	454%

The results show an anticipated great increase in throughput with biased locking. The benchmark is, however, not very realistic as no other work than the actual synchronization is done. This is typically not the case for real world applications where synchronization is only a small fraction of the workload. The benchmark does, however, show differences between the platforms, and gives a very coarse and overestimated upper bound to how much biased locking can improve throughput in applications with frequent thread local locking. Additionally, it is a measure of the cost of CAS instructions on the different platforms, as a higher throughput increase indicates that the CAS is more costly, and a lower increase suggests that the CAS is cheaper.

Again it is clear that Intel Nehalem benefits the least from biased locking, followed closely by Haswell EX. The greatest improvement is gained on the Intel Haswell DT platform, while its server counterpart Haswell EX shows the greatest decrease. A possible explanation to this is that the server platforms have better caches (and possibly better cache coherence protocols) for multi-threaded applications, allowing faster CAS instructions. The relative gain of biased locking would therefore be smaller on the server platform in comparison to the desktop platform.

The multi-threaded measurements show interesting results for the AMD Seoul platform, and partially also for Intel Nehalem and Haswell EX. When the thread count is increased from 1 to 10 the performance gain from biased locking decreases, by almost as much as 250 percentage points for AMD Seoul. This is unexpected as the increased thread count should not really have a negative impact for biased locking on thread local objects. In fact, as CAS instructions sometimes have a negative effect for all threads, one would expect the CAS costs to go up in a multi-threaded setting, and hence the relative gain from biased locking should increase – but instead the opposite is true. One thing these platforms have in common that could possibly explain this behaviour is that they all have multiple CPUs, and not only multiple cores in a single CPU. Due to this, it is very likely the cache coherence protocols differ significantly between single CPU and multiple CPU platforms. In this particular case the results indicate that the CAS instructions are slightly cheaper in the concurrent setting on platforms with multiple CPUs, which would explain the decreased benefit from biased locking.

Table 4.2. Relative change in throughput with biased locking on HotSpot when hashing new objects.

Platform	Throughput
Intel Core	-32,0%
Intel Nehalem	-24,6%
Intel Haswell-DT	-32,2%
Intel Haswell-EX	-28,9%
AMD Seoul	-30,6%

Table 4.2 shows the relative change in throughput for the *Allocate-and-hash* benchmark on HotSpot. Just as expected, the results show a negative impact with biased locking. Due to the additional CAS instruction to revoke the new object the throughput is reduced significantly. Similar to the synchronized looping bench-

mark, this benchmark does not represent a typical real world application. It does, however, reveal the performance decrease for hashcode computations when biased locking is enabled. In other words, applications that are frequently computing the identity hashcode for new objects will see a performance decrease, although most probably smaller than in this benchmark. Furthermore, it is worth mentioning that well written classes typically override the hashcode method, removing the need for identity hashcode computations.

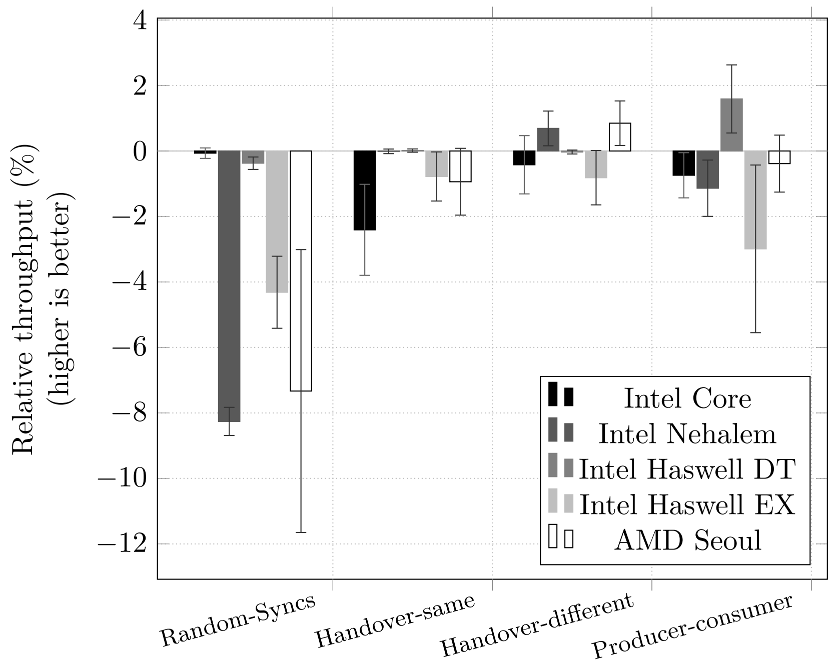

Figure 4.7. Relative throughput of micro benchmarks when biased locking is enabled in HotSpot.

