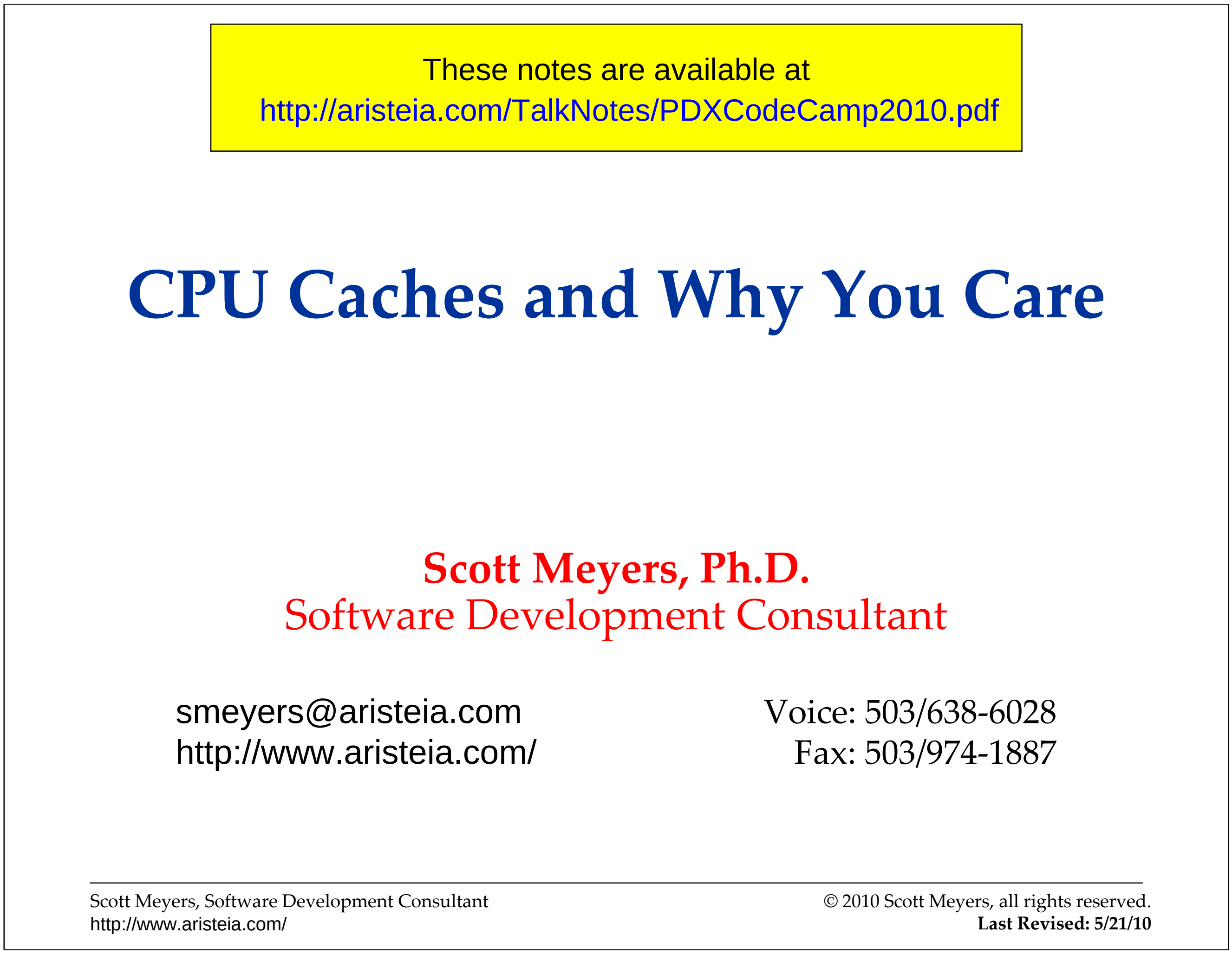


These notes are available at
http://aristeia.com/TalkNotes/PDXCodeCamp2010.pdf

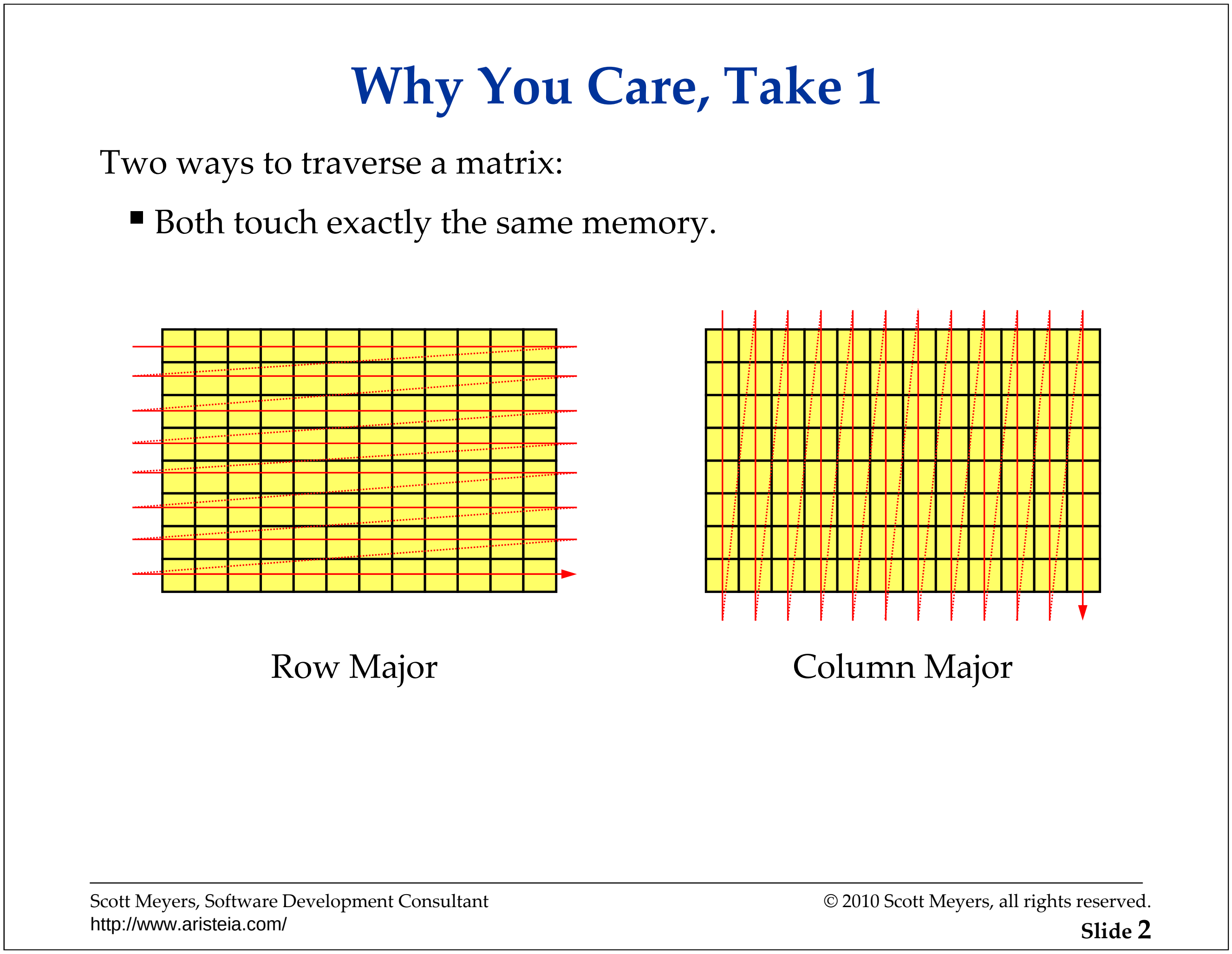
CPU Caches and Why You Care

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/974-1887

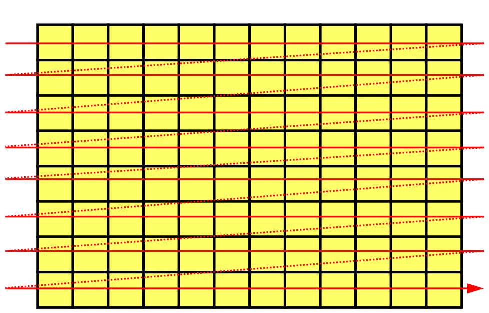
Last Revised: 5/21/10



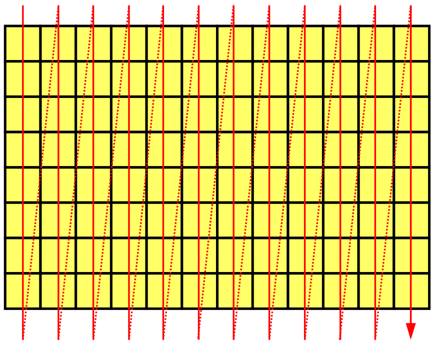
Why You Care, Take 1

Two ways to traverse a matrix:

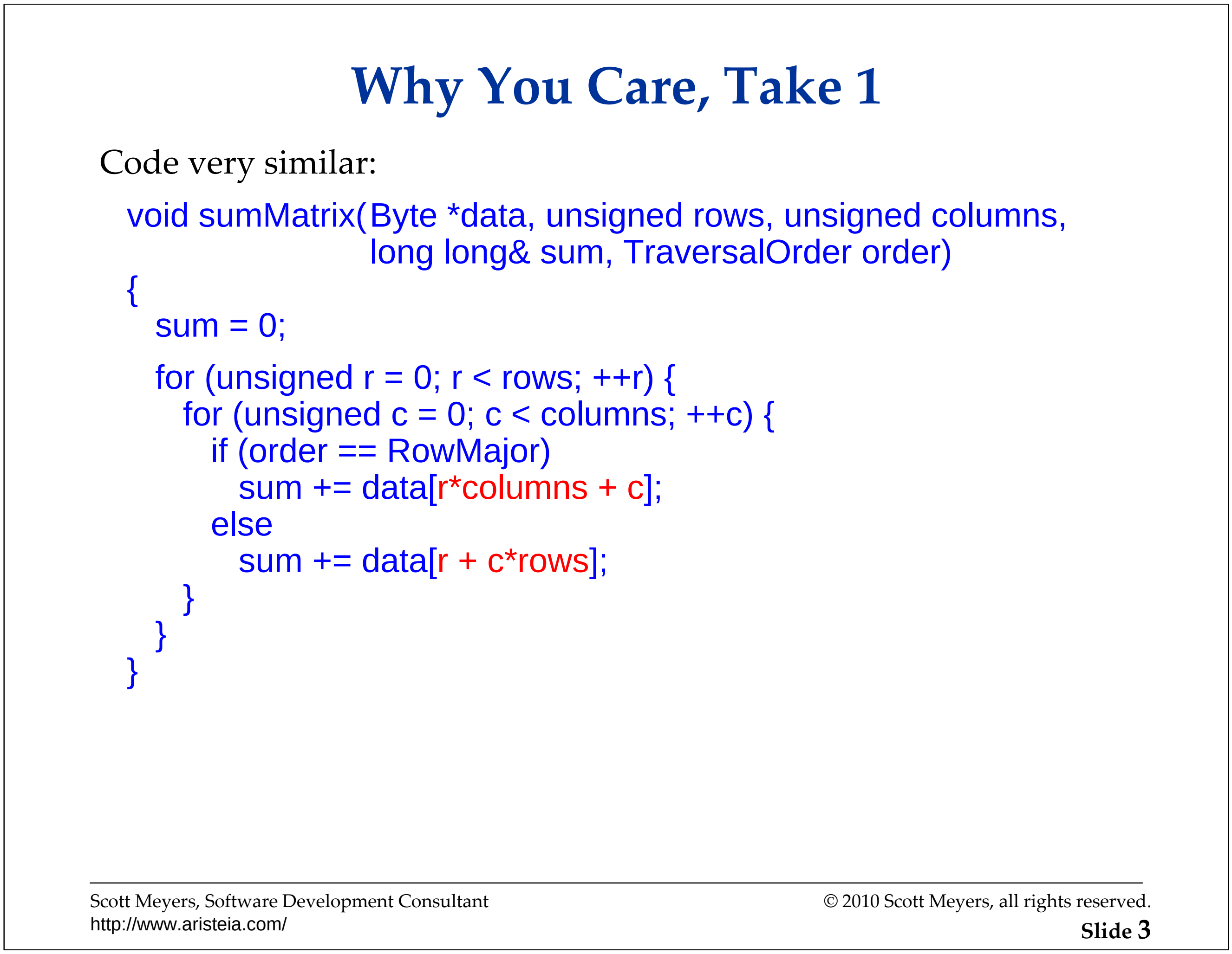
- Both touch exactly the same memory.



