Fine-grained Adaptive Biased Locking

Filip Pizlo
Purdue University
pizlo@purdue.edu

Daniel Frampton
Australian National University
daniel.frampton@anu.edu.au

Antony L. Hosking
Purdue University
hosking@cs.purdue.edu

Abstract

Mutual-exclusion locking is the prevailing technique for protecting shared resources in concurrent programs. Fine-grained locking maximizes the opportunities for concurrent execution while preserving correctness, but increases both the number of locks and the frequency of lock operations. Adding to the frequency of these operations is the practice of using locks *defensively* — such as in library code designed for use in both concurrent and single-threaded scenarios. If the library does not protect itself with locks, an engineering burden is placed on the library's users; if the library does use locks, it punishes those who use it only from a single thread. *Biased* locking is a dynamic protocol for eliminating this trade-off, in which the underlying run-time system optimizes lock operations by biasing a lock to a specific thread when the lock is dynamically found to be thread-local. Biased locking protocols are distinguished by how many opportunities for optimization are found, and what performance trade-offs for non-local locks are experienced. Of particular concern is the relatively high cost involved in revoking the bias of a lock, which makes existing biased locking protocols susceptible to performance pathologies for programs with specific patterns of contention.

This work presents the biased locking protocol used in Jikes RVM, a high-throughput Java virtual machine. The protocol, dubbed FABLE, builds on prior work by adding per-object-instance dynamic adaptation and inexpensive bias revocation. We describe the protocol, detail how it was implemented, and use it in offering the most thorough evaluation of Java locking protocols to date. FABLE is shown to provide speed-ups over traditional Java locking across a broad spectrum of benchmarks while being robust to cases previous protocols handled poorly.

1. Introduction

Mutual-exclusion locking is the prevailing technique used for protecting shared resources in concurrent programs. This is particularly true in managed languages such as Java and .NET, where locking is built into the language syntax. While this encourages programmers to use locks more freely than in C-like languages, it also creates problems for language implementers. Java and .NET alike mandate that any object can act as a lock at any time, imposing additional per-object storage for lock state. Since programmers are given easy access to locking, they end up using it, often more so than is prudent from a performance standpoint. This has led to a number of innovations that attempt to address both the space usage and performance [1, 2, 11–13, 19] of Java locking. A particularly powerful technique is that of *biased locking*, which reduces the overhead of acquiring a lock if that lock is found to be thread-local. But biased locking has its own trade-offs. Thread-locality is inferred dynamically using heuristics. If these turn out to be wrong, it can be expensive to revoke the bias.

This work presents the biased locking protocol used in Jikes RVM.[1] The protocol, which we call FABLE, is based on learning, and adapting to, the behavior of each lock individually. We demonstrate that FABLE uses intuitive and robust heuristics for choosing which locks to bias, making it an easy protocol to reproduce in other systems. FABLE's heuristics make it resilient against performance pathologies that derail other biased locking protocols. Furthermore, FABLE builds on prior work by implementing an inexpensive revocation in case the heuristics fail; this revocation protocol is by itself cheap enough to make biased locking a net performance win on all commonly available Java benchmarks. In addition to presenting a new protocol, we present a detailed technical review of previous locking protocols for Java by showing their implementation in Jikes RVM, a high-performance JIT-based Java virtual machine, and offer perhaps the most thorough performance evaluation of Java locking protocols to date. The main results of our evaluation are (*i*) confirming the results of Kawachiya et al. [11], (*ii*) validating FABLE's throughput, (*iii*) demonstrating FABLE's resilience against common design patterns that cause disastrous performance pathologies (slow-downs in excess of 50×) in previous biased locking protocols, and (*iv*) that the relative gains of *any* biased locking strategy depend heavily on the choice of hardware.

Section 2 reviews previous implementations of locking in Java, Section 3 presents our fine-grained adaptive biased locking protocol (FABLE), Section 4 gives a qualitative comparison of the FABLE approach and prior art, Section 5 presents our experimental validation, and Section 6 concludes.

2. Java Synchronization

The `synchronized` statement is the default locking mechanism for Java. The statement executes a block of code while holding a lock — the lock is acquired prior to block execution, and released after. The Java language mandates that any object be able to act as a lock, so any object can be used in a synchronized statement. This lends itself well to concurrency-aware software design, but also has its disadvantages. First, the underlying implementation must allocate enough space to hold a lock for each object. Second, programmers often end up using more synchronized statements than necessary, causing lock and unlock operations to dominate application execution time. Moreover, many Java libraries (such as those in the collections framework) are made thread-safe by

[1] We released an early version of our biased locking protocol as part of Jikes RVM 3.1. This article is the first to document how Jikes RVM performs biased locking.

PPPJ '11, August 24–26, 2011, Kongens Lyngby, Denmark.

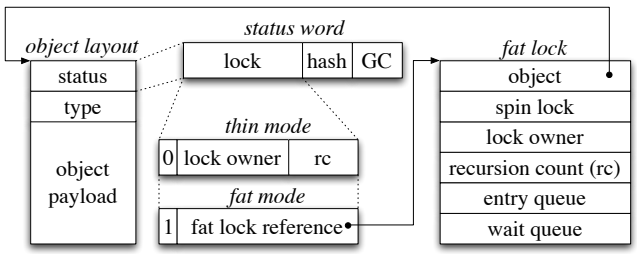

Figure 1: **Thin lock object layout in Jikes RVM.** We assume a two-word object header. The first word is the *status word*, with bits for supporting hash codes, garbage collection, and locking. The second word is the *type word*, holding a pointer to the object's type. The status word's lock data has a high-order bit that indicates if the lock is thin (0) or fat (1); if it is thin, the lock word contains bits identifying the thread that owns it (or zero if none), and for the recursion count to support Java's re-entrant (monitor) locks. If these bits are insufficient to represent the lock's state, then the lock is inflated into fat mode by allocating a structure that has enough storage to contain all of the necessary information.

Value	Purpose
`LOCK_MASK`	Bitmask for lock portion of status word.
`STATE_MASK`	Bitmask for lock state (thin or fat).
`STATE_SHIFT`	Amount to shift to get to lock state.
`OWNER_MASK`	Bitmask for lock owner.
`OWNER_SHIFT`	Shift needed to get to lock owner.
`RC_MASK`	Bitmask for recursion count (RC).
`RC_SHIFT`	Shift needed to get to recursion count.
`UNLOCK_MASK`	Inverse of `OWNER_MASK|RC_MASK`, used to clear owner and RC fields.
`obj->status`	Status word for `obj`.
`thr->lockBits`	Current thread's locking identifier, shifted according to `OWNER_SHIFT`.

Table 1: **Constants and fields used by thin locks.** As shown in Figure 1, each object has a *status word* in its header, and some portion of the status word is devoted to thin lock data. The `LOCK_MASK` masks off just the lock data, while the other shifts and masks are used to access the four possible fields of the lock data: the state, which is either 0 (thin) or 1 (fat), and the owner and recursion count for thin locks.

default, using locking even when programmers never share the underlying data structures among multiple threads.

These disadvantages have motivated a number of novel implementation strategies for Java locks. The first problem — space usage for the lock — was almost completely side-stepped by compressing the lock into a handful of bits in the object header [1, 2, 9, 12]. Such *thin lock* approaches exploit the observation that most locks only require a handful of bits; it is only when the lock is experiencing heavy contention that additional storage is needed. The default mode of a thin lock involves the entire lock state — whether or not it is held, and if so, by whom — being packed inside a single 32-bit word in the object header. If the locking protocol requires more storage, it reuses those bits for a reference to a *fat lock* structure. Figure 1 shows the Jikes RVM object layout, using thin locks to compress lock data in the common cases.

