Reordering Constraints for Pthread-Style Locks

Hans-J. Boehm
Advanced Architecture Laboratory
HP Laboratories Palo Alto
HPL-2005-217(R.1)
September 29, 2006*

threads, locks, memory barriers, memory fences, code reordering, data race, pthreads, optimization

C or C++ programs relying on the pthreads interface for concurrency are required to use a specified set of functions to avoid data races, and to ensure memory visibility across threads. Although the detailed rules are not completely clear[10], it is not hard to refine them to a simple set of clear and uncontroversial rules for at least a subset of the C language that excludes structures (and hence bit-fields).

We precisely address the question of how locks in this subset must be implemented, and particularly when other memory operations can be reordered with respect to locks. This impacts the memory fences required in lock implementations, and hence has significant performance impact. Along the way, we show that a significant class of common compiler transformations are actually safe in the presence of pthreads, something which appears to have received minimal attention in the past.

We show that, surprisingly to us, the reordering constraints are not symmetric for the lock and unlock operations. In particular, it is not always safe to move memory operations into a locked region by delaying them past a pthread mutex lock() call, but it is safe to move them into such a region by advancing them to before a pthread mutex unlock() call. We believe that this was not previously recognized, and there is evidence that it is under appreciated among implementors of thread libraries.

Although our precise arguments are expressed in terms of statement reordering within a small subset language, we believe that our results capture the situation for a full C/C++ implementation. We also argue that our results are insensitive to the details of our literal (and reasonable, though possibly unintended) interpretation of the pthread standard. We believe that they accurately reflect hardware memory ordering constraints in addition to compiler constraints. And they appear to have implications beyond pthread environments.

Reordering Constraints for Pthread-Style Locks

Hans-J. Boehm
HP Laboratories
Hans.Boehm@hp.com

Abstract

C or C++ programs relying on the pthreads interface for concurrency are required to use a specified set of functions to avoid data races, and to ensure memory visibility across threads. Although the detailed rules are not completely clear[10], it is not hard to refine them to a simple set of clear and uncontroversial rules for at least a subset of the C language that excludes structures (and hence bit-fields).

We precisely address the question of how locks in this subset must be implemented, and particularly when other memory operations can be reordered with respect to locks. This impacts the memory fences required in lock implementations, and hence has significant performance impact. Along the way, we show that a significant class of common compiler transformations are actually safe in the presence of pthreads, something which appears to have received minimal attention in the past.

We show that, surprisingly to us, the reordering constraints are not symmetric for the lock and unlock operations. In particular, it is not always safe to move memory operations into a locked region by delaying them past a `pthread_mutex_lock()` call, but it is safe to move them into such a region by advancing them to before a `pthread_mutex_unlock()` call. We believe that this was not previously recognized, and there is evidence that it is under-appreciated among implementors of thread libraries.

Although our precise arguments are expressed in terms of statement reordering within a small subset language, we believe that our results capture the situation for a full C/C++ implementation. We also argue that our results are insensitive to the details of our literal (and reasonable, though possibly unintended) interpretation of the pthread standard. We believe that they accurately reflect hardware memory ordering constraints in addition to compiler constraints. And they appear to have implications beyond pthread environments.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Concurrent programming structures; D.3.4 [*Programming Languages*]: Optimization

General Terms Languages, Performance

Keywords threads, locks, memory barriers, memory fences, code reordering, data race, pthreads, optimization

1. Introduction

Due in large part to the rise of multi-core and hardware-multi-threaded processors, explicitly parallel applications are increasingly essential for high performance, even on mainstream desktop machines. Multi-threaded applications are perhaps the most common way to achieve this, at least when it is not useful to simply run multiple copies of applications. Multiple threads are also often used for structuring even uniprocessor applications to make it easier to deal with multiple event streams. As a result, most processes on a typical Microsoft Windows desktop are already composed of multiple threads[1] as are a large and increasing number of server applications.

Most multi-threaded applications are written in C or C++ with the aid of a thread library. For the purposes of this paper, we will assume that the thread library obeys the pthread[13] specification. We believe that much of the discussion here is applicable to other platforms, but the issues there are often less clear.

The fundamental rule governing shared variable access under Posix threads prohibits *data races*, i.e. simultaneous access to the same location by multiple threads, when one of the accesses is a write. The pthread specification[27] states:

> "Applications shall ensure that access to any memory location by more than one thread of control (threads or processes) is restricted such that no thread of control can read or modify a memory location while another thread of control may be modifying it. Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads. The following functions synchronize memory with respect to other threads:
>
> ...,
> `pthread_mutex_lock()`,
> ...,
> `pthread_mutex_trylock()`,
> `pthread_mutex_unlock()`,
> `pthread_spin_lock()`,
> `pthread_spin_trylock()`,
> `pthread_spin_unlock()`,
> ..."

We believe that, aside from the specific synchronization functions, this also reflects the intended programming model for some other multi-threaded environments, e.g. Microsoft's win32 threads.

All implementations of which we are aware ensure that the above "synchronizing" functions either contain memory fences (sometimes called *barriers*) to restrict reordering of memory operations by the hardware, or the corresponding reordering constraint is implicit in other instructions inside the function.

These implementations also ensure that these functions are treated as opaque by the compiler. The compiler must assume that all such functions potentially read and/or update any variable in the program. Thus the compiler cannot safely move references to potentially global variables across calls to such functions. Movement of references to globals across synchronization calls is avoided, even though the compiler operates as though it were targeting a single-threaded environment.

[1] On the author's Windows 2000 desktop, only 3 out of 37 processes were single-threaded. One of those was the system idle process.

Thus compilers may aggressively rearrange code between calls to these "synchronizing functions". Between synchronization calls even shared variables may appear inconsistent to other threads. But this would only be observable by client code if another thread were to simultaneously access such shared variables, which the above rule prohibits it from doing.

This approach is fundamentally different from that used in Java[21]. No attempt is made to ensure type-safety or security for sand-boxed code. On the other hand, it has some, probably significant, performance advantages[7].

As is pointed out in [10], this approach requires some more precision in the language specification to ensure correctness. For the purposes of this paper, we will give more precise rules, which can be viewed as refinements of the pthreads specification, and avoid those issues. Furthermore none of the discussion here is dependent on language features (e.g. bit-fields or atomic operations) which might make those rules controversial and have made the development of a C++ memory model interesting[5, 4, 7, 23]. Hence we simply omit further discussion of such language features.

Instead we focus on another issue which really must be resolved in order to correctly implement (our interpretation of) the current pthread standard, and to allow meaningful discussion of the performance of pthread implementations.

We ask the seemingly simple question of when memory operations may be reordered with locking operations.

