CS 61C: Great Ideas in Computer Architecture

Performance Programming, Technology Trends

Instructor: Justin Hsia

Review of Last Lecture

- Multilevel caches reduce *miss penalty*
  - Standard to have 2-3 cache levels (and split I$/D$)
  - Makes CPI/AMAT calculations more complicated
- Cache design choices change performance parameters and cost

Question: (based on previous midterm question)

Which of the following cache changes will ***definitely*** increase L1 Hit Time?

(A) Adding unified L2$, which is larger than L1 but smaller than memory

(B) Increasing block size while keeping cache size constant

(C) Increasing associativity while keeping cache size constant

(D) Switching our replacement policy from LRU to Random

Agenda

- Performance Programming
- Administrivia
- Perf Prog: Matrix Multiply
- Technology Trends
  - The Need for Parallelism

Performance Programming

- Adjust memory accesses in *code* (software) to improve miss rate
- With understanding of how caches work, can revise programs to improve cache utilization

Performance of Loops and Arrays

- Array performance often limited by memory speed
- **Goal:** Increase performance by minimizing traffic from cache to memory
  - Reduce your miss rate by getting better reuse of data already in the cache
  - It is okay to access memory in different orderings as long as you still end up with the correct result
- *Cache Blocking:* "shrink" the problem by performing multiple iterations on smaller chunks that "fit well" in your cache
  - Use Matrix Multiply as an example (Lab 6 and Project 2)

Ex: Looping Performance (1/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- What does the increment operation look like in assembly?

```
# A → $s0
lw    $t0,0($s0)
addiu $t0,$t0,1
sw    $t0,0($s0)  ← Guaranteed hit!
addiu $s0,$s0,4
```

Ex: Looping Performance (2/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- What is will our miss rate be for a D$ with 1-word blocks? (array not in $ at start)
  - 50% MR because each array element (word) accessed just once
- Can code choices change this?
  - No

Ex: Looping Performance (3/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- Now for a D$ with 2-word blocks, what are the best and worst miss rates we can achieve?
  - Best: 75% MR via standard incrementation (each block will miss then hit, hit, hit)
  - Code:

```
for(int i=0; i<1024; i++)   A[i]++;
```

Ex: Looping Performance (4/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- Now for a D$ with 2-word blocks, what are the best and worst miss rates we can achieve?
  - Worst: 50% MR by skipping elements (assuming D$ smaller than half of array size)
  - Code:

```
for(int i=0; i<1024; i+=2)  A[i]++;
for(int i=1; i<1024; i+=2)  A[i]++;
```

Ex: Looping Performance (5/5)

- We have an array `int A[1024]` that we want to increment (i.e. `A[i]++`)
- For an I$ with 1-word blocks, what happens if we don't use labels/loops?
  - 100% MR, as all instructions are explicitly written out sequentially
- What if we loop by incrementing `i` by 1?
  - Will miss on first pass over code, but should be found in I$ for all subsequent passes

Agenda

- Performance Programming
- Administrivia
- Perf Prog: Matrix Multiply
- Technology Trends
  - The Need for Parallelism

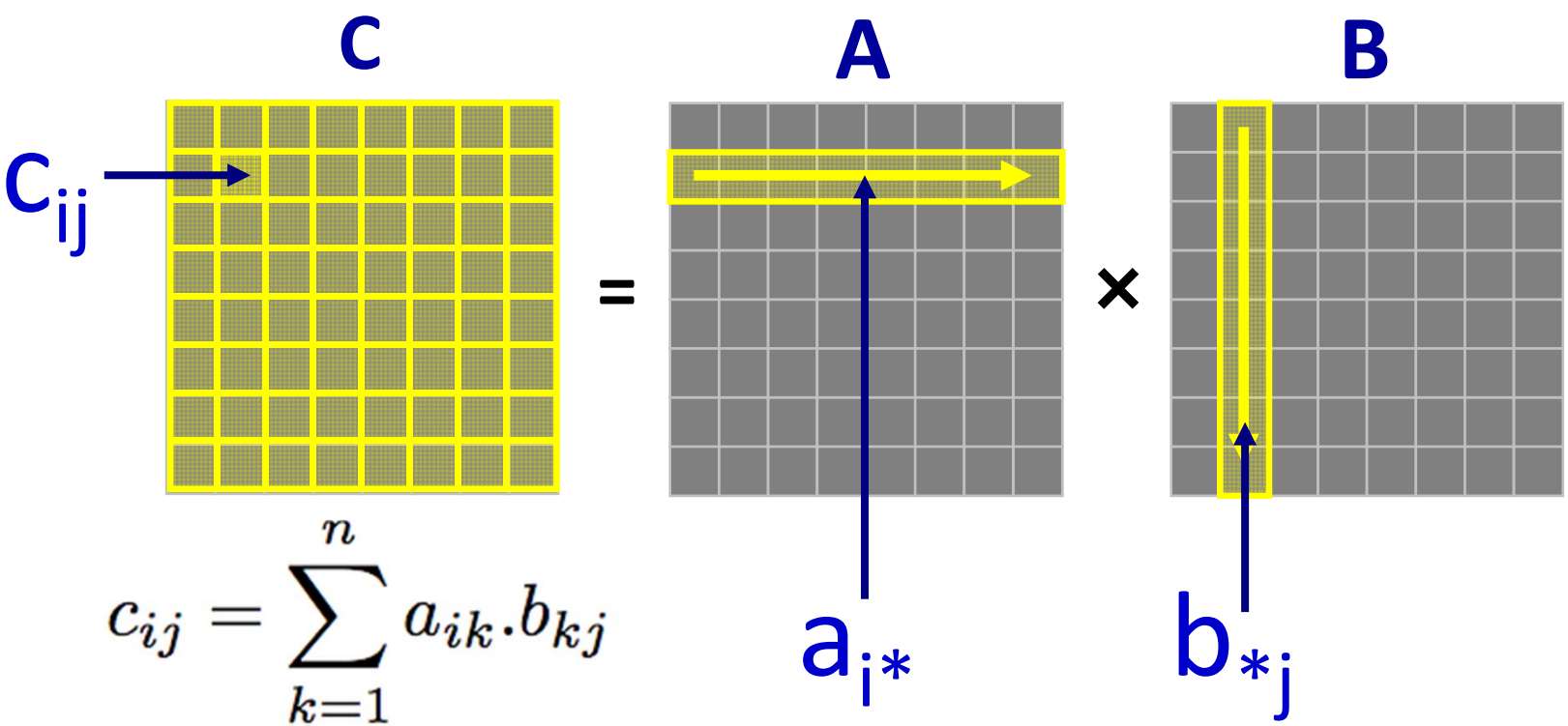
Administrivia

- HW4 due Sunday
- Midterm: 7/19 @ 9am in 1 Pimentel
  - Take old exams for practice (see Piazza post @366)
  - Double-sided sheet of handwritten notes
  - MIPS Green Sheet provided; no calculators
  - Will cover up through caches
- Mid-Semester Survey (part of Lab 6)

