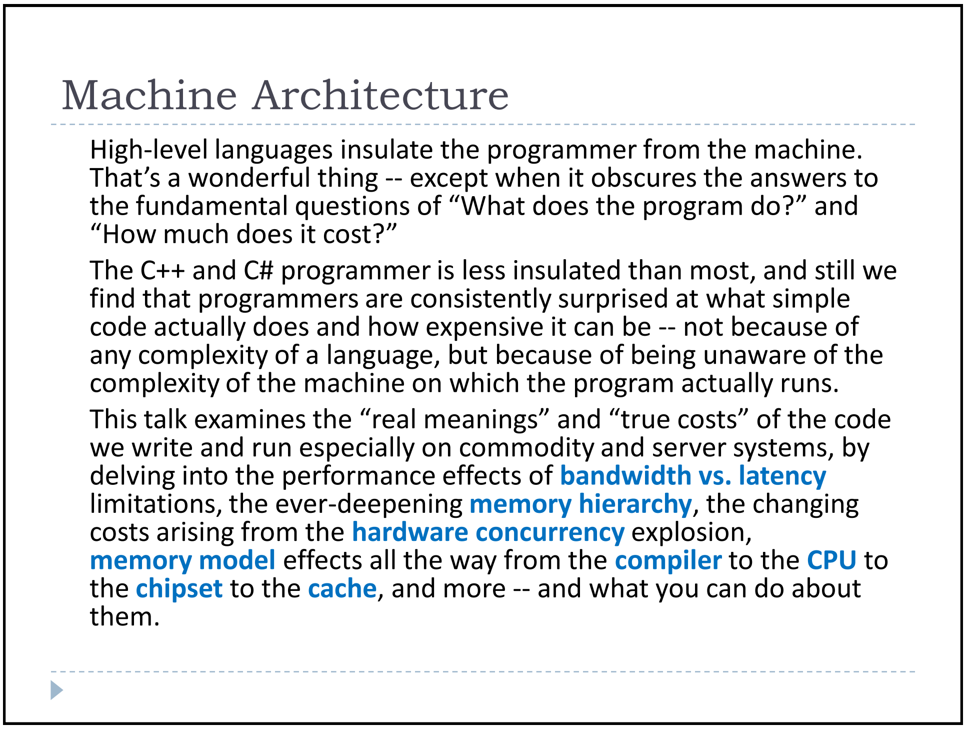
Machine Architecture

(Things Your Programming Language Never Told You)

Herb Sutter

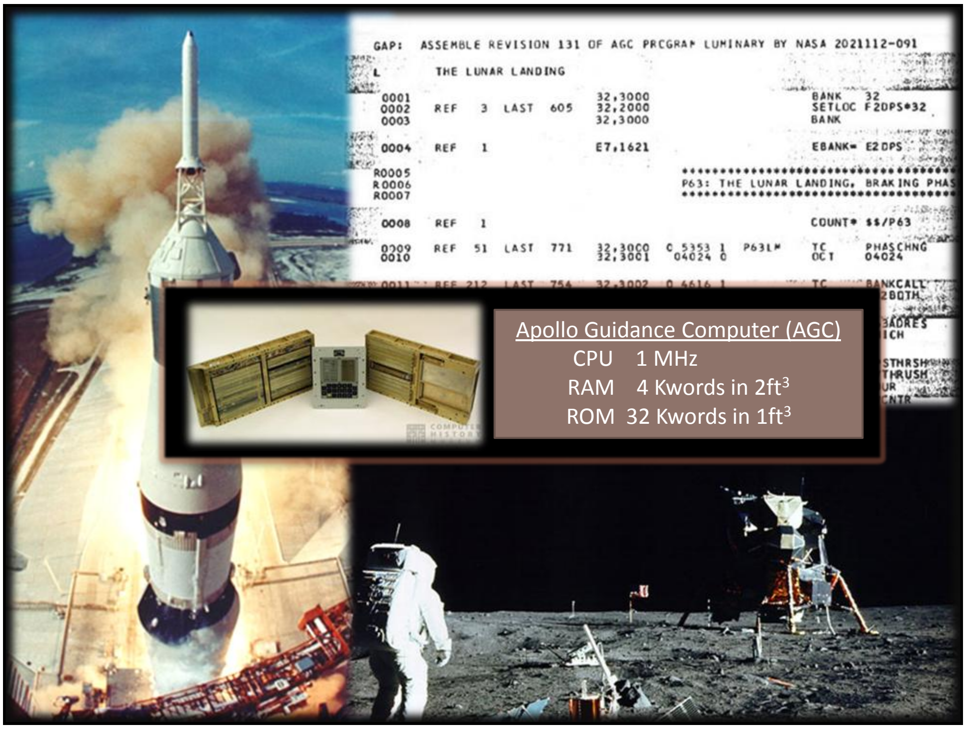


Machine Architecture

High-level languages insulate the programmer from the machine. That’s a wonderful thing -- except when it obscures the answers to the fundamental questions of “What does the program do?” and “How much does it cost?”

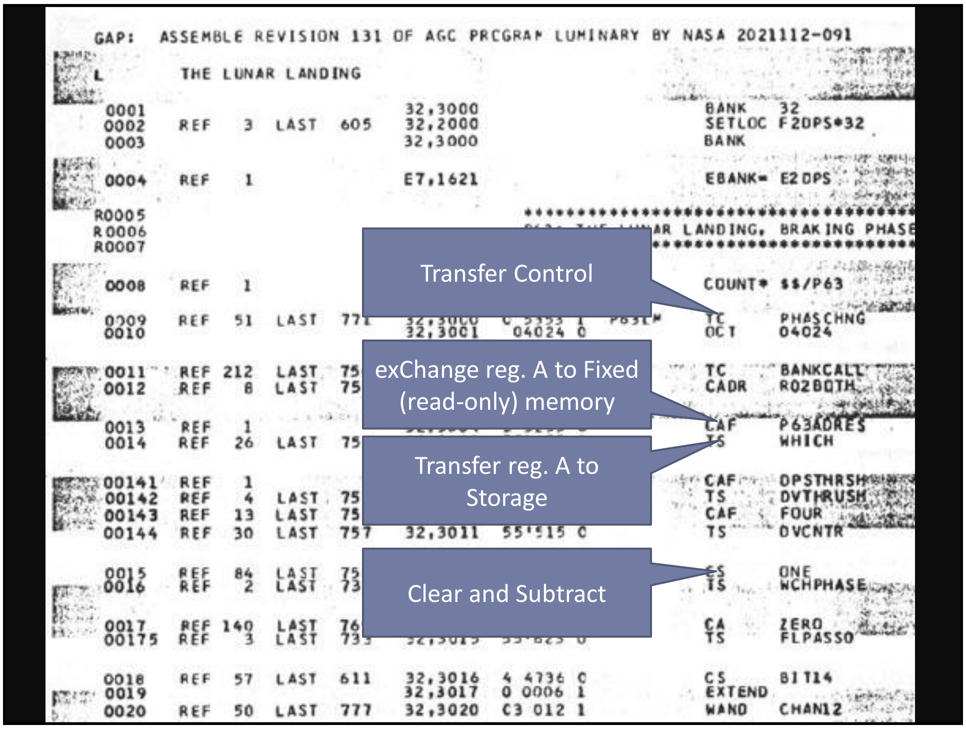
The C++ and C# programmer is less insulated than most, and still we find that programmers are consistently surprised at what simple code actually does and how expensive it can be -- not because of any complexity of a language, but because of being unaware of the complexity of the machine on which the program actually runs.

This talk examines the “real meanings” and “true costs” of the code we write and run especially on commodity and server systems, by delving into the performance effects of **bandwidth vs. latency** limitations, the ever-deepening **memory hierarchy**, the changing costs arising from the **hardware concurrency** explosion, **memory model** effects all the way from the **compiler** to the **CPU** to the **chipset** to the **cache**, and more -- and what you can do about them.



```
GAP:   ASSEMBLE REVISION 131 OF AGC PRCGRAM LUMINARY BY NASA 2021112-091

L         THE LUNAR LANDING

  0001                         32,3000                               BANK     32
  0002    REF    3   LAST   605   32,2000                            SETLOC  F2DPS*32
  0003                         32,3000                               BANK

  0004    REF    1               E7,1621                             EBANK=  E2DPS

 R0005                                            ****************************************
 R0006                                            P63: THE LUNAR LANDING, BRAKING PHAS
 R0007                                            ****************************************

  0008    REF    1                                                   COUNT*  $$/P63

  0009    REF   51   LAST   771   32,3000   0 5353 1   P63LM         TC      PHASCHNG
  0010                            32,3001    04024 0                 OCT     04024
```

Apollo Guidance Computer (AGC)
CPU 1 MHz
RAM 4 Kwords in 2ft^3
ROM 32 Kwords in 1ft^3



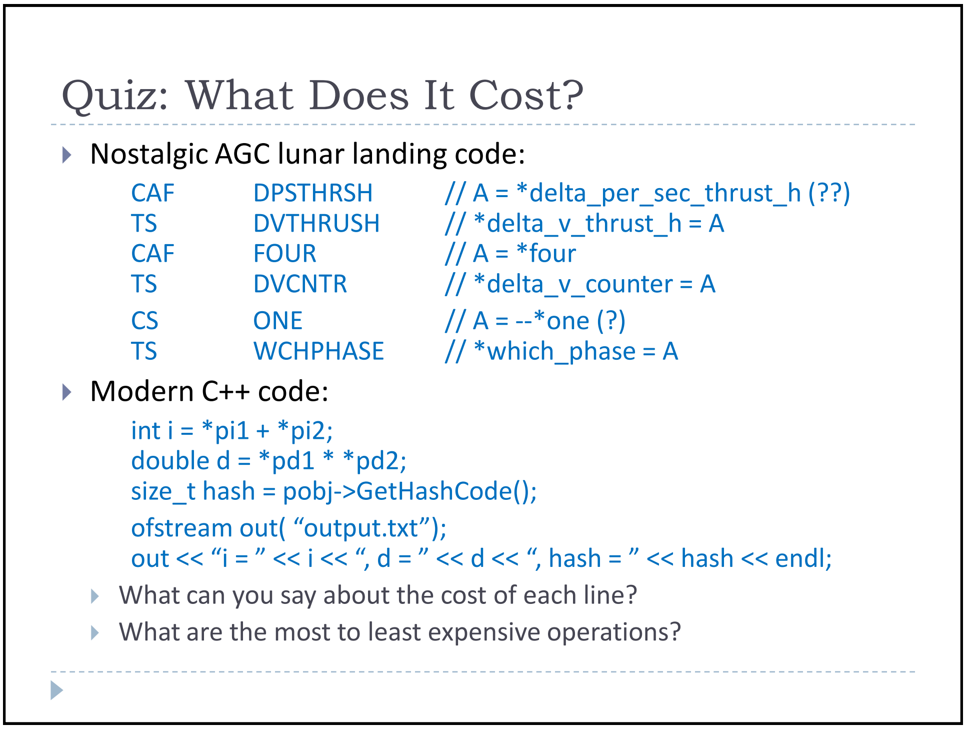
```
GAP:   ASSEMBLE REVISION 131 OF AGC PRCGRAM LUMINARY BY NASA 2021112-091

L         THE LUNAR LANDING

  0001                            32,3000                            BANK     32
  0002    REF    3   LAST   605   32,2000                            SETLOC  F2DPS*32
  0003                            32,3000                            BANK

  0004    REF    1                E7,1621                            EBANK=  E2DPS

 R0005                                            ****************************************
 R0006                                            ...AR LANDING, BRAKING PHASE
 R0007                                            ****************************************

  0008    REF    1                                                   COUNT*  $$/P63

  0009    REF   51   LAST   771                                      TC      PHASCHNG
  0010                            32,3001    04024 0                 OCT     04024

  0011    REF  212   LAST   75                                       TC      BANKCALL
  0012    REF    8   LAST   75                                       CADR    R02BOTH

  0013    REF    1                                                   CAF     P63ADRES
  0014    REF   26   LAST   75                                       TS      WHICH

  00141   REF    1                                                   CAF     OPSTHRSH
  00142   REF    4   LAST   75                                       TS      DVTHRUSH
  00143   REF   13   LAST   75                                       CAF     FOUR
  00144   REF   30   LAST   757   32,3011   55515 0                  TS      DVCNTR

  0015    REF   84   LAST   75                                       CS      ONE
  0016    REF    2   LAST   73                                       TS      WCHPHASE

  0017    REF  140   LAST   76                                       CA      ZERO
  00175   REF    3   LAST   733                                      TS      FLPASSO

  0018    REF   57   LAST   611   32,3016   4 4736 0                 CS      BIT14
  0019                            32,3017   0 0006 1                 EXTEND
  0020    REF   50   LAST   777   32,3020   C3 012 1                 WAND    CHAN12
```

Transfer Control

exChange reg. A to Fixed (read-only) memory

Transfer reg. A to Storage

Clear and Subtract



Quiz: What Does It Cost?

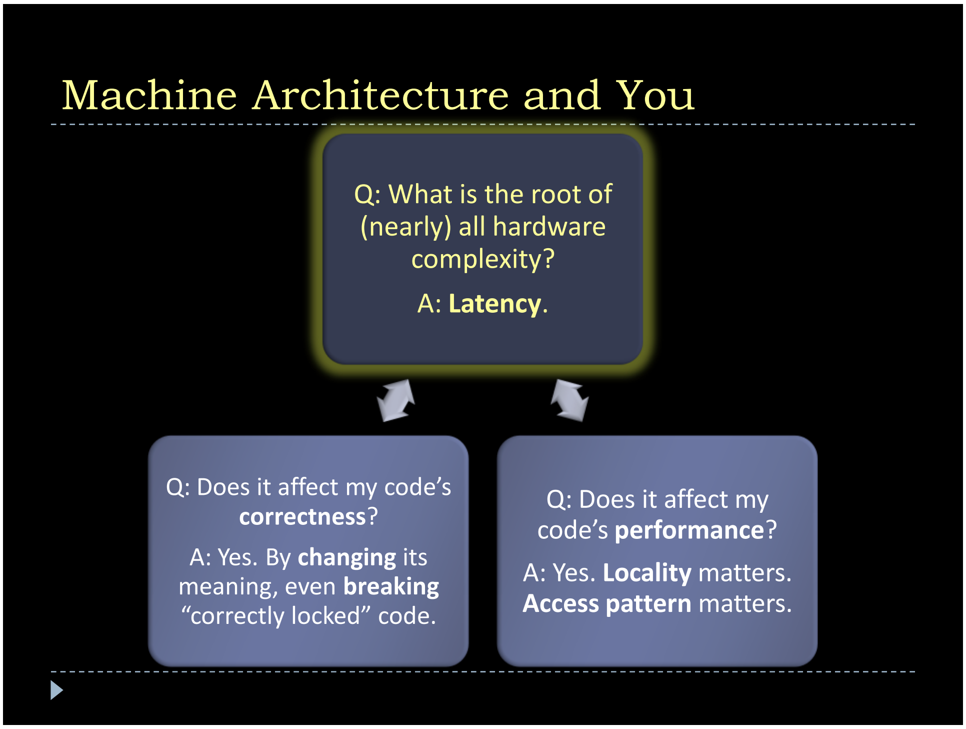
- Nostalgic AGC lunar landing code:

```
CAF    DPSTHRSH    // A = *delta_per_sec_thrust_h (??)
TS     DVTHRUSH    // *delta_v_thrust_h = A
CAF    FOUR        // A = *four
TS     DVCNTR      // *delta_v_counter = A
CS     ONE         // A = --*one (?)
TS     WCHPHASE    // *which_phase = A
```

- Modern C++ code:

```
int i = *pi1 + *pi2;
double d = *pd1 * *pd2;
size_t hash = pobj->GetHashCode();
ofstream out( “output.txt”);
out << “i = ” << i << “, d = ” << d << “, hash = ” << hash << endl;
```

  - What can you say about the cost of each line?
  - What are the most to least expensive operations?



