Threads Cannot Be Implemented As a Library

Hans-J. Boehm
HP Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
Hans.Boehm@hp.com

ABSTRACT

In many environments, multi-threaded code is written in a language that was originally designed without thread support (e.g. C), to which a library of threading primitives was subsequently added. There appears to be a general understanding that this is not the right approach. We provide specific arguments that a pure library approach, in which the compiler is designed independently of threading issues, cannot guarantee correctness of the resulting code.

We first review why the approach almost works, and then examine some of the surprising behavior it may entail. We further illustrate that there are very simple cases in which a pure library-based approach seems incapable of expressing an efficient parallel algorithm.

Our discussion takes place in the context of C with Pthreads, since it is commonly used, reasonably well specified, and does not attempt to ensure type-safety, which would entail even stronger constraints. The issues we raise are not specific to that context.

1. INTRODUCTION

Multi-threaded programs are rapidly becoming pervasive, driven primarily by two considerations:

- Many programs need to carry on several different logically concurrent interactions. For example, they may need to concurrently serve several different client programs, or provide several related services which can progress asynchronously, usually in separate windows, to a single user. Threads provide a clean structuring mechanism for such programs.

- Multiprocessors are finally becoming mainstream. Many of the most popular processors in desktop computers support multiple hardware contexts in a single processor, making them logically multiprocessors. In addition, essentially every microprocessor manufacturer who is not already shipping chip-level multiprocessors has announced the intention to do so within a year. This even applies to some intended for embedded applications, where multiprocessors can provide more compute throughput with less total power consumption (cf. [3]).

  In many cases, there is no way to effectively utilize the performance of the additional processor cores or hardware threads without writing explicitly multi-threaded programs.[1]

Most multi-threaded programs communicate through memory shared between the threads. Many such programs are written in a language such as Java, C#, or Ada, which provides threads as part of the language specification. Recent experience has shown that it is quite tricky to correctly specify such a language, particularly when type-safety and security considerations are involved. However, these issues are becoming reasonably well-understood.[19]

Here we focus on another large set of applications which are written in languages such as C or C++ that do not provide for multiple threads as part of the language specification. Instead thread support is provided by add-on libraries. In order to focus these discussions, we concentrate on what is arguably the best specified representative of these, Posix[13] threads.[2]

We argue that these environments are as under-specified as the original Java memory model[23], though for somewhat different reasons. In particular, essentially any application must rely on implementation-defined behavior for its correctness. Implementations appear to have converged on characteristics that make it possible to write correct multi-threaded applications, though largely, we believe, based on painful experiences rather than strict adherence to standards. We believe that there is little general understanding on what those characteristics are, nor the exact constraints

[1] Other alternatives include running multiple independent processes, automatically parallelizing existing sequential code, or using a hardware context to prefetch data for the main sequential thread[8]. The first two are clearly significant in certain important domains, e.g. to many network server or numerical applications, but are generally less applicable to, for example, typical desktop or hand-held applications, even when those require substantial CPU resources, such as for image processing. The last applies to a wide variety of domains, but has much more limited benefit than explicitly parallel client code.

[2] We explicitly do not address the less frequently used environments in which multiple concurrent threads communicate communicate primarily through message passing, such as C programs communicating only through the MPI[29] library, or Erlang[9] or Concurrent ML[24] programs.

that language implementations need to obey. Several recent papers suggest that these issues are also not fully appreciated by the research community.

Here we point out the important issues, and argue that they lie almost exclusively with the compiler and the language specification itself, not with the thread library or its specification. Hence they cannot be addressed purely within the thread library.

Most of the pieces we present have been at least superficially mentioned elsewhere, though we believe that particularly the important relationship between register promotion and thread-safety is not widely understood.[3] Our contribution is to present these pieces coherently as an argument that concurrency must be addressed at the language level.[4]

Together with several others, we are joining in an effort[1] to revise the C++ language standard to better accommodate threads. Our goal here is to describe precisely why that is necessary, and why similar efforts are needed for most other environments relying on library-based threads.

2. OVERVIEW

We review the approach to threads exemplified by the Pthreads approach, explaining how and why it appears to work. We then discuss three distinct deficiencies in this approach, each of which can lead, and at least two of which have lead, to subtly incorrect code. Each of these failures is likely to be very intermittent, and hard to expose during testing. Thus it is particularly important to resolve these issues, since they are likely to lead to unreliable production code, and make it impossible to guarantee correctness of multi-threaded programs.

