Towards a Memory Model for C++

Hans-J. Boehm
HP Laboratories

© 2004 Hewlett-Packard Development Company, L.P.
The information contained herein is subject to change without notice

Acknowledgments

- Some of this has benefited greatly from discussions with Bill Pugh, Doug Lea, Peter Dimov, Clark Nelson, Alexander Terekhov, Andrei Alexandrescu, Herb Sutter, David Callahan, Jeremy Manson, ...
- Some of the observations here, e.g. the difficulty in defining a "memory location" have been known for a long time, but seemed to be underappreciated.

Rough Outline

1. What's wrong with the C/C++ & Pthreads/win32 programming model
   - And why we need a proper memory model to describe thread interaction.
2. What we are doing about it
   - Overview of the technical challenges.
   - Why atomic operations matter.
   - Our approach.
   - Implementation consequences.

Multi-threaded programming is increasingly important:

- It continues to be widely used to deal with multiple event streams.
- We need parallel programs to take advantage of multi-core processors.
  - And those are likely to be the main source of improved performance.
- Threads are the obvious way to get there.
  - The APIs exist and are widely used.
  - And sometimes only shared memory yields sufficient performance.

Common approaches:

- Threads integral to the language:
  - Java
    - Hard to get the semantics right.
    - See Manson, Pugh, Adve, “The Java Memory Model”, POPL05
  - C#, …
- Single-threaded language + threads library.
  - Assume no type-safety/sandboxing concerns.
  - C/C++ & Pthreads, Win32 threads, …
  - Simple. Compiler & language spec oblivious to threads.
  - This talk: Close, but inherently not quite correct.

We use pthreads for details

- The rules governing shared variable accesses are relatively well specified.
- Widely used, surprisingly close to “correct” for such a simple spec.
- Original win32 threads approach appears identical.
  - But I couldn’t find the analogous specification.
  - Differences in synchronization primitives not relevant.
- Java/C# present different issues

Pthreads rules

No concurrent modification to shared variables (no races):

"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. *[i.e. no data races.]* Such access is restricted using functions that synchronize thread execution and also synchronize memory with respect to other threads…"

- Single Unix SPEC V3 & others

These functions include **`pthread_mutex_lock()`** …

- Seemingly independent of language specification.
- C and C++ specifications don't mention threads.

Why no data races?

- Almost dodges "memory model" issues:

(Initially x = y = 0)

Thread 1	Thread 2
x = 1;	y = 1;
r1 = y;	r2 = x;

Can r1 = r2 = 0?

  - Intuitively (or under sequential consistency) no; some thread executes first.
  - In practice, yes; compilers and hardware can reorder.

- Under pthreads rules this is simply illegal.
  - We don't really get to ask the question.

How Pthreads Implementations (and win32 threads?) *almost* work

- Synchronization-free code can be optimized as though it were single-threaded.
  - If a thread could observe the difference, the observer would introduce a race.
- Synchronization functions contain any needed hardware memory barriers.
- Synchronization functions are treated as opaque by compilers.
  - The compiler views them as potentially reading or writing any global.
- Compilers can't normally move memory references across them.
- Compilers that follow single-threaded optimization rules *rarely* break multi-threaded code.

Why this doesn't quite work
… and threads need to be in the language

1. The basic rules are circular.
   - You need a memory model to define concurrent modification and races.
2. What's a "memory location"?
   - The language spec must say.
   - Impacts compiler.
3. What does it mean to "synchronize memory"?
   - The straight-forward interpretation
     - Is easy to implement.
     - Can unexpectedly break code.
4. Performance for the 2% (??) of code that needs shared variable access without locks.
   - Which may affect overall performance substantially.

Concurrent modification

- Does the following program access a memory location "while another thread of control may be modifying it"?

Initially **x** = **y** = 0

Thread 1

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

Thread 2

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

or how about

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

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

?

- Need a semantics for the concurrent language to define when there is a race.

"Memory location":

- Consider

```
x = y = 0
```

Thread 1
```
x = 1;
```

Thread 2
```
y = 1;
```

- Can we guarantee x = y = 1 when both threads finish?
  - If x and y are adjacent bit-fields, no.
  - If x and y are adjacent char members on an ancient Alpha, no.
  - If x and y are adjacent char members on a recent Alpha, compiled for backwards compatibility, no.
  - If x and y are "close" data members, and the compiler decided to combine the store with another one, maybe not.
  - If one of x or y is a bit-field, and the other is close, maybe not .
- Pthread's uniform answer: No. They may occupy the same "memory location".

