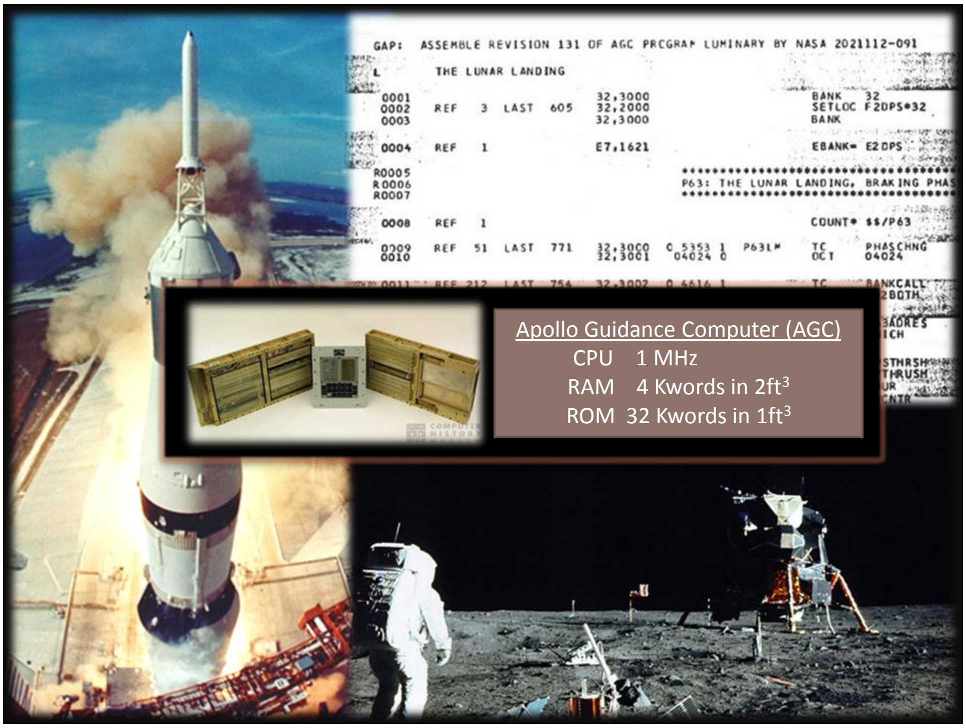
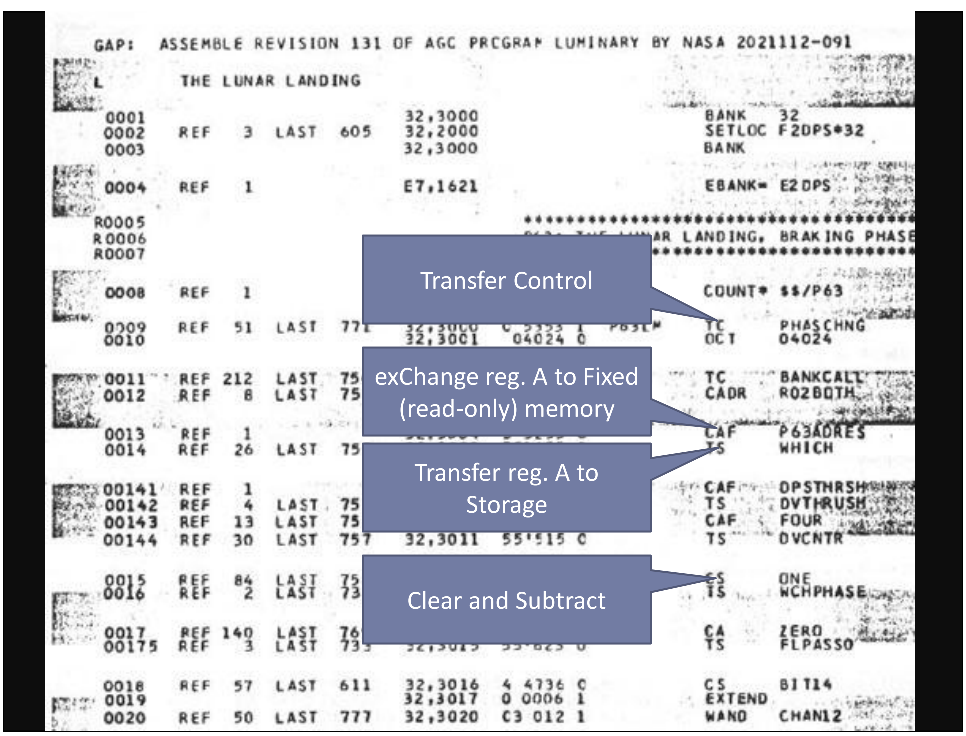
Machine Architecture

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++ 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 C++ the 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.