Figure 4.7 shows the results for the remaining microbenchmarks. The expected behaviour for the *RandomSyncs* benchmark is that most locking will not be contended as only 10 threads will randomly synchronize on 10^6 objects. After a while, most objects will be biased to some thread, which then causes revocations. In fact, most of the time synchronizations should cause revocation, and only rarely should a thread synchronize an object that is already biased to itself. The revocations will quickly provoke a bulk revocation, after which thin locking is used instead. Because of this we should see limited performance degradation, if any at all. This seems to be the case for Intel Core and Haswell DT, yet the other platforms perform less ideal. Again, it is the results for the multiple CPU platforms that stand out. Just as before, in the *Synchronized looping* microbenchmark, this benchmark has 10 threads running in parallel. The negative results confirms that thin locking increases its relative performance compared to biased locking when used in a parallel setting. A notable difference in this benchmark is, however, that biased locking only adds additional overhead and decreases the performance relative to thin locking.

The results for the *Handover-same* and *Handover-different* benchmarks do not show very significant changes. As the whole benchmark consists of synchronization alone one would expect to see larger effects of the biased locking scheme. The results, however, indicate that the bulk revocation algorithm quickly bulk revoke the biases on these objects, after which thin locking is used instead. This is also reflected in the results, as there is hardly any difference when biased locking is enabled. The small differences we can see might depend on the benchmarking overhead rather than the actual benchmark itself.

Results for the *Producer-Consumer* benchmark indicate a small impact of biased locking. Most platforms lean towards decreased performance except for Intel Haswell DT, and again we see the two Haswell platforms with contradicting results. However, the large variations in the results make it difficult to draw an overall conclusion of how biased locking impacts the performance in this benchmark.

4.3 Analysis

As locking is only a small part of any typical application, such is the case for benchmark suites as well. Modifications to the locking implementation will have a proportional effect to the amount of concurrency and locking in each benchmark. It is therefore only natural to see limited effects of biased locking in certain benchmarks, while other benchmarks reveal significant effects. Even in very concurrent and locking-intense benchmarks the majority of the workload will probably still not be locking, but rather some computations specific to the benchmark. Because of this, one should not expect to see changes to the biased locking implementation increasing the JVM performance by several factors, but rather expect improvements or differences in the range of some lower, yet significant, percentage.

Although a few of the benchmark results are somewhat inconclusive, there are still a some observations that can be made. For most benchmarks, there is a clear and similar pattern when comparing the results for the different platforms on the two JVMs (HotSpot and JRockit). This pattern is especially visible in the SPECjvm2008 and DaCapo results. The pattern suggest that Intel Core benefits greatly from biased locking. This makes sense, since the platform is oldest in these benchmarks, making it very likely to have the most expensive CAS instructions. The AMD Seoul platform produces similar results as Intel Core, occasionally benefiting even more from biased locking. The Nehalem platform seems to have reduced the cost of atomic CAS instructions significantly. As is visible in Table 4.1, Figure 4.3 and 4.4 among others, Nehalem consistently produces results that indicate very small gains in performance from biased locking – occasionally even negative. This is a natural consequence of reduced CAS instruction costs, causing the biased locking mechanism to add unnecessary complexity. The Intel Haswell EX platform seems to keep the inexpensive CAS instructions from Nehalem, as indicated by the microbenchmark results. The desktop version of Haswell (DT) is, however, not consistent with these results, and produces results very similar to the Intel Core platform. Table 4.1 indicates CAS instructions being about 10 times slower than the instructions required for biased locking, suggesting that CAS instructions are still very costly on this platform.

