Simple and Fast Biased Locks

Abstract

Locks are typically used to ensure exclusive access to shared memory locations. Unfortunately, lock operations are expensive, so much work has been done on optimizing their performance for common access patterns. One such pattern found for e.g., is in networking applications, where there is a single thread dominating lock accesses. An important trivial case of this occurs when a single-threaded program calls a thread-safe library that uses locks.

An effective way to optimize this dominant-thread pattern is to "bias" the lock implementation so that accesses by the dominant thread have negligible overhead. We take this approach in this work: we simplify and generalize existing techniques for biased locks, producing a large design space with many trade-offs. For example, if we assume the dominant process acquires the lock infinitely often (a reasonable assumption for packet processing), it is possible to make the dominant process perform a lock operation without expensive fence or compare-and-swap instructions. This gives a very low overhead solution; we confirm its efficacy by experiments. We show how these constructions can be extended for lock reservation, re-reservation, and to reader-writer situations.

1. Introduction

Programmers typically use locks to control access to shared memory. While using locks correctly is often the biggest challenge, programmers are also concerned with their efficiency. We are too: this work improves the performance of locking mechanisms by using knowledge of their access patterns to speed the common case.

Figure 1 shows the standard way of implementing a spinlock using an atomic compare-and-swap operation (most modern processors have either this instruction or an equally powerful atomic test-and-set). To acquire the lock, a thread first waits ("spins") until the lock variable *lck* is 0 (indicating no other thread holds the lock), then attempts to change the lock value from 0 to 1. Since other threads may also be attempting to acquire the lock at the same time, the change is done atomically to guarantee only one thread changes the value. Although other threads' *while* loops would see the lock variable become 0, their compare-and-swap would fail because the winning thread would have changed the lock to 1.

Executing an atomic instruction is costly because most processors implement it by locking the memory bus to prevent other processors from executing memory operations. For example, we found the compare-and-swap instruction on an unloaded 1.66 GHz Intel Core Duo took seven times longer than "*counter++*," a comparable non-atomic read-modify-write operation. The cost when there is contention among multiple processors can be substantially higher, especially if a

```
void lock(int *lck) {
 bool success;
 do {
     while (*lck != 0) {} /* wait */
     success = compare_and_swap(lck, 0, 1);
   } while (!success);
}

void unlock(int *lck) { *lck = 0; }

atomic /* function is one atomic machine instruction */
bool compare_and_swap(int *lck, int old, int new) {
   if (*lck == old) {
     *lck = new; return 1;
   } else
     return 0;
}
```

Figure 1. A spin lock implemented with atomic compare-and-swap

cache miss is involved. This overhead can be prohibitive for a performance-critical application such as packet processing, which may have to keep up with line rates of over 1 Gbps and thus has a very limited cycle budget for actual processing. Reducing locking overhead, therefore, can be very useful.

Bacon et al.'s thin locks for Java [3] are an influential example of lock optimization. Their technique was motivated by the observation that sequential Java programs often needlessly use locks indirectly by calling thread-safe libraries. To reduce this overhead, thin locks overlay a compare-and-swap-based lock on top of Java's more costly monitor mechanism. Thus a single-threaded program avoids all monitor accesses yet would operate correctly (i.e., use monitors) if additional threads were introduced. Thin locks considerably reduce overhead but still require one atomic operation per lock acquisition.

A refinement of this technique [2, 9, 12] further improves performance by allowing a single thread to reserve a lock. Acquisitions of the lock by the reserving thread do not require an atomic operation but do require still costly memory fences to ensure another thread did not acquire the lock.

Lamport [10] also optimizes for the low contention access pattern by avoiding atomic operations. Unfortunately, he uses a bakery-style algorithm to resolve contention, which has been found to be less efficient than algorithms that do use atomic operations, such as the MCS lock [11].

Lopsided lock access patterns in network packet processing applications motivated our work. In a typical architecture, a packet is read off a network card by a dedicated core and then dispatched to one of several processing cores. In the commercial network traffic analyzer with which we are familiar, the packets are partitioned among cores by source address; i.e., all packets with the same source address are sent to the same core. Each processing core maintains data structures for its group of source addresses, and nearly all accesses to this data is from the core that owns them. Occasionally, however, a core might update information held by a different core; thus, it is necessary to maintain atomicity of updates using locks. Such an arrangement of data and processing results in a highly biased access pattern for a data item: the owner is responsible for a large (90% or more) fraction of the accesses to its data, the rest originate from other cores.

This work looks at the question of optimizing lock performance under such lopsided access patterns. It makes four contributions. First, we provide a generic method for building biased locks. In a nutshell, we implement biased locks with a two-process mutual exclusion algorithm between the dominant thread and a single representative of all of the other threads, chosen with a generic *N*-process mutual exclusion algorithm. This construction simplifies and generalizes the algorithm of Kawachiya et al. [12], which is a specific combination of this type that intertwines a Dekker-lock for two threads and a CAS-based lock for an *N*-thread mutex. Our experiments show that different choices for the *N*-process mutex algorithm can improve overall performance.

Our second contribution is a simple scheme for changing the primary owner of a lock (“re-reservation”). The scheme given by Kawachiya et al. [9] is heavyweight: it requires suspending the thread owning the lock and often modifying its program counter to a retry point; in their later work, they abandoned it for this reason [12]. By contrast, we show a simple way to change a lock’s owner without suspending the existing owner.

In our third contribution, we establish conditions under which atomic and memory fence operations in a dominant thread can be *dispensed with entirely*. Most multiprocessor memory systems do not provide sequential consistency across threads: a sequence of writes by one thread may appear to occur in a different order to a different thread. Few synchronization algorithms can cope with such an unruly communication mechanism, so multiprocessors typically provide costly but effective “fence” instructions that force all outstanding writes to complete. Experiments on the Intel Core Duo chip show that their “mfence” instructions require about two to three clock cycles. We show memory fences are essential for the biased lock construction described above, assuming the weaker memory ordering imposed by store-buffer forwarding, which is a feature of most modern processors. We prove that for a processor with store-forwarding, any mutual exclusion algorithm with a “symmetric choice” property requires memory fences. The symmetric choice property is that there is a protocol state where either of two contending threads may acquire the lock. Since standard algorithms such as those by Dekker [7], Peterson [13], and Lamport [10] have the symmetric choice property, they all require memory fences to be correct. Our proposed solution, therefore, is asymmetric by nature: it requires the dominant thread to “grant” access to the lock after receiving a request from a non-dominant thread. The protocol as a whole is free from starvation provided the dominant thread checks for such requests infinitely often.

