Creating safe multi-threaded applications in C++11

Jos van Eijndhoven
jos@vectorfabrics.com

ACCU conference
Bristol, UK
April 2014

Vector Fabrics

Vector Fabrics' activities

PRODUCTS

Pareon makes your C/C++ code run faster

PAREON OVERVIEW >

CONSULTANCY

Software optimization

CHECK OUT OUR EXPERTISE >

TRAINING

Multicore programming training

SEE OUR TRAINING PROGRAM >

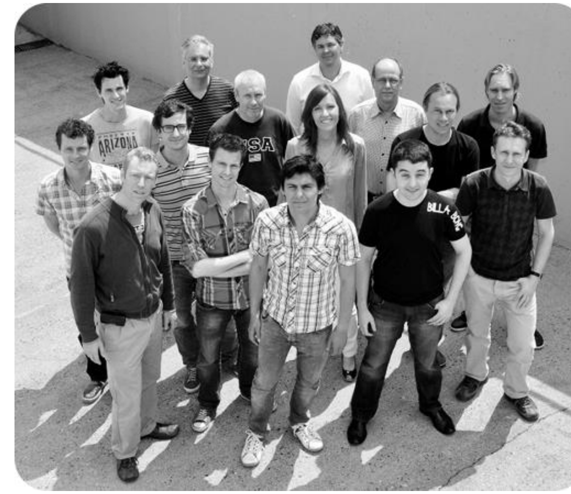
Tool development and licensing

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)	Galaxy S2 (2011)	Galaxy S3 (2012)	Galaxy S4 (2013)
1 processor	**2** cores	**4** cores	**8** cores

Multi-core systems drive programmer awareness

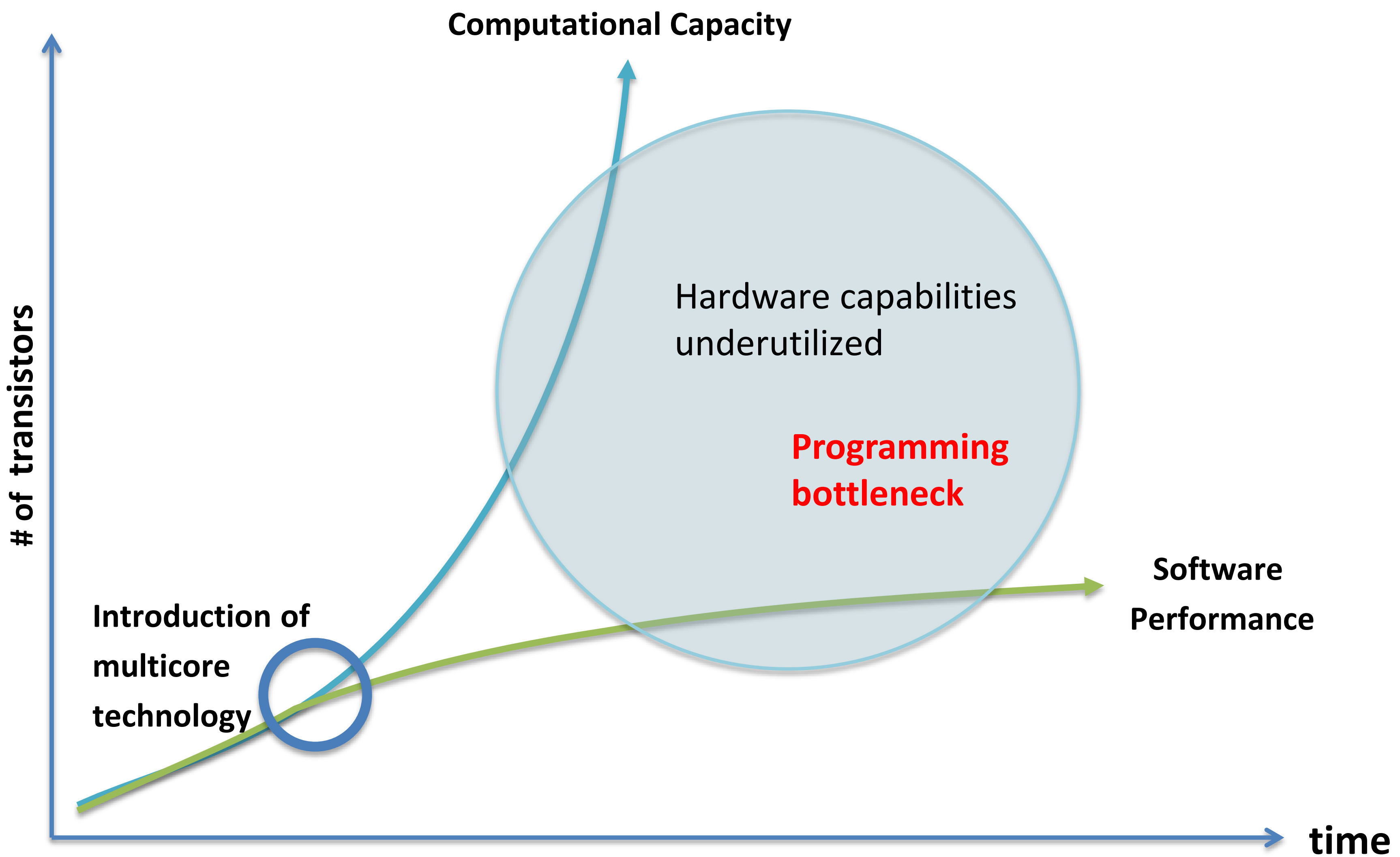
Homogeneous multi-core, hardware cache-coherency, one shared OS kernel:
Industry proven successful combination, long history