Part of this has an obvious answer: It is clearly not generally acceptable to move memory operations out of critical sections, i.e. regions in which a lock is held; doing so would introduce a data race. But it is far less clear whether it is acceptable to move memory operations *into* critical sections, which is the question we pursue here.

This affects the implementation in two distinct ways:

1. The implementation may treat functions like `pthread_mutex_lock` specially, and allow some compiler reordering around direct calls to them. Indirect calls may still have to be treated as completely opaque.
2. It determines the memory fence (or memory barrier) instructions that must be included in the library implementations of synchronization primitives.

Of these, we believe that the latter has by far the greatest performance impact, primarily because memory fence instructions are expensive on many current architectures.

Nonetheless, we choose to address this issue by looking at allowable compile-time transformations of the source programs, mostly because it is easiest to reason at that level. We apply arguments that also apply to the reordering of particular instruction executions. By doing so, we avoid the need to reason about the semantics of specific hardware fence instructions, which are much better defined for certain architectures than others.

In the next two sections, and in the appendix, we argue that the performance impact of this issue on lock-intensive applications can be large, and that this is an issue that impacts current pthread implementations.

We then define a small multi-threaded language, which we claim is sufficient for modeling C/pthread behavior. We define its semantics to be consistent with the pthread definition.

Finally, we show that the rules for reordering memory operations across locking primitives are in fact different from what we believe most implementors would have expected. In particular, a memory operation immediately following a `pthread_mutex_unlock` operation may always be moved to just before the `pthread_mutex_unlock`. But the corresponding reordering of a `pthread_mutex_lock` with a preceding memory operation is not generally safe. More general movement into critical sections is possible in the absence of the `pthread_mutex_trylock` call.

As we point out in section 3, many current implementations either do not follow these rules, or add extra fences. Nor is it completely apparent that they should, since these semantics appear to have been largely accidental. On the other hand, an understanding of these issues does seem to be essential to any revision of the current rules.

2. Performance Impact

The impact of adding memory fences to lock implementations can be substantial. In particular:

- The cost of locking can significantly affect program execution time, even when there is little lock contention[22]. In some cases, this is due to questionable programming practice. In other cases, such as the examples in [10], or reference counting in [11], or in many implementations of the C++ standard string library such as [19], or in memory allocator implementations[8], it is inherent in the problem, and sufficiently serious to have forced significant pieces of software to include non-portable code to reduce locking overhead.

  Although multi-threaded C or C++ benchmarks appear to be rare, the same effect has been regularly measured in Java benchmarks (cf. [6]), though there it is sometimes aggravated by unnecessary synchronization.
- The locking cost is often strongly affected by, or even largely determined by, the number of memory fences (or related instructions that have that effect) in the lock implementation. Thus the cost of a `lock-unlock` operation pair can vary by roughly a factor of two depending on whether a fence is needed in the `unlock` operation. This can be true for spin locks on some common X86 processors. The impact of memory fences was also observed in a Java context in [6].

We present an example of a program dominated by locking, together with some simple measurements, in appendix A, to illustrate the last point.

Based on the above, it appears critical to understand requirements for memory fences before attempting to, for example, benchmark such multi-threaded applications.

Note that both the examples cited above and at least half the measurements in the appendix, exhibit little or no lock contention. The performance of locks under contention is an orthogonal issue, which has been well studied (see e.g. [25] for some recent work and references), and is not addressed here. As the cited examples illustrate, applications may acquire locks with great frequency, but acquire sufficiently many different locks that contention becomes an issue only with very high processor counts.

3. Common Implementations

Our experience is that in practice there is a great deal of confusion surrounding ordering requirements for lock implementations. Since any resulting failures are likely to occur extremely rarely, and perhaps not at all on specific hardware implementations, we attempted to confirm this by inspecting some open source pthreads implementations.

Perhaps the most commonly used modern pthreads implementation is the NPTL implementation distributed as part of glibc. We also suspect that its treatment of fencing requirements in locking operations is fairly typical, and probably relatively carefully considered.

We examined its implementation of `pthread_spin_lock` and `pthread_spin_unlock` in glibc-2.4, and confirmed some of our

Environment	type	lock fence	unlock fence
glibc 2.4 NPTL Itanium	spin	full	release[a]
glibc 2.4 NPTL X86	spin	full	release[b]
glibc 2.4 NPTL Alpha	spin	acquire	release
glibc 2.4 NPTL PowerPC	spin	acquire	release
glibc 2.4 NPTL Itanium	mutex	full	full
glibc 2.4 NPTL X86	mutex	full	full
glibc 2.4 NPTL Alpha	mutex	acquire	release
glibc 2.4 NPTL PowerPC	mutex	acquire	release
FreeBSD 6.1 Itanium	spin	acquire	acquire[c]
FreeBSD 6.1 X86	spin	full	full

[a] The ordering is enforced both with `st.rel` and a blatantly redundant fence.

[b] This makes the common assumption that ordinary X86 stores have release semantics

[c] Based on what appeared to be the intent of the source. The disassembled code actually enforces no ordering for a lock release at all, probably due to an error in an assembly code specification, apparently allowing an `xchg` operation to be optimized away. The incorrect ordering constraint for `pthread_spin_unlock` does not appear to be limited to Itanium, though it does not arise on X86. The FreeBSD mutex code is relatively long and opaque, and we did not attempt to analyze it.

Table 1. Fences used by some pthread implementations.

conclusions for X86 and Itanium with a debugger. We also inspected the corresponding FreeBSD code.

The results are presented in table 1. Lock operations should at least ensure that later memory operations may not become visible before the lock operation. An entry of "acquire" in the table indicates that the implementation obeys this constraint. Similarly, lock release operations should ensure that preceding memory operations may not become visible after the unlock. An entry of "release" indicates that this constraint is enforced. A table entry of "full" indicates that a stronger, and usually more expensive, constraint is enforced, which prevents reordering of all memory operations with respect to the lock. We argue below that this is technically required for lock acquisition, but not release.

On X86, it is difficult or impossible to implement lock acquisition without ensuring "full fence" semantics. Hence these entries do not correspond to additional overhead. A full fence on X86 spin-lock release can easily be avoided. For X86 mutex unlock, there is a tradeoff involved.

(On Alpha and PowerPC, "acquire" and "release" are implemented with a trailing and leading memory fence, respectively. This gives marginally, but we believe uselessly, stronger semantics, which are not otherwise distinguished here.)

Based on the results we give below, all entries should read "full release" (or possibly "full full" for an X86 mutex). But we observe little consistency in practice. In fact, the higher level code is generally written to provide "acquire release" semantics, and deviations from that tend to result from implementation details of the low level atomic memory operations.

