# Reordering Constraints for Pthread-Style Locks

Hans-J. Boehm
HP Laboratories Palo Alto
HPL-2005-217
November 29, 2005*

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[8], it is not terribly 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. Although our precise arguments are limited to a small subset language, we believe that our results capture the situation for a full C/C++ implementation, together with a literal (and reasonable, though possibly unintended) interpretation of the pthread standard. And they appear to have implications for other environments as well.

The results appear to be surprising, and to not have been anticipated by pthread implementors, in spite of their significant performance impact on multi-threaded applications.

* Internal Accession Date Only
Approved for External Publication

# 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[8], it is not terribly 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. Although our precise arguments are limited to a small subset language, we believe that our results capture the situation for a full C/C++ implementation, together with a literal (and reasonable, though possibly unintended) interpretation of the pthread standard. And they appear to have implications for other environments as well.

The results appear to be surprising, and to not have been anticipated by pthread implementors, in spite of their significant performance impact on multi-threaded applications.

***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 the application. Multiple threads are also often used for structuring even uniprocessor applications to make it easier to deal with multiple event streams.

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[10] specification. We believe that much of the discussion here is applicable to other libraries, and even some completely different 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 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 contain memory fences (sometimes called *barriers*) to restrict reordering of memory operations by the hardware. These implementations also ensure that these functions are treated as sufficiently opaque in the compiler, so that the compiler does not move memory references across calls in inappropriate ways. Compilers otherwise compile code as if it targeting a single-threaded environment.

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

This approach is fundamentally different from that used in Java[17]. 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[6].

As is pointed out in [8], 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 develop-

ment of a C++ memory model interesting[5, 4, 6]. 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.

We ask the 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 ask 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.

We address this issue by looking at allowable compile-time transformations of the source programs, mostly because it is easiest to reason at that level. Nonetheless, we expect that the greatest actual performance impact of this issue is on the library implementation of the locking primitives, primarily because memory fence instructions are expensive on many current architectures.

In the next section, and in the appendix, we argue that the performance impact of this issue on real applications can be large.

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.

We expect that a number of current implementations either do not follow these rules, or add extra fences.[1] 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 particularly bad cases:

- Program execution cost can be dominated by locking. One example of such a program is given in [8]. Unfortunately, other examples are regularly encountered in practice, though often as a result of poor programming practices.
- The locking cost can be primarily 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.

We present an example of all of these, together with some simple measurements, in appendix A.

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

## 3. 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]:

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

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.

[1] This is based in part on discussions with the developers and implementors of the standard, and in part on examination of the Linux/Itanium implementation.

[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.

A *program* is a finite sequence of statements. Informally, the $i$th statement describes the actions executed by the $i$th thread.[3] We will assume that the register values 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.

A statement generates sets of possible *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, such that values assigned by store statements, written by output statements, computed by expression evaluation statements, or tested by loops, correspond to the values assigned to the input registers by the last prior assignments in that thread execution, if there are such prior assignments, and any possible input values if there are not.[4]

Hence the statement $r_2$ `=` $r_1$`;` $r_3$ `=` `2` `*` $r_2$ generates all thread executions of the form

$$(r_2 \texttt{ = } r_1\texttt{, } x)\texttt{, } (r_3 \texttt{ = 2 * } r_2\texttt{, } 2x)$$

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.

We refer to the property that retrieved register values match assigned register values as *register-consistency*.

Similarly, the 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, the last is zero, and again values read from registers are consistent with those written by the execution.

## 4. 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*[13] or just *program execution* of a program $P$ is an interleaving[5] of finite prefixes of thread executions generated by the statements making up $P$, such that

- Registers or global variables which are read or tested prior (in the interleaving) to being assigned a value are treated as holding the value zero. (In the case of registers, this is equivalent to insisting that this holds for each thread execution.)
- 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 immediately following `unlock(`$l_i$`)` action must be executed by the same thread, i.e. it must correspond to a sub-statement of the same statement in the program.
- A $r_i$ `= try_lock(`$l_i$`)` value 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 (with the first restricted to globals) 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 (implying the other must be 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[10] 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 [8], 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 advocated by him.

[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.

[4] 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.