Overall, it seems like biased locking has a positive effect on performance for all of the involved platforms. While the Intel Nehalem platform does not benefit as much as the others, it still does not have very significant reductions in performance. In applications such as DaCapo's `fop` with frequent thread local locking, Nehalem still shows very good performance improvements using biased locking. In contrast, the few cases where the performance is affected negatively, the decrease is only by a few percent.

An interesting observation mentioned earlier is that the cost of the CAS instructions seems to be much higher in the desktop version of the Haswell platform than in the server counterpart. This is likely due to different cache coherence protocols for server and desktop platforms. It is reasonable to assume that server platforms have different cache coherence protocols in order to support both multiple CPUs and more CPU cores. The benchmark results indicate that these protocols allow faster CAS instructions. This would mean that desktop platforms will still have relatively costly CAS instructions, even on newer platforms, and hence continue benefiting from biased locking. In other words, biased locking seems especially beneficial for desktop platforms.

Some of the microbenchmarks were not really successful in finding significant impacts of the biased locking algorithm. In particular *Handover-same*, *Handover-different* and *Producer-consumer* seemed to provoke too many revocations, causing thin locking to be used instead, hence not really measuring the effect of the bulk operations as was intended. This reveals a difficulty to microbenchmark the bulk operations (other than a single bulk revocation, of course), since the heuristics thresholds involve the decay timer that will prevent biasing completely if revocations happen too often. Lowering the decay time in order to run such benchmarks would result in massive amounts of revocations, and in comparison thin locking would be much faster. Benchmarks for evaluating the bulk operations would need to be larger than a typical microbenchmark, possibly running with more serious workloads and for longer durations.

Chapter 5

Improving the implementation

Using the idea of adaptive biased locking with learning as described in Section 2.2, an improvement to the current implementation is investigated and implemented. The moderately simple implementation of this improvement is the main motivation to why this particular improvement is chosen.

5.1 New algorithm description

5.1.1 Mark word

To support learning, an additional state for biased locks is necessary. In order to distinguish this from the biased state, an additional bit is required in the mark word. Figure 5.1 illustrates the new mark word layouts, where the biased state has been split into two separate states, using the 11th least significant bit in the mark word to distinguish them. If the bit is 0, this indicates the lock is truly biased, just as in the original implementation. If the bit is 1, it means the lock is in bias learning mode, and the locking thread must use thin locking until the lock is biased.

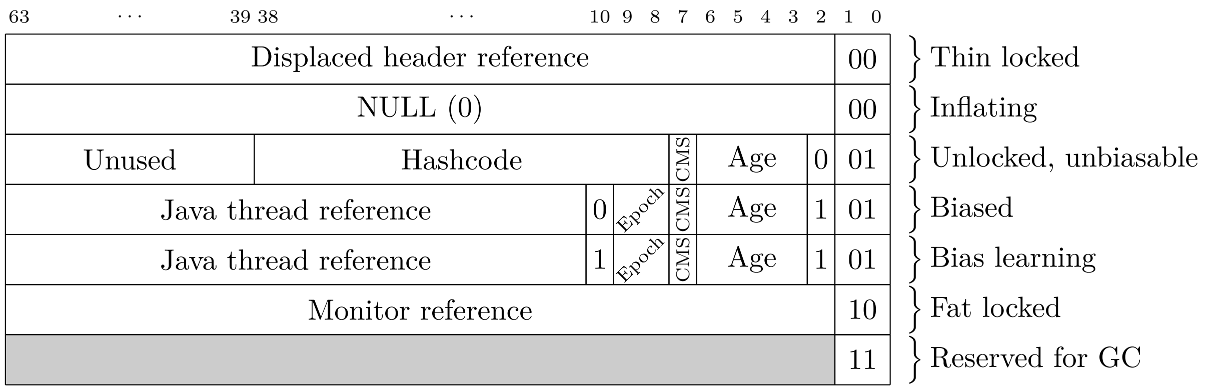

Figure 5.1. New 64-bit mark word's states and layouts.

The additional bit required for this is taken from the thread reference field, adding additional constraints to thread metadata allocation as threads must be aligned to addresses that end with 11 zeros, rather than the previous 10. (Otherwise the thread reference will not fit in the reserved area in the mark word.) This might cause the thread allocations to require more memory, but should have no impact on performance.