Apollo Guidance Computer (AGC)
CPU 1 MHz
RAM 4 Kwords in 2ft³
ROM 32 Kwords in 1ft³


GAP: ASSEMBLE REVISION 131 OF AGC PRCGRAM LUMINARY BY NASA 2021112-091
THE LUNAR LANDING
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:

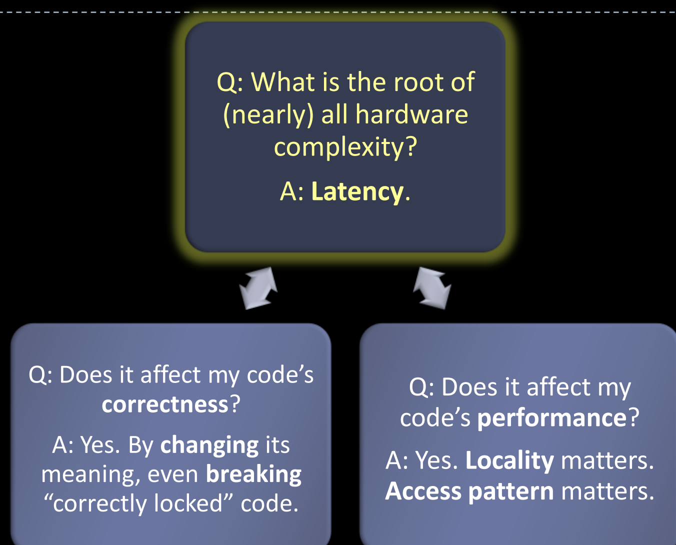
```
CAF    DPSTHRSH    // A = *delta_per_sec_thrust_h (??)
TS     DVTHRUSH    // *delta_v_thrust_h = A
CAF    FOUR        // A = *four
TS     DVCNTR      // *delta_v_counter = A
CS     ONE         // A = --*one (?)
TS     WCHPHASE    // *which_phase = A
```

- Modern C++ code:

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

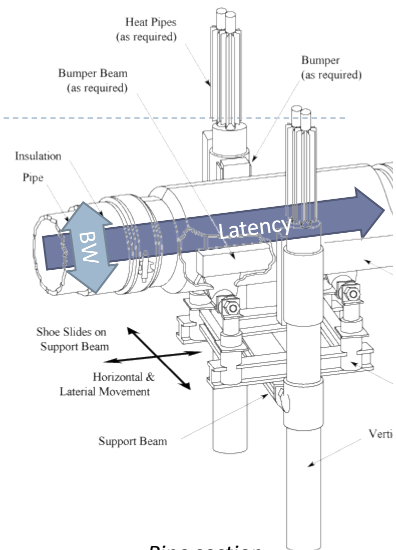
  - What can you say about the cost of each line?
  - What are the most to least expensive operations?

Machine Architecture and You

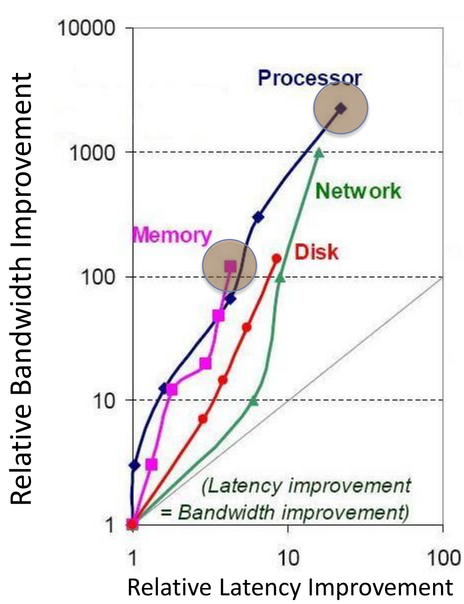


Bandwidth and Latency

- **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.
- Quoting Burton Smith: *"You can solve a bandwidth problem with money, but only God can solve 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*)