Row Major



Column Major

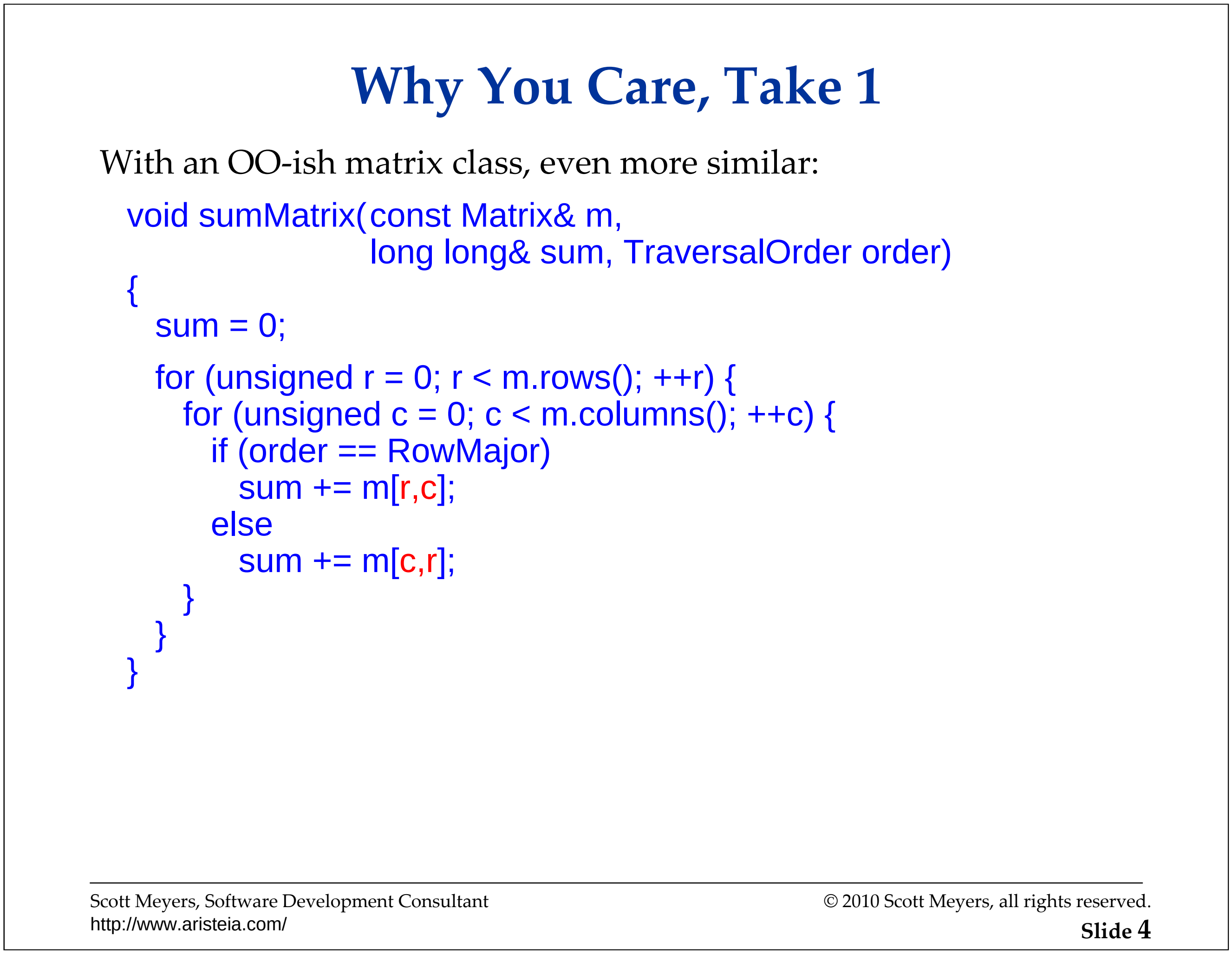


Why You Care, Take 1

Code very similar:

```
void sumMatrix(Byte *data, unsigned rows, unsigned columns,
               long long& sum, TraversalOrder order)
{
  sum = 0;

  for (unsigned r = 0; r < rows; ++r) {
    for (unsigned c = 0; c < columns; ++c) {
      if (order == RowMajor)
        sum += data[r*columns + c];
      else
        sum += data[r + c*rows];
    }
  }
}
```

Actual C++ code I tested.



Why You Care, Take 1

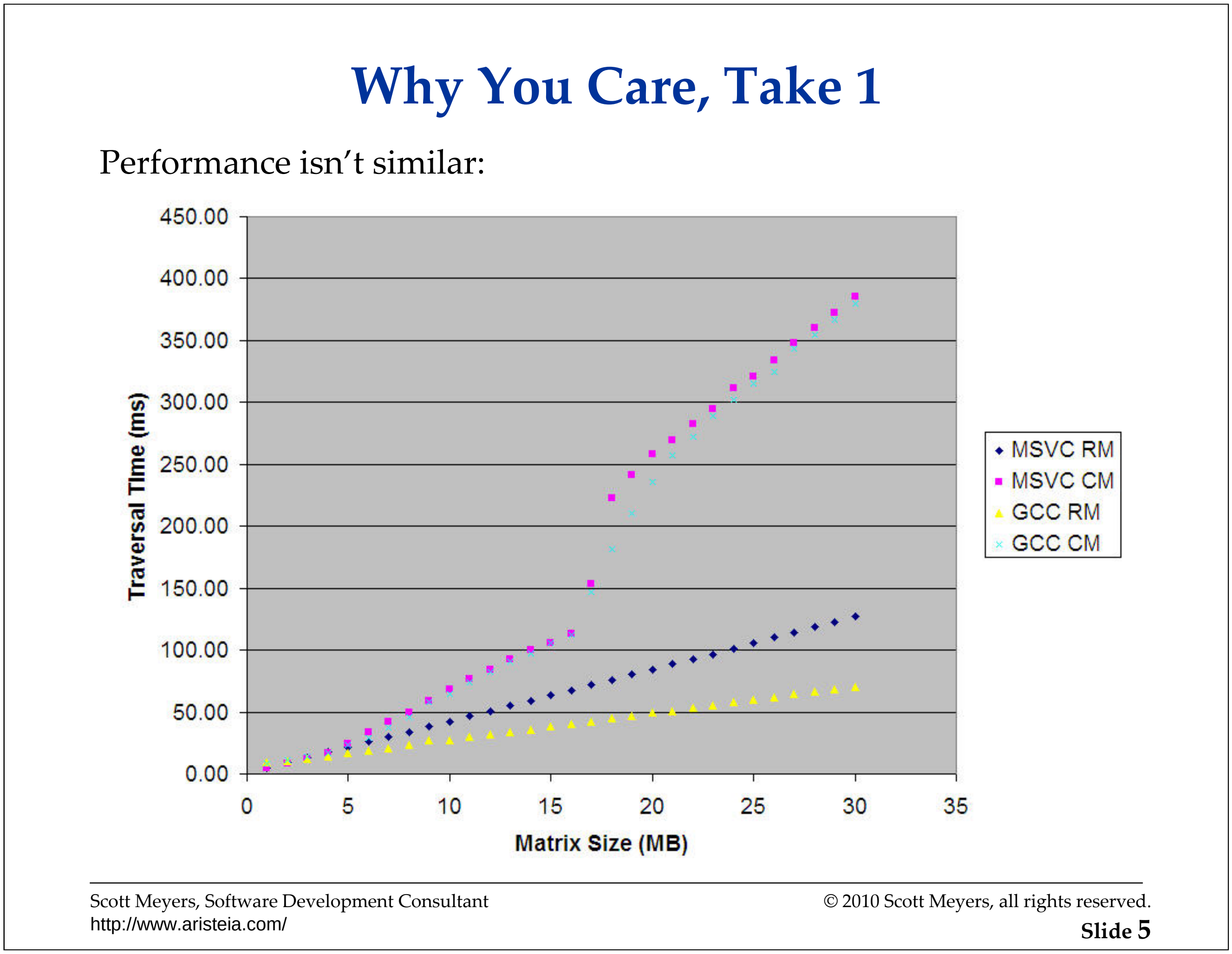
With an OO-ish matrix class, even more similar:

```
void sumMatrix(const Matrix& m,
               long long& sum, TraversalOrder order)
{
  sum = 0;

  for (unsigned r = 0; r < m.rows(); ++r) {
    for (unsigned c = 0; c < m.columns(); ++c) {
      if (order == RowMajor)
        sum += m[r,c];
      else
        sum += m[c,r];
    }
  }
}
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2010 Scott Meyers, all rights reserved.
Slide 4

Code I imagined (i.e., did not test).



Why You Care, Take 1

Performance isn't similar:

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

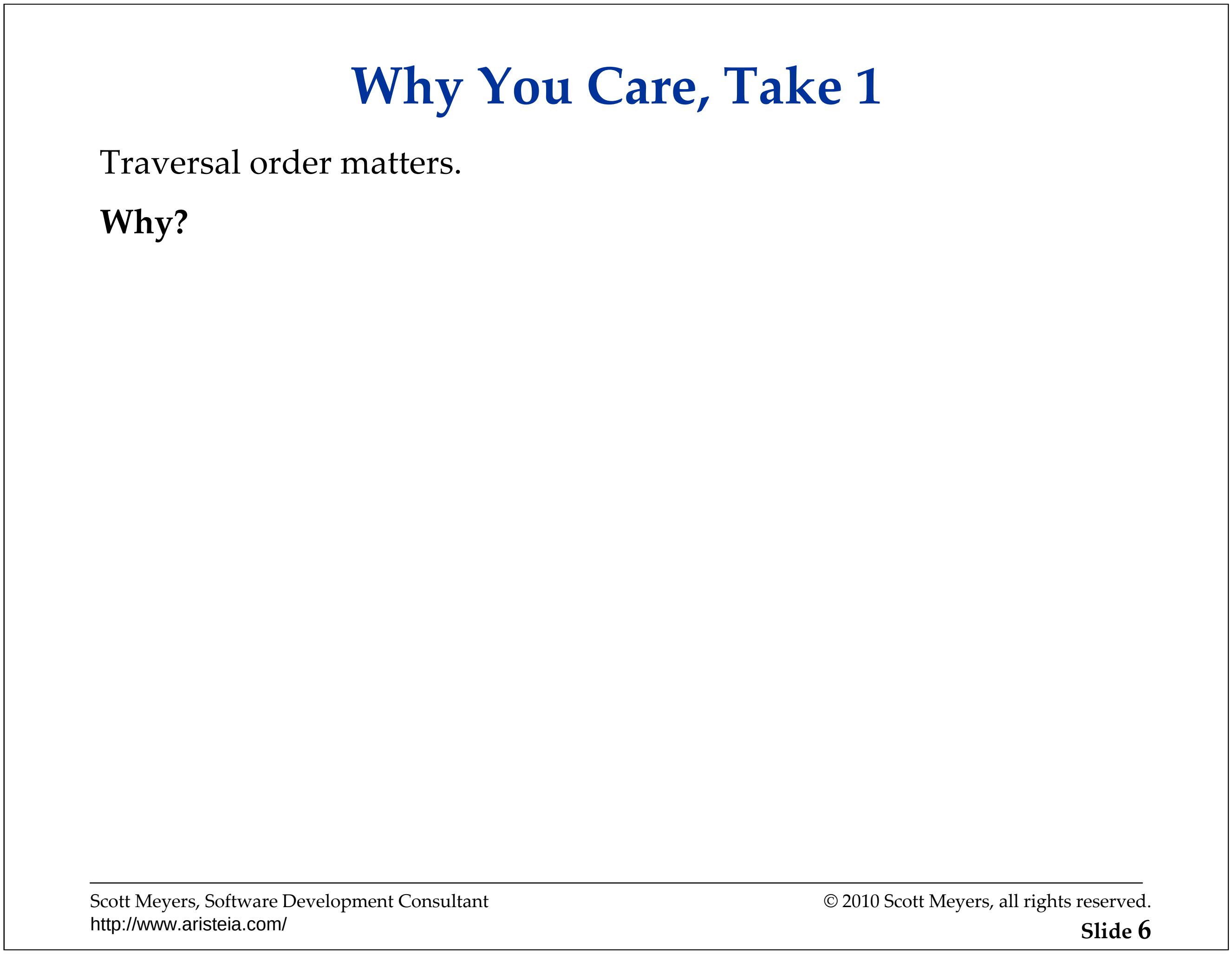
© 2010 Scott Meyers, all rights reserved.

Slide 5

Results are from tests run on my laptop, which has an Intel Core 2 Duo inside.

The row major traversals scale linearly, I assume, because the hardware prefetching is keeping up with the traversal. Why the column major curve breaks around 16MB of array size instead of 4MB (the size of my L2 cache), I don't know.

All my performance numbers are suspect, because I didn't examine the object code to ensure that compilers weren't doing something unexpected that would throw off my results.



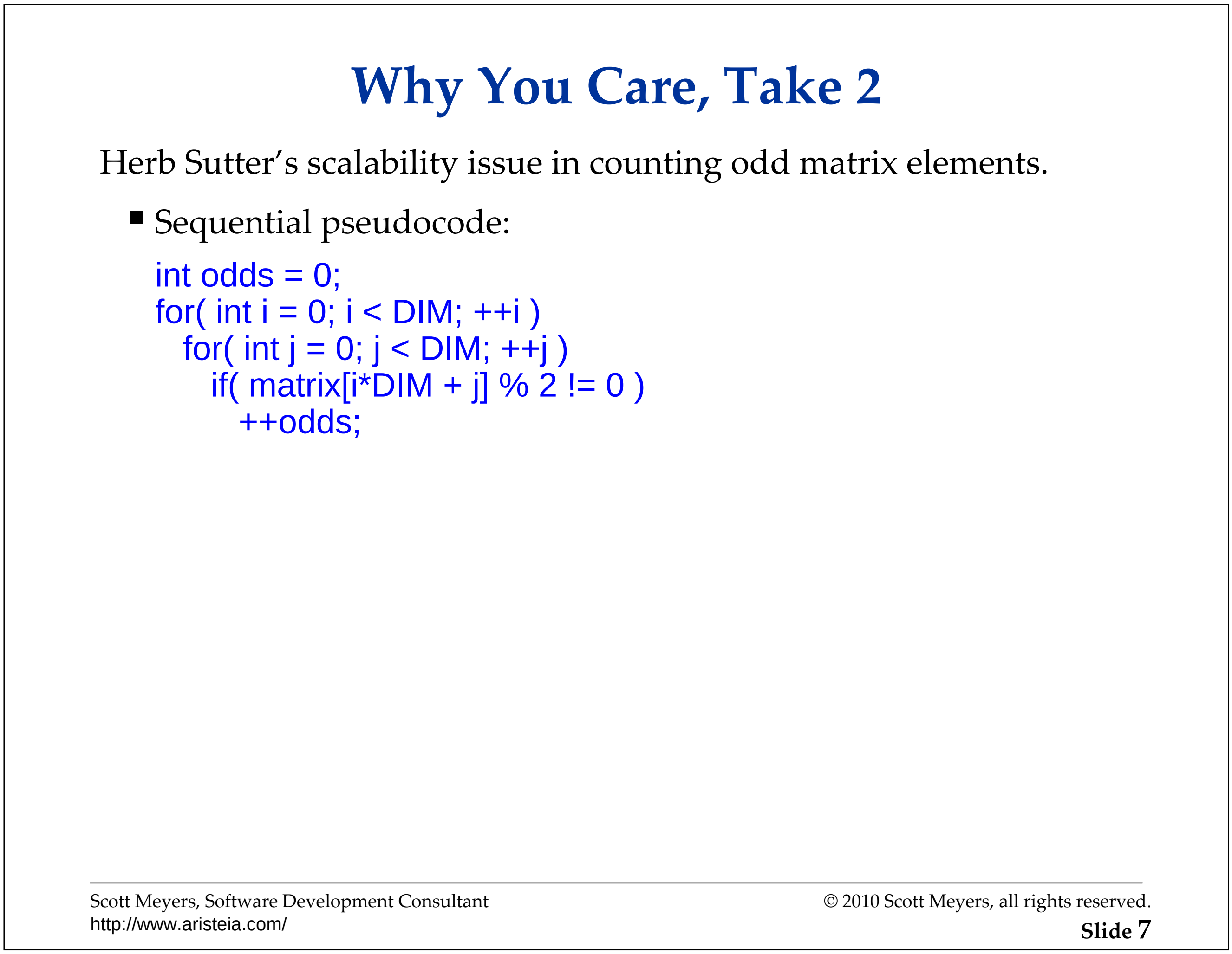
Why You Care, Take 1

Traversal order matters.

Why?

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2010 Scott Meyers, all rights reserved.
Slide 6

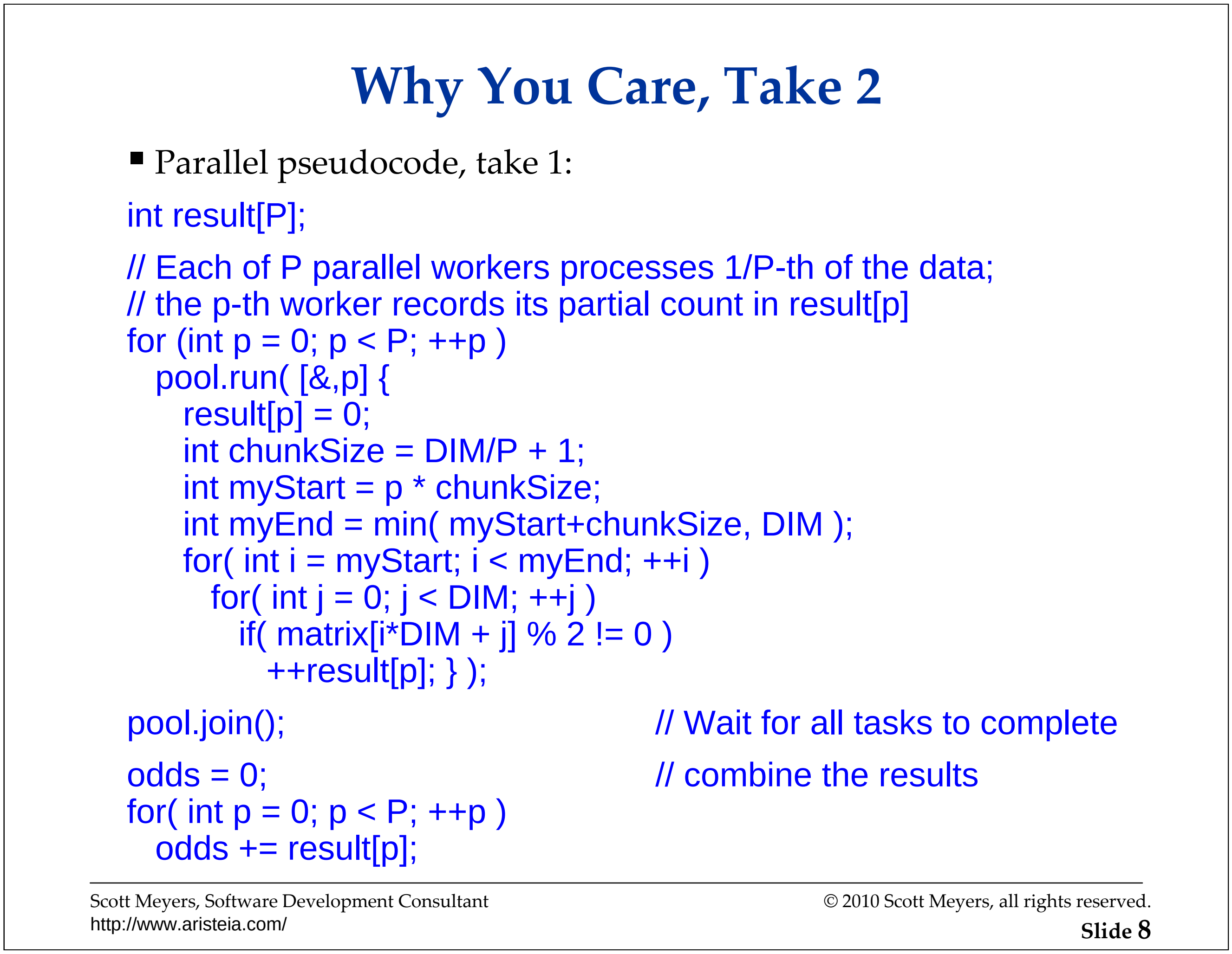


Why You Care, Take 2

Herb Sutter’s scalability issue in counting odd matrix elements.

- Sequential pseudocode:

