Creating safe multi-threaded applications in C++11

Jos van Eijndhoven
jos@vectorfabrics.com

ACCU conference
Bristol, UK
April 2014
Vector Fabrics’ activities

**PRODUCTS**

- Pareon makes your C/C++ code run faster

**CONSULTANCY**

- Software optimization

**TRAINING**

- Multicore programming training

**Tool development and licensing**

**Consultancy services**

**Training**

in-house and on-site
Vector Fabrics – the company

- Founded February 2007 in Eindhoven, the Netherlands

- Currently 15 FTE: 6 PhD, 7 MSc

- Recognition
  - “Hot Startup” in EE Times Silicon 60 list, since 2011
  - Selected by Gartner as “Cool vendor in Embedded Systems & Software” 2013
  - Global Semiconductors Alliance award, March 2013
You all see the proliferation of multi-core

Galaxy S (2010) 1 processor

Galaxy S2 (2011) 2 cores

Galaxy S3 (2012) 4 cores

Galaxy S4 (2013) 8 cores
Multi-core systems drive programmer awareness

Homogeneous multi-core, hardware cache-coherency, one shared OS kernel:
Industry proven successful combination, long history

- IBM 3084: 4-cpu mainframe (1982)
- Intel Pentium D: dual core single chip (2005)
- Sun Niagara: 8-core single chip (2005)
- ARM Cortex-A9 dual-core on on Nvidia tegra-2 chip (2011)

And more recent on the server side:
- Intel Xeon Phi: 60-core single chip (2012)
- IBM Blue Gene/Q: 1.6M cores, 1.6PB memory (2012)
Moore’s law versus Amdahl’s law

---

### Computational Capacity

- **Hardware capabilities underutilized**
  - **Programming bottleneck**

---

### Software Performance

- **time**

---

### # of transistors vs. time

- **Introduction of multicore technology**

---

- **ACCU conference**
  - April 11, 2014
Creating parallel programs is hard...

Edward A. Lee, EECS professor at U.C. Berkeley:
“Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism.”

Herb Sutter, ISO C++ standards committee, Microsoft:
“Everybody who learns concurrency thinks they understand it, ends up finding mysterious races they thought weren’t possible, and discovers that they didn’t actually understand it yet after all”
Problems, anyone?

Nissan recalls some Infiniti Q50 sedans with steer-by-wire software glitch

Toyota is recalling 1.9 million of its top-selling Prius hybrid cars because of a software fault that may cause the vehicle to slow down suddenly

Nissan recalls some Infiniti Q50 sedans with steer-by-wire software glitch

2013 Ram 1500 Recalled For Stability Control Software Glitch

Recall Roundup: Software Glitches Force Several Recalls
http://autos.jdpower.com/content/blog-post/AuY6uUi/recall-roundup-software-glitches-force-several-recalls.htm

Delhi-bound AI Dreamliner lands in Kuala Lumpur due to software glitch

Volvo recalls 2014 models to correct software glitch
http://uk.reuters.com/article/2013/09/05/uk-autos-volvo-recall-idUKBRE9840U320130905

Bug Sends Space Probe 'Spinning Out of Control,' NASA Says

2013 Ram 1500 Recalled For Stability Control Software Glitch
Multi-threading: non-deterministic behavior

```c
int x=0;

++x;
printf("%d ",x);

++x;
printf("%d ",x);
```

**Quiz:** Without further synchronization, which are valid print-outs according to C (and Java) language semantics?

- 1 1
- 1 2
- 2 1
- 2 2
Learning raises the awareness of complexity

- Provides good insight in C++ concurrency
- C++11 standardizes concurrency primitives
- Warns for *many many* subtle problems

  - The authoritative description (4th edition)
  - Apparently requires 1300+ pages...

- Safe concurrency by defensive design
- Shows that Java shares many concurrency issues with C++
HOWTO: Parallelization of sequential C/C++

- Analyze behavior of sequential program: Establishes *functional reference*, *deterministic* behavior
- Look for loops that provide good opportunity:
  - Contain a significant amount of all work
  - Loop-carried dependencies seem manageable…
- Make an inventory of loop-carried dependencies (group by object, or by class type)
- Do a ‘what if resolved’ performance estimate…
- …maybe for different target architectures
- Verify the correctness of your concurrent implementation
PAREON: performance analysis

Loop-carried dependencies hinder parallel execution of loop iterations

Other performance statistics: iteration counts, cache penalties

View on call tree with relative workload
PAREON: data dependency analysis

Detailed info on loop-carried dependencies: producer & consumer source locations, allocation location, symbol name
PAREON: Schedule data dependencies

Estimate multi-thread fork/join overhead

Obtain a preview on a potential parallelization assume synchronization on complex dependencies
Histogram on execution time per iteration: wide variation is not nice....
WHAT IF application is partially parallelized?

- Some parallelization was done before using Pareon
- Or some parallelization was done on Pareon’s advice, but we want to look for more opportunities…

Tracing load-store dependencies becomes harder!

- Obtaining the inter-thread load-store dependencies is OK, but:
- Actual load-store interleaving over time (mutual ordering) is schedule-dependent (is non-deterministic)
- How to decide whether observed inter-thread data exchange is good or wrong?

**C++11 comes to rescue! 😊**
Pre-11 C/C++ constructs for threading

Three basic primitives, and some OS-level functionality

- Volatile variable declarations:
  force compiler load/store generation, limit compiler re-orderings

- Memory fence operations:
  force load/store ordering at runtime in the memory system

- Atomic operations:
  indivisible read-modify-write (increment, test-and-set)

- Higher-level abstractions (semaphores, condition variables) that include OS and kernel support ➔ thread sleep and wakeup

Only ‘volatile’ is standardized in C/C++. Originally designed for I/O to hardware.