Machine Architecture and You

Q: What is the root of (nearly) all hardware complexity?

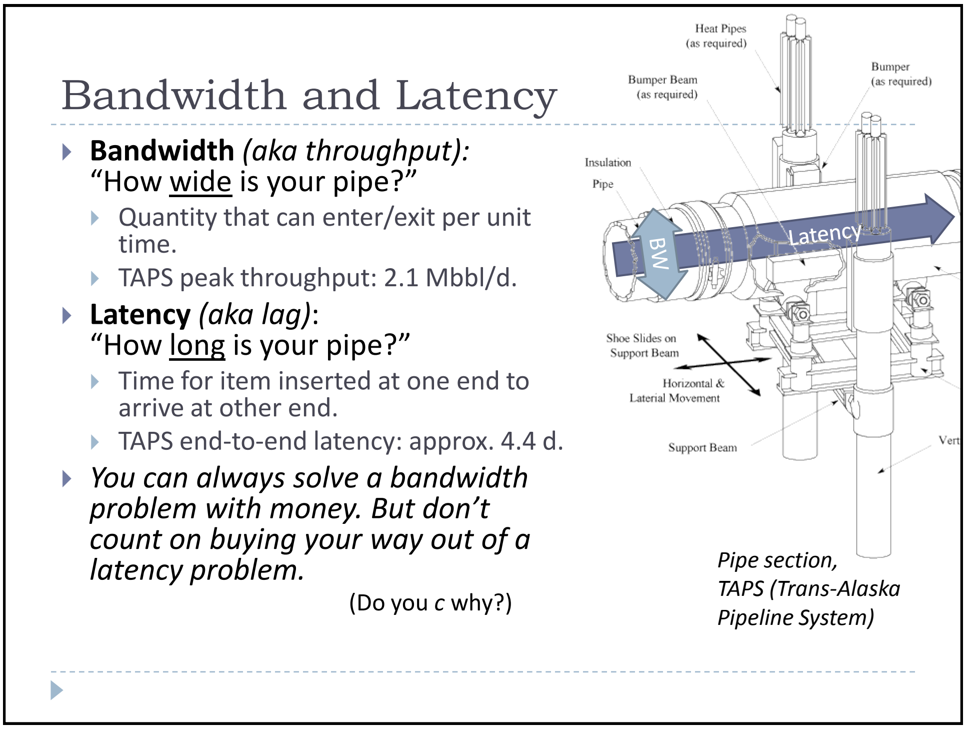
A: **Latency**.

Q: Does it affect my code’s **correctness**?

A: Yes. By **changing** its meaning, even **breaking** “correctly locked” code.

Q: Does it affect my code’s **performance**?

A: Yes. **Locality** matters. **Access pattern** matters.

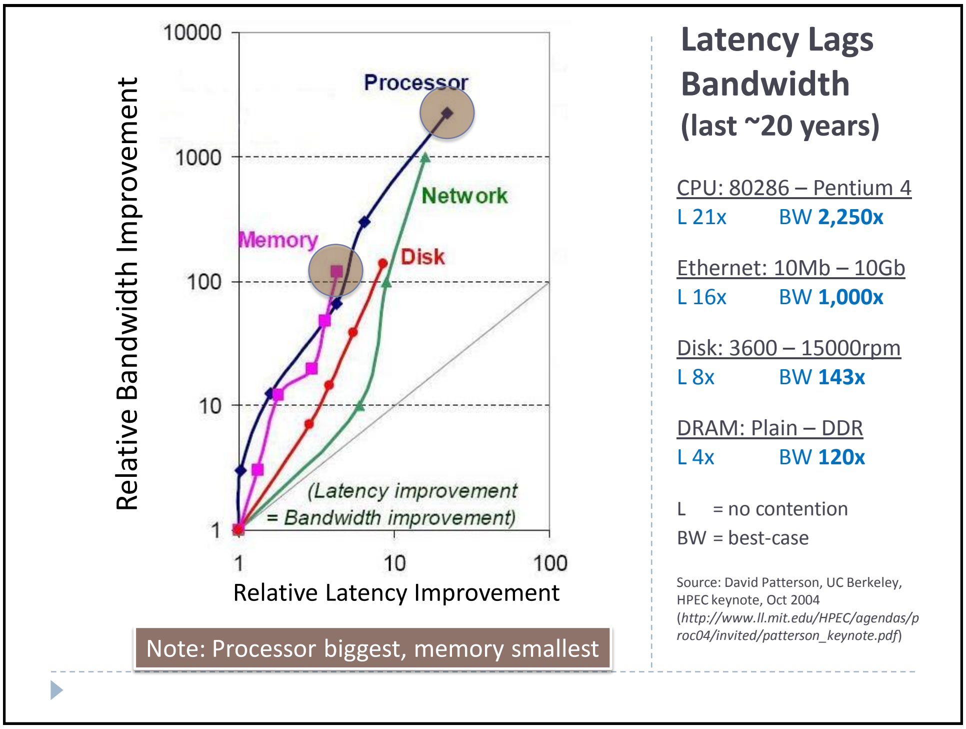


Bandwidth and Latency

- **Bandwidth** *(aka throughput):* "How <u>wide</u> is your pipe?"
  - Quantity that can enter/exit per unit time.
  - TAPS peak throughput: 2.1 Mbbl/d.
- **Latency** *(aka lag)*: "How <u>long</u> is your pipe?"
  - Time for item inserted at one end to arrive at other end.
  - TAPS end-to-end latency: approx. 4.4 d.
- *You can always solve a bandwidth problem with money. But don't count on buying your way out of a latency problem.*

(Do you *c* why?)

Pipe section, TAPS (Trans-Alaska Pipeline System)



Note: Processor biggest, memory smallest

Latency Lags Bandwidth (last ~20 years)

<u>CPU: 80286 – Pentium 4</u>
L 21x BW **2,250x**

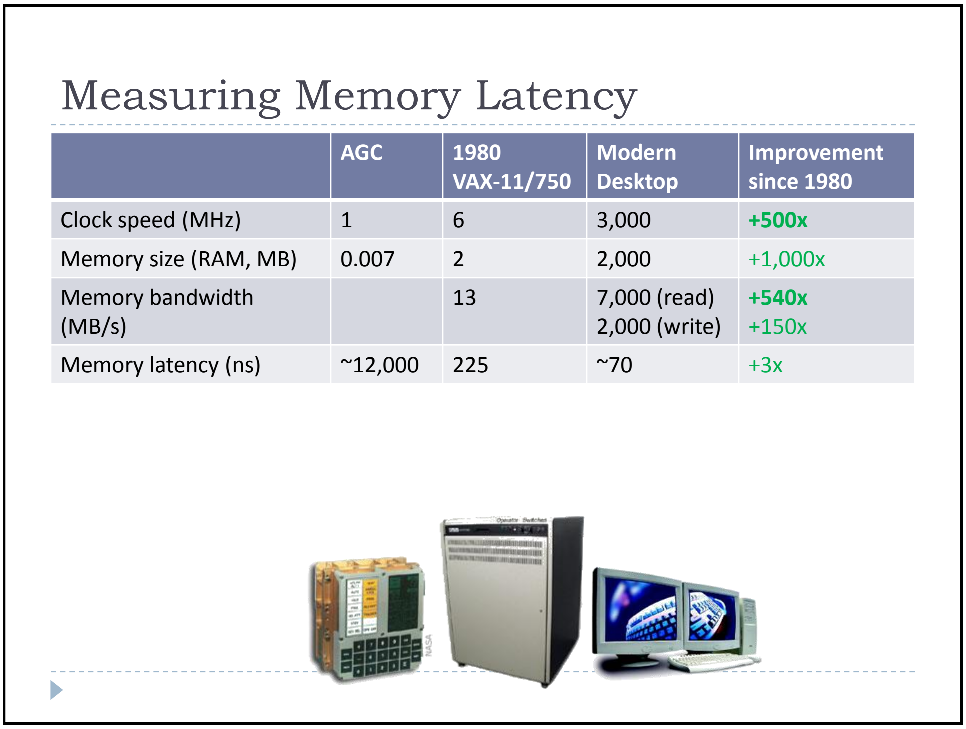
<u>Ethernet: 10Mb – 10Gb</u>
L 16x BW **1,000x**

<u>Disk: 3600 – 15000rpm</u>
L 8x BW **143x**

<u>DRAM: Plain – DDR</u>
L 4x BW **120x**

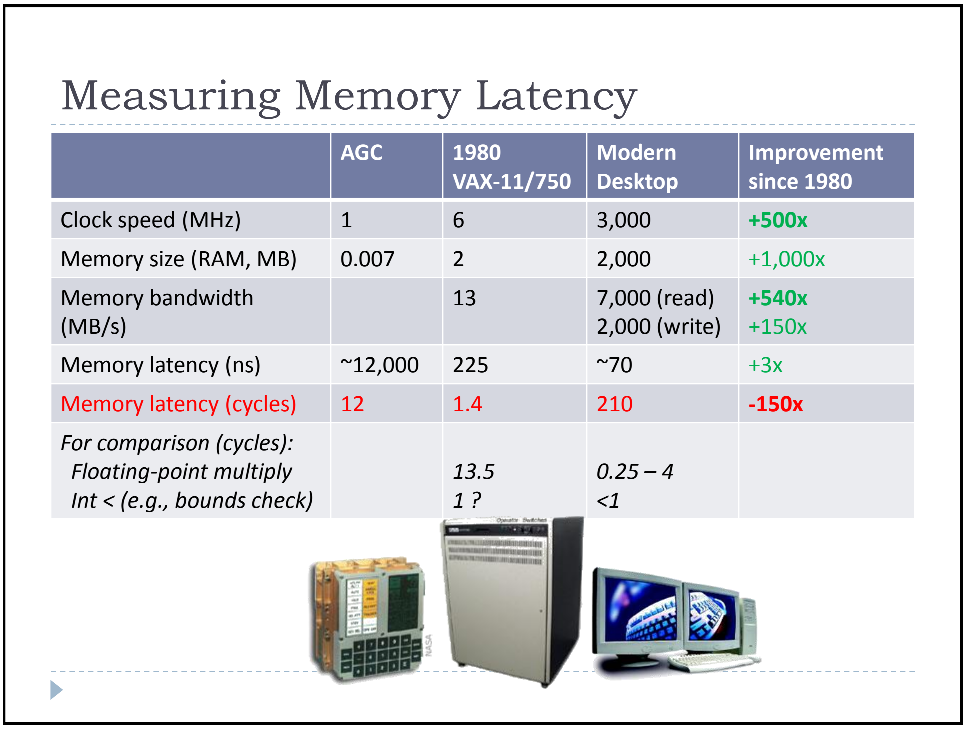
L = no contention
BW = best-case

Source: David Patterson, UC Berkeley, HPEC keynote, Oct 2004 (*http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf*)



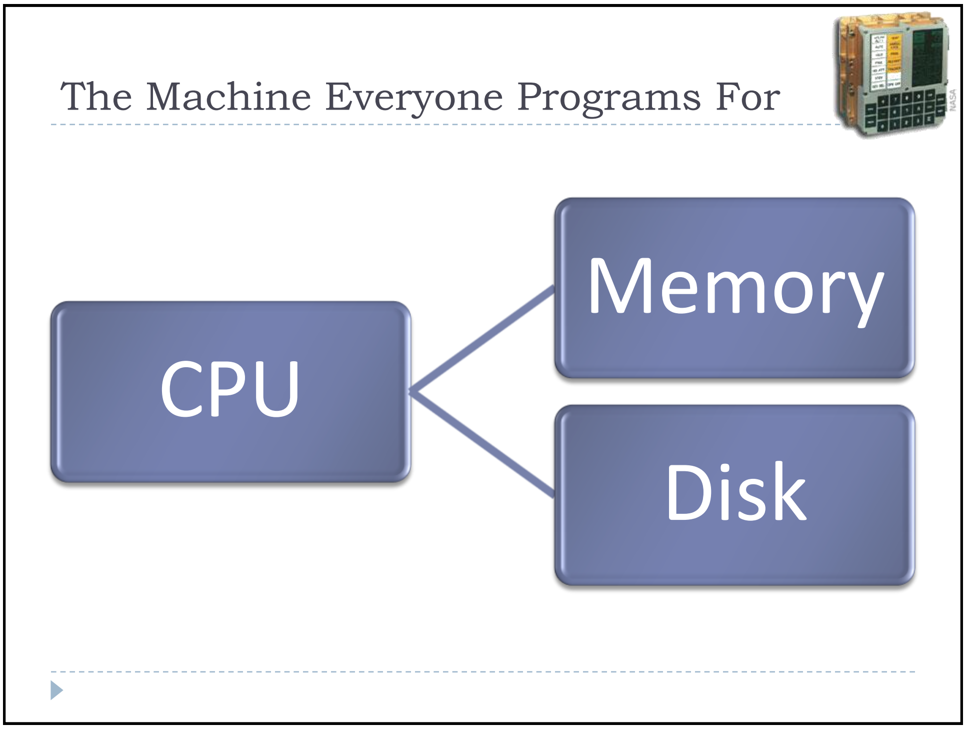
Measuring Memory Latency

	AGC	1980 VAX-11/750	Modern Desktop	Improvement since 1980
Clock speed (MHz)	1	6	3,000	**+500x**
Memory size (RAM, MB)	0.007	2	2,000	+1,000x
Memory bandwidth (MB/s)		13	7,000 (read) 2,000 (write)	**+540x** +150x
Memory latency (ns)	~12,000	225	~70	+3x



Measuring Memory Latency

	AGC	1980 VAX-11/750	Modern Desktop	Improvement since 1980
Clock speed (MHz)	1	6	3,000	**+500x**
Memory size (RAM, MB)	0.007	2	2,000	+1,000x
Memory bandwidth (MB/s)		13	7,000 (read) 2,000 (write)	**+540x** +150x
Memory latency (ns)	~12,000	225	~70	+3x
Memory latency (cycles)	12	1.4	210	**-150x**
For comparison (cycles): *Floating-point multiply* *Int < (e.g., bounds check)*		 *13.5* *1 ?*	 *0.25 – 4* *<1*	

