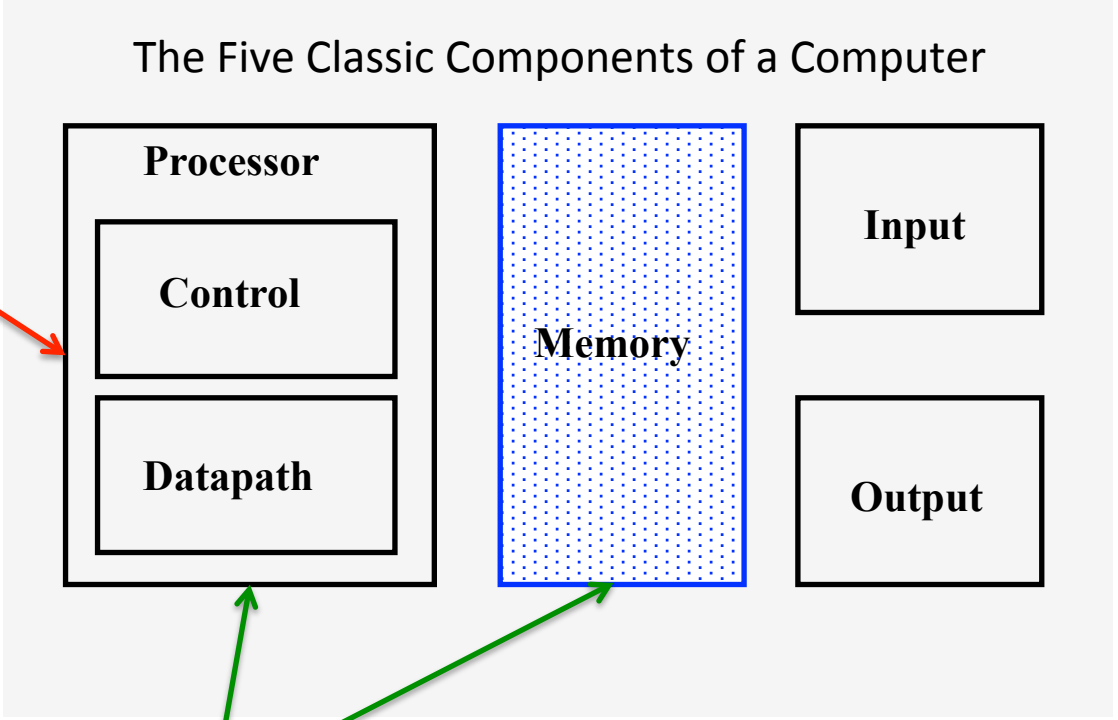
Multicore and Roofline Model

Sarala Arunagiri

26 September 2013

The Big Picture: Where are We Now?

Yesterday
Achieve performance increase
Via
Discovering ILP and exploiting it using superscalar, pipelining, and SIMD techniques.



Today
Achieve high performance
Via

(a) Solving performance problems caused by the discrepancy in speed between CPU and RAM (memory hierarchy using caches)
(b) Learning to write codes that exploit caches.
(c) Exploiting technology trends stated by Moore's law in Processor design (Multicore).
Learn about cache coherence and some protocols for achieving it.

Source: Adapted from David Patterson's lecture slide for CS252 Spring 2000

Today

Achieve high performance Via

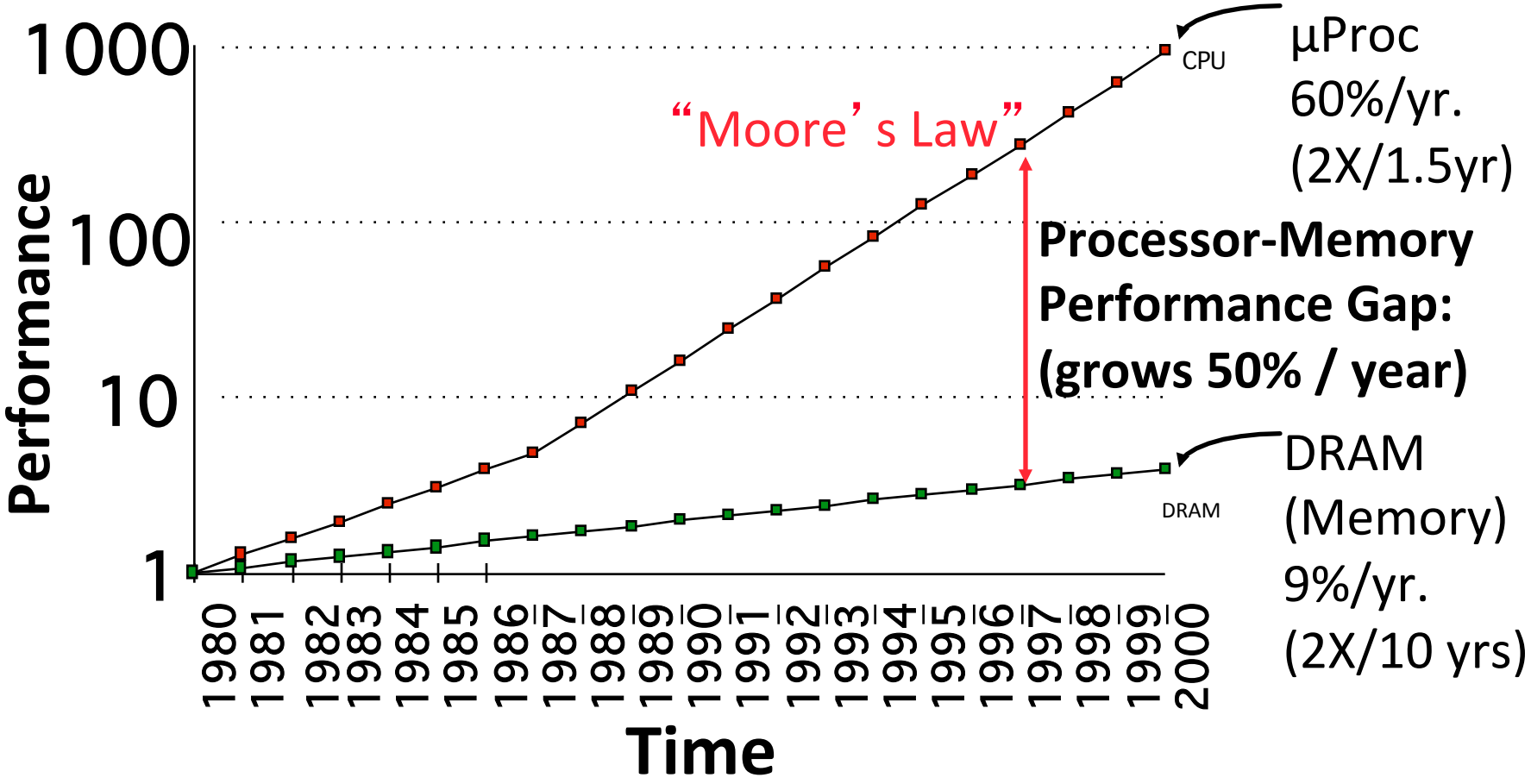
a) Solving performance problems caused by the discrepancy in speed between CPU and RAM (memory hierarchy using caches)

b) Learning to write codes that exploit caches.

c) Exploiting technology trends stated by Moore's law in Processor design (Multicore). Learn about cache coherence and some protocols for achieving it.