Posix thread library in 1995, fences/atomics are compiler specific intrinsics
Pre-11 C/C++ constructs for threading

Creation of multi-threaded programs:

- The C/C++ compiler performs strong optimizations that are only valid in single-threaded execution mode.
- ‘volatiles’ and ‘fences’ are required, often forgotten, clutter your program, degrade performance beyond need.

This forgotten leads to rarely occurring bugs, which are not reproducible.

And: programs that seemed correct on X86, appear buggy on ARM.
1995-2011 C/C++ constructs for threading

Three basic primitives, and some OS-level functionality

- Volatile variable declarations:
  force compiler load/store generation, limit compiler re-orderings

- Memory fence operations:
  force load/store orderings at runtime in the memory system

- Atomic operations:
  indivisible read-modify-write (increment, test-and-set)

- Higher-level abstractions (semaphores, condition variables) that include OS and kernel support ➔ thread sleep and wakeup

Get rid of all of this 15 years of programming practice

**bold** move by the C++11 committee!
C11/C++11 parallel programming

Creation of multi-threaded programs:

- The C/C++ compiler will always assume multi-threaded access to variables with global scope. This inhibits some optimizations. (C++11 has no ‘volatile’ to denote inter-thread data exchange)

- **Atomic** operations are overloaded with **memory fence** behaviors. These are the basic building blocks for inter-thread synchronization.

If the programmer creates **SC-DRF** programs, then the system ensures correct (deterministic) behavior!

Sequently Consistent Data Race Free

**Finally:** multi-threaded behavior is properly specified for C/C++ !!
Sufficient condition to satisfy ‘Data Race Free’:

Whenever a variable is accessed by operations from two threads:
- Both operations are loads -or-
- Both are executed in a well-defined order

Inter-thread order requires explicit memory barriers.

Carefully chosen barrier semantics should limit performance penalties: impose weak ordering constraints
Building ordering relations

Local order relations allow to extract global ordering (transitive closures)
If you want to learn more...

atomic<> Weapons
The C++11 Memory Model and Modern Hardware
Herb Sutter
3hr presentation at “C++ and Beyond”, Aug. 2012
Example: ping-pong buffer

```cpp
std::atomic<int> flag;
int bucket;

void consume() { // thread A
    while (true) {
        while (!flag.load(std::memory_order_acquire))
            ; // busy wait
        int my_work = bucket;
        flag.store(0, std::memory_order_release);
        consume_stuff(my_work);
    }
}

void produce() { // thread B
    while (true) {
        int my_stuff = produce_stuff();
        while (flag.load(std::memory_order_acquire))
            ; // busy wait
        bucket = mystuff;
        flag.store(1, std::memory_order_release);
    }
}
```
Example: ping-pong buffer

```cpp
std::atomic<int> flag;
int bucket;

void consume() { // thread A
    while (true) {
        while (!flag.load(std::memory_order_acquire)) { // busy wait
            int my_work = bucket;
            flag.store(0, std::memory_order_release);
            consume_stuff(my_work);
        }
    }
}

void produce() { // thread B
    while (true) {
        int my_stuff = produce_stuff();
        while (flag.load(std::memory_order_acquire)) { // busy wait
            bucket = mystuff;
        }
        flag.store(1, std::memory_order_release);
    }
}
```

Ordered data dependency (RaW)
Example: ping-pong buffer

```cpp
std::atomic<int> flag;
int bucket;

void consume() { // thread A
    while (true) {
        while (!flag.load(std::memory_order_acquire))
            ; // busy wait
        int my_work = bucket;
        flag.store(0, std::memory_order_release);
        consume_stuff(my_work);
    }
}

void produce() { // thread B
    while (true) {
        int my_stuff = produce_stuff();
        while (flag.load(std::memory_order_acquire))
            ; // busy wait
        bucket = mystuff;
        flag.store(1, std::memory_order_release);
    }
}
```

Ordered data dependency (RaW)
Ordered anti dependency (WaR)
Learn from this simple example

- Such low-level synchronization is still **hard and error-prone**.
  - You should **re-use higher-level functionality** offered through libraries.
  - Have clear semantics through well-known design patterns

- Checking for SC-DRF should be a tool responsibility. But, we are **not there yet**...
Example with datarace (BAD!)

```cpp
int main()
{
    // create an empty bucket
    std::set<int> bucket;

    // Use a background task to insert value '5' in the bucket
    std::thread t([&](){ bucket.insert(5); });

    // Check if value '3' is in the bucket (not expected :-)
    bool contains3 = bucket.find(3) != bucket.cend();
    std::cout << "Foreground find: " << contains3 << std::endl;

    // Wait for the background thread to finish
    t.join();

    // verify that value '5' did arrive in the bucket
    bool contains5 = bucket.find(5) != bucket.cend();
    std::cout << "Background: " << contains5 << std::endl;

    return 0;
}
```
C++ STL containers are not thread-safe for write access! Programmers would know to not create such code if they read their documentation.

If your job is to create concurrency in an existing large code base (>100K lines), code inspection would easily overlook this (the read and write could be far apart, in different files).

The program seems to run fine: the bug reveals itself rarely.

Today’s data-race checking tools seem to miss this one.
Conclusion

- C++11 obtained a properly defined memory model and threading primitives, finally allowing to create portable programs!
- Bold change: Atomics and volatile became totally different. Some compiler optimizations are now illegal.
- Creating deterministic (SC-DRF) programs remains challenging.

- The programmer community needs more and better tools to improve productivity and bridge the gap with multi-core hardware
Thank you

Check www.vectorfabrics.com for a free demo on concurrency analysis