INTRODUCING THE C++ MEMORY MODEL

by Mike Long / @meekrosoft

WHY C++?

http://en.wikipedia.org/wiki/Bjarne_Stroustrup

SIMULA

The poor run-time characteristics were a function of the language and its implementation rather than a function of the application. The overhead problems were fundamental to Simula and could not be remedied

A History of C++: 1979-1991, Bjarne Stroustrup

BCPL

I re-wrote the simulator in BCPL and ran it on the experimental CAP computer. The experience of coding and debugging the simulator in BCPL was horrible. BCPL makes C look like a very high level language

C WITH CLASSES

Stroustrup wanted to write efficient systems programs in the styles encouraged by Simula67. To do that, he added facilities for better type checking, data abstraction, and object-oriented programming to C. The more general aim was to design a language in which developers could write programs that were ***both efficient and elegant****. Many languages force you to choose between those two alternatives.*

http://isocpp.org/wiki/faq/big-picture#why-invented, emphasis added

C++ was designed for applications that had to work under the most stringent constraints of run-time and space efficiency. That was the kind of applications where C++ first thrived: operating system kernels, simulations, compilers, graphics, real-time control, etc. This was done in direct competition with C. Current C++ implementations are a bit faster yet.

http://accu.org/index.php/journals/1356

PERFORMANCE

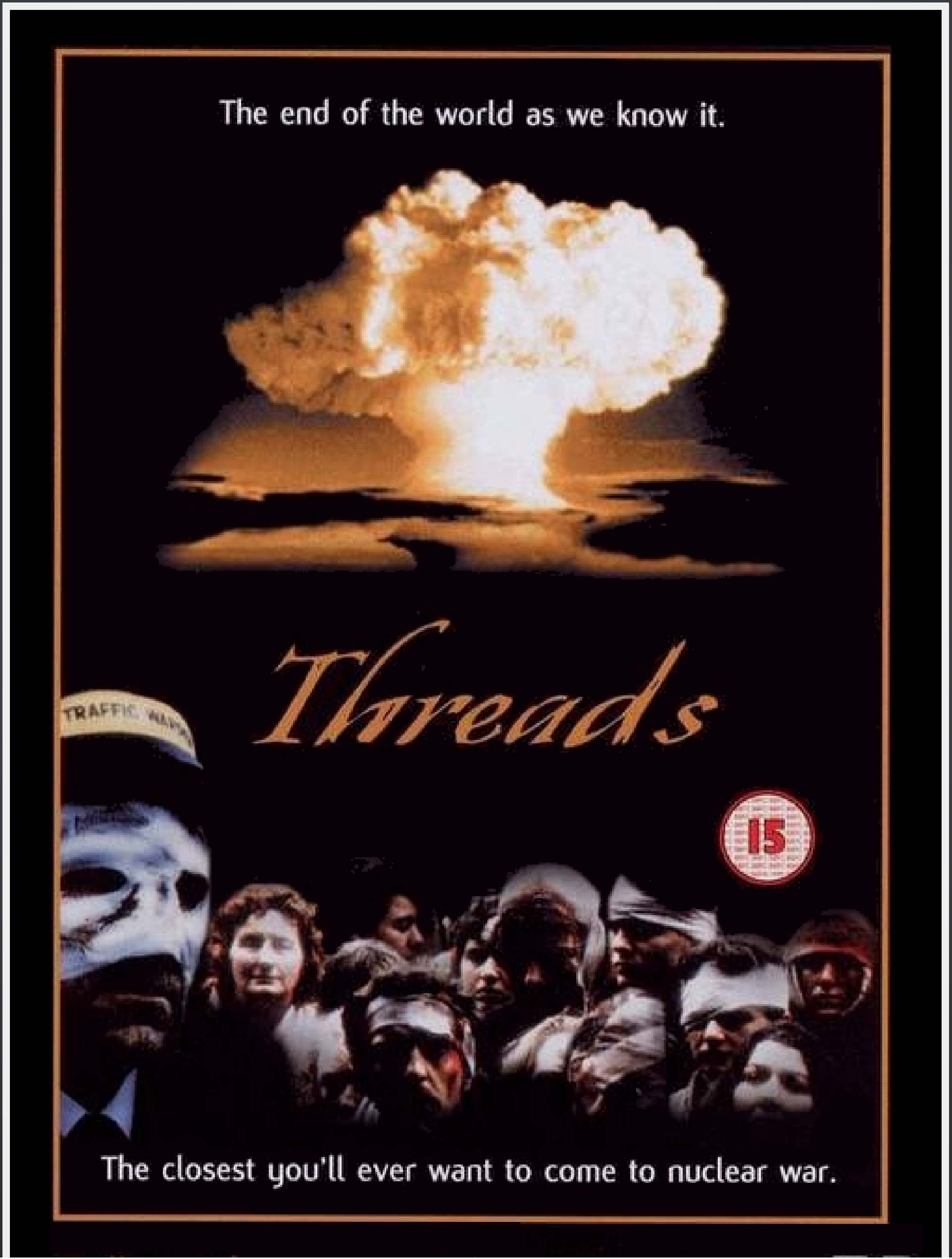
`Performance(1979) != Performance(2004)`

