Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes 1

28Feb06

Moore's Law

James Goodman

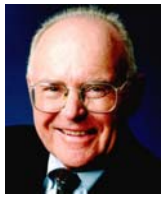
Department of Computer Science

The University of Auckland | New Zealand

2006 YEAR | PRESENTATION

Dr. Gordon Moore

1965



2003

Moore's Data: 1965

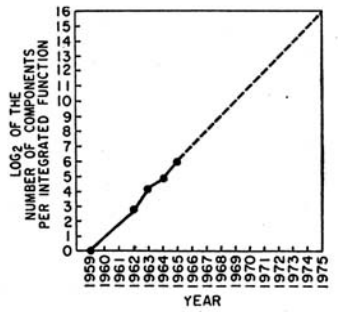


Fig. 3.

Moore's Observation

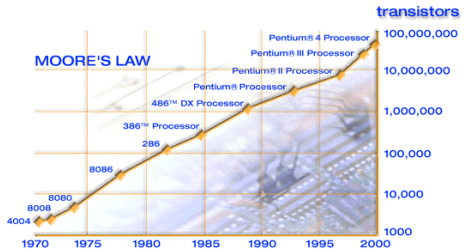
"The complexity for minimum component costs has increased at a rate of roughly ***a factor of two per year****.... Certainly over the short term* ***this rate can be expected to continue, if not to increase.*** *Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65 000."*

— Gordon E. Moore
"Cramming more components onto integrated circuits,"
Electronics, pp. 114-117, Apr. 1965.

Moore's Prediction

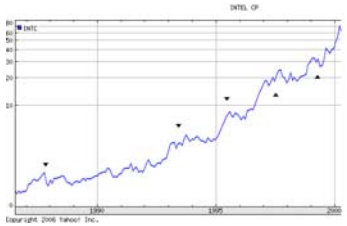
Fig. 2.

Moore's Company



http://www.intel.com/research/silicon/mooreslaw.htm

Moore's Motivation



Moore's Formula

Minimum cost # of components/chip

$$N = 2^{(year - 1959)}$$

Extrapolating to 2006,

$$N = 2^{(2006 - 1959)} = 141 * 10^{12}$$

$$= 141 \text{ Trillion transistors}$$

Moore's Correction: 1975

There is no room left to squeeze anything out by being clever. Going forward from here we have to depend on the two size factors – bigger dice and finer dimensions.

— Gordon E. Moore
Electronic Devices Meeting, 1975.

Moore's Corrected Formula

Minimum cost # of components

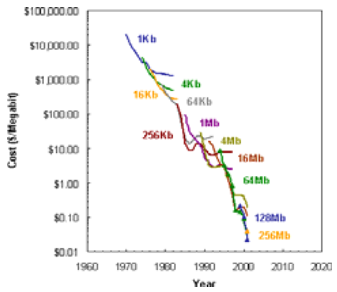
$$N = 2^{(year - 1959)/1.5} = 1.59^{(year - 1959)}$$

Extrapolating to 2006,

$$N = 1.59^{(2006 - 1959)} = 2.9 * 10^{9}$$

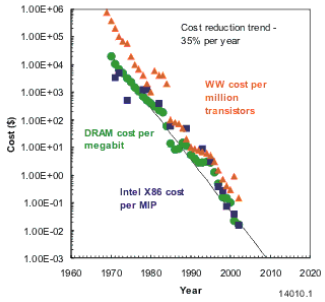
= 2.9 *billion transistors*

Drop in DRAM Cost per Bit



http://www.icknowledge.com/economics/dramcosts.html
IC Knowledge, 2001

Other Measures of Cost Reduction



http://www.icknowledge.com/economics/productscostscosts2.html
IC Knowledge, 2001

Total Transistor Production

- Reduction in cost: 35%/year
- Increase in sales volume: 15%/year
- Increase in transistor production:

1.15/.65 = 77%/year

My estimate for 2006: ~620,000,000,000,000,000,000
i.e., 620 quintillion transistors!

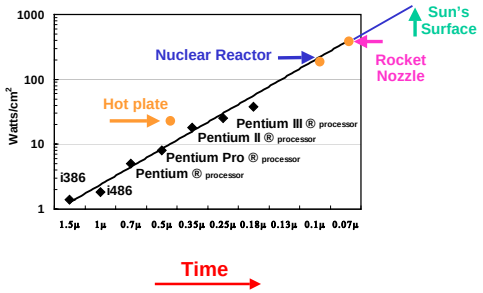
Transistors ≠ Performance

- Limited gain in performance derives directly from semiconductor gains
- The rest comes from better architecture

Performance Gains from Physics

Smaller transistors are closer together
- switch faster
- communicate faster
- require less energy

A Different Exponential Law



Fred Pollack, Intel Corp. 2000

Joy's Law

"PERFORMANCE of a microprocessors doubles every three years."