- IBM 3084: 4-cpu mainframe (1982)
- Silicon Graphics ‘Origin’: 1024-cpu supercomputer (2000)
- 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



Creating parallel programs is hard…

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”

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.”

Problems, anyone?

Nissan recalls some Infiniti Q50 sedans with steer-by-wire software glitch
http://www.autonews.com/article/20131216/RETAIL05/131219890/nissan-recalls-some-infiniti-q50-sedans-with-steer-by-wire-software#

2013 Ram 1500 Recalled For Stability Control Software Glitch
http://www.thecarconnection.com/news/1085613_2013-ram-1500-recalled-for- ... ol-software-glitch

Delhi-bound AI Dreamliner lands in Kuala Lumpur due to software glitch
http://ibnlive.in.com/news/delhibound-ai-dreamliner-lands-in-kuala-lumpur-due-to-software-glitch/450184-2.html

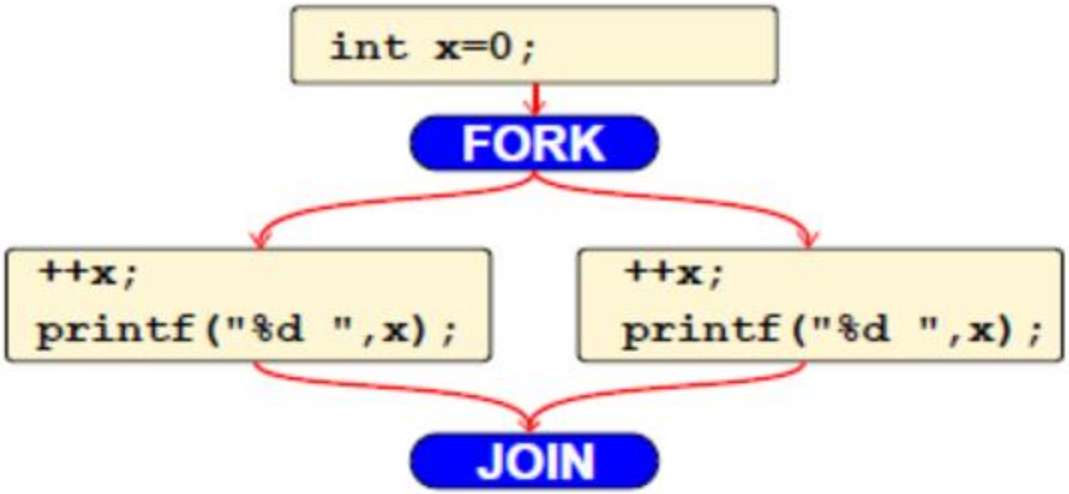
Toyota is recalling 1.9 million of its top-selling Prius hybrid cars because of a software fault that may cause the vehicle to slow down suddenly
http://www.bbc.com/news/business-26148711

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

Bug Sends Space Probe 'Spinning Out of Control,' NASA Says
http://www.weather.com/news/science/space/deep-impact-spacecraft-20130910