PERFORMANCE(1979)

- Simula minus the language overhead
- Only pay for what you use

PERFORMANCE(2004)

- Free lunch is over


The end of the world as we know it.
Threads
TRAFFIC WARDEN
15
The closest you'll ever want to come to nuclear war.

"Brilliant, informative - and shattering" - Daily Express
DVD VIDEO

WHERE IS THE LOVE?

Although threads usage is very widespread and growing, the basic rules for programming with threads, and particularly for accessing shared variables, have been confusing. Even "experts" have often advocated contradictory approaches.

Hans Boehm, http://www.hpl.hp.com/techreports/2009/HPL-2009-259html.html

WHERE ARE THE THREADS?

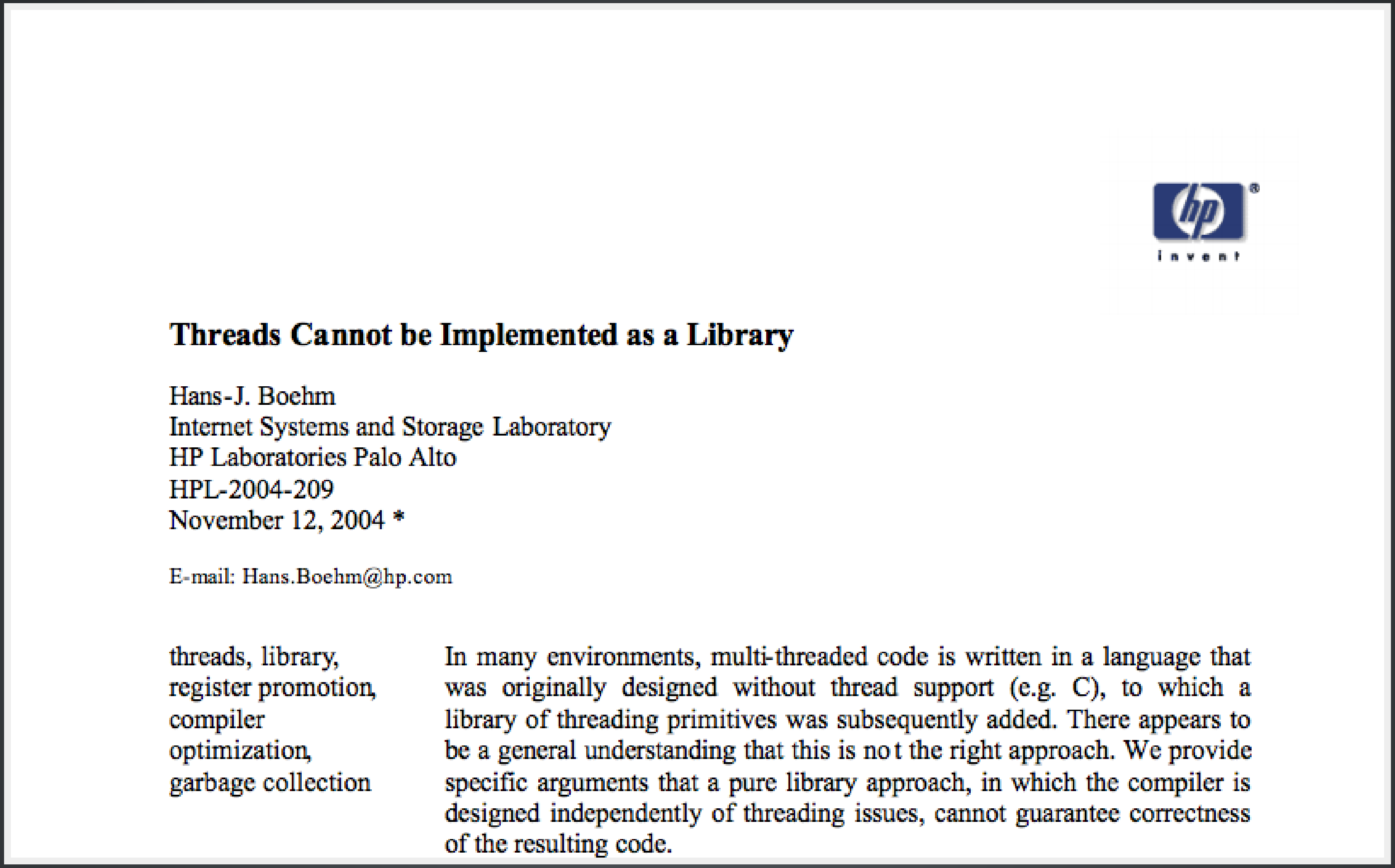
OVER HERE!