—William Joy, 1980

Also known as "Popular Moore's Law"

Joy's Law

Relative Performance of Microprocessor

$$P = 2^{(year - 1980)/3} = 1.26^{(year - 1980)}$$

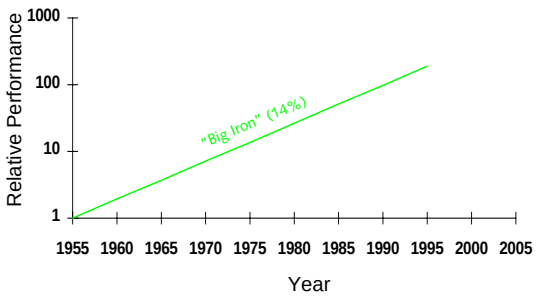
Extrapolating to 2006 relative to 1980,

$$P = 1.26^{(2006 - 1980)} = 407X$$

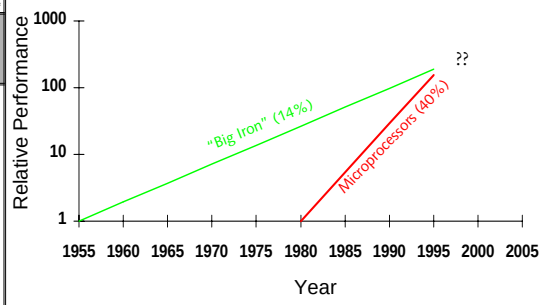
Realistic rate has been closer to 40%/year:

$$P = 1.40^{(2006 - 1980)} = 6300X$$

Best Uniprocessor Performance 1955-1995



Best Uniprocessor Performance 1955-1995



Microprocessor Improvements

- Microprocessors are a good match for Moore's Law: single-chip processors
- Previous technology created a "bag of tricks" to be exploited

Architectural Advances 1950-1990

- Branch prediction: 1995 (1959)
- Out-of-order issue: 1993 (1963)
- Multi-threading: 1995 (1963)
- Cache memories: 1985 (1965)
- Superscalar Processing (mult instrs/cycle): ~1990 (1960s)
- Register renaming: ~1992 (1967)
- Deep pipelining: ~1993 (1976)
- Speculative execution: ~1995 (1983)

When Does It End?

"We're half way down the learning curve [after 11 years]"
— Professor Carlo Sequin, UC-Berkeley, 1976.

"It can't go on much longer. We're pushing against some really fundamental limits!"
— Dr. Joel Emer, DEC, 1996.

The End is Not in Sight

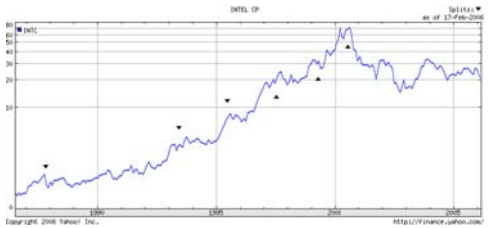
The Roadmap continues to call for reduction [until 2012] in geometric dimensions in accordance with Moore's Law, but allows for short-term adjustments based on current practices.

— Semiconductor Industry Association: The (US) National Technology Roadmap for Semiconductors, 1997.

The End is Not in Sight

International Technology Roadmap for Semiconductors, 2005 Edition (Executive Summary). Figure 10, p. 68.

Moore's Motivation



News [Processors]

Friday 15th October 2004

Intel abandons clock speed chase and drops 4GHz Pentium 9:31AM

In another embarrassing admission in a year scattered with them, Intel has announced that it is to cease development of the 4GHz Pentium processor. For now the 3.8GHz part will remain the highest clock speed. Instead, the company says that it will be transferring its engineers to work on the dual core designs demonstrated at the recent Developer's Forum.

The unexpected announcement marks the end of an era for Intel. Ever since the launch of the 4.77MHz 8088 around which the original IBM PC was designed, the company has largely depended on ever increasing clock speeds to boost performance. However, recently Intel and arch-rival AMD have been having greater problems in producing reliable parts that could cope with the amount of heat generated by these clock speeds.

Source: http://www.comp-buyer.co.uk/buyer/processors/news/64478/intel-abandons-clock-speed-chase-and-drops-4ghz-pentium.html

Today's Hot Technology

- Hyperthreading
- Multithreading
- Multicore

That is, Multiprocessing

Do I belong in this course?