Library-based approaches to concurrency normally require a very disciplined style of synchronization by multi-threaded programs. Although we agree that this is appropriate for perhaps 98% of uses, we argue that it eliminates some low-level programming techniques which, in some cases, may be essential for obtaining any performance benefit from multiple processors. In other cases, such low-level programming techniques can be used to improve the performance of higher-level library primitives, and thus provide pervasive performance improvements for a large collection of applications. Thus, although these usage rules are highly desirable guidelines, we argue that they are inappropriate as absolute requirements in a world in which we need to rely on multiprocessors for performance.

3. THE PTHREADS APPROACH TO CONCURRENCY

Any language supporting concurrency must specify the semantics of multi-threaded execution. Most fundamentally, it must specify a "memory model", i.e. which assignments to a variable by one thread can be seen by a concurrently executing thread.

[3]The only discussion we could find is in an HP Technical Brief[12], which addresses these issue only very partially and from a different perspective.

[4]Peter Buhr [7] makes a similar sounding argument, but he focusses on code written under very different assumptions from the Pthreads model, such as the implementation of the threads library itself. We concentrate on problems that cannot be isolated to the threads library.

Traditionally[15] concurrent execution was viewed as simply an interleaving of the steps from the threads participating in the computation. Thus if we started in an initial state in which all variables are zero, and one thread executes:

```
x = 1; r1 = y;
```

while another executes

```
y = 1; r2 = x;
```

either the assignment to x or the assignment to y must be executed first, and either r1 or r2 must have a value of 1 when execution completes.[5]

This is probably the most intuitive memory model, though not necessarily the one that is easiest to use in practice.[6] It is referred to as sequential consistency.

In practice, it appears unlikely that such a restrictive memory model can be implemented with reasonable performance on conventional architectures. Essentially all realistic programming language implementations supporting true concurrency allow both r1 and r2 to remain zero in the above example.

There are two reasons for this:

- Compilers may reorder memory operations if that doesn't violate intra-thread dependencies. Each pair of actions in the above threads could be reordered, since doing so does not change the meaning of each thread, taken in isolation. And performing loads early may result in a better instruction schedule, potentially resulting in performance improvements. (Cf. [2].)
- The hardware may reorder memory operations based on similar constraints. Nearly all common hardware, e.g. X86 processors, may reorder a store followed by a load[17]. Generally a store results immediately in a write-buffer entry, which is later written to a coherent cache, which would then be visible to other threads.

Thus it is customary to specify a much weaker memory model, which allows results such as r1 = r2 = 0 in the above example. Both the original Java memory model and the new one described in [19] do so, as does the Pthreads standard[13].

The Pthreads standard intentionally avoids specifying a formal semantics for concurrency. The rationale for the standard states in part:

> "Formal definitions of the memory model were rejected as unreadable by the vast majority of programmers. In addition, most of the formal work in the literature has concentrated on the memory as provided by the hardware as opposed to the application programmer through the compiler and runtime system. It was believed that a simple statement intuitive to most programmers would be most effective".

[5]Many more such examples were discussed as part of the work on the Java memory model, which is captured in [22]

[6]As was pointed out during the discussions in [22], it has the clear disadvantage that shared variables used for synchronization are not identified by the source.

Instead, it informally decrees:

> "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_unlock(), ...
>
> [Many other synchronization functions listed]"[7]

Unlike in Java, it is acceptable to leave the semantics of programs with "races", i.e. concurrent reads and writes or concurrent writes, formally undefined, and the standard chooses to do so. (In Java, this is unacceptable because the language is designed to limit the damage that can be caused by untrusted, and possibly malicious code. Thus the semantics of such code need to be specified. We make no guarantees about malicious C/C++ code.)

In practice, C and C++ implementations that support Pthreads generally proceed as follows:

1. Functions such as `pthread_mutex_lock()` that are guaranteed by the standard to "synchronize memory" include hardware instructions ("memory barriers") that prevent hardware reordering of memory operations around the call.[8]

2. To prevent the compiler from moving memory operations around calls to functions such as `pthread_mutex_lock()`, they are essentially treated as calls to opaque functions, about which the compiler has no information. The compiler effectively assumes that `pthread_mutex_lock()` may read or write any global variable. Thus a memory reference cannot simply be moved across the call. This approach also ensures that transitive calls, e.g. a call to a function `f()` which then calls `pthread_mutex_lock()`, are handled in the same way more or less appropriately, i.e. memory operations are not moved across the call to `f()` either, whether or not the entire user program is being analyzed at once.