THREADING IMPLEMENTED AS A LIBRARY

Parallel processors are becoming more common, but so are amazingly fast single-processors. ... In addition, networking (both WAN and LAN) imposes its own demands. Because of this diversity I recommend parallelism be represented by libraries within C++ rather than as a general language feature. ... It is possible to design concurrency support libraries in C++ that approaches built-in concurrency support in both convenience of use and efficiency.

http://accu.org/index.php/journals/1356, 1993

THREADING CANNOT BE IMPLEMENTED AS A LIBRARY



Threads Cannot be Implemented as a Library

Hans-J. Boehm
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2004-209
November 12, 2004 *

E-mail: Hans.Boehm@hp.com

threads, library, register promotion, compiler optimization, garbage collection

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 no t 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.

RE-ORDERING RULES

- What we sometimes forget...
  - Code is frequently not executed in the order it is written
  - Out-of-order execution invisible to a thread can be visible to external threads
  - Can be reordered by:
    - The compiler
    - The processor
    - The memory subsystem

AS-IF

THIS:

```
void Init() {
_data = 42;
_initialized = true;
}
```

COULD JUST AS EASILY BE REORDERED LIKE THIS:

```
void Init() {
_initialized = true;
_data = 42;
}
```

QUIZ TIME!

```
struct s { char a; char b; } x;
// T1:              // T2:
x.a = 1;            x.b = 1;
```

Can the compiler transform T1 to this?

```
struct s tmp = x;
tmp.a = 1;
x = tmp;
```

YES. COMPILER TRANSFORMATIONS CAN INTRODUCE NEW WRITES.

QUIZ TIME!

```
// T1:                   // T2:
r1=X                     r2=Y
if (r1==1)               if (r2==1)
   Y=1                     X=1
```

Is outcome r1=r2=1 allowed?

YES, BECAUSE OF SPECULATIVE BRANCHING.

AND THAT IS JUST THE BEGINNING

- Hardware:
  - Out of order execution
  - Pipelining
  - hardware threads
  - instruction cache, data cache, store buffer
  - speculative execution and prediction
- Compiler:
  - Re-ordering
  - Register promotion

YIKES!

THE C++ MEMORY MODEL

WHAT IS IT?