5.1.2 Algorithm

Acquiring a lock in the bias learning phase is done using thin locking. However, instead of displacing the mark word unmodified, the thread inserts its thread reference into the mark word as it displaces it. The next time the thread acquires the lock, it will see if it was the previous thread acquiring that lock, and make the biasing decision based on that.

As a very first step of improving the implementation, the learn mode is introduced without any sort of counter, biasing locks whenever the acquiring thread finds its own thread reference in the mark word. This means that threads will acquire each lock at least once using regular thin locking before biasing them, corresponding to a bias threshold of 1. While a higher threshold might be desirable, this implementation is as simple as it can get. It will at least avoid the scenarios where a lock is acquired by a different thread every other time, which would normally cause revocations on each acquire. Also, this implementation could very easily be extended with a counter, either in the mark word or in a separate location, or by using random counting as explained in Section 2.2.

A notable difference to the original implementation is that whenever revocations occur, no immediate rebiasing of the object is allowed. Instead, upon revocation the object is truly revoked instead of rebiased, after which the object must be locked using thin locking. Alternatively one could reset the object to the learning state, somehow representing rebiasing by enforcing a new learning phase.

Also, whenever an invalid epoch is encountered the lock is considered to be in learning mode again (provided it still has biasing enabled). This means that when bulk rebias operations occur, any biased locks not held by their owner will revert to learning mode. If the previous owner wants to acquire the lock again, it will have to do so using thin locking once, before it can successfully rebias the lock to itself again.

A simplified transition graph of the lock states for the new algorithm can be seen in Figure 5.2. Just like before the transitions for `wait`, `notify` and `hashCode` are omitted. Additionally, the unlocked state without hashcode is merged with the unlocked state with a hashcode, in order to simplify even further. As mentioned, the rebiasable state will now not directly transition to the biased state but instead it has to go through the learning phase again.

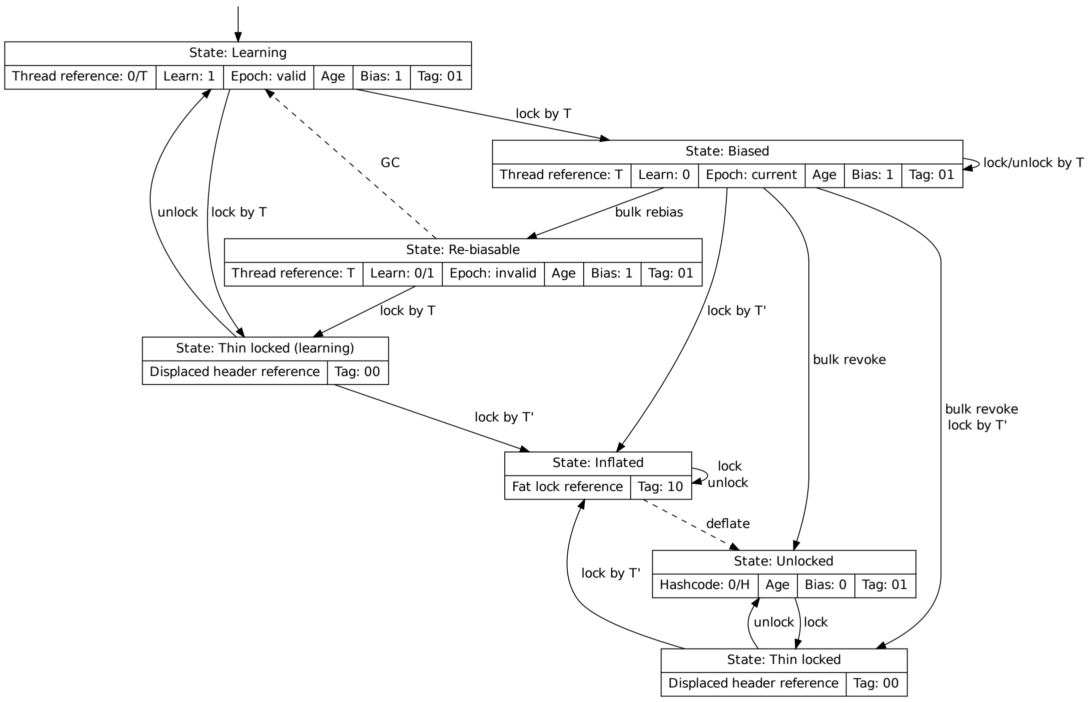
State: Learning
Thread reference: 0/T | Learn: 1 | Epoch: valid | Age | Bias: 1 | Tag: 01
lock by T
State: Biased
Thread reference: T | Learn: 0 | Epoch: current | Age | Bias: 1 | Tag: 01
lock/unlock by T
GC
unlock
lock by T
bulk rebias
State: Re-biasable
Thread reference: T | Learn: 0/1 | Epoch: invalid | Age | Bias: 1 | Tag: 01
lock by T
lock by T'
State: Thin locked (learning)
Displaced header reference | Tag: 00
bulk revoke
bulk revoke
lock by T'
lock by T'
State: Inflated
Fat lock reference | Tag: 10
lock
unlock
deflate
lock by T'
State: Unlocked
Hashcode: 0/H | Age | Bias: 0 | Tag: 01
unlock
lock
State: Thin locked
Displaced header reference | Tag: 00