Finally, we introduce biased read-write locks. A read-write lock allows multiple readers to read at the same time, but only one writer to access the critical section at any time. We show, along with experiments that the general construction of bias in normal locks can be extended to provide biased read-write locks.

In the next section, we describe our generic owner-based locking scheme, which assumes a fixed owner. We then discuss the algorithm for switching ownership (Section 3). The formalization of memory fence operations, the symmetric choice property, and the subsequent proofs are discussed in Section 4. We define asymmetric locks in Section 5. We outline our verification procedure for these techniques in Section 7 and discuss experimental results in Section 8.

2. Flexible, Fixed-Owner Biased Locks

In this section, we define a flexible biased locking scheme that assumes a lock is owned by a fixed, pre-specified thread. The scheme reduces the cost of access for the owning thread. In particular, the scheme does not incur the cost of a compare-and-swap operation, but it does require memory fences for correctness. From this point on, we focus on the x86 architecture; the kind of fences and their placement may differ for other architectures.

At its center, our scheme employs different locking protocols for the owner and the non-owners. For the owner, we use any two-process mutual exclusion protocol with operations *lock2* and *unlock2*; for the other threads, we use a generic *N*-process mutual exclusion protocol, with operations *lockN* and *unlockN*. This makes use of complementary characteristics: protocols that rely only on atomicity of read and write operations, such as Peterson’s algorithm [13], are efficient for two processes but not necessarily for larger numbers of threads; protocols based on atomic primitives, such as the MCS lock [11], are more effective when there are many contending threads.

Figure 2 shows our biased lock scheme. The identifier *this_thread_id* contains a unique number identifying the current thread. The non-owner threads first compete for the *N*-process lock; whichever thread wins the competition then competes for the two-process lock with the owner process.

It is easy to see the scheme assures mutual exclusion among the threads provided the two locking protocols work and thread IDs are well-behaved; other properties depend on the locking protocols themselves. For example, the combined protocol is starvation-free if both locking protocols are; if only the 2-process locking protocol is starvation-free, the owner is always guaranteed to obtain the lock but one or more of the non-owning threads could remain forever in the waiting state. Similar results hold for bounded waiting, assuming starvation-freedom.

This scheme can be implemented by employing Dekker’s algorithm for 2-process locking and the compare_and_swap spin-lock algorithm from the introduction for *N*-process locking. Such an implementation is similar to the scheme proposed by Onodera et al. [12], but differs in the details of how *N*-process locking is invoked.

An alternative: use Peterson’s algorithm (Figure 3) for 2-process locking and the MCS algorithm for *N*-process locking. On the Intel architecture, Peterson’s algorithm requires memory fences to ensure operations issued before the fence

```
typedef struct {
  Thread Id owner;
  Lock2 t; /* lightweight, 2−process lock */
  Locks n; /* N−process lock */
} Lock;

biased_lock(Lock *l) {
  if (this_thread_id == l−>owner)
    lock2(l−>t);
  else {
    lockN(l−>n);
    lock2(l−>t);
  }
}

biased_unlock(Lock *l) {
  if (this_thread_id == l−>owner)
    unlock2(l−>t);
  else {
    unlock2(l−>t);
    unlocks(l−>n);
  }
}
```

Figure 2. Our general biased-lock scheme

```
flag[i] = 1;
turn = j;
fence(); /* force other threads to see flag and turn */
while (flag[j] && turn == j) {} /* spin */
/* ...critical section... */
fence(); /* make visible changes made in critical section */
flag[i] = 0;
```

Figure 3. Peterson's mutual exclusion algorithm for process *i*. Its correctness demands memory fences.

are carried out before operations issued after the fence and to ensure that updates to shared variables are made visible to other threads. This is because newer x86 implementations employ "store-forwarding" that effectively propagates memory updates lazily, depositing them in processor-local store buffer before ultimately dispatching them to the memory system. Hence the store buffer functions as an additional level of cache and improves performance.

Unfortunately, store buffers break sequential memory consistency between processors. To ensure local sequential consistency, a processor always consults its local store buffer on a read to ensure it sees all its earlier writes, but the contents of each processor's (local) store buffer are not made visible to other processors, meaning a shared memory update may be delayed or even missed by other processors. For instance, if variables *x* and *y* are both initialized to 0, one thread executes *write x 1; read y*, and another thread executes *write y 1; read x*, it is possible under store-forwarding for both threads to read 0 for both *x* and *y*, an outcome that is impossible under sequential consistency. Intel's reference manual [1] provides more details and examples.

In the protocol in Figure 3, absent the first fence, thread *i* may not see the updated flag value of thread *j* and thread *j* may not see the updated flag value of thread *i*. This would would allow both threads to enter the critical section concurrently, violating mutual exclusion. The second fence ensures that all changes to global variables made in the critical section are made visible to other processors.

