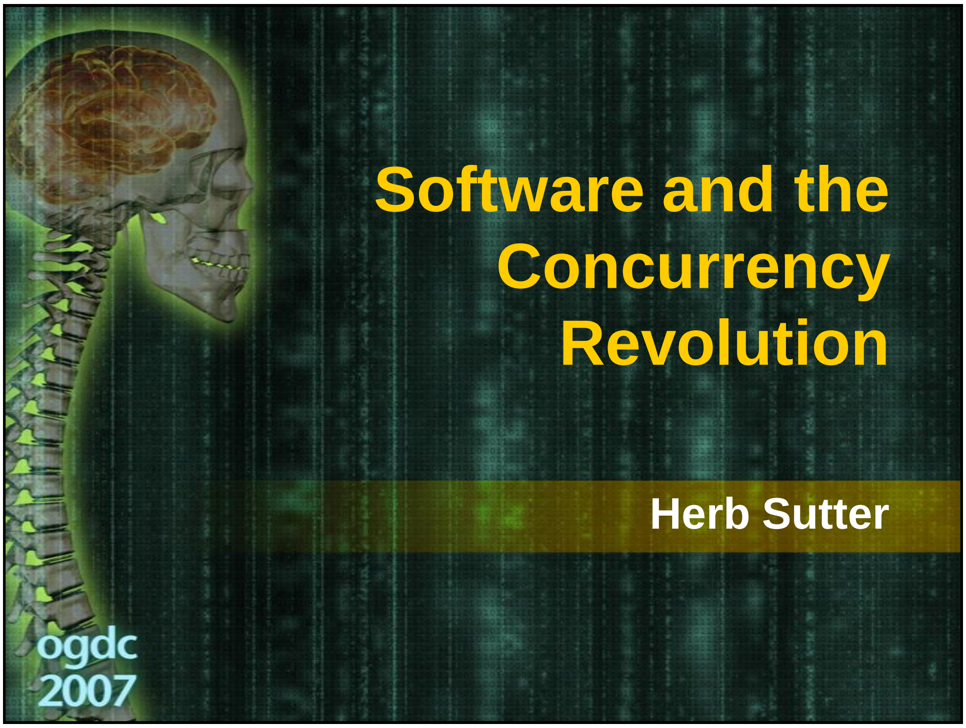
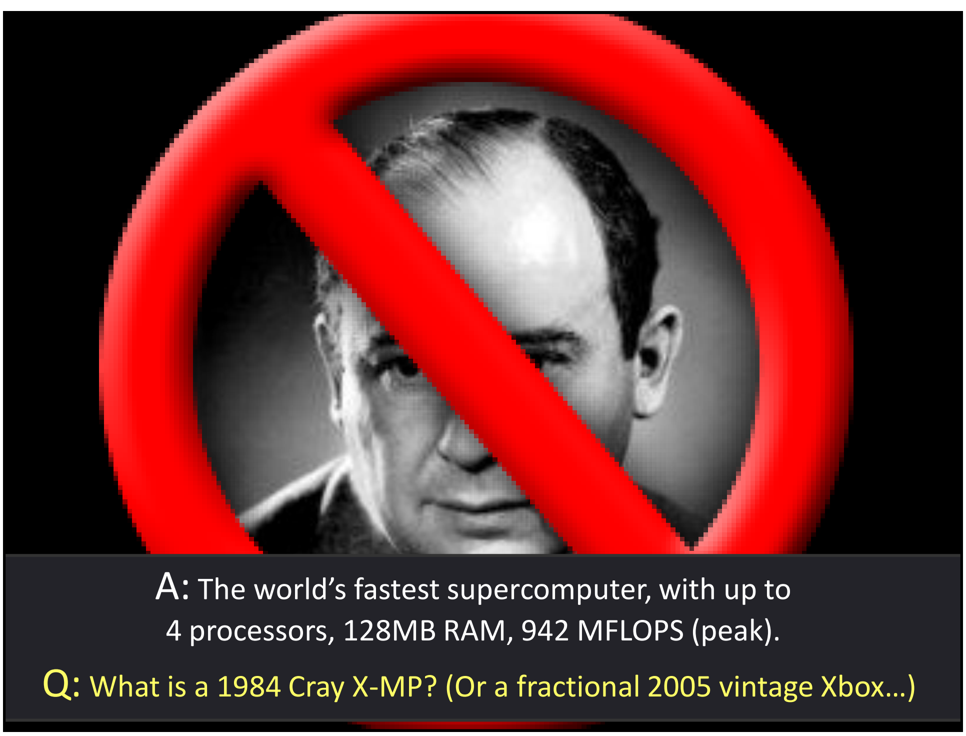
Herb Sutter
Software Architect, Microsoft
Software Development Consultant, *www.gotw.ca/training*