4. Foundations

Our arguments are generally insensitive to the specific non-synchronization primitives we allow in our programming language. But to keep the discussion as precise as possible, we will define a specific programming language, modeled on the relevant aspects of C.

Nothing here should be the least bit surprising to the reader. Our only goals are to convince the reader that this could all be completely formalized, if we chose to do so, and to establish the terminology we use later.

Define a statement to be one of the following[2]:

$$\begin{array}{c} v\ \texttt{=}\ r\texttt{;} \\ r\ \texttt{=}\ v\texttt{;} \\ r\ \texttt{=}\ E\texttt{;} \\ r\ \texttt{= in();} \\ \texttt{out(}r\texttt{);} \\ \texttt{lock(}l_i\texttt{);} \\ \texttt{unlock(}l_i\texttt{);} \\ \texttt{r = try\_lock(}l_i\texttt{);} \\ \texttt{while (}r\texttt{)}\ \ S \\ S\ \ S \end{array}$$

Here S denotes another statement, r denotes one of a set of thread-local variables or *registers*, which we will normally write as r_i, and v denotes one of a set of global variables (written v_i). We'll assume that both kinds of variables range over integers, though allowing other variable types does not impact our arguments.

The first two kinds of assignments simply copy variables between globals and registers. We'll refer to the former as a store, and the latter as a load operation.

The third form of assignment statement describes a computation on registers. We do not precisely define the expressions E that may appear on the right sides of such assignments, but we assume that all such expressions mention only register variables. Thus a real C-language assignment would often correspond to one or more loads, followed by a computational assignment, followed by a store.

The statements r `= in();` and `out(`r`);` read a value from an input stream, and write a value to an output stream, respectively.

The `lock` and `unlock` statements acquire and release a lock, respectively. They correspond to the pthread `pthread_mutex_lock` and `pthread_mutex_unlock` primitives. They can operate on locks l_i. In order to keep things as specific and simple as possible, we will require that threads may not reacquire a lock they already hold.

The `trylock` statement behaves like `lock` and returns 1 if the lock is not currently held. But if the lock is held, it simply returns 0 instead of blocking. It models `pthread_mutex_trylock`, except that we use a different return value convention to simplify matters later. The presence of `trylock` affects our results.

A *program* is a finite sequence of statements. Informally, the ith statement describes the actions executed by the ith thread.[3] We will assume that the register names used in the different statements (effectively by the different threads) are disjoint.

We will assume that the individual sub-statements of a program have associated labels, so that we can easily distinguish textually identical sub-statements.

A *thread action* is a pair consisting of a program statement label and the value assigned, tested, read, or written, as appropriate, if any.

Since every statement label is associated with a specific thread, it also identifies the executing thread. Since there is generally no ambiguity, we will normally fail to distinguish between a statement and its label.

[2] We do not explicitly consider condition variables. This is not a substantive restriction, since `pthread_cond_wait()` can be treated as a `pthread_mutex_unlock()` followed by a `pthread_mutex_lock()`, together with a scheduler hint. And the other primitives can be viewed as purely scheduler hints. While these scheduler hints are of critical practical importance, they do not affect correctness.

[3] This is admittedly a very simplistic view in that it does not allow dynamic thread creation. But that again appears to have no bearing on our results.

We will say that a sequence of thread actions is *register consistent* if values assigned by store statements, written by output statements, computed by expression evaluation statements, or tested by loops, are consistent with the values assigned to the input registers by the last prior assignments to that register in the sequence, or with a value of zero if there was no prior assignment to the register.[4]

A statement generates sets of possible *candidate thread executions*, which are possibly infinite sequences of thread actions. Specifically, all statement types, except the last two, describe all sequences consisting of a single action involving that statement. A composite statement describes all possible concatenations of thread executions generated by its components. [5]

Similarly, the candidate thread execution of a while loop consists of alternating thread actions corresponding to the while loop (which reflect the tests) and thread executions generated by the loop body, such that the values of all loop tests except the last are nonzero, and the last is zero.

A candidate thread execution is called a *thread execution* if it is also register consistent.

Hence the statement r_2 `=` v_1`;` r_3 `= 2 *` r_2 generates all candidate thread executions of the form

$$(r_2 = v_1,\ x),\ (r_3 = 2 * r_2,\ y)$$

with any values of x and y, and thread executions of the form

$$(r_2 = v_1,\ x),\ (r_3 = 2 * r_2,\ 2x)$$

with any value of x.

In most cases, we will write sequences of thread actions (or just *actions*) simply as comma-separated lists of sub-statements, and leave the value components either implicit, or to be discussed separately.

5. A Basic Semantics of Multi-threaded Programs

The definitions we use here are similar to those used by others, notably Adve's[1, 3] definition of Data-Race-Free-0.

A *sequentially consistent program execution*[16] or just *program execution* of a program P is an interleaving[6] of finite prefixes of thread executions generated by the statements making up P, such that

- Global variables which are read or tested prior (in the interleaving) to being assigned a value are treated as holding the value zero.[7]
- The value associated with every other load of a global variable is the value associated with the last prior store to that global variable.
- For a given lock l_i, `lock(`l_i`)` and `unlock(`l_i`)` actions must alternate in the program execution, starting with a `lock(`l_i`)` action. For this purpose, r_i `= try_lock(`l_i`)` is treated as `lock(`l_i`)` if the operation succeeds, i.e. if the associated assigned value is zero.
- For a given `lock(`l_i`)`, the next `unlock(`l_i`)` action must be executed by the same thread, i.e. it must correspond to a substatement of the same statement in the program.
- A r_i `= try_lock(`l_i`)` statement succeeds, i.e. has an associated zero value, if and only if there are either no prior operations on l_i, or the last preceding operation on l_i is `unlock(`l_i`)`.

We refer to the first two conditions as *globals consistency*, and the last three as *lock consistency*.

The input read by a program execution is the sequence of values associated with `in` statements in the execution. The output generated by a program is the sequence of values associated with `out` statements in the execution.

Two thread actions *conflict* if they both access the same global variable, at least one of them is a store (the other being either a load or a store), and they are performed by different threads, i.e. the statements in the actions correspond to different threads.

A (sequentially consistent) execution has a *data race* if and only if it contains two adjacent conflicting operations.

A program P has a data race on input I, if it has an execution with a data race which reads input I.

A program P may generate output O on input I if

1. It has a data race on I, or
2. There is an execution of P on I which generates O.

We intentionally allow programs to have any effect whatsoever, i.e. produce any output, for inputs on which they have a data race.

This represents a reasonable interpretation of the pthreads[13] rules. We have interpreted the pthreads restriction on simultaneous access to mean the absence of a data race under a sequentially consistent program execution. The pthreads notion of a "memory location" is taken to mean a single global variable in our simple scenario.