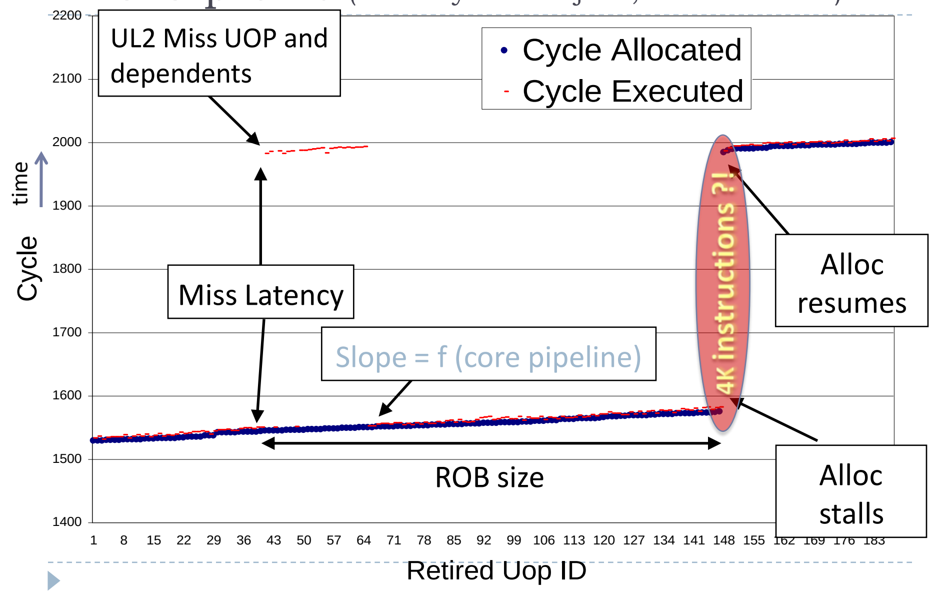
Measuring Memory Latency

	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

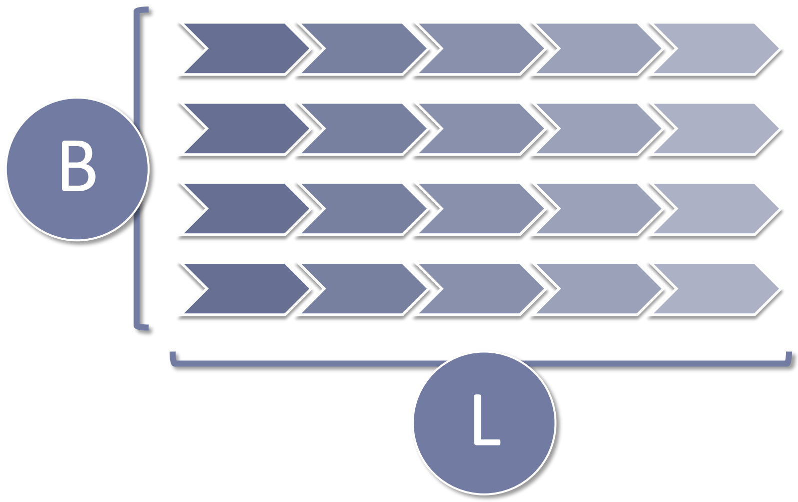
	AGC	1980 VAX-11/750	Modern Desktop	Improvement since 1980
Clock speed (MHz)	1	6	3,000	**+500x**
Memory size (RAM, MB)	0.007	2	2,000	+1,000x
Memory bandwidth (MB/s)		13	7,000 (read) 2,000 (write)	**+540x** +150x
Memory latency (ns)	~12,000	225	~70	+3x
Memory latency (cycles)	12	1.4	210	**-150x**
For comparison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