- There is no portable pthreads code. Needs fix.

"Synchronize memory"?

- Really not sufficient: (register promotion)

[g is global]

```
for(...) {
  if(mt) lock();
  use/update g;
  if(mt) unlock();
}
```

```
r = g;
for(...) {
  if(mt) {
    g = r; lock(); r = g;
  }
  use r instead of g;
  if(mt) {
    g = r; unlock(); r = g;
  }
}
g = r;
```

“Synchronize memory” (contd.)

- Memory contents at lock() and unlock() calls reflect logical state.
  - Clearly not sufficient.
- Again compiler adds stores (and introduces races) not present in the source.
  - Invisible with single thread.
  - Visible to another thread.
  - This is
    - Unacceptable.
    - Common optimization for both commercial and research compilers.
    - Rarely visible, hard to test.
- Language definition, compiler must preclude this.

Speculative register promotion, again

- Note that even very simple cases can be unsafe:

[count is global]

```
for (p = q; p != 0; p = p -> next) {
    if (p -> val > 0.0) count++;
}
```

- May not touch count if q has only negative entries.
- Promoting count to a register introduces unconditional access, but
- Unconditionally setting global count at the end of the loop may introduce race and is unsafe!
- Even gcc –O2 does this.

Performance

- So far we were dealing with correctness.
- Pthreads rules require “fully synchronized” programs.
- A good idea for 98% of code.
- The rest of this is about the other 2%
  - … which may account for > 50% of application performance.

Fully synchronized programs can be slow

- Traditional pthread_mutex's require:
  - Dynamic library call
  - 2 x (Atomic op + memory barrier(s))
- Sample cycle costs:

	CAS	barrier	lock/unl.
2.0 Xeon	124	125*	336
1.0 Itan.	10	4+	109
500 PIII	25	19*	156

*Not needed with compare-and-swap (CAS)

Faster alternatives:

- Extensive literature on lock-free algorithms.
- Here we use two examples:
  - "Double-checked" locking idiom.
  - Parallel GC with
    - locked mark bit update, vs.
    - "cheating" and relying on hardware facilities
      - In this case atomic byte stores.
      - Concurrent stores are safe in this case.
- 2005 PLDI paper has Sieve of Eratosthenes example.

A pervasive example: Double-checked locking

hp invent

- Assume we want to lazily initialize x
  - while avoiding locking on the fast path

```
if (!x_initialized) {
  lock();
  if (!x_initialized) x = …;
  x_initialized = true;
  unlock();
}
… x …
```

- Incorrect as is:
  - Data race on `x_initialized`!
  - May fail in practice for multiple reasons, esp. on some hardware.
- But we want something like this to work.

P4 parallel GC trace performance (Time to trace 200 MB, msecs)

1 thread
2 threads
4 threads

2000
1500
1000
500
0

Mutex
spinl.
at. upd.

Parallel GC performance summary

- `Pthread_mutex_lock()` version on 4 "processors" (2x2 threads) is slower than uniprocessor version.
- Why bother with a multiprocessor?
- Sometimes concurrent writes to shared variables are unavoidable.

What does all of this mean?

- Pthread-like thread specifications are inadequate.
  - Language specification must be involved.
    - When can there be a data race?
    - When can adjacent data be rewritten as part of an assignment?
    - Can additional reads and writes of globals be introduced by the compiler?
    - Any guarantees with data races?
    - These are all compiler/language issues.
  - Need atomic operations support for occasional un-locked access to shared globals.
  - A proper memory model for the language makes all of this feasible.

We’re doing something about this:

- We’re defining a “memory model” or thread semantics describing visibility of memory accesses to other threads, and answering those questions.
  - Java now has a reasonable one (Pugh, Manson, Adve, others)
  - We’re working on C++.
    - Also: Andrei Alexandrescu, Doug Lea, Bill Pugh, Clark Nelson, Maged Michael, Ben Hutchings, Peter Dimov, Alexander Terekhov and others.
    - Herb Sutter is pursuing similar goals for Microsoft (not .NET)
    - Currently we have strong support from C++ committee.
    - Different Technical challenges from Java.
    - Rest of this talk.