The pthreads statement that locking operations "synchronize memory with respect to other threads" is more problematic, as is pointed out in [10], since it doesn't even clearly prohibit the compiler from introducing reads and writes of unrelated variables between locking calls, which is clearly unacceptable. Hence that statement has been reinterpreted here to mean that data-race-free programs should behave as expected, i.e. as though the execution were sequentially consistent. If anything, this is a stronger restriction than pthreads, and thus allows fewer reorderings. But it will become clear below that our main negative results holds for any reasonable interpretation of the Pthread rules.

Note that, as in the pthreads case, this allows the implementation a large degree of freedom in reordering load and store operations executed between lock operations. Any thread that could observe such reordering would introduce a data race, and would thus render the program semantics undefined.

Although we have made frequent reference to Lamport's definition of sequential consistency, and have used it to define the notion of a data race, our actual semantics are far different from those originally advocated by him.

6. Allowed Reorderings

The central question we now wish to answer is: Under what circumstances can load and store operations be moved into a critical section?[8]

[4] For brevity, we keep this definition informal. We claim that it would be extremely straightforward, and uninteresting, to make it completely precise.

[5] We technically allow infinite thread executions, but the reader may ignore that fact. We will not allow infinite program executions, since they are composed of finite prefixes of *thread* executions.

[6] More formally a sequence consisting of all the actions in each prefix, such that the ordering of the actions in each prefix are preserved

[7] The definition of thread execution enforces a similar rule for registers. Since we assume that register names are distinct among threads, it does not matter whether we impose this requirement on each thread execution or on the program execution.

[8] We will address purely the correctness issues associated with such transformations, with an eye towards reducing fence requirements. Compiler movement of operations into a critical section may aid instruction scheduling, but sometimes also negatively impacts performance and fairness, particularly if adjacent critical sections are combined. The lock may clearly be

As mentioned earlier, we will address this in terms of "compiler" transformations on the source program. In our setting this has the large advantage that we can avoid discussion of the more complicated memory visibility rules that are likely to apply at the hardware level, and reason entirely in terms of sequentially consistent executions and absence of data races.

The following lemma is straightforward, and basically outlines our proof approach:

LEMMA 6.1. *Consider a program transformation T such that every program P is transformed to a program $T(P)$, such that*

1. *T preserves data-race-freedom. More precisely, if $T(P)$ on input I contains a data race, then so does P on input I.*
2. *Whenever $T(P)$ is data-race-free on input I and $E_{T(P)}$ is a sequentially consistent execution of $T(P)$ on I, there is a sequentially consistent execution E_P of P on I that generates the same output.*

Then the transformation preserves meaning, i.e. the transformed program $T(P)$ can generate output O on input I only if the original program P can.

Proof If $T(P)$ has a data race on input I, then so can P on input I. Hence both have undefined semantics, and can generate any output whatsoever.

Now consider the case in which $T(P)$ on I does not have a data race and generates O. There must be a sequentially consistent execution $E_{T(P)}$ which reads I and generates O. Thus there must be an execution E_P of P on I that generates O. Thus the original program could generate the same output. •

We will show that transformations preserve data-race freedom by showing how to map an execution of the transformed program which contains a data race to a corresponding execution of the original program with a data race.

As we stated earlier, it is easy to show via simple examples that transformations which move memory operations out of a region in which a lock is held do not generally preserve meaning. For example, the program section

Thread1: `lock(l1); r1 = v1; unlock(l1);`

is clearly not in general equivalent to

Thread1: `lock(l1); unlock(l1); r1 = v1;`

since the latter introduces a race when run concurrently with

Thread2: `lock(l1); v1 = r1; unlock(l1);`

while the former does not. Thus we must in general both prevent the compiler from performing such movement, and insert memory fences to prevent the hardware from doing so.

Our first goal will be to show that it is unnecessary to prevent movement of memory operations into a critical section past the `unlock()` call.

In order to do so, some lemmas, and a fairly general theorem about allowed compiler reorderings will be helpful:

LEMMA 6.2. *Consider an execution E of P on I, and another sequence of thread actions E' which differs from E only in that two adjacent thread actions have been reordered, and the two actions satisfy the following conditions:*

- *If one of them is a load, expression, or input statement, then the other action is not a load, expression, input, output, or while statement that mentions (i.e. alters or depends on) that register.*
- *If one of them is a store statement, then the other may not be a load or store statement on the same global variable, i.e. the two actions do not conflict.*
- *If one of them is a* `lock`, `unlock`, *or* `trylock` *statement, then the other action is either not a lock operation or applies to a different lock.*

Then the resulting sequence remains register, globals, and lock-consistent, i.e. all values associated with thread actions in E' may remain unchanged. If the exchanged actions correspond to different threads, and the exchanged actions are not both input actions, then E' is also an execution of P on I. If output operations are not exchanged, the output is preserved.

Proof It follows from the first assumption that the resulting sequence is register-consistent, and from the second that it is globals-consistent. Based on the last assumption, we know that the sequence of operations performed on any single lock is unaffected, and thus the resulting sequence is lock-consistent.

If the exchanged operations correspond to different threads, then E' also remains an interleaving of the same thread executions, and hence an execution of P. If no input actions are exchanged, the same input is consumed by corresponding input actions. •

Note that the above lemma deals with reordering actions in executions, not statements in programs. In particular, if we reorder two actions corresponding to the same thread, it is quite possible that the corresponding program reordering, if it even exists, would introduce a data race.

We now consider specifically reordering of actions corresponding to *different* threads:

LEMMA 6.3. *Consider an execution E of P on I, and another sequence of thread actions E' which differs from E only in that two adjacent thread actions corresponding to different threads have been reordered, they are not both input or output actions, and they are also not both* `lock`, `unlock`, *or* `trylock` *statements operating on the same lock. Then either E contains a data race, or E' is also a sequentially consistent program execution of P on I producing identical output.*

Proof Since we are reordering actions of different threads, the first condition of the preceding lemma is satisfied; the two actions cannot mention the same register. Since there is no data race, the second condition of lemma 6.2 must also be satisfied. And we explicitly require the third condition. Since I/O operations are not exchanged, I/O behavior is preserved. Hence the conclusion holds by lemma 6.2. •

By repeated application of the above lemma we can conclude that if we have an execution E of P on I, and P on I allows no data races, then we can repeatedly reorder a non-locking, non-IO action in E with those of other threads, without impacting the validity of the execution.