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



Little's Law

Bandwidth × Latency = Concurrency

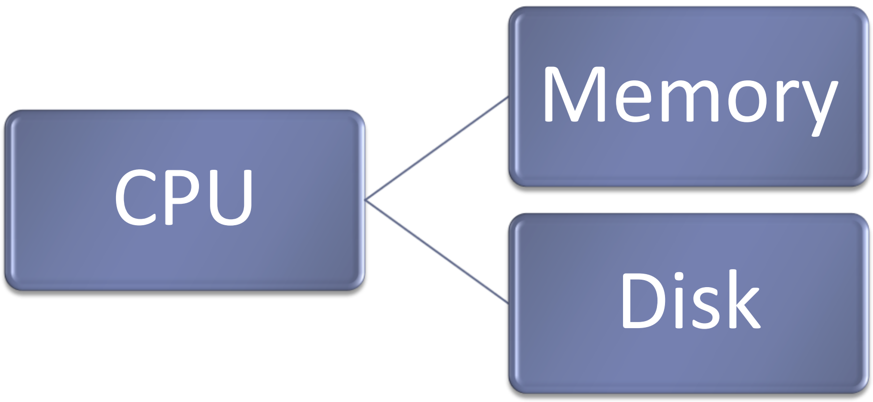


Q: So How Do We Cope With Latency? A: **Add Concurrency… Everywhere…**

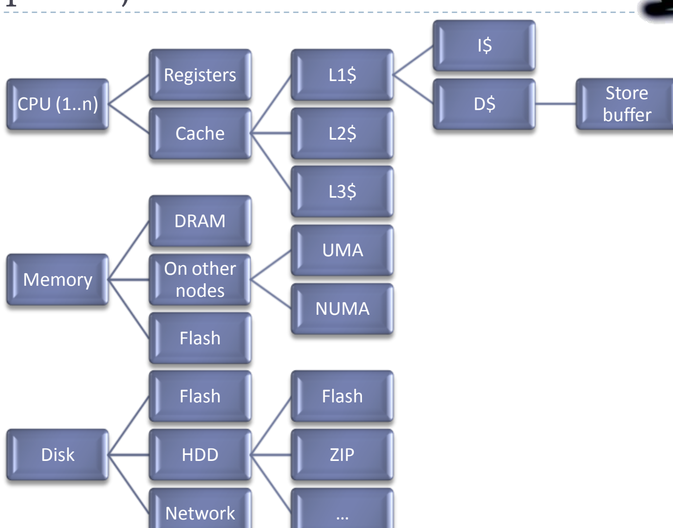
Strategy	Technique	Can affect your code?
Parallelize (leverage compute power)	**Pipeline, execute out of order ("OoO"):** Launch expensive memory operations earlier, and do other work while waiting.	**Yes**
	Add hardware threads: Have other work available for the *same CPU core* to perform while other 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!*

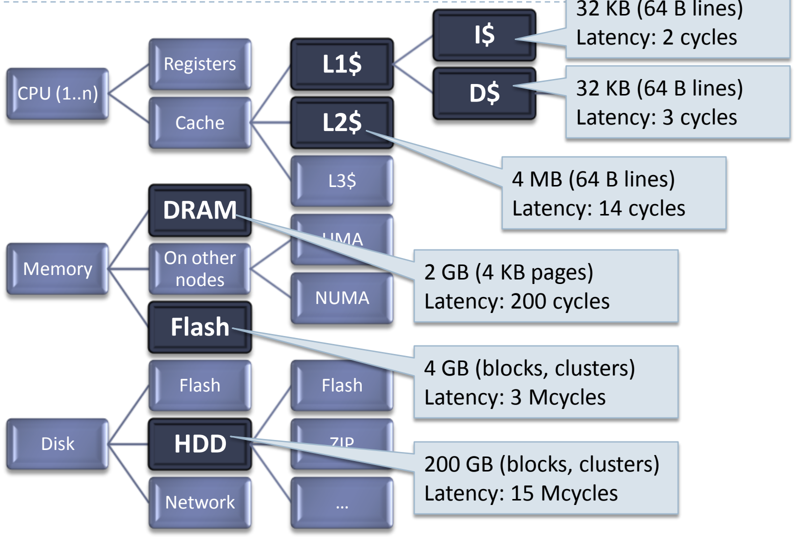
The Machine Everyone Programs For



The Actual Memory Hierarchy (Simplified)