- You should have taken CS313, SE363, or equivalent (Patterson/Hennessy book)
  - Processor design (pipelining)
  - Memory systems
    - memory hierarchies
    - virtual memory (TLBs)
- You should have learned about operating systems
  - Virtual memory
  - Critical sections
  - Process model

Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes 2

1Mar06

Multiprocessing & Multithreading

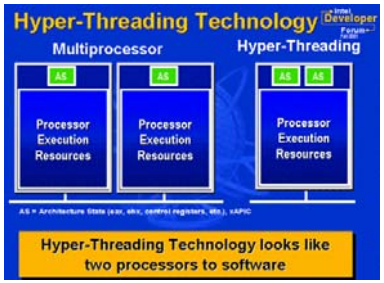
James Goodman

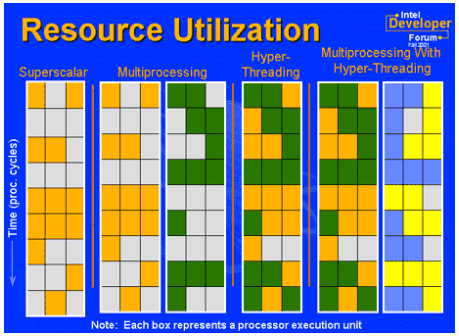
Department of Computer Science

Multiprocessors, Multi-Cores, Multi-threading, and Hyperthreading

Terminology

- Multiprocessors: multiple processors sharing a common memory (SMP, tightly-coupled MP)
- Multi-cores: multiple processors sharing a common silicon die and memory system (CMP)
- Multithreading: a single processor capable of maintaining the state of multiple threads or processes while executing
- Hyperthreading: Intel's term for a certain type of multithreading (SMT)
- Chip Multithreading (CMT): a multi-core die with multithreaded processors





Lectures this week

- Wednesday: Multithreading/Hyperthreading
- Thursday: Multiprocessing
- Friday: MP Programming and Hardware Support

Lectures Next Week

- Tuesday: P-threads
- Wednesday: Interconnection Networks
- Thursday: Interconnection Networks

Why Multi-threading

- Resources can be used more effectively
  - Up to 30% more throughput from two threads (Intel)
  - About 5% additional die area for second thread
- Threads can actively share memory data very efficiently

Memory System

- Threads on same core share all memory except registers
- Multi-cores often share L2 cache, not L1

Variations of Multithreading

- Fine-grained multithreading switches between threads on each instruction, causing the execution of multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time.
- Coarse-grained multithreading switches threads only on costly stalls, such as level-2 cache misses.
- Simultaneous multithreading (SMT) is a variation on multithreading that uses the resources of a multiple-issue, dynamically scheduled processor to exploit thread-level parallelism (TLP) at the same time it exploits instruction-level parallelism (ILP)

Hennessey &Patterson, pp. 608-609.

Comparison of Multi-threading and True Multiprocessing

- Multi-threading is limited to exploiting wasted resources
- Multi-threading can have faster communication through memory
- Multi-threading can share code in L1 cache (but may also require more cache if code is not shared)

Interesting Multithreading Trade-off

- Multiple threads implies greater tolerance for cache misses

but…

- Multiple threads implies multiple contexts
- Multiple contexts implies larger memory requirements

Multithreading makes sense if throughput is important!

Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes 3
2Mar06
Multiprocessing Issues

James Goodman

Department of Computer Science

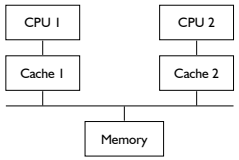
Multiprocessing Issues

- Cache Coherence
- Memory Consistency
- "Missing Update Problem"

Multiprocessing Issues

- **Cache Coherence**
- Memory Consistency
- "Missing Update Problem"

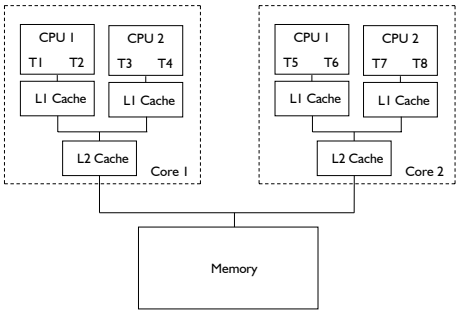
Basic MESI Protocol



Cache blocks may be in one of four possible states

- **M***odified*: copy in cache is different than copy in main memory, which is stale
- **E***xclusive*: copy in cache can be modified without external permission, but is the same as main memory.
- **S***hared*: copy in cache is valid for reading, but may not be modified without eliminating other potential copies
- **I***nvalid*: data in cache is stale