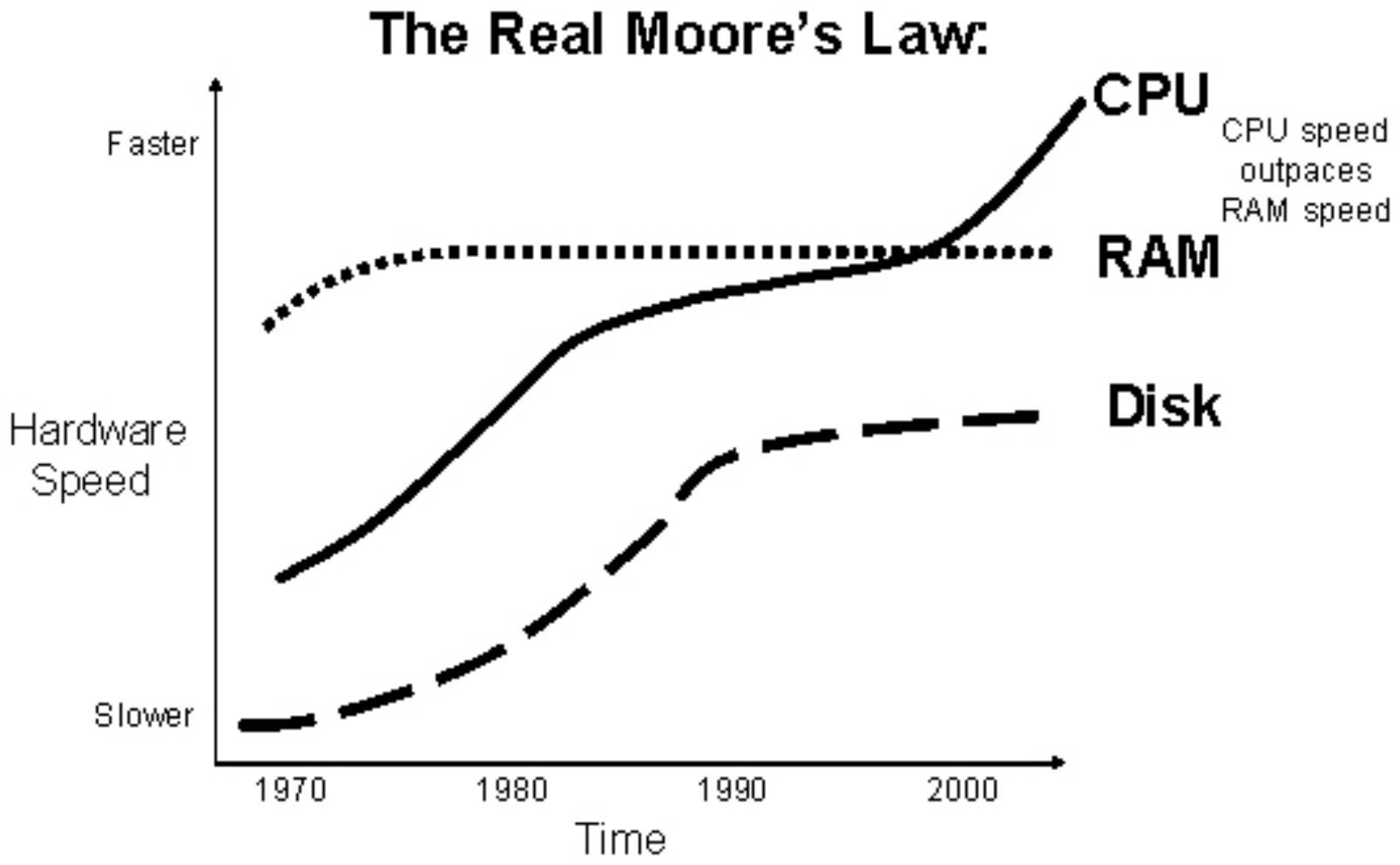
Technology trends

Processor-DRAM Memory Gap (latency)



Source: Adapted from David Patterson's lecture slide for CS252 Spring 2000

Rate of Growth of CPU, RAM, and Disk Speeds



Source: http://www.dba-oracle.com/art_dbazine_oracle_10g_data_warehouse.htm

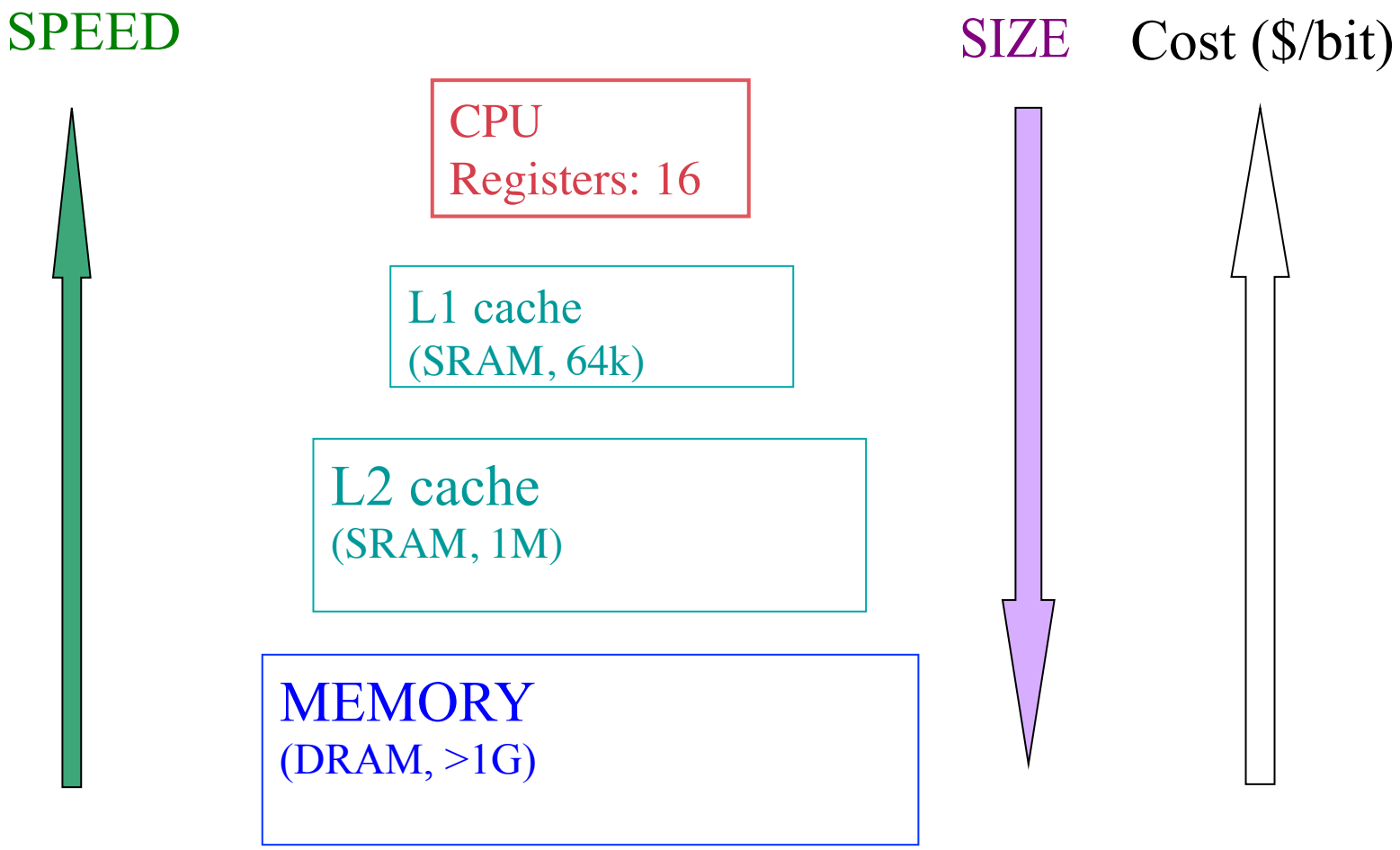
Facts about memory

- Memory is too slow to keep up with the processor
  - 100--1000 cycles latency before data arrives
- At considerable cost it's possible to build faster memory
- Cache is small amount of fast memory

Memory Hierarchies

- Memory is divided into different levels:
  - Registers
  - Caches
  - Main Memory
- Memory is accessed through the hierarchy
  - registers where possible
  - ... then the caches
  - ... then main memory

Memory Relativity



Cache and register access

Access is transparent to the programmer

- Data is in a register or in cache or in memory
- Loaded from the highest level where it's found
- Processor/cache controller/Memory Management Unit (MMU) hides cache access from the programmer

…but a programmer might be able to influence it.

Need an understanding of: temporal locality, spatial locality, data reuse, and register use.

Also need to understand the performance penalties of TLB misses and how to manage it. This will be covered in the lecture on the 3rd October 2013.

Today

Achieve high performance Via

a) Solving performance problems caused by the discrepancy in speed between CPU and RAM (memory hierarchy using caches)

b) Learning to write codes that exploit caches.

c) Exploiting technology trends stated by Moore's law in Processor design (Multicore). Learn about cache coherence and some protocols for achieving it.

Locality of Reference: a Type of Predictable Behavior in Computer Systems – Spatial Locality

Analysis of data locations referenced in a short period of time can be expected to have clusters. Here we study two types of locality: Spatial and Temporal.

(1) Spatial locality: if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future.

Influence on cache design: cache blocks/cache line size and prefetching.

How can a programmer use this information:

- Cache lines: if the cache line is already loaded, other elements are 'for free'.
- Reuse, as soon as possible, items known to be stored close together in the address space.
- TLB: don't jump more than 512 words too many times.

Spatial Locality: Example

Exercise 1: Assuming the matrix is stored in a row major fashion (C programming language), which of the following codes have spatial locality of reference?

Code segment (1)

```
for (j=0; j<N; j++) {
        for(i=0; i<M; i++) {
            x[i,j] = x[i,j]*x[i,j]
        }
    }
```

Code segment (2)

```
for (i=0; i<N; i++) {
    for(j=0; j<M; j++) {
        x[i,j] = x[i,j]*x[i,j]
    }
}
```

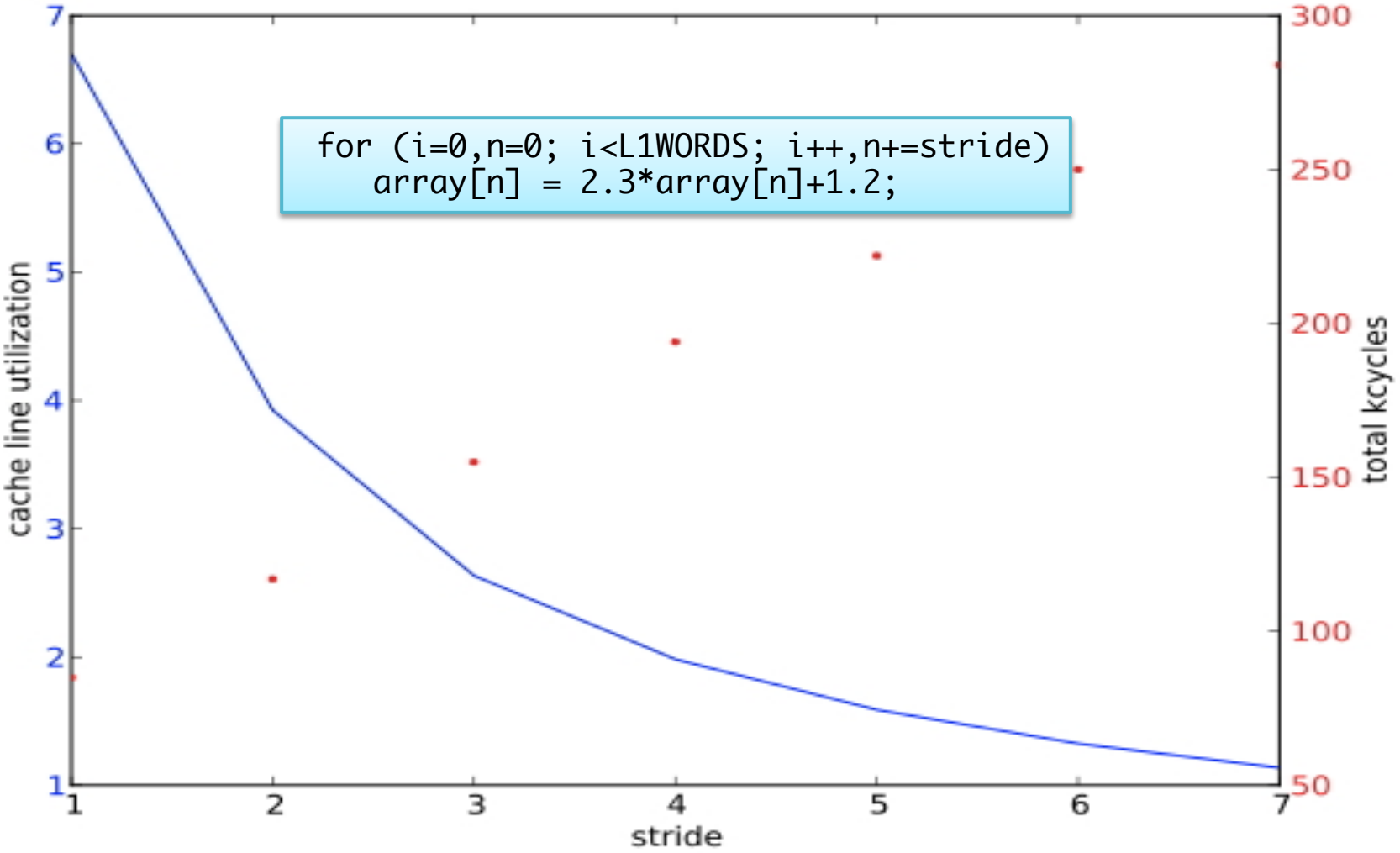
Exercise 2: How about if the array was stored in a column major fashion, like with FORTRAN?