We are interested in program transformations that reorder statements, specifically locks and memory operations. The following theorem strengthens lemma 6.1 to simplify our proofs for the cases we are interested in:

THEOREM 6.4. *Consider a program transformation T which exchanges the order of a pair of adjacent non-loop statements S_a and S_b, i.e. whenever a program P contains S_a; S_b meeting certain constraints, then the resulting program $T(P)$ will contain S_b; S_a. Assume further that*

1. *T preserves data-race-freedom, i.e. if $T(P)$ on input I contains a data race, then so does P on input I.*
2. *S_a is not a* `lock` *statement. If S_a is an* `unlock` *or* `trylock` *statement, then S_b is not a* `lock`, `unlock`, *or* `trylock` *statement.*

held longer than the programmer intended as a result of such transformations.

3. *Neither S_a nor S_b is an input or output statement.*
4. *T does not exchange two statements that both access the same register or global, unless the accesses are both reads. Since the preceding two conditions handle the lock and I/O constraints, we have effectively ensured that statements are not exchanged is they conflict in the sense of lemma 6.2. (This and the preceding constraints together also imply that the transformation is sequentially correct.)*

Then the transformation preserves meaning.

Proof In order to apply lemma 6.1, we must show that if $T(P)$ is data-race-free on input I and $E_{T(P)}$ is a consistent execution of $T(P)$ on I, then there is a consistent execution E_P on I with the same output. We proceed to construct E_P.

$E_{T(P)}$ will, in general, contain subsequences of actions

$$S_b, E_{other}, S_a$$

corresponding to the transformed code in thread t. Here E_{other} is a possibly empty sequence of intervening actions executed by threads other than the transformed one. (If the execution instead contains an S_b without the corresponding S_a, we can append S_a while preserving consistency of the execution, returning us back to this case. This requires the assumption that S_a is not a `lock` or I/O statement, and hence cannot be blocked from executing, or affect I/O behavior.)

Assume temporarily that S_b is not a locking statement (`lock`, `unlock`, or `trylock`). By lemma 6.3, the execution obtained by replacing each such instance by

$$E_{other}, S_b, S_a$$

is another consistent execution of $T(P)$. (We've excluded the problematic case in which S_b is a locking or I/O action and is exchanged with another like action.) By lemma 6.2, if we further replace each such sequence by

$$E_{other}, S_a, S_b$$

the result remains register, globals, and lock consistent. And the actions in this sequence corresponding to thread t clearly correspond to the untransformed program P. Hence this sequence can serve as the desired execution E_P.

This leaves the case in which S_b is a locking statement. In that case we know, based on our assumptions, that S_a is not. In that case, we instead replace the sequences

$$S_b, E_{other}, S_a$$

in $E_{T(P)}$ by

$$S_b, S_a, E_{other}$$

and then

$$S_a, S_b, E_{other}$$

to obtain E_P. The remainder of the argument remains unchanged. ●

Note that in addition to reducing our proof obligations in the rest of this section, and partially justifying out intuition that compiler transformations which preserve sequential correctness and do not introduce data races are generally safe, this more precisely justifies a number of standard compiler transformations that are normally performed in a pthreads environment. (As Boehm points out in [10], not all commonly performed compiler transformations are actually safe in a pthreads environment, so this result is not completely trivial.)

We now focus on the specific transformations of interest.

THEOREM 6.5. *A transformation which alters the input program P only by reordering a program section*

$$\texttt{unlock(}l\texttt{);}\ memop;$$

to

$$memop;\ \texttt{unlock(}l\texttt{);}$$

where $memop$ is a load or store statement, preserves meaning.

Proof Clearly all but the first requirement of theorem 6.4 are satisfied.

We now show that T preserves data-race-freedom. Assume $T(P)$ on the given inputs allows a data race. Let $E_{T(P)}$ be a shortest execution exhibiting this data race, subject to the constraint that it includes the `unlock(`l`)` if it includes the $memop$ from a transformed code section. (Note that `unlock(`l`)` can always be appended to an execution that includes $memop$.)

To help in the construction, we first define E_P as in theorem 6.4. Wherever $E_{T(P)}$ contained actions

$$memop, E_{other}, \texttt{unlock(}l\texttt{)}$$

of the transformed code, we substitute

$$E_{other}, \texttt{unlock(}l\texttt{)}, memop$$

As we showed in theorem 6.4, the result is a consistent execution of P on the same input. We now show how to transform E_P into an execution E'_P that preserves the data race in $E_T(P)$

If the data race in $E_{T(P)}$ does not involve the transformed $memop$, then E_P preserves the race, and we're done. Thus we may assume that $E_{T(P)}$ ends with the `unlock(`l`)` of a transformed section of code and the race involves $memop$ in the trailing sequence

$$prev\_action, memop, E_{other}, \texttt{unlock(}l\texttt{)}$$

Assume again that the thread executing $memop$ and `unlock(`l`)` is t.

If there is a race between $prev\_action$ and $memop$, we instead let E'_P be E_P, but with the final

$$prev\_action, memop, E_{other}, \texttt{unlock(}l\texttt{);}$$

in $E_{T(P)}$ instead replaced by

$$\texttt{unlock(}l\texttt{)}, prev\_action, memop$$

This corresponds to first removing the actions in E_{other} and then performing the `unlock(`l`)` action earlier. The first step generates another valid execution of $T(P)$, since the actions in E_{other} are the final ones for their respective threads in $E_{T(P)}$, and their removal cannot violate lock-consistency. By lemma 6.2, and the fact that $prev\_action$ and $memop$ must be memory operations, performing the `unlock(`l`)` earlier leaves the action sequence consistent, and the original values associated with the actions remain valid.[9]

Since the resulting sequence contains the actions of t in an order consistent with the original P, and the other threads' actions were not reordered, this gives us an execution of P that preserves the data race.

Hence E'_P is an execution of P that exhibits the same race as $E_{T(P)}$.

If the first race is between $memop$ and the first operation in E_{other}, a similar argument applies for the execution of P ending in

[9] The same could not be said without dropping E_{other}, since it may contain `trylock(`l`)` calls. That does not matter, since we only have to exhibit *an* execution with a data race.

$$prev\_action, \texttt{unlock}(l), memop, first\_action\_of\_E_{other}$$

Thus a race in $T(P)$ always maps back to a race in P on the same input, and hence data-race-freedom is preserved by T. •

A similar result holds for moving memory operations into the locked region past the initial `lock`, but only in the absence of `trylock`:

THEOREM 6.6. *A transformation which alters the input program P only by reordering a program section*

$$memop; \texttt{lock}(l);$$

to

$$\texttt{lock}(l); memop;$$

where memop is a load or store statement, and only when there are no occurrences of `trylock(`l`)` *in the program, preserves meaning.*

Proof Again the only nontrivial proof obligation remaining from theorem 6.4 is that data-race-freedom is preserved.