The second problem — time overhead of lock and unlock operations — continues to inspire new locking protocols. The lock operation in C-like languages is typically implemented by an out-of-line procedure whose fast path involves one or more atomic operations such as compare-and-swap (CAS). While this approach is usable for Java, it leads to reduced performance when synchronized statements are executed frequently. Thus, modern Java lock implementations inline the lock fast path because an inlined CAS is much faster than a procedure call.

```
void inlineLock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK) == 0
      && CAS(&obj->status, oldStatus,
           oldStatus + curThread->lockBits
                + (1<<RC_SHIFT)))
      return;
   lockSlow(obj);
}

void inlineUnlock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK)
       == curThread->lockBits + (1<<RC_SHIFT)
      && CAS(&obj->status, oldStatus,
           oldStatus & UNLOCK_MASK))
      return;
   unlockSlow(obj);
}
```

Figure 2: **Inline lock and unlock paths for thin locks.** We show the Jikes RVM thin locking fast paths. The `inlineLock` procedure covers the most common case for locking: the lock is thin and not held by anyone. This corresponds to all-zero lock bits in the status word (see Figure 1). The `inlineUnlock` procedure covers the most common case for unlocking: the lock is thin, and held by the current thread exactly once. Both locking and unlocking required compare-and-swap to cover races against other threads locking the same object, and other system services that may also be accessing the object status word. These *fast paths* are inlined into user code by the optimizing compiler. Other cases are handled by the out-of-line slow paths (`lockSlow` and `unlockSlow`).

2.1 The Inline Thin Lock Protocol in Jikes RVM

We now discuss the original inline thin lock protocol used in Jikes RVM; this serves as the basis for our description of other locking protocols. Jikes RVM uses the object layout shown in Figure 1: objects have a two-word header, which includes a multi-purpose *status word* and a *type word* that refers to the object's type. In Java, each object has a lock and a system-supplied hash code, and unreachable objects are garbage collected. The status word contains bits for supporting each of these capabilities. We do not go into the details of the hashing and garbage collection implementation in Jikes RVM other than to say that the hash code support is address-based (with special support for hashed objects that move from their first-hashed address), and the garbage collector is parallel, generational, copying, accurate, and stop-the-world by default, but has a number of variants including concurrent collection. The lock portion of the status word uses 22 bits total, most of which comprise the lock owner field, which identifies the thread that currently owns the lock.

Figure 2 shows the inline locking code paths, and Table 1 describes some of the code conventions we use. The inline thin lock procedures cover the common cases for lock and unlock: for lock acquisition the common case is that the lock is thin and not held, while for lock release (unlock) the common case is that the lock is thin, the current thread owns it, and we are in the outermost synchronized statement for that lock (i.e., the current thread has not acquired the lock recursively). All other cases are handled by out-of-line slow paths. The lock slow path covers the following cases:

Recursive lock acquisition: If the lock is already held by the current thread, the recursion count is incremented and checked for overflow.

Recursion count overflow: If the RC bits in the status word are too few to account for the number of times that the lock has been acquired, then the lock is *inflated* into fat mode.

Thin lock spin-acquire: If the lock is thin and is held by another thread, the slow path first makes 40 attempts to acquire the lock by spinning. Spinning includes telling the operating system to yield the current thread. The most common case for lock contention is that it is acquired after spinning (i.e., the previous holder releases it within 40 spins).

Spin limit overflow: If a thin lock cannot be acquired after 40 spins, the lock is inflated into fat mode.

Fat lock acquire: If the lock is fat, lock acquisition proceeds much like a standard mutex: if it is already held by another thread, the acquiring thread is enqueued and suspended. Atomicity for internal fat lock operations — enqueueing and dequeuing threads that are waiting on the lock — is ensured using a spin-lock internal to the fat lock.

Similarly, the slow path for unlock covers:

Recursive lock release: The lock is held by the current thread and the recursion count is greater than 1, so the recursion count is decremented.

Lock not held by current thread: If the lock is not held by the current thread, an `IllegalMonitorState` exception is thrown.

Fat lock release: If the lock is fat, the lock release proceeds by dequeueing and waking a suspended thread from the lock's entry queue. If there are no threads on the fat lock's queues and the recursion count is zero, the lock is *deflated*. This eager deflation strategy maximizes overall performance by minimizing the likelihood of fat lock operations, which require out-of-line procedure calls.

The Jikes RVM thin lock algorithm is largely based on the *Tasuki lock* of Onodera and Kawachiya [12] in that it deflates locks eagerly, but omits their CAS-less unlock optimization because other threads may be accessing the status word concurrently (for hashing and GC-related operations). To validate the performance of our thin lock implementation, we have extended Jikes RVM with a toolkit for experimenting with various Java locking protocols (which we make publicly available [16]). This includes a variety of locking protocols, including ones that change the fat lock deflation strategy, and others based on futexes [10] as well as `java.util.concurrent.lock.ReentrantLock`. Measurements that we have performed on this framework show that while some variations may lead to performance progressions on individual benchmarks, the geometric mean over all of our benchmarks does not vary significantly. This has led us to believe that our thin locking implementation is near optimal. But there is a case where thin locks *can* be further optimized. For some Java workloads, the frequency of *uncontended* lock acquisition is so high that even inlining of the fast path is not enough. The problem stems from the hardware implementation of CAS. CAS can be quite slow — often an *order of magnitude slower* than a normal memory access. For example, even on Intel's Nehalem processors, lock operations implementing CAS still serialize the whole pipeline.

2.2 Biased Locking

The CAS in lock/unlock fast paths can be avoided if the lock is biased to a thread. This thread becomes the *bias owner*, and can use a fast path that relies on non-atomic loads and stores. If another thread attempts to acquire the lock, a special *bias revocation* protocol is invoked. This protocol must avoid entering into a race condition with the non-atomic lock acquisition path used by the bias owner. Typically this is done by suspending the owner, verifying that the owner is not currently executing the locking path, and then marking the lock as no longer biased. Variations on biased locking [11, 19] have been broadly adopted and can be found in production Java virtual machines [8].

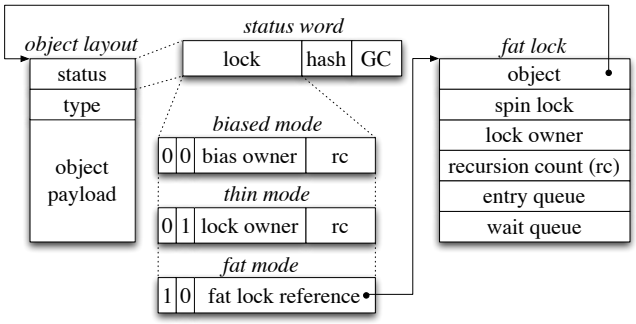

Figure 3: **Biased lock object layout in Jikes RVM.** We assume the same object layout as in Figure 1, but add one bit to the lock data to indicate whether or not the lock is biased. In biased mode, the *bias owner* bits identify the thread to which the lock is biased. Only the bias owner is allowed to manipulate the lock data portion of the status word. This allows the bias owner to use fast, non-atomic increment and decrement operations on the *recursion count* (rc) in order to acquire and release the lock. Attempts to acquire the lock by any other thread lead to bias revocation. When in thin mode, the lock behaves like a normal thin lock as in Figure 1.

Our Jikes RVM implementation of biased locking is shown in Figure 3. This is a relatively straight-forward extension of the thin lock protocol: we add an extra bit to the lock data (or rather, we steal it from the recursion count) to distinguish between the biased and thin states. The biased state tells the system that the *bias owner* has a monopoly on the lock — it is the only thread that can make changes to the lock data. This allows it to avoid using a CAS to acquire and release the lock. The fast path implementation for biased locking that we use in Jikes RVM is shown in Figure 4. Unlike thin locking, where the *lock owner* bits are reset to zero when the lock is released, the value of *bias owner* persists even after the lock is released. A biased lock is known to be not held if its recursion count is zero.

What happens in the slow paths is far more subtle, and is the main distinguishing characteristic between different biased locking protocols. In particular, the choice of which locks to bias is made dynamically by the biased locking protocol. This makes biased locking risky from a performance standpoint. Because bias revocations are much more expensive than other forms of lock contention, biasing many locks that are then acquired by another thread will lead to severe overheads due to revocation. On the other hand, if too few locks are biased, then the protocol becomes redundant — or worse, it leads to a performance loss because its only impact on the system is code bloat. Consequently the main challenges of designing a good biased locking protocol are (*i*) choosing the right heuristics for selecting which locks to bias and (*ii*) optimizing the bias revocation operation as a safeguard in case the heuristics fail.