Observation:

- Type-safe languages supporting sandboxing have relatively strong requirements:
  - Completely undefined semantics are generally not acceptable.
    - Malicious code may exercise undefined semantics.
    - We do care what malicious code can do.
  - Need some immutability guarantees.
    - Otherwise security checks involving e.g. strings are meaningless.
- Those requirements seem to add complexity.
  - Something like Java's causality treatment appears hard to avoid.

C++ constraints are different:

- Undefined semantics are OK.
- Fully defined semantics are relatively more expensive?

  ```
  foo *y = 0;
  ```
  Thread 1: `foo x(); y = &x;`
  Thread 2: `if (y != 0) y -> bar();`
  - Is thread 2 allowed to see a partially initialized pointer, or a bad vtable pointer and take a wild branch?
- Disallowing this requires:
  - Atomic pointer assignments (even on small embedded processors).
  - Potentially memory fences at the end of constructors.
    - And those are more frequent than in Java.

Our Approach:

- "Pthreads-like" memory model.
- Data races (updates to a memory location concurrent with another access) have undefined semantics. (Still convincing Herb Sutter …)
- Otherwise: Sequential consistency.
- Careful and restrictive definition of "memory location“ and “data race”.
  - A “memory location” is any non-bit-field object, or contiguous sequence of (!= 0 length) bit-fields declared in the same structure.
- Implicitly, extra stores cannot be visibly introduced.

But atomic operations add complications

- We need atomic operations which support concurrent access.
  - For sophisticated lock-free data structures.
  - For a few idioms like double-checked locking.
- This complicates the definition of data race
- … and potentially raises a lot of the (hard) Java issues
- … and then some
  - Because we want more general atomic operations (?)

Atomic operations: Allow concurrent access in special situations

- Such as double-checked locking:
  - Loads and stores of `x_initialized` must be done specially:
    - Tell compiler (and programmer) that a race is involved.
    - Ensure atomicity.
    - Specify ordering constraints.
  - Current thinking is to provide two levels:
    - Java-volatile-like high level.
    - Lower level with finer control.
    - Here we concentrate on the low-level
      - Since memory model needs to accommodate full generality.

Double-checked locking: Correct, with ll atomic operations

- Using tentative, intentionally ugly, very explicit, syntax:

```
atomic<bool> x_initialized;
                //Only required change

if (!x_initialized.load<acquire>()) {
   lock();
   if (!x_initialized.load<raw>())
     x = ...;
   x_initialized.store<release>(true);
   unlock();
}
... x ...
```

- `Store<release>` ensures that preceding stores are visible to a `load<acquire>` reading variable in another thread.
  - Allow concurrent access
  - Inhibit compiler, hardware reordering

Atomic operation semantics

Variables x, y and z initially 0

Thread 1:

```
x.store<raw>(1);
r1 = y.load<raw>();
if (r1 == 0) z = 17;
```

Thread 2:

```
y.store<raw>(1);
r2 = x.load<raw>();
if (r2 == 0) z = 42;
```

- Does this have a data race?
- Simultaneous accesses through atomics don't count.
- No race on z under sequentially consistent interpretation.
- But simultaneous accesses are really possible.
- This must have undefined semantics in order to preserve the compilers optimization ability.
- But how do we specify this? While dodging Java issues?

Current approach (ordering)

- Define a "happens-before" relation to take the place of sequential ordering for update visibility.
  - As usual, not a total order.
- Happens-before is basically
  - intra-thread sequencing where it affects outside ordering +
    - (Some complexity here, but can usually be ignored.)
  - release op happens-before next acquire of same lock or atomic object
- Object reads do not see assignments that "happen-after" them, or that are hidden by assignments that "happen in-between".
- If two accesses (> 1 write) to the same non-atomic "memory location" occur without a "happens-before" ordering, there is a data race, and meaning is undefined.
  - Otherwise reads are allowed to see any value consistent with above.