This approach clearly works most of the time. Unfortunately, we will see that it is too imprecise to allow the programmer to reason convincingly about program correctness, or to provide clear guidance to the compiler implementor. As a result, apparently correct programs may fail intermittently, or start to fail when a new compiler or hardware version is used. The resulting failures are triggered by specific thread schedules, and are thus relatively hard to detect during testing.

A secondary problem with this approach is that, in some cases, it excludes the best performing algorithmic solutions. As a result, many large systems, either intentionally, or unintentionally, violate the above rules. The resulting programs are then even more susceptible to the above problems.

4. CORRECTNESS ISSUES

We list three different issues that we have encountered with the current Pthreads approach. We are not aware of cases in which the first problem led to an actual failure. But anecdotes abound about failures caused by the second problem, and we have personally encountered an intermittent failure caused by the third.

4.1 Concurrent modification

The Pthreads specification prohibits races, i.e. accesses to a shared variable while another thread is modifying it. As pointed out by the work on the Java memory model[19], the problem here is that whether or not a race exists depends on the semantics of the programming language, which in turn requires that we have a properly defined memory model. Thus this definition is circular.

As a concrete example (essentially figure 6 from [14]), consider two threads, each executing one of the following two statements, again in an initial state in which `x` and `y` are zero:

```
if (x == 1) ++y;

if (y == 1) ++x;
```

Does this program contain a race? Is `x` == 1 and `y` == 1 an acceptable outcome?

Under the sequentially consistent interpretation, there is no race, since no variable can become nonzero. Hence we can argue that this is a valid Pthreads program, and `x` == 0 and `y` == 0 is the only valid outcome. (This is in fact the approach taken in [19].)

On the other hand, if our compiler is allowed to transform sequential code not containing calls to pthread operations in any way that preserves sequential correctness, the above could be transformed to[9]

```
++y; if (x != 1) --y;

++x; if (y != 1) --x;
```

[7] This is somewhat further clarified in the rationale. Aside from some insightful discussion of hardware memory reordering, probably the most relevant statements are: "All these functions would have to be recognized by advanced compilation systems so that memory operations and calls to these functions are not reordered by optimization. All these functions would potentially have to have memory synchronization instructions added, depending on the particular machine." There is no discussion of indirect calls to these functions.

[8] It is probably acceptable to guarantee that, e.g. for `pthread_mutex_lock()` and `pthread_mutex_unlock()`, memory operations not move out of a critical section. Full memory barriers may not be needed. See below.

[9] This is probably far-fetched in this example. But it is hard to argue that similar speculative execution is never profitable, especially in the presence of (possibly misleading) profile information, and potentially complex instruction scheduling constraints. And the real question here is whether the transformation is correct, not whether it is profitable.

This would argue both that there is a race, the semantics of this program is formally undefined and `x` == 1 and `y` == 1 is a potential outcome.

Indeed, under the implementation strategy we outlined above, in which the compiler is unaware of threads, it is allowed to transform code subject only to sequential correctness constraints and hence could generate the code containing a race.

Thus we believe that the circularity in the definition is a real issue, though probably not one likely to generate frequent practical problems. Resolving it essential requires a programming-language-defined and compiler-respected memory model, simply to ensure that the user and compiler can agree on when there is a data race.

The remaining two issues are much more serious in practice.

4.2 Rewriting of Adjacent Data

In our preceding example, a compiler could potentially introduce a race by speculatively executing a store operation early. There is in fact no prohibition against storing into a variable that is never mentioned in the source. And indeed, for C or C++ (but not Java), that is often unavoidable.

Consider a C "struct" containing bit fields on a little-endian 32-bit machine:

```
struct { int a:17; int b:15 } x;
```

Since very few machines support a 17-bit-wide store instruction, the assignment `x.a = 42` is likely to be implemented as something like

```
{
  tmp = x; // Read both fields into
           // 32-bit variable.
  tmp &= ~0x1ffff; // Mask off old a.
  tmp |= 42;
  x = tmp;  // Overwrite all of x.
}
```

Note that this effectively stores into both `x.a` and `x.b` normally storing the original value of `x.b` back into `x.b`.