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

## 5. 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?[6]

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 5.1. *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.

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

In order to do so, the following lemma will be helpful:

[6] We will adress purely the correctness issues associated with such transformations. Moving operations into a critical section sometimes also negatively impacts performance and fairness, particularly if adjacenet critical sections are combined. The lock may clearly be held longer than the programmer intended as a result of such transformations.

LEMMA 5.2. *Recall that thread actions consist of pairs, the second component of which is the value assigned, read, written, or tested.*

*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, then $E'$ is also an execution of $P$ on $I$.*

**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 $P$ on $I$. •

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.

THEOREM 5.3. *A transformation which alters the input program $P$ only by reordering a program section*

`unlock(`$l$`);` $memop$`;`

*to*

$memop$`; unlock(`$l$`);`

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

**Proof**

Consider a sequentially consistent execution $E_{T(P)}$ of $T(P)$ on input $I$. Assume that whenever it contains an instance of $memop$ that was generated by the above transformation, it also contains the corresponding `unlock(`$l$`)`. (Any execution violating this constraint can be extended to one that satisfies it simply by adding the `unlock` action.)

We define a corresponding execution $E'_P$ of the original $P$, which will help with the rest of the proof, as follows:

$E_{T(P)}$ contains some number of subsequences of the form

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

corresponding to the moved $memop$, where $memop$ and `lock(`$l$`)` are performed by the same thread $t$, and $E_{other}$ represents the actions performed by other threads between the two operations.

Note that if $E_{other}$ contains an action that conflicts with $memop$, then by repeated application of lemma 5.2 the action sequence $E'_{T(P)}$ constructed from $E_{T(P)}$ by moving $memop$ to just before the first conflicting access is also a valid execution of $T(P)$ on $I$. (The condition on registers is automatically satisfied since we

are exchanging actions of different threads.) Clearly $E'_{T(P)}$ contains a data race, and consequently $T(P)$ allows a data race. Thus such a conflict is impossible in the absence of a data race for $T(P)$ on $I$.[7]

We define $E'_P$ to be $E_{T(P)}$ with each such subsequence replaced by

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

If $T(P)$ on $I$ does not allow a data race then, by the above observation, no action in $E_{other}$ can conflict with $memop$. Since $E'_P$ can be obtained from $E_{T(P)}$ by repeatedly exchanging $memop$ with an adjacent action, by lemma 5.2 the resulting action sequence is register-, globals-, and lock-consistent. The actions of the transformed thread $t$ appear in $E'_P$ in an order that is generated by $P$, since only the actions corresponding to the exchanged statements are changed. The order of actions performed by other threads is unchanged, as is the corresponding program. Hence $E'_P$ is a valid execution of $P$ with the same input and output as $E_{T(P)}$, and we have satisfied the second condition of lemma 5.1.

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 the shortest execution exhibiting this data race, but again including the `unlock(`$l$`)` if it includes the $memop$ from a transformed code section. (Note that this is likely to be an "incomplete" execution.)

Let $E'_P$ be as defined above.

There are two cases:

1. $E_{T(P)}$ ends with the `unlock(`$l$`)` of a transformed section of code and the race is contained in a sequence

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

   If the race is contained entirely in $E_{other}$, then $E'_P$ is an execution of $P$ that preserves the race.

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

   If there is a race between $prev\_action$ and $memop$, we instead consider $E''_P$, in which the original

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

   in $E_{T(P)}$ has been 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 5.2, 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.[8]

   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 ending in

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

2. $E_{T(P)}$ ends with an unrelated race. In that case, $E'_P$ already preserves the race.

Thus races are always 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 5.4. *A transformation which alters the input program $P$ only by reordering a program section*

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

*to*

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

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

**Proof**

We apply an argument similar to the above. Given an execution $E_{T(P)}$ of $T(P)$ on $I$, we again define a corresponding execution $E'_P$ of $P$, which has the same behavior in the race-free case, and is used as a basis for the proof that race-freedom is preserved.