Again consider a shortest execution $E_{T(P)}$ of $T(P)$ on I containing a data race If the race does not involve $memop$, then it is present in the execution E_P of P, constructed as in theorem 6.4 and we are done.

If $memop$ conflicts with the next action in $E_{T(P)}$, then $E_{T(P)}$, since it is minimal, must end in

$$\texttt{lock}(l), E_{other}, memop, next\_action$$

which would be transformed to

$$memop, \texttt{lock}(l), E_{other}, next\_action$$

by the construction of E_P in theorem 6.4.

The subsequence E_{other} consists entirely of actions performed by other threads. If it contained `lock` or `unlock` operations on l, the sequence could not be lock consistent. We assumed that it does not contain `trylock` actions on l. We know that $next\_action$ is a load or store. Hence by lemma 6.2, we obtain an equivalent execution of P by moving the `lock` action to the end to obtain

$$memop, E_{other}, next\_action, \texttt{lock}(l)$$

If $memop$ conflicted with an action in E_{other}, there would have been a shorter execution with a data race. Hence by applying lemma 6.2 once more, we see that this is equivalent to

$$E_{other}, memop, next\_action, \texttt{lock}(l)$$

which exhibits the race in an execution of P on the same input.

This leaves the case in which $memop$ is the second operation involved in the race. By lemma 6.2 that is equivalent to the execution ending in

$$\texttt{lock}(l), E_{other}\_except\_last, memop, last\_of\_E_{other}$$

which gets us back to the first case. •

7. Disallowed Reorderings

In Theorem 6.6, we assumed that there are no `trylock` operations on the lock in question. We now show that this assumption is in fact essential, and the theorem does not hold without this assumption:

THEOREM 7.1. *There exist programs involving* `try_lock` *for which the above is not safe.*

Proof Recalling that our version of `try_lock` returns a nonzero value on *success*, i.e. if it acquires the lock, consider the following program:

```
T1: v1 = 1; lock(l1);

T2: r1 = try_lock(l1);
    while (r1 /* was unlocked */) {
        unlock(l1); r1 = try_lock(l1);
    }
    r2 = v1; out(r2);
```

Although few would defend this as good, or even reasonable, programming style, it is data-race-free. `T2` can only read `v1` after the loop terminates. This can only happen once `T1` acquires the lock. Hence this program has sequentially consistent semantics, and therefore `r2` and the output are guaranteed to be 1.[10]

Now consider this program after it has been transformed as in Theorem 6.6. Thus we now have for the first thread:

```
T1: lock(l); v1 = 1;
```

The resulting program clearly has a data race, and hence completely undefined semantics.[11] •

Note that the same applies if the loads and stores of `v1` are interchanged, so that we are transforming

```
T1: r2 = v1; lock(l1); out(r2);
```

to

```
T1: lock(l1); r2 = v1; out(r2);
```

while the second thread assigns 1 to `v1` after waiting for the lock to be acquired by the first thread.

With our semantics the same race would be introduced.[12]

Note that this transformation clearly fails to preserve the meaning of this example under any reasonable interpretation of Pthread rules. Unlike our positive results it does not rely on a very specific interpretation of those rules.

8. Related Work

Much has been published about hardware memory consistency models (cf. [2, 12]). Probably the closest to our work in that arena is the development of the Data-Race-Free-0 synchronization model by Adve and Hill[1, 3], which we previously mentioned.

However that work addresses the issue at a lower level. By viewing synchronization operations as generic memory operations, and not as locking operations that provide very restricted kinds of access to special lock locations, the kinds of issues discussed here do not arise.

Another thread of work has focused on compiler optimizations that allow removal of memory fences while preserving a sequentially consistent programming model for the programmer, a much stronger guarantee than is made by other us or pthreads. See for example [26], [15] and [18]. This work addresses compiler analysis techniques to minimize memory fences in specific programs, but it does not address the questions addressed here, dealing with cases in which reordering is always safe, or fences are needed in generic library routines.

The closest related work at the programming language level appears to be that by Manson, Pugh, and Adve on the Java memory

[10] Posix does not guarantee memory "synchronization" when a function fails, as the final `try_lock` call does. But the example could easily be made Posix conforming by adding a `lock(l2); unlock(l2);` immediately after the `while loop`, where `l2` is otherwise unused.

[11] Even if we guaranteed sequentially consistent execution, the transformed program would clearly still not preserve meaning, since it would allow the case in which the output is zero.

[12] With sequentially consistent semantics, this would again allow an output of zero, where the original does not.

model.[24, 21, 20]. Among other things, they prove the safety of a large set of compiler reordering transformations under the Java model. (See theorem 1 of [21].) However, the model and proof methodology are different. In particular, the Java model allows reordering of memory operations with monitor entry, in contradiction to our Theorem 7.1.

It is important to note that this s caused by Java's notion of a data race being different from ours, and does not correspond just to the simulataneous execution of two conflicting operations. The equivalent of the example used in the proof of Theorem 7.1 in fact has a Java data race,[13] and hence the programmer is not allowed to expect sequentially consistent semantics.

9. Conclusions

We have shown that if we take the pthreads specification at face value (except for necessary adjustments to prohibit compiler-introduced races), high performance implementations must exhibit an asymmetry:

- In the general case, they cannot allow any reordering of memory operations across `pthread_mutex_lock()` or `pthread_spin_lock()`. This will generally imply that on architectures on which atomic memory updates do not completely inhibit reordering (e.g. Itanium, Alpha, PowerPC, see [17]), a suitable memory fence (affecting both loads and stores) is needed.
- On the other hand, `pthread_mutex_unlock()` and `pthread_spin_unlock()` do not require such conservative treatment. On architectures that do not allow memory operations to be reordered with a following store (e.g. most X86 processors), it is acceptable to implement `pthread_spin_unlock()` with an ordinary store instruction, as is often done.

It is completely unclear to us whether this was intended or an accident. It comes about as a result of programs that essentially invert the sense of a lock, by using `trylock` as in the proof of Theorem 7.1. There are strong arguments for discouraging such programs. Supporting them by significantly penalizing `pthread_mutex_lock()`, which tends to be both pervasive and performance-critical in multi-threaded code, in order to support this idiom, appears to be a questionable trade-off.

If we do want to support the existing specification, this raises the possibility of link-time optimizations to take advantage of the absence of `trylock` in many programs.

As it stands, this asymmetry cooperates well with spin lock implementations on processors with atomic exchange or compare-and-swap instructions which include a full memory fence: The fence is needed exactly where it is implicitly included, and not required where it would be expensive. It works less well on processors for which atomic memory updates do not completely enforce memory ordering.