Performance Illustration

```
for (i=0; i< K; i++){
    for(n=0; n<N ; n+=stride){
    array[n] = 2.3*array[n]+1.2;
    }
}
```

As you increase the value of stride, assuming N>> size of L1 cache, what trends do you expect for the time taken to compute the code segment?

Cache line Utilization



Observations:

- Same amount of data, but increasing stride
- Increasing stride: more cachelines loaded, slower execution

Locality of Reference: a type of predictable behavior in computer systems – Temporal locality

(2) Temporal locality: if at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future.

Influenced cache replacement policy - LRU (Least Recently Used) .

How can a programmer use this:

Use an item, use it again before it is flushed from register or cache.

Temporal locality: Example I

Exercise 3. Assuming that the cache size is smaller than N, does the code segment 3 exploit spatial locality for performance?

Code segment (3):

```
for (p=0; p<10; p++) {
    for (i=0; i<N; i++) {
        if (i==0) x[i]=0;
        x[i] = x[i]+p*x[i]
    }
}
```

Temporal locality: Example II

No. Code segment (3) long time between successive uses of *x[i]*. When the loops are rearranged as in Code segment (4) *x[i]* is reused.

```
Code segment (4)
for (i=0; i<N; i++) {
    for (p=0; p<10; p++) {
        if(i==0) x[i]=0;
        x[i] = x[i]+p*x[i];
    }
}
```

Register use

- y[i] can be kept in register
- Declaration is only suggestion to the compiler
- Compiler can usually figure this out itself

```
Code segment(5)
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    Y[i] = Y[i]+A[i][j]*X[j];
  }
}
```

```
Code segment(6)
register double s;
for (i=0; i<n; i++) {
  s = 0.;
  for (j=0; j<n; j++) {
    s = s+A[i][j]*X[j];
  }
  Y[i] = s;
}
```

Data reuse

Since memory accesses could be detrimental for high performance, data items are used multiple times, i.e., reused, while they reside in the cache, whenever possible, rather than accessing the same data again from the memory.

Theoretical analysis of performance

Given the different speeds of memory & processor, the question is: does my algorithm exploit all these caches? from a theoretic analysis, is there a potential to do so? In practice, does it indeed exploit the caches?

Analysis to Determine Potential for Data Reuse

Exercise 4: Is there a potential for data reuse in

(a) Matrix addition C=A+B?

(b) Matrix multiplication C=A*B?

(c) Matrix vector product AX=Y?

where A,B, and C are n x n matrices and X and Y are n vectors.

Data reuse: matrix-matrix product

Matrix-matrix product: $2n^3$ ops, $2n^2$ data

```
Code segment(7)
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    s = 0;
    for (k=0; k<n; k++) {
      s = s+A[i][k]*B[k][j];
    }
    C[i][j] = s;
  }
}
```

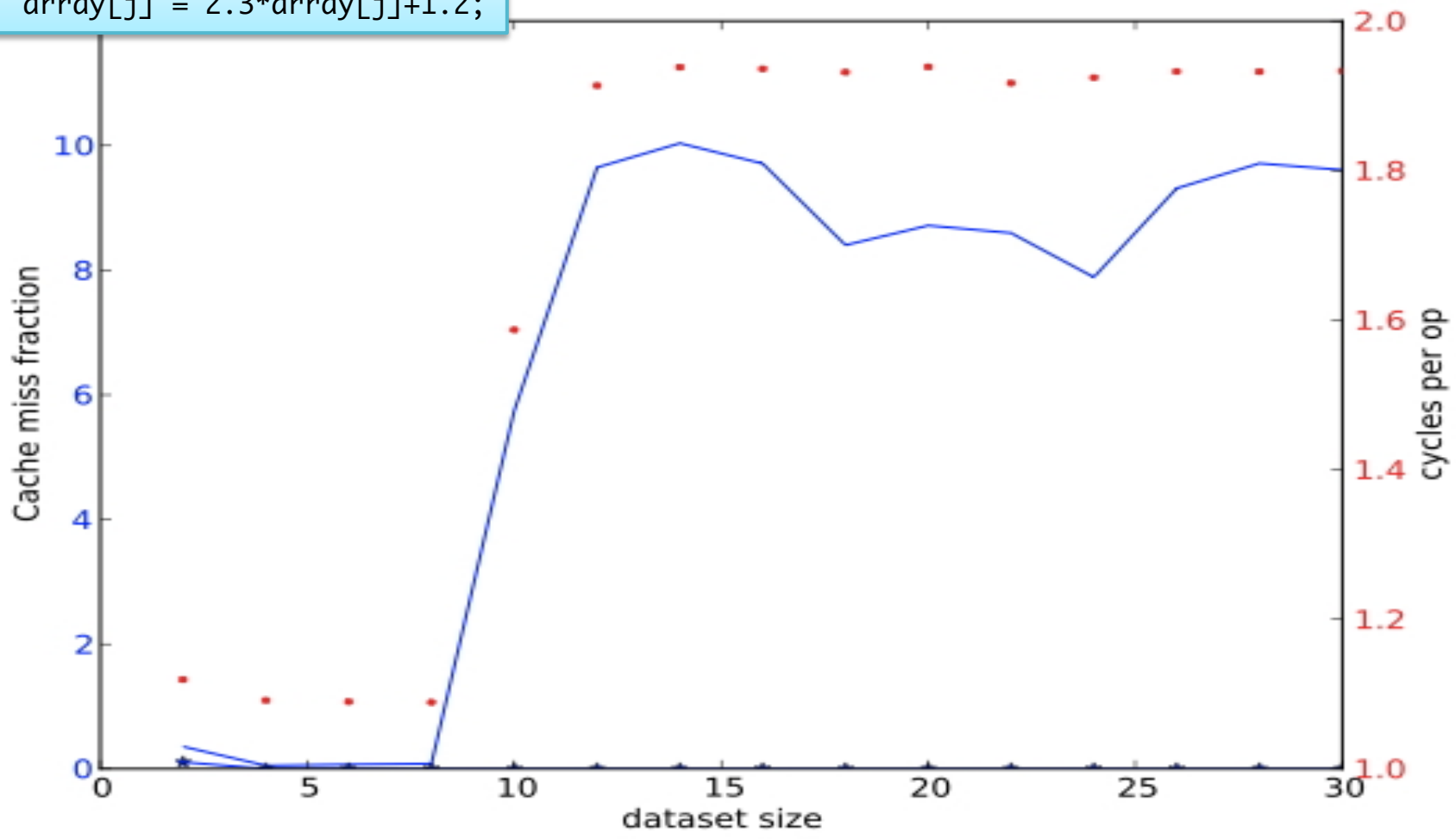
Exercise 5: Is there data reuse in this algorithm?

(a)Assuming matrices A, B, and C fit on the L1 cache.

(b)The cache is not large enough to hold any one of the Matrices A,B, or C.