3. Transferring Ownership On-The-Fly

Our biased lock scheme from the last section assumes that the dominant thread is fixed and known in advance. However, certain applications may need to change a lock's dominant thread, such as when ownership of shared data is passed

```
typedef struct {
  Thread Id owner;
  Lock2 t; /* lightweight, 2−process lock */
  lockN n; /* N−process lock */
  bool try[NTHREADS];
} Lock;

void biased_lock(Lock *l) {
  l−>try[this_thread_id] = 1;
  fence();
  if (this_thread_id == l−>owner) {
    lock2(l−>t);
    if (this_thread_id != l−>owner) {
      /* owner has changed */
      unlock2(l−>t);
      goto NON_OWNER;
    } else {
      /* owner has not changed */
      l−>try[this_thread_id] = 0;
    }
  } else {
NON_OWNER:
    l−>try[this_thread_id] = 0
    lockN(l−>n);
    lock2(l−>t);
  }
}

void switch_to_dominant(Lock *l)
{
  lockN(l−>n);
  lock2(l−>t);
  Pres_owner = l−>owner;
  l−>owner = this_thread_id;
  unlock2(l−>t);
  while (l−>try[prep_owner]) {}
  unlock N(l−>n);
}

void biased_unlock(Lock *l){
  if (this_thread_id == l−>owner)
    unlock2(l−>t);
  else {
    unlock2(l−>t);
    unlockN(l−>n);
  }
}
```

Figure 4. Bias Transfer

to a different thread. We call this ownership transfer or re-reservation. In this section, we describe a simple method for effecting this transfer. Figure 4 shows the outline of our method. We do not fix a particular condition for switching ownership—each application may define its own condition for when a switch is necessary. One such scheme, for instance, is to maintain an average frequency of usage of a lock by each thread, and switch ownership when the frequency of a non-dominant thread exceeds that of the dominant one.

The bias transfer mechanism necessarily switches the status of a non-dominant thread. There are certain times when doing so is not safe. For example, it would be incorrect to do so when the dominant thread is about to enter its critical section, so we require a non-dominant thread hold the biased lock before switching its status to dominant. This requirement is not, however, sufficient in itself. A thread may switch to being dominant at a point in time where the earlier dominant thread (line 12) is waiting for its lock. Therefore, we demand additional synchronization between the old and new dominant threads.

The *try* flag array (line 5), which has one entry per thread, provides synchronization. If thread *A* is the dominant thread, the *try*[*A*] entry, if set, indicates to other threads that the owner *may be* in the process of acquiring the lock in lines 9–12. Meanwhile, if some other thread (say, *B*) tries to become dominant by calling *switch_to_dominant*, then *B* changes the owner, and waits for the previously dominant thread *A* to reach a stable state, one where it is certain that *A* has reacted to the change of ownership (line 36).

This procedure adds a few instructions (starting line 9) to the lock algorithm for the owner thread. The overhead is two additional assignments and one test, due to the infrequency of owner switching and the expected infrequency in non-owner locks.

4. Mutual Exclusion and Memory Fences

Given the high cost of atomic and fence operations, one may wonder whether there are mutual exclusion schemes where these operations are not needed. Classical algorithms such as Dekker or Peterson do not use atomic operations, but do require fences to be correct on modern architectures. In this section, we show that the use of fences is unavoidable if the architecture supports store-buffer forwarding unless certain requirements are relaxed.

Fence and atomic operations have the property that they both make prior memory updates "visible" to all other processors in a shared-memory system. Hence, the following definition.

Definition 1 *A* revealing operation *makes updates to shared variables performed in the current thread prior to the operation visible to other processors, i.e., if a different processor reads one of these variables, it obtains the same value as the current thread would if it were to read just before the revealing operation.*

Without a revealing operation, updates may never be propagated to other processors. The statement of our first theorem is not particularly surprising, but it is interesting to see where the requirement of a revealing operation arises in the proof.

Theorem 1 *Any mutual exclusion protocol that ensures freedom from starvation must use a revealing operation within every matched lock-unlock pair for each thread in the protocol.*

PROOF by contradiction. Suppose there is a protocol meeting the assumptions, i.e., the protocol meets the following conditions: it (C1) ensures mutual exclusion, (C2) ensures starvation-freedom for each thread, assuming each thread stays in its critical section for a finite amount of time, and (C3) and does so assuming a demonic scheduler.

Consider the operation of the protocol on a pair of threads, A and B, where operations in A's lock, unlock, and critical section code do not use any revealing operation. Now, suppose A and B start at their initial state. If A is at a lock operation and runs by itself, by (C2), it must enter its critical section. After the point where A enters its critical section, consider a new continuation, E1, where B executes its lock instruction. By (C1), thread B is enabled but must wait since A is in its critical section.

Continue E1 so that thread A exits its critical section and then B runs by itself. By (C2), B must enter its critical section. The decision by B to enter its critical section cannot be made on local information alone since otherwise there is a different schedule where, by (C3), the demonic scheduler can give sufficient time to B to make its decision while A is in its critical region, violating (C1). Thus, between the point in E1 where B waits to the point where it enters its critical section, A must have changed at least one global variable also visible to B and these critical changes must have been made be visible to B.

Now consider an alternative execution, E2, from the point in E1 where A enters its critical section and B is waiting such that in E2, thread A exits its critical section but changes to the global variables by A are not made visible to B. Such an execution is allowed since A is assumed not to execute a revealing operation within its lock-unlock actions. Without a revealing operation, the architecture is not constrained to make the shared-variable updates in A visible to B. Thus, the state visible to B is unchanged. There are two cases to consider at this point. If A cannot acquire the lock again (e.g., if the lock is turn-based), then both A and B are blocked, leading to starvation. If A can acquire the lock, the prior sequence can be repeated, leading to starvation for B. In either case, thread B does not enter its critical section, contradicting (C2). □

This theorem raises the question of whether revealing operations can be eliminated by giving up starvation-freedom. The next theorem shows that this is not possible for most standard protocols, all of which have the following property.

Definition 2 *A* symmetric choice *point in a mutual exclusion protocol is a state where two or more threads are waiting to enter a critical section and* either *process can win the race.*

A mutual exclusion protocol has the *symmetric choice* property if there is a reachable symmetric choice point. The standard mutual exclusion protocols by Dekker, Peterson, and Lamport as well as the spin-lock protocol presented in the introduction have the symmetric choice property.