```
int odds = 0;
for( int i = 0; i < DIM; ++i )
  for( int j = 0; j < DIM; ++j )
    if( matrix[i*DIM + j] % 2 != 0 )
      ++odds;
```



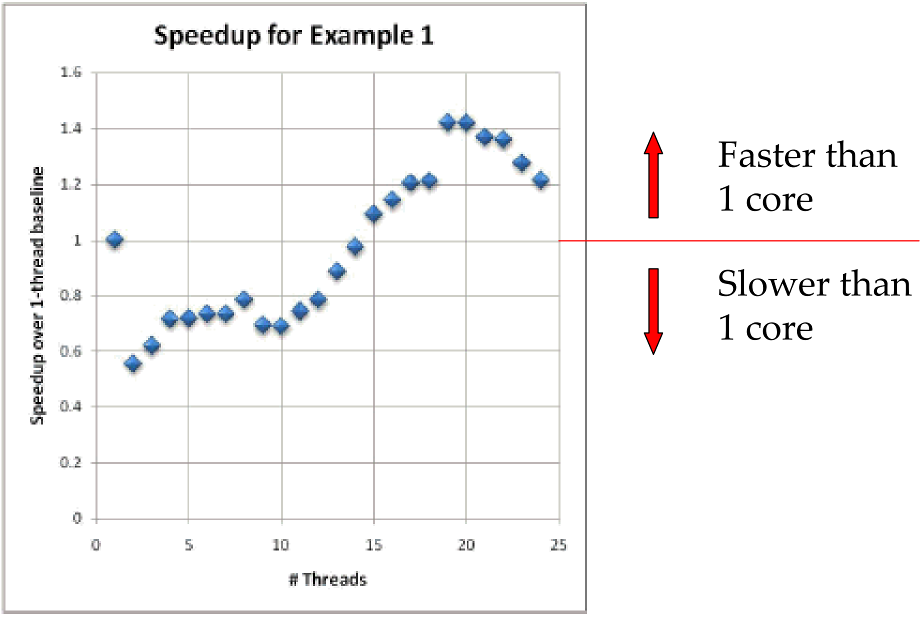
Why You Care, Take 2

- Parallel pseudocode, take 1:

```
int result[P];

// Each of P parallel workers processes 1/P-th of the data;
// the p-th worker records its partial count in result[p]
for (int p = 0; p < P; ++p )
   pool.run( [&,p] {
      result[p] = 0;
      int chunkSize = DIM/P + 1;
      int myStart = p * chunkSize;
      int myEnd = min( myStart+chunkSize, DIM );
      for( int i = myStart; i < myEnd; ++i )
         for( int j = 0; j < DIM; ++j )
            if( matrix[i*DIM + j] % 2 != 0 )
               ++result[p]; } );

pool.join();                                    // Wait for all tasks to complete

odds = 0;                                       // combine the results
for( int p = 0; p < P; ++p )
   odds += result[p];
```

Why You Care, Take 2

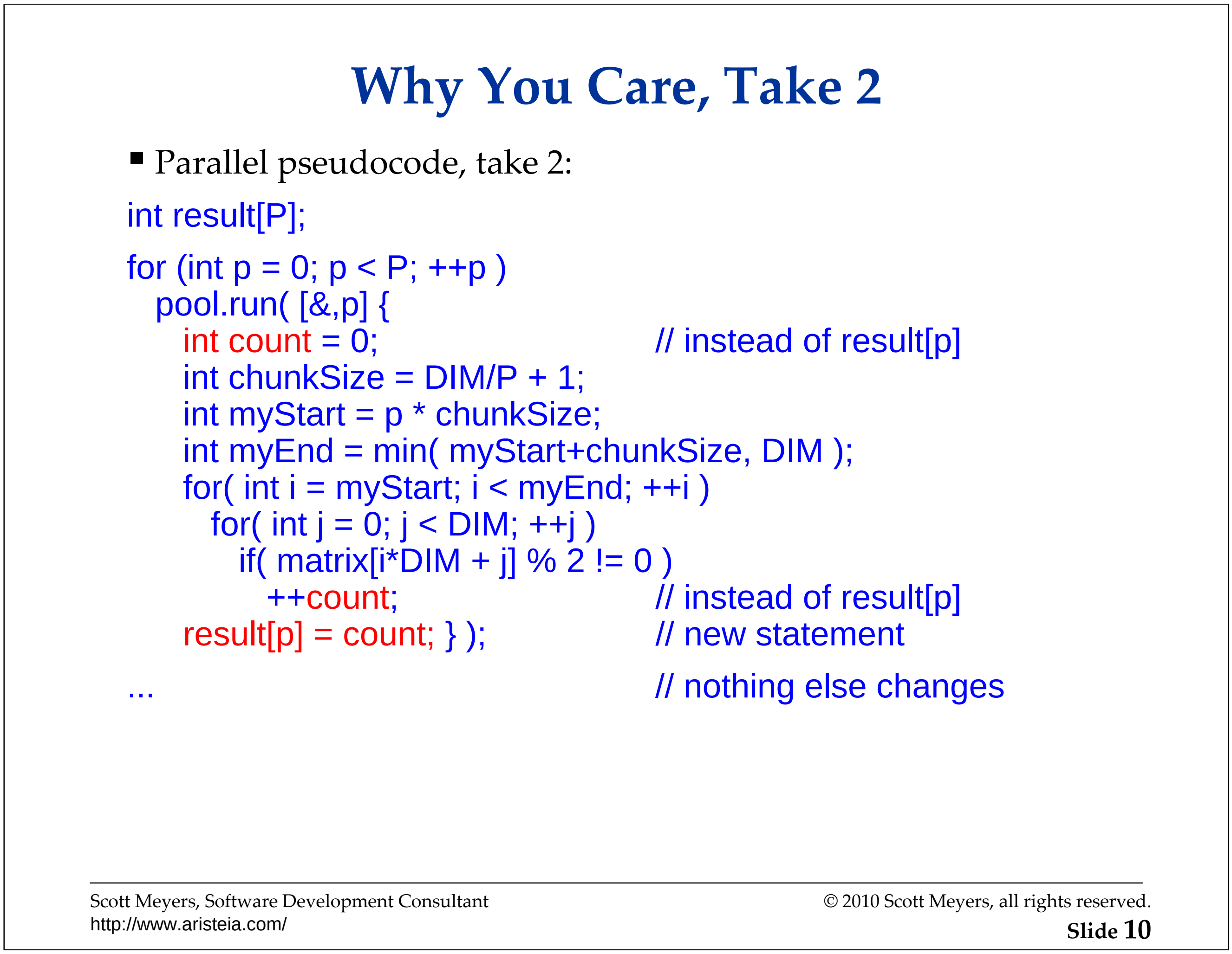
Scalability, er, unimpressive:



Faster than 1 core

Slower than 1 core

These data from a machine with 24 hardware threads. Note that the best speedup (at ~19-20 hardware threads) is only about 40% above that of a single processor!



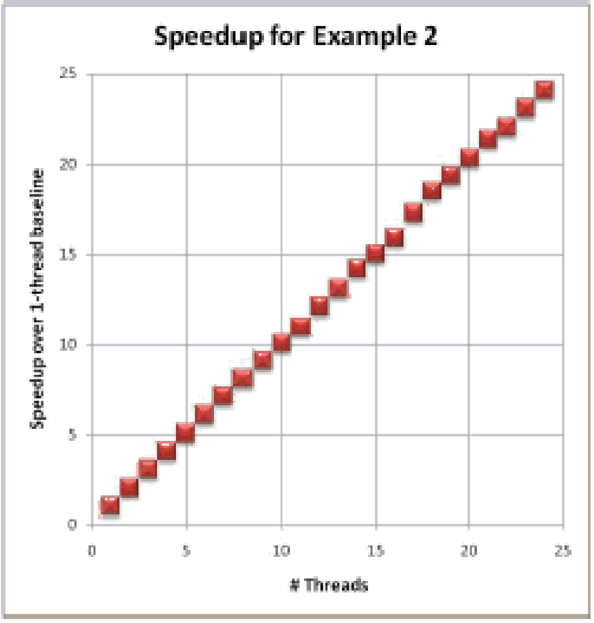
Why You Care, Take 2

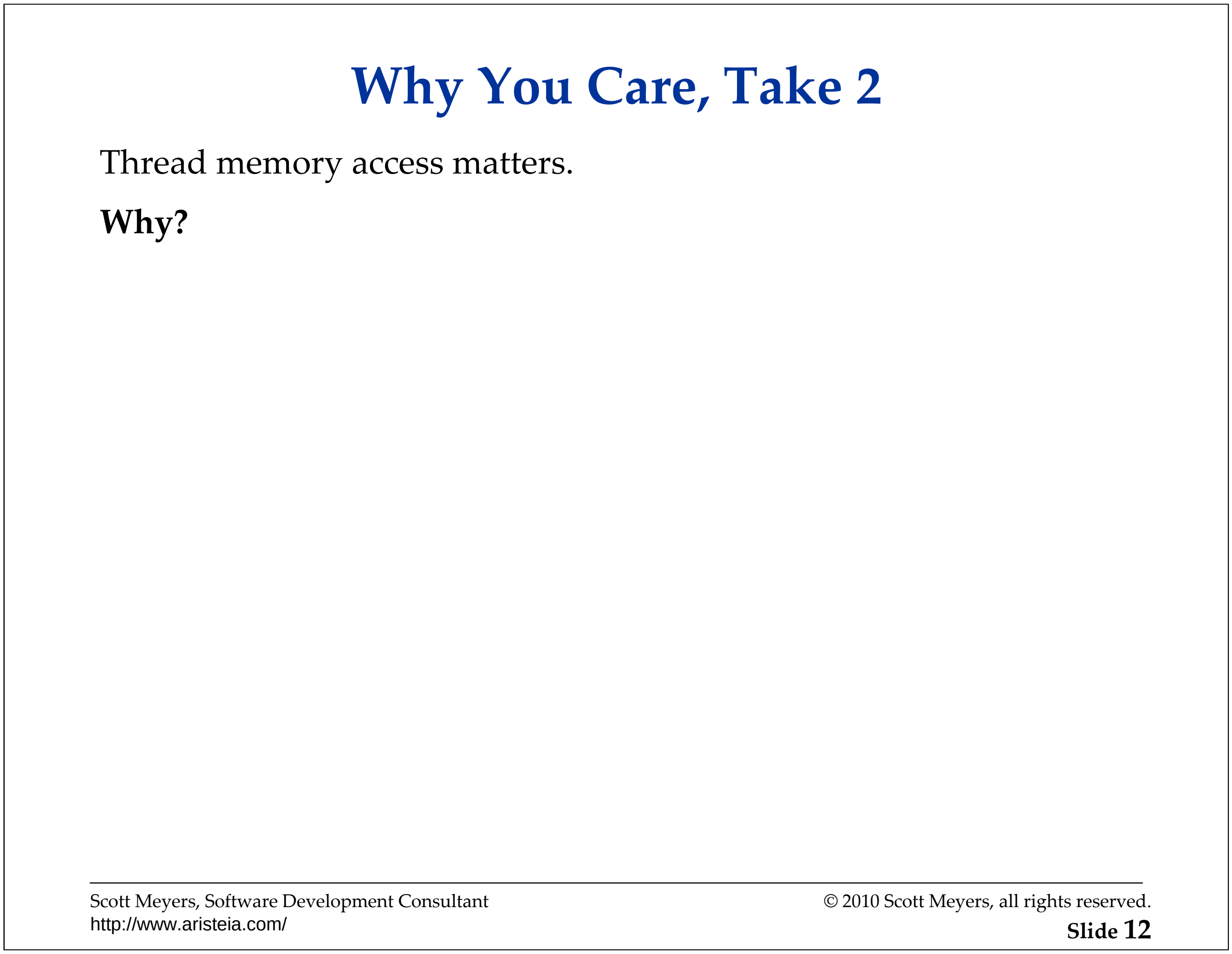
- Parallel pseudocode, take 2:

```
int result[P];

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                      // instead of result[p]
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                      // instead of result[p]
    result[p] = count; } );             // new statement

...                                     // nothing else changes
```

Why You Care, Take 2

Scalability now perfect!

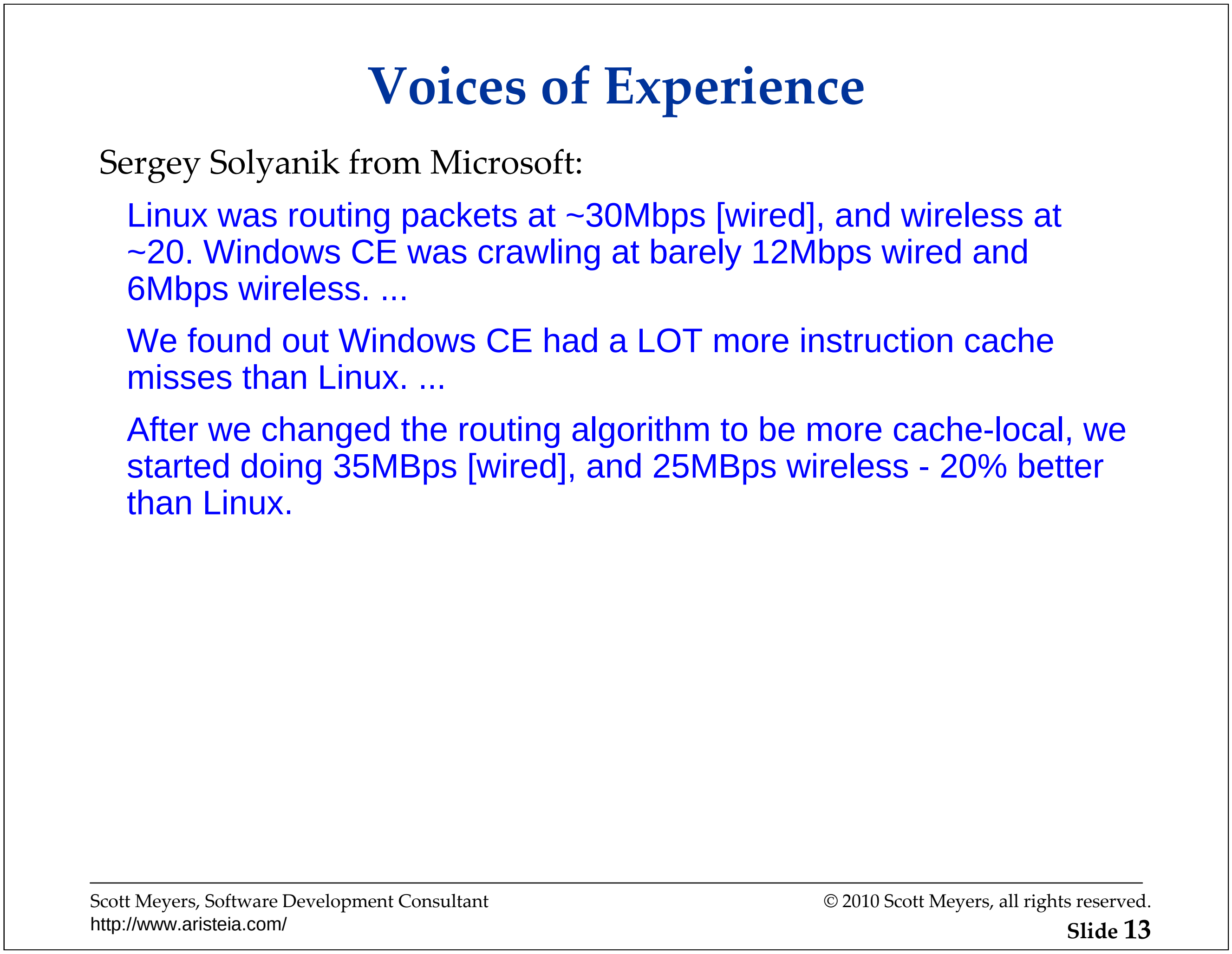




Why You Care, Take 2

Thread memory access matters.

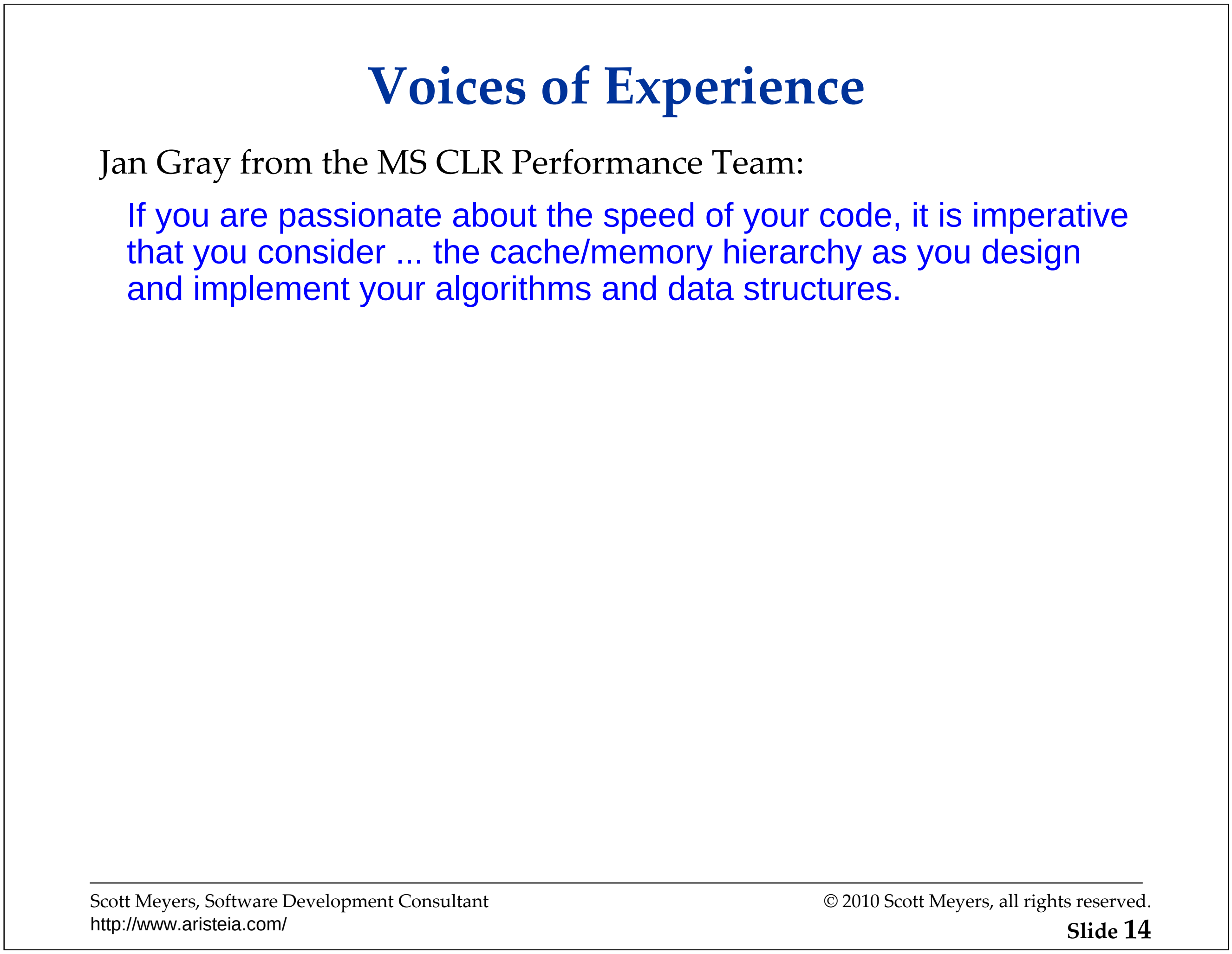
Why?



Voices of Experience

Sergey Solyanik from Microsoft:

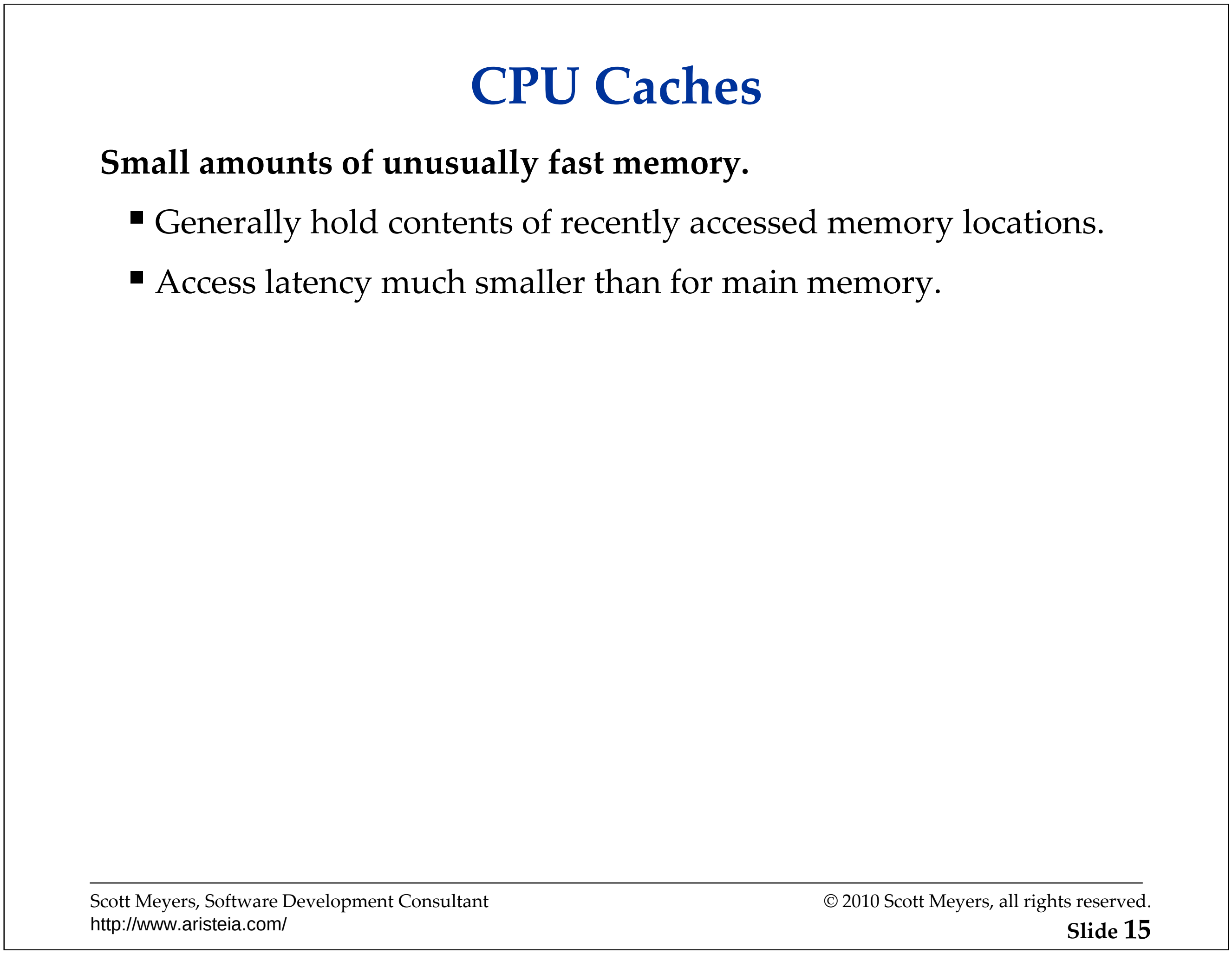
> Linux was routing packets at ~30Mbps [wired], and wireless at ~20. Windows CE was crawling at barely 12Mbps wired and 6Mbps wireless. ...
>
> We found out Windows CE had a LOT more instruction cache misses than Linux. ...
>
> After we changed the routing algorithm to be more cache-local, we started doing 35MBps [wired], and 25MBps wireless - 20% better than Linux.



Voices of Experience

Jan Gray from the MS CLR Performance Team:

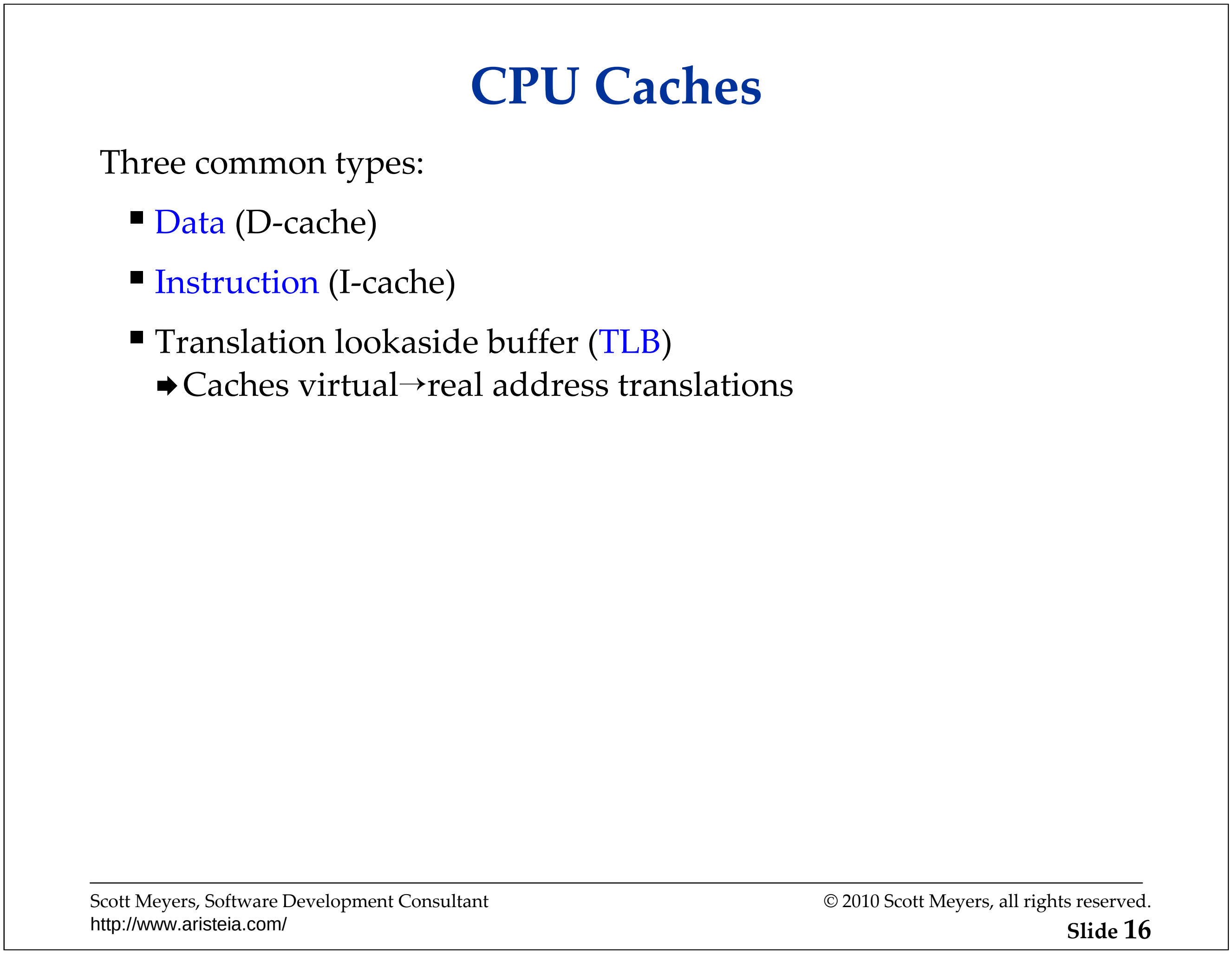
> If you are passionate about the speed of your code, it is imperative that you consider ... the cache/memory hierarchy as you design and implement your algorithms and data structures.



CPU Caches

Small amounts of unusually fast memory.

- Generally hold contents of recently accessed memory locations.
- Access latency much smaller than for main memory.



CPU Caches

Three common types:

- Data (D-cache)
- Instruction (I-cache)
- Translation lookaside buffer (TLB)
  - ➡ Caches virtual→real address translations

Beyond mentioning them on this slide, I don’t have anything to say about TLBs in this presentation.

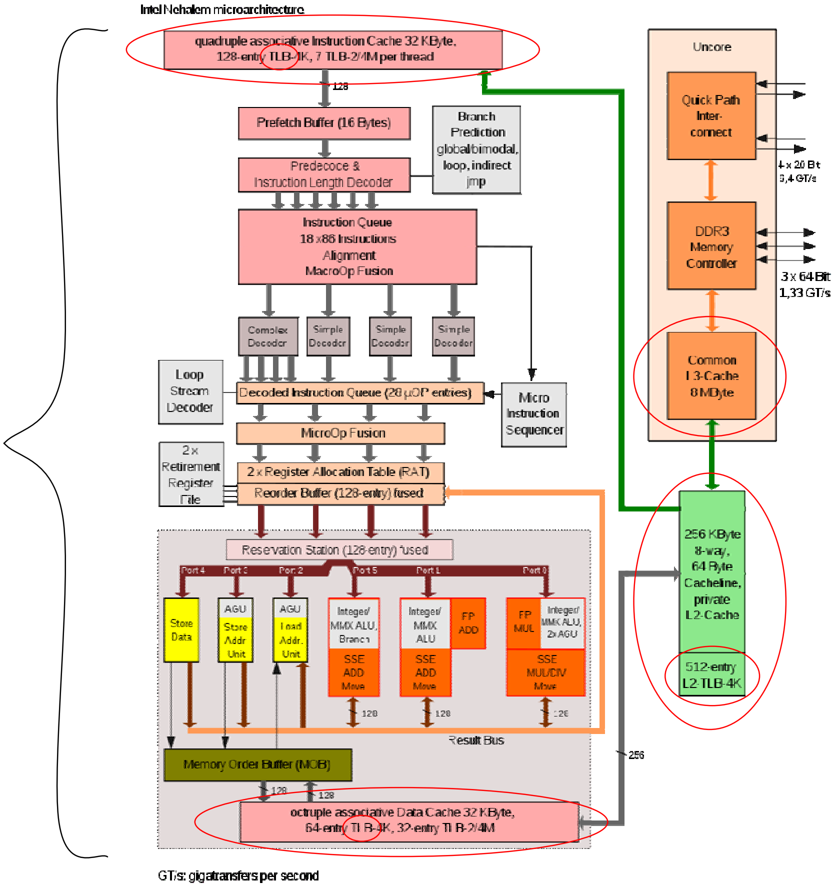
Cache Hierarchies

Cache hierarchies (*multi-level caches*) are common.

E.g., Intel Core i7-9xx processor:

- 32KB L1 I-cache, 32KB L1 D-cache per core
  - Shared by 2 HW threads
- 256 KB L2 cache per core
  - Holds both instructions and data
  - Shared by 2 HW threads
- 8MB L3 cache
  - Holds both instructions and data
  - Shared by 4 cores (8 HW threads)

Nehalem Architecture

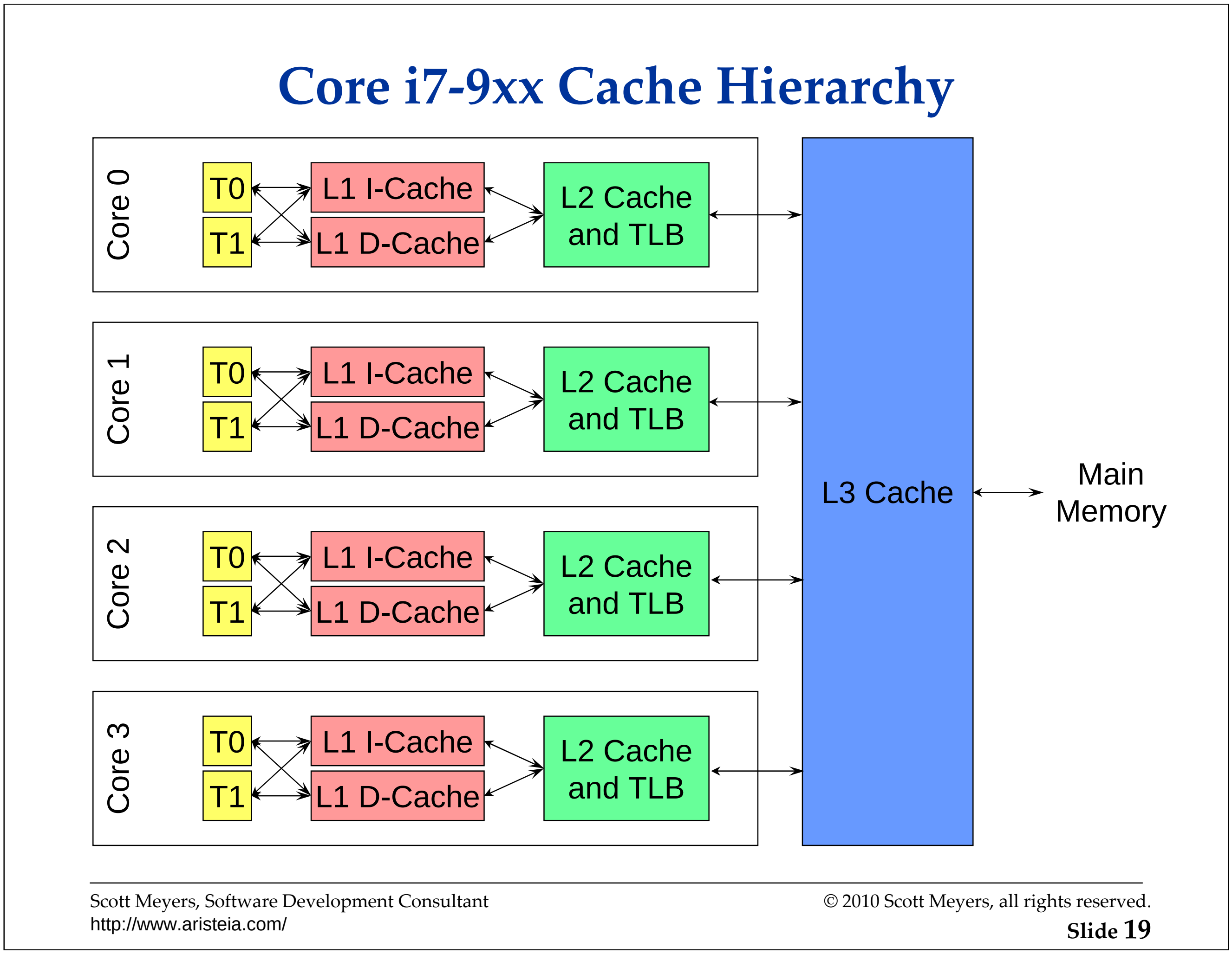

Intel Nehalem microarchitecture
One Core