Figure 5.2. Simplified state graph of the new algorithm.

5.2 Results

A prototype of the improved algorithm is implemented in HotSpot. The complexity of the current HotSpot implementation makes it difficult to implement the new algorithm fully functioning, however. The prototype therefore has some implementation issues, which causes inconsistent lock states occasionally. This only seems to occur in the more involved and complicated locking scenarios. As such, the implementation is unable to run the larger benchmarks such as the SPEC or DaCapo suites. However, the implementation manages to run the microbenchmarks without problems. To at least verify the benefits of a learning phase to a lesser extent, and possibly investigate any negative effects from learning, the prototype is benchmarked with some microbenchmarks described in the following sections.

Just like before, the benchmarks are run using JMH. The two platforms reused for these benchmarks are Intel Nehalem and Intel Haswell DT. Nehalem benefited the least from biased locking while Haswell DT showed some of the most substantial performance gains. Therefore it seems reasonable that these two platforms should provide sufficiently interesting results to evaluate the improved implementation.

5.2.1 Different use patterns

Since the bulk rebias and revocation mechanism works at a class granularity, different use patterns for objects of the same class might cause one part of the application to prevent the other part from using biased locking. If two threads are frequently synchronizing on shared objects, this will eventually cause bulk revocation, preventing all other threads from using biased locking, even though these synchronizations might be completely thread local.

The learning phase will help prevent this, as the shared synchronizations will hopefully be detected before any of the two threads successfully bias the locks, which in turn will prevent the necessity of revoking those biases. This allows all other threads to continue using biased locking.

To exercise this scenario, a microbenchmark is designed as follows:

- Two threads take turns to synchronize on 10^4 shared objects
- Concurrently with the above, a third thread repeatedly synchronizes on 10^4 different, non-shared objects of the same class as above

Just like with previous microbenchmarks, the only workload in this benchmark is the actual synchronization.