Theorem 2 *A mutual exclusion protocol requires a revealing operation for each acquire operation at a symmetric choice point.*

```
 1  typedef struct {
      Thread Id owner;
 3    lockN n; /∗ N−process lock ∗/
      bool request;
 5    bool grant;
    } Lock;
 7
    biased_lock(Lock ∗l) {
 9    if (this_thread_id == l−>owner)
        while (l−>grant) {} /∗ wait ∗/
11    else {
        lockN(l−>n);
13      l−>request = 1;
        while (!l−>grant) {} /∗ wait ∗/
15    }
    }
17
    biased_unlock(Lock ∗l) {
19    if (this_thread_id == l−>owner) {
        if (l−>request) {
21        l−>request = 0;
          fence(); /∗ make visible all memory updates ∗/
23        l−>grant = 1;
        }
25    } else {
        fence();
27      l−>grant = 0;
        unlockN(l−>n);
29    }
    }
```

Figure 5. Our asymmetric lock algorithm

PROOF by contradiction. Suppose there is a mutual exclusion protocol with a symmetric choice point, *s*, where two threads, A and B are waiting to enter the critical section and A does not have a revealing operation in its acquire operation.

By definition of symmetric choice, there is an execution E1 from *s* where A acquires the lock first and another execution, E2, from *s* where B acquires the lock first. Construct execution E3 by first executing E1, then E2. Since there is no revealing operation in E1, the values of the shared variables as seen by process B at the end of E1 are the same as that in *s*, and the local state of *B* is unchanged by E1 (*B* remains in its waiting state). Therefore, it is possible to append execution E2 to E1, but the sequence E1;E2 results in both A and B acquiring the lock concurrently, violating mutual exclusion. □

5. Asymmetric Locks

Theorem 2 implies an algorithm that avoids revealing operations in locks must avoid the formation of a symmetric choice state—i.e., it must be asymmetric in some sense. In this section, we present such an algorithm.

For this section only, we return to assuming there is a fixed, known dominant thread. The algorithm is made asymmetric by forcing the non-dominant threads to request permission from the dominant thread to proceed. The outline of the algorithm is shown in Figure 5

Before entering the critical section, the dominant thread checks whether another thread is accessing the critical section by probing the grant variable (line 10). While leaving the critical section (lines 20–24), it checks the request flag to determine whether another (non-dominant) thread wishes to enter the critical section. If the flag is set, the dominant process hands the lock to the other thread by calling *fence* and setting the *grant* variable to 1. The call to *fence* commits any changes to shared variables made in the critical section before it passes the lock to any other thread.

A non-dominant thread that desires to enter the critical section (lines 12–14) must first acquire a n-process lock, then set the request flag and wait for a grant. While leaving the critical section (lines 26–28), it resets the *grant* variable after calling *fence*. The call to *fence* commits all local changes to the main memory before the lock is passed back to the dominant process.

This method has the disadvantage that a non-dominant requesting thread must wait for the dominant process to grant it permission. This, in turn, implies that the dominant thread must periodically check the request flag. Thus, the algorithm ensures starvation-freedom for the non-dominant threads only when the dominant thread checks the request flag infinitely often in any infinite computation. One way to ensure this is to use a periodic polling strategy.

The advantage of the algorithm is that the dominant thread does not use a compare-and-swap instruction and uses a fence instruction *only* only when it passes control of the critical region to a non-dominant thread. In periods of no contention from other threads, the dominant thread does not use any atomic or fence instructions, so locking incurs very little overhead.

6. Read-Write Biased Locks

In this section, instead of considering only exclusive locks, we discuss the design of biased read-write locks that incur very little overhead on the dominant thread. In general, a read-write lock allows either multiple readers or a single writer to access a critical section at any time.

We use a combination of 2-process lock and n-process lock. For the 2-process lock, we use a modified version of Peterson's algorithm; see Figure 6 and Figure 7. The flag variable can take three values: *READ*, *WRITE*, and *UNLOCK*. When a dominant thread *i* tries to obtain a read lock, it spins if at the same time there is another thread *j* writing (lines 13–15 in Figure 6). When the dominant thread tries to obtain a write lock, it waits if there is another thread that is either reading or writing (lines 4–6, Figure 7).

For a non-dominant process to acquire a write lock (lines 9–11, Figure 7), it first acquires a normal *n*-process write lock, *nrw*. This write lock *nrw* is contended only by non-dominant processes. Once this lock is obtained, the process checks if the dominating process is in the unlock state and then enters the critical section. At this point the non-dominant process is the only process in the critical section because the *nRWLock* provides exclusive access among the non-dominant processes. The Peterson-like algorithm that follows it provides exclusive access from the dominant thread.

For a non-dominant process to acquire a read lock (lines 17–25, Figure 6), it first acquires a normal *n*-process read lock on *n*. Since the *n*-process read lock on *nrw* can be held by multiple non-dominating processes, the first non-dominant reader competes with the dominant process. If the dominant process

```
   typedef struct {
 2   Thread-Id owner;
     int flagi; /* Owner's flag */
 4   int flagi; /* Non-owner's flag */
     bool turn;
 6   rwlock nrw; /* N-process read-write lock */
     Locke n; /* N-process lock*/
 8   int non_owner_readers; /* No. of non-dominant readers */
   } Lock;
10
   biased_r_lock(Lock *l) {
12   if (this_thread_id == l->owner) {
       l->flagi = READ;
14     l->turn = j;
       while (l->turn == j && l->flags == WRITE) {}
16   } else {
       nRWLock(l->nrw, READ); /* Get a read lock */
18     lockN(l->n); /* Get an exclusive lock */
       l->non_owner_readers++;
20     if (l->non_owner_readers == 1) { /* First reader */
          l->flagi = READ;
22        l->turn = i;
          while (l->turn == i && l->flagi == WRITE) {}
24     }
       unlockN(l->n);
26
     }
28 }

30 biased_r_unlock(Lock *l) {
     if (this_thread_id == l->owner)
32     l->flagi = UNLOCK;
     else {
34     lockN(l->n);
       l->non_owner_readers--;
36     if (l->non_owner_readers == 0)
         l->flags = UNLOCK;
38     unlockN(l->n);
       nrw-Unlock(l->nrw); }
40 }
```

Figure 6. Read function of biased read-write locks

is busy writing, the first non-dominant reader spins on the flag variable. The last non-dominant reader to exit the critical section sets the *flagj* variable to *UNLOCK*. The first and last readers are maintained by a counter variable *non_owner_readers* and the field is protected by a normal *n*-process lock *n*.