Agenda

- Performance Programming
- Administrivia
- Perf Prog: Matrix Multiply
- Technology Trends
  - The Need for Parallelism

Matrix Multiplication



$$c_{ij} = \sum_{k=1}^{n} a_{ik}.b_{kj}$$

Naïve Matrix Multiply

```
for (i=0; i<N; i++)
   for (j=0; j<N; j++)
     for (k=0; k<N; k++)
           c[i][j] += a[i][k] * b[k][j];
```

Advantage: Code simplicity

Disadvantage: Blindly marches through memory and caches

Matrices in Memory (1/2)

- Matrices stored as 1-D arrays in memory
  - Column major: `A(i,j)` at `A+i+j*n`
  - Row major: `A(i,j)` at `A+i*n+j`
    - C default is row major

Column major:

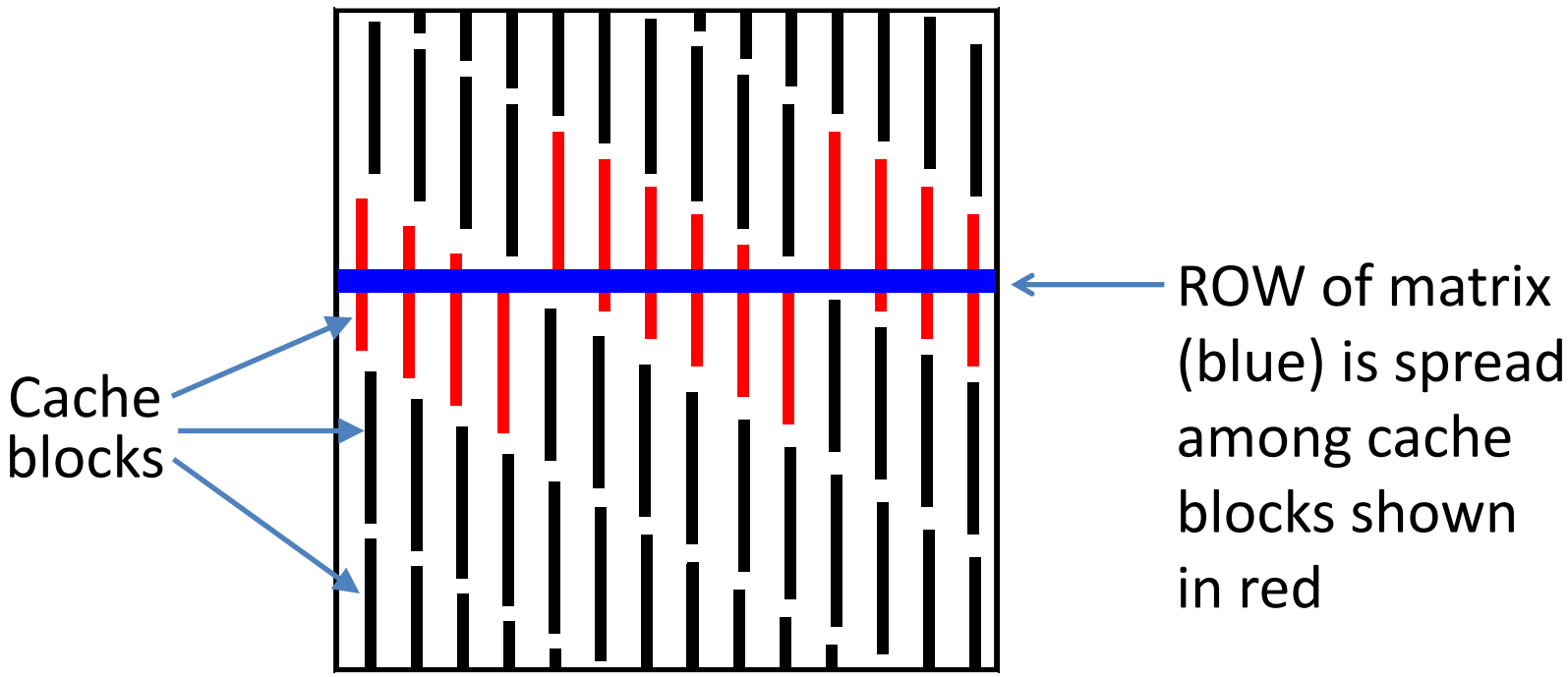
0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

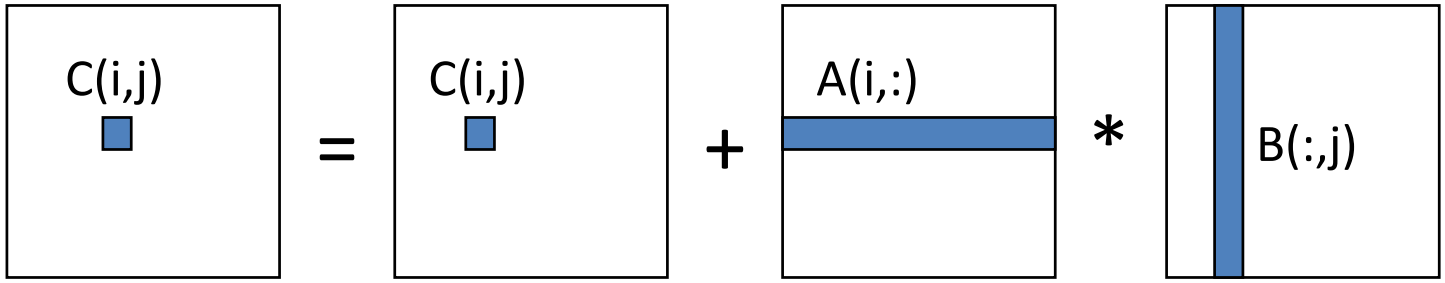
Matrices in Memory (2/2)

- How do cache blocks fit into this scheme?
  - Column major matrix in memory:



Naïve Matrix Multiply (cache view)

```
# move along rows of A
for i = 1 to n
  # move along columns of B
  for j = 1 to n
    # EACH k loop reads row of A, col of B
    # Also read & write c(i,j) n times
    for k = 1 to n
      c(i,j) = c(i,j) + a(i,k) * b(k,j)
```



Linear Algebra to the Rescue (1/2)

- Can get the same result of a matrix multiplication by splitting the matrices into smaller submatrices (matrix "blocks")
- For example, multiply two 4×4 matrices:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \text{ with } B \text{ defined similarly.}$$

$$\boxed{AB = \begin{bmatrix} (A_{11}B_{11} + A_{12}B_{21}) & (A_{11}B_{12} + A_{12}B_{22}) \\ (A_{21}B_{11} + A_{22}B_{21}) & (A_{21}B_{12} + A_{22}B_{22}) \end{bmatrix}}$$

Linear Algebra to the Rescue (2/2)

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{144}

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

Matrices of size N×N, split into 4 blocks of size r (N=4r).