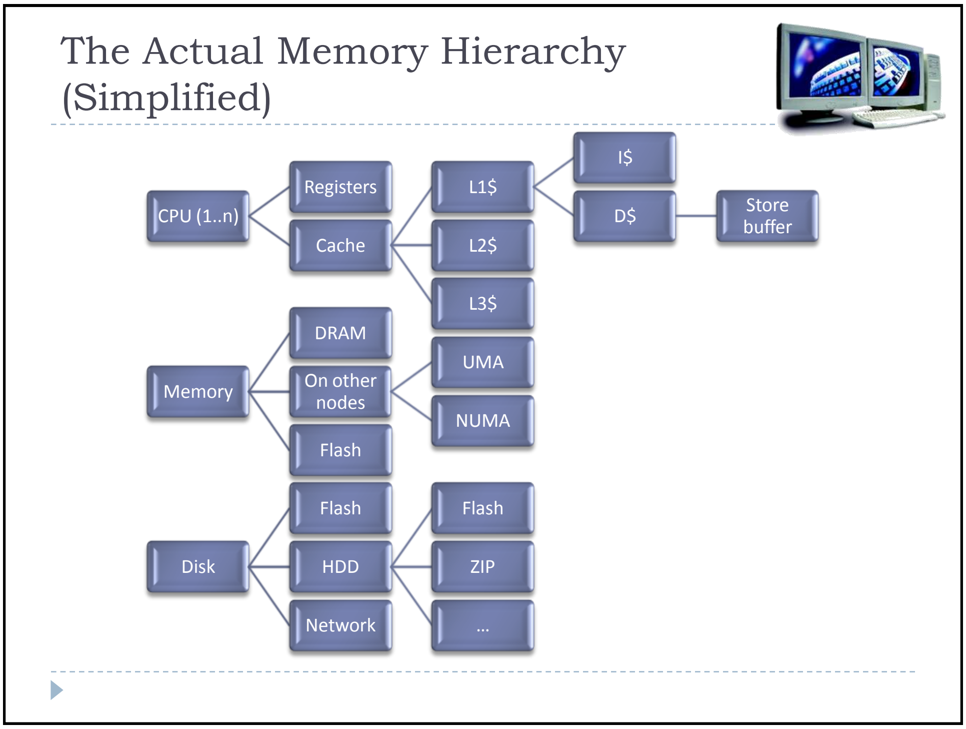


The Machine Everyone Programs For

CPU

Memory

Disk



The Actual Memory Hierarchy (Simplified)

CPU (1..n)

Registers

Cache

L1$

L2$

L3$

I$

D$

Store buffer

Memory

DRAM

On other nodes

Flash

UMA

NUMA

Disk

Flash

HDD

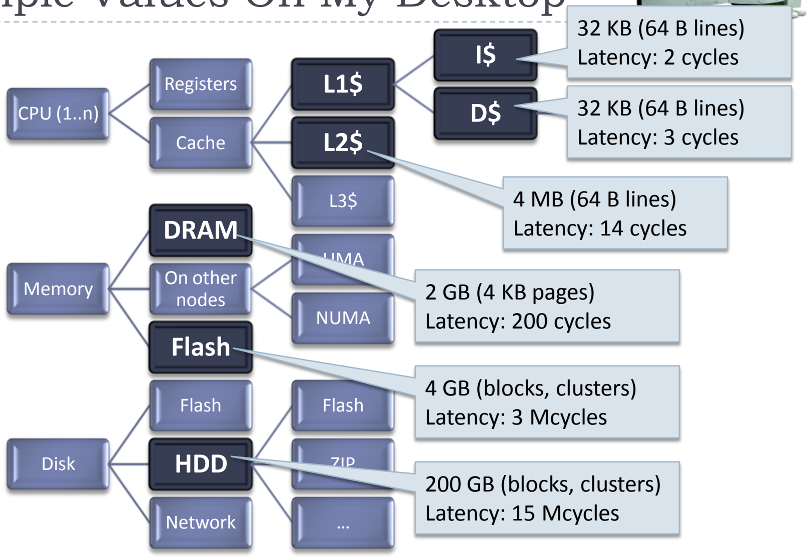
Network

Flash

ZIP

...

Sample Values On My Desktop



32 KB (64 B lines)
Latency: 2 cycles

32 KB (64 B lines)
Latency: 3 cycles

4 MB (64 B lines)
Latency: 14 cycles

2 GB (4 KB pages)
Latency: 200 cycles

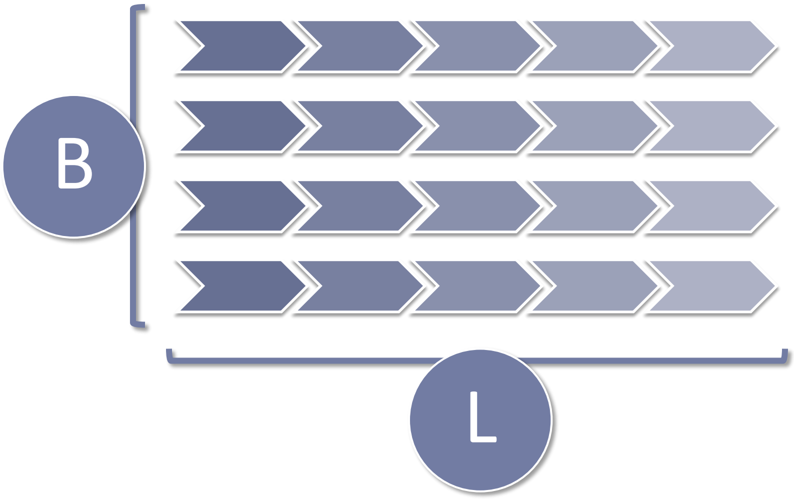
4 GB (blocks, clusters)
Latency: 3 Mcycles

200 GB (blocks, clusters)
Latency: 15 Mcycles

Little's Law

$B \times L = C$

Bandwidth × Latency = Concurrency



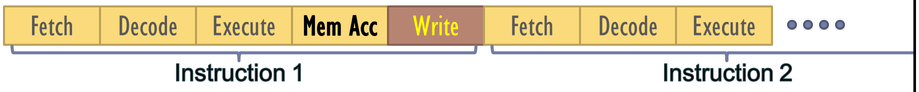
Q: So How Do We Cope With Latency? A: **Add Concurrency… Everywhere…**

Strategy	Technique	Can affect your code?
Parallelize (leverage compute power)	**Pipeline, execute out of order ("OoO"):** Launch expensive memory operations earlier, and do other work while waiting.	**Yes**
	Add hardware threads: Have other work available for the *same CPU core* to perform while other work is blocked on memory.	No *
Cache (leverage capacity)	**Instruction cache**	No
	Data cache: Multiple levels. Unit of sharing = cache line.	**Yes**
	Other buffering: Perhaps the most popular is store buffering, because writes are usually more expensive.	**Yes**
Speculate (leverage bandwidth, compute)	**Predict branches:** Guess whether an "if" will be true.	No
	Other optimistic execution: E.g., try both branches?	No
	Prefetch, scout: Warm up the cache.	No

** But you have to provide said other work (e.g., software threads) or this is useless!*

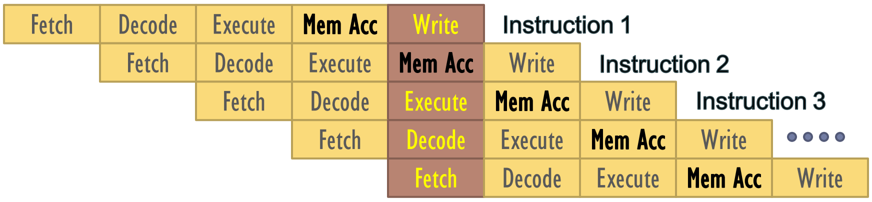
Processor Pipelining

- Why do this (one instruction at a time):



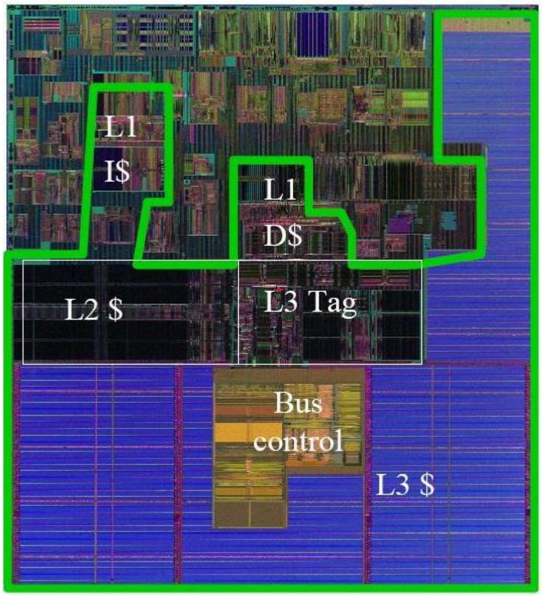
Fetch	Decode	Execute	Mem Acc	Write	Fetch	Decode	Execute	••••
Instruction 1					Instruction 2			

- When you can do this instead (sample 5-stage pipeline):



Fetch	Decode	Execute	Mem Acc	Write	Instruction 1			
	Fetch	Decode	Execute	Mem Acc	Write	Instruction 2		
		Fetch	Decode	Execute	Mem Acc	Write	Instruction 3	
			Fetch	Decode	Execute	Mem Acc	Write	••••
				Fetch	Decode	Execute	Mem Acc	Write

  - Pentium: 5 stages → 20 (P4) → **31** (Prescott P4) → 14 (Core 2)
- One benefit is to try to launch memory ops sooner:
  Keep pipeline full, and have more work while waiting.
  (But we also run out of work faster…)



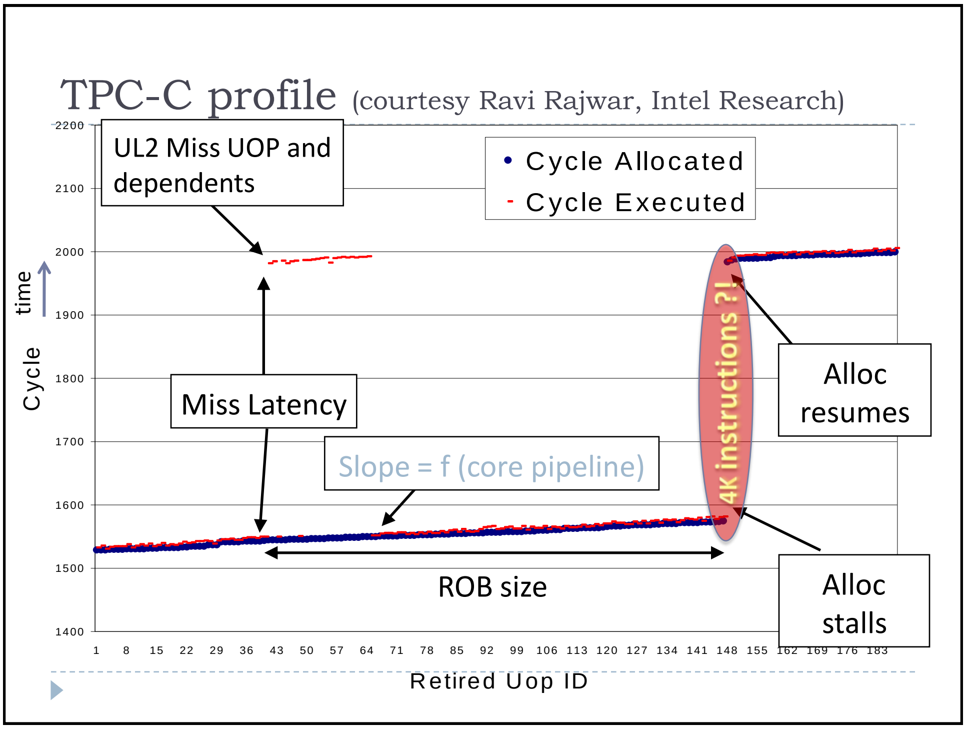
Sample Modern CPU

Original Itanium 2 had 211Mt, **85%** for cache:
- 16 KB L1I$
- 16 KB L1D$
- 256 KB L2$
- 3 MB L3$

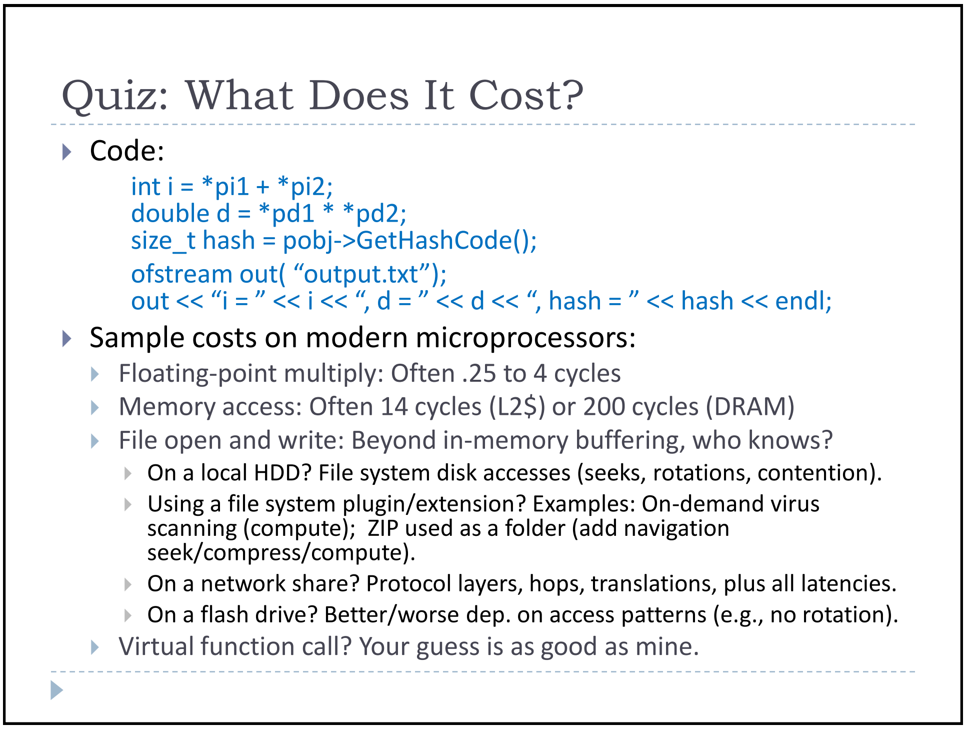
1% of die to compute, **99%** to move/store data?

Itanium 2 9050:
- Dual-core
- **24 MB L3$**

Source: David Patterson, UC Berkeley, HPEC keynote, Oct 2004 (*http://www.ll.mit.edu/HPEC/agendas/proc04/invited/patterson_keynote.pdf*)



TPC-C profile (courtesy Ravi Rajwar, Intel Research)



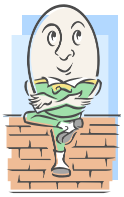
Quiz: What Does It Cost?