Happens-before illustration (atomics)

Thread 1

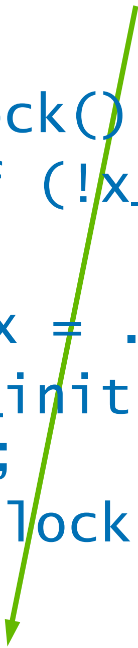
```
if (!x_init.ld<acq>())
{
  lock();
  if (!x_init.ld<…>())
    x = ...;
```

```
  x_init.store<rel>(1)
    ;
  unlock();
}
... x ...
```

Thread 1

```
if (!x_init.ld<acq>())
{
  lock();
  if (!x_init.ld<…>())

    x = ...;
  x_init.store<rel>(1)
    ;
  unlock();
}
... x ...
```



Happens-before illustration (2)

Thread 1

```
x.st<rel>(0);
```

```
x.st<rel>(1);

r1 = y.ld<acq>();
```

Thread 2

```
y.st<rel>(0);
```

```
y.st<rel>(1);

r2 = x.ld<acq>();
```

- `r1` = `r2` = 0 possible (unlike Java)
- No race, since these are atomics.

Current approach (summary)

- Claim: This guarantees sequential consistency for race-free programs using locks for synchronization.
  - (If you use `trylock()`, this is true, but only because our definition of data race is different from Posix'.)

Does require compiler changes:

- No speculative or unnecessary stores.
  - Stores to struct/class members may not unnecessarily overwrite adjacent members.
    - Intel Example:
      - `struct {char a; int b:9; int c:7; char d;}`
    - A store to `b` must be implemented as 2 separate 1-byte stores.
  - Speculative register promotion often illegal:

```
for (T *p = q; p != 0; p = p -> next)
   if (p -> data > 0) ++count;
```

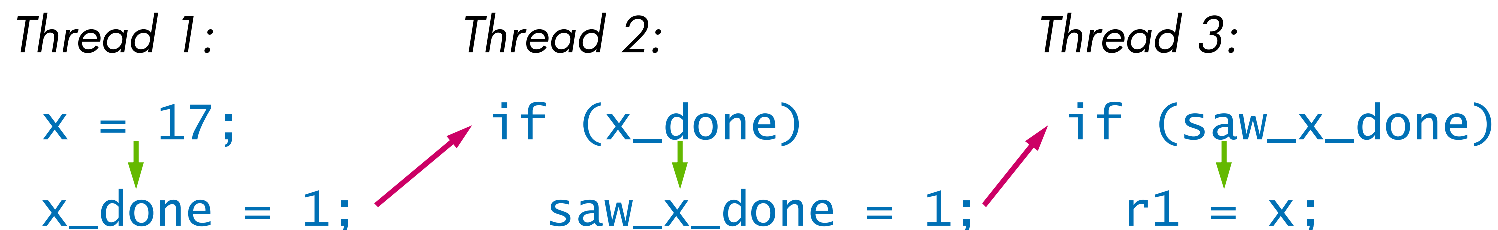
    - Standard register promotion of `count` becomes illegal.
  - Some kinds of code hoisting are problematic.

Compiler Changes continued

- Traditional compiler analysis assumes that code just before a loop is executed iff code just after loop is executed.
- False for nonterminating loops.
  - Doesn’t matter for sequential code.
- Often hard for compiler to prove loops terminate.
- Two alternatives:
  - Break lots of compiler analyses
    - And a few useful transformations (loop jamming)
  - Make synchronization / I/O – free infinite loops invoke unspecified behavior
  - We’re proposing the latter. (Is it even a change?)

Architectural implications

- Byte stores are basically required.
- Atomic operations (e.g. cmpxchg) highly desirable.
- Need a cheap way to enforce transitivity of happens-before (basically causal ordering):
  - Assume x_done, saw_x_done are atomic:



- As with Java volatiles, `r1` must be 17.

Current status

- Web page at
  http://www.hpl.hp.com/personal/Hans_Boehm/c++mm
- Includes informal proposal, pointers to standards committee papers, …
- Clark Nelson and I proposed an initial version of "standardese". (WG21/N2052)
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2052.htm
- Needs further scrutiny.
- Preliminary atomic operations library interface.
  - Converging but we're still iterating.
- An official C++ threads API is also progressing …

Backup slides

Double-checked locking: Why it is prohibited by pthreads.

- Compiler/hardware may reorder

```
if (!x_initialized) {
   lock(); // Not real syntax
   if (!x_initialized) x = ...;
   x_initialized = true;
   unlock();
}
... x ...
```

- E.g., compiler may load `x` early after discovering that it misses cache.
- Some architectures allow hardware reordering of loads of `x` and `x_initialized`.
- In this case, we also need to restrict reordering.