The execution $E_{T(P)}$ contains some number of subsequences

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

corresponding to the transformed code, where `lock(`$l$`)` and $memop$ are executed by thread $t$ and the actions in $E_{other}$ correspond to other threads. Define $E'_P$ to be $E_{T(P)}$ with each such sequence replaced by

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

This remains register-consistent, since no operations on registers or globals were moved in the action sequence.

It also remains lock-consistent:

- If $E_{other}$ contained a `lock` or `unlock` operation on the same lock, the original execution would not have been lock-consistent.
- We have assumed that there are no `trylock` operations on the same lock.[9]

Clearly $E'_P$ is a valid execution of $P$ with the same input-output behavior as $E_{T(P)}$.

It remains to be shown that data-race-freedom is preserved.

Again consider a shortest execution $E_{T(P)}$ of $T(P)$ on $I$, subject to the constraint that if the execution contains the transformed $memop$, then it also contains the corresponding lock. If the race either does not involve $memop$, or involves the previously executed action, then it is present in the execution $E'_P$ of $P$, constructed as above, and we are done.

If $memop$ conflicts with the next action $next\_action$ in $E_{T(P)}$, then we instead consider the execution $E''_P$ of $P$ in which

[7] It is also worth noting that if we allowed the transformed code to be executed by multiple threads, the relevant subsequences of $E_{T(P)}$ cannot overlap, since that would correspond to $l$ being held by multiple threads at the same time. Thus the arguments would still apply.

[8] The same could not be said if we had not dropped $E_{other}$, since it may contain `trylock(`$l$`)` calls. That does not matter, since we only have to exhibit *an* execution with a data race.

[9] If we just moved $memop$ to the beginning of the new execution, this part of the proof would go through even with `trylock`. But $E'_P$ would be less useful in reasoning about preservation of data-race-freedom, and that argument would only go through in the absence of `trylock`.

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

is replaced by

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

Since $next\_action$ must be a load or a store executed by a different thread, this remains valid for the same reasons as above, and preserves the race. •

## 6. Disallowed Reorderings

In Theorem 5.4, 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 6.1. *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` iand the output are guaranteed to be 1.[10]

Now consider this program after it has been transformed as in Theorem 5.4. 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 really rely on very precise semantics.

[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.

## 7. Related Work

Much has been published about hardware memory consistency models (cf. [2, 9]). 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 [19], [12] and [15]. 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 model.[18, 17, 16]. Among other things, they prove the safety of a large set of compiler reordering transformations under the Java model. (See theorem 1 of [17].) 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 6.1. The cause is that Java's notion of a data race is different from ours, and the equivalent of the example used in the proof of Theorem 6.1 in fact has a has a Java data race,[13] and hence the programmer is not allowed to expect sequentially consistent semantics.

## 8. 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 [14]), 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 6.1. 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.

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

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. 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.

Our test program is one that copies 10 million characters from one file to another, one character at a time. 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. The resulting program is cpu-bound, with little time 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.[14]

**None** Identical to "Spin", except that we use custom spin locks implemented with our atomic operations library[7]. 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[11], 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.