quadruple associative Instruction Cache 32 KByte, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch Buffer (16 Bytes)
Branch Prediction global/bimodal, loop, indirect jmp
Predecode & Instruction Length Decoder
Instruction Queue 18 x86 Instructions Alignment MacroOp Fusion
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
Loop Stream Decoder
Decoded Instruction Queue (28 µOP entries)
Micro Instruction Sequencer
MicroOp Fusion
2 x Retirement Register File
2 x Register Allocation Table (RAT)
Reorder Buffer (128-entry) fused
Reservation Station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store Data
AGU
Store Addr. Unit
AGU
Load Addr. Unit
Integer/ MMX ALU, Branch
SSE ADD Move
Integer/ MMX ALU
SSE ADD Move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV Move
128
128
128
Result Bus
Memory Order Buffer (MOB)
128
128
octruple associative Data Cache 32 KByte, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 Memory Controller
3 x 64 Bit 1,33 GT/s
Common L3-Cache 8 MByte
256 KByte 8-way, 64 Byte Cacheline, private L2-Cache
512-entry L2-TLB-4K
GT/s: gigatransfers per second

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Slide 18

Image source: http://cs466.andersonje.com/public/images/800px-intel_nehalem_arch.svg.png

The Core i7 is one manifestation of this basic architecture.



Core i7-9xx Cache Hierarchy

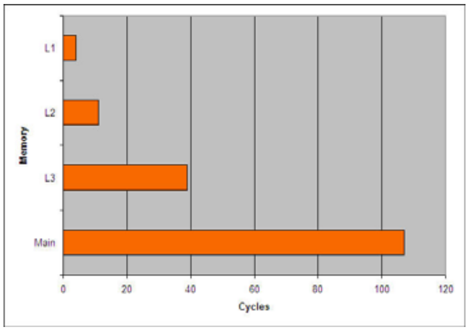
The sizes of the cache boxes are not to scale. Each L2 cache is 8 times as big as each L1 cache, and the L3 cache is 8 times as big as each L2 cache.

CPU Cache Characteristics

Caches are small.

- Assume 100MB program at runtime (code + data)
  - ➡ 8% fits in core-i79xx's L3 cache.
    - ◆ L3 cache shared by *every running process* (incl. OS)
  - ➡ 0.25% fits in each L2 cache.
  - ➡ 0.03% fits in each L1 cache.

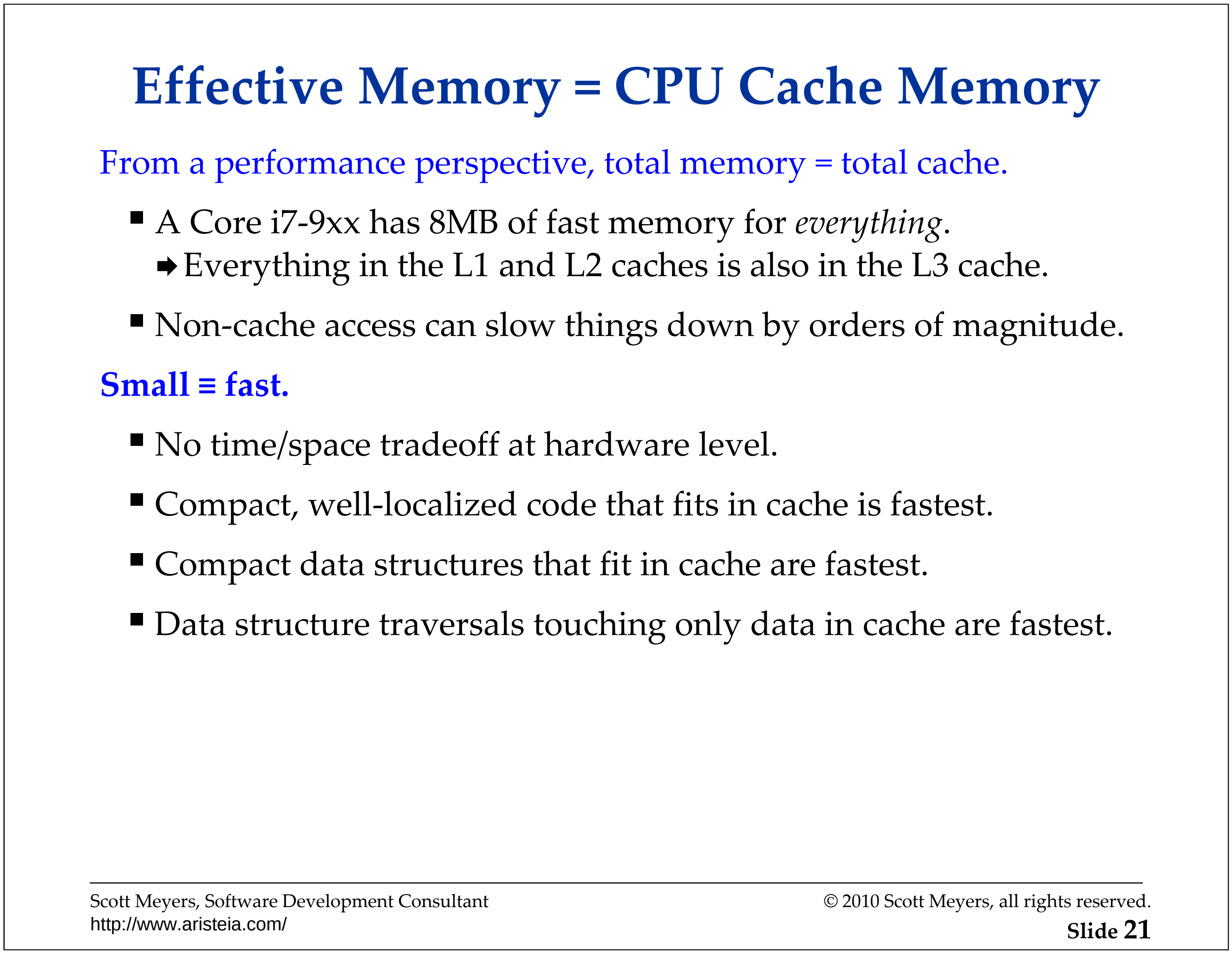
Caches much faster than main memory.



- For Core i7-9xx:
  - ➡ L1 latency is 4 cycles
  - ➡ L2 latency is 11 cycles
  - ➡ L3 latency is 39 cycles
  - ➡ Main memory latency is 107 cycles
    - ◆ 27 times slower than L1!
    - ◆ 100% CPU utilization ⇒ >99% CPU idle time!

Source for latency data: http://www.anandtech.com/show/2658/4