Sample Values On My Desktop



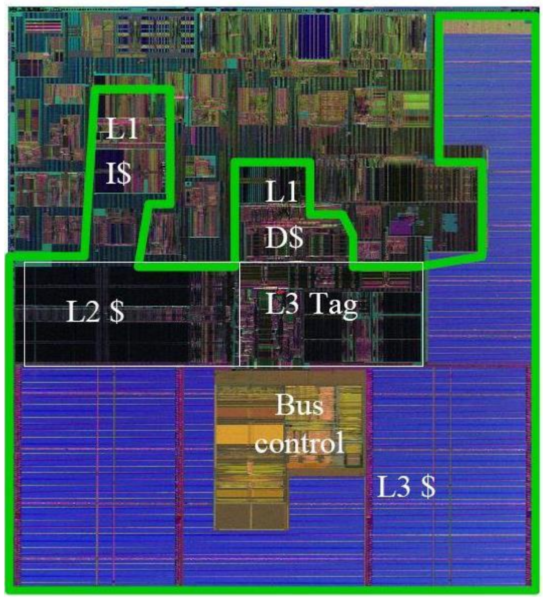
Quiz: What Does It Cost?

- Code:

```
int i = *pi2 + *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: 4 cycles
  - DRAM access: 200 cycles
  - 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).

Herb Sutter
Software Development Consultant
www.gotw.ca/training



Sample Modern CPU

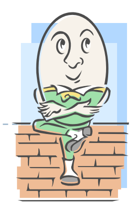
Original Itanium 2 had
211Mt, **85%** for cache:
- 16 KB L1I$
- 16 KB L1D$
- 256 KB L2$
- 3 MB L3$

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

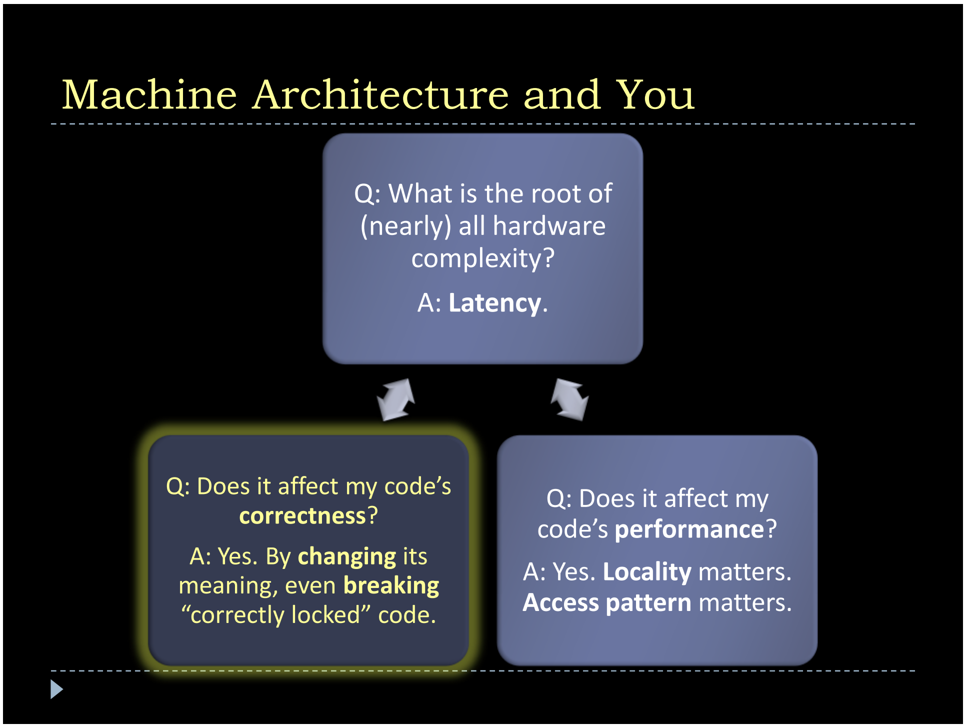
Itanium 2 9050:
- Dual-core
- **24 MB L3$**

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

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.



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

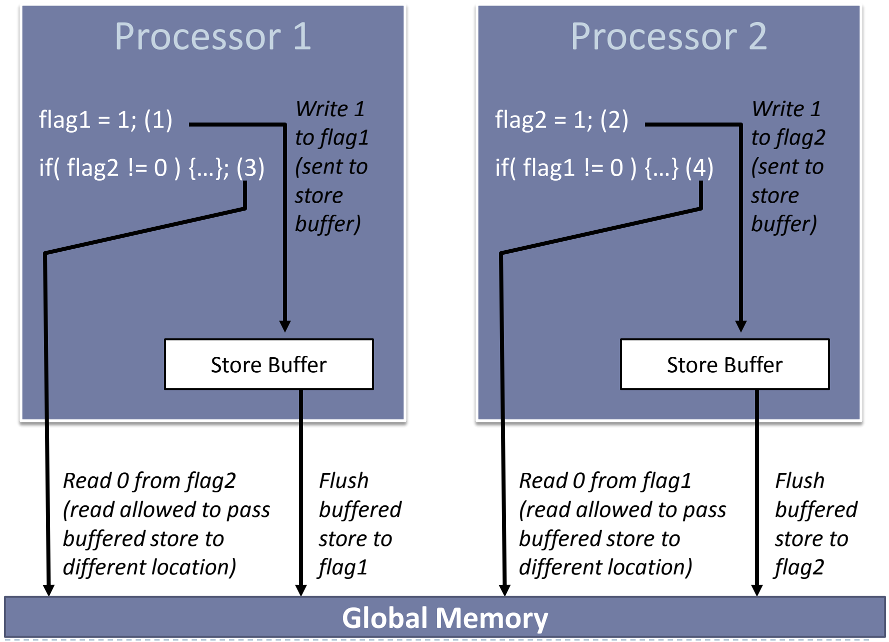
- Consider (flags are shared and atomic, initially zero):
  - Thread 1:

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

  - Thread 2:

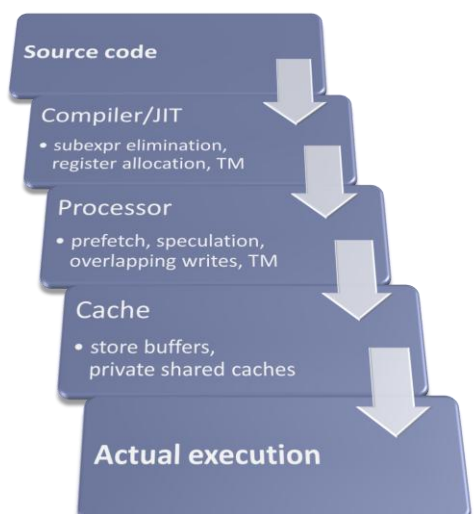
```
flag2 = 1;               // c: declare intent to enter
if( flag1 != 0 ) { ... } // d: detect and resolve contention
// enter critical section
```

- 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/VC++ "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.



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 the system 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.



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/VC++ "volatile", C++0x **std::atomic<>**) are automatically safe from reordering. Declare flag1 and flag2 appropriately (e.g., `atomic<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;
mb();                    // Linux full barrier (on x86, generates mfence)
if( flag2 != 0 ) { ... }
```

- Disadvantages:
  - Nonportable: Different flavors on different processors.
  - Tedious: Have to be written at every 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 only to compiler reordering or only 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( saMutex );
  s.a = 1;
}
```

  - Thread 2:

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

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

Object Layout Considerations (2)

- What about two global variables char c; and char d; ?
  - Thread 1:
    ```
    {
     Lock hold( cMutex );
     c = 1;
    }
    ```
  - Thread 2:
    ```
    {
     Lock hold( dMutex );
     d = 1;
    }
    ```
- Is there a race? **No ideally and in C++0x, but maybe today:**
  - Say we lay out c then d contiguously, and transform “d = 1;” to:
    ```
    char tmp[4];
    memcpy( &tmp[0], &c, 4 );
    // … in tmp, write to the bits corresponding to d …
    memcpy( &c, &tmp[0], 4 );
    ```

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.

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;          //   if we did, back it out... no harm no foul, right?
```

- 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.