- Code:

```
int i = *pi1 + *pi2;
double d = *pd1 * *pd2;
size_t hash = pobj->GetHashCode();
ofstream out( “output.txt”);
out << “i = ” << i << “, d = ” << d << “, hash = ” << hash << endl;
```

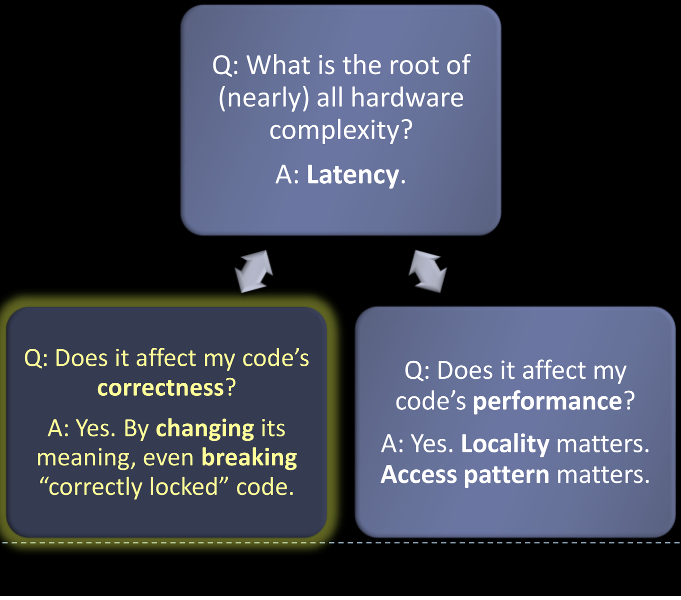
- Sample costs on modern microprocessors:
  - Floating-point multiply: Often .25 to 4 cycles
  - Memory access: Often 14 cycles (L2$) or 200 cycles (DRAM)
  - File open and write: Beyond in-memory buffering, who knows?
    - On a local HDD? File system disk accesses (seeks, rotations, contention).
    - Using a file system plugin/extension? Examples: On-demand virus scanning (compute); ZIP used as a folder (add navigation seek/compress/compute).
    - On a network share? Protocol layers, hops, translations, plus all latencies.
    - On a flash drive? Better/worse dep. on access patterns (e.g., no rotation).
  - Virtual function call? Your guess is as good as mine.

Memory Latency Is the Root of Most Evil



- **The vast majority of your hardware's complexity is a direct result of ever more heroic attempts to hide the Memory Wall.**
  - In CPUs, chipsets, memory subsystems, and disk subsystems.
  - In making the memory hierarchy ever deeper (e.g., flash thumbdrives used by the OS for more caching; flash memory embedded directly on hard drives).
  - Hardware is sometimes even willing to change the meaning of your code, and possibly break it, just to hide memory latency and make the code run faster.
- Latency as the root of most evil is a unifying theme of this talk, and many other talks.

Machine Architecture and You



Q: What is the root of (nearly) all hardware complexity?

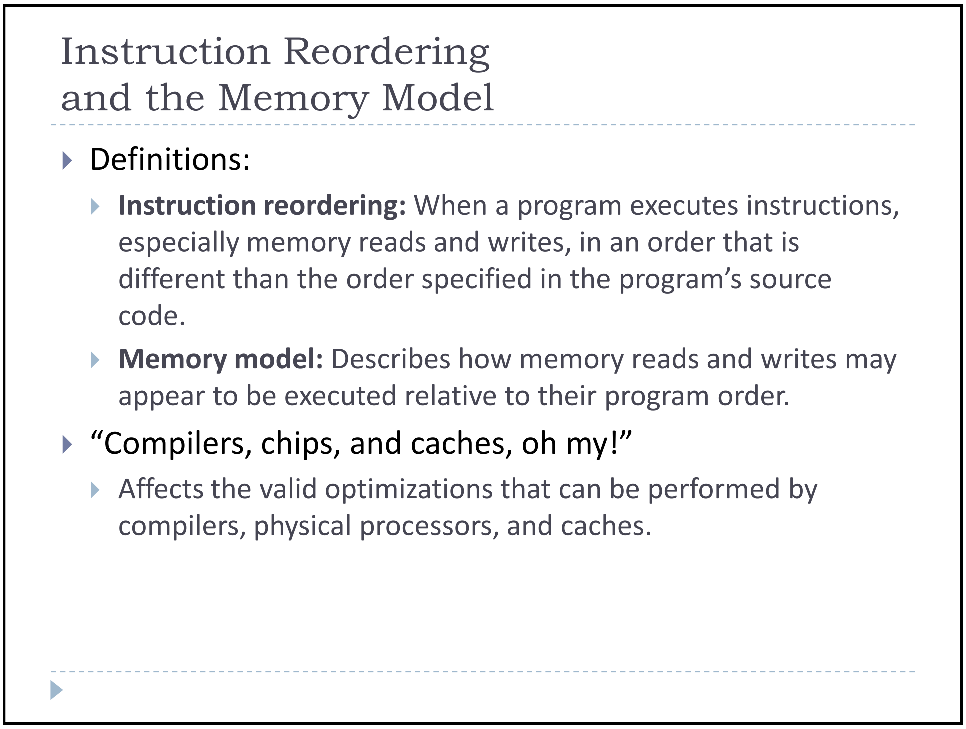
A: **Latency**.

Q: Does it affect my code's **correctness**?

A: Yes. By **changing** its meaning, even **breaking** "correctly locked" code.

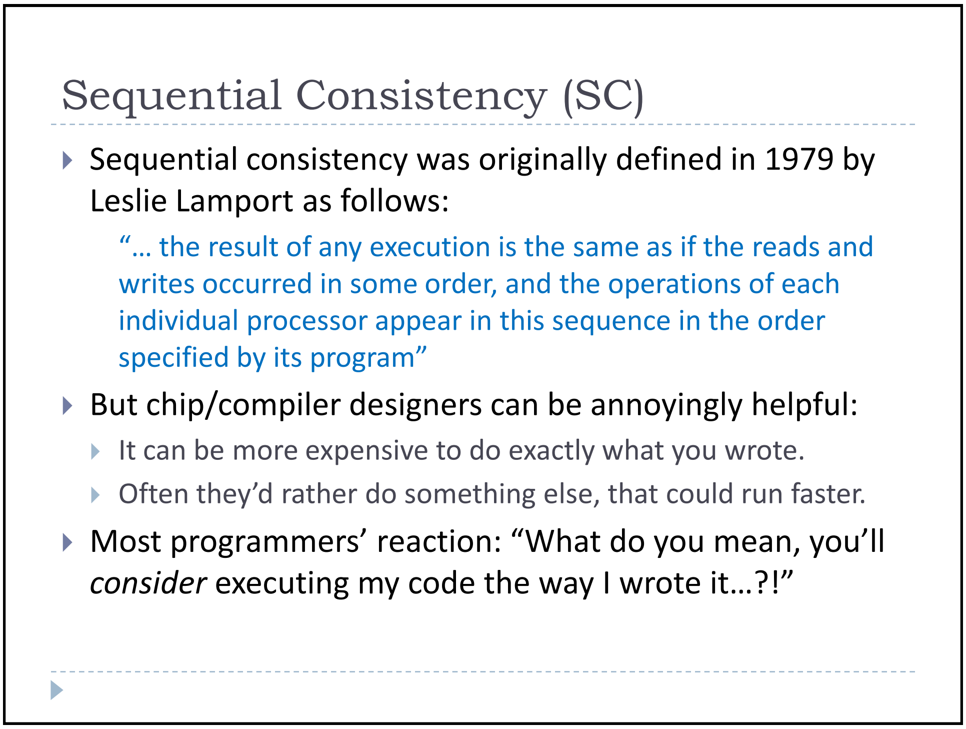
Q: Does it affect my code's **performance**?

A: Yes. **Locality** matters. **Access pattern** matters.



Instruction Reordering and the Memory Model

- Definitions:
  - **Instruction reordering:** When a program executes instructions, especially memory reads and writes, in an order that is different than the order specified in the program's source code.
  - **Memory model:** Describes how memory reads and writes may appear to be executed relative to their program order.
- "Compilers, chips, and caches, oh my!"
  - Affects the valid optimizations that can be performed by compilers, physical processors, and caches.



Sequential Consistency (SC)

- Sequential consistency was originally defined in 1979 by Leslie Lamport as follows:

  > "… the result of any execution is the same as if the reads and writes occurred in some order, and the operations of each individual processor appear in this sequence in the order specified by its program"

- But chip/compiler designers can be annoyingly helpful:
  - It can be more expensive to do exactly what you wrote.
  - Often they'd rather do something else, that could run faster.
- Most programmers' reaction: "What do you mean, you'll *consider* executing my code the way I wrote it…?!"

Dekker's and Peterson's Algorithms

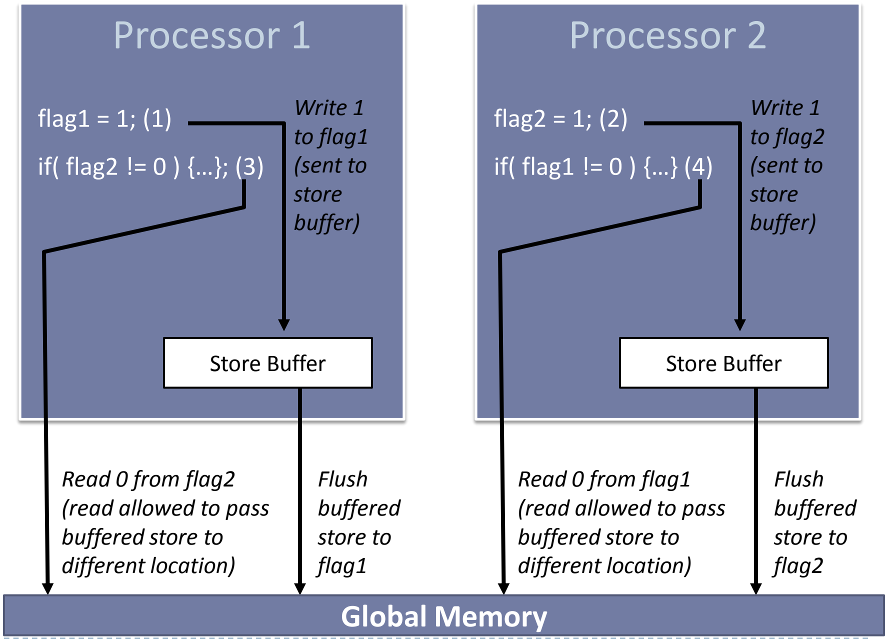
- Consider (flags are shared and atomic, initially zero):
  - Thread 1:

```
flag1 = 1;                // a: declare intent to enter
if( flag2 != 0 ) { ... }  // b: detect and resolve contention
// enter critical section
```

  - Thread 2:

```
flag2 = 1;                // c: declare intent to enter
if( flag1 != 0 ) { ... }  // d: detect and resolve contention
// enter critical section
```

- Could both threads enter the critical region?
  - **Maybe:** If a can pass b, and c can pass d, we could get b→d→a→c.
  - Solution 1 (good): Use a suitable atomic type (e.g., Java/.NET "volatile", C++0x **std::atomic<>**) for the flag variables.
  - Solution 2 (good?): Use system locks instead of rolling your own.
  - Solution 3 (problematic): Write a memory barrier after a and c.



Processor 1

flag1 = 1; (1)

if(flag2 != 0) {…}; (3)

Write 1 to flag1 (sent to store buffer)

Store Buffer

Read 0 from flag2 (read allowed to pass buffered store to different location)

Flush buffered store to flag1

Processor 2

flag2 = 1; (2)

if(flag1 != 0) {…} (4)

Write 1 to flag2 (sent to store buffer)

Store Buffer

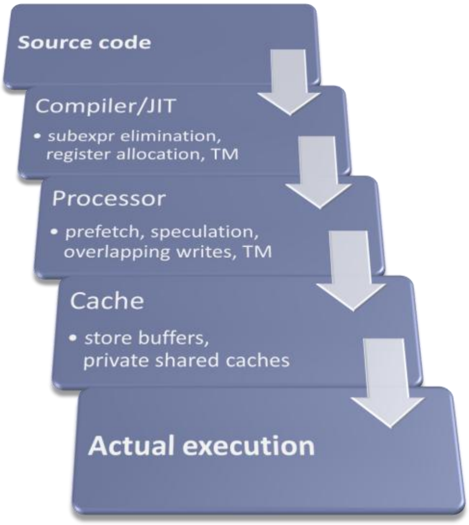
Read 0 from flag1 (read allowed to pass buffered store to different location)

Flush buffered store to flag2

Global Memory

Transformations: ***Reordering + invention + removal***

- The level at which the transformation happens is (usually) invisible to the programmer.
- The only thing that matters to the programmer is that a correctly synchronized program behaves as though:
  - The order in which memory operations are actually executed is equivalent to some sequential execution according to program source order.
  - Each write is visible to all processors at the same time.
- Tools and hardware (should) try to maintain that illusion. Sometimes they don't. We'll see why, and what you can do.



Source code

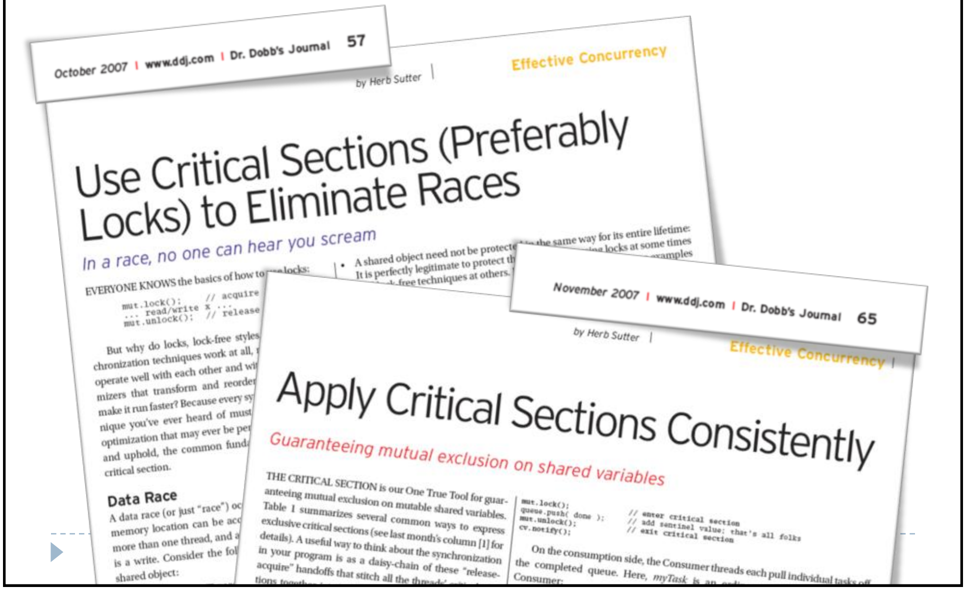
Compiler/JIT
- subexpr elimination, register allocation, TM

Processor
- prefetch, speculation, overlapping writes, TM

Cache
- store buffers, private shared caches

Actual execution

On the Back Table



October 2007 | www.ddj.com | Dr. Dobb's Journal 57

by Herb Sutter | Effective Concurrency

Use Critical Sections (Preferably Locks) to Eliminate Races

In a race, no one can hear you scream

EVERYONE KNOWS the basics of how to use locks:

```
mut.lock();         // acquire
... read/write x ...
mut.unlock();       // release
```

But why do locks, lock-free styles
chronization techniques work at all,
operate well with each other and wi
mizers that transform and reorder
make it run faster? Because every sy
nique you've ever heard of must
optimization that may ever be pe
and uphold, the common fund
critical section.