A. Cost of Fences in Locks: Measurements

In order to illustrate the performance impact of memory ordering constraints on locking, we measured a very lock-intensive program, with a variety of lock implementations.

Our test program is one that copies 10 million characters from one file to another, one character at a time, with different types of locking used to control per-character access to the I/O buffers.

[13] There is no "release" operation in the example, and hence there can be no "happens before" ordering between the store and the load, even though there are intervening synchronization operations.

This clearly results in more frequent lock acquisition and release than anyone would like to see in production code though, in our experience, something along these lines is sometimes used in production code.[14] However, since the example is so lock intensive, it provides a convenient, and perhaps not completely contrived, way to assess the performance impact of memory ordering constraints.

In order to minimize time spent in the OS kernel and on disk accesses, we copy from the `/dev/zero` pseudo-file on a Linux system to a file on `/tmp`, on machines with enough memory that no actual disk access was required. We use the default stdio buffer size of 16K. The resulting program is cpu-bound, with little time (always $< 20\%$, much less with locking) spent in the OS kernel.

We measured 8 variants of this simple program. The first four of these are included to allow the reader to calibrate our results. The latter four demonstrate the impact of allowing no reordering of memory operations across synchronization primitives.

Unsafe The file copy is implemented using the Posix `getc_unlocked()` and `putc_unlocked()` primitives. The result is not thread safe. Attempting to run multiple copies of this program concurrently on a multiprocessor is extremely unlikely to result in a file of the right length, and, unlike the other variants, running multiple copies often results in a segmentation faults.

Default The file copy is implemented using the Posix `getc()` and `putc()` calls. These each acquire and release a lock.

Mutex We call `getc_unlocked()` and `putc_unlocked()`, but explicitly ensure thread safety by acquiring and releasing a Pthread mutex around each of them. We use separate locks for the input and output files. This is similar to what `getc()` and `putc()` use internally.

Spin Identical to the preceding one, except that we use Pthread spin locks instead of mutexes. In the contention-free case, these generally perform better, largely because the lock release can be implemented more cheaply. However they tend to be less robust in the presence of contention.[15]

None Identical to "Spin", except that we use custom spin locks implemented with our atomic operations library[9]. We add no extra fences. Memory operations are only constrained to not move out of the critical section. This appears to suffice for all reasonably designed code. It corresponds to the guarantees provided for Java locks[14], for example.

Lock Identical to "None", except that we add a full memory fence to the lock implementation.

Unlock Identical to "None", except that we add a full memory fence to the unlock implementation.

Both Identical to "None", except that we add a full memory fence to both the lock and unlock implementation. Thus the hardware is also prevented from moving any memory operations *into* the critical section.

We use gcc as the compiler in all cases. It does not move memory references past locks. However, we can adjust the memory fences used by the lock implementation.

All experiments were done on Linux machines using a recent NPTL threads implementation and a 2.6.12 kernel. We give results for two different machines:

[14] In particular, the "Default" variant below is arguably the most natural way to copy a file in Posix code.

[15] In our experiments, the mutex-based solutions were generally better and much more predictably in very high contention situations, as expected.