Speculation (2)

- Let's add a little more context to show a real-world issue:

```
void f( /*...params...*/, bool doOptionalWork ) {
  if( doOptionalWork ) xMutex.lock();
  ...
  if( doOptionalWork ) x = 42;        // possibly far down the call tree
  ...
  if( doOptionalWork ) xMutex.unlock();
}
```

- Now we see the problem with this transformation:

```
r1 = x;
x = 42;
if( !doOptionalWork )
  x = r1;                             // oops: not holding the lock
```

- If so, it's not safe to have a conditional lock.
  - (Workaround coming up, hold on for a few slides…)

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 still 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;                              // write is still conditional
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? Dealing With Conditional Locks

- Problem:
  - Your code conditionally takes a lock, but your system changes a conditional write to be unconditional.
  - If so, it's not safe to have a conditional lock. The "SC for correctly locked code" illusion breaks because of a tool/hardware bug.
- Workaround:
  - 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.
  - 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**.

What Have We Learned? Dealing With Conditional Locks (2)

- In general, code may conditionally update a shared variable:
  - In a block that may not be executed:

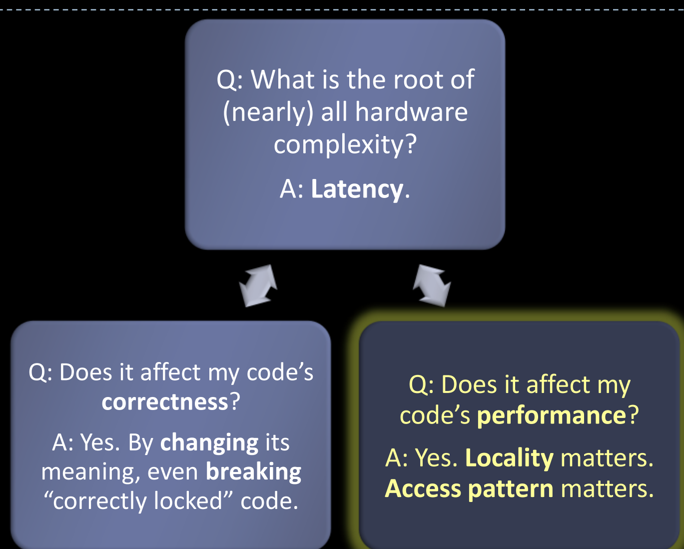
```
if( cond ) {        if( cond ) goto Label:
 …                  …
 ++x;               ++x;
 …                  …
}                   Label:
```

  - In a loop that may not be executed (incl. loops inside algorithms):

```
for( … ) {      while( … ) {        for_each( …, UpdateX() );
 …               …                  transform( …, UpdateX() );
 ++x;            ++x;               remove_if( …, BadPredUpdatesX() );
 …               …
}               }
```

- Be careful of system bugs if you want to use the “may not be executed” information to save taking a lock that you “know” you don’t need (where “know” = “because of SC reasoning”).

Machine Architecture and You



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?

Working Set Effects: Storing 1M ints

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

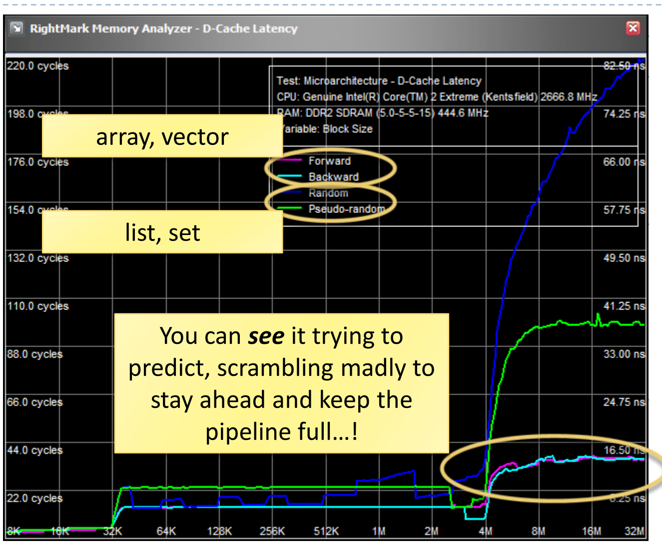
Working set arrangement	**Cache**		**Memory**	
	# lines	total	# pages	total
Perfectly contiguously (e.g., vector<int>)	65,536	4 MB	1,024	4 MB
Full cache lines, half-full pages (e.g., vector< array<int,512>*>)	65,536	4 MB	min 1,024 **max 2,048**	min 4 MB **max 8 MB**
5.33 ints per cache line (e.g., list<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.