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$$

- Multiplication operates on small "block" matrices
  - Choose size so that they fit in the cache!

Blocked Matrix Multiply

- Blocked version of the naïve algorithm:

```
for(i = 0; i < N/r; i++)
  for(j = 0; j < N/r; j++)
    for(k = 0; k < N/r; k++)
      C[i][j] += A[i][k]*B[k][j]
```

r × r matrix addition

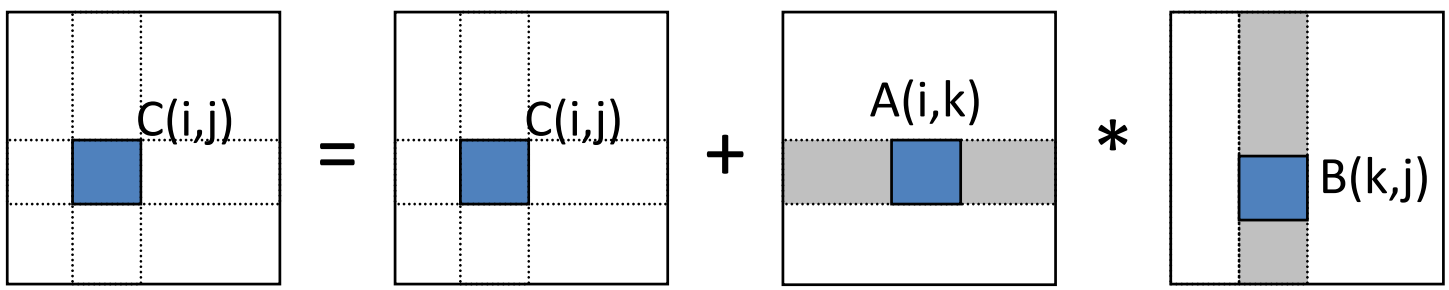
r × r matrix multiplication

  - r = matrix block size (assume r divides N)
  - X[i][j] = submatrix of X, defined by block row i and block column j