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

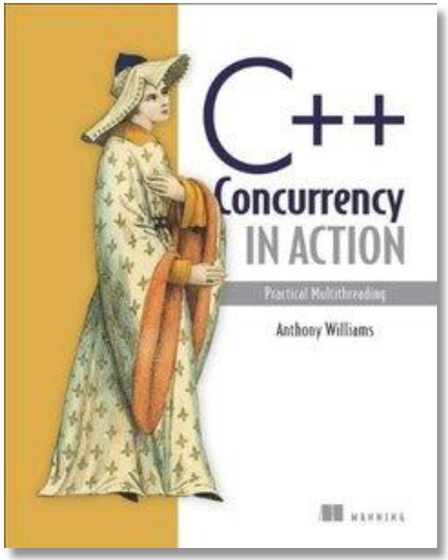
Multi-threading: non-deterministic behavior



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

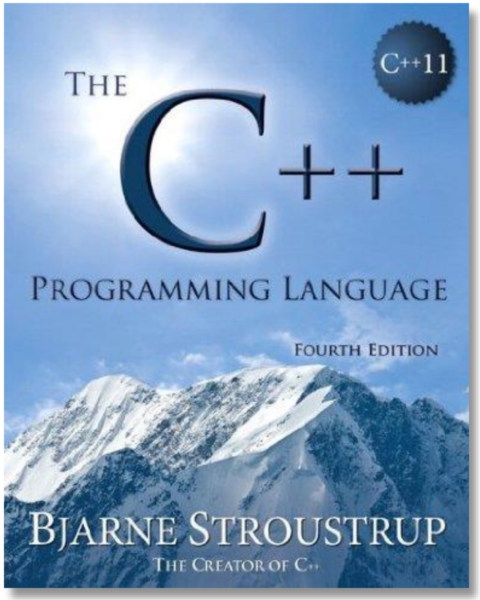
- 1 1
- 1 2
- 2 1
- 2 2

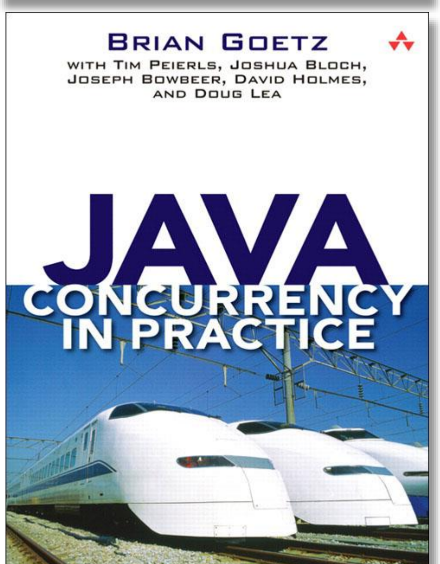
Learning raises the awareness of complexity