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: It's the cache, silly! (usual answer, but this time with data)
  - Fewer linked list items fit in any given level of cache. All those object headers and links waste space.
  - But it's also the traversal: **Visitation order matters!**
- A2: Out-of-order processors really like arrays.
  - 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…

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



Observations

Access patterns matter:

- Linear = not bad.
- Random = awful.

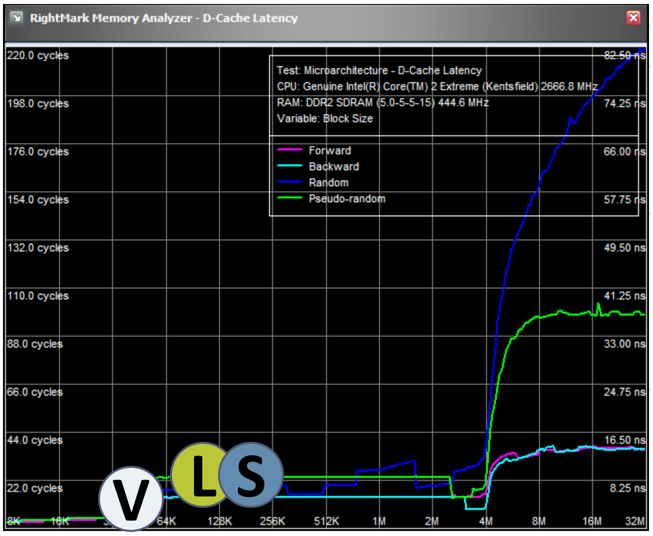
Vectors:

- 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$, 4M L2$



Observations

Access patterns matter:

- Linear = not bad.
- Random = awful.

Vectors:

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

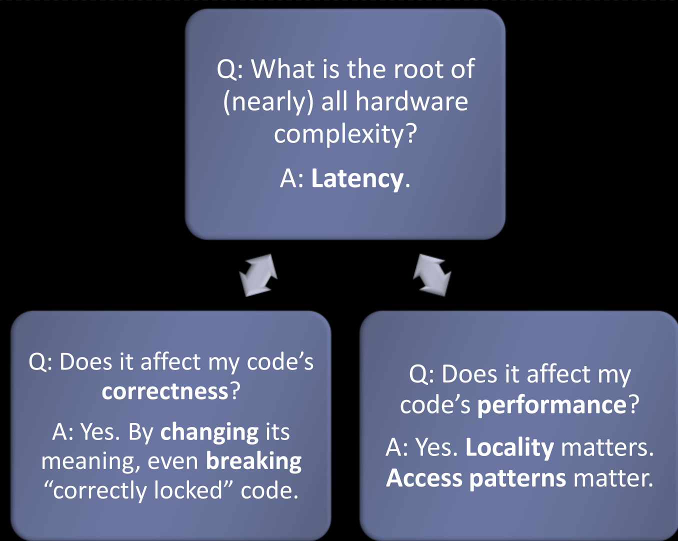
Lists and sets:

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

Cache-Conscious Design *(with thanks to Jan Gray)*

- **Locality is a first-order issue.**
- **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.
- **Prefer compact representations.** 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.
- **Consider usage patterns.** Some usage patterns favor hybrid structures—lists of small arrays, arrays of arrays, or B-trees.
- **Consider treating memory like disk.** Disk-access-sensitive scheduling algorithms were designed back when disk accesses cost only 50,000 CPU instructions. It may be time to recycle them now that DRAM accesses can take thousands of CPU operations.
- **Consider partitioning your data.** Separate "hot" parts that are frequently traversed and must fit in cache, and "cold" parts that are infrequently used and can be 'cached out.'

Machine Architecture and You



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*