Truths

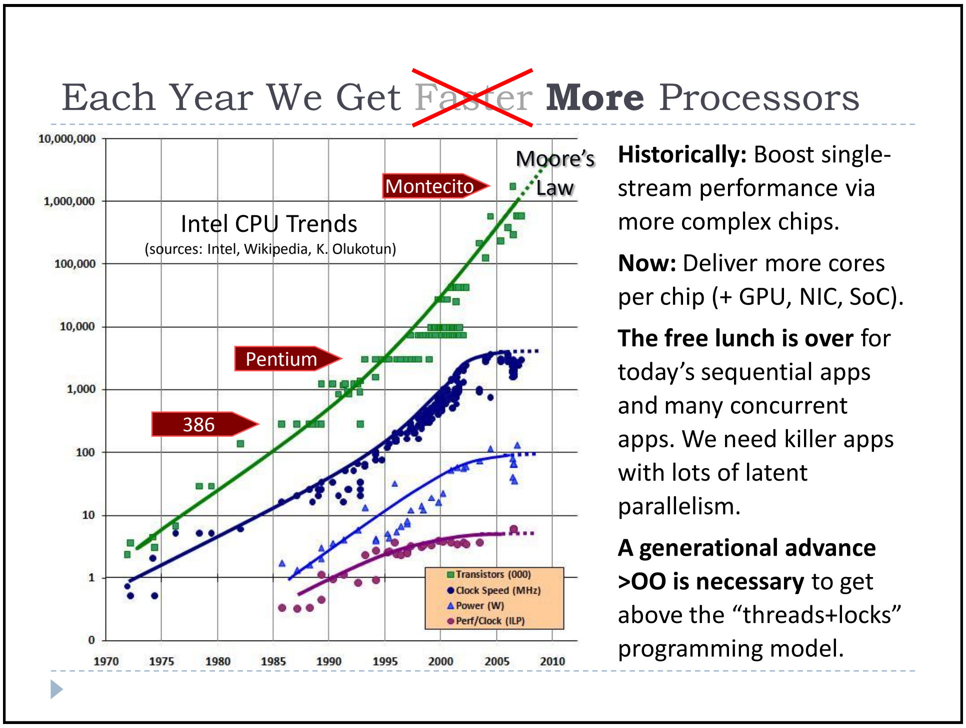
Consequences

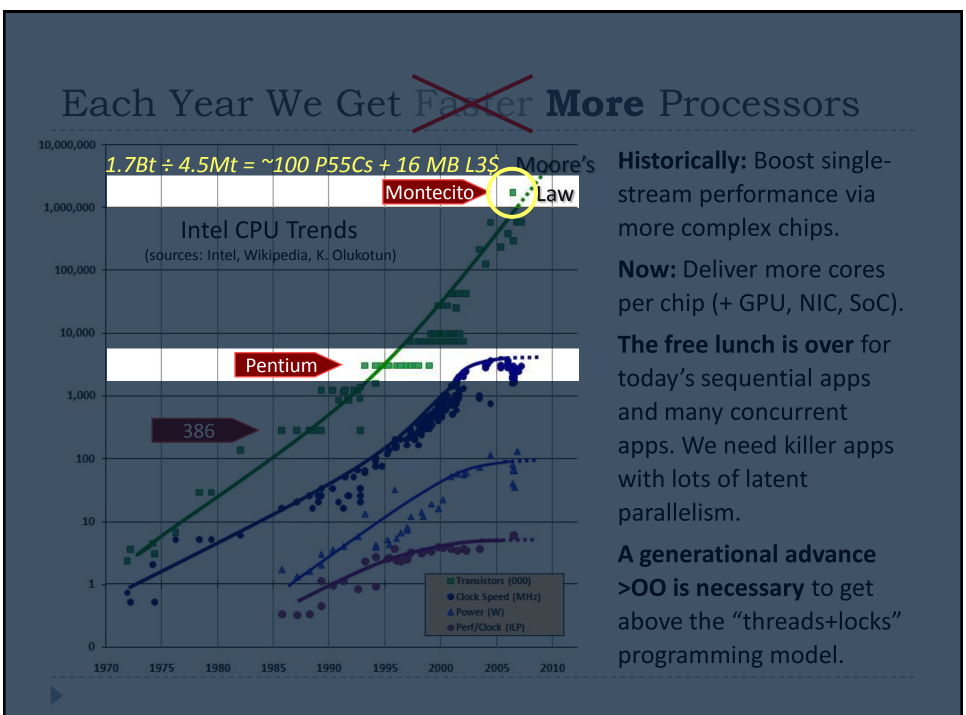
Futures

The Last Slide First

- Although driven by hardware changes, the concurrency revolution is **primarily a software revolution**.
- Parallel hardware is not "more of the same."
  - It's a fundamentally different mainstream hardware architecture, even if the instruction set looks the same. (But using the same ISA does give us a great compatibility/migration story.)
- Software requires the most changes to regain the "free lunch."
  - The concurrency sea change impacts the entire software stack: Tools, languages, libraries, runtimes, operating systems.
  - No programming language can ignore it and remain relevant.
  - (Side benefit: Responsiveness, the other reason to want async code.)
- Software is also the gating factor.
  - **We will now do for concurrency what we did for OO and GUIs.**
  - Beyond the gate, hardware concurrency is coming **more** and **sooner** than most people yet believe.

Herb Sutter
Software Architect, Microsoft
Software Development Consultant, *www.gotw.ca/training*





Educational State of the Union: May 2007

- We have achieved **general awareness**.
  - Most people know that "the free lunch is over" for sequential applications, and that the future is multicore and manycore.