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

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



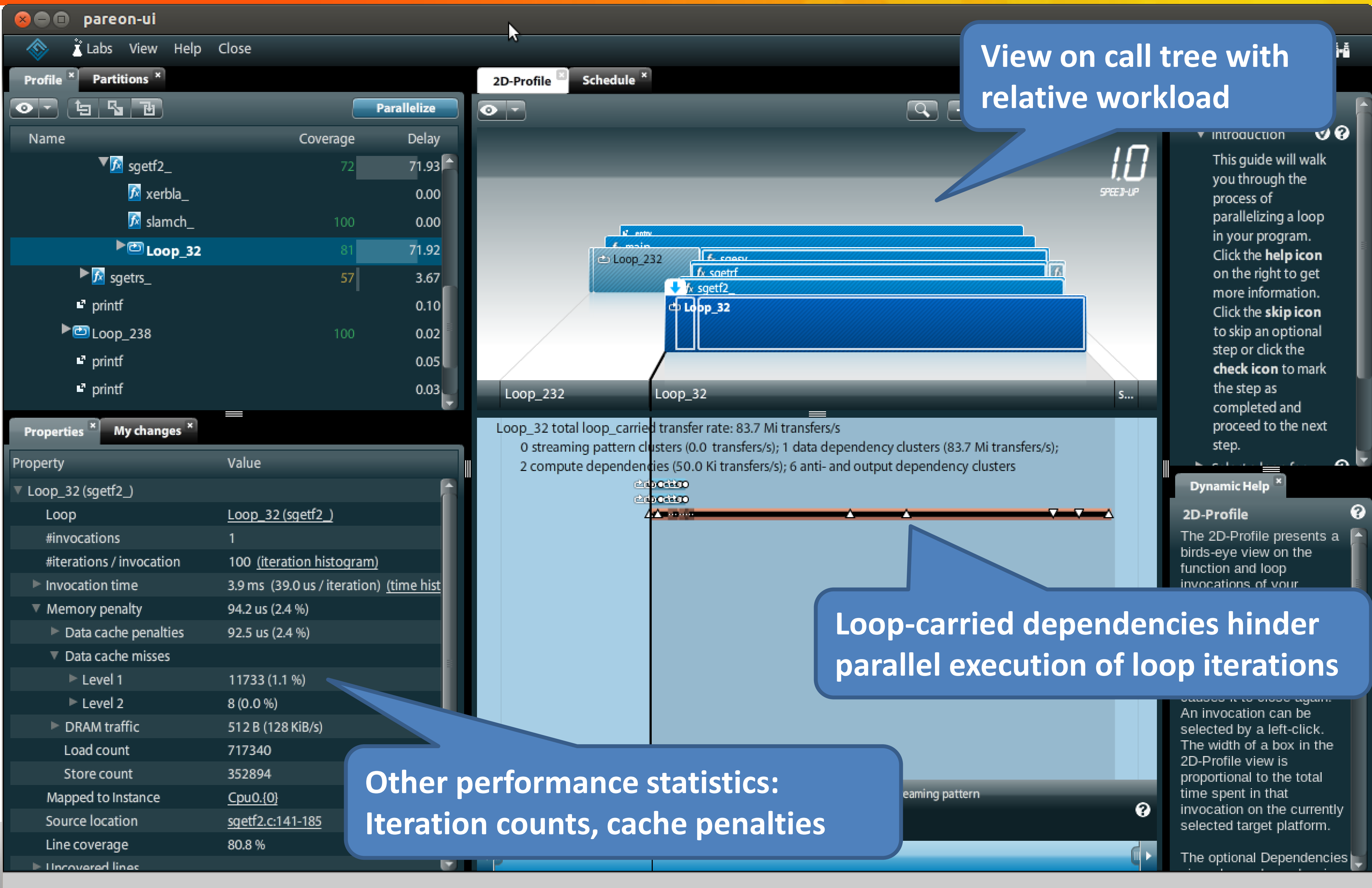


- Safe concurrency by defensive design
- Shows that Java shares many concurrency issues with C++