2.2.1 Heuristics for Biasing

The original work on biased locking in Java is due to Kawachiya et al. [11] (KKO). They used a greedy approach to biasing: the first time that a new lock is acquired, it is biased to the thread

```
void inlineLock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits) {
      obj->status = oldStatus + (1<<RC_SHIFT);
      return;
   }
   if ((oldStatus & LOCK_MASK)
      == (1<<STATE_SHIFT)
      && CAS(&obj->status, oldStatus,
           oldStatus + curThread->lockBits
           + (1<<RC_SHIFT)))
      return;
   lockSlow(obj);
}

void inlineUnlock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits + (1<<RC_SHIFT)) {
      obj->status = oldStatus - (1<<RC_SHIFT);
      return;
   }
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits
        + ((1<<STATE_SHIFT)|(1<<RC_SHIFT))
      && CAS(&obj->status, oldStatus,
           oldStatus & UNLOCK_MASK))
      return;
   unlockSlow(obj);
}
```

Figure 4: **Inline lock and unlock paths for biased locks.** We show our implementation of Jikes RVM biased locking fast paths. As before (see Figure 2), the `inlineLock` and `inlineUnlock` procedures cover the common cases that are profitable to inline. To support biased locking, these procedures are modified to include one extra case (the non-atomic biased lock/unlock).

that acquired it. If the bias is ever revoked, the lock can never be rebiased. The *bias-on-first-acquire*, *never-rebias* heuristic works surprisingly well; both the original Kawachiya et al. [11] work and all subsequent literature on biased locking that we are aware of agree that the KKO algorithm is a net performance win for well-known benchmarks. However, a greedy approach requires that two properties hold to get good performance:

Shared locks are created rarely: We say that a lock is *shared* if it will be acquired by more than one thread during its lifetime. KKO will perform exactly one bias revocation per shared lock. KKO will exhibit good performance if the number of shared locks is small, or if they are all long-lived — since the cost of revocation is only paid once over the lifetime of each shared lock.

Bias revocation is not too inefficient: As will be discussed in the next section, KKO uses a data-access parallel bias revocation protocol — that is, a bias revocation only perturbs the execution of the bias owner and the bias revoker; all other threads are free to execute concurrently.

The frequency of shared lock creation is a property of the application. This implies that KKO will perform poorly on some pathological programs — up to 50× slower than thin locks as measured in our experiments.

The alternative is to bias only those locks that are unlikely to become shared. One instance of such an *adaptive* approach is the protocol proposed by Russell and Detlefs [19] (RD). Their protocol tracks the locking behavior of objects at class (i.e., type) granularity, and either biases or revokes locks based on dynamically gathered *per-class* (rather than per-object) statistics. RD will also rebias locks for which the bias was previously revoked if data gathered for the lock's class implies that it would be profitable. If the heuristics are tuned appropriately, this approach will lead to fewer bias revocations than the simpler KKO approach. But RD locks have problems of their own:

Empirically tuned heuristics: RD locks rely on complex trigger threshold heuristics for revocation and rebiasing. Russell and Detlefs [19] show that if these thresholds are empirically tuned for a particular benchmark suite, then RD will perform well, on that suite. The actual thresholds that Russell and Detlefs derived are not published, to our knowledge. Furthermore, performance on benchmarks for which the heuristics were not optimized is not shown. This is a major disadvantage versus the simpler KKO protocol: while the KKO protocol uses an intuitive and well-described heuristic, the RD protocol appears to require tuning revoke/rebias trigger thresholds to get good performance. There is a danger that the tuning will favor particular benchmarks over general applications.

Adaptation is coarse-grained: The RD adaptation heuristics conflate the behavior of all objects of a particular class. This can lead to problems if one part of the program uses instances of class `Foo` in a thread-local manner, while another part of the program shares it among many threads. Russell and Detlefs [19] argue that their approach might be extended to per-allocation-site adaptation, but concede that this would require significant changes to how a Java virtual machine structures objects.

More complex fast paths: To perform a biased lock acquisition in RD locks, the fast path must check if the object's class is *biasable*. This implies three additional loads, and two additional branches, that are not present in Figure 4 [19].

To summarize, KKO locks are simple, but have a clear pathology if the program frequently allocates shared locks — for each of these locks, the KKO protocol will have to perform a revocation. RD locks address these issues, but they do so at a coarse-grained level which may result in some thread-local locks not benefiting from biased locking.

2.2.2 Protocols for Bias Revocation

When a lock is biased, the bias owner is permitted to manipulate the lock data in the object header without atomic operations (see Figure 4). If another thread wishes to acquire the lock, the bias must first be revoked. A correct protocol for bias revocation must side-step the race condition that exists when a thread other than the bias owner attempts to manipulate the lock data.

The KKO protocol [11] uses thread suspension with machine-code-level roll-back to side-step the race. The revocation protocol begins by telling the operating system to suspend the bias owner. Once suspended, KKO inspects the thread's registers to determine if the thread is currently executing a biased lock operation. If not, the state of the lock can safely be flipped from biased to thin. On the other hand, if the bias owner is executing a biased lock operation, then KKO performs a manual roll-back of the biased lock operation by manipulating the thread's register set. Once this is complete, the lock's state can safely be flipped to thin. After the lock's state is changed to indicate that it is no longer biased, the bias owner is resumed. The KKO bias revocation protocol is scalable because it is data-access parallel: the only threads affected in the case of revocation are the bias owner and the thread performing the revocation. However, KKO revocation requires detailed tracking of the machine code generated for biased locking. If the biased locking code is inlined (which it should be for maximum perfor-

mance), then KKO locks must have details on every copy of the lock fast path in every method into which it was inlined.

The RD protocol [19] uses a simpler bias revocation protocol based on *safepoints*. Safepoints are inserted by the compiler (or interpreter) into the code stream at static locations where certain properties are known to hold. Typically this is used for garbage collection: at each safepoint the compiler generates accurate stack maps to allow for stack scans. Java virtual machines typically have built-in support for bringing all threads to a safepoint. This is used to aid various VM services including garbage collection and on-stack replacement. Russell and Detlefs [19] exploit the fact that the lock/unlock fast paths do not have safepoints inside them. Thus, triggering a system-wide safepoint automatically ensures that any biased lock can be revoked without risking a race condition. But what this approach gains in elegance, it loses in performance. While the KKO bias revocation is data-access parallel, the RD one is not: every revocation halts the progress of all threads. This is likely the main reason why the RD protocol is less greedy than KKO when biasing locks. The KKO protocol has a revocation algorithm that is fast enough to support high frequency revocations, while the RD protocol cannot support the same level of revocation frequency without causing performance pathologies.

One additional biased locking protocol is due to Onodera et al. [13] (OKK), who combine a spin-based lock with the KKO protocol to allow threads other than the bias owner to acquire the lock without revocation. However, this protocol increases the complexity of the biased lock acquisition fast path by requiring memory fences, and requires threads other than the bias owner sometimes to use a purely spin-based protocol for contention. We only consider biased locking protocols that maximize throughput by eliminating the need for either fences or atomic operations in the common case.

To summarize, KKO locks are simple but may experience pathological behavior if programs allocate shared locks at a high rate. The RD protocol addresses this problem but introduces problems of its own: namely, it uses complex heuristics for biasing which may not be suitable for all programs, and it employs a bias revocation protocol that is much more disruptive than the one used by KKO. We would like a protocol that combines the best of both worlds, by having KKO's simple heuristics, the elegance of RD's revocations, and the ability to better adapt to program behavior so as to prevent performance pathologies.

3. FABLE: Fine-grained Adaptive Biased Locks

We now present a new biased locking protocol called FABLE that includes improvements to both the biasing heuristics and the bias revocation protocol used in KKO locks:

Fine-grained Adaptation: FABLE adapts itself to each lock instance. No per-class or per-allocation-site statistics are gathered. Instead, FABLE *learns* the locking behavior of each lock, and chooses to enable either biased or thin mode locking depending on what it learns. This adaptation is done automatically, requires no user involvement, and requires at most one additional bit in the object header.