Relationship Between Cache Size, Input Size, and Performance

```
for (i=0; i<NRUNS; i++)
  for (j=0; j<size; j++)
    array[j] = 2.3*array[j]+1.2;
```



- If the data fits in L1 cache, the transfer is very fast
- If there is more data, transfer speed from L2 dominates

Reuse analysis: matrix-vector product

```
Code segment(8)
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    Y[i] = Y[i]+A[i][j]*X[j];
  }
}
```

Y[i] invariant but not reused: arrays get written back to memory, so 2 accesses just for y[i]

```
Code segment(9)
for (i=0; i<n; i++) {
  s = 0.;
  for (j=0; j<n; j++) {
    s = s+A[i][j]*X[j];
  }
  Y[i] = s;
}
```

s stays in register

Programming for high performance is based on spatial and temporal locality

- Temporal locality:
  - Group references to one item close together:
- Spatial locality:
  - Group references to nearby memory items together

Today

Achieve high performance Via

a) Solving performance problems caused by the discrepancy in speed between CPU and RAM (memory hierarchy using caches)

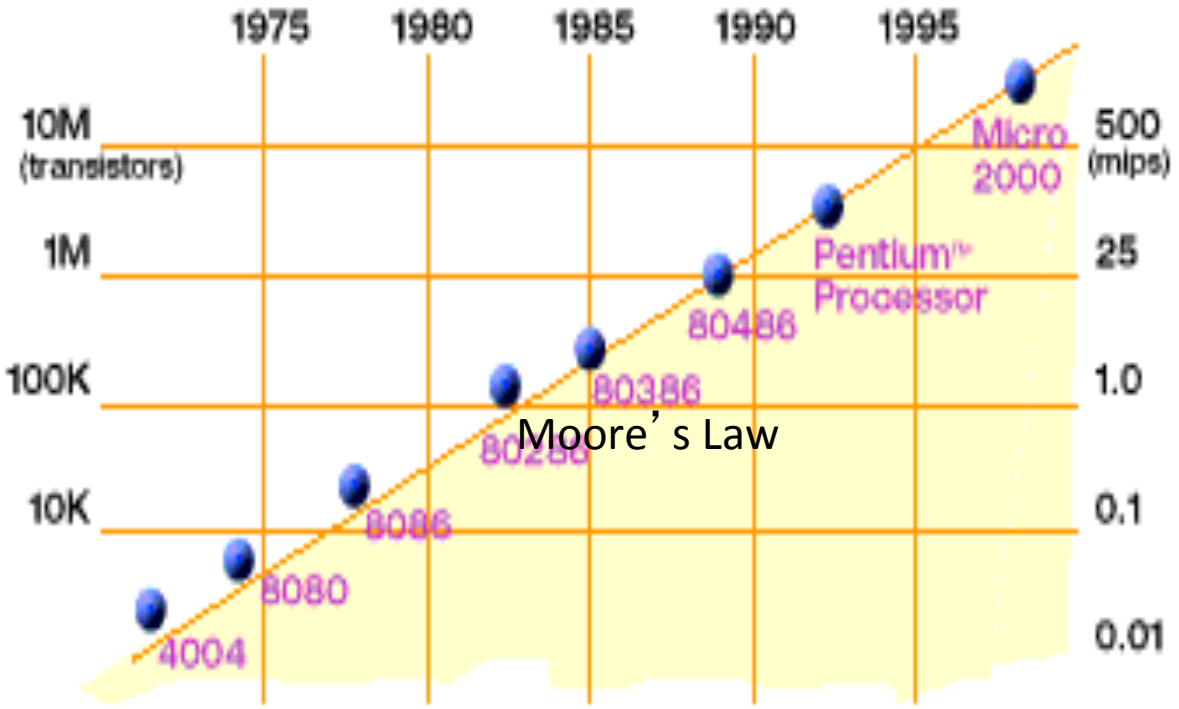
b) Learning to write codes that exploit caches.

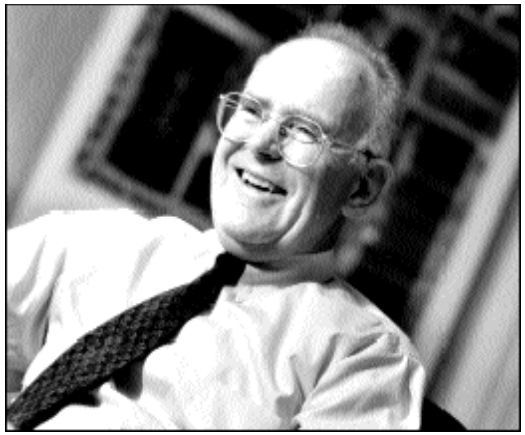
c) Exploiting technology trends stated by Moore's law in Processor design (Multicore). Learn about cache coherence and some protocols for achieving it.

Multi-core Processors

- What is a processor? Instead, talk of “socket” and “core”
- Cores have separate L1, shared L2 cache
  - Hybrid shared/distributed model
- Cache coherency problem: conflicting access to duplicated cache lines.

Technology Trends: Microprocessor Capacity





2X transistors/Chip Every 1.5 years

Called "Moore's Law"

Microprocessors have become smaller, denser, and more powerful .

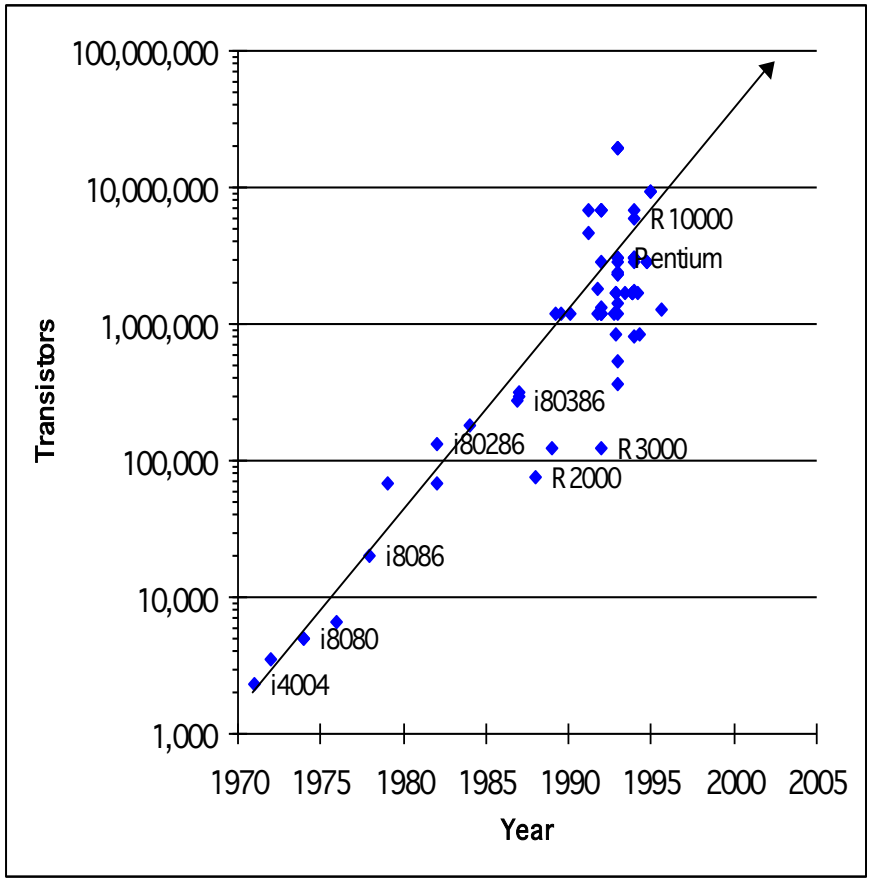
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Slide source: Jack Dongarra

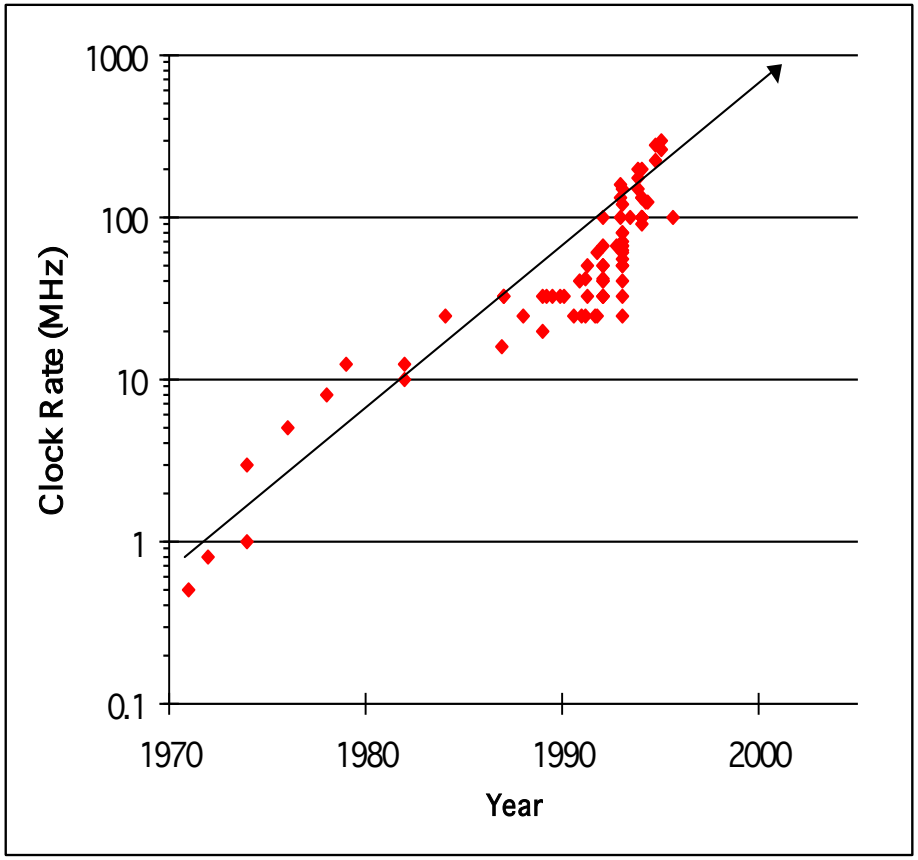
Source: Adapted from Katherine Yelick's lecture slide for CS194