- But few people really yet believe the **magnitude**, **speed**, and **gating factor** of the change:
  - **Magnitude:** Comparable to the GUI revolution plus moving to a new hardware platform, simultaneously. Enabling manycore affects the entire software stack, from tools to languages to frameworks/libraries to runtimes.
  - **Speed:** (Recall: Intel could build 100-Pentium chips today if they wanted to.) 100-way HW concurrency could be available in commodity desktops as soon as the year …
  - **Gating factor:** … manycore-exploiting software is available.

Truths

Consequences

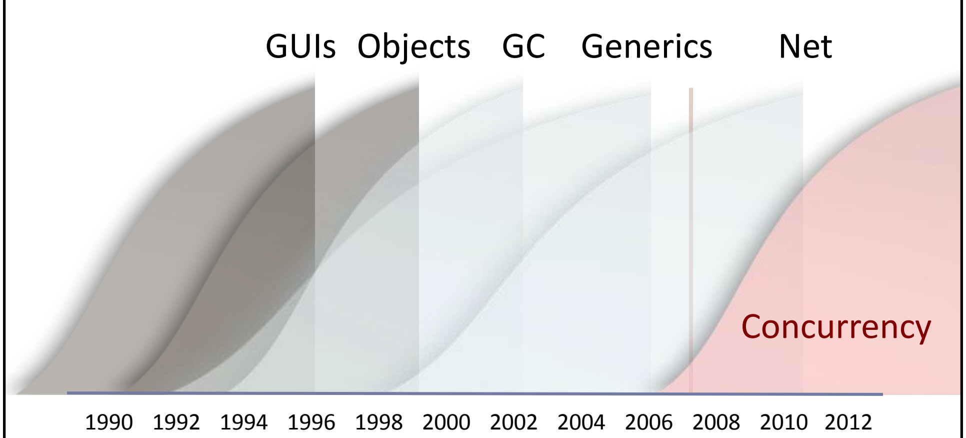
Futures

The Issue Is (Mostly) On the Client

- "Solved": Server apps (e.g., DB servers, web services).
  - Lots of **independent requests** – one thread per request is easy.
  - Typical to execute many copies of the same code.
  - Shared data usually via **structured databases**:
    Automatic implicit concurrency control via transactions.
  - With some care, "concurrency problem is already solved" here.
- Not solved: Typical client apps (i.e., not Photoshop).
  - Somehow employ many threads **per user "request."**
  - Highly atypical to execute many copies of the same code.
  - Shared data in **unstructured shared memory**:
    Error prone explicit locking – where are the transactions?