Fast Revocation: FABLE uses safepoints to revoke bias, but unlike RD locks, FABLE only safepoints one thread.

FABLE is designed to *reduce the likelihood of detrimental performance pathologies* while ensuring good throughput for common programs.

3.1 Fine-grained Adaptation with Random Counting

Locking protocols based on KKO use the simple *bias-on-first-acquire* heuristic for choosing which locks to bias. This leads to all locks being biased unless the bias is revoked. If the program allocates a lot of shared locks, then bias revocation dominates execution time. FABLE addresses this issue by using a *bias-after-multiple-acquires* heuristic: a lock must be acquired multiple times by the the same thread before it gets biased. We implement this heuristic with an additional lock state, called the learn mode. Locks in the learn mode have two fields: the *bias guess* and the *recursion count*. The bias guess identifies the thread that first acquired the lock. The intuition behind FABLE is that if the bias guess reacquires a lock N times non-recursively for some system-wide value of N, then bias the lock, otherwise switch to thin mode. We refer to N as the *learn limit*. The first lock acquisition does not count, since this acquisition only triggers the clear→learn transition. Thus, a total of $N+1$ non-recursive lock acquisitions must be performed by the same thread after the lock is allocated for FABLE to enable biased locking.

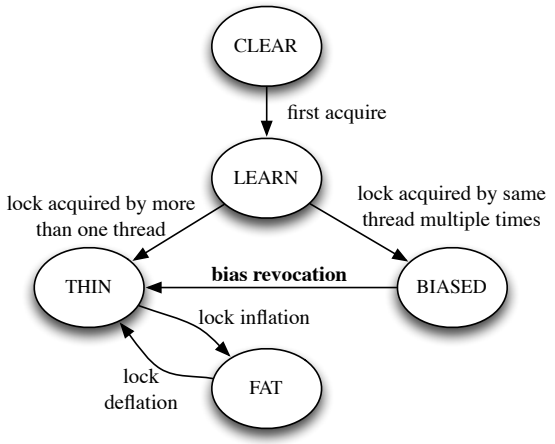

Figure 5: **State transitions of a FABLE lock.** Locks in FABLE start out in clear mode, and transition to learn mode upon first acquisition. While in learn mode, if the lock is acquired multiple times by the same thread, the lock gets biased to that thread. If more than one thread acquires the lock in learn mode, it immediately becomes a thin lock. After the lock is biased, if an acquisition attempt is made by some thread other than the bias owner, the bias gets revoked. Thin locks may be inflated if the status word does not provide sufficient storage to handle the lock state, which happens if threads get enqueued on the lock, or if the recursion count overflows. Of these state transitions, all but the bias revocation can be done at any time, by any thread, using a single CAS.

A naïve implementation of FABLE would require $\log(N)$ bits in the object header for counting lock acquisitions. We avoid this overhead by random counting [4]: every time that the bias guess performs a lock acquisition in learn mode, we bias the lock with probability $1/N$. One way to implement this is with a pseudo-random number generator, but we choose a simpler approach: each thread contains a `learnCount` field that we increment on each learn mode lock acquire. When it reaches the learn limit, we bias the lock. This results in $1/N$ biasing probability on average, but it is not a truly random operation: for extremely simple programs, it may suffer from "resonance" with the application program if the limit matches application phase transitions in the use of the lock.

The only tunable heuristic in FABLE is the learn limit. When first implementing FABLE, we arbitrarily chose `learnCount==5`. The only intuition is that larger values are likely to degrade performance since learn mode lock acquisition is slower than either thin or biased mode acquisition, while smaller values are less likely to detect when a lock is shared. Our experimental validation shows that this is a reasonable, if not optimal, choice. Except for a micro-

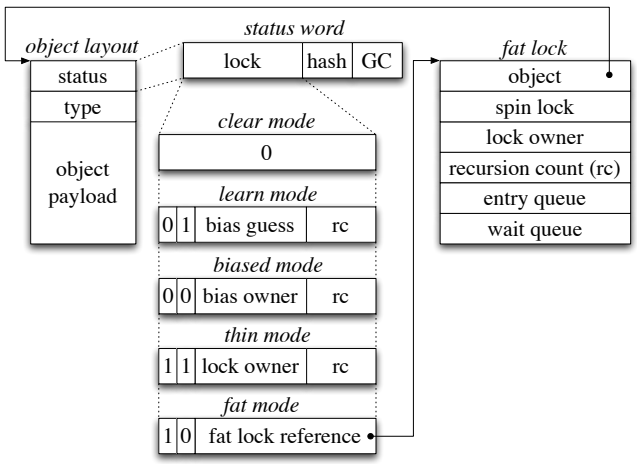

Figure 6: FABLE **object layout.** We assume the same object layout as in Figure 3, but exploit an unused value of the two-bit state field to introduce a new learn mode. The new learn mode is used for learning whether the lock will become shared. The first lock acquisition flips the lock from the initial clear mode to learn mode, with the *bias guess* set to the thread that first acquired it. If during some number of lock acquisitions, the bias guess is the only thread that acquires the lock, then the lock transitions to the bias mode; otherwise it transitions to the thin mode.

benchmark designed to measure FABLE's worst case by simply allocating objects in a loop and locking them k times for small values of k, the differences that arise from varying the learn limit are barely statistically significant. This leads us to believe that FABLE is robust, and changing its learn limit heuristic is unlikely to lead to surprising behavior.

Like KKO, FABLE employs the *never-rebias* rule. Once a lock becomes thin, it can never become biased. One could imagine introducing rebiasing heuristics (for example by reverting thin locks to learn mode after a time). The complete set of states in FABLE, as well as all possible state transitions, are shown in Figure 5. Figure 6 shows the object layout we use. Note that the number of bits used is the same as for our biased locks (Figure 3); this is because the 11 value for the lock state field was previously unused.

3.2 Optimized fast paths

Our goal with FABLE is to avoid reducing the throughput of those programs that already run fast with other locking protocols. However, some overhead is unavoidable since lock acquisition in learn mode requires random counting. We mitigate these overheads by ensuring that the biased and thin mode lock/unlock fast paths are no more complicated than the biased lock paths shown in Figure 4, and by ensuring that the learn mode unlock operation is identical to the thin mode unlock operation, allowing both to use the same fast path. The tricks used to make this happen are:

Biased mode corresponds to the state field having a 00 value: This ensures that detecting if a lock is biased does not require any additional arithmetic.

Combined learn/thin unlock fast path: We inline learn mode unlock to maximize performance, but we wanted to avoid bloating the inlined unlock code. This is accomplished via two tricks. First, the second-highest-order bit is 1 for both the learn mode and the thin mode, allowing the unlock fast path to detect if the lock is in either state without additional arithmetic. Second, thin lock release simply decrements the recursion count. This makes the same unlock code work for both learn and thin mode.

```
void inlineLock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits) {
      obj->status = oldStatus + (1<<RC_SHIFT);
      return;
   }
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits + (3<<STATE_SHIFT)
      && CAS(&obj->status, oldStatus,
           oldStatus + (1<<RC_SHIFT)))
      return;
   lockSlow(obj);
}

void inlineUnlock(Object obj) {
   oldStatus = obj->status;
   if ((oldStatus & LOCK_MASK)
      == curThread->lockBits + (1<<RC_SHIFT)) {
      obj->status = oldStatus - (1<<RC_SHIFT);
      return;
   }
   if ((oldStatus
       & ((1<<STATE_SHIFT)|OWNER_MASK|RC_MASK))
      == curThread->lockBits
        + ((1<<STATE_SHIFT)|(1<<RC_SHIFT))
      && CAS(&obj->status, oldStatus,
           oldStatus - (1<<RC_SHIFT)))
      return;
   unlockSlow(obj);
}
```

Figure 7: **Inline lock and unlock paths for** FABLE**.** FABLE's fast paths cover three cases: learn mode locks, biased locks, and thin locks. The lock acquisition fast paths are limited to the thin and biased modes, and are optimized to be as efficient as the fast path in Figure 4. The unlock fast path covers all three modes without introducing code bloat by using a polymorphic thin/learn unlock case: if a lock is held in either thin or learn mode, our unlock fast path will handle it using the same code.