For sequential code this is completely uncontroversial. But if there is a concurrent update to `x.b` that occurs between the `tmp = x` and `x = tmp` assignments in the above code, we have introduced a race, and the concurrent update to `x.b` may be lost, in spite of the fact that the two threads operate on completely distinct fields.

On most architectures this is unavoidable for bit-fields. The behavior is sanctioned by the Pthreads standards, since it prohibits a concurrent write to a "memory location" (a formally undefined term) not just a concurrent write to a program variable.[10]

Unfortunately, this behavior is currently not restricted to adjacent bit-fields. A compiler may read and rewrite any other fields sharing the same "memory location" being assigned. And it may be quite profitable for a compiler to take advantage of this. As an admittedly extreme example, consider the following structure on a 64-bit machine, where it is know to the compiler that `x` is 64-bit aligned:

```
struct { char a; char b; char c; char d;
         char e; char f; char g; char h; } x;
```

where the programmer intended `a` to be protected by one lock, and the other fields by another. If the compiler sees the sequence of assignments:

```
x.b = 'b'; x.c = 'c'; x.d = 'd';
x.e = 'e'; x.f = 'f'; x.g = 'g'; x.h = 'h';
```

It would almost certainly be more efficient to compile this into (taking some liberties with the C notation):

```
x = 'hgfedcb\0' | x.a;
```

i.e. to compile it into a load of `x.a`, which is then 'or'ed with a constant representing the values of the other seven fields, and stored back as a single 64-bit quantity, rewriting all of `x`.

Again, this transformation introduces a potential race, this time with a concurrent assignment to `x.a`, even though the two threads may in fact access disjoint sets of fields. It would also break code that accesses all fields from multiple threads, but chooses to protect `x.a` with a different lock than the other fields, a fairly common occurrence in practice.

The current Pthreads specification explicitly allows this, without any restriction on the field types. By our reading, it even allows it for adjacent global variables outside of a `struct` declaration. Since linkers may, and commonly do, reorder globals, this implies that an update to any global variable may potentially read and rewrite any other global variable.

We do not believe that any interesting Pthreads programs can be claimed to be portable under these rules.

Fortunately, the original motivation for this lax specification seems to stem from machine architectures that did not support byte-wide stores.[11] To our knowledge, no such architectures are still in wide-spread multiprocessor use. And in the case of uniprocessors, restartable atomic sequences[4] can be used to make byte stores appear atomic.

The real issue here is that for a language such as C, the language specification needs to define when adjacent data may be overwritten. We believe that for the language to be usable in a multi-threaded context, this specification needs to be much more restrictive than what a highly optimizing compiler for a single-threaded language would naturally implement, e.g. by restricting implicit writes to adjacent bit-fields.

4.3 Register promotion

There are other optimizing transformations that introduce variable updates were there were none in the source code.

Consider the following program which repeatedly updates the global shared variable `x` inside a loop. As is common in some library code, the access to `x` is protected by a lock, but the lock is acquired conditionally, most probably depending on whether a second thread has been started inside

[10] This formulation was the subject of a clarification request for the Posix standards[28]. The result makes it clear that this was intentional, and "memory location" is intended to be implementation defined.

[11] The first iteration of the Alpha architecture had this characteristic, as did some earlier word-addressable machines. In the case of the Alpha, this part of the architecture was quickly revised.

the process:

```
for (...) {
  ...
  if (mt) pthread_mutex_lock(...);
  x = ... x ...
  if (mt) pthread_mutex_unlock(...);
}
```

Assume the compiler determines (e.g. based on profile feedback or on static heuristics as in, for example, [31]) that the conditionals are usually not taken, e.g. because this application rarely creates a second thread. Following the implementation strategy outlined above, and treating `pthread_mutex_lock()` and `pthread_mutex_unlock()` simply as opaque function calls, it is beneficial to promote `x` to a register `r` in the loop, using, for example, the algorithms outlined in [26] or [18]. This results in

```
r = x;
for (...) {
  ...
  if (mt) {
    x = r; pthread_mutex_lock(...); r = x;
  }
  r = ... r ...
  if (mt) {
    x = r; pthread_mutex_unlock(...); r = x;
  }
}
x = r;
```

The pthread standard requires that memory must be "synchronized with" the logical program state at the `pthread_mutex_lock()` and `pthread_mutex_unlock()` calls. By a straightforward interpretation of that statement, we believe that this requirement is technically satisfied by the transformation.