Multilevel Cache Coherence



Summary of Snooping Caches

- Snooping cache coherence protocols are the *dominant multiprocessor technique* used today
- Most microprocessors conform to snooping cache protocols (e.g., Intel Pentium: up to 4 processors on the bus)
- Snooping has been extended to much larger systems by a series of creative methods, but scalability is fundamentally limited by broadcast requirements

Coherency in Multiple-Bus Systems

- Scalable protocols involve maintaining a directory auxiliary information keeping track of which caches have copies of which cache lines
- Directory-based scheme can use point-to-point connections, which potentially have both higher speed and much higher bandwidth than a bus
- Because of the additional delay (typically three hops for most transactions), only very large systems benefit from directory-based schemes.
- There have been several commercial products, but so far, no directory-based scheme has been highly successful.

Terminology

- Snooping-based schemes have been extended beyond a single bus (maintaining the notion of a single "logical" bus). Broadcast-based schemes are called *Symmetric MultiProcessing* (SMP)
- Directory-based schemes continue to be widely studied, and there are many variations proposed and some products. Such schemes are often referred to as *Non-Uniform Memory Access* (NUMA) systems.

Multiprocessing Issues

- Cache Coherence
- **Memory Consistency**
- "Missing Update Problem"

Memory Ordering

Initial state: `A = 0` `B = 0`

CPU 1	CPU 2
`Write A = 1`	`Read B = 1`
`Write B = 1`	`Read A = 0`

Is this possible?

Is it *acceptable?*

Sequential Consistency

Definition: "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

Memory Ordering Requirement (1)

Initial state: `A = 0` `B = 0`

CPU 1	CPU 2
`Write A = 1`	`Read B = 1`
`Write B = 1`	`Read A = 0`

SC: No
Intel: No
Alpha: Yes

Is this permitted?

Alpha Memory Ordering Requirements

- Reads and writes may appear out of order
- A *memory barrier* assures that all previous operations have been made globally visible before any subsequent operations are made visible

Memory Ordering Requirement (1)

Initial state: `A = 0` `B = 0`

CPU 1	CPU 2
`Write A = 1`	`Read B = 1`
`MemBar`	`MemBar`
`Write B = 1`	`Read A = 0`

SC: No
Intel: No
Alpha: *No*

Is this permitted?

Memory Ordering Requirement (2)

Initial state: `A = 0`
`B = 0`

CPU 1	CPU 2
`Write A = 1`	`Write B = 1`
`Read B = 0`	`Read A = 0`

SC: No
Intel: *Yes*
Alpha: Yes

Is this permitted?

Yet Another Memory Model

- Release Consistency
  - Assumes that locks are used to protect shared data
  - No reads may be performed before acquiring the lock
  - All writes must be completed before releasing the lock

Summary of Memory Ordering

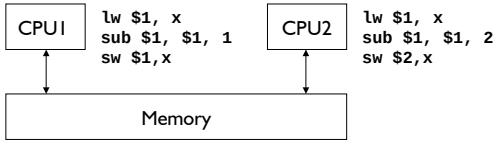
- Identical code sequences may result in different *acceptable* answers on multiprocessors with different memory models
- Compilers must account for memory model
  - Recognize potential data races
  - Insert barriers if necessary to assure correctness

Multiprocessing Issues

- Cache Coherence
- Memory Consistency
- "Missing Update Problem"

The "Missing Update" Problem

- Assume there is a shared `int x = 3;`
- CPU1 executes a program fragment `x = X - 1;`
- CPU2 executes a program fragment `x = X - 2;`
- What is the final value of the shared variable `x`?



Possible Answer: 0

CPU 1	CPU 2
`lw $1, x`	
`sub $1, $1, 1`	
`sw $1,x`	
	`lw $1, x`
	`sub $1, $1, 2`
	`sw $1,x`

Result: x = 0

Possible Answer: 0

CPU 1	CPU 2
	`lw $1, x`
	`sub $1, $1, 2`
	`sw $1,x`
`lw $1, x`	
`sub $1, $1, 1`	
`sw $1,x`	

Result: x = 0

Possible Answer: 1

CPU 1	CPU 2
`lw $1, x`	
`sub $1, $1, 1`	
`sw $1,x`	
	`lw $1, x`
	`sub $1, $1, 2`
	`sw $1,x`

Result: x = 1

Is this acceptable?

Possible Answer: 2

CPU 1	CPU 2
	`lw $1, x`
	`sub $1, $1, 2`
	`sw $1,x`
`lw $1, x`	
`sub $1, $1, 1`	
`sw $1,x`	

Result: x = 2

Is this acceptable?