The unlock fast path optimizations introduce a side-effect: unlocking a thin lock leads to the *lock owner* having a value corresponding to the *previous* lock owner. In FABLE a lock is known to be *thin but not held* if the recursion count is zero. To avoid bloating the thin lock acquisition fast path, the fast path now only covers the case where the last thread to hold the lock is the same as the thread acquiring it. This seems like it should cause slow-downs; however, we separately confirmed that changing a baseline thin locking implementation to exhibit this "sticky lock owner" property has no effect on performance. We suspect that if the thread acquiring the lock is different than the last thread to hold the lock then performance is dominated by cache effects rather than by the slight overhead of a procedure call.

The complete fast path code for FABLE is shown in Figure 7. Our results show that for common programs, FABLE is rarely slower than biased locks while sometimes being a lot faster. The biggest observed speed-up exceeds 50×. On programs for which we had previously optimized our KKO-style biased locking implementation, FABLE exhibits nearly identical performance.

3.3 Bias Revocation by Thread Safepointing

Like KKO locks, FABLE uses a data-access parallel bias revocation protocol: the only threads affected are the bias owner and the revoking thread. Like RD locks, FABLE uses safepoints for revocation,

which leads to a simple implementation. This combination is possible because as part of our work towards the Jikes RVM 3.1 release, we added the ability to safepoint threads individually. This capability is engineered to allow multiple disjoint pairs of threads to safepoint each other in parallel, while also ensuring deadlock freedom when two or more threads attempt to simultaneously safepoint each other. We call this safepointing protocol the *pair handshake*. We support two forms of pair handshake:

Synchronous Pair Handshake: A synchronous pair handshake is exposed as two operations, `beginPairHandshake` and its converse, `endPairHandshake`. The first operation stops the target thread at a safepoint and returns. This requires waiting for the target thread to reach a safepoint and informing it to suspend itself until the `endPairHandshake` call. When using synchronous pair handshakes, FABLE performs the bias revocation using a CAS after calling `beginPairHandshake` but before calling `endPairHandshake`.

Asynchronous Pair Handshake: An asynchronous pair handshake, or `asyncPairHandshake`, supplies the target thread with a callback that is invoked at the next safepoint. The target thread never stops in this scheme. If the target thread is in native code, the callback is immediately executed; otherwise `asyncPairHandshake` waits until the target thread reaches a safepoint and executes the callback. FABLE exploits this by passing a closure that performs the revocation.

The asynchronous approach has the benefit of never suspending the target thread, which sometimes results in speed-ups, especially on systems that have many cores.

4. Qualitative Comparison of Locking Protocols

We have presented four locking protocols so far: thin locks, KKO biased locks [11], RD biased locks [19], and FABLE. We have our own implementation of KKO locks that uses both the synchronous and asynchronous styles of FABLE's pair handshakes; we consider this approach to be almost equivalent to KKO and we will use it as a performance baseline for validation. We refer to these as Bias-Sync and Bias-Async for convenience. This section compares the protocols on a qualitative basis. The variants are as follows:

Thin Locks: We expect thin locks to have optimal performance when the majority of locks are shared, if the platform supports a very fast CAS operation, or when locking is used sparsely by the program. Our previous experiments on varying the thin lock/fat lock implementation show that many possible alternative implementations of thin locks exist, and that their performance is not very different.

KKO: We expect KKO to perform well on common benchmark programs, but exhibit pathologies on some corner cases. Previous literature on KKO locks [11] shows this to be the case. It is only in the case of pathological programs that KKO will perform poorly. Two kinds of pathologies exist, and we believe that though rare, these pathologies could easily happen in production code. One example is producer-consumer. Consider a program that allocates an object, locks it a few times, and enqueues it. This may happen if the object comes from a third-party library that is using locking defensively. The object is then consumed by a separate thread, and again locked, due to defensive locking. Every time that an object moves from one thread to another in such a program, KKO locks will suffer revocation. RD and FABLE will be able to avoid these revocations in most cases. The other pathology is one we call "cloud of objects": consider that a multi-threaded program is operating over a large shared data structure full of many objects. Accesses to the objects are protected with locks. KKO locks will bias each object on the first access, leading to subsequent revocations. If the number of objects is large enough, this may cause severe slow-downs. RD and FABLE will be able to avoid these revocations in most cases.

RD: RD locks are the most conservative in choosing which objects to bias, accomplished by heuristics that track per-class locking behavior. On the other hand, RD locks can easily be *too* conservative: if any object of any class becomes shared and experiences heavy contention, other objects of that class may become ineligible for biased locking even if those objects are thread-local. FABLE addresses this problem by using per-object heuristics. KKO addresses this problem by being greedy: every object is biased until the bias is revoked.

Bias-Sync: This is almost exactly like the KKO protocol, but uses a simpler revocation strategy based on per-thread safepoints. Kawachiya et al. [11] state that their particular implementation of the KKO revocation protocol is sometimes slow on their platform of choice (Windows) due to the high costs of manipulating a suspended thread's register file. One of our contributions is that we avoid this overhead using thread safepoints.

Bias-Async: Asynchronous pair handshakes avoid having to stop the bias owner, which should make them perform faster in some cases.

FABLE**:** The main advantage of FABLE is that it has RD's resilience against pathological programs without resorting to complex heuristics. FABLE has only one heuristic, the *learn limit*. We show that changing the value of this parameter does not change performance much. We expect FABLE to be slower than simple biased locking on programs that frequently lock short-lived objects. On typical programs we expect FABLE to perform like regular biased locking.

FABLE and RD locks both employ more sophisticated heuristics than KKO so as to reduce the number of bias revocations. However, neither style is strictly superior to the other. FABLE will systematically fall into the producer-consumer pathology if the producer always non-recursively relocks an object more than N times before enqueueing it, where N is the learn limit. RD is better equipped to handle this case because it marks entire classes as bulk unbiasable. On the other hand, FABLE will beat RD if the same class is used for a producer-consumer pattern in one part of the program while being used in a thread-local fashion in another part of the program. RD will have trouble separating these uses because it tracks lock behavior on a per-class (rather than per-object) basis.

FABLE has two other advantages over RD locks: FABLE uses a fast data-access parallel bias revocation (which is by itself enough to make biased locking a net win on most programs) and FABLE's lock/unlock fast paths require fewer instructions than RD's.

4.1 Fat locking and lock inflation strategies

Production Java virtual machines are engineered to minimize the number of header words in order to reduce garbage collection pressure and memory footprint. It is common to reduce the header to two words (as is the case in Oracle HotSpot [7] and Jikes RVM) or even one word (as is the case in Azul VM [6]). The full fat lock structure will typically require at least three additional words (for storing the lock status and the queues associated with lock acquisition and wait/notify). If object header overhead was not an issue, there would be no reason for thin locks: one could simply inline the fat lock allocation fast paths, and even make them use biased locking. But the space overheads of fat locks mean that all production Java virtual machines that we are aware of make some effort to avoid allocating fat locks. Jikes RVM allocates fat locks only when

there is contention, and deallocates them as soon as contention subsides. Some VMs use a more relaxed strategy and allocate a fat lock when an object is first locked, rather than only when an object is contended. The stack-locking protocol used in Oracle HotSpot [7][2] is a particularly interesting instance of this strategy. When an object is locked, a word in the object header is replaced with a pointer to a stack-allocated structure; the previous value of that word is displaced into this structure. This works well since Java locks are lexically scoped, and results in only 1 bit of object header overhead for locking. Fiji VM [17] allocates a fat lock in the heap on the first lock acquisition and deallocates it only when the object becomes unreachable. The lock pointer replaces the class pointer, and the lock data references the class. Double-indirection is used to access an object's class, and a Brooks-style [5] forwarding pointer in the class is used to make this double indirection unconditional and relatively inexpensive. This results in zero header overhead for locking, for objects that have never been locked. With this strategy Fiji VM achieves performance that is close to that of HotSpot and Jikes RVM, likely because the number of objects in the heap that ever get locked is small (typically 10%, as measured by Bacon et al. [2]).[3]

