

C++0x Thread Support for Open64

Hans-J. Boehm

HP Labs



Overview

- We don't have thread support for Open64 ☹️
 - I'm not even an Open64 expert.
 - But I did lead the C++0x memory model effort ...
 - Which broke all (?) C++ (and C!) compiler back-ends.
- This is an overview of C++0x threads, and what's needed.
- Very quick overview of
 - Front end support.
 - Library support.
- More discussion of back-end-relevant changes.

Threads in C++0x

Threads are finally part of the language! (C1x, too!)

1. Quick intro to threads API

2. Memory model

– What exactly do shared variables mean?

- Not quite the naïve sequential consistency model.

– When does thread a see an update by thread b ?

– When is it OK to simultaneously access variables from different threads?

3. Atomic operations

Thread creation example:

```
int fib(int n) {  
    if (n <= 1) return n;  
    int fib1, fib2;  
    thread t([=, &fib1]{fib1 = fib(n-1);});  
    fib2 = fib(n-2);  
    t.join();  
    return fib1 + fib2;  
}
```

Disclaimers:

- Untested code!
- Don't really do this! It creates too many threads.
- Runs in exponential time. There is a $\log(n)$ algorithm.

A cleaner way to write parallel fib()

```
int fib(int n) {  
    if (n <= 1) return n;  
    int fib2;  
    auto fib1 =  
        async( [= ] { return fib(n-1); } );  
    fib2 = fib(n-2);  
    return fib1.get() + fib2;  
}
```

Counter with a mutex

```
mutex m;
```

```
void increment() {  
    m.lock();  
    x = x + 1;  
    m.unlock();  
}
```

- Lock not released if critical section throws.

Counter with a lock_guard

```
mutex m;
```

```
void increment() {  
    lock_guard<mutex> _(m);  
    x = x + 1;  
}
```

- Lock is released in destructor.

Other API components

- Mutual exclusion, monitors: `recursive_mutex`, `timed_mutex`, `call_once`, `unique_lock`, `condition_variable`, `condition_variable_any`.
- `async` building blocks: `promise`, `future`, `shared_future`, `packaged_task` blocks, cross-thread exception propagation.
- `quick_exit`: Partially addresses problems with detached threads.
- Atomic variable access: `atomic<T>` (stay tuned).
- Conspicuously absent (postponed): Reader-writer locks, general semaphores, thread pools, fork-join framework, ...

And some base language support:

- Function-local statics are thread safe.
- `thread_local` variables can have constructors and destructors.
- Unordered constructors may run in parallel.

But library APIs are not enough!

What do shared variables mean?

- Simple interleaving-based semantics overconstrain the implementation.
 - In ways that don't really help the programmer.
- “Dekker's” example (everything initially zero) should allow $r1 = r2 = 0$:

Thread 1

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

Thread 2

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

- Compilers like to perform loads early.
- Hardware likes to buffer stores.

C++0x threads memory model

- *Data race* = simultaneous non-read-only accesses to the same scalar or bit-field-sequence.
- **Data races are (still!) disallowed!**
 - No notion of a “benign” data race.
- If you avoid data races and constructs that explicitly say otherwise:
 - We guarantee sequential consistency.
 - Programs behave as though steps are interleaved.
- Compiler may not introduce visible data races!
- Consistent with Java approach, though weaker, and more correct.

Dekker's example, again:

- (everything initially zero):

```
int x,y;
```

Thread 1

```
x = 1;
```

```
r1 = y; // reads 0
```

Thread 2

```
y = 1;
```

```
r2 = x; // reads 0
```

- This has a data race:
 - `x` and `y` can be simultaneously read and updated.
- Has undefined behavior.

Data-race-free property: State in sync-free-region is unobservable.

- Compiler may reorder code within sync-free region.
 - Compiler may transform sync-free regions almost as sequential code.
- Any observation of intermediate state would constitute a data race:

Thread 1 (not atomic):

```
a = 1;
```

```
b = 1;
```

race!

Thread 2(observer):

```
if (a == 1 && b == 0) {
```

```
...
```

```
}
```

Not all sequential transformations are valid:

- Single thread compilers currently may add data races: (PLDI 05)

```
struct {char a; char b} x;
```

```
x.a = 'z';
```



```
tmp = x;  
tmp.a = 'z';  
x = tmp;
```

- `x.a = 1` in parallel with `x.b = 1` may fail to update `x.b`.
- Still broken (open64, gcc) for assignment to `b` in
 - `struct { char a; int b:9; }`

Some restrictions are a bit more annoying:

- Compiler may not introduce “speculative” stores:

```
int count;    // global, possibly shared
...
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++count;
```



```
int count;    // global, possibly shared
...
reg = count;
for (p = q; p != 0; p = p -> next)
    if (p -> data > 0) ++reg;
count = reg; // may spuriously assign to count
```

But the news for optimizers isn't all bad:

- Clarity about memory model allows us to deduce new properties:
- Assume x global, x and y not modified in loop, ellipses synchronization free:

```
while (...) {  
    y = a[x]; ...  
    m.lock();  
  
    ...  
    m.unlock();  
}
```

- x and y are loop invariant!
- See Effinger-Dean, Boehm, Chakrabarti, Joisha in MSPC11

Potential problem:

Data races were already disallowed

- ... and many programs contain intentional data races anyway.

Synchronization variables

- C++0x: `atomic<int>`, `atomic_int`
- C1x: `_Atomic(int)`, `_Atomic int`, `atomic_int`
- Java: `volatile`, `java.util.concurrent.atomic`.
- C# : none, though `volatile` is closest.
- *not* C/C++ `volatile`!
- Guarantee indivisibility of operations.
- “Don’t count” in determining whether there is a data race:
 - Programs with “races” on synchronization variables are still sequentially consistent.
 - Though there are “escapes” in C++0x.

Dekker's example, yet again:

```
atomic<int> x(0), y(0);
```

Thread 1

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

Thread 2

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

- No data race: The only concurrent accesses are to atomics.
- Implementation must ensure sequential consistency.
- $r1 = r2 = 0$ is not possible.

Compilation of atomics

- On X86, for sufficiently small & aligned data:
 - Atomic stores require trailing **MFENCE**, or use of **XCHG**.
 - Atomic RMW ops compile to **LOCKed** instructions.
 - Compilation into opaque library calls OK as stopgap?
 - There are opportunities to do better.
- For other architectures talk to the experts!
 - The rules are subtle ...

Problem:

- Compilation of `atomic<T>` accesses requires memory fences to enforce sequential consistency.
- Expensive on some architectures.
- C++0x compromise:
 - Atomic operations are sequentially consistent by default.
 - But relaxed versions are also provided.

C++0x explicitly ordered (low-level) atomics

- Pairs of atomic operations cannot form a data race.
- Operations that do not specify `memory_order_seq_cst` (the default) are not guaranteed to execute in a single total order.
- A `memory_order_release` store still guarantees memory visibility to `memory_order_acquire` load that reads the value.

```
atomic<bool> flag;
```

Thread 1:

```
data = 42;  
flag.store(true, memory_order_release);
```

Thread 2:

```
if (flag.load(memory_order_acquire)){  
    assert (data == 42)
```

Open64 status

- No stores introduced as a result of register promotion?
 - Unlike gcc
- But bit-fields are not yet handled correctly.
- Atomic operations unimplemented.
 - Which means that programmers will try to cheat.

Note: Threads in C1x

- Are also there.
- API more like Posix subset.
 - Mutex kinds distinguished at initialization time.
 - Not by type.
- Memory model identical.
- Atomic operations similar.
 - `_Atomic(t)` vs. `atomic<t>`

Questions?