Expectation of Atomicity & **Isolation**

- In the example, we expect that the code
  `x = x - 1`
  will be executed *atomically* and in *isolation*

Isolation: the appearance that a sequence of operations occur at a single instant in time.

Atomicity: the requirement that the sequence of operations either occurs in its entirety or not at all.

Parallel "Correctness"

- Our programs must execute "correctly" no matter how the two sequences of instructions are interleaved
- But correctness must be defined. The example introduces a *data race*
- If only a result of zero is acceptable, the code must explicitly eliminate data races
- Data races can be eliminated by the use of *locks (semaphores)* and *critical sections*

Observation: This problem has nothing to do with cache consistency or coherence!

Programming a Multiprocessor

- Multiprocessors may simply execute independent tasks that require more computing power that is available on a single processor.
  - Particularly useful if one or more jobs is computationally intensive
  - Often a maximum of two processors can handle all the jobs
- Major challenge: divide up a single job into pieces that can be computed concurrently.
- Two general models of parallel computation
  - The *epoch* model
  - The *work queue* model

The Epoch Model

- The program involves similar operations on large amounts of data (large, regular data structures)
- The data is partitioned into non-overlapping parts and assigned to various threads
- A fixed amount of computation is performed independently, then coalesced through synchronization
  - All nodes run the same code, over a different range of data
  - This is an *epoch*
- This process is repeated
  - A *barrier* assures that none of the threads proceed beyond the synchronization point until all have arrived at it
- Within an epoch, *usually* no races are allowed, i.e., no variable can be written by some node and read by another.

Synchronization Mechanism for Epochs

The barrier: wait for all nodes to arrive here before continuing:

Initially, **`Count`** = # of threads

```
barrier() {
  Count -= 1;
  while (Count > 0)
   ;
}
```

Note: decrementing **`Count`** *on multiple nodes introduces a race condition!*

The Work Queue Model

- The system is initialized by identifying a set of tasks to be performed. These are placed on a queue with information identifying the task and its parameters.
- Processors remove an assignment from the queue and perform the task. In the process, they may identify new tasks to be performed and place them on the queue.
- This process continues until all the tasks have been completed.
- The challenge is to divide tasks up fine enough so that all the threads can be kept busy, but course enough so that the threads don't spend all their time dealing with the work queue

Computer Science 703

Advance Computer Architecture

2006 Semester 1

Lecture Notes 3

2Mar06

Multiprocessor Programming & Hardware Support

James Goodman

Department of Computer Science

Conventional Wisdom

- Writing a correct parallel program is not hard
- Writing a fast parallel program is not hard
- *Writing a fast, correct parallel program is hard*

Why is Parallel Programming Hard?

- Cache Coherence
- Memory Consistency
- "Missing Update Problem"

Airline Reservation Problem

- Want to travel AKL->LHR
- Fastest way:
  - AKL/BNE on Air New Zealand 131
  - BNE/HKG on Qantas 97
  - HKG/LHR on British Airways 26
- How to book?

Database ACID Properties

Atomicity refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are. The transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account won't be debited if the other is not credited as well.

Consistency refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break the rules, or *integrity constraints*, of the database. If an integrity constraint states that all accounts must have a positive balance, then any transaction violating this rule will be aborted.

Isolation refers to the ability of the application to make operations in a transaction appear isolated from all other operations. This means that no operation outside the transaction can ever see the data in an intermediate state; a bank manager can see the transferred funds on one account or the other, but never on both—even if she ran her query while the transfer was still being processed. More formally, isolation means the transaction history is *serializable*. **For performance reasons, this ability is the most often relaxed constraint.**

Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Typically, all transactions are written into a log that can be played back to recreate the system to its state right before the failure. A transaction can only be deemed committed after it is safely in the log.

http://en.wikipedia.org/wiki/ACID --1Mar06

Serializability

Serializability ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order.

Events in a Transaction

A Transaction is a set of changes to the state of a database (and possible external effects, such as I/O)

- Writes to the database are events that change its state.
- Reads from the database are observations that events have previously occurred.

Note similarity of serializability to Sequential Consistency

Overlapping of Transactions

- Two transactions can be executed concurrently as long as the events occurring in one transaction are not observed to have occurred before any of the events in the other are observed.
- No event will be observed unless another transaction accesses *the same* memory location, either for reading or writing. This is is called a *conflict*.

Goal: Transactional Memory

```
atomic {
  mumble;
} catch (AbortedException e) {
}
```

- Can software provide this model?
  - Yes, but…
  - It's not easy (i.e., it's slow)
- Can hardware assist in providing this model?
  - Yes, but…
  - Not if the transaction is too big