Data Race

A data race (or just "race") oc
memory location can be acc
more than one thread, and a
is a write. Consider the fol
shared object:

- A shared object need not be protecte... the same way for its entire lifetime:
It is perfectly legitimate to protect th... locks at some times
...-free techniques at others. ... examples

November 2007 | www.ddj.com | Dr. Dobb's Journal 65

by Herb Sutter | Effective Concurrency |

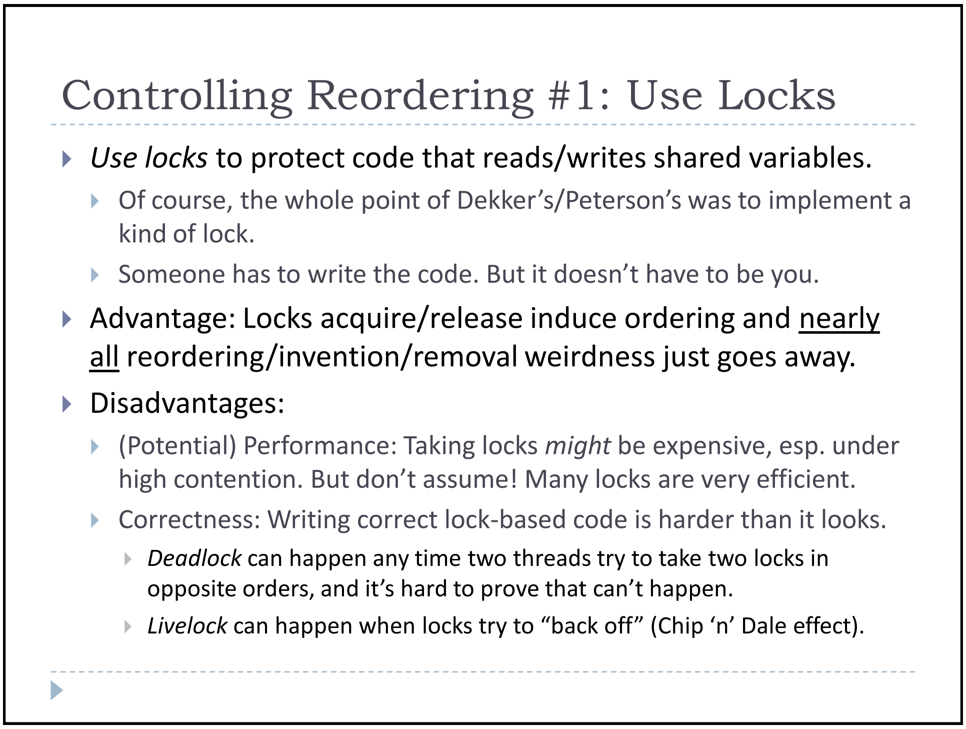
Apply Critical Sections Consistently

Guaranteeing mutual exclusion on shared variables

THE CRITICAL SECTION is our One True Tool for guaranteeing mutual exclusion on mutable shared variables. Table 1 summarizes several common ways to express exclusive critical sections (see last month's column [1] for details). A useful way to think about the synchronization in your program is as a daisy-chain of these "release-acquire" handoffs that stitch all the threads' ...

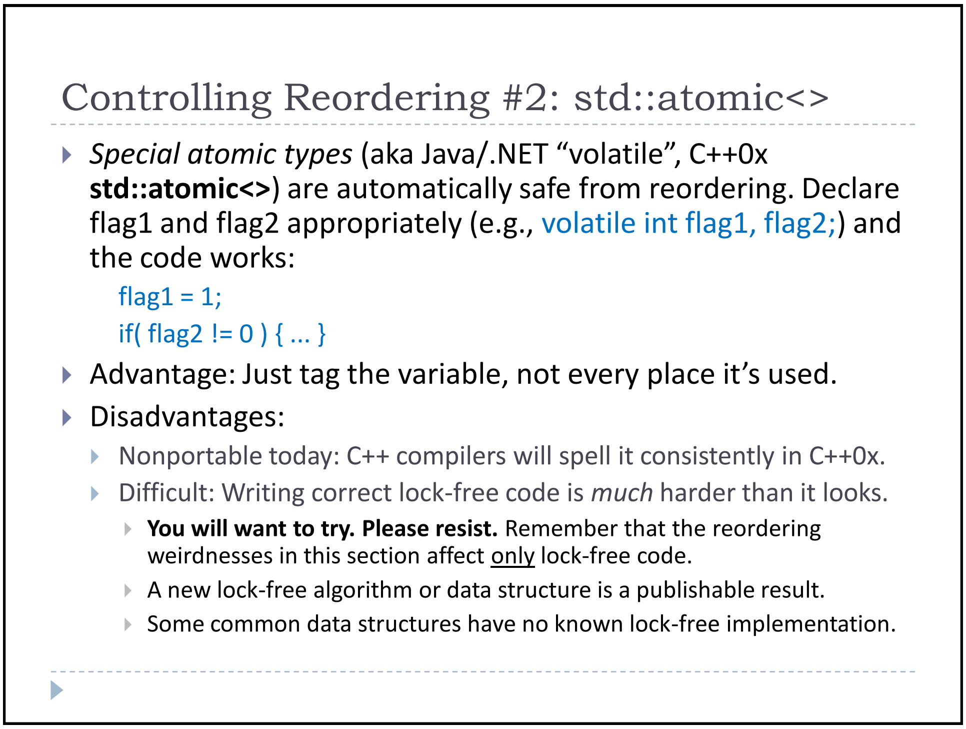
```
mut.lock();              // enter critical section
queue.push( done );      // add sentinel value: that's all folks
mut.unlock();            // exit critical section
cv.notify();
```

On the consumption side, the Consumer threads each pull individual tasks off the completed queue. Here, *myTask* is an ...
Consumer:



Controlling Reordering #1: Use Locks

- *Use locks* to protect code that reads/writes shared variables.
  - Of course, the whole point of Dekker's/Peterson's was to implement a kind of lock.
  - Someone has to write the code. But it doesn't have to be you.
- Advantage: Locks acquire/release induce ordering and nearly all reordering/invention/removal weirdness just goes away.
- Disadvantages:
  - (Potential) Performance: Taking locks *might* be expensive, esp. under high contention. But don't assume! Many locks are very efficient.
  - Correctness: Writing correct lock-based code is harder than it looks.
    - *Deadlock* can happen any time two threads try to take two locks in opposite orders, and it's hard to prove that can't happen.
    - *Livelock* can happen when locks try to "back off" (Chip 'n' Dale effect).

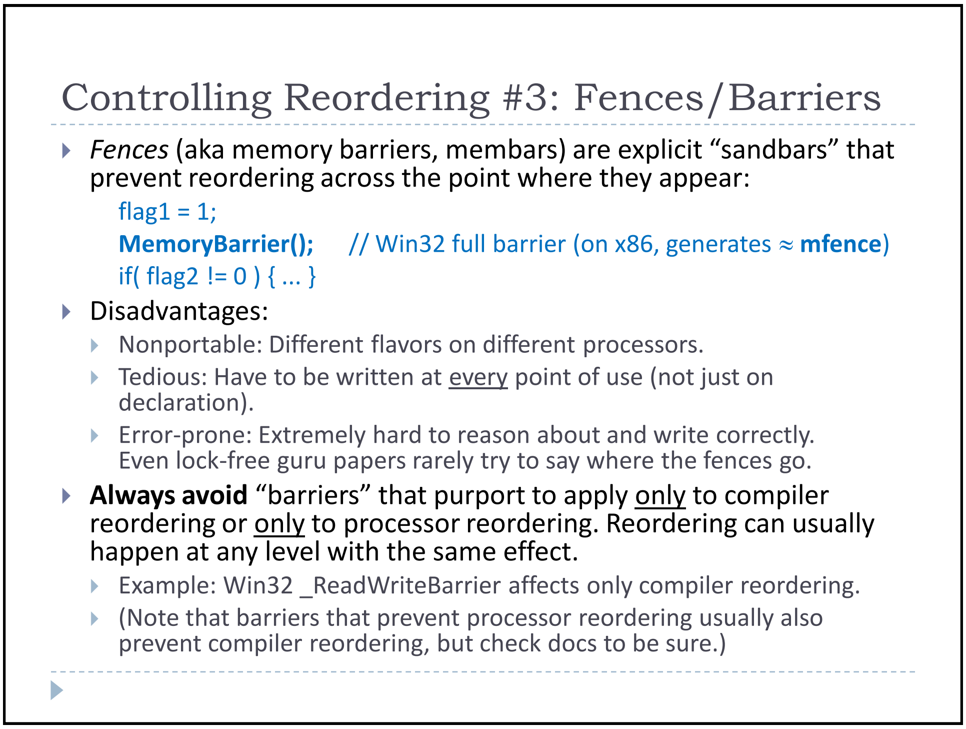


Controlling Reordering #2: std::atomic<>

- *Special atomic types* (aka Java/.NET "volatile", C++0x **std::atomic<>**) are automatically safe from reordering. Declare flag1 and flag2 appropriately (e.g., volatile int flag1, flag2;) and the code works:

```
flag1 = 1;
if( flag2 != 0 ) { ... }
```

- Advantage: Just tag the variable, not every place it's used.
- Disadvantages:
  - Nonportable today: C++ compilers will spell it consistently in C++0x.
  - Difficult: Writing correct lock-free code is *much* harder than it looks.
    - **You will want to try. Please resist.** Remember that the reordering weirdnesses in this section affect only lock-free code.
    - A new lock-free algorithm or data structure is a publishable result.
    - Some common data structures have no known lock-free implementation.

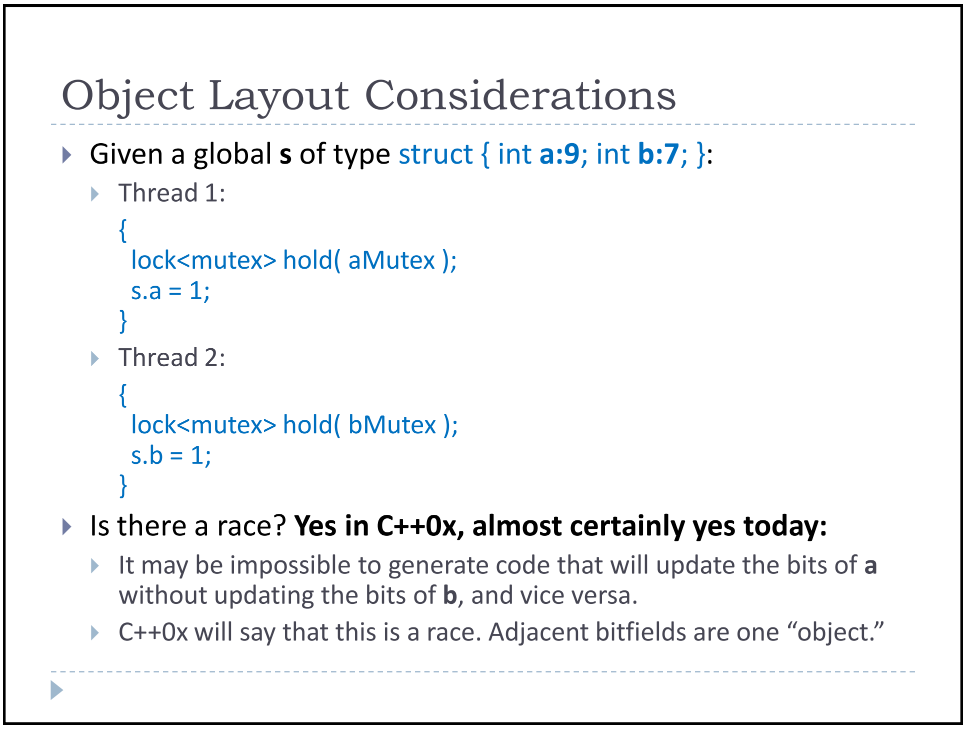


Controlling Reordering #3: Fences/Barriers

- *Fences* (aka memory barriers, membars) are explicit "sandbars" that prevent reordering across the point where they appear:

```
flag1 = 1;
MemoryBarrier();     // Win32 full barrier (on x86, generates ≈ mfence)
if( flag2 != 0 ) { ... }
```

- Disadvantages:
  - Nonportable: Different flavors on different processors.
  - Tedious: Have to be written at <u>every</u> point of use (not just on declaration).
  - Error-prone: Extremely hard to reason about and write correctly. Even lock-free guru papers rarely try to say where the fences go.
- **Always avoid** "barriers" that purport to apply <u>only</u> to compiler reordering or <u>only</u> to processor reordering. Reordering can usually happen at any level with the same effect.
  - Example: Win32 _ReadWriteBarrier affects only compiler reordering.
  - (Note that barriers that prevent processor reordering usually also prevent compiler reordering, but check docs to be sure.)



Object Layout Considerations

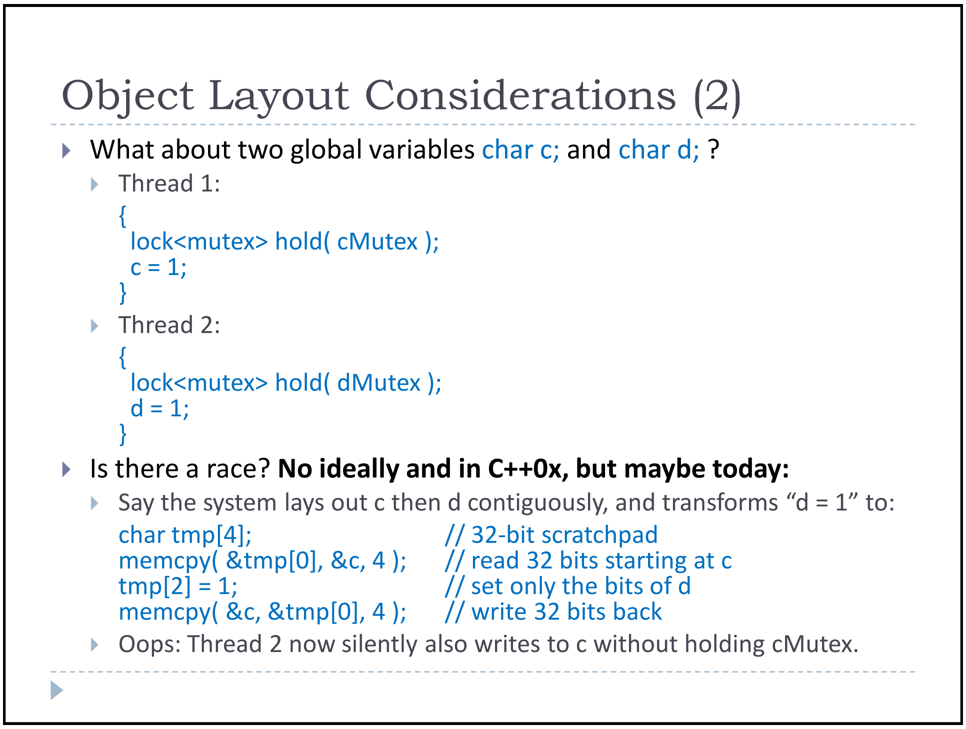
- Given a global **s** of type struct { int **a:9**; int **b:7**; }:
  - Thread 1:

```
{
 lock<mutex> hold( aMutex );
 s.a = 1;
}
```

  - Thread 2:

```
{
 lock<mutex> hold( bMutex );
 s.b = 1;
}
```

- Is there a race? **Yes in C++0x, almost certainly yes today:**
  - It may be impossible to generate code that will update the bits of **a** without updating the bits of **b**, and vice versa.
  - C++0x will say that this is a race. Adjacent bitfields are one "object."