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

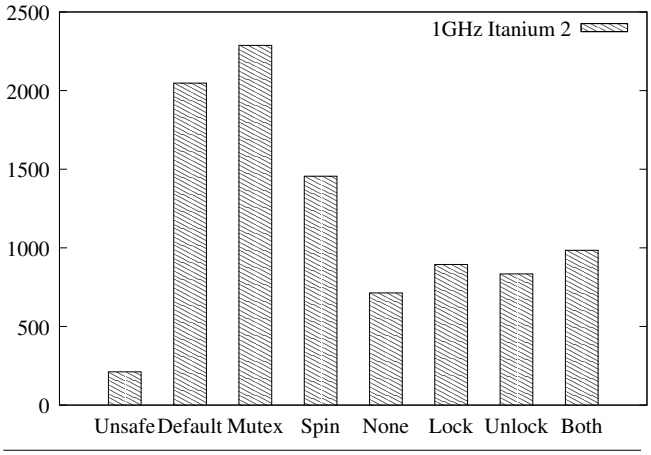


**Figure 1.** Msecs to copy 10MB on 1GHz Itanium 2

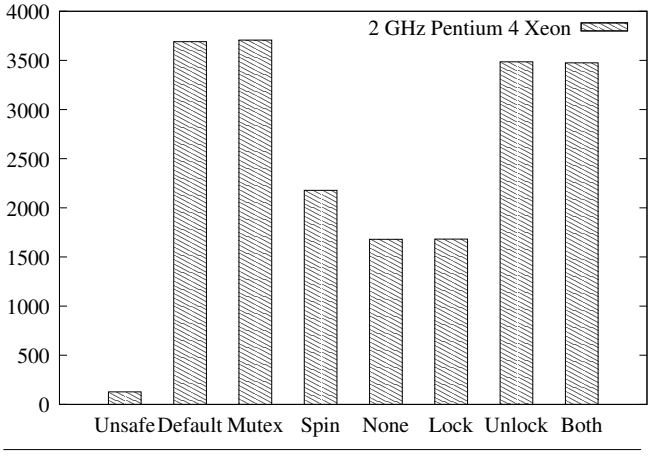


**Figure 2.** Msecs to copy 10MB on 2GHz Pentium 4 Xeon

**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.

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:

**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 (required 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, but it vastly increases the cost of releasing a spin-lock, thus roughly doubling the cost of the file copy with spin-locks.[15]

**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

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

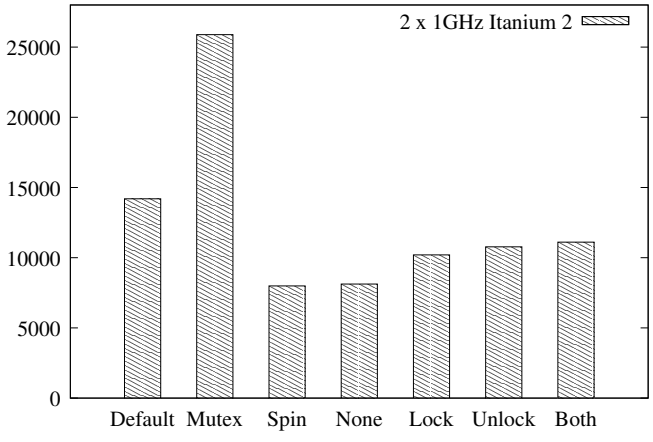


**Figure 3.** Msecs to copy 10MB/thread in 2 threads on 1GHz Itanium 2

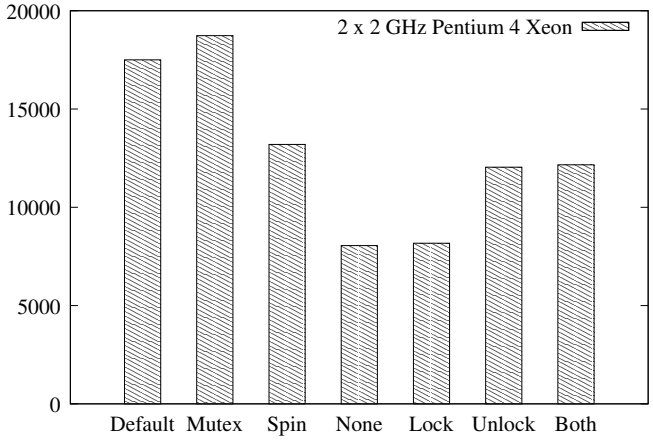


**Figure 4.** Msecs to copy 10MB/thread in 2 threads on 2GHz Pentium 4 Xeon

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.

We also give measurements for two threads performing the same copy operation concurrently. Both our test machines had at least two processors, so this results in more lock and cache-line contention than one would hope for in a real application. The exact contention level is probably dependent on instruction timing and accidents of compilation. As a result we also observed significant variation in the measurements between runs, exceeding 10 percent in one case, but less for the last four variants.

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 provide the fence semantics. On Itanium, we expect that the effect on mutexes is similar.[16]

## 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. Doug Lea and David Butenhof provided useful feedback on 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. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1777.pdf`.

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

[6] H. Boehm, D. Lea, and B. Pugh. Memory model for multithreaded C++: August 2005 status update. `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1777.pdf`.

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

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

[16] 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. It's unclear why the difference disappears in the high contention case.

[15] 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.

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

[17] 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.

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

[19] 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.