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

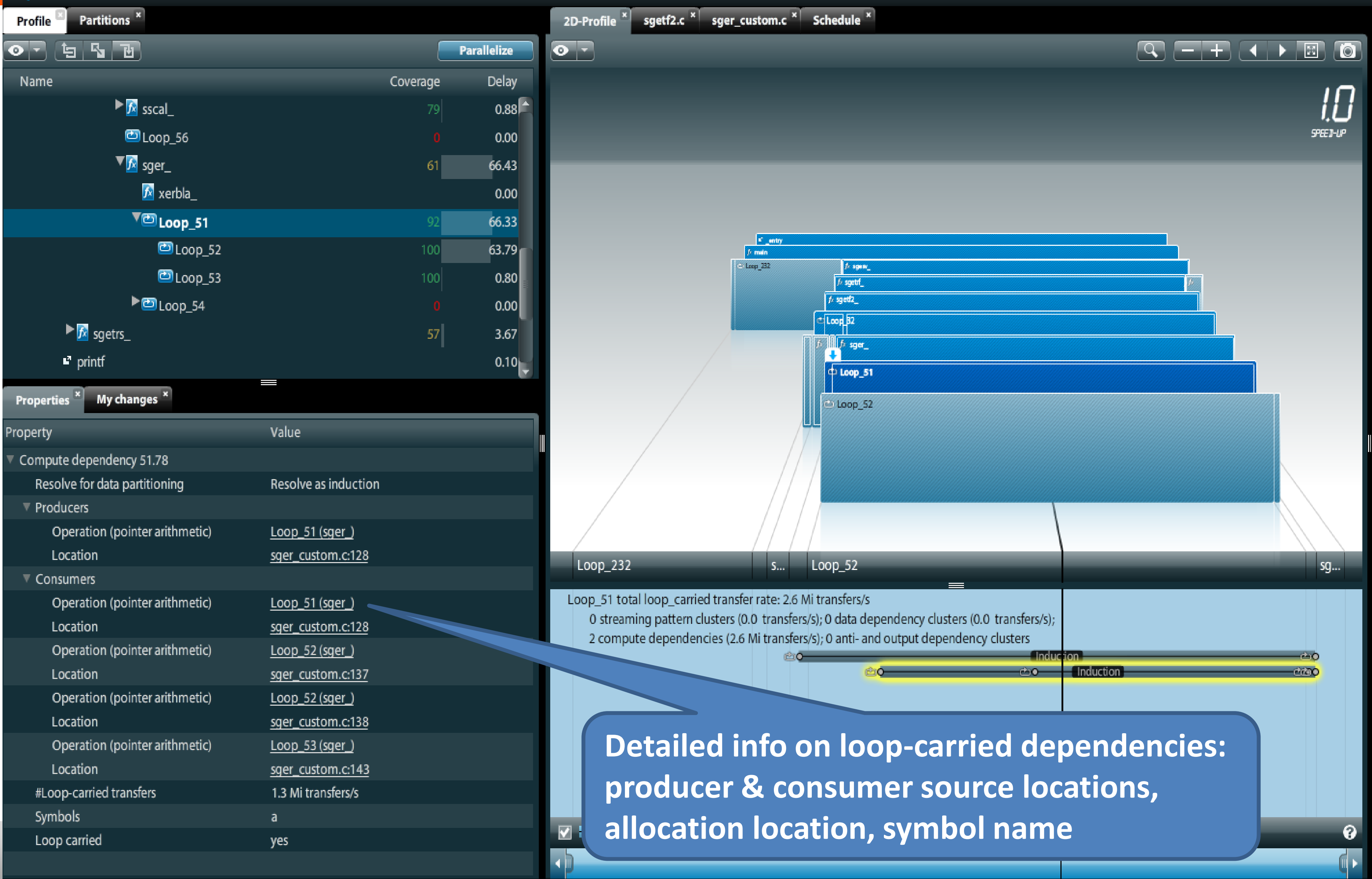


View on call tree with relative workload

Loop-carried dependencies hinder parallel execution of loop iterations

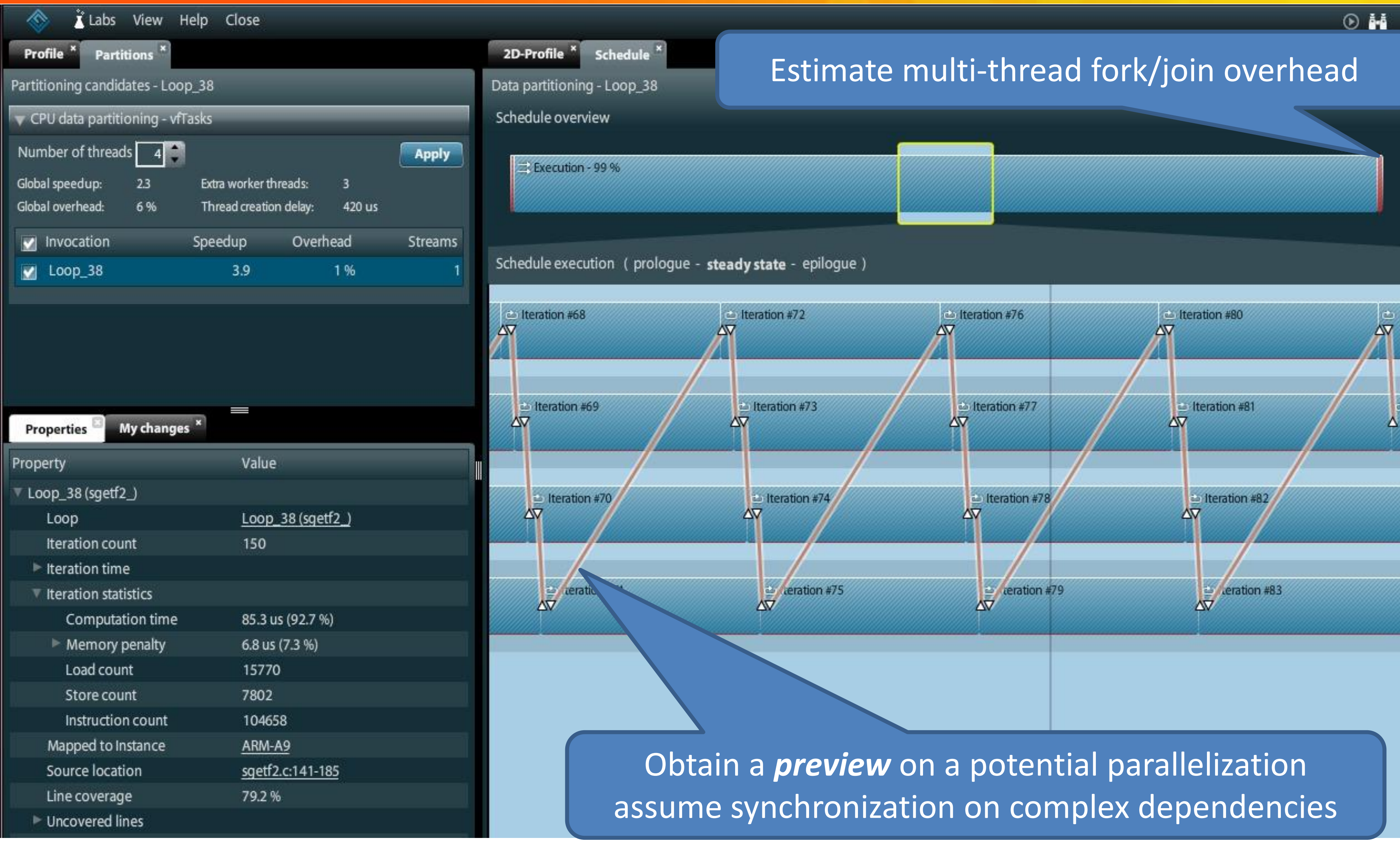
Other performance statistics: Iteration counts, cache penalties