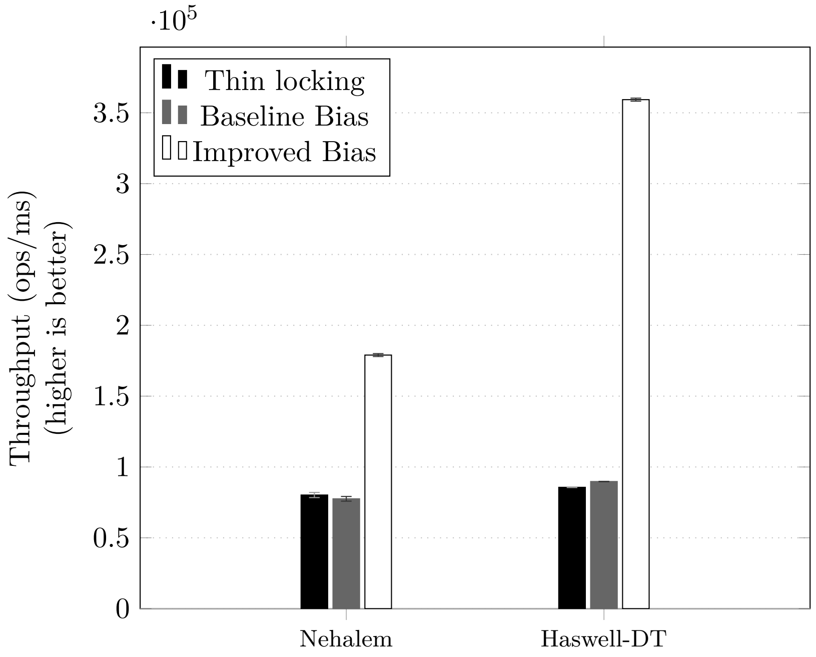

Figure 5.3. Benchmark results for the different use pattern microbenchmark.

Just as anticipated, the results in Figure 5.3 reveals significant improvement over the original biased locking implementation. The learning phase successfully prevents revocations on the shared objects, preventing bulk revocation from occurring, and allowing the third thread to continue bias locking its objects. Naturally, the throughput is expected to increase even further if additional threads working on non-shared objects are added to the benchmark.

5.2.2 Synchronized looping

It might be of interest to measure the additional cost of the learning phase to some extent. To accomplish this, the *Synchronized looping* microbenchmark is reused.

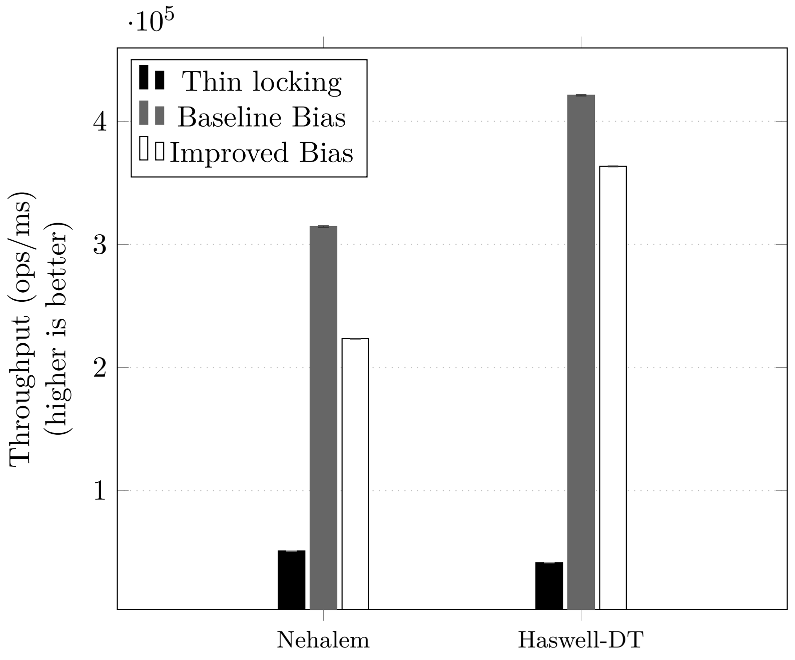

Figure 5.4. Benchmark results for the synchronized looping microbenchmark.

The additional overhead of the learning phase is visible in Figure 5.4. Although significantly better than thin locking, the improved biased locking implementation performs worse than the original implementation in this benchmark. This is not really unexpected, since the improved algorithm requires the locks to be thin locked once before biased. This requires 3 CAS instructions before the lock is biased, whereas the baseline biased locking implementation only requires a single CAS. In a benchmark such as this, with no other workload than the synchronization and only thread local locks, it is obvious that the baseline implementation should perform better.

5.2.3 Random synchronization on shared objects

The benchmark involving random synchronization on shared objects from before is reused to evaluate the improved implementation. The frequent revocations observed in the original implementation could possibly be avoided with the learning phase. Objects should rarely be biased but rather stay in the learning phase.