As in the previous sections, for the dominant process to obtain either a read lock (lines 12–15) or write lock (lines 3–6) when there is no contention, it only needs to set two flags and compare the two flags, It therefore has far less overhead than normal *n*-process read-write locks.

The *nRWLock* function, which obtains a normal *n*-process read or write lock, can use standard reader writer locks and implemented to be reader starvation-free or writer starvation-free. Between the dominant and non-dominant process, the writer dominant process may starve, especially when non-dominant readers keep coming in and never relinquish the lock. But since these readers are non-dominant, we expect the readers to arrive infrequently and therefore the dominant process cannot starve to: write.

```
   biased_w_lock(Lock *l)
 2 {
     if (this_thread_id == owner) {
 4     l->flagi = WRITE;
       l->turn = j;
 6     while (l->turn == j && l->flagi != UNLOCK) {}
     } else {
 8     nRWLock(l->nrw, WRITE);
       l->flags = WRITE;
10     l->turn = i;
       while (l->turn == i && l->flagi != UNLOCK) {}
12   }
   }
14
   biased_w_unlock(Lock *l) {
16   if (this_thread_id == l->owner)
       flagi = UNLOCK;
18   else {
       flagj = UNLOCK;
20     nrw-Unlock(l->nrw);
     }
22 }
```

Figure 7. Write functions of biased read-write locks

7. Algorithm Verification

The correctness of the algorithm presented in Section 2 can be inferred easily from its construction. The *n*-lock provides mutual exclusion among non-dominant threads. The 2-lock provides mutual exclusion among the dominant and the non-dominant thread.

The correctness of the asymmetric algorithm is less obvious and in fact, we discovered several pitfalls while developing it. We verified the algorithm from Section 5 using the SPIN [8] model checker. We created two processes, one dominant; the other non-dominant, and verified mutual exclusion and progress properties. Even when there is more than one non-dominant thread in the system, mutual exclusion holds because the normal *n*-lock provides exclusive access among the non-dominant threads. The progress property also holds if the normal lock satisfies the progress property. The bounded waiting property however is not satisfied because non-dominant threads are dependent on the dominant thread to acquire the lock.

For the ownership transfer protocol, we were able to verify with SPIN a configuration with one dominant thread and two non-dominant threads. Each non-dominant thread attempts non-deterministically to change ownership. We believe that this configuration describes all interesting interactions; the generalization of this automatic proof to arbitrary numbers of threads is ongoing work.

We also verified the biased read-write protocol using SPIN. We created one dominant thread and two non-dominant threads in SPIN. Each of these threads non-deterministically attempt a read or a write lock. The mutual exclusion property is satisfied even when there are more than two non-dominant threads because the non-dominant thread has to acquire either

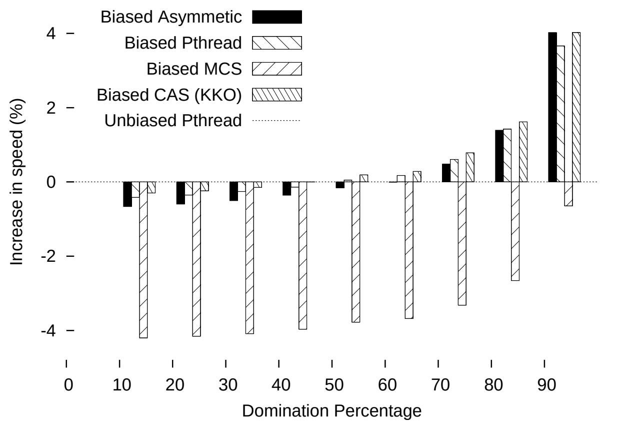

Figure 8. A comparison of the algorithms for varying domination percentages.

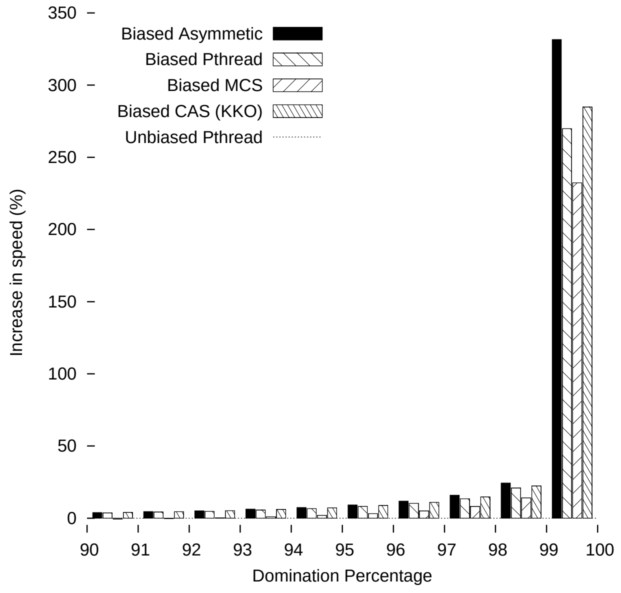

Figure 9. A comparison of the algorithms for high domination percentages.

a normal *n*-write-lock or *n*-read-lock depending on the action before entering the critical section.

The verification with SPIN is based on a sequentially consistent model. By perturbing the sequence of assignments, it is possible to discover which orderings are relevant for the proof of correctness; this indicates positions where fences must be inserted for correctness on modern architectures with weaker ordering guarantees. In the future, we plan to use tools such as Check-Fence [4] to determine optimum placement of fences.