- A contract with the machine architects and compiler writer with you
- Data-race-free

THE NEW C++ ABSTRACT MACHINE

- Semantics of concurrent operations at an abstract level
  - how memory reads and writes may be executed by a processor relative to their program order
  - how writes by one processor may become visible to other processors

```
           // Global
           int x, y;
// Thread 1         // Thread 2
x = 17;             cout << y << " ";
y = 37;             cout << x << endl;
```

What does this program output?

- Meaningless question in C++03
- Undefined Behavior in C++11
- ...and this is an improvement!

```
-std=c++11
```

```
// Thread 1:
{
  std::lock_guard<std::mutex> lg(m);
  x = 17;
  y = 37;
}
// Thread 2:
{
  std::lock_guard<std::mutex> lg(m);
  cout << y << " ";
  cout << x << endl;
}
```

- Defined behavior in C++11
- Result can be `'0 0'` or `'37 17'`

BUT IS ADDING MUTEX THE ANSWER?

- Pros:
  - acq/rel enforce ordering
  - familiar
- Cons:
  - deadlocks, livelocks, races
  - coarse-grained
  - sometimes overly restrictive

And wouldn`t it be nice if we could just tag the variable (not everywhere it is used)?

<std::atomic>

```
      // Global
      atomic<int> x, y;

// Thread 1        // Thread 2
x.store(17);       cout << y.load() << " ";
y.store(37);       cout << x.load() << endl;
```

- Defined behavior in C++11
- Result can be `'0 0'`, `'37 17'`, `or` `'0 17'`

SEQUENTIAL CONSISTENCY

(std::memory_order_seq_cst)

```
// Thread 1
x.store(17);
y.store(37);
```

SEQUENTIAL CONSISTENCY

(std::memory_order_seq_cst)

```
// Thread 1
x.store(17, memory_order_seq_cst);
y.store(37, memory_order_seq_cst);
```

- *Atomicity*
- *Ordering*

MEMORY ORDERS

```
namespace std {
    typedef enum memory_order {
        memory_order_relaxed,
        memory_order_consume,
        memory_order_acquire,
        memory_order_release,
        memory_order_acq_rel,
        memory_order_seq_cst
    } memory_order;
}
```

RELAXED ATOMICS

(`std::memory_order_relaxed`)

Now we have:

- *Atomicity*
- *Absolutely no ordering guarantees*

(Still following the as-if rule)

```
            // Global
        atomic<int> x, y;

          // Thread 1
x.store(17,memory_order_relaxed);
y.store(37,memory_order_relaxed);

          // Thread 2
cout << y.load(memory_order_relaxed) << " ";
cout << x.load(memory_order_relaxed) << endl;
```

- Result can be `'0 0'`, `'37 17'`, `'0 17'` or `'37 0'`

RELEASE AND ACQUIRE

```
namespace std {
    typedef enum memory_order {
        memory_order_relaxed,
        memory_order_consume,
        memory_order_acquire,
        memory_order_release,
        memory_order_acq_rel,
        memory_order_seq_cst
    } memory_order;
}
```

RELEASE AND ACQUIRE

(`std::memory_order_release`, `std::memory_order_acquire`)

- Release semantics:
  - prevent memory reordering of preceding operations
- Acquire semantics:
  - prevent memory reordering of following operations
- can move operations prior to locked section into critical section
- can move operations following locked section into critical section
- Establishes a *happens-before* relationship

```cpp
std::atomic<std::string*> ptr;
int data;

void producer()
{
  std::string* p = new std::string("Hello");
  data = 42;
  ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_acquire)))
        ;
    assert(*p2 == "Hello"); // never fails
    assert(data == 42);     // never fails
}
```

RELEASE AND ACQUIRE RELATIONSHIPS

- *sequenced-before* relationship established by:
  - Good ol' fashioned sequence points (ST)
- *happens-before* relationship established by:
  - Program order (in single-threaded context)
  - *synchronizes-with* relationship (in multi-threaded context)
- *synchronizes-with* relationship established by:
  - Mutex lock/unlock
  - Thread create/join
  - Acquire & release semantics (atomic< > and fences)