Faster alternatives: A Simple Example

- Sieve of Eratosthenes
  - Compute primes between 10K and 100M
  - Each thread executes:

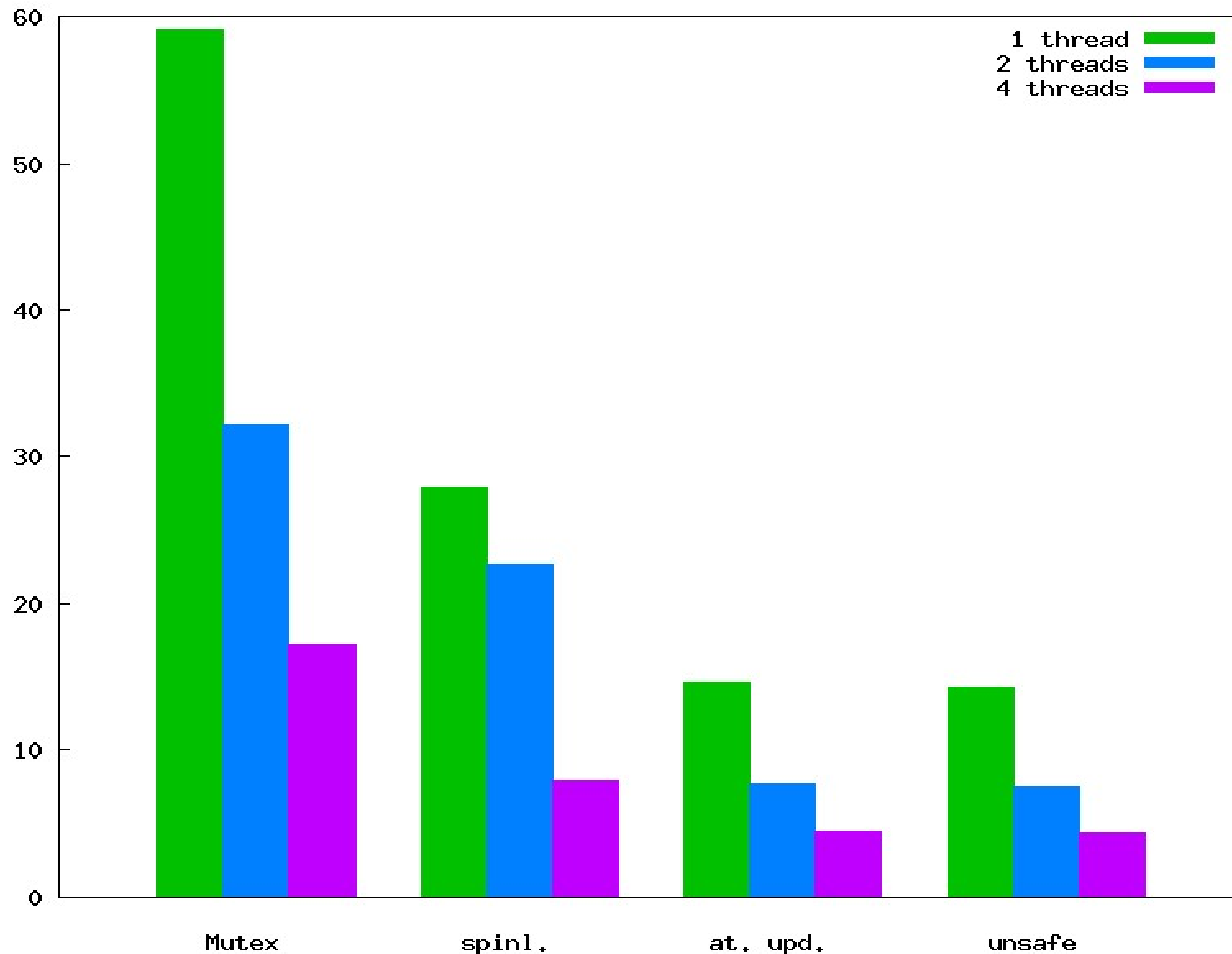
```
for (my_prime = start; my_prime < 10000; ++my_prime)
    if (!get(my_prime)) {
        for (multiple = my_prime; multiple < 100000000;
              multiple += my_prime)
            if (!get(multiple)) set(multiple);
    }
```

  - Get/set operate on 100M "bit" array.
  - Similar to core of some parallel garbage collectors.