PAREON: data dependency analysis



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

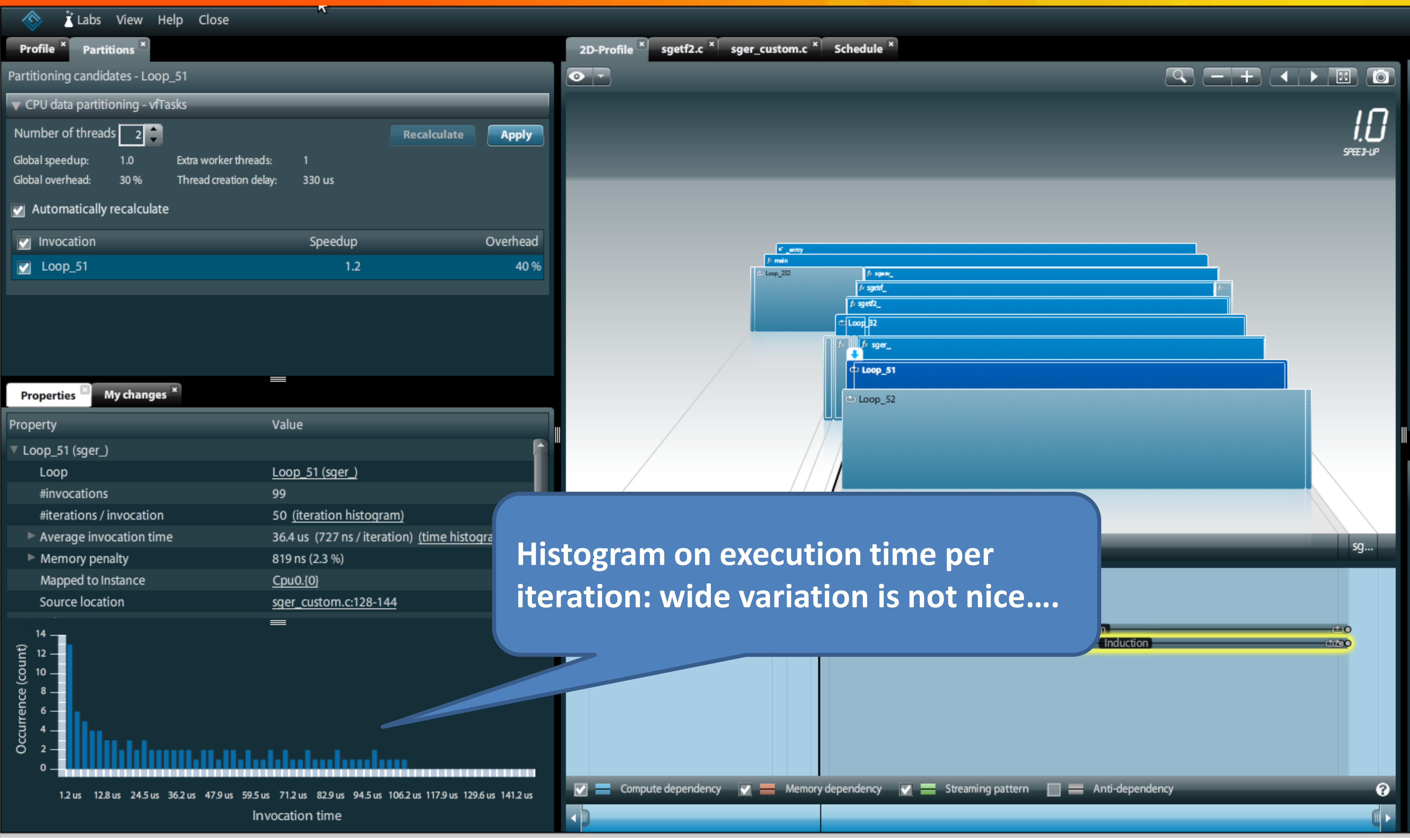
PAREON: Schedule data dependencies



Estimate multi-thread fork/join overhead

Obtain a ***preview*** on a potential parallelization assume synchronization on complex dependencies