8. Experimental Results

The experiments described in this section have two purposes: to compare the performance of the new biased lock algorithms against similar algorithms proposed in earlier work using the pthread spin-lock implementation on Linux as the base reference, and to confirm our intuition about the behavior

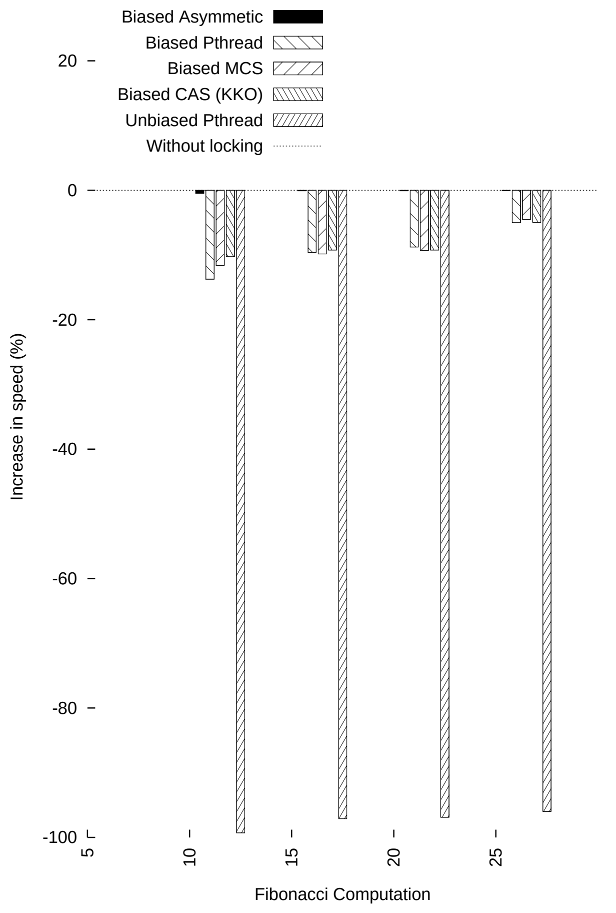

Figure 10. Lock overhead for a sequential program.

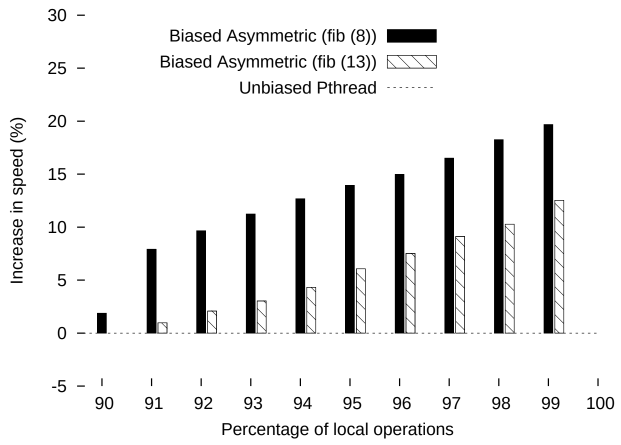

Figure 11. Behavior of our packet-processing simulator with asymmetric locks.

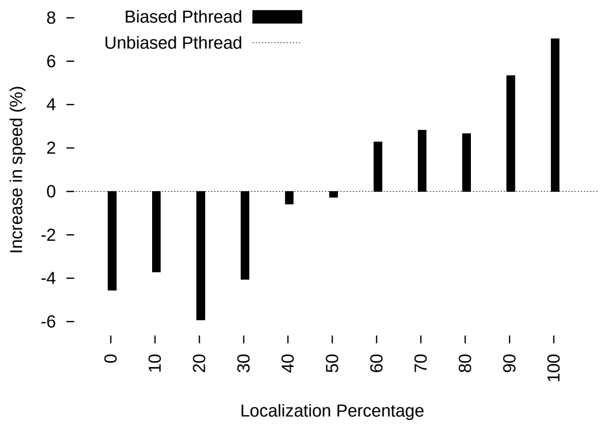

Figure 12. Performance of our biased locks on a database simulator for the query SELECT SUM(C2) GROUP BY C1.

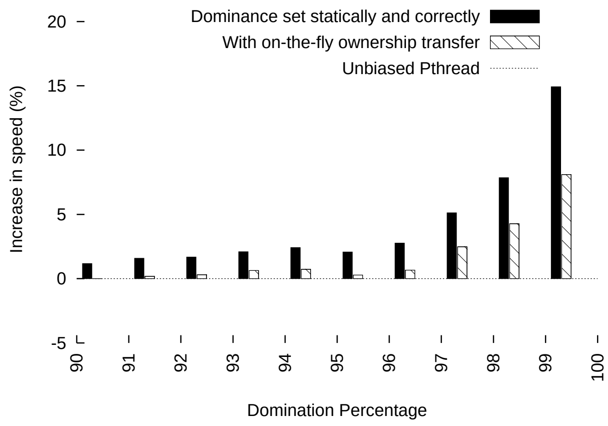

Figure 13. The effect of bias transfer.

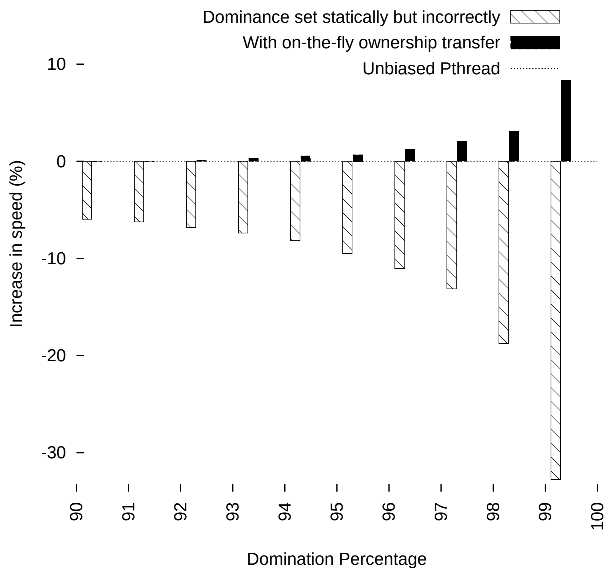

Figure 14. The effect of bias transfer for incorrect biasing.