Object Layout Considerations (2)

- What about two global variables char c; and char d; ?
  - Thread 1:

```
{
 lock<mutex> hold( cMutex );
 c = 1;
}
```

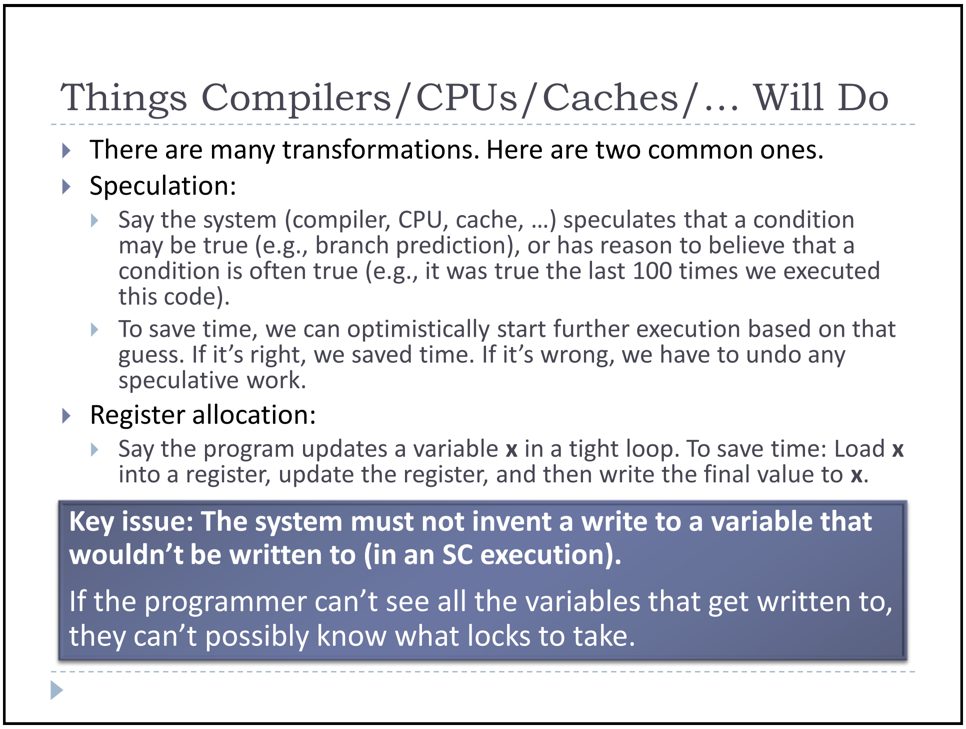
  - Thread 2:

```
{
 lock<mutex> hold( dMutex );
 d = 1;
}
```

- Is there a race? **No ideally and in C++0x, but maybe today:**
  - Say the system lays out c then d contiguously, and transforms "d = 1" to:

```
char tmp[4];                   // 32-bit scratchpad
memcpy( &tmp[0], &c, 4 );      // read 32 bits starting at c
tmp[2] = 1;                    // set only the bits of d
memcpy( &c, &tmp[0], 4 );      // write 32 bits back
```

  - Oops: Thread 2 now silently also writes to c without holding cMutex.



Things Compilers/CPUs/Caches/… Will Do

- There are many transformations. Here are two common ones.
- Speculation:
  - Say the system (compiler, CPU, cache, …) speculates that a condition may be true (e.g., branch prediction), or has reason to believe that a condition is often true (e.g., it was true the last 100 times we executed this code).
  - To save time, we can optimistically start further execution based on that guess. If it's right, we saved time. If it's wrong, we have to undo any speculative work.
- Register allocation:
  - Say the program updates a variable **x** in a tight loop. To save time: Load **x** into a register, update the register, and then write the final value to **x**.

Key issue: The system must not invent a write to a variable that wouldn't be written to (in an SC execution).

If the programmer can't see all the variables that get written to, they can't possibly know what locks to take.

A General Pattern

- Consider (where x is a shared variable):

```
if( cond )
  lock x
…
if( cond )
  use x
…
if( cond )
  unlock x
```

- Q: Is this pattern safe?
- A: In theory, yes. In reality, maybe not…

Speculation

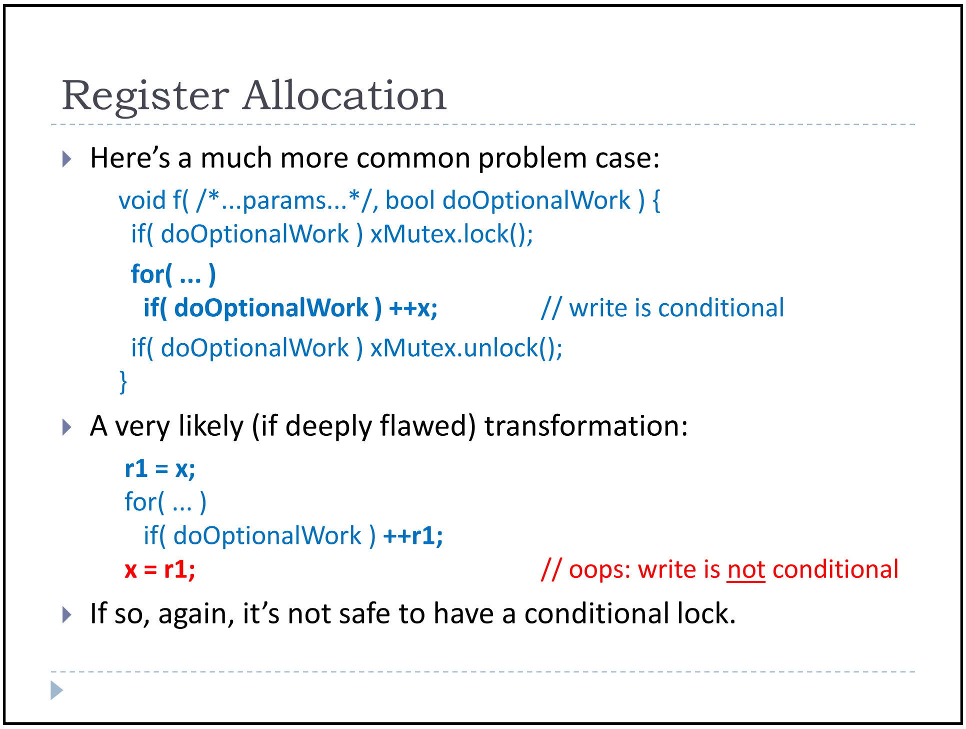
- Consider (where x is a shared variable):

```
if( cond )
  x = 42;
```

- Say the system (compiler, CPU, cache, …) speculates (predicts, guesses, measures) that **cond** (may be, will be, often is) true. Can this be transformed to:

```
r1 = x;          // read what's there
x = 42;          // perform an optimistic write
if( !cond )      // check if we guessed wrong
  x = r1;        // oops: back-out write is not SC
```

- In theory, No… **but on some implementations, Maybe.**
  - Same key issue: Inventing a write to a location that would never be written to in an SC execution.
  - If this happens, it can break patterns that conditionally take a lock.



Register Allocation

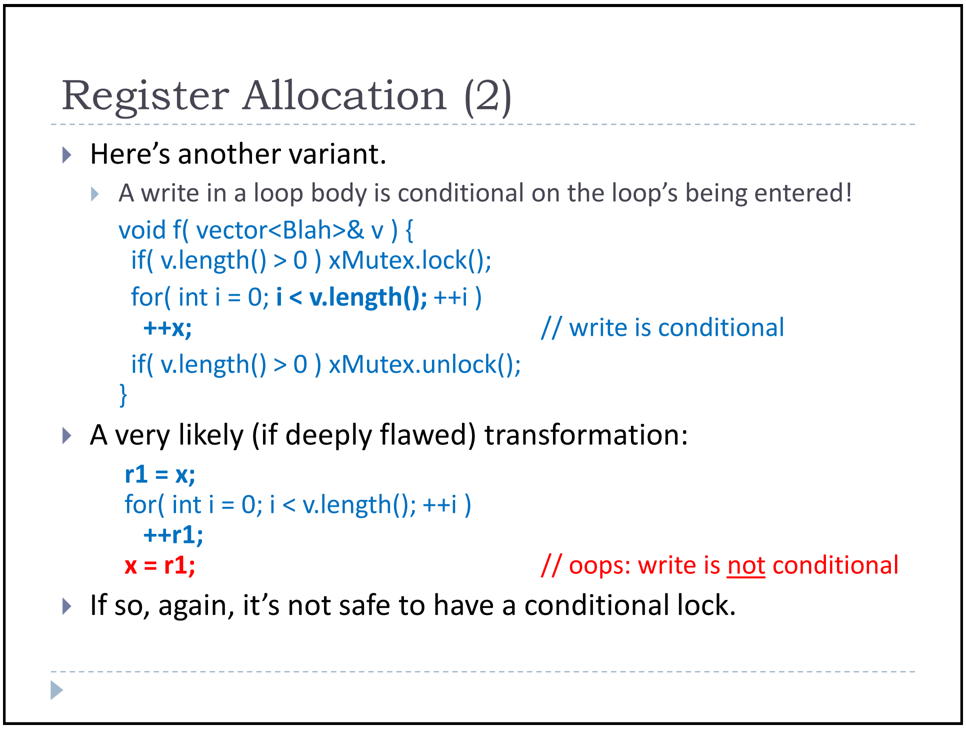
- Here’s a much more common problem case:

```
void f( /*...params...*/, bool doOptionalWork ) {
  if( doOptionalWork ) xMutex.lock();
  for( ... )
    if( doOptionalWork ) ++x;          // write is conditional
  if( doOptionalWork ) xMutex.unlock();
}
```

- A very likely (if deeply flawed) transformation:

```
r1 = x;
for( ... )
  if( doOptionalWork ) ++r1;
x = r1;                                // oops: write is not conditional
```

- If so, again, it’s not safe to have a conditional lock.



Register Allocation (2)

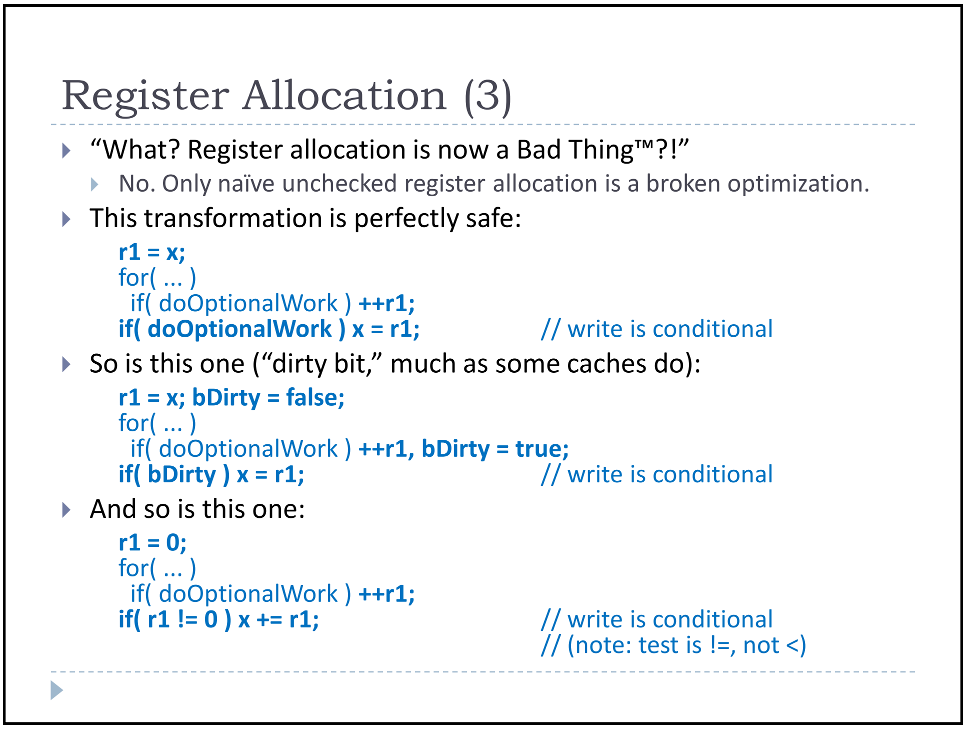
- Here’s another variant.
  - A write in a loop body is conditional on the loop’s being entered!

```
void f( vector<Blah>& v ) {
  if( v.length() > 0 ) xMutex.lock();
  for( int i = 0; i < v.length(); ++i )
    ++x;                               // write is conditional
  if( v.length() > 0 ) xMutex.unlock();
}
```

- A very likely (if deeply flawed) transformation:

```
r1 = x;
for( int i = 0; i < v.length(); ++i )
  ++r1;
x = r1;                                // oops: write is not conditional
```

- If so, again, it’s not safe to have a conditional lock.



Register Allocation (3)

- "What? Register allocation is now a Bad Thing™?!"
  - No. Only naïve unchecked register allocation is a broken optimization.
- This transformation is perfectly safe:

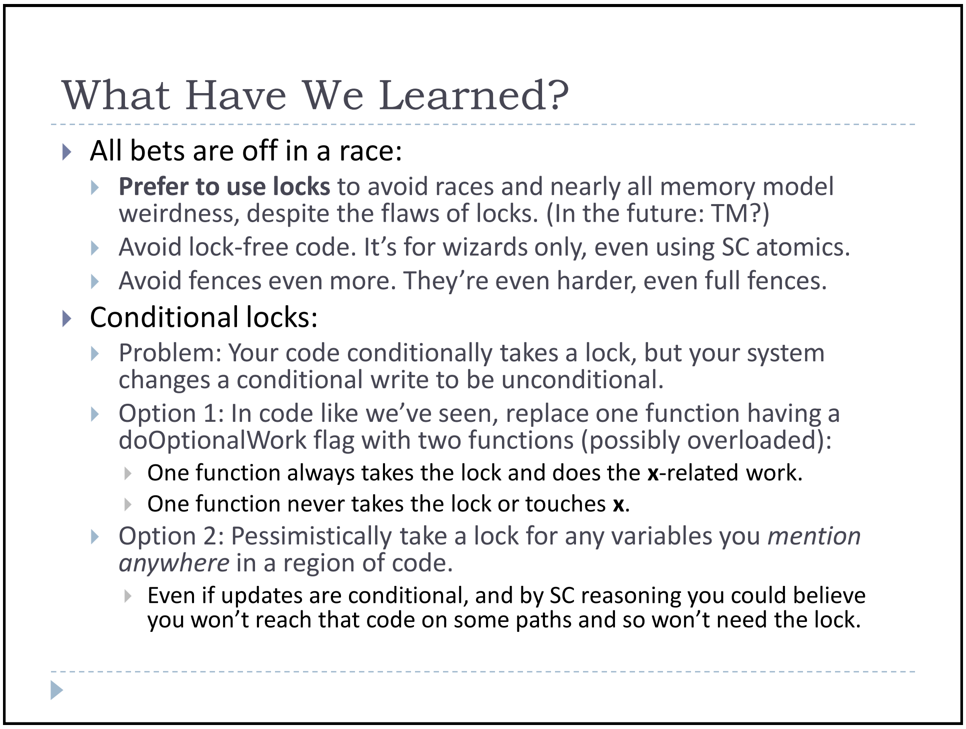
```
r1 = x;
for( ... )
  if( doOptionalWork ) ++r1;
if( doOptionalWork ) x = r1;            // write is conditional
```

- So is this one ("dirty bit," much as some caches do):

```
r1 = x; bDirty = false;
for( ... )
  if( doOptionalWork ) ++r1, bDirty = true;
if( bDirty ) x = r1;                    // write is conditional
```

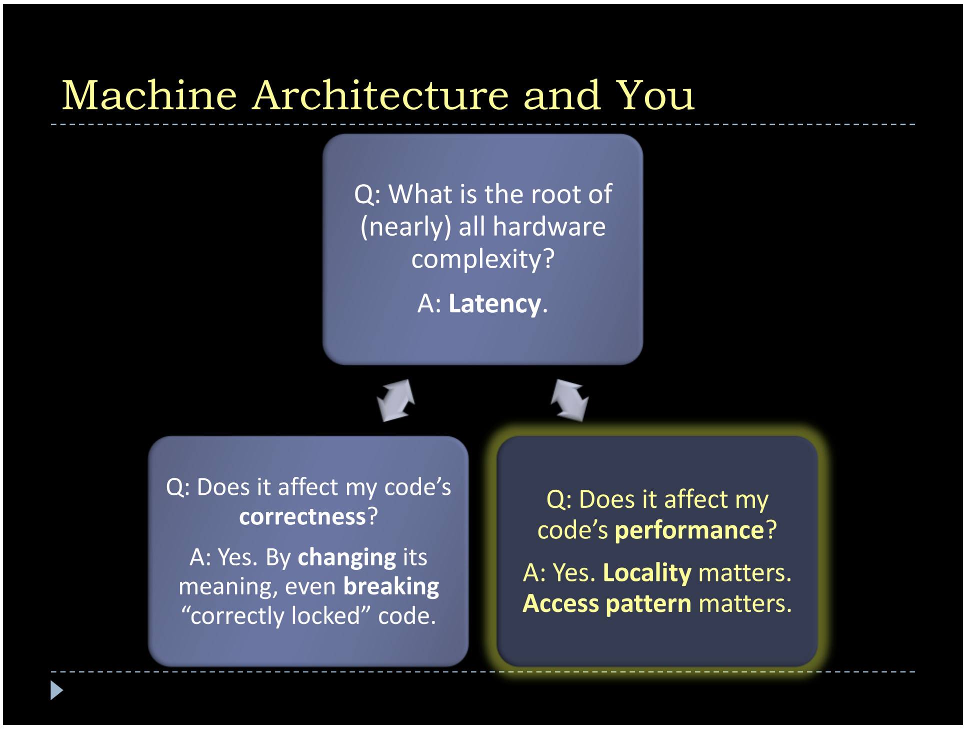
- And so is this one:

```
r1 = 0;
for( ... )
  if( doOptionalWork ) ++r1;
if( r1 != 0 ) x += r1;                  // write is conditional
                                        // (note: test is !=, not <)
```



What Have We Learned?