Based on cross-VM comparisons, it appears that there is no correlation between how aggressively a Java virtual machine avoids allocating fat locks and how much throughput the VM achieves. The choice of fat lock allocation strategy is often dictated not by throughput but by constraints from other parts of the system. For example, Jikes RVM only has 22 bits in the object header for lock data. Header word displacement, used in HotSpot [7] and Exact VM [1], or Brooks forwarding as in Fiji VM, is not used in Jikes RVM, as either technique would slow down access to the hash code, garbage collection state, and the class pointer. It also leads to a simpler overall design, which is essential for making Jikes RVM an easy-to-use platform for experimenting with new VM and garbage collection techniques. This means that at most 2^{22} (about 4 million) objects with fat locks are allowed to exist at any time in the heap. Jikes RVM side-steps this limit by allocating fat locks only on contention or recursion count overflow, and deallocating them as soon as they are no longer contended or when their recursion count is reduced. The number of locks being contended is bounded by the number of threads, and the recursion count is bounded by the stack height; as a result we have never seen a program that hits the 4 million fat lock limit. The fact that Jikes RVM's thin locks are necessitated by other requirements in the system means that unfortunately, this work cannot show a direct comparison between FABLE and pure fat locking. However, past results such as those of Bacon et al. [2] and Onodera and Kawachiya [12] show quite conclusively that fat locking alone is not an efficient lock implementation strategy.

4.2 Related work

Minimizing the cost of bias revocation is an active area of research. An alternative to revoking bias is to use a secondary locking protocol to serialize contention by threads other than the bias owner. [13] Such schemes suffer from increased complexity in the biased lock acquisition path. FABLE attempts to make biased lock acquisition fast paths as fast as possible. Russell and Detlefs [19] use a different strategy, based on tracking the object classes for which bias revocation would be frequent, and avoiding biasing for those classes. This approach is neither fundamentally better, nor fundamentally worse, than FABLE, in the sense that each technique may experience pathologies that the other handles gracefully. We believe that it would be sensible to explore a combination of FABLE and RD [19]. However, FABLE has the advantage of being simple and easy to implement, and has a simpler fast path for biased locks. We are also aware of a similar protocol to FABLE in unpublished work by people at Azul. We are not aware of any performance results that show that the their protocol is as fast as biased locking, or even simple thin locking. FABLE provides more than just a promise of good performance; as the next section shows, FABLE soundly outperforms thin locking on platforms with expensive atomic hardware primitives, is almost as fast as KKO-style [11] biased locking on those benchmarks that KKO was tuned for, and achieves enormous speed-ups on corner cases that FABLE, RD [19], and OKK [13] are attempting to address.

[2] The best description of this protocol that we are aware of is in Agesen et al. [1].

[3] The most up-to-date metrics of Fiji VM performance, with direct comparisons to HotSpot and other VMs, are found in Pizlo et al. [18].

Machine	Description
i7-1x4x2	Intel Core i7 975 3.33GHz, 4 cores, hyperthreading (total 8 logical CPUs), 12GB RAM, Fedora 11, Linux 2.6.29.4-167.fc11.x86_64
Core2-2x4	Intel Xeon E5410 2.33GHz, 2 processors, 4 cores/processor (total 8 logical CPUs), 8GB RAM, Fedora 10, Linux 2.6.27.21-170.2.56.fc10.x86_64
Core2-4x4	Intel Xeon E7310 1.6GHz, 4 processors, 4 cores/processor (total 16 logical CPUs), 16GB RAM, Ubuntu 8.10, Linux 2.6.27-11-generic
i7-4x8x2	Intel Xeon X7560 2.27GHz, 4 processors, 8 cores/processor, hyperthreading (total 64 logical CPUs), 32GB RAM, Ubuntu 10.10, Linux 2.6.35-22-generic

Table 2: **Machines used in our performance evaluation.** To mitigate systematic effects due to experimental setup, we use four different hardware configurations, with different operating system versions and different Intel architectures. Two machines are based on the Core 2 architecture, and two are based on Core i7, which has very fast CAS.

5. Experimental Validation

We claim that FABLE (*i*) results in speed-ups even with a simple biasing heuristic, (*ii*) handles those pathologies that simple biased locking cannot handle, (*iii*) is not much slower than simple biased locking on those programs for which biased locking was optimized, and (*iv*) has bounded overheads even for those programs that are adversarially designed to induce FABLE's worst-case. To validate these claims we have assembled an extensive set of performance comparisons, using four machines that vary in both OS and hardware (see Table 2). Our findings also show that there is little benefit to using biased locking on Intel Core i7 based architectures.

5.1 Handling Pathologies

Our goal in designing FABLE was to handle pathological cases better than KKO locks. To evaluate this claim we wrote two benchmarks that are designed to create disastrous performance for biased locking: Cloud and ProdCons. Cloud begins by allocating an array of a million objects and starting ten threads that randomly lock these objects. Successful lock acquisitions constitute progress. We run the benchmark for 25 three-second runs per configuration. We report the total number of successful lock acquisitions by all threads. ProdCons is a simple two-thread producer-consumer benchmark where the producer allocates an object, locks it once, and enqueues it. The consumer dequeues it, locks it once, and throws it away. We run the benchmark for five million object productions and measure the time taken.

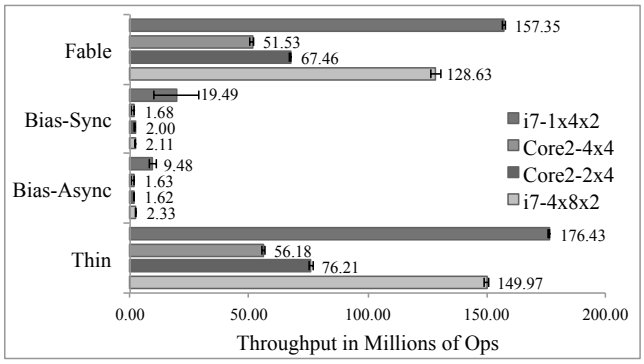

Figure 8: **Handling pathologies: Cloud benchmark.** This shows the throughput (*higher is better*) of the Cloud benchmark, which has 10 threads randomly locking 1,000,000 objects. FABLE is between 8× and 55× faster than simple biased locking; it appears that the more processors a machine has, the bigger the speed-ups due to FABLE. FABLE is only slightly slower than thin locks on this benchmark.

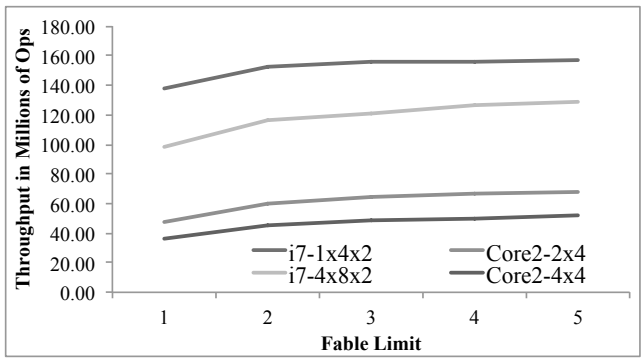

Figure 9: **Cloud benchmark for different** FABLE **limits.** This shows how FABLE's ability to handle pathologies changes with different values of the *learn limit*.

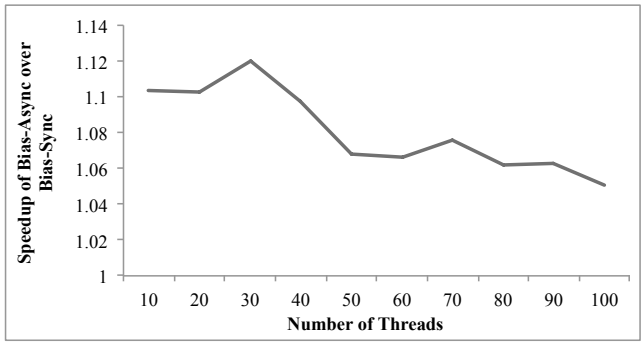

Figure 10: **Speed-up of Bias-Async on i7-4x8x2 for the Cloud benchmark for different thread counts.** The asynchronous pair handshake is never faster than the synchronous pair handshake, except on the 32-way (64-way with hyperthreading) machine. To investigate this further we varied the number of threads between 10 and 100, and found that the speed-up persists, peaking at 30, which corresponds to almost 1-to-1 thread-to-core ratio.