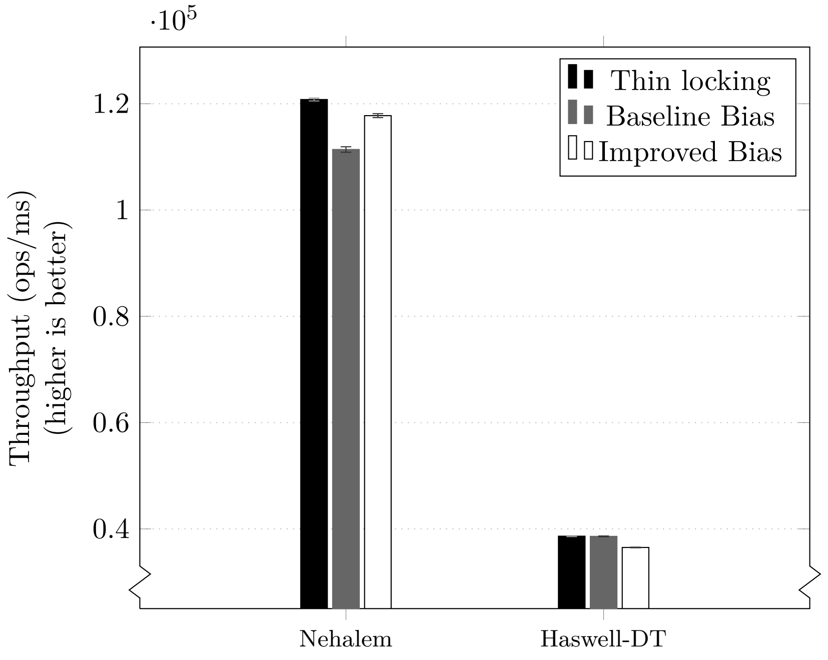

Figure 5.5. Benchmark results for the random synchronization microbenchmark. Note the discontinuity in the y axis.

In Figure 5.5 it is visible that the improved implementation performs slightly worse than the baseline with thin locking. The slightly decreased throughput can be explained by the additional overhead introduced by the learning phase. Comparing the results for Nehalem with that of Haswell DT one can see that the improved implementation performs better than the baseline bias on Nehalem, but worse on Haswell DT. This confirms the fact that the slightly reduced performance is due to the additional CAS instructions required, since Nehalem has cheaper CAS instructions compared to Haswell DT.

5.3 Analysis

The improved algorithm performs as expected. In cases similar to the *different use case benchmark* it could potentially increase performance greatly. Unfortunately the prototype is unable to run the larger benchmark suites that would give a more overall and realistic performance measurement. There is no evidence of significantly decreased performance with the improved algorithm, yet the microbenchmarks suggest improved throughput in certain cases. It is therefore reasonable to assume that larger benchmark results would show little to no performance decrease, and occasional moderate performance improvement.

The additional overhead from the learning phase seems to be small. The most significant extra cost of the learning phase is of course the extra CAS instructions required for the initial thin locking. Cheaper CAS instructions, such as the Intel Nehalem platform demonstrates, diminishes the extra cost of the learning phase, making it even more appealing.

Another interesting aspect of the learning phase is that it can possibly avoid performance issues from biased locking during JVM startup. This would then replace the current heuristics with biased locking disabled for the first few seconds of the JVM. The current prototype will probably not avoid these issues completely, as the threshold for biasing is only 1. An extension of the improved implementation could, however, include support for higher thresholds, and if properly tuned this could avoid issues with startup performance using the learning phase.

A promising extension to the improved algorithm is to introduce different biasing thresholds. Each class could have a separate threshold to allow data types that are more frequently used thread locally to be biased more quickly. At the same time objects that are often shared could require a much longer learning phase before allowing threads to bias such locks. Additionally, these thresholds could be modified dynamically as required. Whenever a revocation occurs the threshold could then be increased, hopefully reducing the amount of future revocations. Similarly when a large amount of objects have been biased without any revocations occurring the threshold could be decreased, allowing new objects of that class to be biased much faster.