The problem is that we have introduced extra reads and writes of `x` while the lock is not held, and thus the resulting code is completely broken, in spite of the fact that the implementation seems to satisfy the letter of the specification, and is performing transformations that are reasonable without threads.

It is worth noting that identical problems arise if the above code had called functions named `f` and `g` instead of `pthread_mutex_lock` and `pthread_mutex_unlock`, since `f` and `g` may (indirectly) call thread library synchronization primitives. Hence, again in this case, thread-safe compilation restricts transformations on code that may not be known to invoke thread primitives, and it has implications beyond the semantics of added library calls; register promotion around unknown procedure calls is generally unsafe.

This again argues that compilers must be aware of the existence of threads, and that a language specification must address thread-specific semantic issues. And this one appears to have profound practical implications. We know of at least three optimizing compilers (two of them production compilers) that performed this transformation at some point during their lifetime; usually at least partially reversing the decision when the implications on multi-threaded code became known.

Unfortunately, we expect that in this case thread-safety has a measurable cost in single-threaded performance. Hence confusion about thread-safety rules may also make it hard to interpret even single-threaded benchmark performance.

5. PERFORMANCE

The only parallel programming style sanctioned by the Pthreads standard is one in which Pthread library mutual exclusion primitives are used to prevent concurrent modification of shared variables. And it is this restriction that allowed the implementation strategy outlined above to almost work. With this restriction, the order in which memory operations become visible is intended to be irrelevant unless memory operations are separated by a call to a Pthreads-library routine.

There is much evidence that, at least in our context, this was mostly a reasonable choice. Programs that rely on memory ordering without explicit synchronization are extremely difficult to write and debug.

However, there is a cost involved in following this discipline. Operations such as `pthread_mutex_lock()` and `pthread_mutex_unlock()` typically require one hardware atomic memory update instruction each. On some architectures (e.g. X86 processors), these instructions also implicitly prevent hardware reordering of memory references around the call. When they don't, a separate memory barrier instruction may be required as well. In addition, a dynamic library call is often involved.

The cost of atomic operations and memory barriers varies widely, but is often comparable to that of a hundred or more register-to-register instructions, even in the absence of a cache miss. For example, on some Pentium 4 processors, hardware instructions to atomically update a memory location require well over 100 processor cycles, and these can also double as one of the cheaper mechanisms for ensuring that a store operation becomes visible to other threads before a subsequent load.

As a result of the high cost of these hardware instructions, and the even higher cost of the pthread primitives built on them, there are a small number of cases in which synchronization performance is critical, and more careful and direct use of the hardware primitives, together with less constrained use of shared variables, is essential. In some cases it may also be necessary to avoid dead-lock issues inherent in lock-based programming[6], or desirable because a different parallel programming model is preferable for an application (cf. [27]).

The potential practical benefit of parallel algorithms involving races has long been recognized. A variant of Gauss-Seidel iteration that took advantage of data races on a multiprocessor was described by Rosenfield in 1969[25].

The continued use of the "double-checked locking" idiom, even in contexts such as ours, where it is both technically incorrect[21], and often dangerous in practice, is another indication of at least the perceived need for such techniques.

There is a large literature on lock-free and wait-free programming techniques (cf. [30, 10, 11]) that addresses programming techniques which rely directly on atomic memory operations, in addition to simple atomic loads and stores, but avoid locks. Java recently added a facility for supporting this kind of programming[16]. These all involve races in our present sense.

Although these techniques are currently only appropriate for a small fraction of multi-threaded code, they are often desirable in lower level libraries, and hence may affect the performance of many programs whose authors are unaware of them. For example, it is quite common to use atomic increment operations in the implementation of reference counting in the standard C++ string library.[12]

These techniques generally rely on the ability to access shared variables with ordinary load and store instructions. In practice, some control over reordering memory references is needed as well, but much of the performance of these techniques is attributable to minimizing such restrictions on reordering.

5.1 Expensive Synchronization: An Example

There are many examples in the literature in which lock-free code provides a performance advantage. See for example [20] for a recent and particularly interesting one.

What follows is a particularly simple example, which we believe more clearly illustrates the issues. It takes the form of a simple parallel Sieve of Eratosthenes algorithm, for which shared variable access without ordering or synchronization overhead appears to be critical.