Blocked Matrix Multiply (cache view)

```
# move along block row of A
for i = 1 to N
  # move along block col of B
  for j = 1 to N
    # each k loop reads block of A and B
    # Also read and write block of C
    for k = 1 to N
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Matrix multiply on blocks



Matrix Multiply Comparison

- Naïve Matrix Multiply
  - N = 100, 1000 cache blocks, 1 word/block
  - Youtube: Slow/Fast-forward
  - ≈ 1,020,0000 cache misses
- Blocked Matrix Multiply
  - N = 100, 1000 cache blocks, 1 word/block, r = 30
  - Youtube: Slow/Fast-forward
  - ≈ 90,000 cache misses

Maximum Block Size

- Blocking optimization only works if the *blocks fit in cache*
  - Must fit 3 blocks of size r × r in memory (for A, B, and C)
- For cache of size M (in elements/words), we must have $3r^2 \approx M$, or $r \approx \sqrt{(M/3)}$
- Ratio of cache misses unblocked vs. blocked *up to* $\approx \sqrt{M}$ (play with sizes to optimize)
  - From comparison: ratio was ≈ 11, $\sqrt{M} = 31.6$

Get To Know Your Staff

- Category: **Food**

Agenda

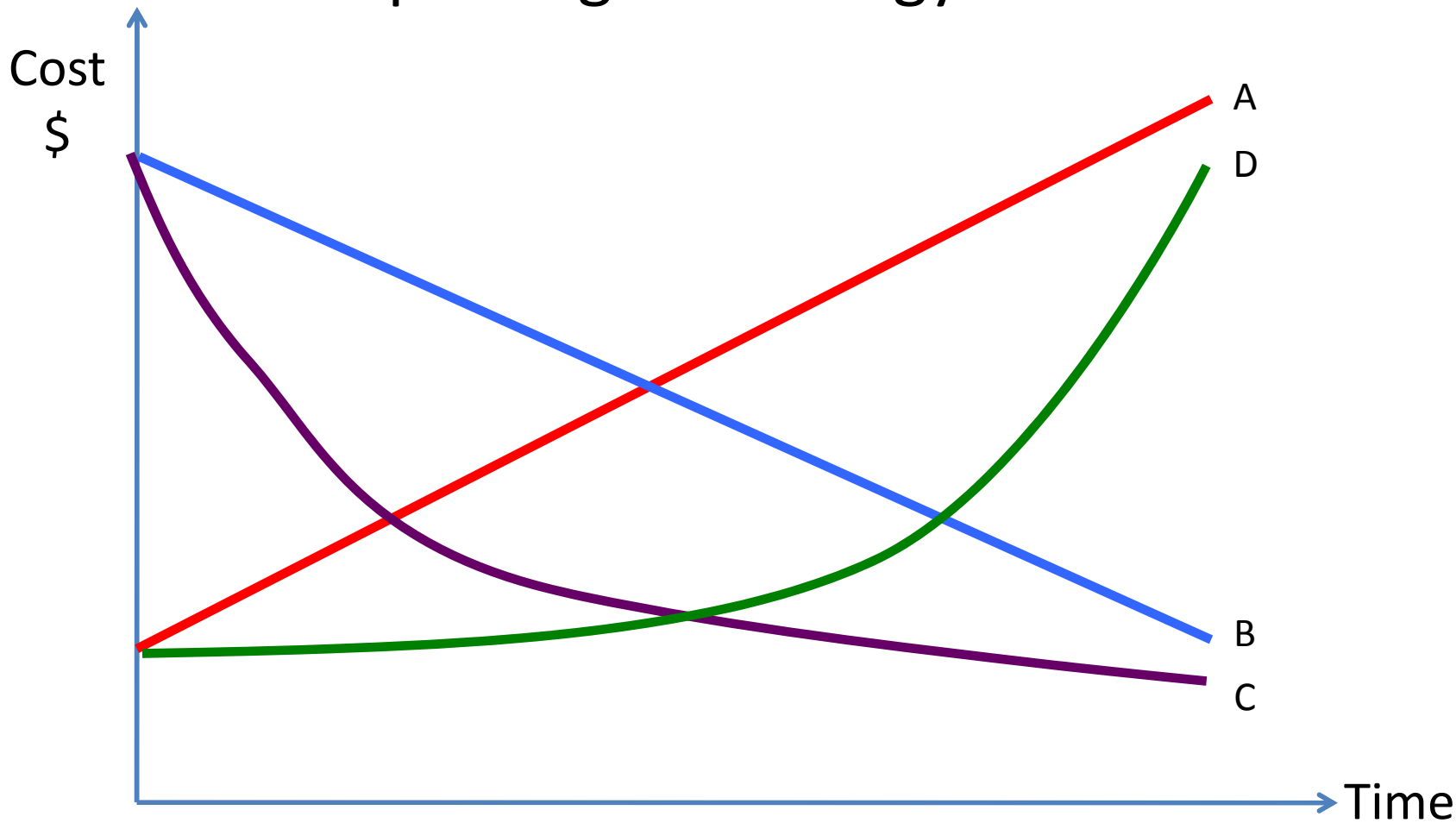
- Performance Programming
- Administrivia
- Perf Prog: Matrix Multiply
- Technology Trends
  - The Need for Parallelism

Six Great Ideas in Computer Architecture

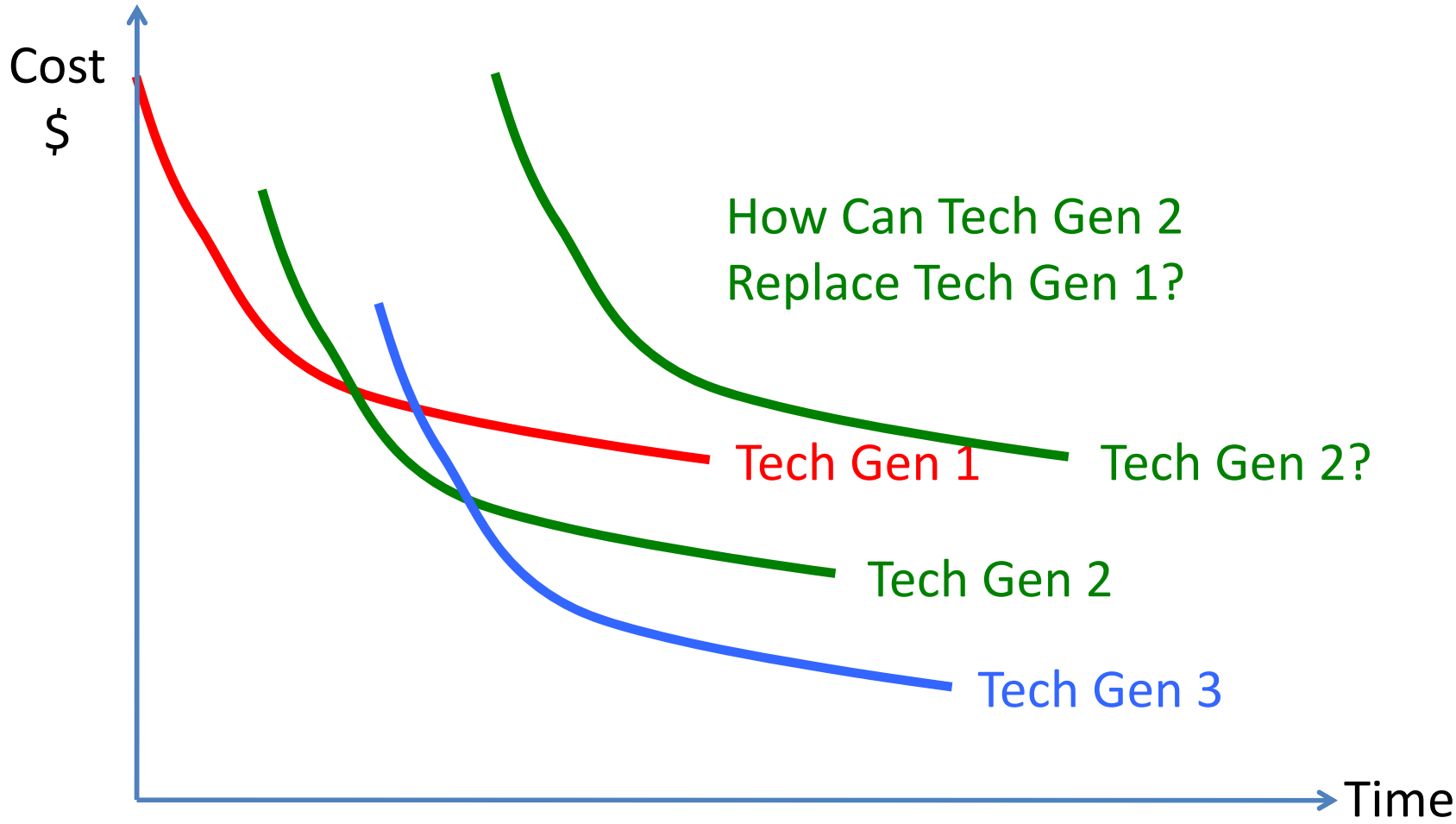
1. Layers of Representation/Interpretation
2. Moore's Law
3. Principle of Locality/Memory Hierarchy
4. Parallelism
5. Performance Measurement & Improvement
6. Dependability via Redundancy

Technology Cost over Time

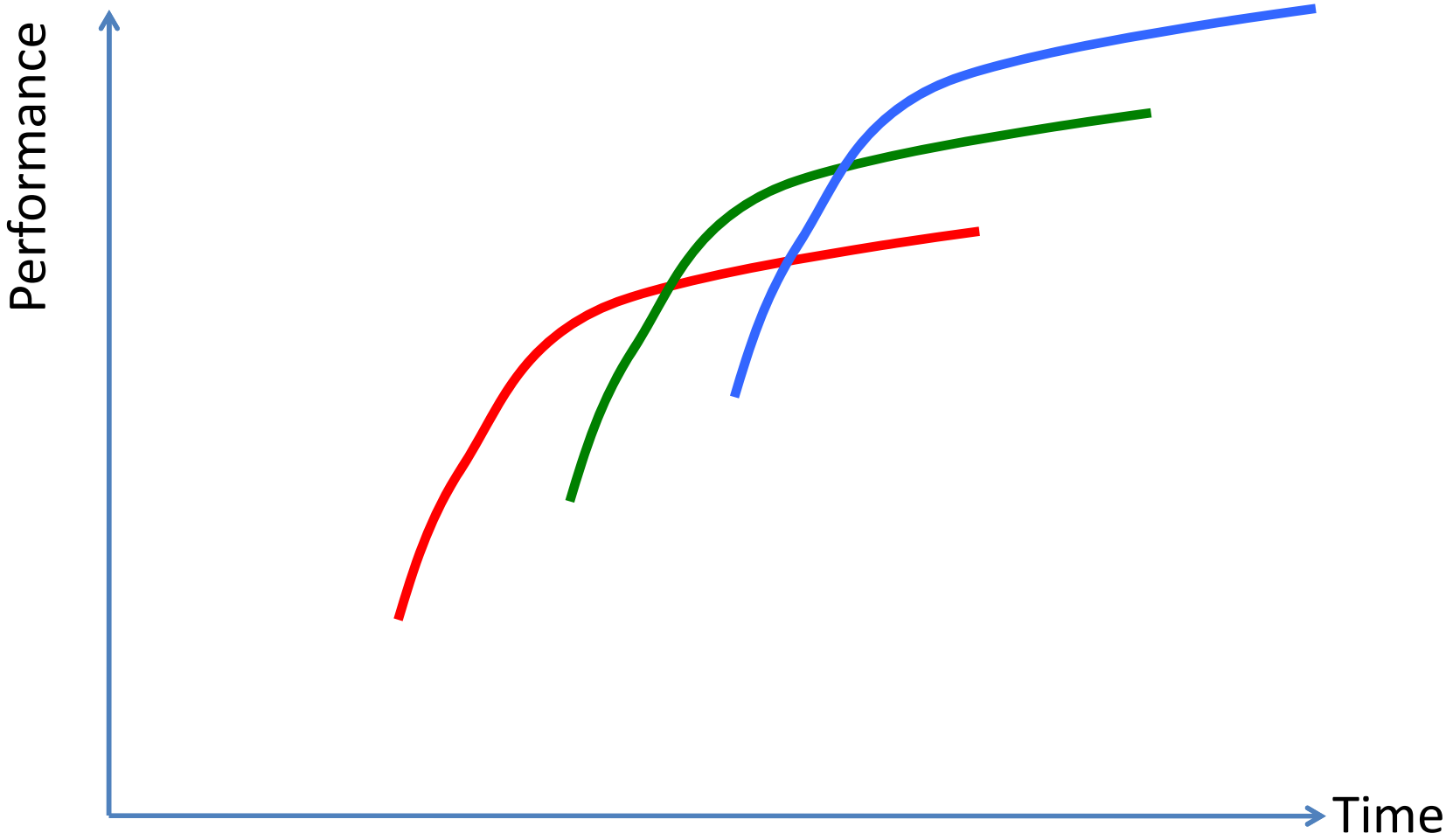
What does improving technology look like?



Tech Cost: Successive Generations

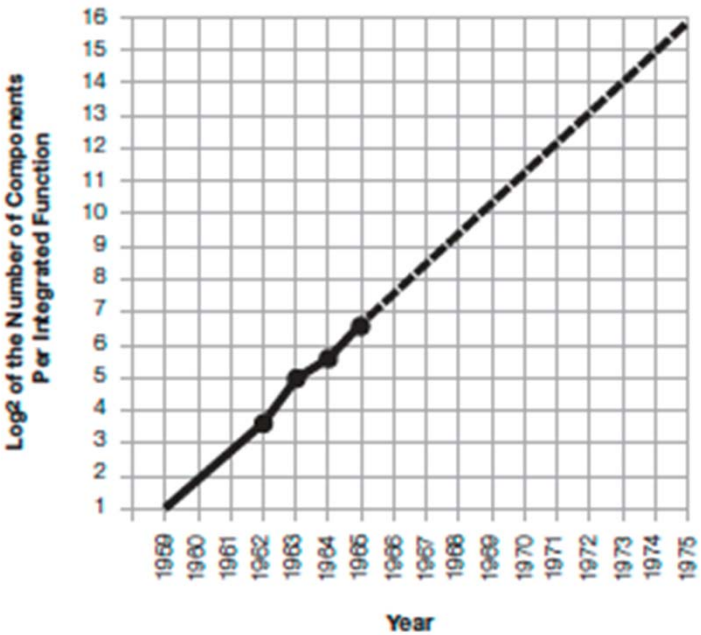


Tech Performance over Time



Moore's Law

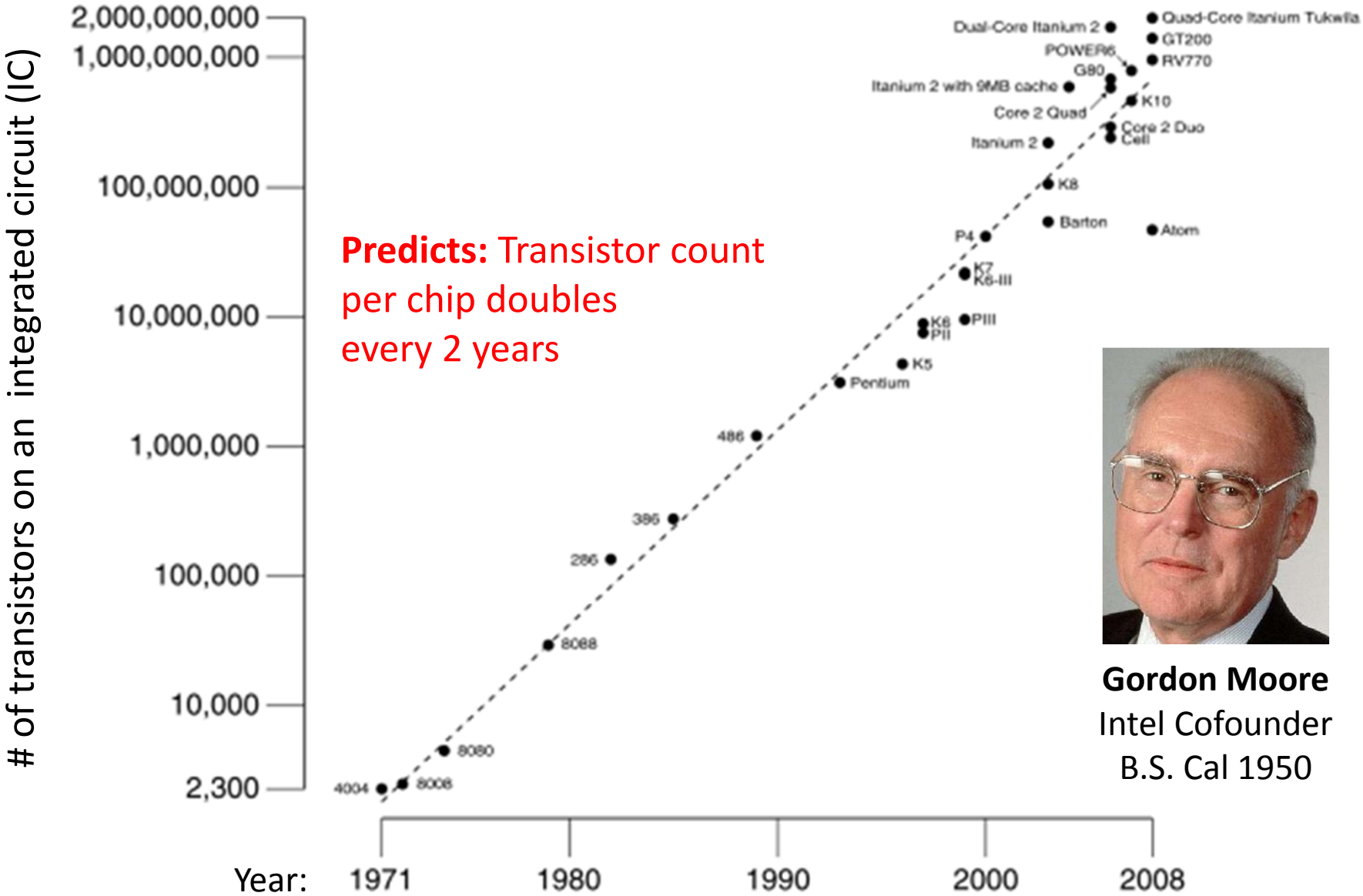
"The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. …That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000." *(from 50 in 1965)*