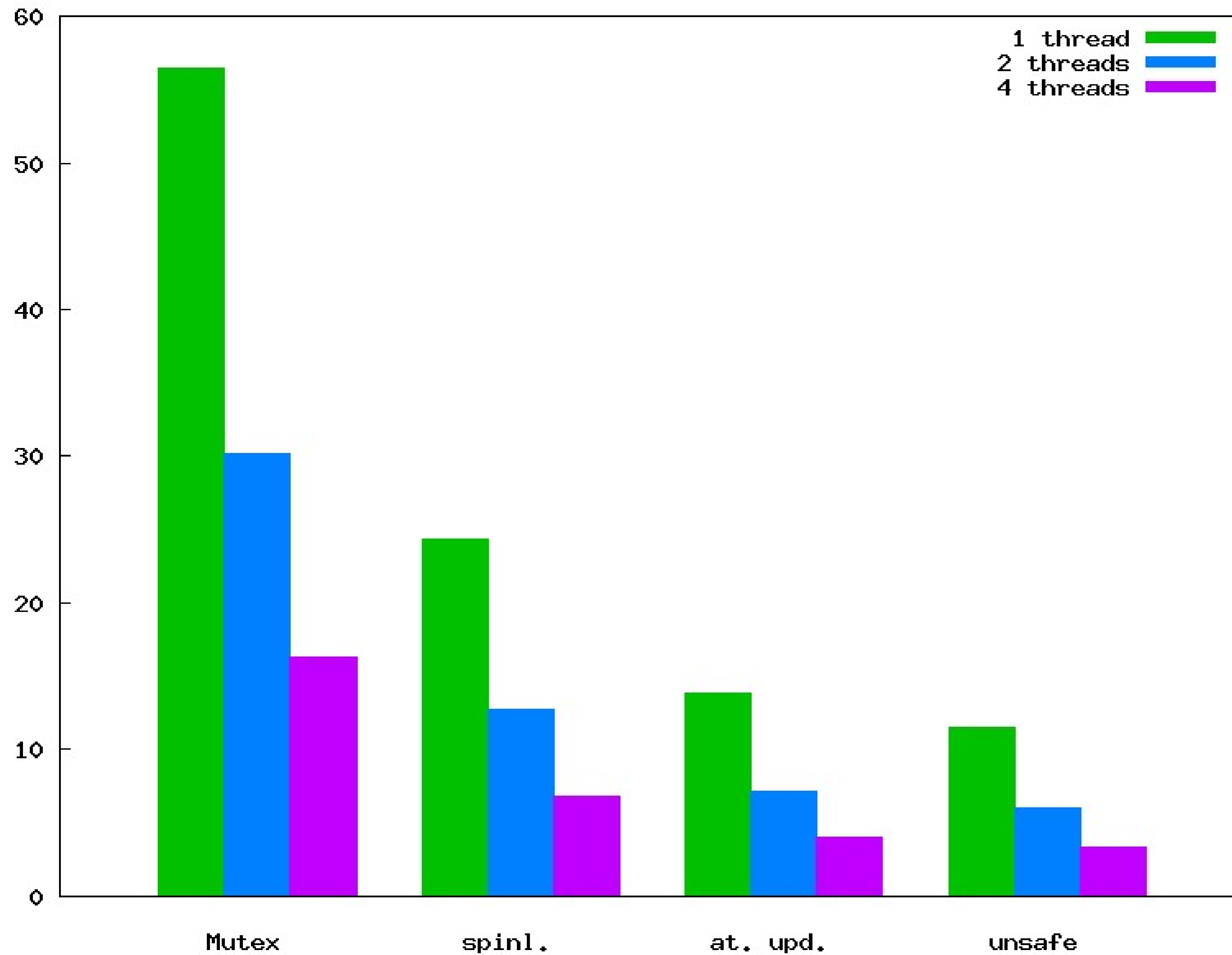
Measurements: Alternatives

- Bit array vs. byte array
- Synchronization alternatives:
  - None (thread unsafe, but usually "works")
  - Atomic byte stores for bytes
  - Atomic or into bit array
    - Needs portable access to e.g. cas
  - **`Pthread_mutex`** locks (every 256 bits)
  - **`Pthread_spin`** locks (every 256 bits)