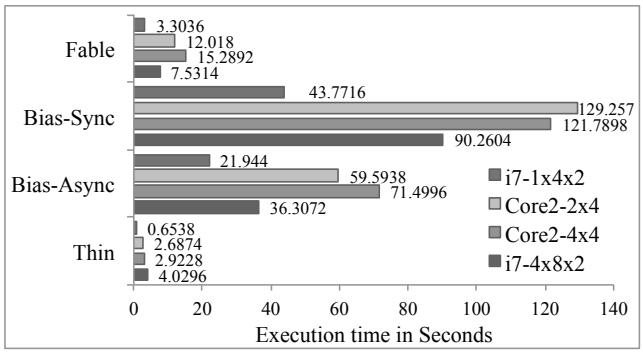

Figure 11: **Handling pathologies: ProdCons benchmark.** This shows the execution time (*lower is better*) of the ProdCons benchmark, which has two threads communicating over a queue, with each thread acquiring (and releasing) a lock on the objects being passed. This causes a storm of bias revocations in Bias-Sync and Bias-Async, with Bias-Async performing much better. FABLE performs about 6× better than Bias-Async, but is still about 3× slower for this pathological scenario than thin locks.

Figure 8 summarizes the results of the Cloud benchmark. FABLE produces speed-ups up to 55× on one platform, and manages to perform within approximately 10% of thin locks. This shows that FABLE is resilient against one of the kinds of pathologies for biased locks. We further investigate this in Figure 9, which shows how the performance varies as the *learn limit* heuristic is changed. Higher values increase resilience, but not by much. This benchmark shows that Bias-Sync and Bias-Async perform quite differently depending on the platform. The only platform for which Bias-Async helps on the Cloud benchmark is our large 32-way machine with hyperthreading. Figure 10 investigates this further by showing the Bias-Async speed-up on this machine for different thread counts.

Figure 11 shows a summary of the results of the ProdCons benchmark. FABLE handles this pathology much better than either Bias-Async or Bias-Sync. Here, Bias-Async is better than Bias-Sync. FABLE is roughly 3× slower than thin locking in some cases — this is likely because lock acquisitions that would have taken the fast path in thin locks are now taking the slow path. This shows that FABLE isn't a silver bullet — but it also shows that our simple heuristics are powerful enough to dramatically improve performance over the simple KKO-style greedy biasing approach [11] without having to resort to the more sophisticated heuristics of Russell and Detlefs [19].

5.2 Throughput

FABLE is designed to make programs run faster than with thin locking. Unlike the RD protocol [19], this is true even if a simple biasing heuristic is used. When using the more robust per-object adaptation strategy, we expect a slight slow-down on common benchmarks. This section aims to show that the slow-down is so small that it is outweighed by the increased robustness of per-object adaptation. To validate this claim we assembled 16 standard benchmarks from Dacapo 2006 [3] and SPECjvm98. KKO-based biased locking will perform well on these benchmarks since they have very few shared locks. The goal is to show that even for programs where FABLE is not needed, it still performs well enough that it is reasonable to make it the default locking protocol for a production JVM. We use 25 samples for each benchmark/platform/configuration. Each benchmark is run for five "plans"; each plan contains six warm-up iterations followed by five sampling iterations. The plans are executed at random to minimize potential systematic experimental bias due to execution order. We tested five configurations: Thin, Bias-

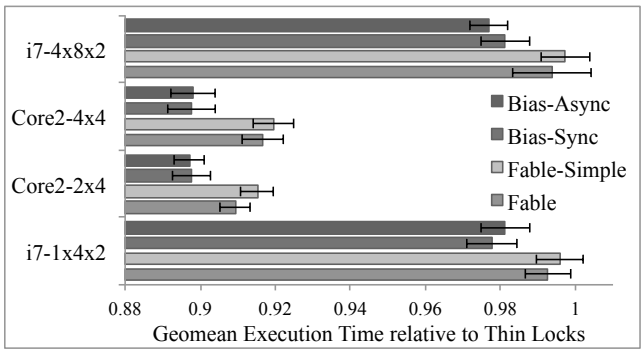

Figure 12: **Summary of throughput benchmarks.** This shows the execution time (*lower is better*) of various locking protocols across 16 benchmarks (DaCapo and SPECjvm98), relative to thin locking. Biased locking is much faster than thin locking on Core 2 based systems, and barely faster on Core i7 based systems. FABLE appears to be slower than simple biased locking, but the difference exceeds the 95% confidence intervals on only two platforms and is typically just 1%.

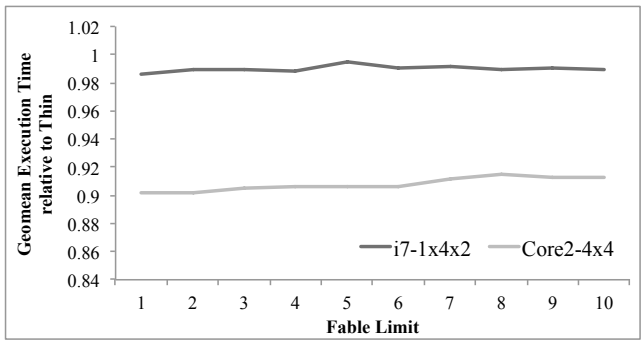

Figure 13: **Throughput as a function of** FABLE **limit.** We reran all of the benchmarks for different values of the *learn limit* to see how throughput varies. FABLE's learn limit heuristic is very robust overall: changing its value does not lead to variations in overall system throughput.

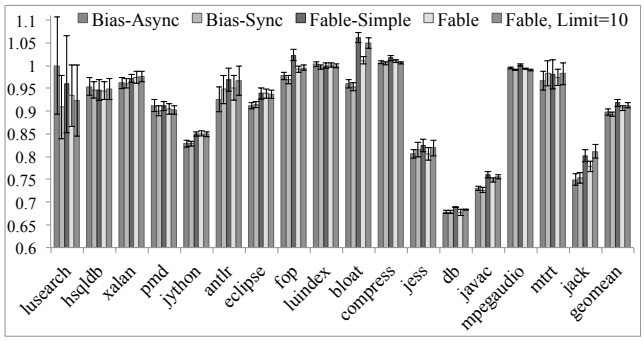

Figure 14: **Details of throughput benchmarks on Core2-4x4.** This shows the execution time scaled against thin locks (*lower is better*) for all benchmarks. Out of all benchmarks, only bloat, javac, and jack experience statistically significant slow-downs (versus simple biased locks) from using FABLE at any learn limit.

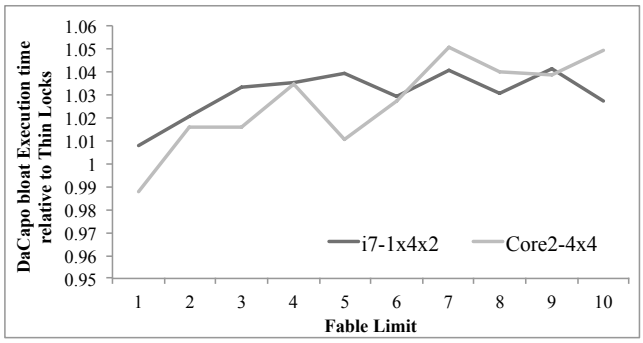

Figure 15: **Throughput on DaCapo bloat as a function of** FABLE **limit.** Only the DaCapo bloat benchmarks slows down significantly due to FABLE. The margin of error in this chart is roughly 1%; thus much of the variation is likely just noise. But it is possible that it is due to resonance (see Section 5.3).

Sync, Bias-Async, FABLE, and FABLE without the optimizations in Section 3.2 (FABLE-Simple).