Gordon Moore, "Cramming more components onto integrated circuits," *Electronics*, Volume 38, Number 8, April 19, 1965

"Integrated circuits will lead to such wonders as home computers--or at least terminals connected to a central computer--automatic controls for automobiles, and personal portable communications equipment. The electronic wristwatch needs only a display to be feasible today."

Great Idea #2: Moore's Law



Predicts: Transistor count per chip doubles every 2 years

Gordon Moore
Intel Cofounder
B.S. Cal 1950

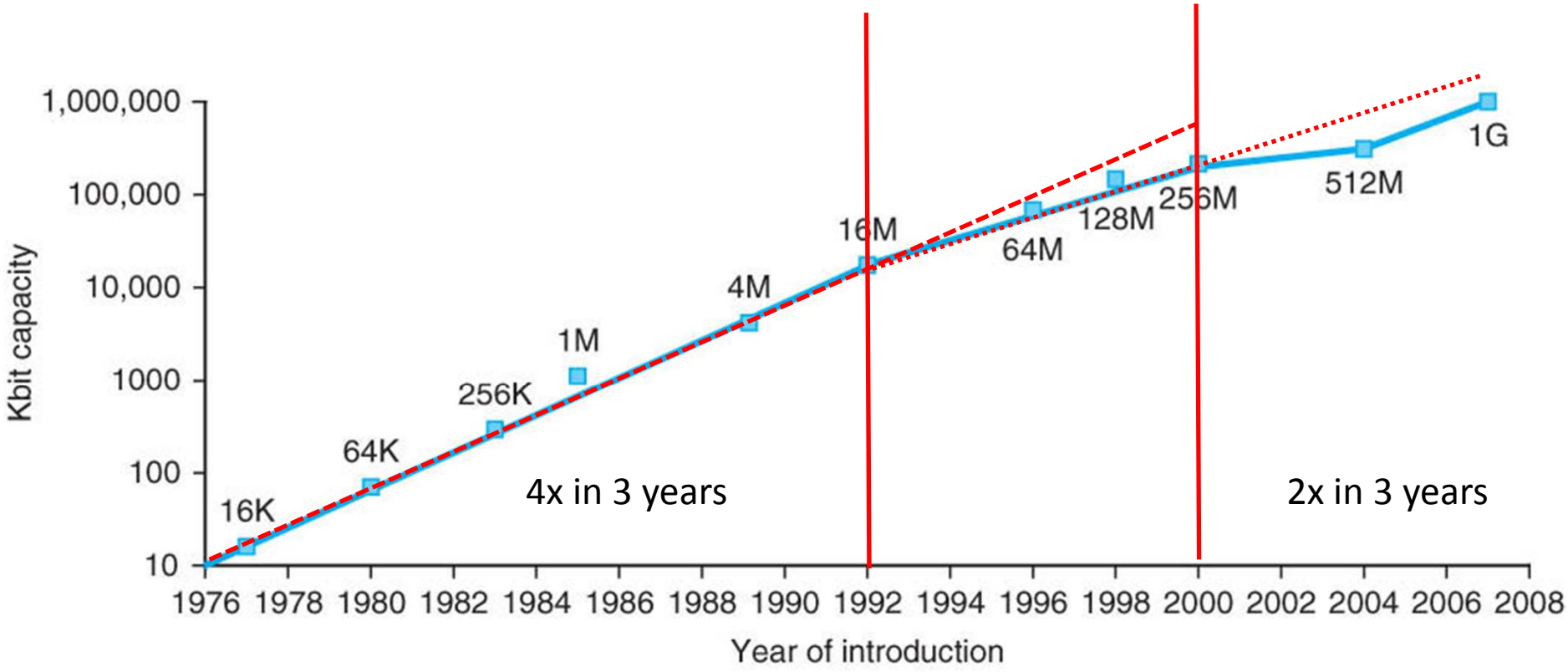
End of Moore's Law?

- Exponential growth cannot last forever
- More transistors/chip will end during your careers
  - 2020? 2025?
  - (When) will something replace it?
- It's also a law of investment in equipment as well as increasing volume of integrated circuits that need more transistors per chip