CONSUME

```
namespace std {
    typedef enum memory_order {
        memory_order_relaxed,
        memory_order_consume,
        memory_order_acquire,
        memory_order_release,
        memory_order_acq_rel,
        memory_order_seq_cst
    } memory_order;
}
```

CONSUME

(`std::memory_order_consume`)

- Limits the synchronized data to direct dependencies
- Establishes a *dependency-ordered-before* relationship

```cpp
std::atomic<std::string*> ptr;
int data;

void producer()
{
    std::string* p  = new std::string("Hello");
    data = 42;
    ptr.store(p, std::memory_order_release);
}

void consumer()
{
    std::string* p2;
    while (!(p2 = ptr.load(std::memory_order_consume)))
        ;
    assert(*p2 == "Hello"); // never fires
    assert(data == 42); // may or may not fire
}
```

CONSUME RELATIONSHIPS

- *carries-a-dependency-to* relationship established by:
  - data dependency between operations (transitive, ST)
- *dependency-ordered-before* relationship established by:
  - Consume semantics

`presentation.join()`

(conclusions)

WEAK ATOMICS ARE TRICKY!

Any time you deviate from sequential consistency, you increase the complexity of the problem by orders of magnitude.

PERHAPS MORE THAN TRICKY?

I had no idea what I was getting myself into when attempting to reason about C++ weak atomics. The theory behind them is so complex that it's borderline unusable. It took three people (Anthony, Hans, and me) and a modification to the Standard to complete the proof of a relatively simple algorithm. Imagine doing the same for a lock-free queue based on weak atomics!

THIS HAS ALWAYS BEEN AVAILABLE

- At a low level
- non-portable
- non-standard way

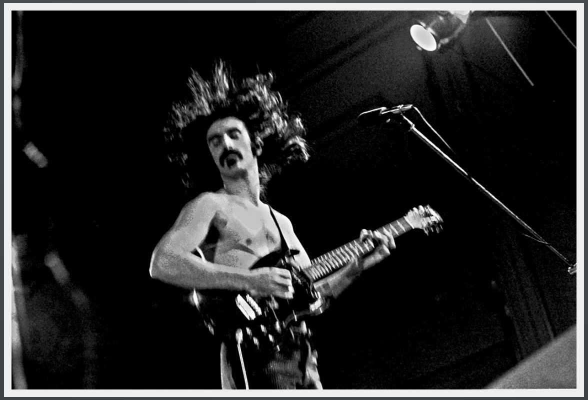
PROGRAMMABILITY vs *PERFORMANCE*

(Stronger guarantees for programmers vs. Greater flexibility to reorder memory operations)

Also affects portability and performance by constraining the transformation that can occur.

THE FREE LUNCH AIN'T DEAD

...IT JUST SMELLS FUNNY



THANKS!

READING LIST

Foundations of the C++ Concurrency Memory Model, Hans-J. Boehm, Sarita V. Adve: http://www.hpl.hp.com/techreports/2008/HPL-2008-56.pdf

std::memory_order explanations: http://en.cppreference.com/w/cpp/atomic/memory_order

Memory barriers in the linux kernel: https://www.kernel.org/doc/Documentation/memory-barriers.txt

Memory orders at compile time: http://preshing.com/20120625/memory-ordering-at-compile-time/

C++ atomics and memory ordering: http://bartoszmilewski.com/2008/12/01/c-atomics-and-memory-ordering/

Speculative memory promotion: http://aggregate.org/LAR/p125-lin.pdf

N1525: Memory-Order Rationale http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1525.htm

CREDITS

http://en.wikipedia.org/wiki/Door_to_Hell http://en.wikipedia.org/wiki/File:Threadsmoviecover.jpg
http://en.wikipedia.org/wiki/File:Handshake_(Workshop_Cologne_%2706).jpeg
http://www.flickr.com/photos/heiner1947/4395120961/in/faves-24788065@N02/