Chapter 6

Conclusions

The biased locking implementation in HotSpot has been investigated and evaluated with benchmarks of different types. The benchmarks include standardized benchmark suites from SPEC as well as the DaCapo benchmark suite, and finally some custom microbenchmarks. Decreased costs of atomic CAS instructions on new CPU platforms diminishes the usefulness of biased locking. Benchmark results, however, still show benefits from using the biased locking algorithm, even on the most recent platforms. The benefits are smaller on platforms with inexpensive CAS instructions, but in an overall measure it is still found beneficial. Additionally, it is discovered that biased locking seems to show better results on desktop platforms than on server platforms. This is most likely an indication of CAS instructions being cheaper on server platforms, explaining a smaller benefit from biased locking on these platforms.

A small implementation specific pathology is found, where biased locking might decrease performance in applications with frequent identity hashcode computations on new objects. This also affects applications without any actual synchronizations, since it does not depend on the synchronization of some object. The obvious remedy for this is to combine the revocation with the hashcode insertion, requiring only a single CAS to perform the two operations.

Some work towards improving the implementation is done, trying to combine *biased locking using learning* with the existing *store free biased locking with bulk rebias and revocation* in HotSpot. While the implementation is only a prototype and not completely functional, smaller experiments with benchmarks that avoid implementation issues indicate positive effects from the improvements to the biased locking implementation. A learning phase can potentially reduce the amount of revocations while still exploiting the thread locality of locks. The learning phase seems like a good improvement of the biased locking implementation in HotSpot, but further research is recommended before adapting it.

6.1 Future work

Initially, it would be a good idea to remedy the issue with the hashcode computation pathology. This is most effectively fixed by combining the revocation with the hashcode insertion. It could also be of interest to investigate the possibility of allowing biased locking for hashed objects as well. This would require the hashcode to be stored differently, either in a separate location, or by using an alternative layout of the mark word that supports both thread reference and hashcode at the same time.

It would also be interesting to fully implement the biased learning prototype in order to run the larger benchmark suites. With such working implementation one could investigate the possible extensions to the algorithm as well. This includes the mentioned dynamically adjusted bias thresholds with class granularity. The counters required for tracking successful learning phase lock attempts would either be implemented using an actual counter, alternatively using *random counting* with a software pseudo random number generator or probabilistic branching (available with Intel Haswell). This choice of counter implementation would also be interesting to investigate.

A somewhat related improvement worth investigating is the support for safepoints involving only specific threads. This would allow revocations to pause only the biased thread rather than all threads in the JVM, which would make biased locking scale better and reduce the cost of revocations.

Bibliography

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 207–222, New York, NY, USA, 1999. ACM.

[2] D. F. Bacon, R. Konuru, C. Murthy, and M. J. Serrano. Thin locks: Featherweight synchronization for java. *SIGPLAN Not.*, 39(4):583–595, Apr. 2004.

[3] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[4] P. A. Buhr, M. Fortier, and M. H. Coffin. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, Mar. 1995.

[5] D. Dice. Implementing fast Java (tm) monitors with relaxed-locks. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 13–13, Berkeley, CA, USA, 2001. USENIX Association.

[6] D. Dice, M. Moir, and W. S. III. Quickly reacquirable locks. Technical report, Sun Microsystems, 2003.
`http://home.comcast.net/~pjbishop/Dave/QRL-OpLocks-BiasedLocking.pdf`.

[7] A. Fog. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2014-02-19 edition, 2014. `http://www.agner.org/optimize/instruction_tables.pdf`.

[8] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java® Language Specification, Java SE 7 edition.* Oracle America, July 2011. Final version.
`http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf`.

[9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375.

[10] M. Hirt and M. Lagergren. *Oracle Jrockit: The Definitive Guide.* Packt Publishing Ltd, Olton, Birmingham, UK, 1st edition, 2010. ISBN 9781847198068.

[11] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 130–141, New York, NY, USA, 2002. ACM.

[12] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 223–237, New York, NY, USA, 1999. ACM.

[13] T. Onodera, K. Kawachiya, and A. Koseki. Lock reservation for java reconsidered. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 559–583. Springer Berlin Heidelberg, 2004.

[14] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 171–181, New York, NY, USA, 2011. ACM.

[15] I. Rogers and B. Iyengar. Reducing biased lock revocation by learning. In *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '11, pages 7:1–7:10, New York, NY, USA, 2011. ACM.

[16] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 263–272, New York, NY, USA, 2006. ACM.

[17] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 65–74, New York, NY, USA, 2010. ACM.