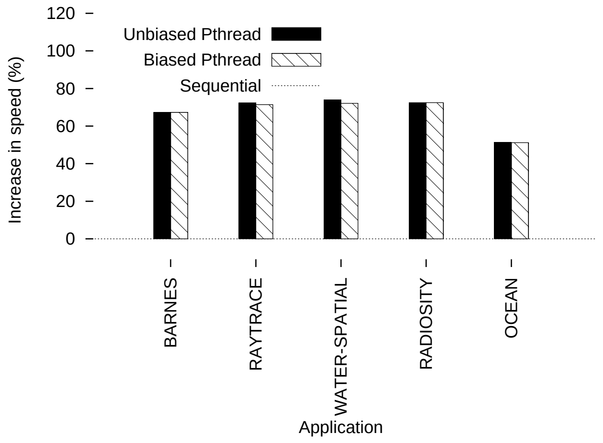

Figure 15. Performance of our biased locks on applications (SPLASH2 benchmark) without dominant behavior.

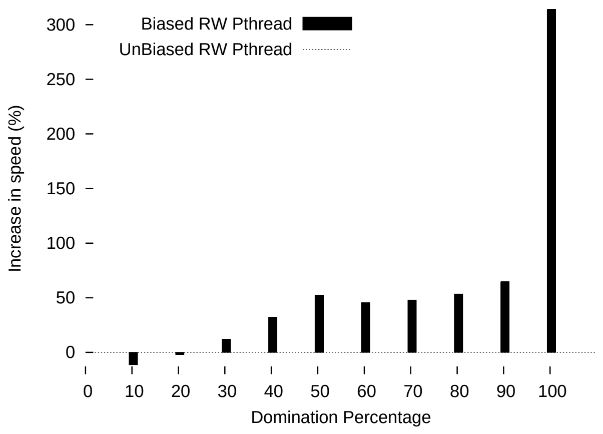

Figure 16. A comparison of our biased rwlock s with Linux thread rwlocks.

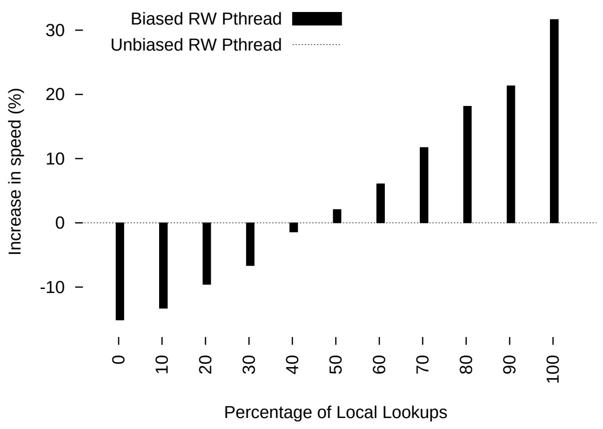

Figure 17. Performance of our biased read-write locks on a router simulator with 95% reads and 5% writes.

of these algorithms, i.e., that the performance improves monotonically with increasing domination.

Performance with varying domination. To compare the different algorithms, we ran a set of experiments on an Intel quad-core machine running Linux with 2 GB of memory. We created four threads and made one of them dominant. The critical section just incremented a counter—a deliberately small task to maximize lock overhead. We varied the dominance percentage and measured the execution times; see Figure 8. A dominance of 90% indicates that for 100 accesses to the critical section, the dominant thread accesses the critical section 90 times and the remaining threads access the critical section 10 times. The lock accesses were evenly spaced: they follow a skewed but non-bursty access pattern.

We tested our micro benchmark on various algorithms. Our base case for comparison is the pthread spin lock (represented by a horizontal line at 0). We described this biased asymmetric lock in Section 5. The biased thread implementation uses Peterson's algorithm for 2-lock and p-threads for *n*-lock. The biased MCS implementation uses Peterson along with MCS locks. The biased CAS is the implementation from Kawachiya et al. [12]

For each of these algorithms, we observed the performance improve as we increased the dominance of the owner thread. Figure 9 shows details of the results from Figure 8 for domination between 90 and 100%. Not surprisingly, the asymmetric method performs best when the domination percentage is high because asymmetric locks are very lightweight and do not require fence instructions in the dominant thread when there is no intervention from other threads. On the other hand, when the domination is less, the non-dominant threads have to wait until the dominant thread signals; this overhead is insignificant for high dominance.

Locked vs. luckless sequential computation. Next, to measure the overhead of each of these locks, we created a sequential program that, to represent work, does the naïve recursive Fibonacci computation $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$, and thus shows exponential behavior with increasing *n*. We protected the counter by a lock and compared the performance of different locks with the version without locks; see Figure 10. This setup merely measures the overhead of these locks. First, we see that the thread locks has the maximum overhead (about 100%) and asymmetric locks has the least (less than 1%). Second, as the computation load increases, the relative overhead decreases slowly.

Performance of a packet-processing simulator with asymmetric locks. From these experiments, we concluded that asymmetric locks are the best for our packet-processing application. Also, since the non-dominant threads require permission from the dominant/owner thread to enter the critical section, asymmetric locks are suitable for applications that have dominant threads that run forever. In our packet-processing application, we replaced the thread locks by our asymmetric locks and compared the performance with the original one with pthread s (Figure 11). Within each lock we also added a synthetic computation that calculates Fibonacci numbers. When the computation time is high (e.g., for fib(13)), the non-dominant threads have to wait more for the dominant thread to signal, therefore we see fib(8) performing better than fib(13). The difference between the two loads is roughly a factor of 10 because $\text{fib}(n)$ scales as ϕ^n, where $\phi \approx 1.618$ is the golden ratio.