PAREON: Loop statistics



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 var... declarations:
  force compiler ...d/store gener...n, limit compiler re-orderings
- Memory fence ope...ns:
  force load/store orderi... runtime in the memory system
- Atomic operations:
  indivisible read-...ify-write (i...ment, test-and-set)
- Higher-level ...stractions (semapho... condition variables) that
  include O... and kernel support ➔ thre... 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!

Sequentially Consistent Data Race Free

Finally: multi-threaded behavior is properly specified for C/C++ !!

Satisfy *Data Race Free*

Sufficient condition to satisfy ‘Data Race Free’:

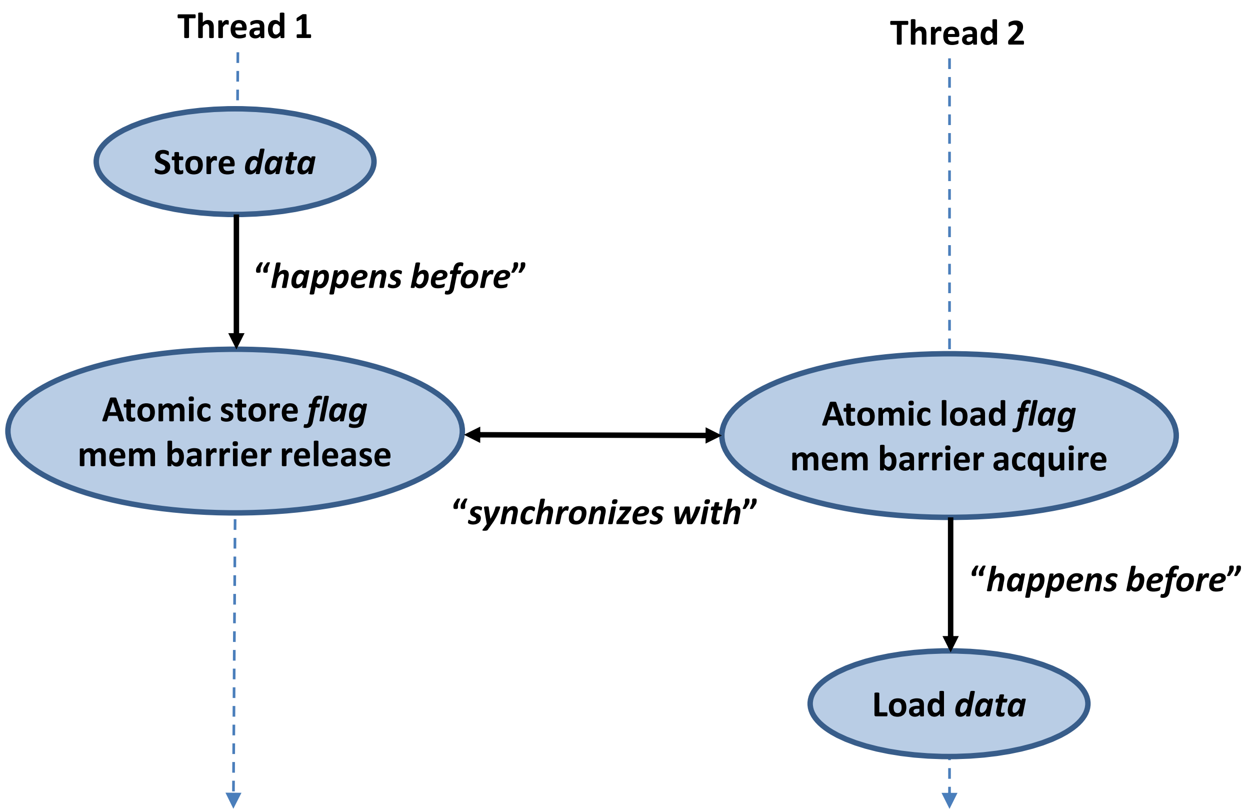
Whenever a variable is accessed by operations from two threads:

- Both operations are loads -or-
- Both are executed in a well-defined order

Inter-thread order requires explicit memory barriers.