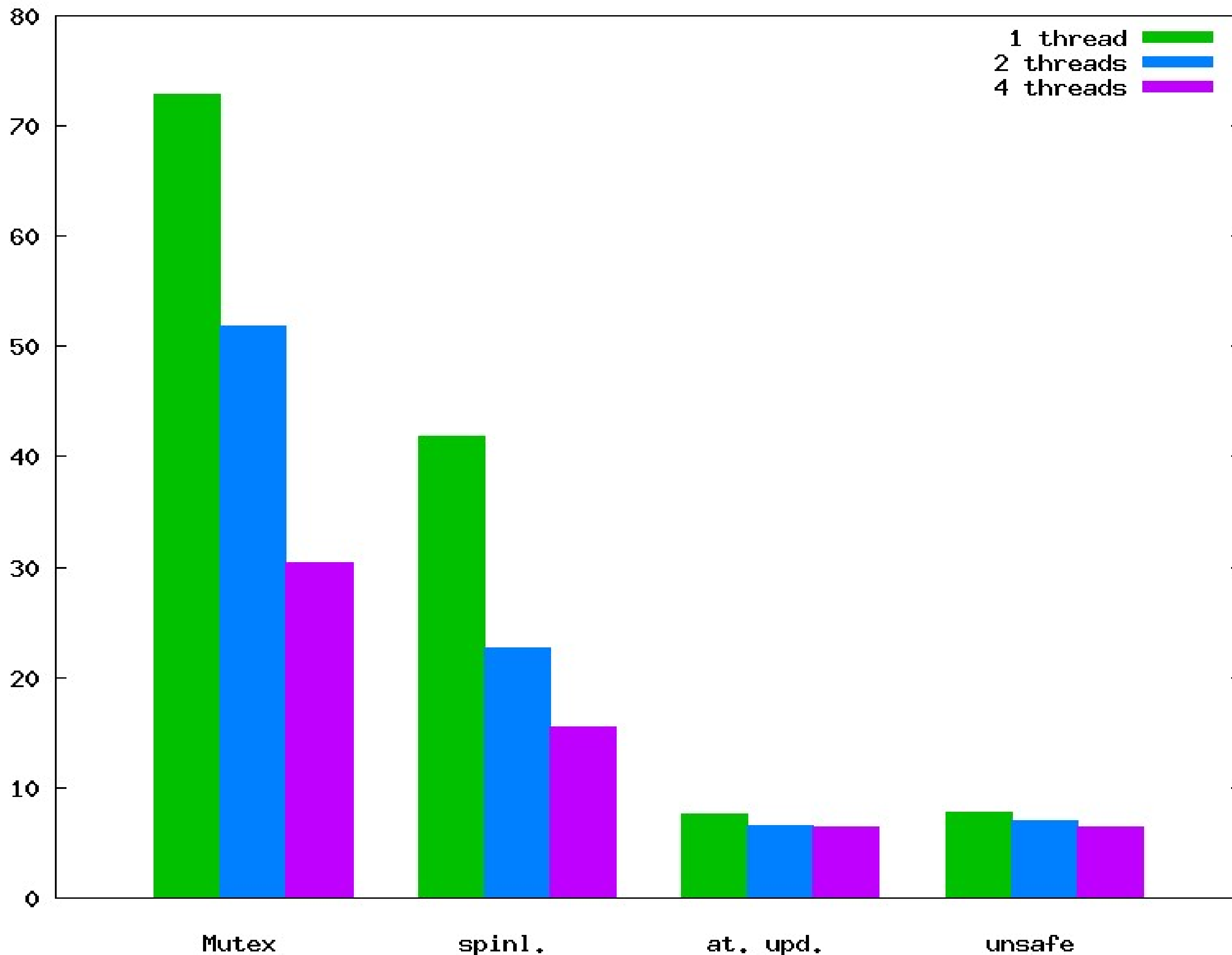
Itanium (gcc) running time 4x1GHz, bytes



Itanium (gcc) running time 4x1GHz, bits



Pentium 4 time (2x HT 2GHz), bytes



Performance summary

- **`Pthread_mutex_lock()`** version on 4 "processors" is slower than uniprocessor version.
  - Why bother with a multiprocessor?
- Versions with atomic operations or byte arrays either
  - Scale reasonably, or
  - Saturate the bus/memory (?)
- In the first case
  - We can get good speedups even in this contrived case.
  - Real code benefits substantially from atomic operation access.
- In the second case
  - Contrived case doesn't speed up.
  - Real code requires atomic operations for speed up.