Computer Technology: Growing, But More Slowly

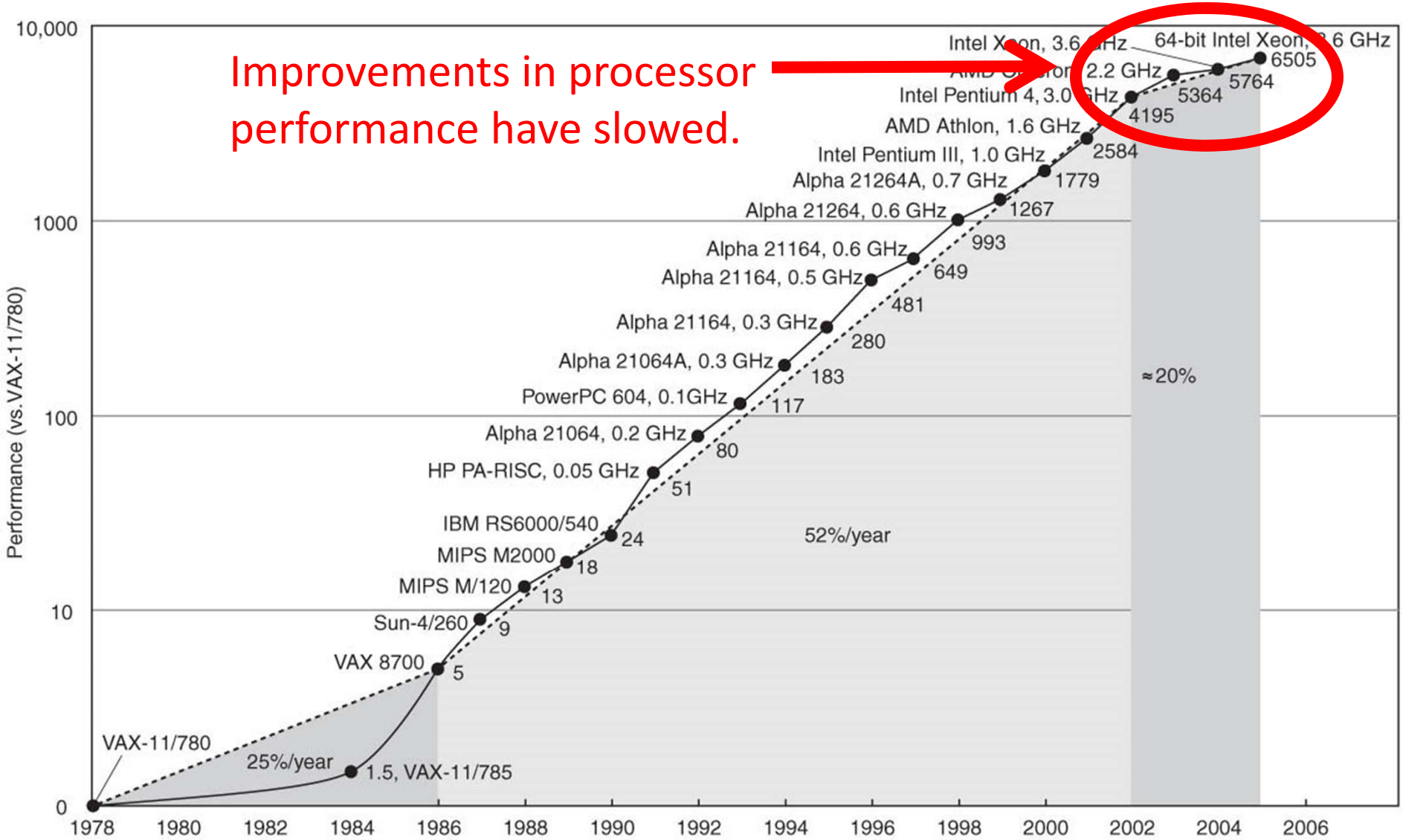
- Processor
  - Speed 2x / 1.5 years (since ’85-’05) [***slowing***!]
  - Now +2 cores / 2 years
  - When you graduate: 3-4 GHz, 6-8 Cores in client, 10-16 in server
- Memory (DRAM)
  - Capacity: 2x / 2 years (since ’96) [***slowing***!]
  - Now 2X/3-4 years
  - When you graduate: 8-16 GigaBytes
- Disk
  - Capacity: 2x / 1 year (since ’97)
  - 250X size last decade
  - When you graduate: 6-12 TeraBytes
- Network
  - Core: 2x every 2 years
  - Access: 100-1000 mbps from home, 1-10 mbps cellular

Memory Chip Size

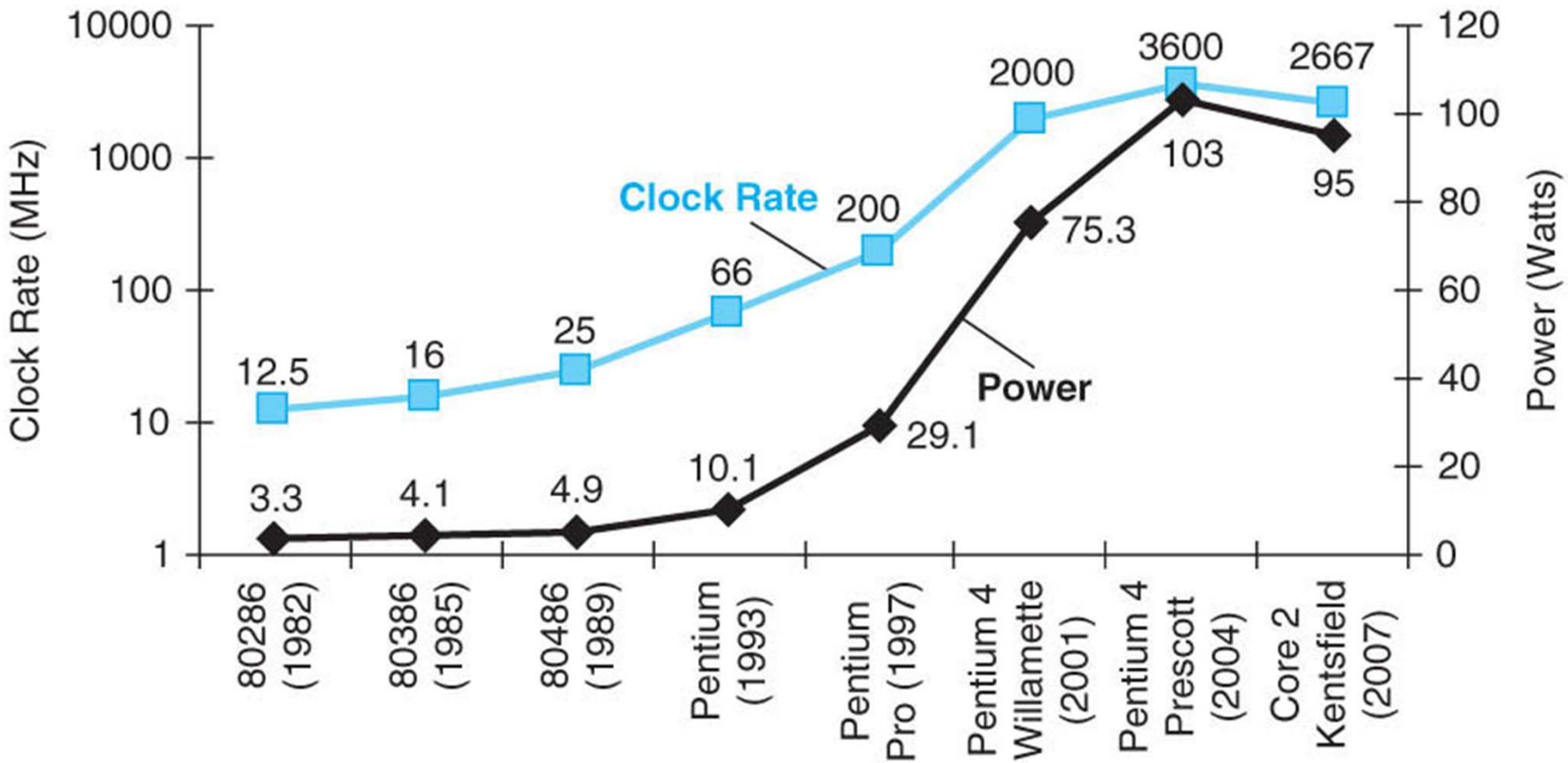


Growth in memory capacity slowing

Uniprocessor Performance



Limits to Performance: Faster Means More Power

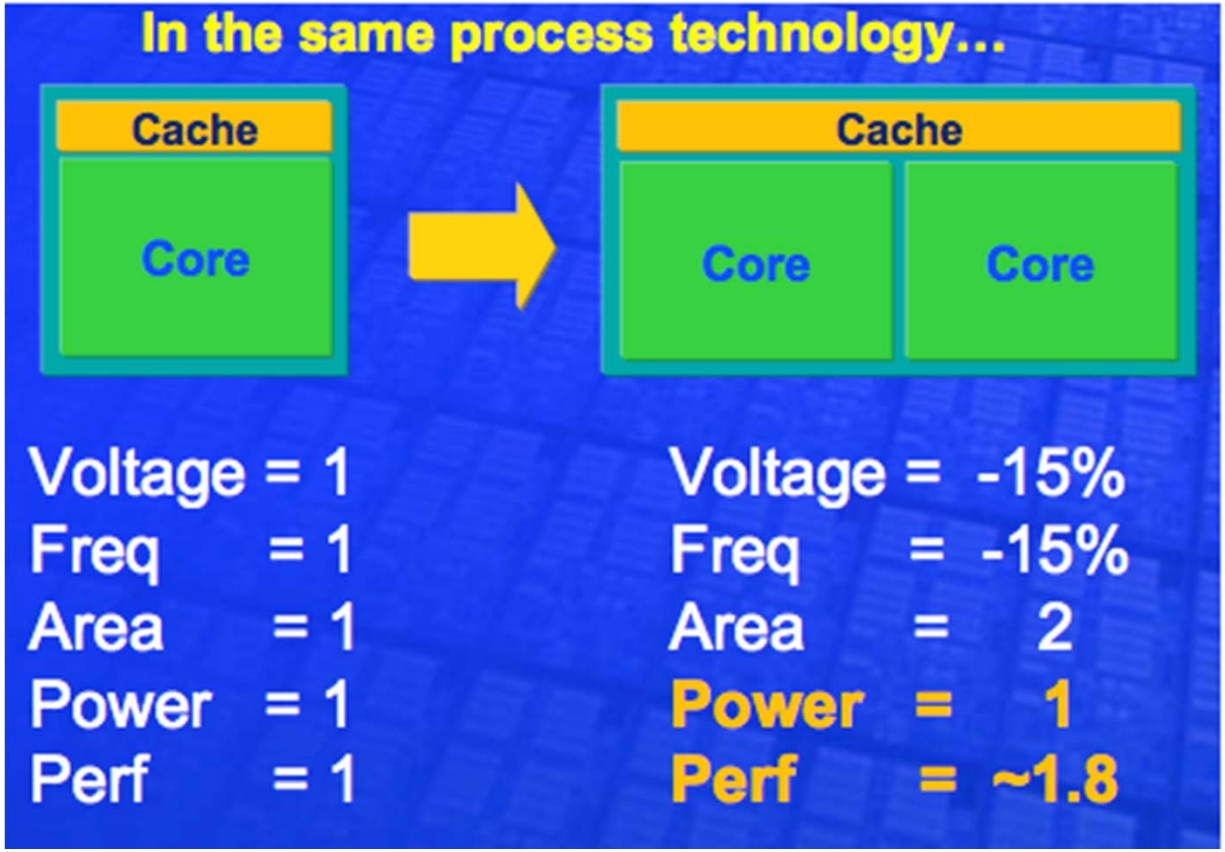


Dynamic Power

- Power = $C \times V^2 \times f$
  - Proportional to capacitance, voltage^2, and frequency of switching
- What is the effect on power consumption of:
  - "Simpler" implementation (fewer transistors)? ↓
  - Reduced voltage? ↓↓
  - Increased clock frequency? ↑

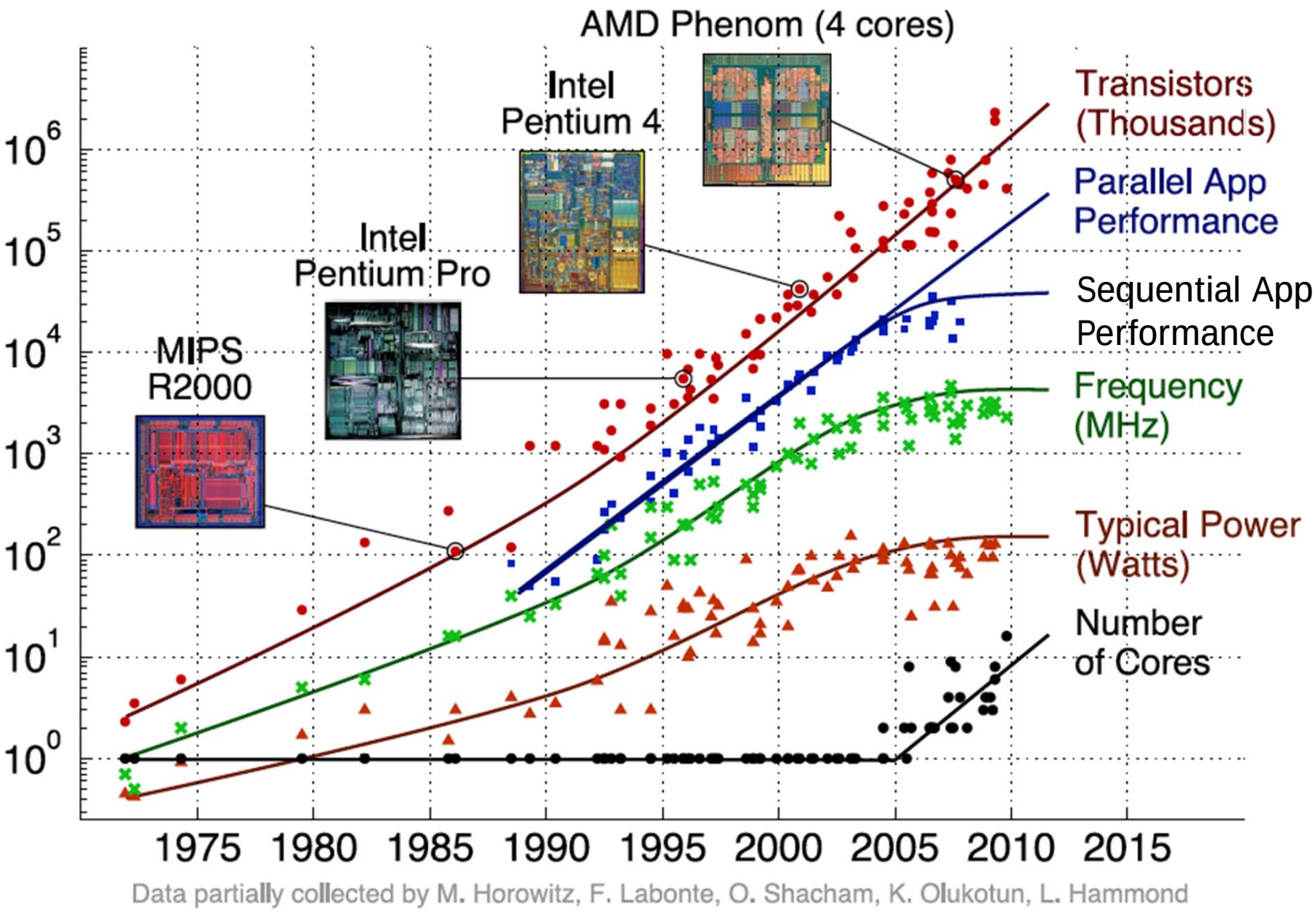
Multicore Helps Energy Efficiency

- Power = C × V^2 × f



From:
William Holt,
HOT Chips 2005

Transition to Multicore



Parallelism - The Challenge

- Only path to performance is parallelism
  - Clock rates flat or declining
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e. that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project #2: fastest matrix multiply code on 16 processor (cores) computers

Summary

- Performance programming
  - With understanding of your computer's architecture, can optimize code to take advantage of your system's cache
  - Especially useful for loops and arrays
  - "Cache blocking" will improve speed of Matrix Multiply with appropriately-sized blocks
- Processors have hit the power wall, the only option is to go parallel