Microprocessor Transistors and Clock Rate

Growth in transistors per chip



Increase in clock rate



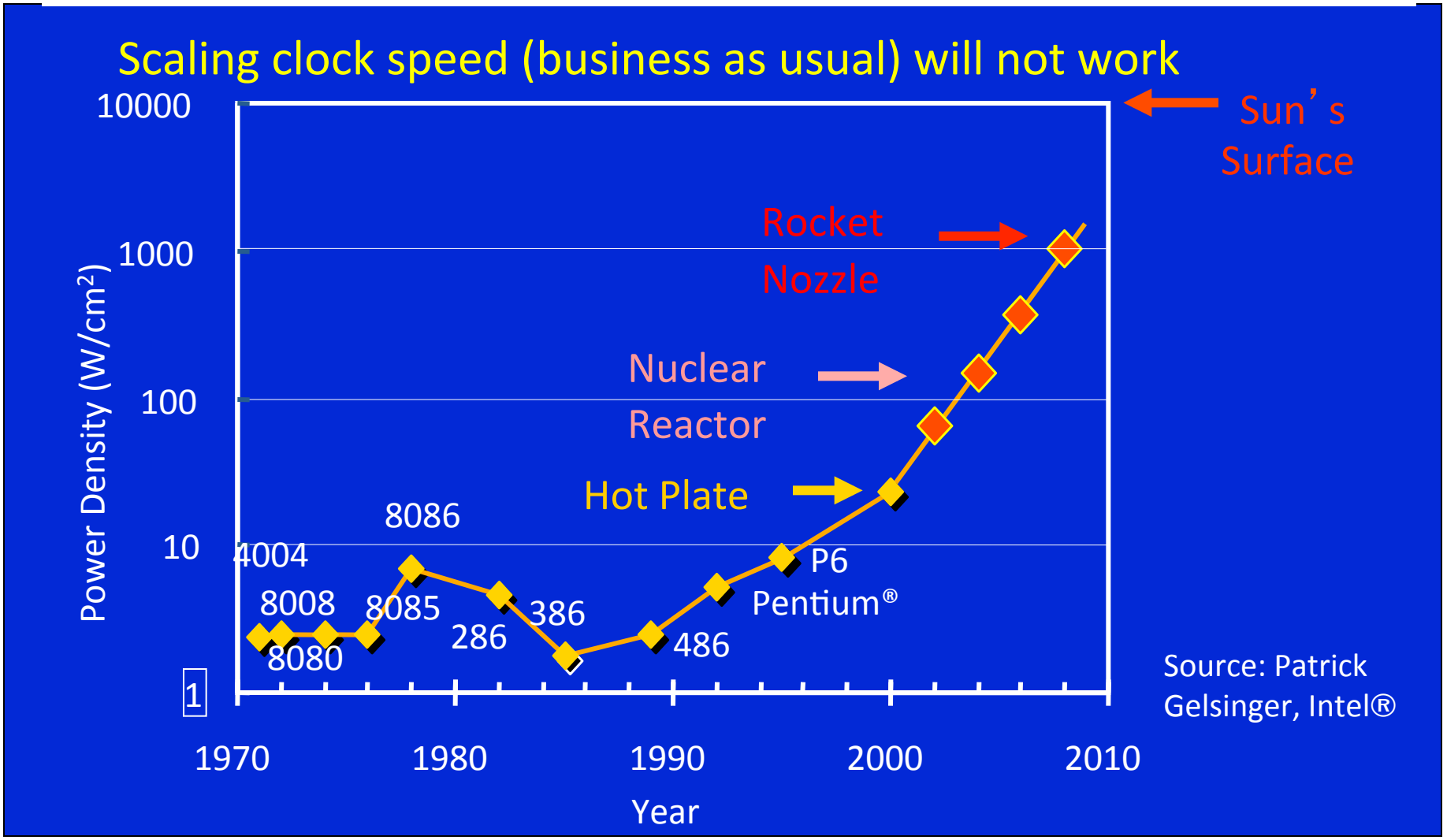
Why bother with parallel programming? Just wait a year or two…

Source: Adapted from Katherine Yelick's lecture slide for CS194

Limit #1: Power density

Can soon put more transistors on a chip than can afford to turn on.

-- Patterson '07



Source: Adapted from Katherine Yelick's lecture slide for CS194

Parallelism Saves Power

- Exploit explicit parallelism for reducing power

$$\text{Power} = (C * V^2 * F)/4 \qquad \text{Performance} = (\text{Cores} * F)*1$$

Capacitance Voltage Frequency

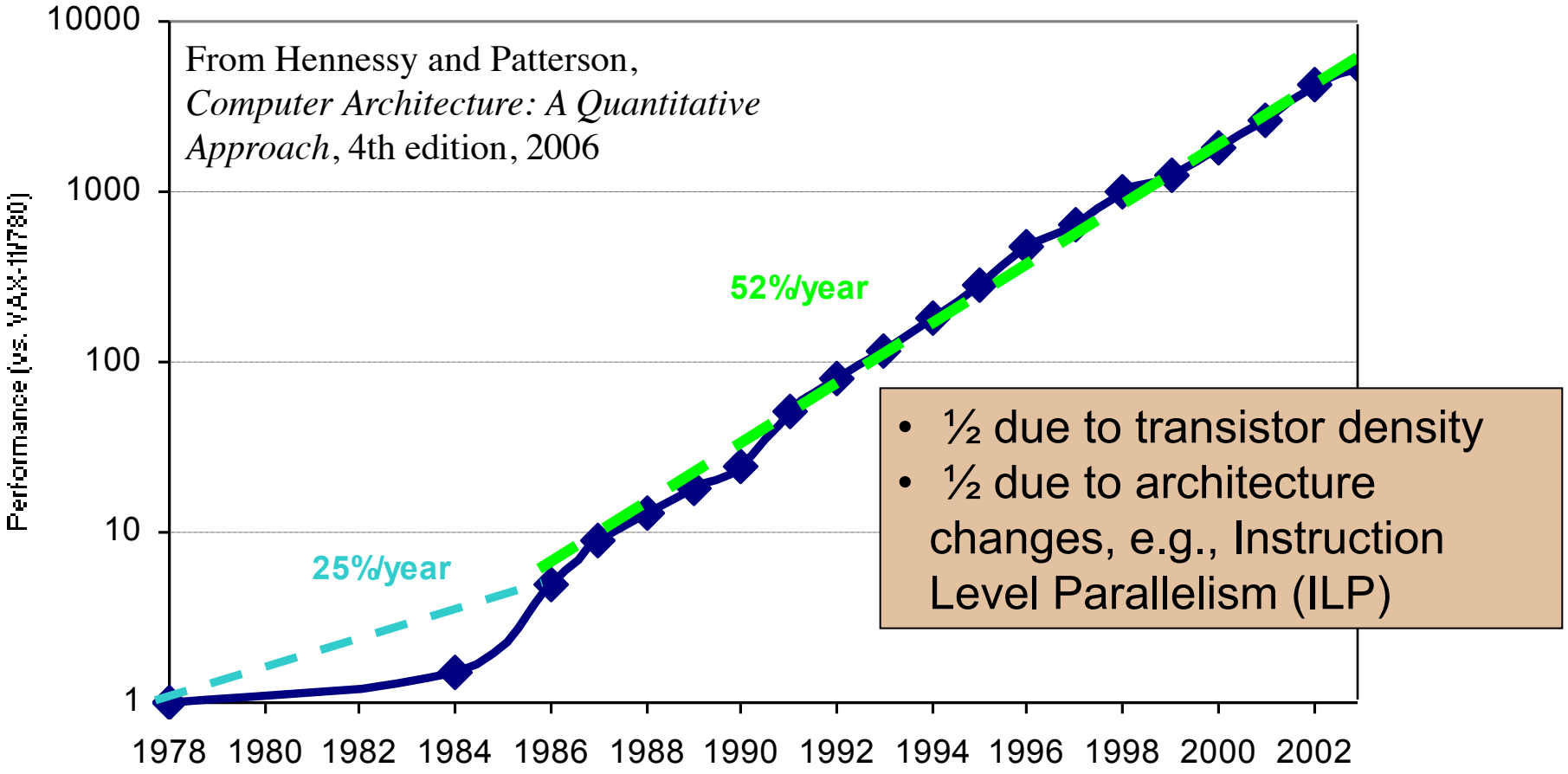
- Using additional cores
  - Increase density (= more transistors = more capacitance)
  - Can increase cores (2x) and performance (2x)
  - Or increase cores (2x), but decrease frequency (1/2): same performance at ¼ the power

- Additional benefits
  - Small/simple cores → more predictable performance

Source: Adapted from Katherine Yelick's lecture slide for CS194

Limit #2: Hidden Parallelism Tapped Out

Application performance was increasing by 52% per year as measured by the SpecInt benchmarks here



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002

Source: Adapted from Katherine Yelick's lecture slide for CS194

Limit #2: Hidden Parallelism Tapped Out

- **Superscalar (SS) designs were the state of the art; many forms of parallelism not visible to programmer**
  - **multiple instruction issue**
  - **dynamic scheduling: hardware discovers parallelism between instructions**
  - **speculative execution: look past predicted branches**
  - **non-blocking caches: multiple outstanding memory ops**