Biased Locks for Database Queries. Flexible locks (Section 2) that consist of 2-locks combined with *n*-locks are more robust to variations in computational load, although they require fence instructions whenever a dominant lock is obtained or released. To test the behavior of these locks, we wrote code that performs the SOL query "SELECT aggregate_function(c1) FROM t group by c2." Such a query is typically processed concurrently. The table *t* is divided into *n* parts; each part is processed by a separate thread, which maintains a local hash table. If the data c2 in *t* is localized, most of the hash updates are local to the thread, otherwise it is necessary to modify the hash table of a different thread; see Figure 12. As the locality of data increases, the biased locks perform better. Although the performance depends strongly on how the data are ordered, in many cases the ordering is such that data are localized.

Ownership transfer. We fixed the ownership of locks in the above applications, but our algorithm in Section 3 allows for ownership transfer. To test its performance, we created four threads that each perform a Fibonacci calculation in the critical section. Figure 13 compares the performance of our biased locks that supports on-the-fly ownership changes with the implementation that only supports static ownership. The implementation that supports change of ownership does not do as well as the the static implementation because of the extra overhead to support bias transfer. The ownership changes to the thread that was recently dominant, i.e, the most recent thread that has been acquiring the lock continuously. However, it does better than the unbiased implementation.

Ownership transfer with incorrect dominance. In Figure 14, we also compare the ownership on-the-fly implementation with a static ownership implementation, but for the latter implementation, we set the dominance incorrectly. The ownership on-the-fly implementation easily adapts itself and changes the dominance to the most recently dominant.

Overheads for non-dominant behavior. The general trend is that as the dominance increases, biased locks perform better than unbiased locks. With applications that do not exhibit dominant behavior, we do not expect any improvement. We tested our biased locks on SPLASH2 benchmarks [15]. Most of these benchmarks exhibit master-slave behavior where work is divided among different threads. Even in the absence of dominance, our biased implementation deteriorated by at most 2% compared to the sequential version for these benchmarks.

Performance of biased read-write locks. Finally, we compared our biased read-write lock implementation with the thread implementation of read-write locks. Figure 16 shows the results. Our biased read-write lock performs very well even when the dominance fraction is not very high because read-write locks are generally very expensive and our dominant read-write lock optimizes it to a large extent.

Performance on a simulated router application. To test the effect of biased read-write locks on actual examples, we simulated a router application. A router maintains a look-up table where the entries are mostly static, but occasionally (5% of the time) the IP addresses change, in which case a write lock is required. It usually maintains a distributed look-up table in which most lookups are local to a thread. Figure 17 suggests that, as expected, as we increase the number of local lookups, the biased read-write locks perform better.

9. Related Work and Conclusions

As we mentioned in the introduction, there is other work on optimizing lock implementations, such as thin locks [3] and lock-reservation algorithms [2, 9, 12]. The original thin lock algorithm requires a compare-and-swap on each lock acquisition, which our algorithm avoids.

The lock-reservation work is closest to ours. In Kawachiya et al. [9], the disadvantage is that when a lock is reserved for the owner and the non-owner tries to attempt the lock, the non-owner stops the owner thread and replaces a lock word. This step is very expensive because the owner thread is suspended. Onodera et al. [12] proposes a modification similar to ours: a hybrid algorithm that tightly intertwines Dekker's 2-process algorithm with an *n*-process CAS algorithm. Our scheme simplifies this by keeping the two algorithms separate and generalizes it by allowing any choice of 2-process and *n*-process mutual algorithms.

We show how to transfer lock ownership among threads without suspending the current owner. Although Russell and Detlefs [14] also support bias transfer, their global safe-point technique for bias revocation is costly. In their technique, it is difficult to determine at any point whether a biased lock is actually held by a given thread. Our technique is simple and inexpensive: it only requires two extra assignments and two comparisons.

Finally, we examined the necessity of memory fence instructions on modern processors and shed light out the key role played by the symmetric choice property of most mutual-exclusion algorithms. The asymmetric algorithm presented in Section 5 is, in a sense, the most efficient possible, since it avoids both memory fence and atomic operations in the dominant process except at the point of transferring control of the lock, where they are unavoidable. Earlier work on asymmetric biased locks [5, 6] has a similar motivation, but the analogue of the request-grant protocol, called SERIALIZE(t) by Dice et al. [5], appear fairly heavyweight, involving either thread suspension and program counter examination, or context-switches.

Our experimental evaluation shows that, in practice, our algorithms perform well when the dominance fraction is high, as expected. This matches the profile of our intended applications, e.g., network packet processing. The evaluations were all carried out on an Intel Quad core machine and the results, therefore, reflect the relatively high costs of fence and atomic operations on the x86 architecture. The trade-off point may be different for other architectures; we leave this to future work.

References

[1] Intel IA-32 Architecture Software Developer's Manual, vol. 3A, 2009. `http://www.intel.com/products/processor/manuals/`.

[2] David Bacon and Stephen Fink. Method and apparatus to provide concurrency control over objects without atomic operations on non-shared objects. US Patent Docket No. YO999-614, 2000.

[3] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Thin locks: featherweight synchronization for java. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, New York, NY, USA, 1998. ACM.

[4] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 12–21, New York, NY, USA, 2007. ACM.

[5] Dave Dice, Hui Huang, and Mingyao Yang. Asymmetric dekker synchronization. Technical report, Sun Microsystems, 2001. http://home.comcast.net/ pjbishop/Dave.

[6] Dave Dice, Mark Moir, and William Scherer. Quickly reacquirable locks. Technical report, Sun Microsystems, 2003. http://home.comcast.net/ pjbishop/Dave.

[7] Edsger W. Dijkstra. Cooperating sequential processes. *Technological University, Eindhoven, The Netherlands, September 1965. Reprinted in Programming Languages, F. Genuys, Ed., Academic Press, New York, 1968, 43-112*, 1965.

[8] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[9] Kiyokuni Kawachiya, Akira Koseki, and Tamiya Onodera. Lock reservation: Java locks can mostly do without atomic operations. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 130–141, New York, NY, USA, 2002. ACM.

[10] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[11] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

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

[13] G. L. Peterson. Myths about the mutual exclusion problem. *IPL 12(3)*, pages 115–116, 1981.

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

[15] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.