- All bets are off in a race:
  - **Prefer to use locks** to avoid races and nearly all memory model weirdness, despite the flaws of locks. (In the future: TM?)
  - Avoid lock-free code. It's for wizards only, even using SC atomics.
  - Avoid fences even more. They're even harder, even full fences.
- Conditional locks:
  - Problem: Your code conditionally takes a lock, but your system changes a conditional write to be unconditional.
  - Option 1: In code like we've seen, replace one function having a doOptionalWork flag with two functions (possibly overloaded):
    - One function always takes the lock and does the **x**-related work.
    - One function never takes the lock or touches **x**.
  - Option 2: Pessimistically take a lock for any variables you *mention anywhere* in a region of code.
    - Even if updates are conditional, and by SC reasoning you could believe you won't reach that code on some paths and so won't need the lock.



Machine Architecture and You

Q: What is the root of (nearly) all hardware complexity?

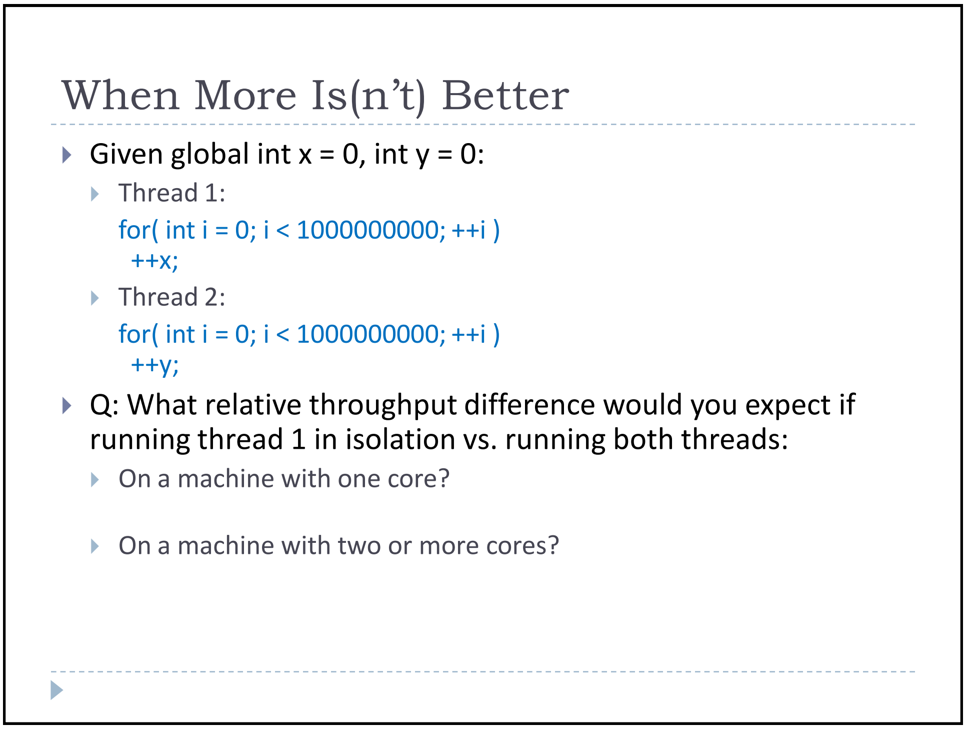
A: **Latency**.

Q: Does it affect my code's **correctness**?

A: Yes. By **changing** its meaning, even **breaking** "correctly locked" code.

Q: Does it affect my code's **performance**?

A: Yes. **Locality** matters. **Access pattern** matters.



When More Is(n't) Better

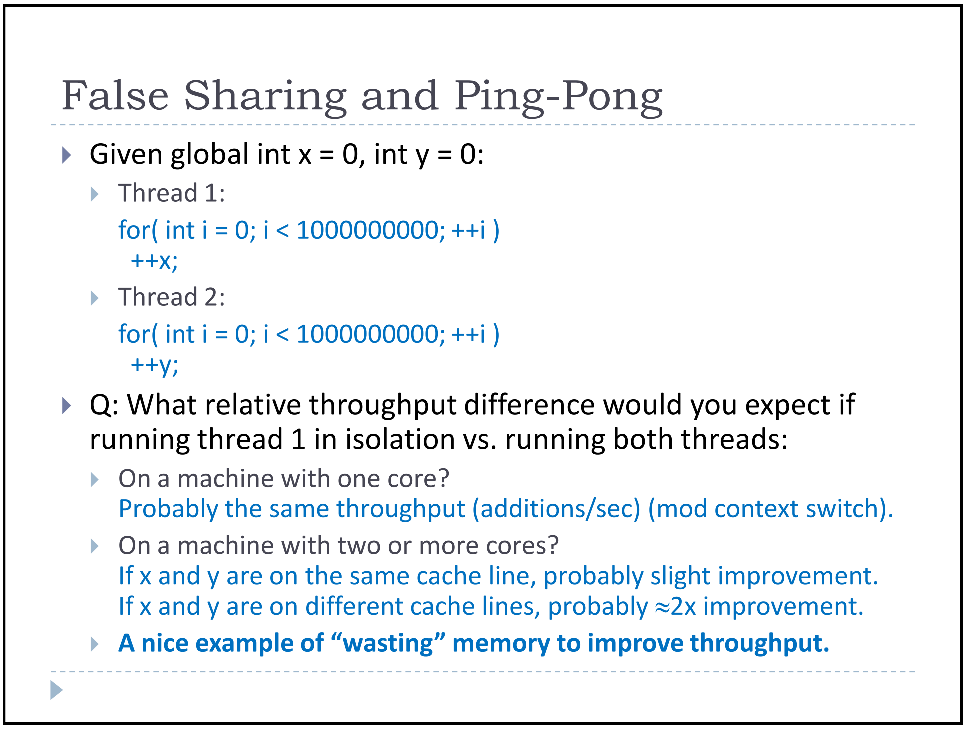
- Given global int x = 0, int y = 0:
  - Thread 1:

```
for( int i = 0; i < 1000000000; ++i )
  ++x;
```

  - Thread 2:

```
for( int i = 0; i < 1000000000; ++i )
  ++y;
```

- Q: What relative throughput difference would you expect if running thread 1 in isolation vs. running both threads:
  - On a machine with one core?
  - On a machine with two or more cores?

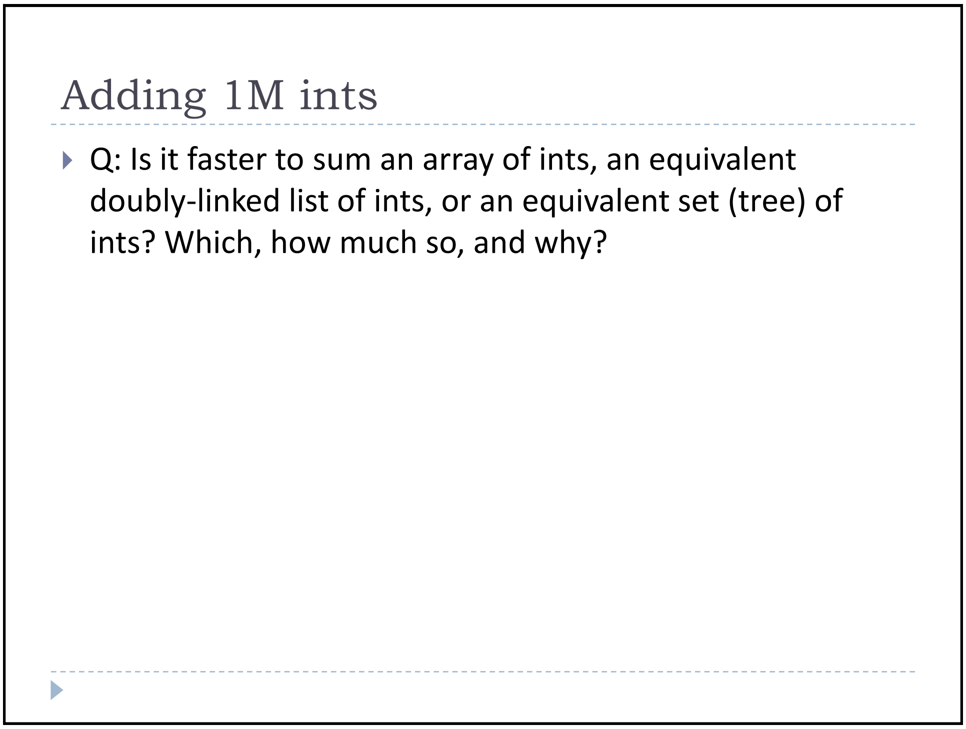


False Sharing and Ping-Pong

- Given global int x = 0, int y = 0:
  - Thread 1:

    ```
    for( int i = 0; i < 1000000000; ++i )
      ++x;
    ```
  - Thread 2:

    ```
    for( int i = 0; i < 1000000000; ++i )
      ++y;
    ```
- Q: What relative throughput difference would you expect if running thread 1 in isolation vs. running both threads:
  - On a machine with one core?
    Probably the same throughput (additions/sec) (mod context switch).
  - On a machine with two or more cores?
    If x and y are on the same cache line, probably slight improvement.
    If x and y are on different cache lines, probably ≈2x improvement.
  - **A nice example of "wasting" memory to improve throughput.**



Adding 1M ints

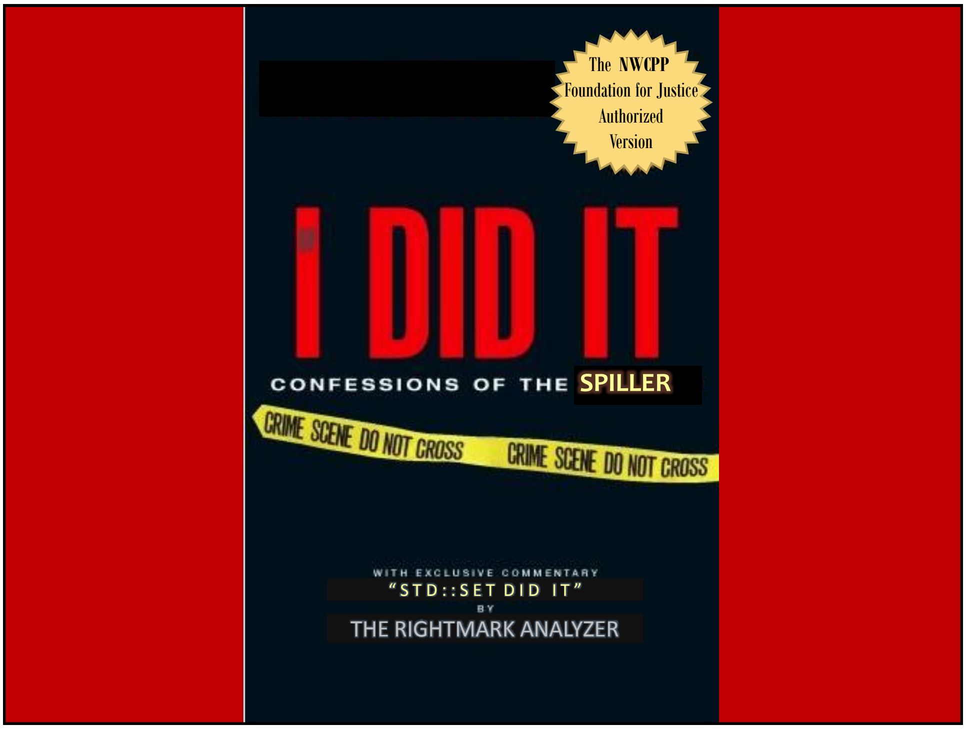
- Q: Is it faster to sum an array of ints, an equivalent doubly-linked list of ints, or an equivalent set (tree) of ints? Which, how much so, and why?

Working Set Effects: Storing 1M ints

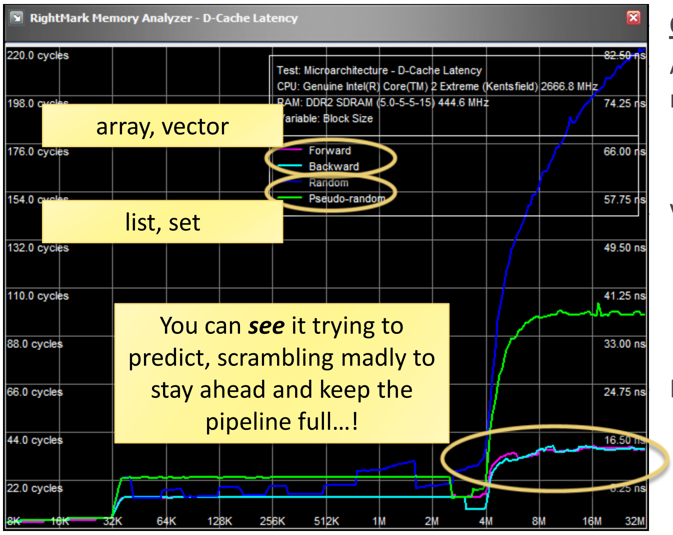
- Imagine 4B int/*, 64B cache lines (HW), 4KB memory pages (OS):

Working set arrangement	**Cache**		**Memory**	
	# lines	total	# pages	total
Perfectly contiguously (e.g., vector<int>)	65,536	4 MB	1,024	4 MB
Full cache lines, half-full pages (e.g., vector< array<int,512>*>)	65,536	4 MB	min 1,024 **max 2,048**	min 4 MB **max 8 MB**
5.33 ints per cache line (e.g., list<int>, 12B/node)	**196,608**	**12 MB**	**min 3,072** *max 327,680*	**min 12 MB** *max 1,311 MB*
3.2 ints per cache line (e.g., set<int>, 20B/node)	**327,680**	**20 MB**	**min 5,120** *max 327,680*	**min 20 MB** *max 1,311 MB*
	< Cache utilization > Cache contention		< Memory utilization > Disk activity	

- Chunking/packing at multiple levels ⇒ sometimes overheads multiply.