Source: Adapted from Katherine Yelick's lecture slide for CS194

Performance Comparison

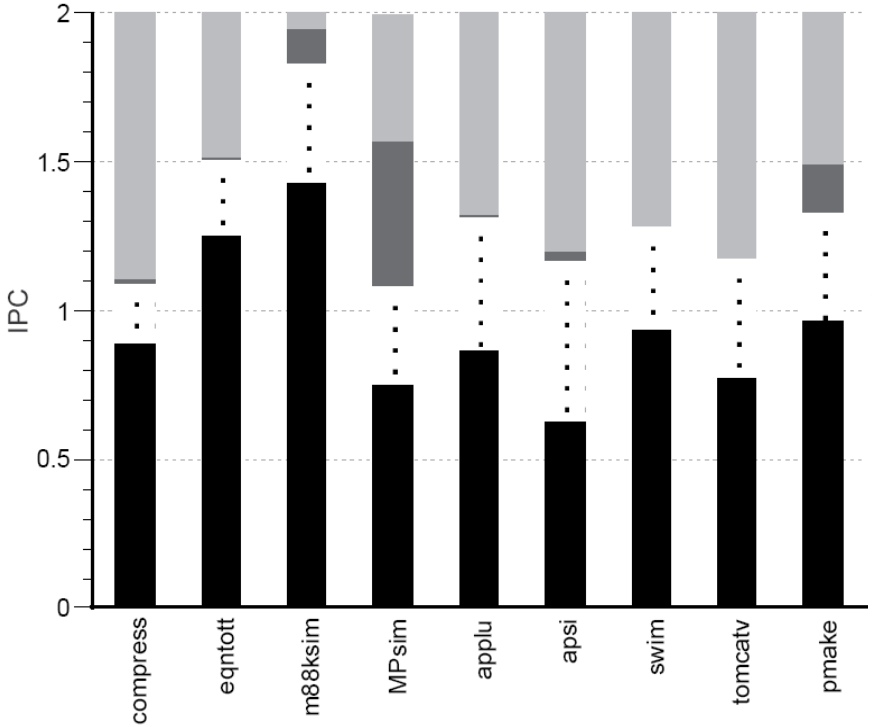


Figure 4. IPC Breakdown for a single 2-issue

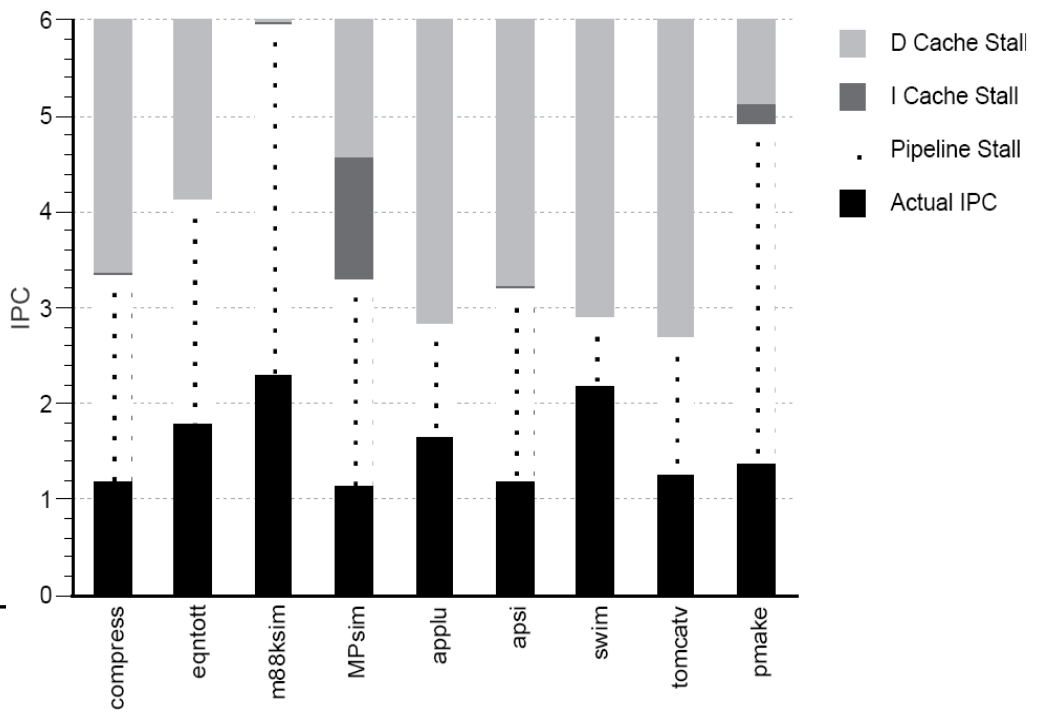


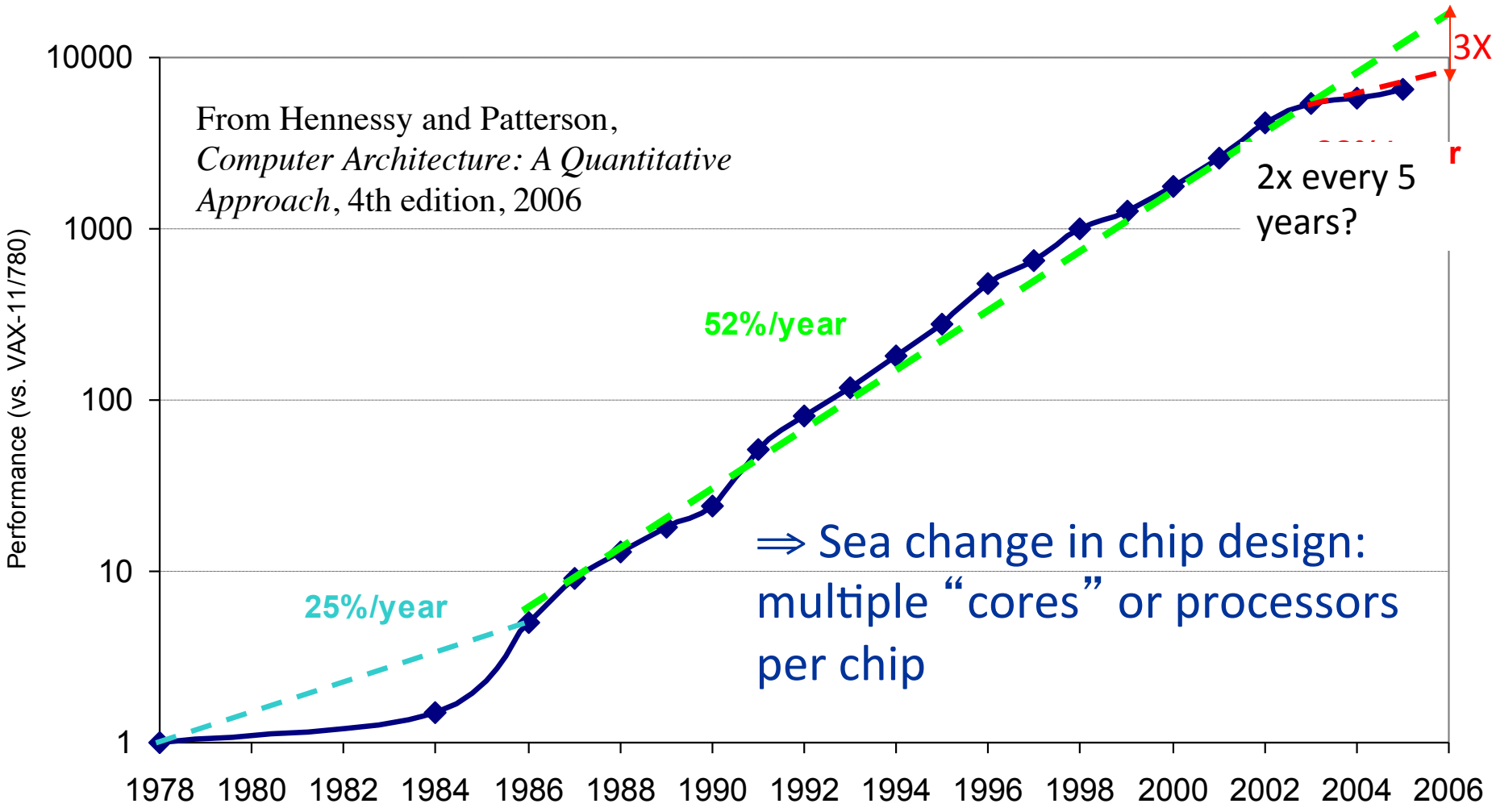
Figure 5. IPC Breakdown for the 6-issue processor.

- Measure of success for hidden parallelism is Instructions Per Cycle (IPC)
- The 6-issue has higher IPC than 2-issue, but far less than 3x
- Reasons are: waiting for memory (D and I-cache stalls) and dependencies (pipeline stalls)

Graphs from: Olukotun et al, ASPLOS, 1996

Source: Adapted from Katherine Yelick's lecture slide for CS194

Uniprocessor Performance (SPECint) Today



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Source: Adapted from Katherine Yelick's lecture slide for CS194

Limit #3: Chip Yield

Manufacturing costs and yield problems limit use of density

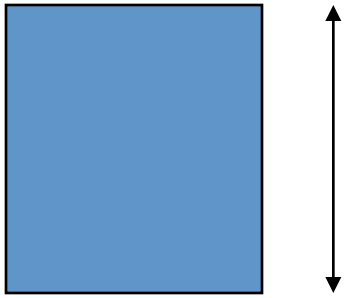
- Moore's (Rock's) 2nd law: fabrication costs go up
- Yield (% usable chips) drops

- Parallelism can help
  - More smaller, simpler processors are easier to design and validate
  - Can use partially working chips:
  - E.g., Cell processor (PS3) is sold with 7 out of 8 "on" to improve yield

Source: Adapted from Katherine Yelick's lecture slide for CS194

Limit #4: Speed of Light (Fundamental)