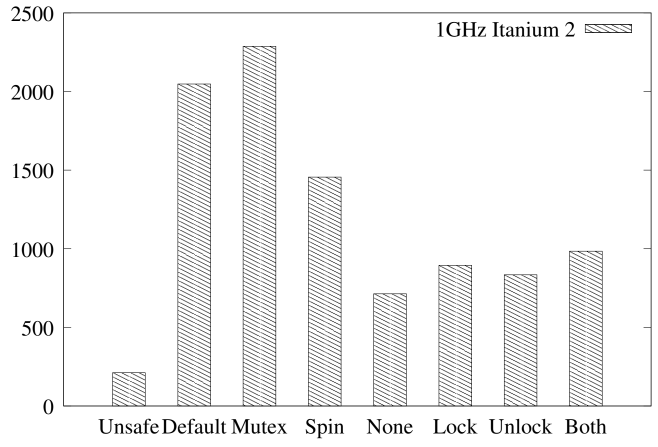

Figure 1. Msecs to copy 10MB on 1GHz Itanium 2

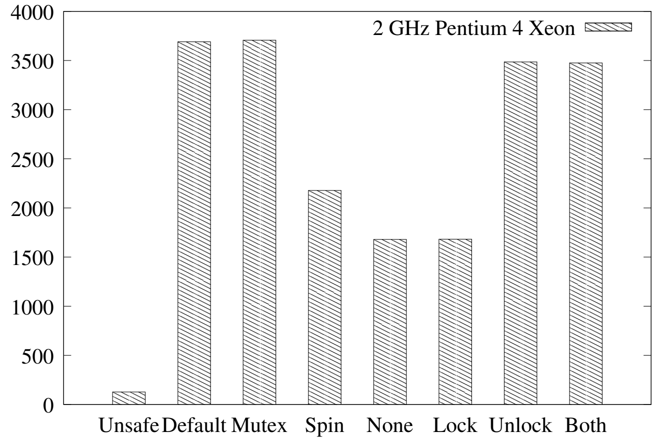

Figure 2. Msecs to copy 10MB on 2GHz Pentium 4 Xeon

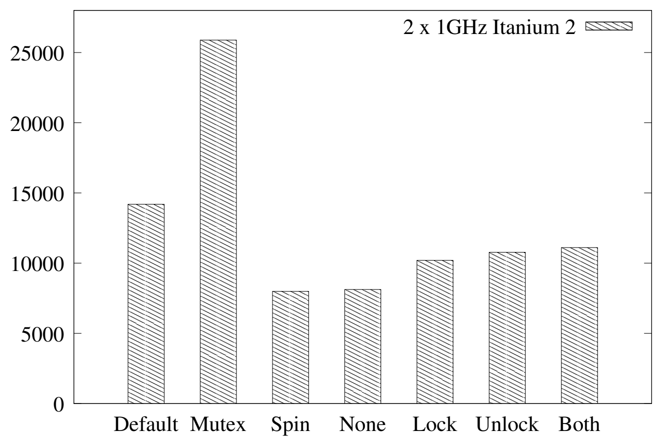

Figure 3. Msecs to copy 10MB/thread in 2 threads on 1GHz Itanium 2

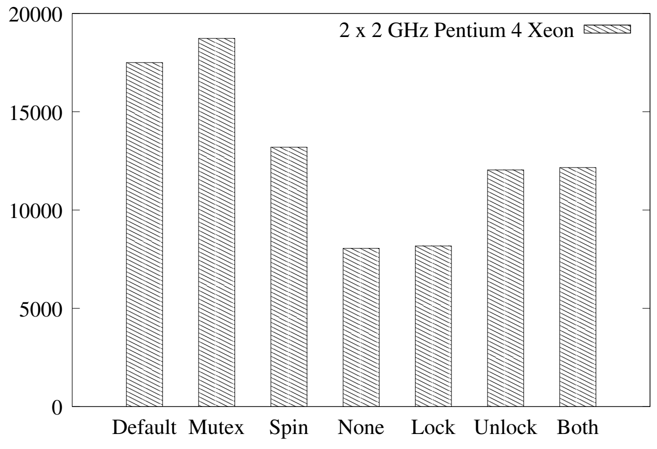

Figure 4. Msecs to copy 10MB/thread in 2 threads on 2GHz Pentium 4 Xeon

Pentium 4 Xeon The results are presented in figure 2 and figure 4. This is a 2 GHz dual processor "hyperthreaded" machine, on which both atomic memory update instructions (used here for `pthread_mutex_lock()`, `pthread_mutex_unlock()` and `pthread_spin_lock()`, but not `pthread_spin_unlock()`) and fences are expensive, and hence dominate the runtime. The atomic memory update operations also act as a fence, and hence adding a fence requirement only affects the unlock operation. It would not affect the time required for mutex operations[16], but it vastly increases the cost of releasing a spin-lock, thus roughly doubling the cost of the file copy with spin-locks.[17]

Itanium 2 The results are presented in figure 1 and figure 3. This is a 1 GHz four processor machine. Both atomic memory updates and memory fences are much cheaper. Atomic memory updates never include a full memory fence, and thus we need to add instructions to prevent reordering of memory operations around these instructions, and hence around locks. However, the time required to execute the memory fence instructions is highly context dependent.

We report the times required for a single thread to perform the copy in a multi-threaded context, i.e. when run from a thread other than the main one. This is representative of the (hopefully typical) low contention case. The reported measurements are averages of three runs, but variations are very low.

For completeness, we give one set of measurements for two threads performing the same copy operation concurrently. However the exact contention level here seems very dependent on accidents of compilation, and we did not find these very reproducible over time.

Note that the scale on the y-axis varies appreciably across the different graphs.

In all cases, we see that added fences have a substantial impact on the performance of at least some locking primitives. In the X86 case, the effect is limited to spin locks, since mutexes implicitly

[16] Some platforms avoid the atomic memory update for an `unlock` operation, even if a queue is involved. This gives rise to a rare lost wakeup, from which it is possible to recover with a timeout, at some cost. If `pthread_mutex_unlock()` is implemented this way, its performance would be closer to `pthread_spin_unlock()` on our platform.

[17] Note that some X86 machines, including some from the same vendor, provide much less expensive atomic update and fence primitives.

provide the fence semantics. On Itanium, we expect that the effect on mutexes is similar.[18]

Acknowledgments

This question arose out of discussions with many others, including Bill Pugh, Doug Lea, Peter Dimov, Alexander Terekhov, and David Butenhof, among others, as a result of our effort to specify semantics for multi-threaded C++. Jeremy Manson explained to me why the Java memory model can claim sequentially consistent semantics for race-free programs, even though Java now also supports a "trylock" primitive. The use of file copy as a simple lock benchmark was suggested by a mailing list discussion with Nick Maclaren. David Dice, Doug Lea, and David Butenhof and several anonymous reviewers contributed to substantial improvements over earlier drafts.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, 1993.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[3] S. V. Adve and M. D. Hill. Weak ordering—A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA'90)*, pages 2–14, 1990.

[4] A. Alexandrescu, H.-J. Boehm, K. Henney, B. Hutchings, D. Lea, and B. Pugh. Memory model for multithreaded C++: Issues. C++ standards committee paper WG21/N1777, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1777.pdf`, March 2005.

[5] A. Alexandrescu, H.-J. Boehm, K. Henney, D. Lea, and B. Pugh. Memory model for multithreaded C++. C++ standards committee paper WG21/N1680, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2004/n1680.pdf`, September 2004.

[6] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: Featherweight synchronization for java. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 258–268, 1998.

[7] H. Boehm, D. Lea, and B. Pugh. Memory model for multithreaded C++: August 2005 status update. C++ standards committee paper WG21/N1876, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1876.pdf`, September 2005.

[8] H.-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, HP Laboratories, December 2000.

[9] H.-J. Boehm. The atomic_ops atomic operations package. `http://www.hpl.hp.com/research/linux/atomic_ops/`, 2005.

[10] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 261–268, 2005.

[11] G. Colvin, B. Dawes, D. Adler, and P. Dimov. The boost shared_ptr class template. `http://www.boost.org/libs/smart_ptr/shared_ptr.htm`.

[12] K. Gharachorloo. Retrospective: memory consistency and event ordering in scalable shared-memory multiprocessors. *International Conference on Computer Architecture, 25 years of the international symposia on Computer architecture (selected papers)*, pages 67–70, 1998.

[13] IEEE and The Open Group. *IEEE Standard 1003.1-2001*. IEEE, 2001.

[14] JSR 133 Expert Group. Jsr-133: Java memory model and thread specification. `http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf`, August 2004.

[15] A. Krishnamurthy and K. A. Yelick. Optimizing parallel programs with explicit synchronization. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–204, 1995.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

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

[18] J. Lee, D. A. Padua, and S. P. Midkiff. Basic compiler alogrithem for parallel programs. In *Principles and Practice of Parallel Programming*, pages 1–12, 1999.

[19] G. libstc++ developers. Gnu standard c++ library: libstdc++-v3. `http://gcc.gnu.org/viewcvs/tags/gcc_4_1_0_release/libstdc++-v3`.

[20] J. Manson, W. Pugh, and S. Adve. The java memory model (expanded version). `http://www.cs.umd.edu/users/jmanson/java/journal.pdf`.

[21] J. Manson, W. Pugh, and S. Adve. The java memory model. In *Conference Record of the Thirty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 378–391, January 2005.

[22] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Engineering at Oregon Health and Science University, 2004.

[23] C. Nelson and H. Boehm. Sequencing and the concurrency memory model. C++ standards committee paper WG21/N2052, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2052.htm`, September 2006.

[24] B. Pugh. The java memory model. `http://www.cs.umd.edu/~pugh/java/memoryModel/`.

[25] M. L. Scott and W. N. S. III. Scalable queue-based spin locks with timeout. In *Principles and Practice of Parallel Programming (PPOPP)*, pages 44–52, 2001.

[26] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1998.

[27] The Open Group and IEEE. The single unix specification, version 3 (ieee standard 1003.1-2001). `http://unix.org/version3/`.

[18] Recall that the standard NPTL `pthread_spin_unlock` implementation on Itanium contains a redundant fence instruction, which may explain its relatively poor performance in the low contention case.