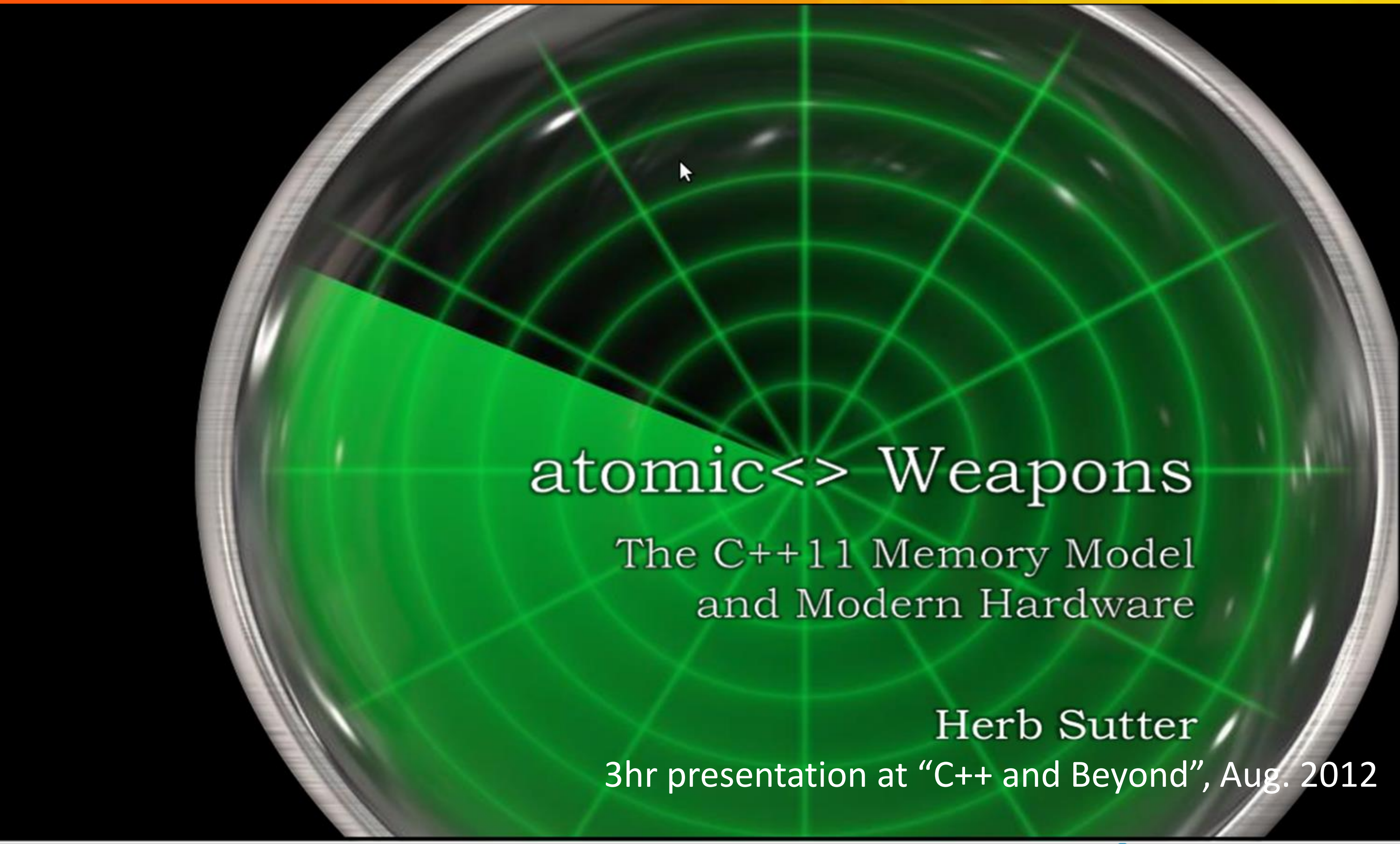
Carefully chosen barrier semantics should limit performance penalties: impose weak ordering constraints

Building ordering relations



Local order relations allow to extract global ordering (transitive closures)

If you want to learn more…



3hr presentation at “C++ and Beyond”, Aug. 2012

Example: ping-pong buffer

```
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

```
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

```
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 anti dependency (WaR)

(?)

Ordered data dependency (RaW)

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!)

```
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;
}
```

Issues with faulty std::set example

- 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

Vector Fabrics