1 Tflop/s, 1 Tbyte sequential machine



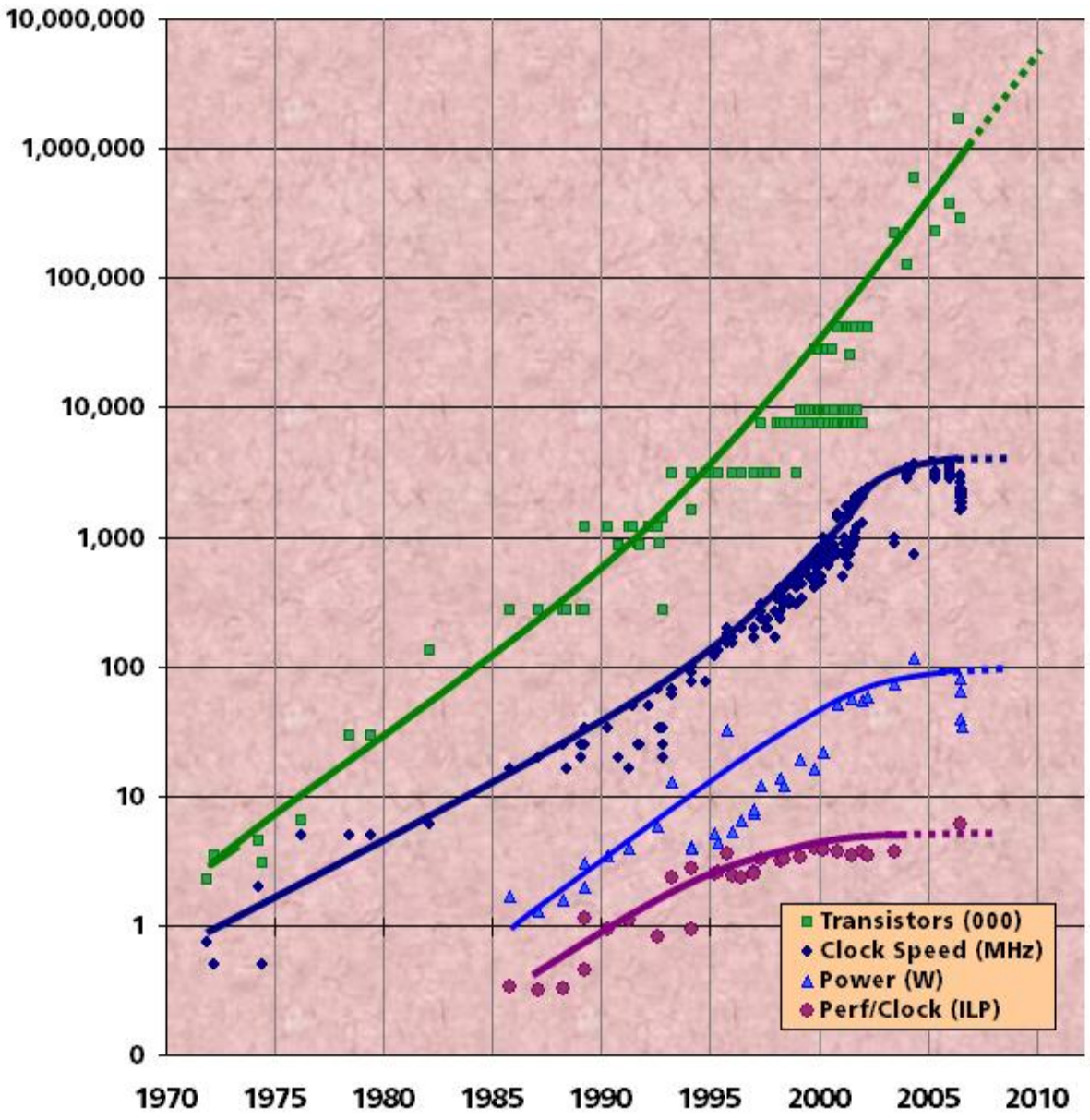
r = 0.3 mm

- Consider the 1 Tflop/s sequential machine:
  - Data must travel some distance, r, to get from memory to CPU.
  - To get 1 data element per cycle, this means 10^{12} times per second at the speed of light, c = $3x10^8$ m/s. Thus r < c/ 10^{12} = 0.3 mm.
- Now put 1 Tbyte of storage in a 0.3 mm x 0.3 mm area:
  - Each bit occupies about 1 square Angstrom, or the size of a small atom.
- No choice but parallelism

Source: Adapted from Katherine Yelick's lecture slide for CS194

Revolution is Happening Now

- Chip density is continuing increase ~2x every 2 years
  - Clock speed is not
  - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software



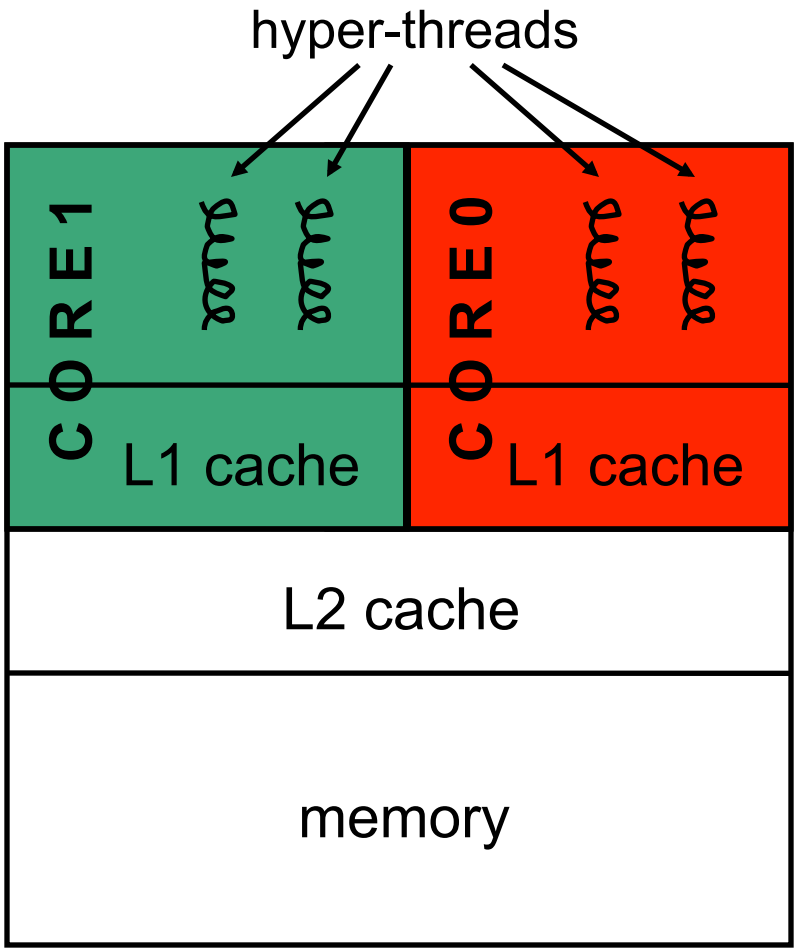
Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)

Why multi-core ?
Summary

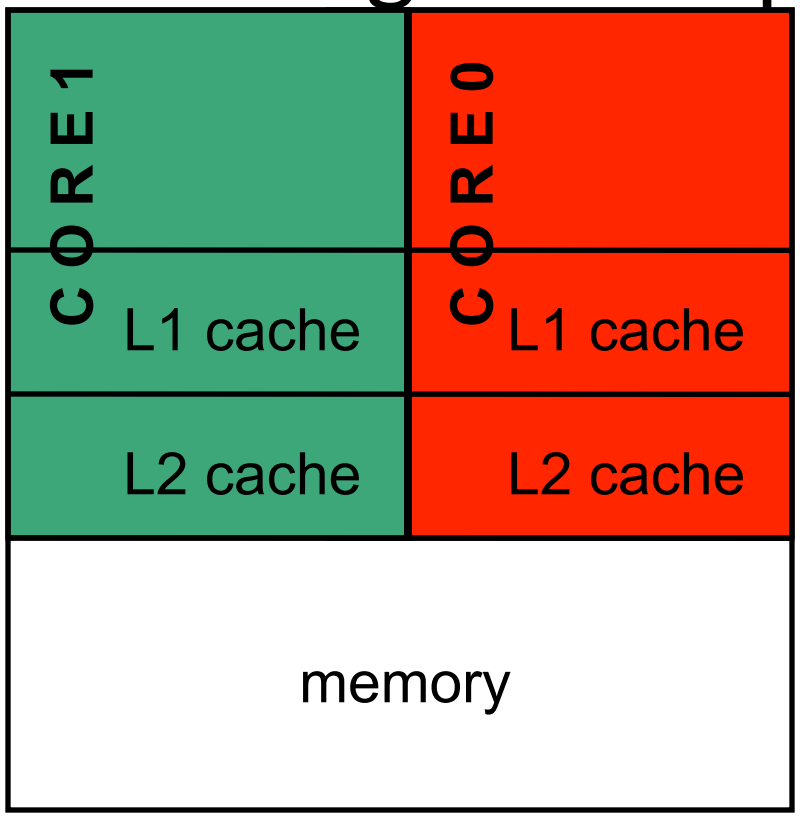
- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning
- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

Intel Xeon Dual-core

- Dual-core
  Intel Xeon processors
- Each core is
  hyper-threaded
- Private L1 caches
- Shared L2 caches