Although this problem appears contrived, it was extracted from a similar issue which occurred in our mark-sweep garbage collector. And indeed, any graph traversal algorithm using mark bits to track visited nodes will incur similar overheads, though it would probably represent a smaller fraction of the entire algorithm.

Consider the following Sieve of Eratosthenes implementation:

```
for (my_prime = start;
     my_prime < 10000; ++my_prime)
  if (!get(my_prime)) {
    for (multiple = my_prime;
         multiple < 100000000;
         multiple += my_prime)
      if (!get(multiple)) set(multiple);
  }
```

where `get` and `set` operations implement a Boolean array A containing 100 million elements. In the simplest case, we might declare A as a sufficiently large array initialized to false values, and `get(i)` and `set(i)` may both be implemented as `A[i]`.

For all values i between 10,000 and 100,000,000, this simple algorithm arranges that on completion `get(i)` is false if and only if i is prime.[13]

Interestingly, it continues to do so if we run multiple copies of this program concurrently in multiple threads, assuming `get` and `set` operate on the same array, in such a way that the entry corresponding to a `set` argument becomes true sometime before program completion, and `get` returns false for entries that have never been `set`.

(Clearly `set(i)` is called only values i which are either smaller than 10,000, or a multiple of such a number. Thus it is never called on a prime in our range of interest. Consider the smallest composite number greater than or equal to 10,000 on which it is not called. This is a multiple of some number $j < 10,000$. For any thread not to invoke `set` on all multiples of j, `get(j)` must have returned true. But then some other thread must have called `set(j)`. That same thread would have invoked `set` on all multiples of j.)

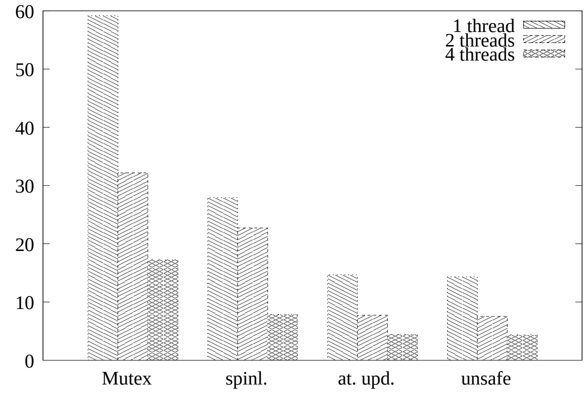

Figure 1: Sieve execution time for byte array (secs)

Thus N copies of the above program running in N threads, correctly compute primes under extremely weak assumptions about the order in which `set` operations become visible to other threads. If updates by one thread become visible to another before program termination, one thread will be able to take advantage of work done by another, and we will see speed-ups due the additional threads. Perhaps more interestingly, there appears to be no good way to take advantage of this with proper synchronization to prevent concurrent modification of array elements.

Figure 1 gives running times of the above program, using 1, 2, or 4 concurrent threads, on a 4-way multiprocessor with relatively low hardware synchronization overhead (1 GHz Itanium 2, Debian Linux, gcc3.3)[14]. Here the "bit-array" A is implemented as an array of bytes. The first set of bars uses traditional pthread "mutex" synchronization (one lock per 256 entries), and the second uses more recently introduced spin-locks, which perform somewhat better in this case.[15] The third uses "volatile" accesses to the array without other synchronization,while the last uses ordinary array accesses. The fourth uses ordinary byte loads and stores. Only the first two are compatible with pthread programming rules.

Figure 2 presents similar data, but with A implemented as a bit array. In this case the third set of bars uses a hardware "cmpxchg" instruction in the `set` implementation to atomically update a bit in the array without risk to the adjacent bits. The fourth set of bars reflects the performance of the program which implements `set` with an ordinary 'or' opera-

[12] The GNU C++ library currently does so. There is a strong argument that, especially in a multi-threaded context, the reference counting here is actually counter-productive. But reference counting implemented with atomic operations greatly outperforms the same algorithms implemented in terms of locks.

[13] As a sacrifice to simplicity, this algorithm does have the minor deficiency that, as stated, it fails to compute primes smaller than 10,000. But even computing those with the normal sequential algorithm would take a trivial amount of time.

[14] Note that gcc3.3 generally does not pipeline such loops, which is important on this architecture, for this simple a loop. Hence absolute performance is almost certainly sub-optimal with this compiler, and the actual overhead of the synchronization operations is understated.