Figure 12 shows the geometric mean performance relative to thin locking for four configurations and four platforms. The 95% confidence intervals were computed by first obtaining 25 individual geometric means for each of the 25 samples (i.e., for all $1 \leq i \leq 25$ we compute the geometric mean for sample i over all benchmarks, and then we find the mean and standard deviation of the 25 means). Using this technique we see that FABLE is only marginally slower than simple biased locking. It is also clear from these results that for Core i7 based systems, biased locking is typically unnecessary due to its very fast CAS implementation. It is not clear to us if the performance of Core i7 represents a trend that will persist as future multi-core architectures are introduced. We have separately investigated CAS performance on POWER and AMD-based x86 systems and found it to be expensive enough to warrant biased locking.

Figure 13 shows FABLE's throughput as a function of the *learn limit*. This confirms that the learn limit is a robust heuristic — changing it slightly does not perturb execution by much. Figure 14 shows per-benchmark details for Core2-4x4. We chose to highlight per-benchmark performance on the Core2-4x4 because it exhibited the widest performance variance across benchmarks and configurations. There is one benchmark for which FABLE in general, and the learn limit in particular, has a very noticeable effect: DaCapo bloat. DaCapo bloat is a single-threaded program that appears to allocate many objects that it locks only a few times, making FABLE a sub-optimal choice of locking protocol. Figure 15 shows the effect of using FABLE on bloat.

We have also run SPECjbb2000 and found that its throughput increases slightly due to biased locking, but the increase is very small; most importantly however, we found that FABLE's data-access parallel bias revocation is sufficient to guard against pathologies in SPECjbb2000. This is unlike the RD protocol, for which SPECjbb2000 performs poorly unless biasing heuristics are carefully tuned [19].

5.3 Bounding Slow-downs

FABLE has a pathology of its own: if a program rapidly allocates short-lived objects and locks them only a few times, the objects will never make it out of learn mode. Learn mode lock acquisition is slower than either thin or biased acquisition. To measure this effect, we created an AllocLock benchmark that loops two hundred million times, each time allocating an object, and locking it k times with an empty synchronized statement. Jikes RVM does not

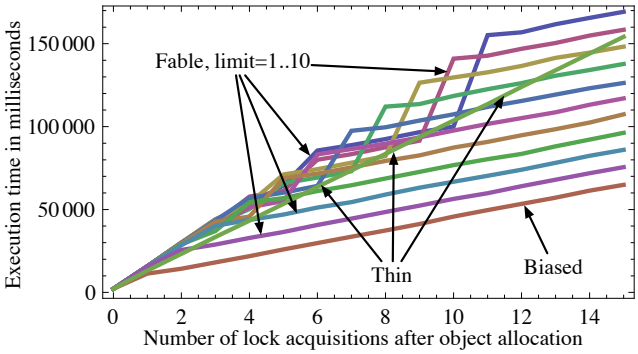

Figure 16: **Slow-downs due to FABLE: AllocLock on Core2-4x4.** This shows how fast a program can allocate and acquire a lock k times (the X-axis) for various locking protocols. This includes simple biased locks (Bias-Sync and Bias-Async perform identically), thin locks, and FABLE for 10 learn limits.

optimize away empty synchronized statements. We vary k in the range $1 \leq k \leq 15$ and vary the learn limit in the range $1 \leq N \leq 10$. We compare results against thin locks and simple biased locks.

Figure 16 shows the results of the AllocLock benchmark. This demonstrates that for short-lived objects that get locked very rapidly (i.e., by a loop that does absolutely nothing other than repeatedly release and reacquire the lock), FABLE is up to 50% slower than thin locks. This figure also shows an interesting resonance effect due to the fact that FABLE's random counting is not completely random. Recall that with a learn limit of N, it takes on average $N+1$ acquisitions to cause an object to become biased, since the first acquisition just puts the object in learn mode. This means that if AllocLock runs with $k = N$ then on one iteration the per-thread `learnCount` will reach $N-1$ and the object will not be biased. On the next iteration the object will be biased on the *second* acquisition, at which point the `learnCount` will reset to 0, and the process repeats. Thus every other iteration of AllocLock will run very quickly (because the object becomes biased after the 2nd lock acquisition), while the other iterations will run at "expected" speed (i.e., the object doesn't get biased).

6. Conclusion

We have presented a new biased locking protocol, called FABLE, which allows for fine-grained per-object adaptation to program behavior. FABLE is faster than simple biased locking on pathological programs, and its tracking of per-object locking behavior incurs only minimal overheads. Our evaluation of locking protocols is to our knowledge the most thorough to date — in addition to running a broad spectrum of industry-standard benchmarks on four different platforms, we have also designed three corner-case benchmarks to show the behavior of different protocols in detail. Our results show that biased locking continues to be a promising approach for hardware that has a slow CAS implementation, and that FABLE is a reasonably simple way of reducing the likelihood that biased locking results in poor performance on some programs.

References

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In OOPSLA 1999 [14], pages 207–222. doi: 10.1145/320384.320402.

[2] D. F. Bacon, R. B. Konuru, C. Murthy, and M. J. Serrano. Thin locks: Featherweight synchronization for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–268, Montréal, Canada, June 1998. doi: 10.1145/989393.989452.

[3] S. M. Blackburn, R. Garner, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In OOPSLA 2006 [15], pages 169–190. doi: 10.1145/1167473.1167488.

[4] M. D. Bond and K. S. McKinley. Bell: bit-encoding online memory leak detection. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61–72, San Jose, CA, Oct. 2006. doi: 10.1145/1168857.1168866.

[5] R. A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *ACM Conference on LISP and Functional Programming*, pages 256–262, Austin, Texas, Aug. 1984. doi: 10.1145/800055.

[6] C. Click, G. Tene, and M. Wolf. The Pauseless GC algorithm. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 46–56, Chicago, IL, June 2005. doi: 10.1145/1064979.1064988.

[7] O. Corporation. Java SE HotSpot at a glance. URL http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[8] D. Dice. Biased locking in HotSpot. URL http://blogs.sun.com/dave/entry/biased_locking_in_hotspot.

[9] D. Dice. Implementing fast Java monitors with relaxed-locks. In *Java Virtual Machine Research and Technology Symposium (JVM)*, pages 79–90, Monterey, CA, Apr. 2001. URL http://www.usenix.org/publications/library/proceedings/jvm01/dice.html.

[10] H. Franke and R. Russell. Fuss, futexes and furwocks: Fast user-level locking in Linux. In *Ottawa Linux Symposium*, pages 479–495, Ottawa, Canada, June 2002. URL http://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf.

[11] K. Kawachiya, A. Koseki, and T. Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–141, Seattle, Washington, Nov. 2002. doi: 10.1145/582419.582433.

[12] T. Onodera and K. Kawachiya. A study of locking objects with bimodal fields. In OOPSLA 1999 [14], pages 223–237. doi: 10.1145/320384.320405.

[13] T. Onodera, K. Kawachiya, and A. Koseki. Lock reservation for Java reconsidered. In M. Odersky, editor, *European Conference on Object Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 559–583, Oslo, Norway, June 2004. Springer. doi: 10.1007/b98195.

[14] OOPSLA 1999. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, CO, Oct. 1999. doi: 10.1145/320384.

[15] OOPSLA 2006. *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, OR, Oct. 2006. doi: 10.1145/1167473.

[16] F. Pizlo, D. Frampton, and the Jikes RVM Team. Configurable lock framework. URL http://jikesrvm.svn.sourceforge.net/viewvc/jikesrvm/rvmroot/branches/RVM-791/working-15440/.

[17] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In M. T. Higuera-Toledano and M. Schoeberl, editors, *International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, pages 110–119, Madrid, Spain, Sept. 2009. doi: 10.1145/1620405.1620421.

[18] F. Pizlo, E. Blanton, A. Hosking, P. Maj, J. Vitek, and L. Ziarek. Schism: Fragmentation-tolerant real-time garbage collection. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 146–159, Toronto, Canada, June 2010. doi: 10.1145/1806596.1806615.

[19] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In OOPSLA 2006 [15], pages 263–272. doi: 10.1145/1167473.1167496.