A Tale of Six Software Waves

- Each was born during 1958-73, bided its time until 1990s/00s, then took 5+ years to build a mature tool/language/framework/runtime ecosystem.



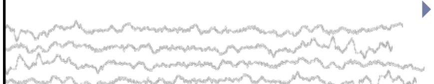
Comparative Impacts Across the Stack

	GUIs	Objects	GC	Generics	Web	Concurrency
Application programming model	●●●	●●●	●	●	●	●●●
Libraries and frameworks	●●●	●●●		●●	●●●	●●
Languages and compilers	●●	●●●	●	●●		●●●
Runtimes and OS	●●		●●		●	●●●
Tools (design, measure, test)	●●●	●	●		●●	●●

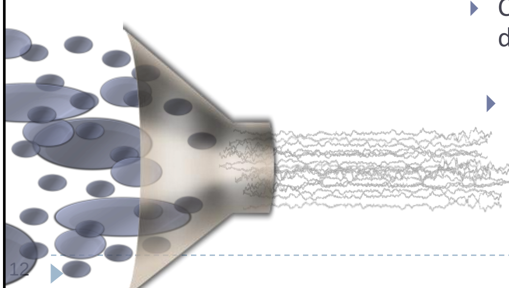
Extent of impact and/or enabling importance
● = Some, could drive one major product release
●● = Major, could drive more than one major product release
●●● = New way of thinking / essential enabler, or multiple major releases

O(1), O(K), or O(N) Concurrency?

- 1. Sequential apps.
  - The free lunch is over (if CPU-bound): Flat or merely incremental perf. improvements.
  - Potentially poor responsiveness.



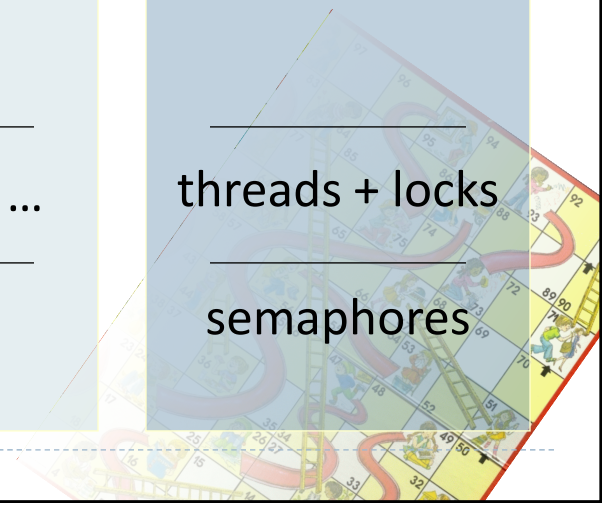
- 2. Explicitly threaded apps.
  - Hardwired # of threads that prefer K CPUs (for a given input workload).
  - Can penalize <K CPUs, doesn't scale >K CPUs.



- **3. Scalable concurrent apps.**
  - **Workload decomposed into a "sea" of heterogeneous chores.**
  - **Lots of latent concurrency we can map down to N cores.**

An OO for Concurrency

OO	
Fortran, C, …	threads + locks
asm	semaphores



Three Pillars of the Dawn

A Framework for Evaluating Concurrency

	Asynchronous Agents	Concurrent Collections	Mutable Shared State
Summary	Tasks that run independently and communicate via messages	Operations on groups of things; exploit parallelism in data and algorithm structures	Avoid races by synchronizing mutable objects in shared memory
Examples	GUIs, background printing, disk/net access	Trees, quicksort, compilation	Locked data (99%), lock-free libraries (written by wizards)
Key metrics	Responsiveness	Throughput, manycore scalability	Race-free, deadlock-free
Requirements	Isolation, messaging	Low overhead	Composability
Today's abstractions	Threads, message queues	Thread pools, OpenMP	Locks
Possible new abstractions	Active objects, futures	Chores, futures, parallel STL, PLINQ	Transactional memory, declarative support for locks