[15] This version performs worse than the mutex implementation if processors are heavily over-committed, as with 20 threads. Hence it is often undesirable in practice. But none of our tests reported here exercise that case. The lock-free implementations are robust against processor over-commitment.

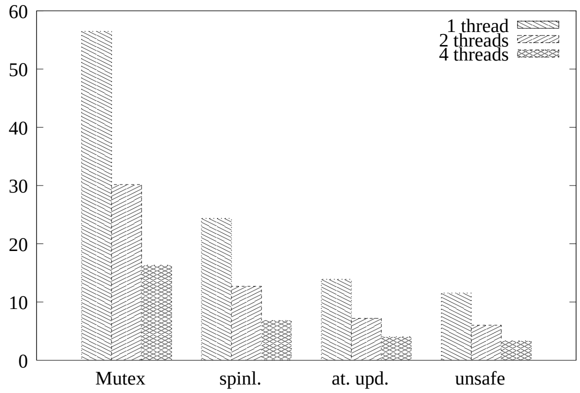

Figure 2: Sieve execution time for bit array (secs)

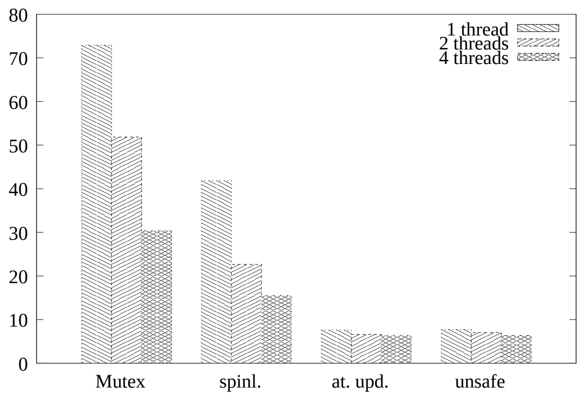

Figure 3: HT P4 execution time for byte array (secs)

tion into the bit vector. This is *incorrect* for more than one thread. (Like most programs with data races, it rarely fails during simple testing like this. Thus time measurements are no problem.)

Note that in either case, we obtain no speed-up over the synchronization-free single-threaded version by using the Pthread mutex primitives, but we see substantial speed-ups (and hence effective multiprocessor use) for the lock-free implementations.

Repeating the byte-array experiment on a hyper-threaded Pentium 4 (2GHz, 2 processors with 2 threads each, Fedora Core 2 Linux), with relatively higher synchronization costs, we see even less promising results for the fully synchronized versions in figures 3. Here the single-threaded version seems essentially optimal, perhaps because it already saturates the memory system.

But for more realistic uses of a shared bit array, we again return to a picture more similar to the Itanium results. Figure 4 gives the time (in milliseconds) required for our garbage collector[5] to trace a heap containing slightly more than 200MB of 24-byte objects, with byte arrays used to represent mark bits. We again vary the number of threads participating in the trace. (For the fully synchronized version, we use a lock per page in the heap.) We see reasonable scaling with thread count, since the number of threads is less than the number of hardware threads. But even with 4 threads, the properly synchronized code only barely exceeds the performance of a single synchronization-free thread, and

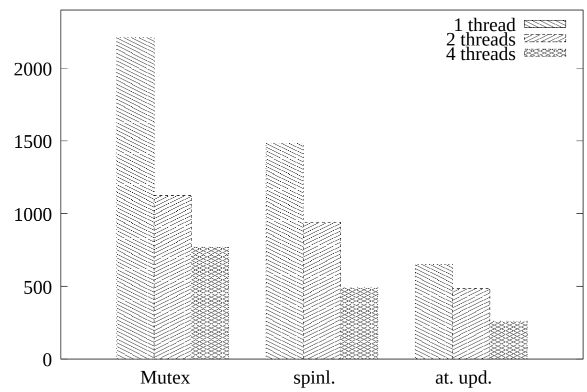

Figure 4: HT P4 time for tracing 200 MB (msecs)

that only with the use of spin-locks.

5.2 Implications of lock-free code

As the above argues, there are cases in which it appears impossible to gain benefit from a multiprocessor without direct fine-grained use of atomic operations. This is impossible in a purely library-based threads implementation in which concurrent variable accesses are disallowed. Once we allow concurrent updates, it falls on the language specification to give their semantics, and on the compiler itself to implement them.