My Desktop Machine: 32K L1D$, 4M L2$



RightMark Memory Analyzer - D-Cache Latency

Test: Microarchitecture - D-Cache Latency
CPU: Genuine Intel(R) Core(TM) 2 Extreme (Kentsfield) 2666.8 MHz
RAM: DDR2 SDRAM (5.0-5-5-15) 444.6 MHz
Variable: Block Size

array, vector

list, set

You can ***see*** it trying to predict, scrambling madly to stay ahead and keep the pipeline full…!

Observations

Access patterns matter:

- Linear = not bad.
- Random = awful.

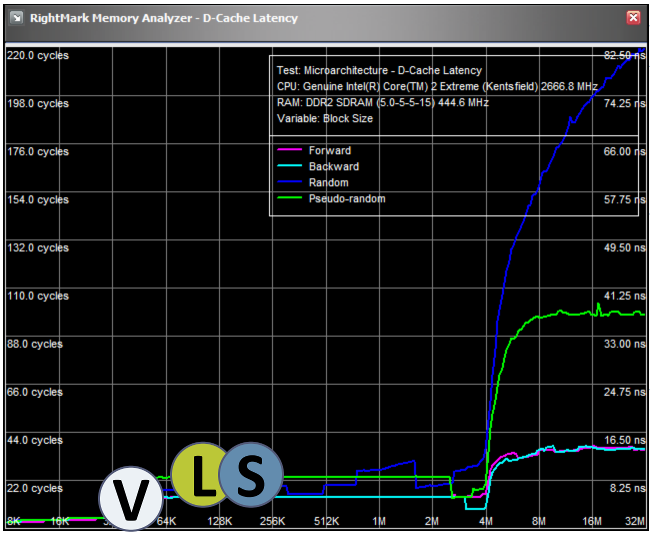
Vectors:

- Smaller = further left, faster.
- Often on lower curves = faster.

Lists and sets:

- Bigger = further right, slower.
- On higher curves = slower.

My Desktop Machine: 32K L1D$, 4M L2$



RightMark Memory Analyzer - D-Cache Latency

Test: Microarchitecture - D-Cache Latency
CPU: Genuine Intel(R) Core(TM) 2 Extreme (Kentsfield) 2666.8 MHz
RAM: DDR2 SDRAM (5.0-5-5-15) 444.6 MHz
Variable: Block Size

Observations

Access patterns matter:

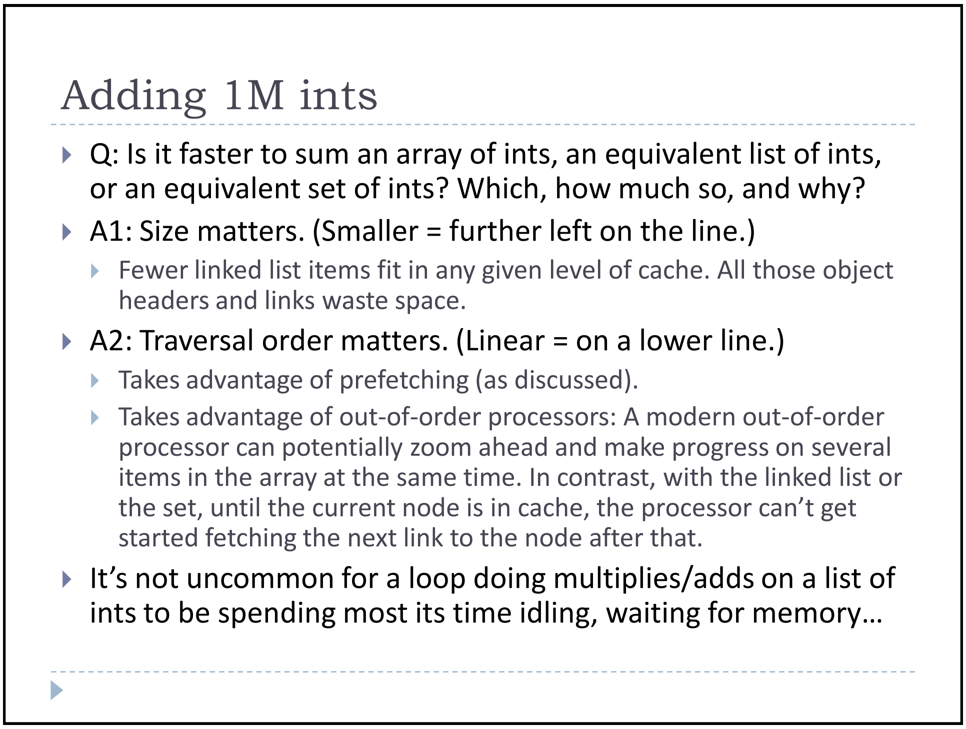
- Linear = not bad.
- Random = awful.

Vectors:

- Smaller = further left, faster.
- Often on lower curves = faster.

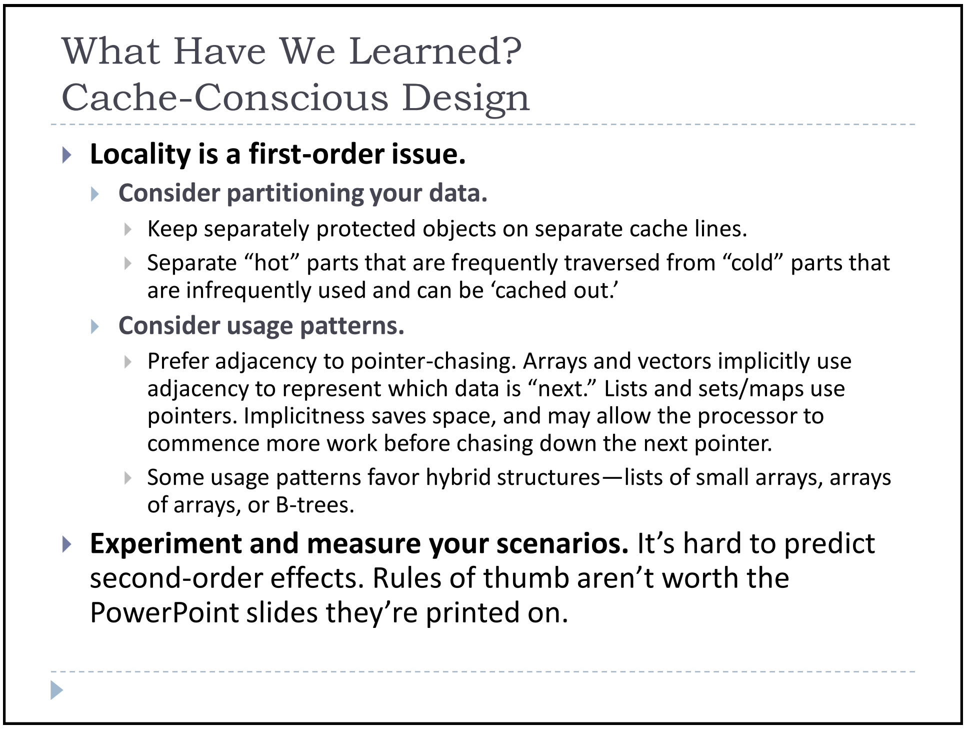
Lists and sets:

- Bigger = further right, slower.
- On higher curves = slower.



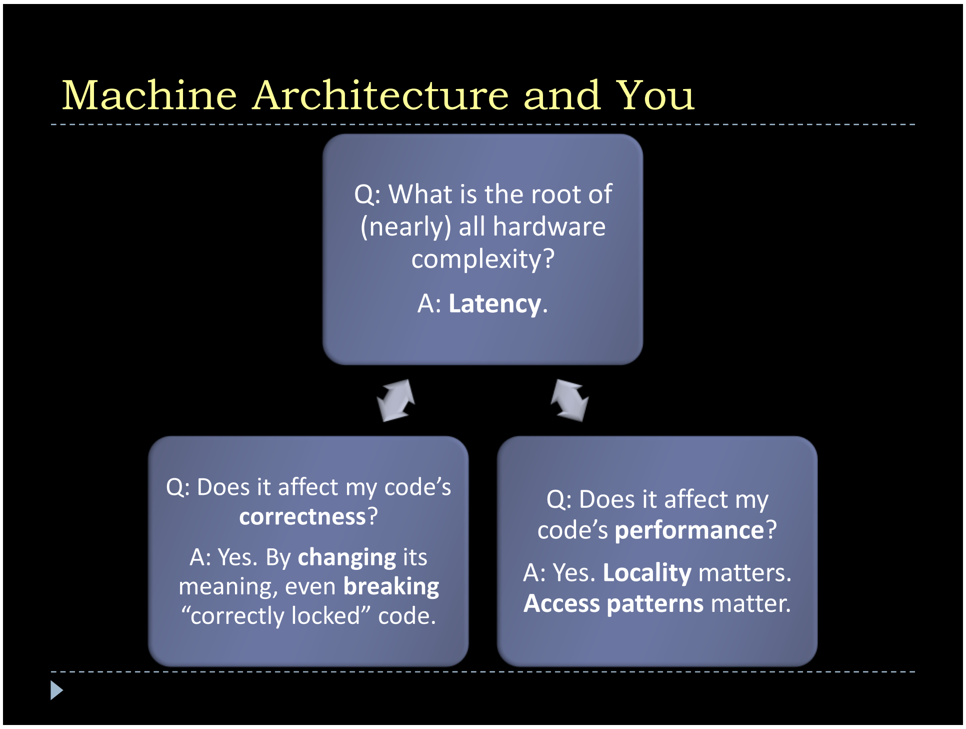
Adding 1M ints

- Q: Is it faster to sum an array of ints, an equivalent list of ints, or an equivalent set of ints? Which, how much so, and why?
- A1: Size matters. (Smaller = further left on the line.)
  - Fewer linked list items fit in any given level of cache. All those object headers and links waste space.
- A2: Traversal order matters. (Linear = on a lower line.)
  - Takes advantage of prefetching (as discussed).
  - Takes advantage of out-of-order processors: A modern out-of-order processor can potentially zoom ahead and make progress on several items in the array at the same time. In contrast, with the linked list or the set, until the current node is in cache, the processor can't get started fetching the next link to the node after that.
- It's not uncommon for a loop doing multiplies/adds on a list of ints to be spending most its time idling, waiting for memory…



What Have We Learned?
Cache-Conscious Design

- **Locality is a first-order issue.**
  - **Consider partitioning your data.**
    - Keep separately protected objects on separate cache lines.
    - Separate "hot" parts that are frequently traversed from "cold" parts that are infrequently used and can be 'cached out.'
  - **Consider usage patterns.**
    - Prefer adjacency to pointer-chasing. Arrays and vectors implicitly use adjacency to represent which data is "next." Lists and sets/maps use pointers. Implicitness saves space, and may allow the processor to commence more work before chasing down the next pointer.
    - Some usage patterns favor hybrid structures—lists of small arrays, arrays of arrays, or B-trees.
- **Experiment and measure your scenarios.** It's hard to predict second-order effects. Rules of thumb aren't worth the PowerPoint slides they're printed on.



Machine Architecture and You

Q: What is the root of (nearly) all hardware complexity?

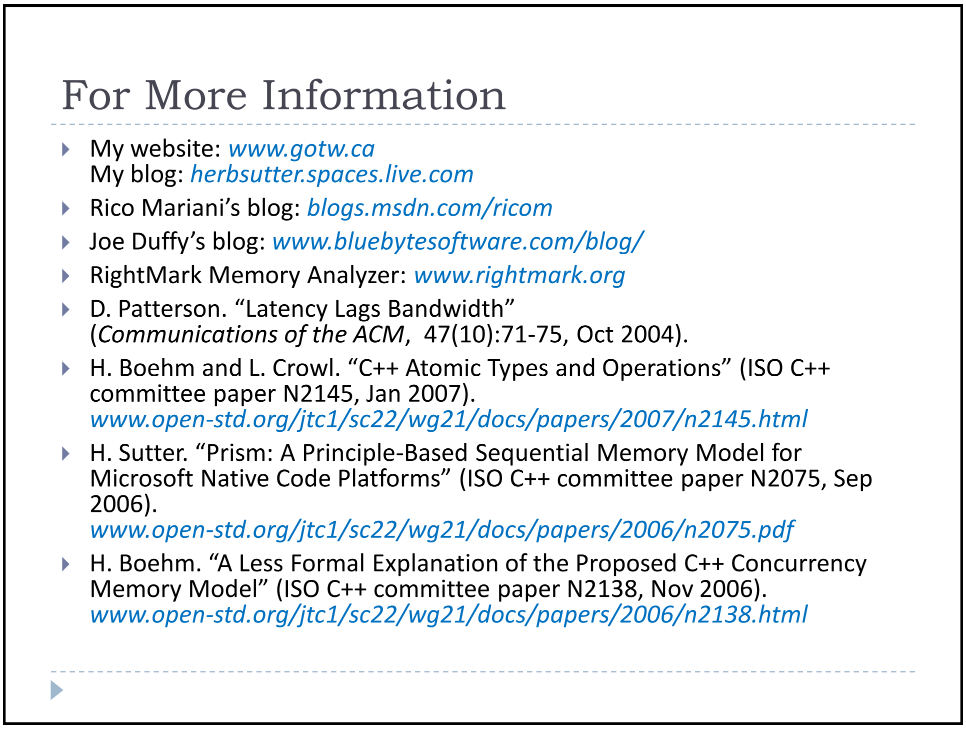
A: **Latency**.

Q: Does it affect my code's **correctness**?

A: Yes. By **changing** its meaning, even **breaking** "correctly locked" code.

Q: Does it affect my code's **performance**?

A: Yes. **Locality** matters. **Access patterns** matter.



For More Information

- My website: *www.gotw.ca*
  My blog: *herbsutter.spaces.live.com*
- Rico Mariani's blog: *blogs.msdn.com/ricom*
- Joe Duffy's blog: *www.bluebytesoftware.com/blog/*
- RightMark Memory Analyzer: *www.rightmark.org*
- D. Patterson. "Latency Lags Bandwidth" (*Communications of the ACM*, 47(10):71-75, Oct 2004).
- H. Boehm and L. Crowl. "C++ Atomic Types and Operations" (ISO C++ committee paper N2145, Jan 2007). *www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2145.html*
- H. Sutter. "Prism: A Principle-Based Sequential Memory Model for Microsoft Native Code Platforms" (ISO C++ committee paper N2075, Sep 2006). *www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2075.pdf*
- H. Boehm. "A Less Formal Explanation of the Proposed C++ Concurrency Memory Model" (ISO C++ committee paper N2138, Nov 2006). *www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2138.html*