The Trouble With Locks

- How may I harm thee? Let me count the ways:

```
{
  Lock lock1( mutTable1 );
  Lock lock2( mutTable2 );
  table1.erase( x );
  table2.insert( x );
}
```

- Locks are the best we have, and known to be inadequate:
  - **Which lock?** The connection between a lock and the data it protects exists primarily in the mind of a programmer.
  - **Deadlock?** If the mutexes aren't acquired in the same order.
  - **Simplicity or scalability?** Coarse-grained locks are simpler to use correctly, but easily become bottlenecks.
  - **Lost wake-ups?** Blocking typically uses condition variables, and it's easy to forget to signal the "correct" ones. *(Example coming up.)*
  - **Not composable.** In today's world, this is a deadly sin.

Enter Transactional Memory

- A transactional programming model:

```
atomic {
  table1.erase( x );
  table2.insert( x );
}
```

  - Idea: Version memory 'like a database.' Automatic concurrency control, rollback and retry for competing transactions.
- Benefits:
  - No need to remember which lock to take.
  - No deadlock: No need to remember a sync order discipline.
  - Both fine-grained and scalable without sacrificing correctness.
  - No wake-up calls as atomic blocks are automatically re-run.
  - **Best of all: Composable.** Can once again build a big (correct) program out of smaller (correct) programs.
- Drawbacks:
  - Still active research. And the elephant in the room is I/O (more later).

Enter Transactional Memory (2)

- What if we want to block if x isn't in table1 yet?
  - Today, we'd typically use a wake-up: Wait on a condition variable, and have every thread inserting into table1 remember to signal the right cv. Tedious, error-prone, etc.
- Instead:

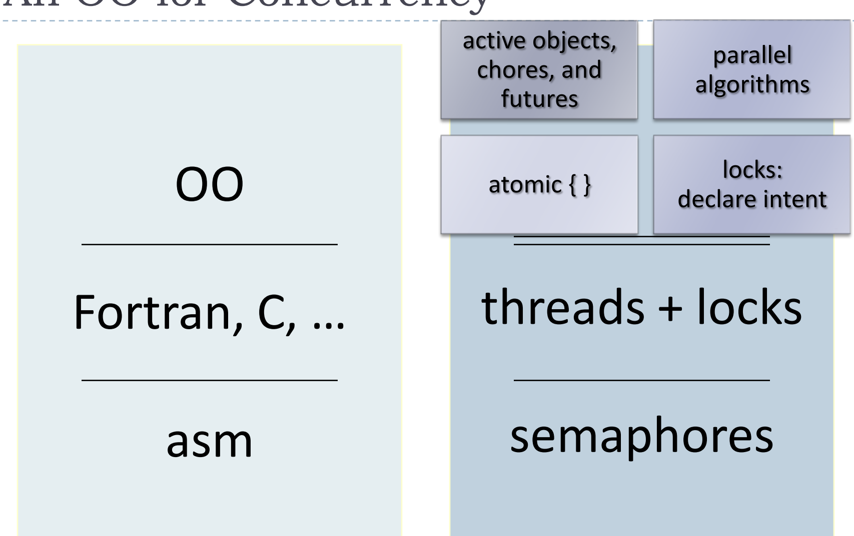
```
atomic {
  if( table1.find(x) == table1.end() )
    retry;
  table1.erase( x );
  table2.insert( x );
}
```

  - **retry** restarts the current atomic block from the beginning.
  - Blocking is entirely modular: The call to retry might be deeply buried inside the implementation of debit, or of credit, or both.
  - Can use transaction log to defer re-running until at least one memory location read by the transaction has changed.

A “4-Step Program” For the Lock Addiction

- Greatly reduce locks. (Alas, not “eliminate.”)
  1. Enable transactional programming: Transactional memory is our best hope. **Composable** atomic { … } blocks. Naturally enables speculative execution. (The elephant: Allowing I/O. The Achilles’ heel: Some resources are not transactable.)
  2. Abstractions to reduce “shared”:
     Messages. Futures. Private data (e.g., active objects).
  3. Techniques to reduce “mutable”:
     Immutable objects. Internally versioned objects.
  4. Some locks will remain. Let the programmer declare:
     - Which shared objects are protected by which locks.
     - Lock hierarchies (caveat: also not composable).