It is still possible to encapsulate the primitives in something that looks to the programmer like a library for accessing shared variables. Some of the recent Java extensions [16] take this route. But to retain full performance, the implementation needs to understand that some of these primitives impose special memory ordering constraints.

The presence of unprotected concurrent accesses to shared variables also implies that additional properties of a pthread-like implementation become visible, and should be specified. Consider the sequence

```
x = 1;
pthread_mutex_lock(lock);
y = 1;
pthread_mutex_unlock(lock);
```

Some implementations of `pthread_mutex_lock()` only include a one-way "acquire" barrier. Thus the above may be executed as

```
pthread_mutex_lock(lock);
y = 1;
x = 1;
pthread_mutex_unlock(lock);
```

with the two assignments reordered. A direct reading of the pthread standard appears to preclude that, but the transformation is undetectable in the absence of races. On some architectures it has a significant performance impact, and is thus desirable.

Once we accept the existence of lock-free code, this distinction becomes visible. To preserve performance, locking operations should probably not be specified as preventing all reordering around them. Indeed, the Java memory model does not.

6. TOWARDS A SOLUTION

Several of us are trying to address these problems in the context of the C++ standard.[1] Other participants in this effort include Andrei Alexandrescu, Kevlin Henney, Doug Lea, Maged Michael, and Bill Pugh.

We currently expect all of the problems to be solvable by a solution based on the approach of the Java Memory Model[19], but adapted to the differing language design goals in a number of ways:

1. Some type-safety and security motivated issues become far less critical. In particular, it is not clear that much of the work on causality in [19] is needed in the context of a type-unsafe language.

2. The Java memory model traded performance for simplicity in a few cases (e.g. the prohibition against reordering a volatile store followed by a volatile load), which may be inappropriate in this context.

3. In the case of at least C++ bit-fields, the compiler must introduce stores, and hence the possibility of races, that were not present in the source. It seems likely that on modern architectures this can be limited to adjacent bit-fields.

7. REFERENCES

[1] 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.

[2] M. Auslander and M. Hopkins. An overview of the PL.8 compiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 22–31, 1982.

[3] A. Bechini, P. Foglia, and C. A. Prete. Fine-grain design space exploration for a cartographic soc multiprocessor. *ACM SIGARCH Computer Architecture News (MEDEA Workshop)*, 31(1):85–92, March 2003.

[4] B. N. Bershad, D. D. Redell, and J. R. Ellis. Fast mutual exclusion for uniprocessors. In *ASPLOS-V: Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 223–233, October 1992.

[5] H.-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[6] H.-J. Boehm. An almost non-blocking stack. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, pages 40–49, July 2004.

[7] P. A. Buhr. Are safe concurrency libraries possible. *Communications of the ACM*, 38(2):117–120, February 1995.

[8] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the 28th International Symposium on Computer Architecture*, pages 14–15, 2001.

[9] Ericsson Computer Science Laboratory. Open source Erlang. http://www.erlang.org.

[10] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):123–149, 1991.

[11] M. Herlihy. A methodology for implementing highly concurrent data structures. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[12] HP Technical Brief. Memory ordering optimization considerations. http://h21007.www2.hp.com/dspp/files/unprotected/ddk/Optmiztn.pdf.

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

[16] D. Lea. Concurrency jsr-166 interest site. http://gee.cs.oswego.edu/dl/concurrency-interest.

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

[18] R. Lo, F. Chow, R. Kennedy, S.-M. Liu, and P. Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 26–37, 1998.

[19] 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*, January 2005.

[20] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, 2004.

[21] B. Pugh. The "double-checked locking is broken" declaration. http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

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

[23] W. Pugh. The java memory model is fatally flawed. *Concurrency - Practice and Experience*, 12(6):445–455, 2000.

[24] J. H. Reppy. Cml: A higher-order concurrent language. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, pages 293–305, 1991.

[25] J. L. Rosenfield. A case study in programming for parallel processors. *Communications of the ACM*, 12(12):645–655, December 1969.

[26] A. V. S. Sastry and R. D. C. Ju. A new algorithm for scalar register promotion based on ssa form. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 15–25, 1998.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, 1995.

[28] A. Terekhov and D. Butenhof. The austin common

standards revision group: Enhancement request 9 (austin/107): Clarification of "memory location". http://www.opengroup.org/ austin/docs/austin_107.txt, May 2002.

[29] The MPI Forum. The message passing interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi/.

[30] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, 1986.

[31] Y. Wu and J. R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 1–11, 1994.