The bullet about >99% idle time is misleading, because the core would try to schedule the second hardware thread to run while waiting on memory latency for the first thread. Also, it's unrealistic to assume that every memory reference would yield a main memory access. Still, there's something to be said for shock value, hence the bullet :-)



Effective Memory = CPU Cache Memory

From a performance perspective, total memory = total cache.

- A Core i7-9xx has 8MB of fast memory for *everything*.
  - ➡ Everything in the L1 and L2 caches is also in the L3 cache.
- Non-cache access can slow things down by orders of magnitude.

Small ≡ fast.

- No time/space tradeoff at hardware level.
- Compact, well-localized code that fits in cache is fastest.
- Compact data structures that fit in cache are fastest.
- Data structure traversals touching only data in cache are fastest.

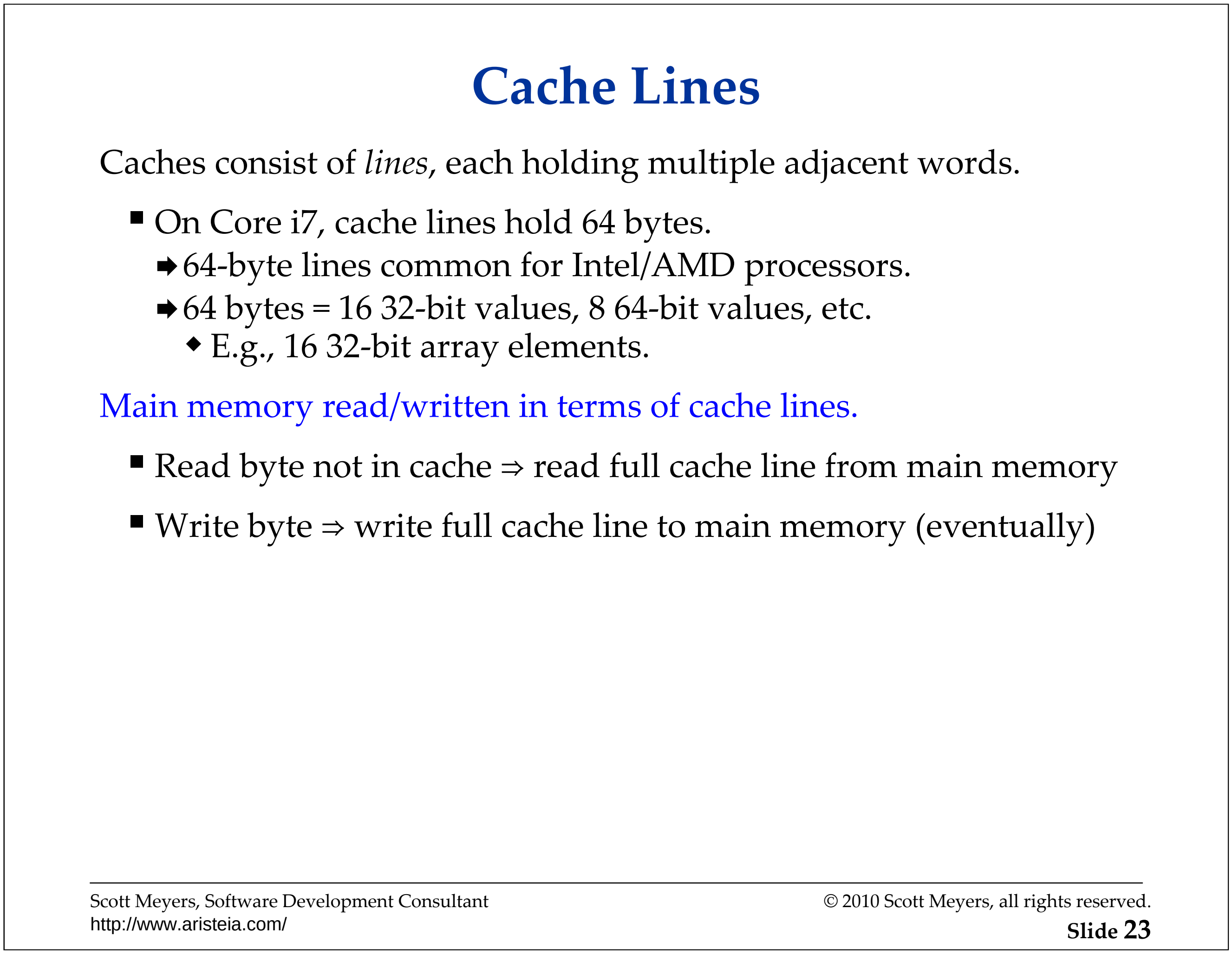
From http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719&p=7: "Nehalem's 8MB and 16 way associative L3 cache is inclusive of all lower levels of the cache hierarchy and shared between all four cores. ... Each core contains 64KB of data in the L1 caches and 256KB in the L2 cache (there may or may not be data that is in both the L1 and L2 caches). This means that 1-1.25MB of the 8MB L3 cache in Nehalem is filled with data that is also in other caches."

Gratuitous Animal Photos: Flying Fish

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2010 Scott Meyers, all rights reserved.
Slide 22

Sources: http://i.telegraph.co.uk/telegraph/multimedia/archive/01523/life-flying-fish_1523792i.jpg and http://www.frogview.com/show6.php?file=10092



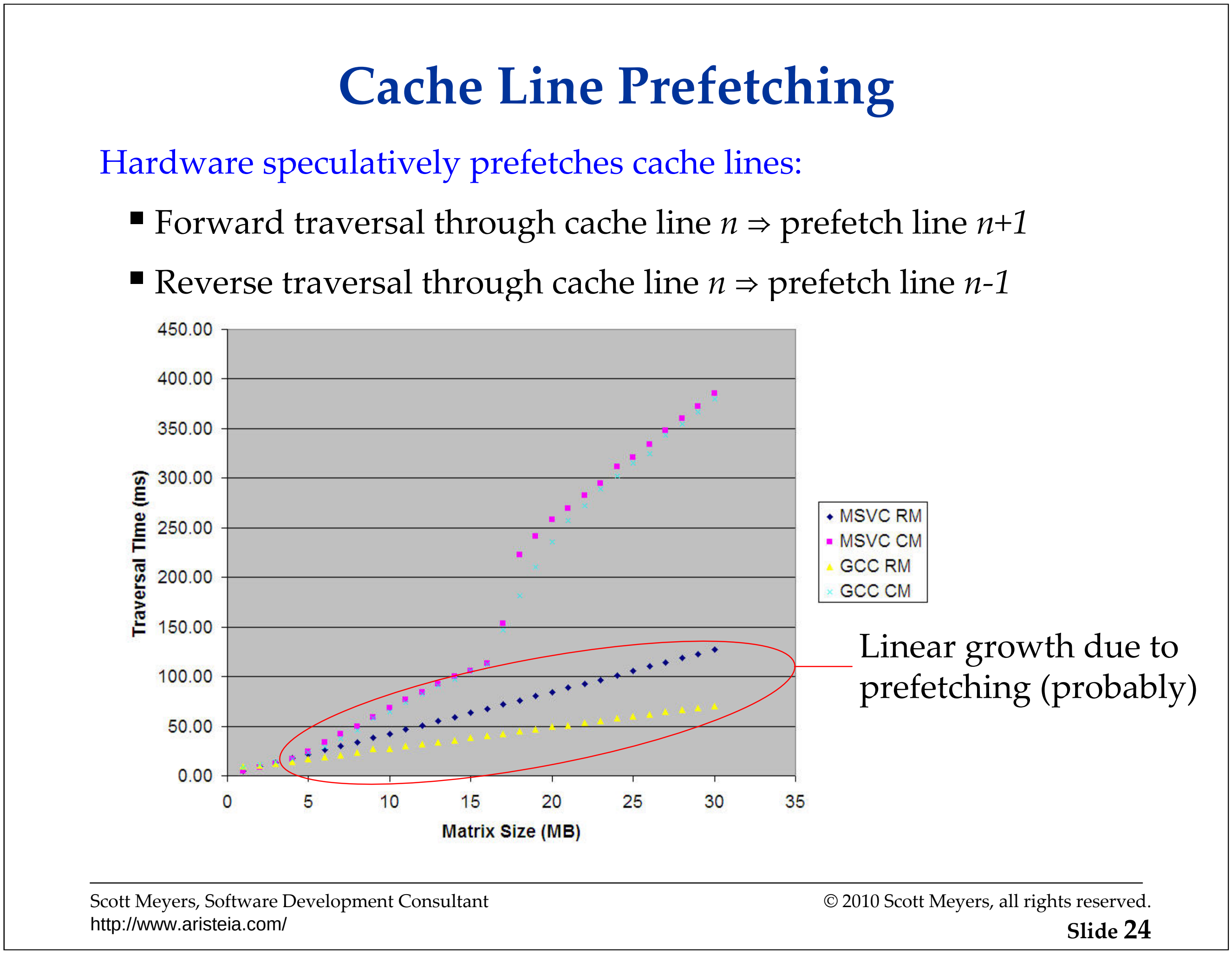
Cache Lines

Caches consist of *lines*, each holding multiple adjacent words.

- On Core i7, cache lines hold 64 bytes.
  - ➡ 64-byte lines common for Intel/AMD processors.
  - ➡ 64 bytes = 16 32-bit values, 8 64-bit values, etc.
    - ◆ E.g., 16 32-bit array elements.

Main memory read/written in terms of cache lines.

- Read byte not in cache ⇒ read full cache line from main memory
- Write byte ⇒ write full cache line to main memory (eventually)



Cache Line Prefetching

Hardware speculatively prefetches cache lines:

- Forward traversal through cache line *n* ⇒ prefetch line *n+1*
- Reverse traversal through cache line *n* ⇒ prefetch line *n-1*

Linear growth due to prefetching (probably)

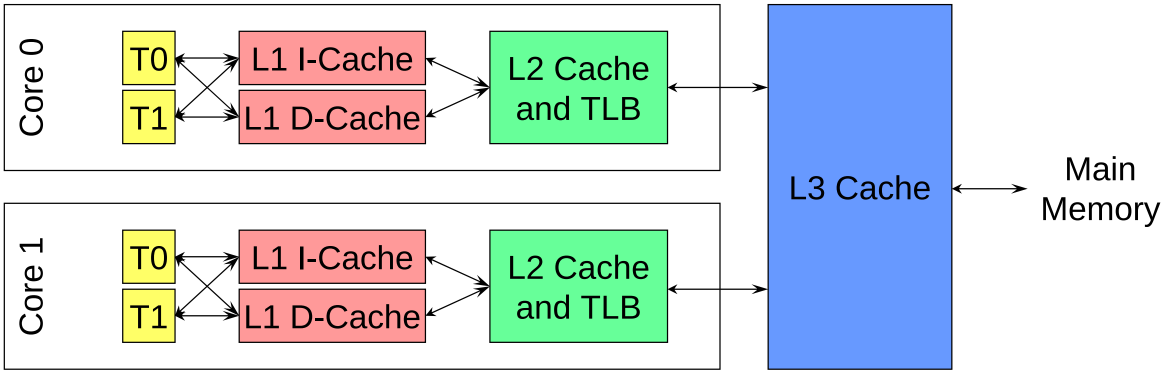