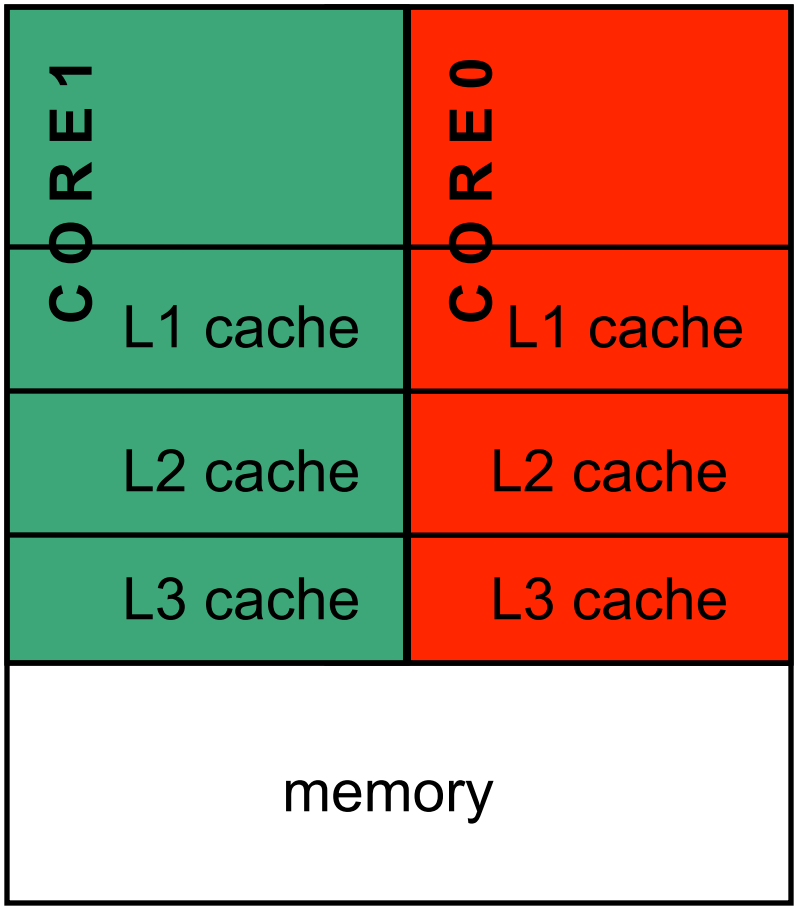


Designs with private L2 caches



Both L1 and L2 are private

Examples: AMD Opteron,
AMD Athlon, Intel Pentium D



A design with L3 caches

Example: Intel Itanium 2

Multi-core Benefits

1. Gain in performance without increasing the original power consumption
2. They improve an operating system's ability to multitask applications
3. Another benefit comes from individual applications optimized for multi-core processors

Private vs shared caches

- Advantages of private:
  - They are closer to core, so faster access
  - Reduces contention
- Advantages of shared:
  - Threads on different cores can share the same cache data
  - More cache space available if a single (or a few) high-performance thread runs on the system

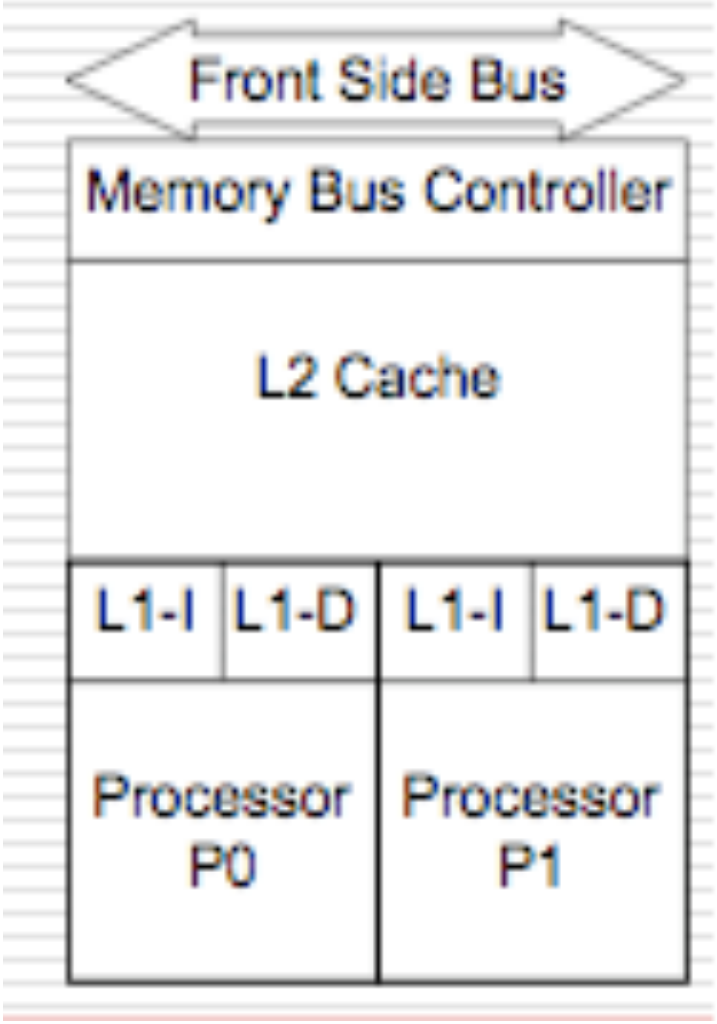
Slides from Stanford University

Refer to slide number 13 to 20 of the presentation for an illustration of cache coherence problem and some solutions (MSI, MESI, and MOESI cache coherence protocols).

http://www.csit-sun.pub.ro/~cpop/Sisteme_cu_Microprocesoare_Avansate_SMPA/SMPA_curs_master5AAC/SMPA_curs6/EE382A/L13-cmp.pdf

Intel Core2Duo

- Two 32-bit Pentiums on chip
- Private 32K L1 caches for instructions (L1-I) and data
- Shared 2M-4M L2 cache
- MESI cc-protocol provides cache coherency
- Shared bus control and memory
- Fast on-chip communication using shared memory

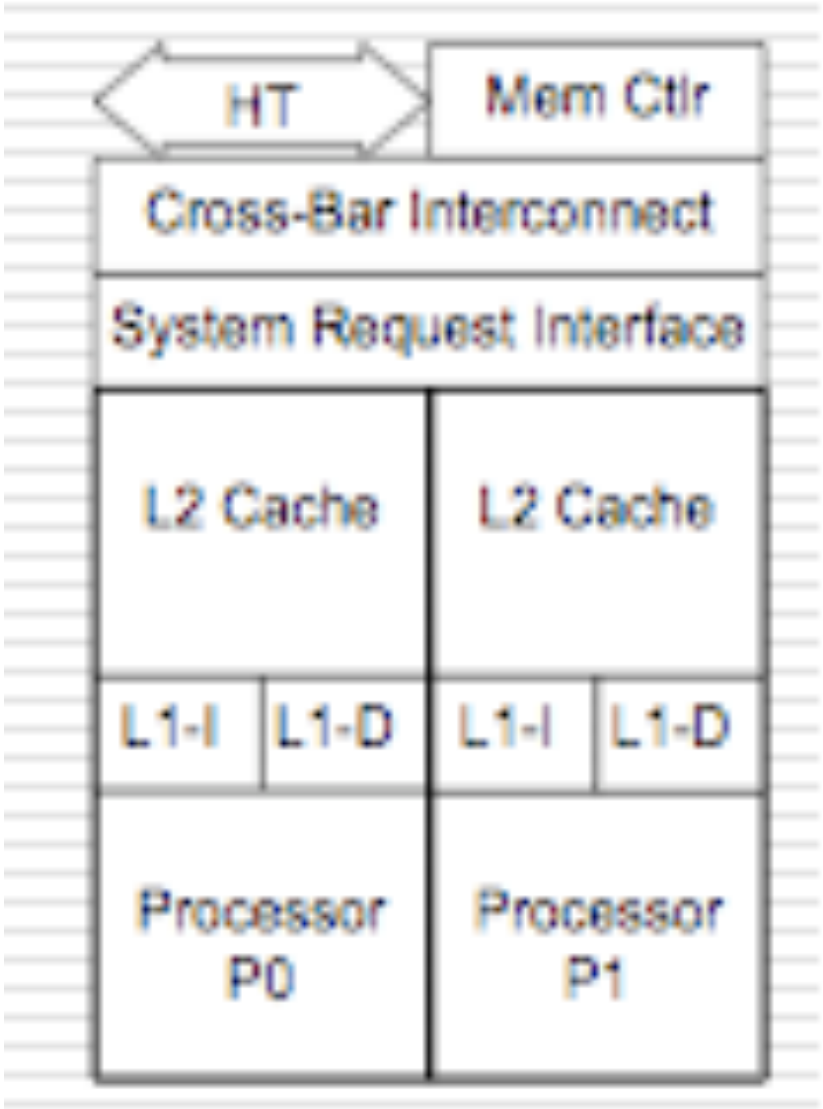


MESI Cache Coherence

- States: M=modified; E=exclusive; S=shared; I=invalid;
- Think of it as a finite state machine
  - Upon loading, a line is marked E, subsequent reads are OK; write marks M
  - Seeing another load, mark as S
  - A write to an S, sends I to all, marks as M
  - Another's read to an M line, writes it back, marks it S
- Read/write to a cache line in I state is a cache miss

AMD Dual-Core Opteron (in CHPC systems)

- Two 64-bit processors on a chip
- 64K private L1 data & instruction caches
- 1 MB private L2 cache
- MOESI cc-protocol
- Direct connect shared memory
- Fast on-chip communication between processors



More General Information on MESI and MOESI Protocols

Here is a free and quickly accessible reference of MSI, MESI, and MOESI protocols, respectively,

http://en.wikipedia.org/wiki/MSI_protocol

http://en.wikipedia.org/wiki/MESI_protocol

http://en.wikipedia.org/wiki/MOESI_protocol

Caution: content on the Internet may not always be reliable or extremely accurate. However it might serve as a quick and easily accessible reference to get a general idea of the concepts. Further reading will need to be done using proper text books.

Why does a Computational Scientist Need to be Aware of Cache Coherence?

Because writing code that invokes tasks to maintain cache coherence could affect the performance of the code. This could be avoided sometimes.

A simple example is that of 'false sharing' presented in page 31 of your textbook. Consider a declaration -

Double x, y

x and y will likely be allocated consecutive memory locations.

Now if core 1 uses x and the core 2 uses the variable y and they keep updating it. Due to the cache coherence protocol, the cache line will continuously be moved between the cores.

Note: this is an oversimplified example and there are ways of fixing it if one is aware of the problem.