An OO for Concurrency

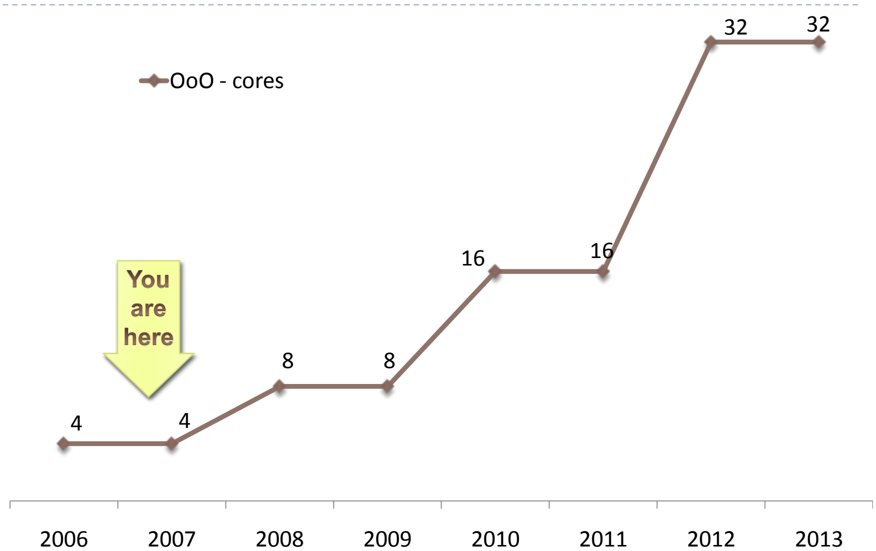


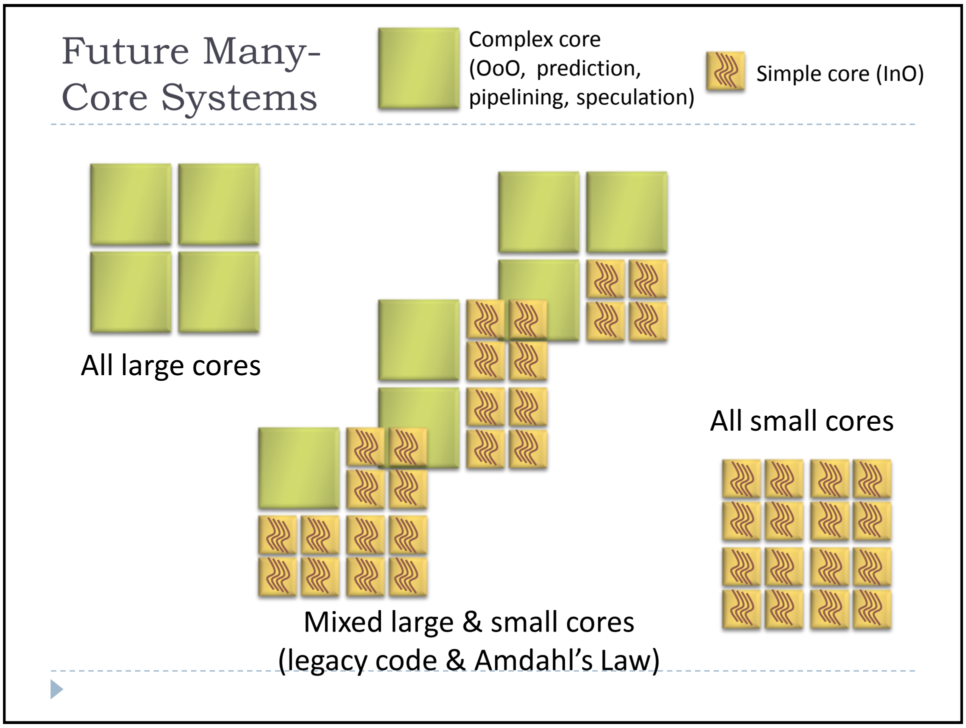
Truths

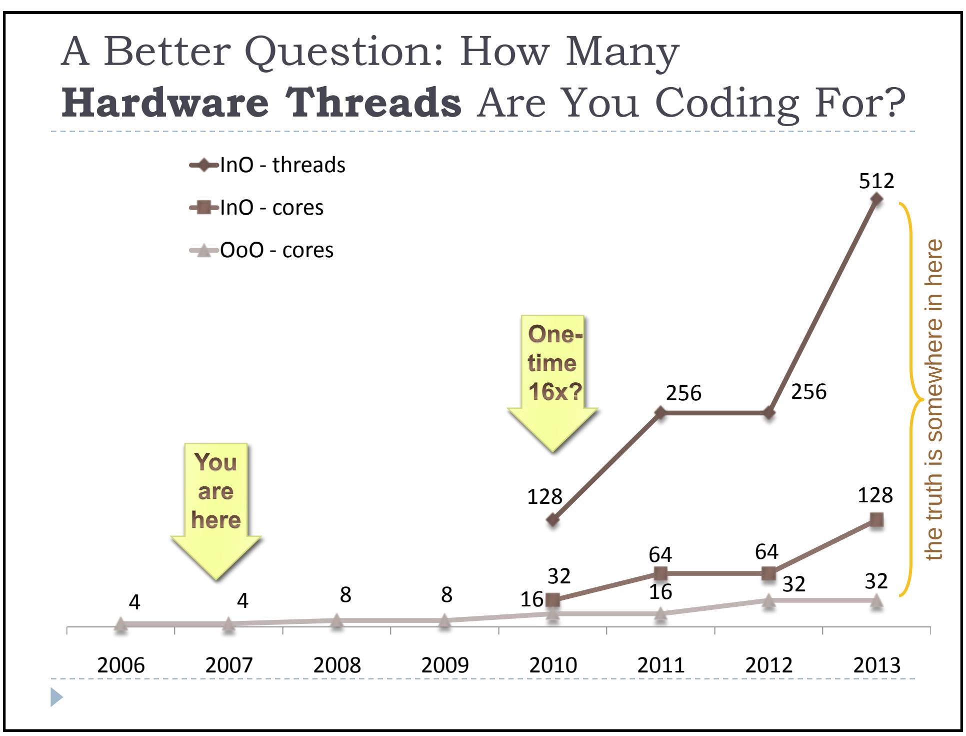
Consequences

Futures

A Good Question: How Many Cores Are You Coding For?




Future Many-Core Systems
Complex core (OoO, prediction, pipelining, speculation)
Simple core (InO)
All large cores
All small cores
Mixed large & small cores (legacy code & Amdahl's Law)


A Better Question: How Many **Hardware Threads** Are You Coding For?
InO - threads
InO - cores
OoO - cores
You are here
One-time 16x?
the truth is somewhere in here
512
256
256
128
128
64
64
32
32
32
16
16
8
8
4
4
2006
2007
2008
2009
2010
2011
2012
2013

The First Slide Last

- Although driven by hardware changes, the concurrency revolution is **primarily a software revolution**.
- Parallel hardware is not "more of the same."
  - It's a fundamentally different mainstream hardware architecture, even if the instruction set looks the same. (But using the same ISA does give us a great compatibility/migration story.)
- Software requires the most changes to regain the "free lunch."
  - The concurrency sea change impacts the entire software stack: Tools, languages, libraries, runtimes, operating systems.
  - No programming language can ignore it and remain relevant.
  - (Side benefit: Responsiveness, the other reason to want async code.)
- Software is also the gating factor.
  - **We will now do for concurrency what we did for OO and GUIs.**
  - Beyond the gate, hardware concurrency is coming **more** and **sooner** than most people yet believe.

For More Information

- "The Free Lunch Is Over"
  (*Dr. Dobb's Journal*, March 2005)
  http://www.gotw.ca/publications/concurrency-ddj.htm
  - The article that first used the terms "the free lunch is over" and "concurrency revolution" to describe the sea change.
- "Software and the Concurrency Revolution"
  (with Jim Larus; *ACM Queue*, September 2005)
  http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=332
  - Why locks, functional languages, and other silver bullets aren't the answer, and observations on what we need for a great leap forward in languages and also in tools.