Successful prefetching decreases the effective latency of main memory.

Implications

- Locality counts.
  - Reads/writes at address *A* ⇒ contents of addresses near *A* already cached.
    - E.g., on the same cache line.
    - E.g., on nearby cache line that was prefetched.
- Predictable access patterns count.
  - "Predictable" ≅ forward or backwards traversals.
- Linear array traversals *very* cache-friendly.
  - Excellent locality, predictable traversal pattern.
  - Linear array search can beat $log_2\ n$ searches of heap-based BSTs.
  - $log_2\ n$ binary search of sorted array can beat *O(1)* searches of heap-based hash tables.
  - Algorithmic complexity wins for large *n*, but hardware caching takes an early lead.

Cache Coherency

From core i7's architecture:



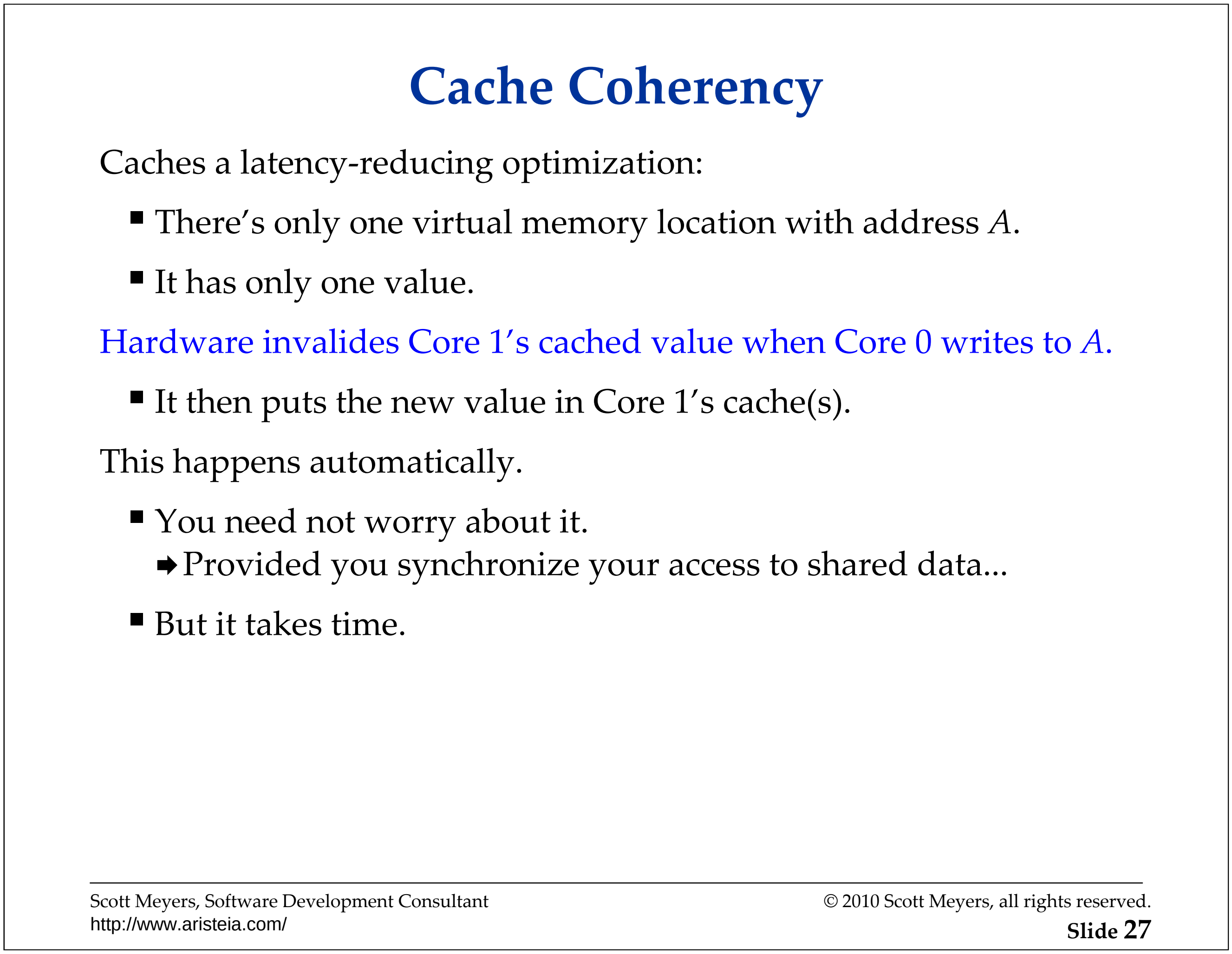
Assume both cores have cached the value at (virtual) address *A*.

- Whether in L1 or L2 makes no difference.

Consider:

- Core 0 writes to *A*.
- Core 1 reads *A*.

What value does Core 1 read?



Cache Coherency

Caches a latency-reducing optimization:

- There’s only one virtual memory location with address *A*.
- It has only one value.

Hardware invalides Core 1’s cached value when Core 0 writes to *A*.

- It then puts the new value in Core 1’s cache(s).

This happens automatically.

- You need not worry about it.
  - ➡ Provided you synchronize your access to shared data...
- But it takes time.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

© 2010 Scott Meyers, all rights reserved.

Slide 27

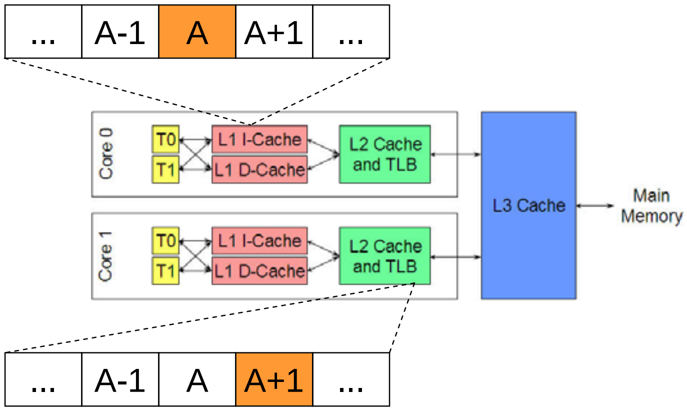
There are multiple ways to synchronize access to shared data, e.g., use a mutex, use atomic machine instructions, use memory barriers. Each of these will ensure that the hardware’s support for cache coherency will work for you.

False Sharing

Suppose Core 0 accesses *A* and Core 1 accesses *A+1*.

- *Independent* pieces of memory; concurrent access is safe.
- But *A* and *A+1* (probably) map to the same cache line.
  - ➡ If so, Core 0's writes to *A* invalidates *A+1*'s cache line in Core 1.
    - ◆ And vice versa.
    - ◆ This is *false sharing*.

Line from Core 0's cache

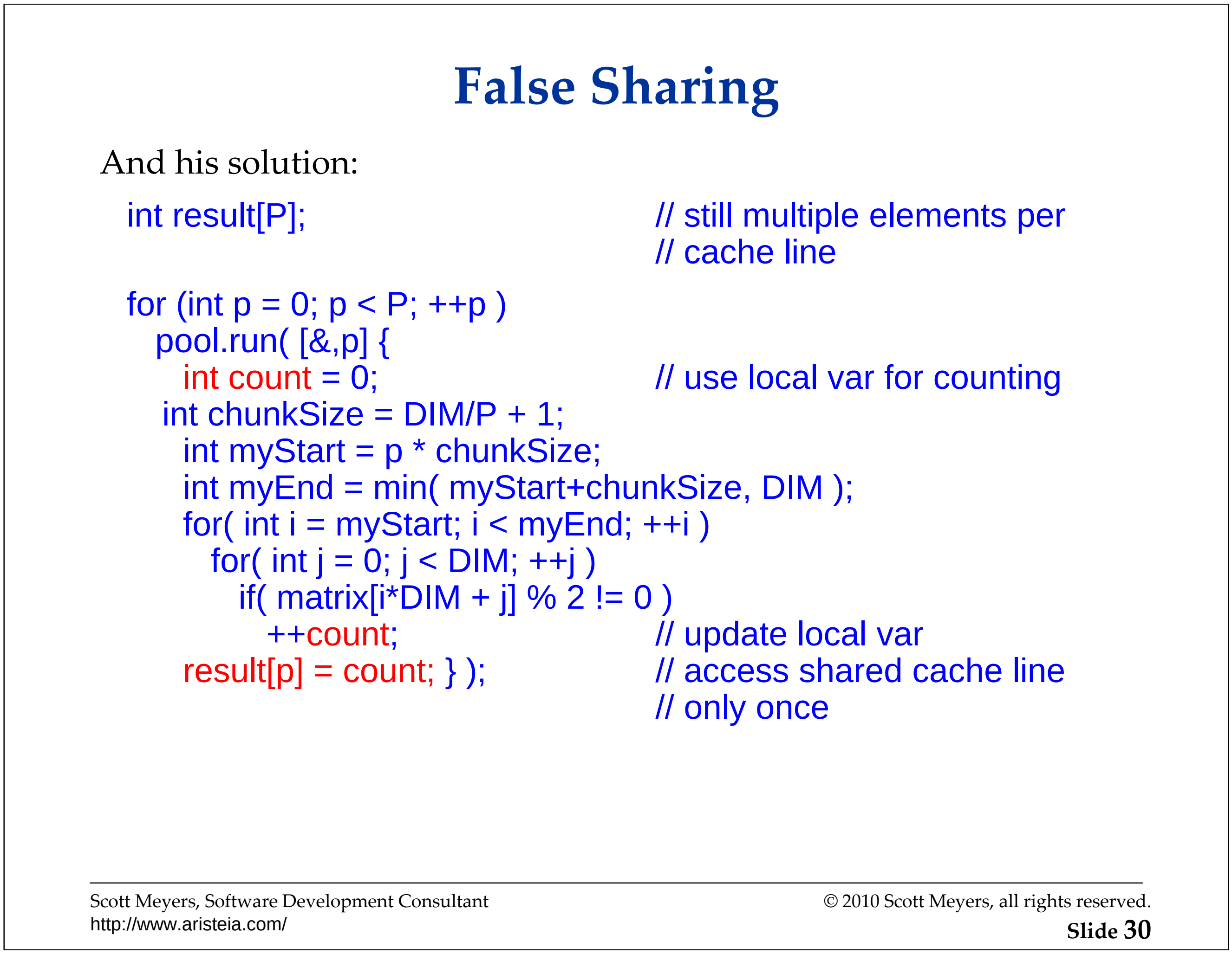


Line from Core 1's cache

False Sharing

It explains Herb Sutter's issue:

```
int result[P];                  // many elements on 1 cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {             // run P threads concurrently
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p]; } );     // each repeatedly accesses the
                                // same array (albeit different
                                // elements)
```



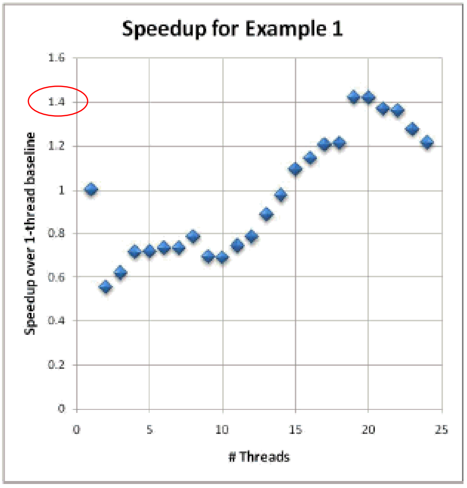
False Sharing

And his solution:

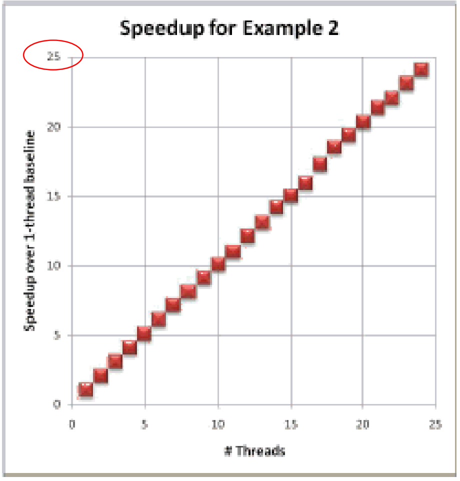
```
int result[P];                       // still multiple elements per
                                     // cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                   // use local var for counting
   int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                   // update local var
    result[p] = count; } );          // access shared cache line
                                     // only once
```

False Sharing

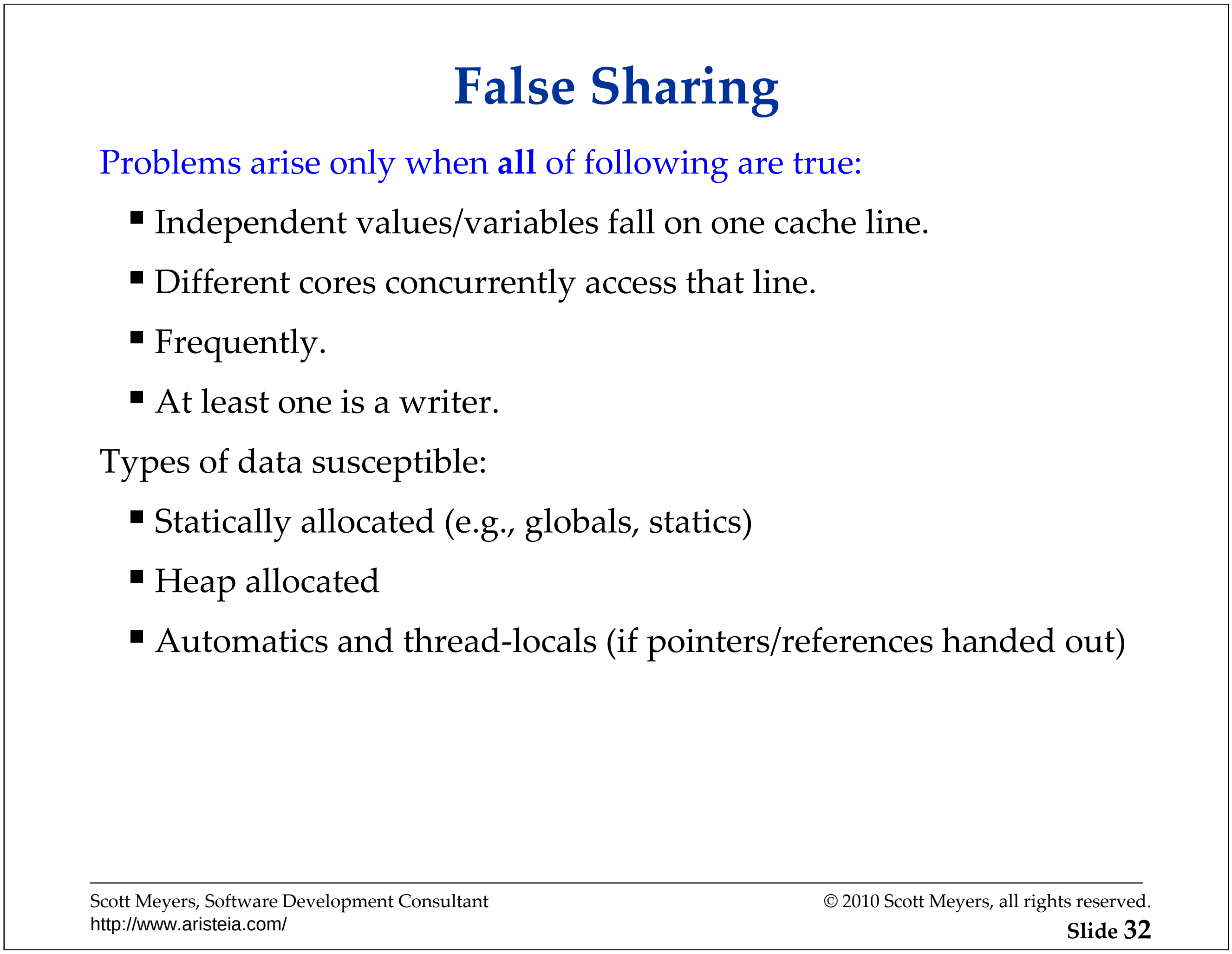
His scalability results are worth repeating:



With False Sharing



Without False Sharing



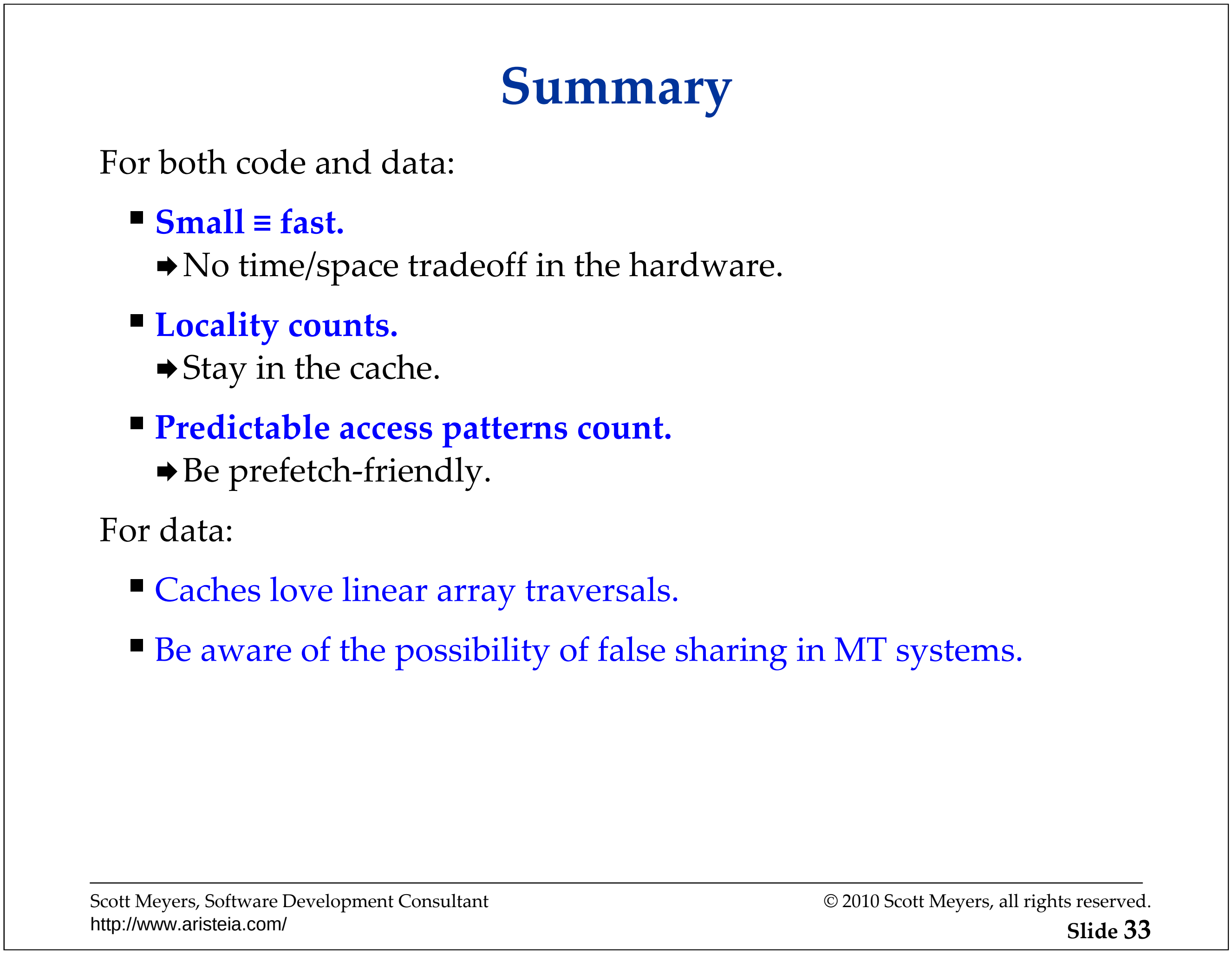
False Sharing

Problems arise only when **all** of following are true:

- Independent values/variables fall on one cache line.
- Different cores concurrently access that line.
- Frequently.
- At least one is a writer.

Types of data susceptible:

- Statically allocated (e.g., globals, statics)
- Heap allocated
- Automatics and thread-locals (if pointers/references handed out)



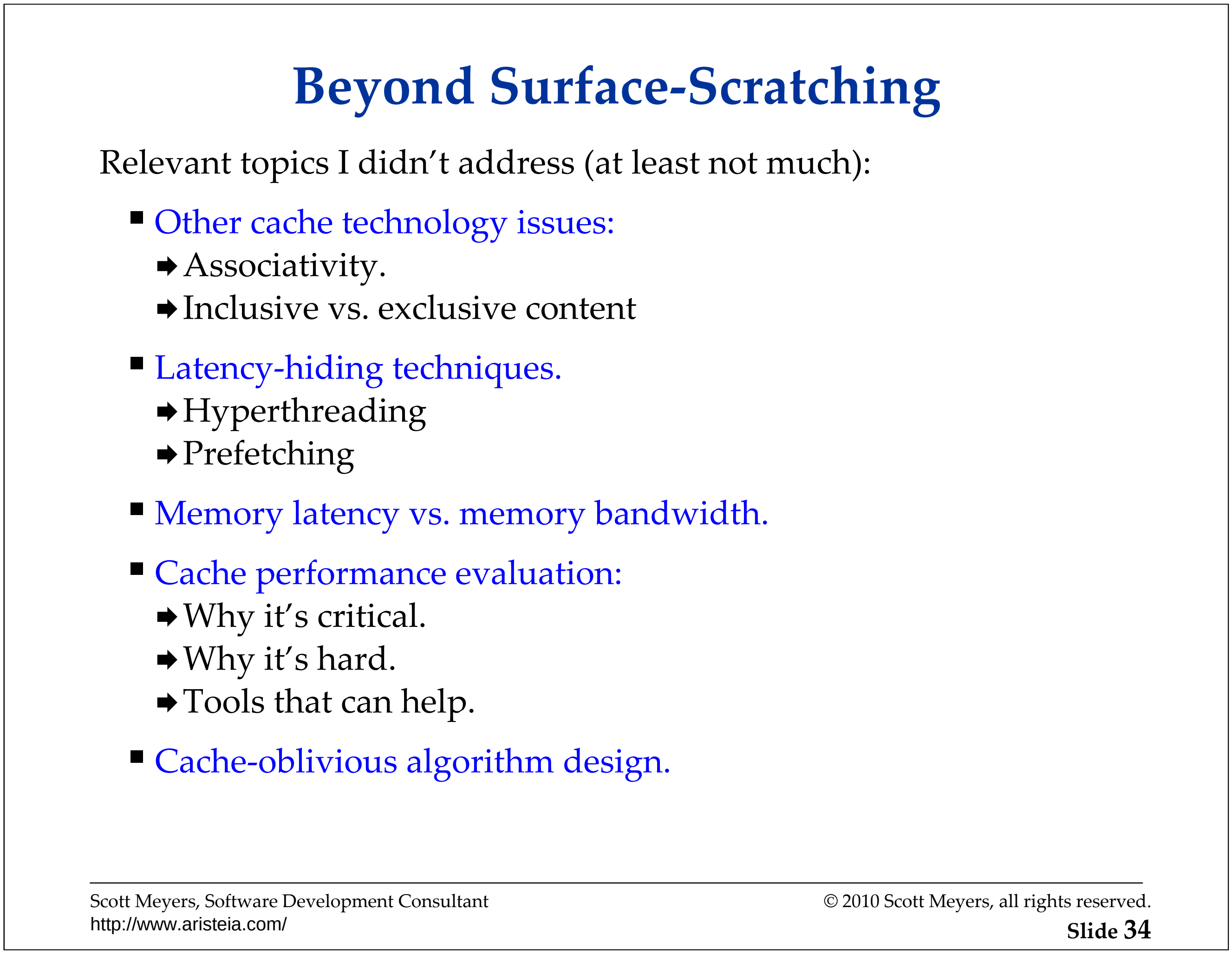
Summary

For both code and data:

- **Small ≡ fast.**
  - ➡ No time/space tradeoff in the hardware.
- **Locality counts.**
  - ➡ Stay in the cache.
- **Predictable access patterns count.**
  - ➡ Be prefetch-friendly.

For data:

- Caches love linear array traversals.
- Be aware of the possibility of false sharing in MT systems.



Beyond Surface-Scratching

Relevant topics I didn’t address (at least not much):

- Other cache technology issues:
  - ➡ Associativity.
  - ➡ Inclusive vs. exclusive content
- Latency-hiding techniques.
  - ➡ Hyperthreading
  - ➡ Prefetching
- Memory latency vs. memory bandwidth.
- Cache performance evaluation:
  - ➡ Why it’s critical.
  - ➡ Why it’s hard.
  - ➡ Tools that can help.
- Cache-oblivious algorithm design.

Further Information

- *What Every Programmer Should Know About Memory*, Ulrich Drepper, 21 November 2007, http://people.redhat.com/drepper/cpumemory.pdf.
- "CPU cache," *Wikipedia*.
- "Gallery of Processor Cache Effects," Igor Ostrovsky, *Igor Ostrovsky Blogging* (Blog), 19 January 2010.
- "Writing Faster Managed Code: Know What Things Cost," *MSDN*, June 2003.
  - Relevant section title is "Of Cache Misses, Page Faults, and Computer Architecture"
- "Memory is not free (more on Vista performance)," Sergey Solyanik, *1-800-Magic* (Blog), 9 December 2007.
  - Experience report about optimizing use of I-cache.

Further Information

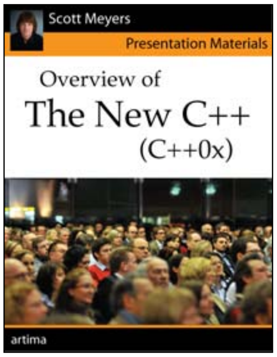
- "Eliminate False Sharing," Herb Sutter, *DrDobbs.com*, 14 May 2009.
- Coreinfo v2.0, Mark Russinovich, 21 October 2009.
  - ➡ Gives info on cores, caches, etc., for Windows platforms.

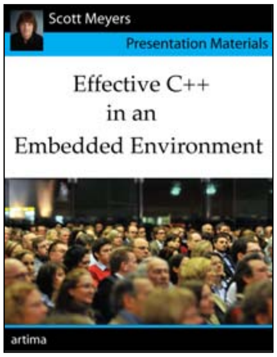
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses available for personal use include:





About Scott Meyers

Scott is a trainer and consultant on the design and implementation of software systems, typically in C++. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog