High-Performance Parallel Computing

P. (Saday) Sadayappan

Rupesh Nasre

Course Overview

- Emphasis on algorithm development and programming issues for high performance
- No assumed background in computer architecture; assume knowledge of C
- Grading:
  - 60% Programming Assignments (4 x 15%)
  - 40% Final Exam (July 4)
- Accounts will be provided on IIT-M system

Course Topics

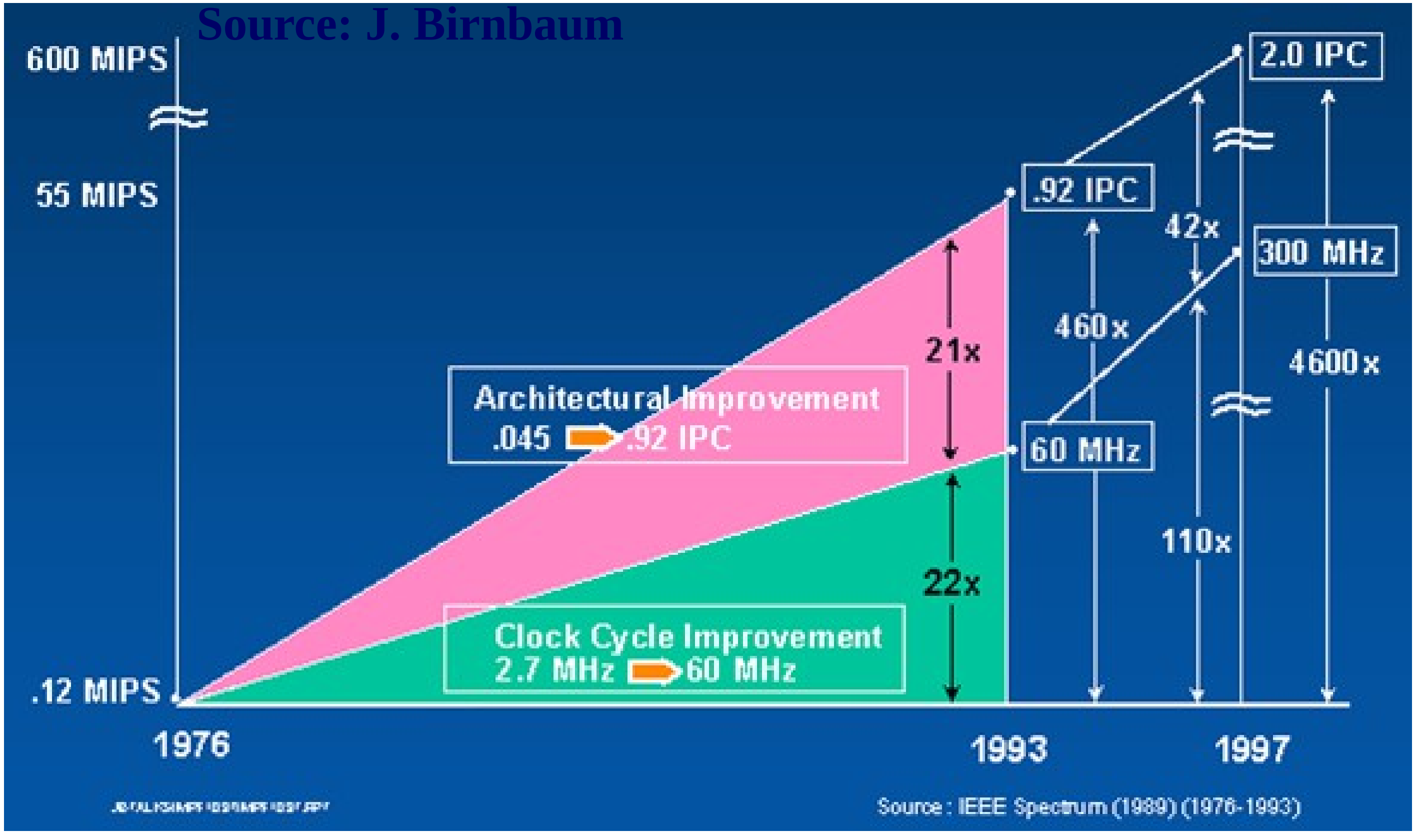
- **Architectures**
  - Single processor core
  - Multi-core and SMP (Symmetric Multi-Processor) systems
  - GPUs (Graphic Processing Units)
  - Short-vector SIMD instruction set architectures
- **Programming Models/Techniques**
  - Single processor performance issues
    - Caches and Data locality
    - Data dependence and fine-grained parallelism
    - Vectorization (SSE, AVX)
  - Parallel programming models
    - Shared-Memory Parallel Programming (OpenMP)
    - GPU Programming (CUDA)

Class Meeting Schedule

- Lecture from 9am-12pm each day, with mid-class break
- Optional Lab session from 12-1pm each day
- 4 Programming Assignments

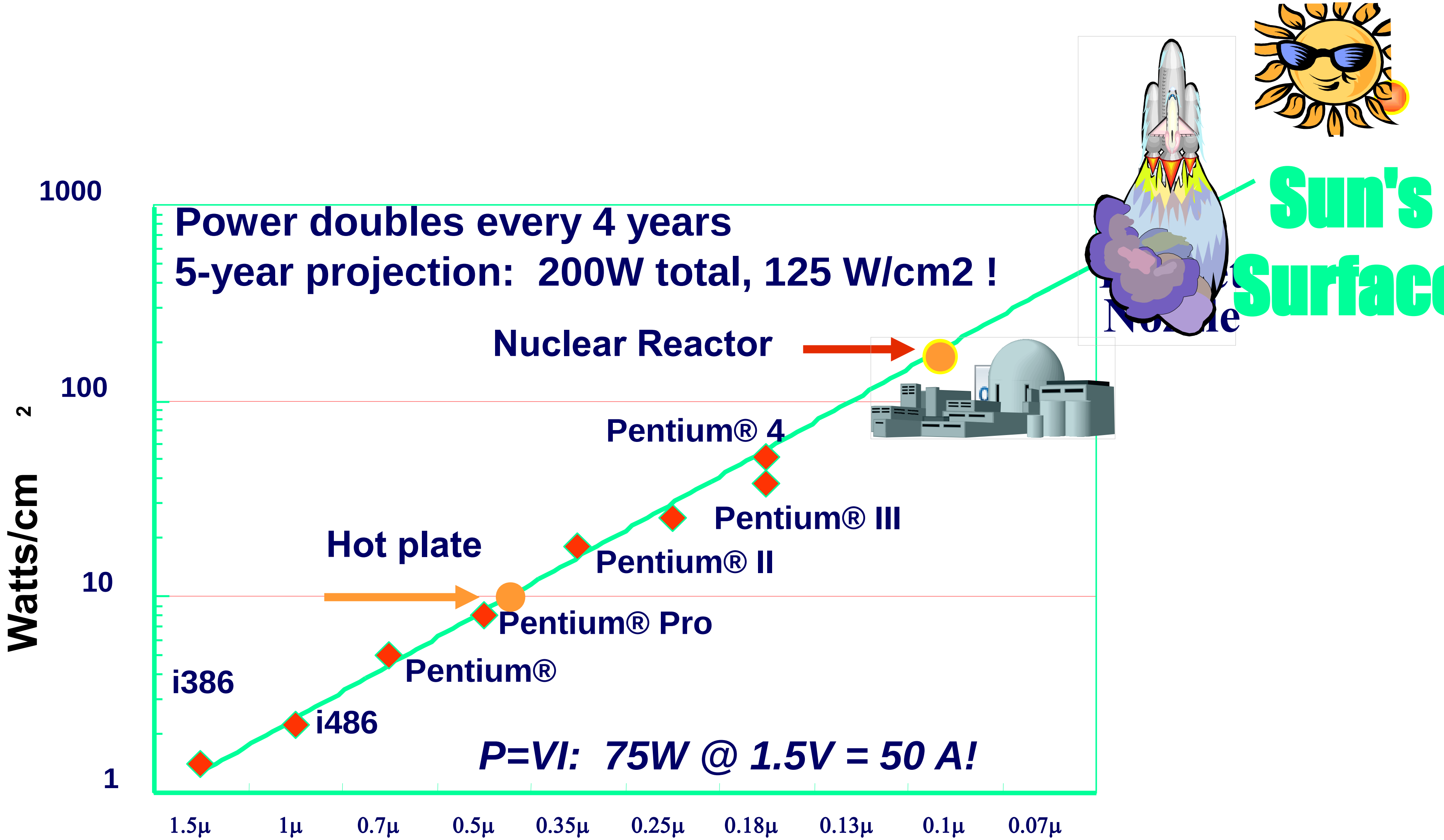
Data Locality	June 24	15%
OpenMP	June 27	15%
CUDA	June 30	15%
Vectorization	July 2	15%

The Good Old Days for Software



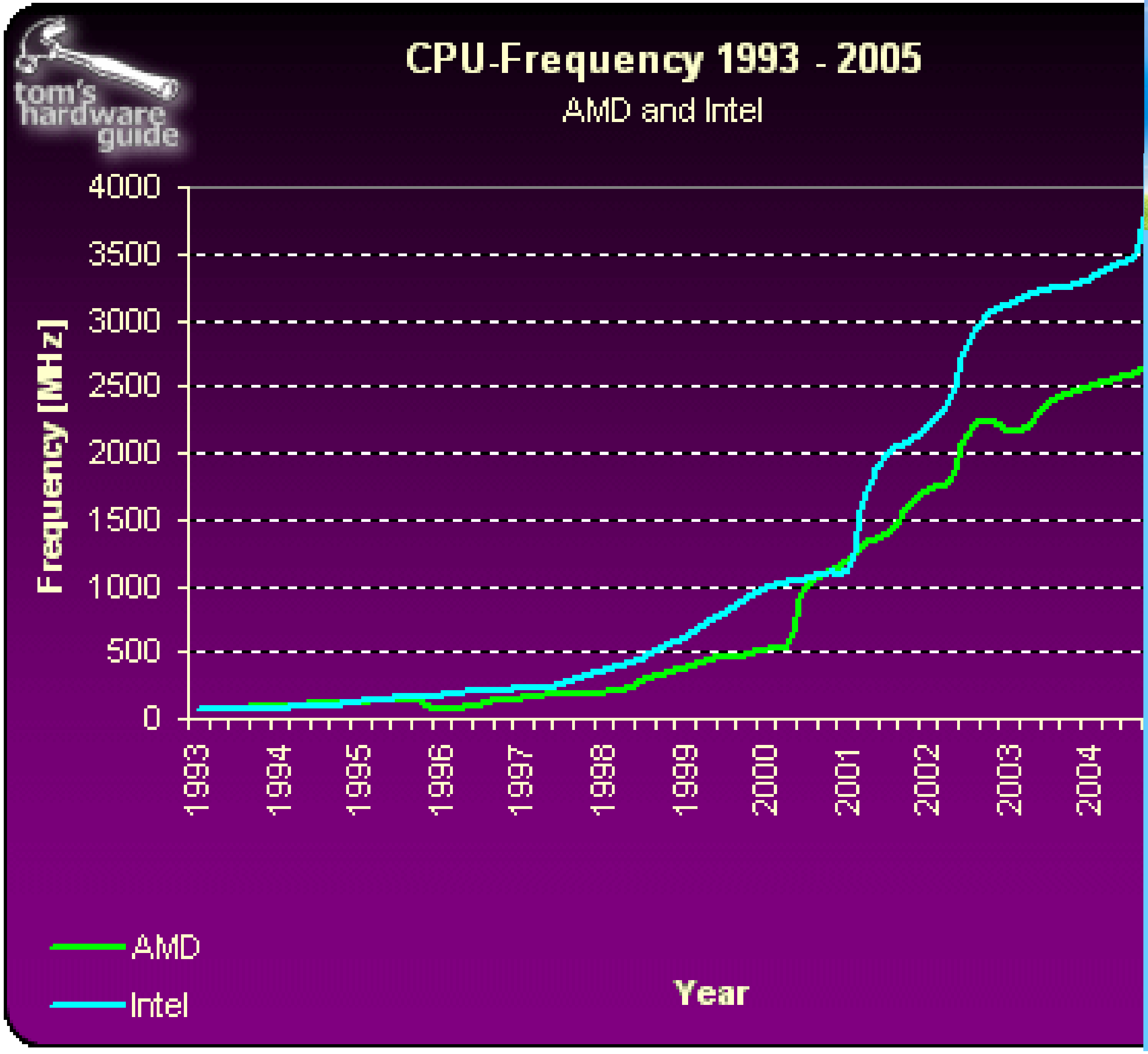
- Single-processor performance experienced dramatic improvements from <u>clock</u>, and <u>architectural</u> improvement (Pipelining, Instruction-Level-Parallelism)
- Applications experienced automatic performance

Hitting the Power Wall



* “New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies” – Fred Pollack, Intel Corp. Micro32 conference key note - 1999. Courtesy Avi Mendelson, Intel

Hitting the Power Wall



toward

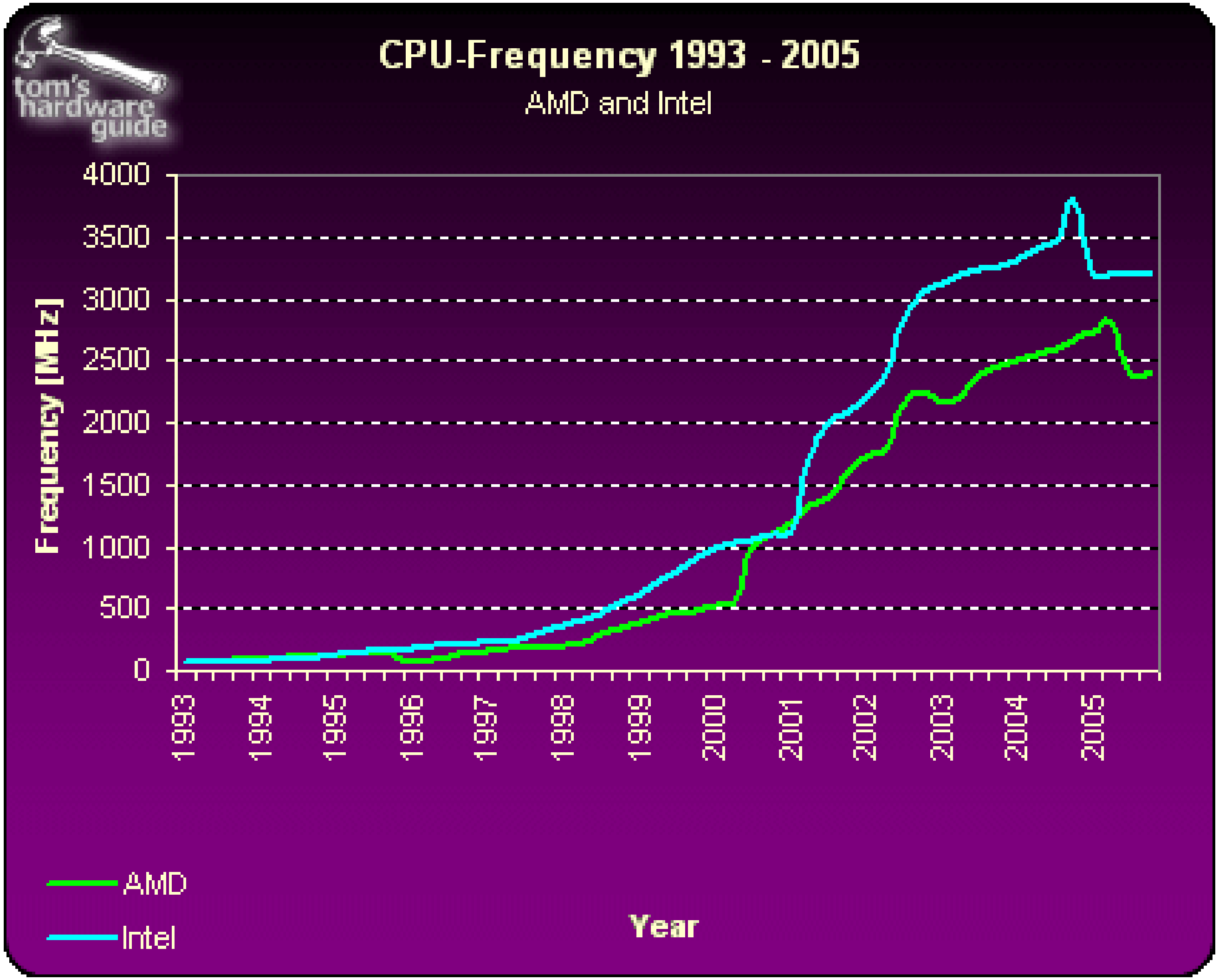
a

brighter

tomorrow

http://img.tomshardware.com/us/2005/11/21/the_mother_of_all_cpu_charts_2005/cpu_frequency.gif

Hitting the Power Wall



http://img.tomshardware.com/us/2005/11/21/the_mother_of_all_cpu_charts_2005/cpu_frequency.gif

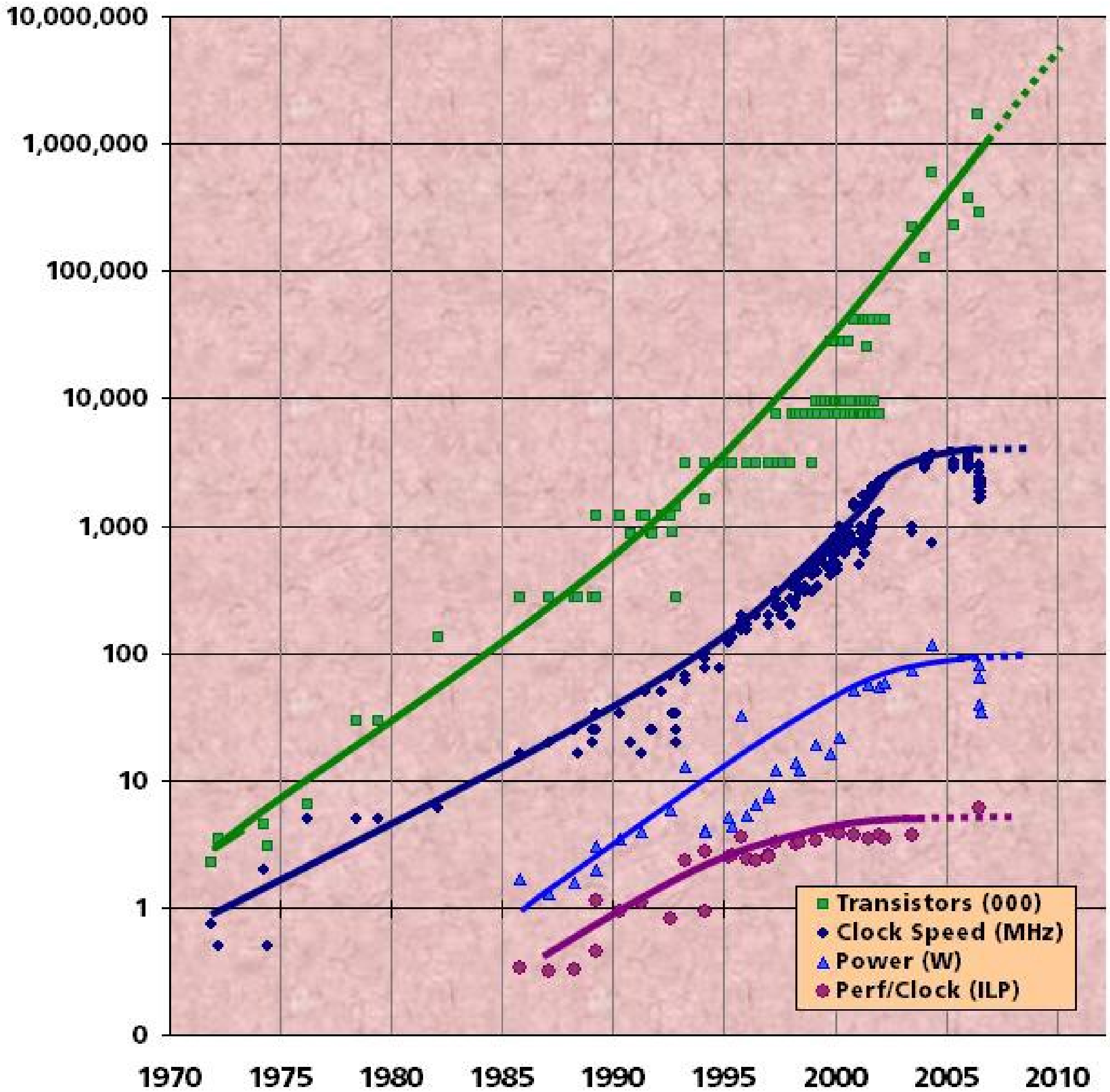
2004 – Intel cancels Tejas and Jayhawk due to "heat problems due to the extreme power consumption of the core ..."

The Only Option: Use Many Cores

Chip density is continuing increase ~2x every 2 years
- Clock speed is not
- Number of processor cores may double

There is little or no more hidden parallelism (ILP) to be found

Parallelism must be exposed to and managed by software



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

Can You Predict Performance?

- E1 takes about 0.4s to execute on this laptop
  - Performance in GFLOPs: billions (Giga) of FLoating point Operations per Second =
- About how long does E2 take?
  1. [0-0.4s]
  2. [0.4s-0.6s]
  3. [0.6s-0.8s]
  4. More than 0.8s

E2 takes 0.35s => 2.27 GFLOPs

```
double W,X,Y,Z;

for(j=0;j<100000000;j++){
  W = 0.999999*X;
  X = 0.999999*W;}
// Example loop E1

for(j=0;j<100000000;j++){
W = 0.999999*W + 0.000001;
X = 0.999999*X + 0.000001;
Y = 0.999999*Y + 0.000001;
Z = 0.999999*Z + 0.000001;
}
// Example loop E2
```

ILP Affects Performance

- **ILP (**Instruction Level Parallelism): Many operations in a sequential code could be executed concurrently if they do not have dependences
- Pipelined stages in functional units can be exploited by ILP
- Multiple functional units in a CPU be exploited by ILP
- ILP is automatically exploited by the system when possible
- E2's statements are independent and provide ILP, but E1's statements are not, and do not provide ILP

```
double W,X,Y,Z;

for(j=0;j<100000000;j++){
  W = 0.999999*X;
  X = 0.999999*W;}
// Example loop E1

for(j=0;j<100000000;j++){
W = 0.999999*W + 0.000001;
X = 0.999999*X + 0.000001;
Y = 0.999999*Y + 0.000001;
Z = 0.999999*Z + 0.000001;
}
// Example loop E2
```

Example: Memory Access Cost

```
#define N 32
#define T 1024*1024
// #define N 4096
// #define T 64
double A[N][N];

for(it=0; it<T; it++)
  for (j=0; i<N; i++)
    for (i=0; i<N; i++)
      A[i][j] += 1;
```

- About how long will code run for the 4Kx4K matrix?
  1. [0-0.3s]
  2. [0.3s-0.6s]
  3. [0.6s-1.2s]
  4. More than 1.2s

	32x32	4Kx4K
FLOPs	230	230
Time	0.33s	15.5s
GFLOPs	3.24	0.07

Data Movement overheads from main memory to cache can greatly exceed computational costs

Cache Memories

Adapted from slides
by Prof. Randy Bryant (CMU CS15-213)
and Dr. Jeffrey Jones (OSU CSE 5441)

Topics

- Generic cache memory organization
- Impact of caches on performance

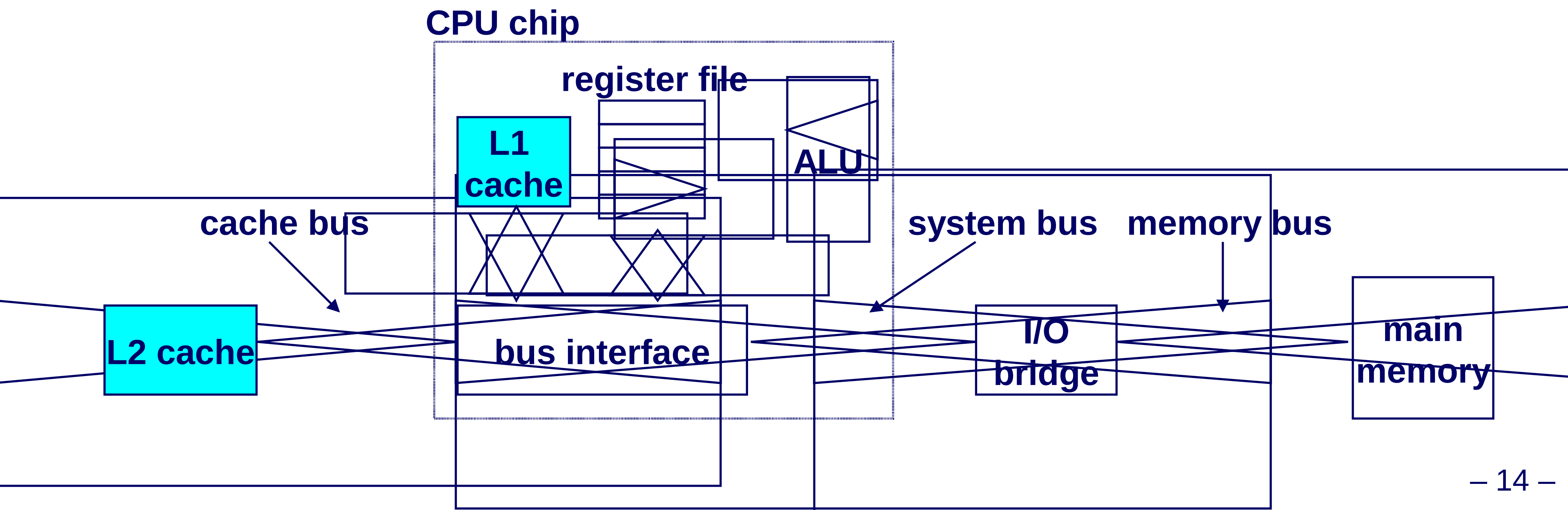
Cache Memories

Cache memories are small, fast SRAM-based memories managed automatically in hardware.

- Hold frequently accessed blocks of main memory

CPU looks first for data in L1, then in L2, then in main memory.

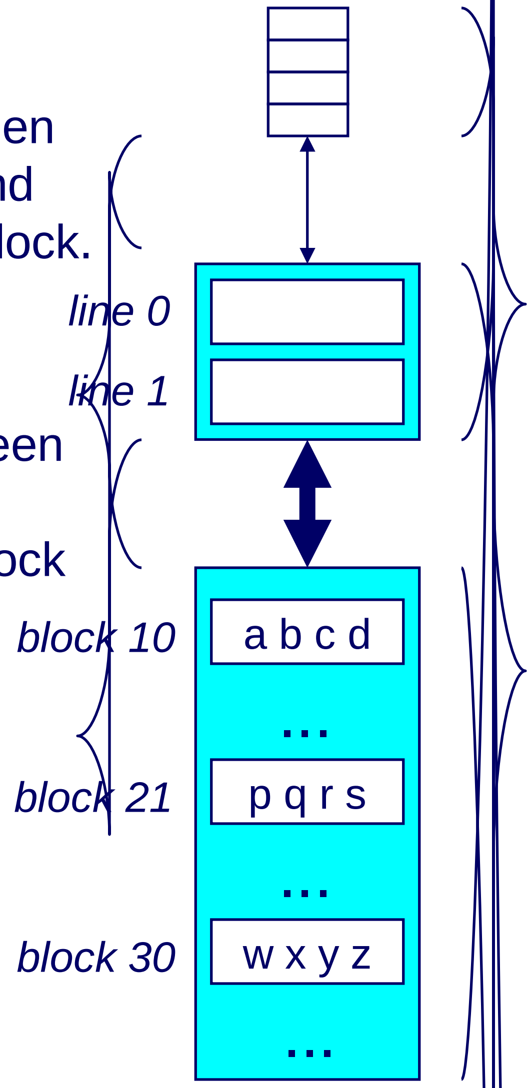
Typical bus structure:



Inserting an L1 Cache Between the CPU and Main Memory

The transfer unit between the CPU register file and the cache is a 4-byte block.

The transfer unit between the cache and main memory is a 4-word block (16 bytes).

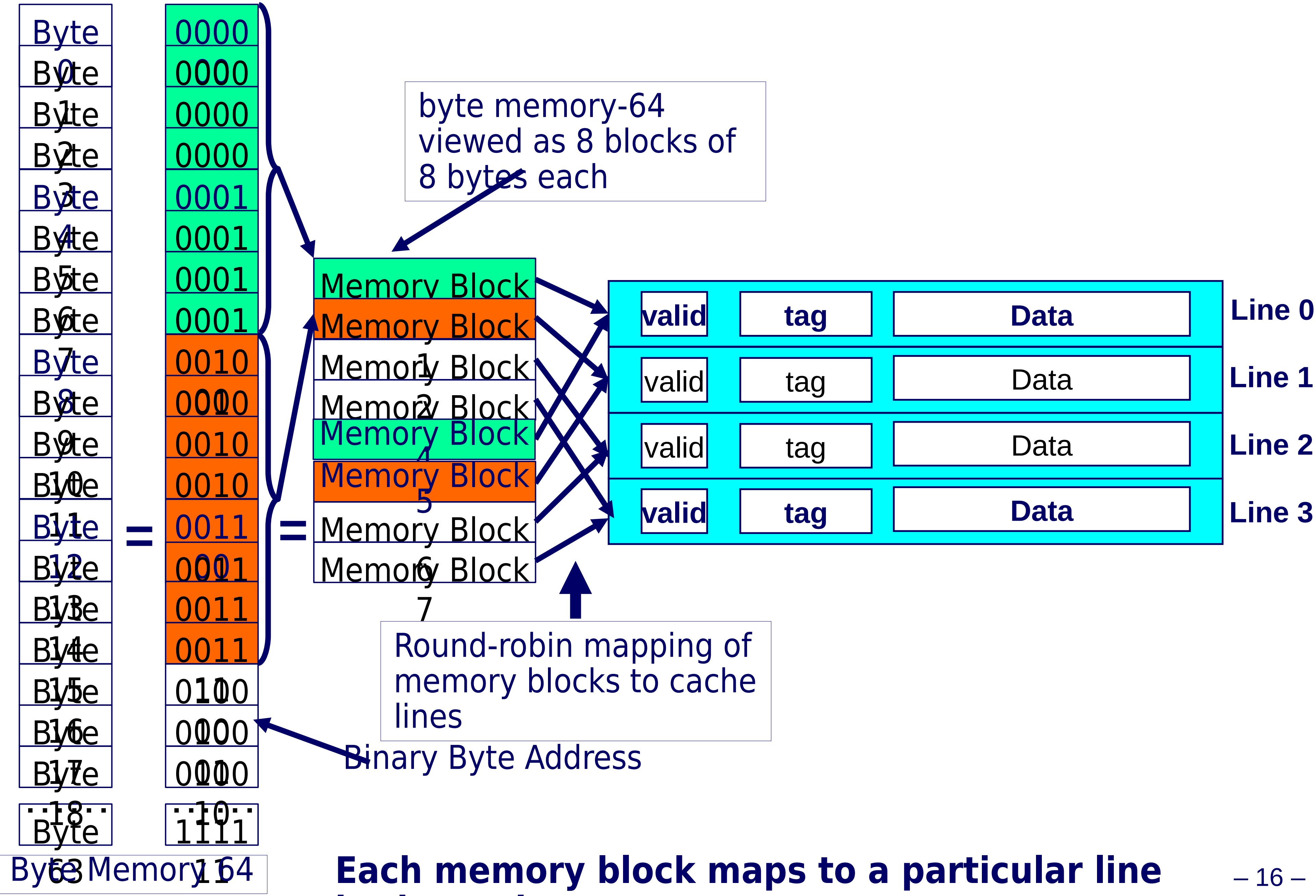


The tiny, very fast CPU **register file** has room for four 4-byte words.

The small fast **L1 cache** has room for two 4-word blocks.

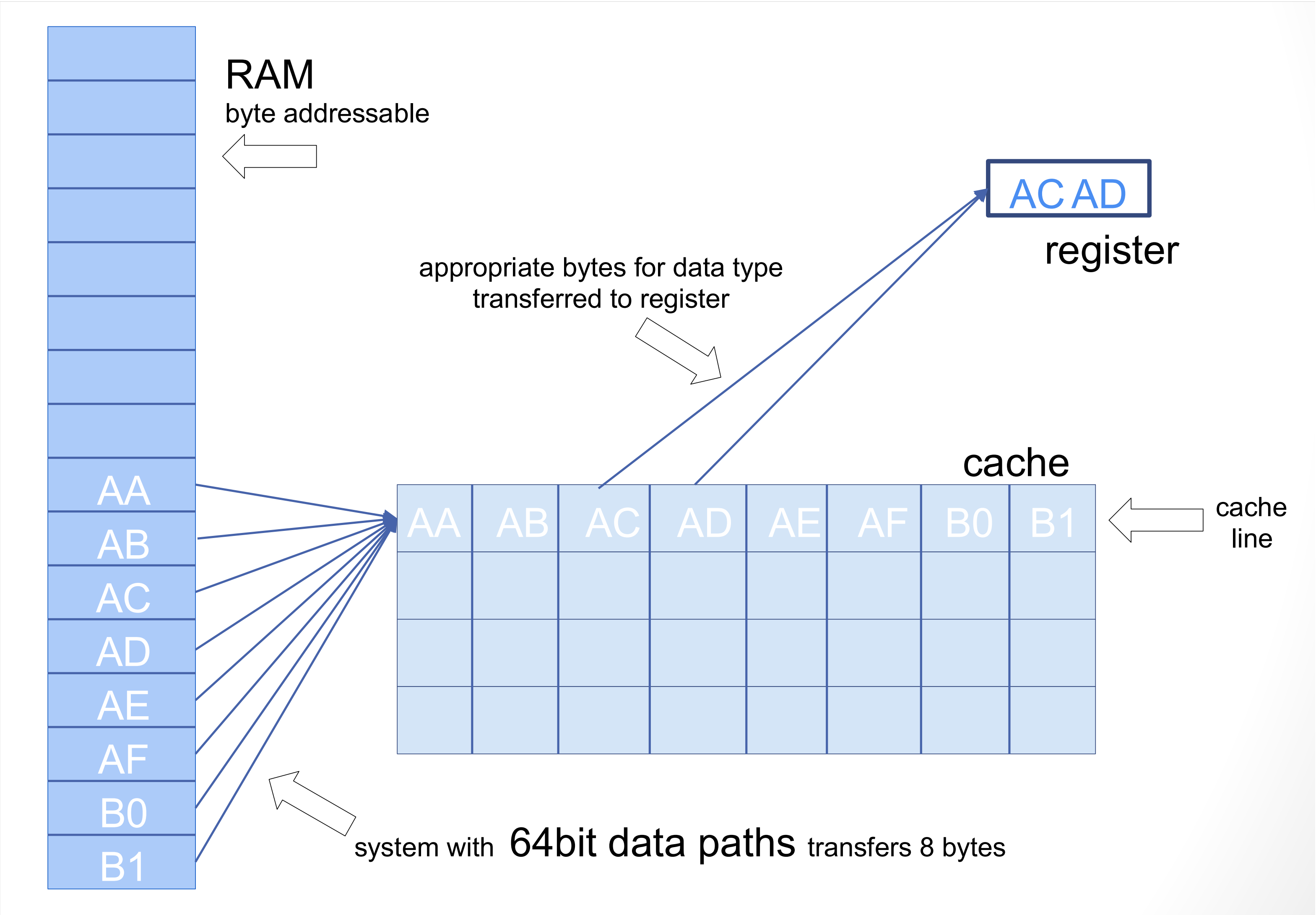
The big slow **main memory** has room for many 4-word blocks.

Direct-Mapped Cache



Each memory block maps to a particular line in the cache

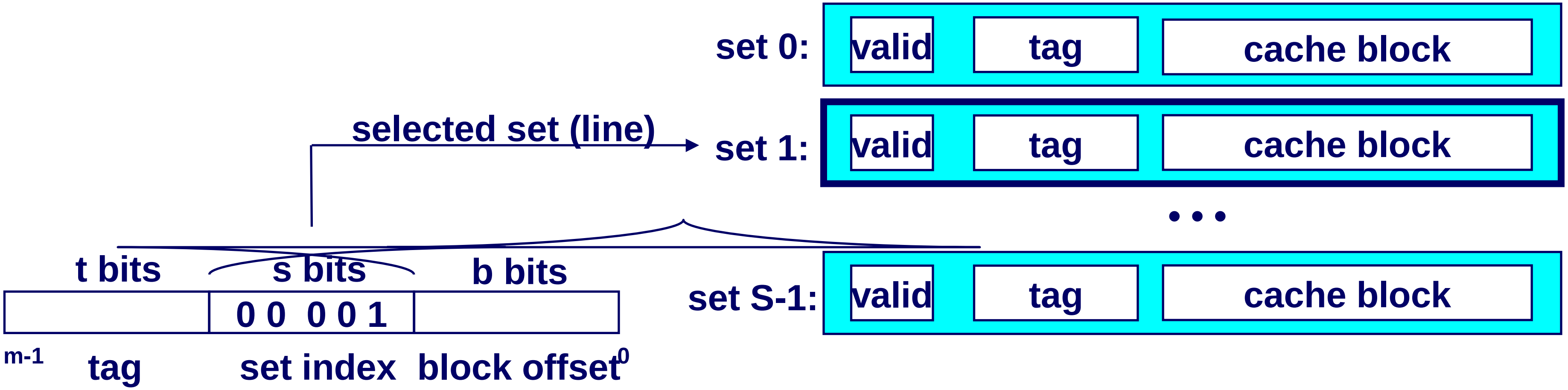
Direct-Mapped Cache



Accessing Direct-Mapped Caches

Set selection

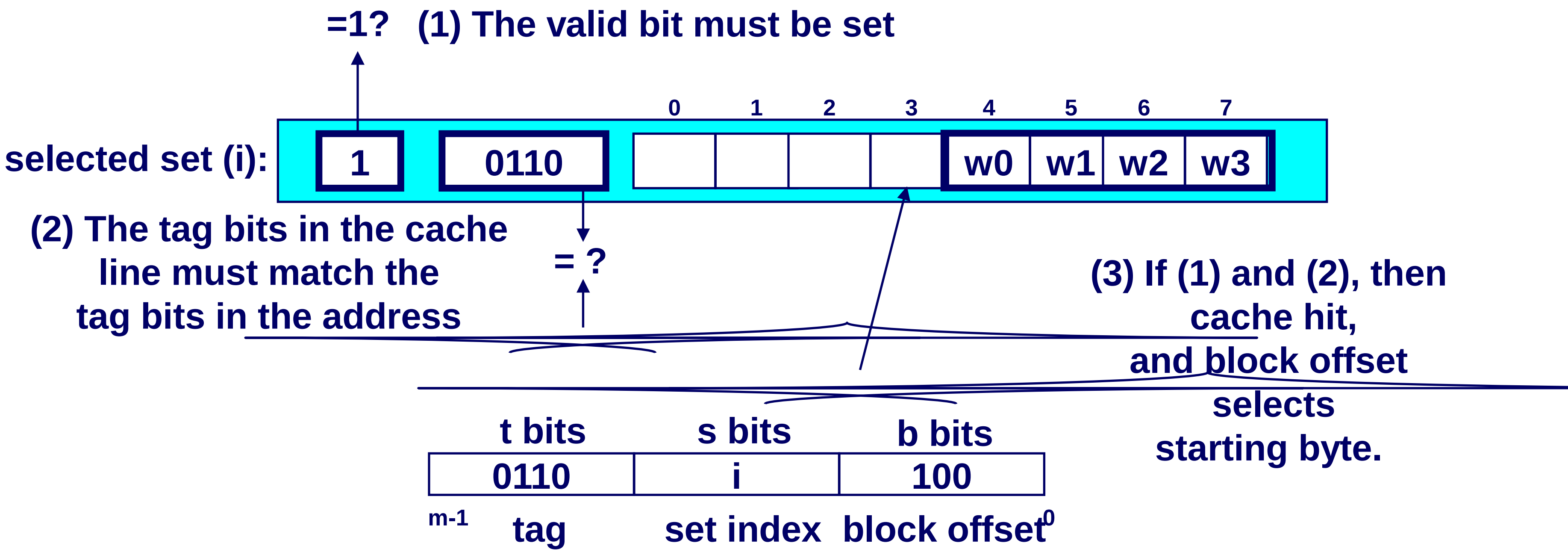
- Use the set (= line) index bits to determine the set of interest.



Accessing Direct-Mapped Caches

Line matching and word selection

- Line matching: Find a valid line in the selected set with a matching tag
- Word selection: Then extract the word



Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

M=16 byte addresses, B=2 bytes/block,
S=4 sets, E=1 entry/set

Address trace (reads):
0 [00002], 1 [00012], 13 [11012], 8 [10002], 0 [00002]

(1) 0 [00002] *(miss)*

v	tag	data
1	0	M[0-1]

(3) 13 [11012] *(miss)*

v	tag	data
1	0	M[0-1]
1	1	M[12-13]

(4) 8 [10002] *(miss)*

v	tag	data
1	1	M[8-9]
1	1	M[12-13]

(5) 0 [00002] *(miss)*

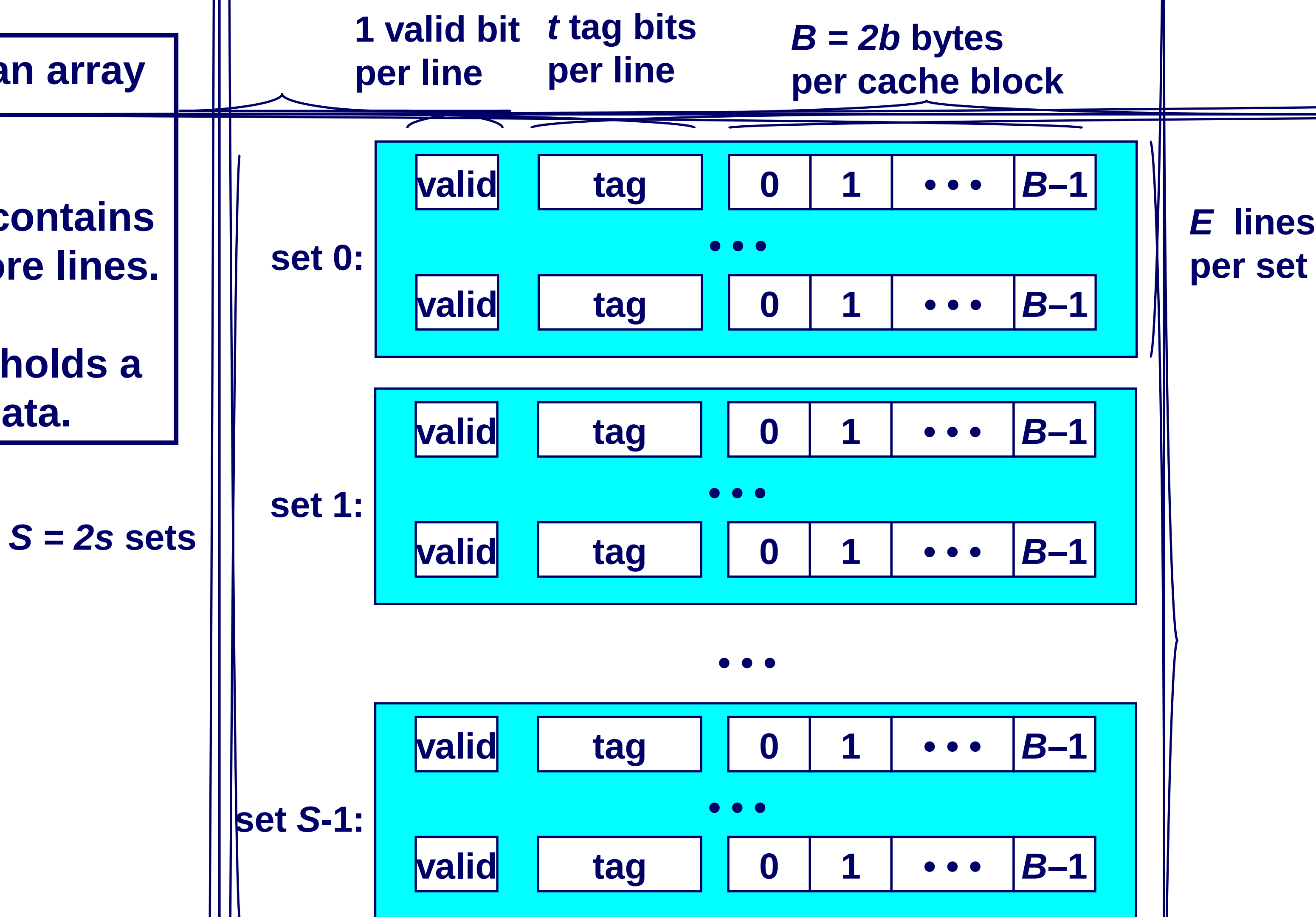
v	tag	data
1	0	M[0-1]
1	1	M[12-13]

General Structure of Cache Memory

Cache is an array of sets.

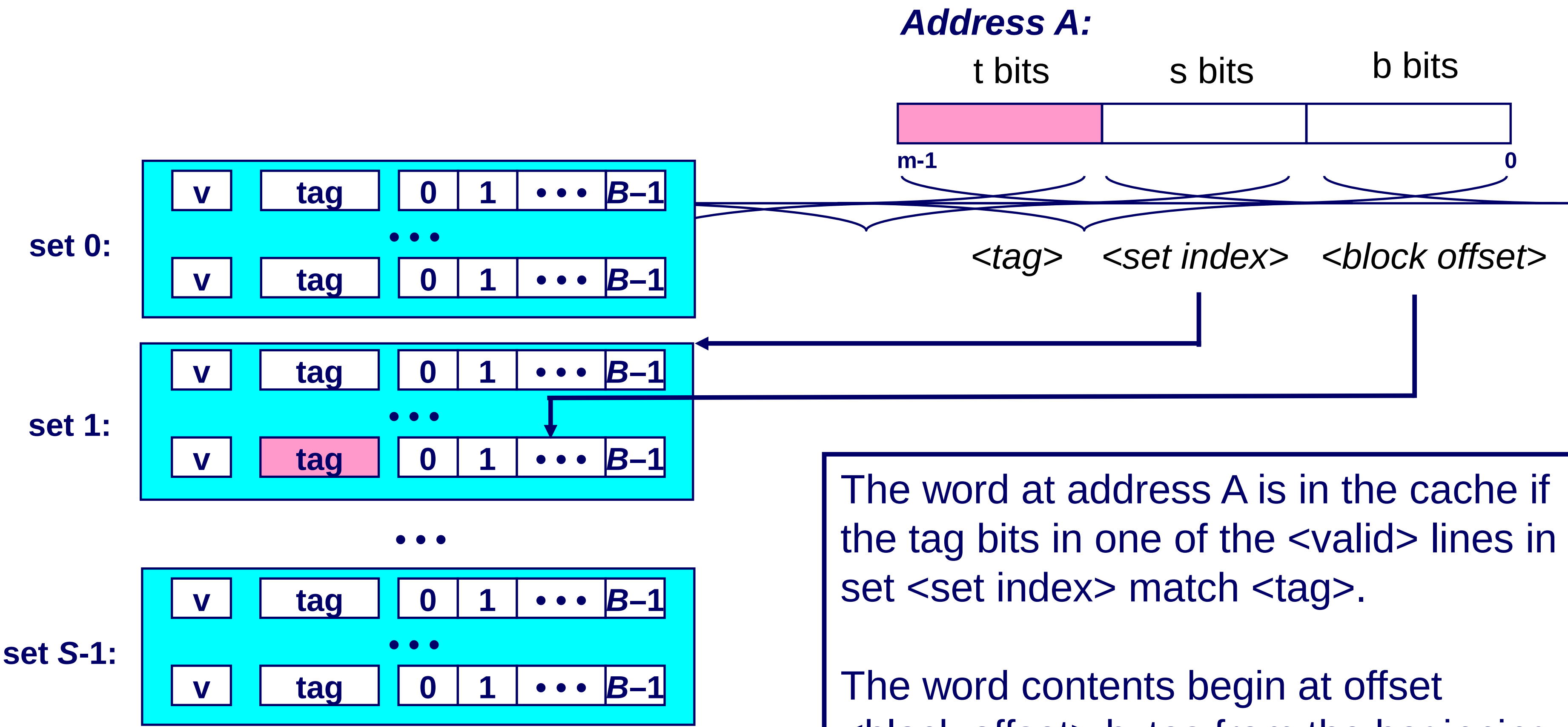
Each set contains one or more lines.

Each line holds a block of data.



Cache size: $C = B \times E \times S$ data bytes

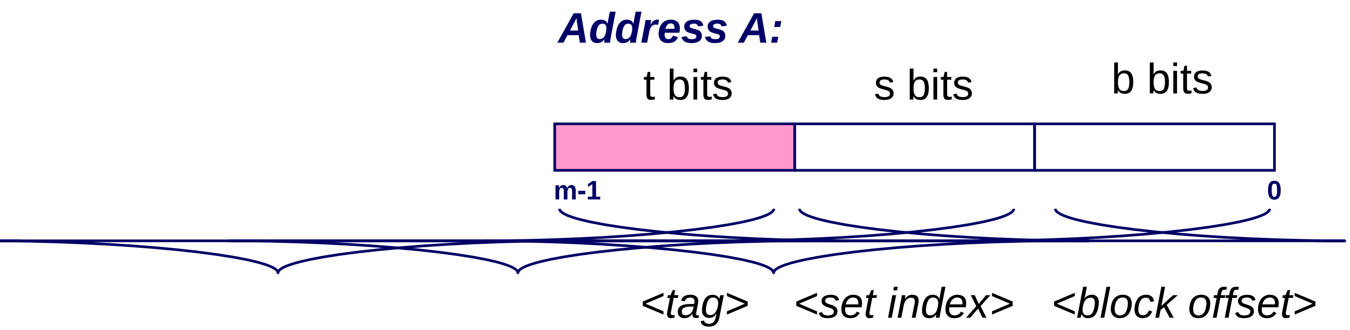
Addressing Caches



The word at address A is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>.

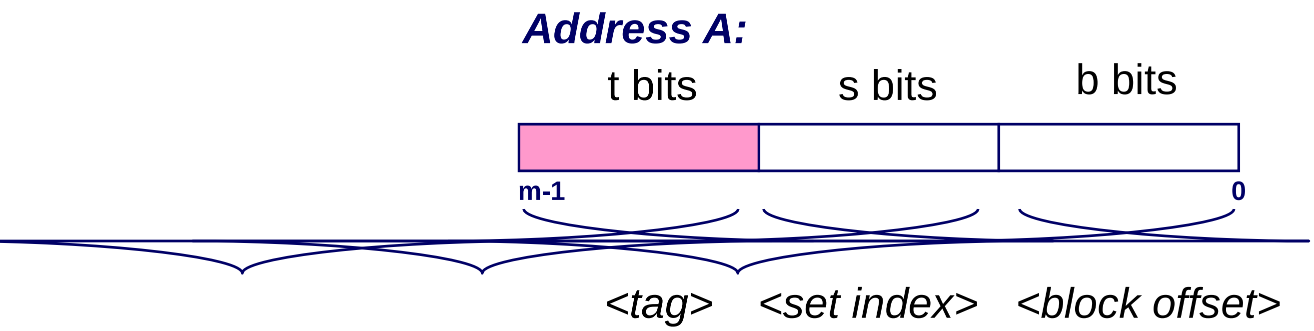
The word contents begin at offset <block offset> bytes from the beginning of the block.

Cache Mapping Examples



Example	m	C	B	E
1	32	1024	4	1
2	32	1024	8	4
3	32	1024	32	32
4	64	2048	64	16

Cache Mapping Examples



Example	m	C	B	E	S	t	s	b
1	32	1024	4	1	256	22	8	2
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	27	0	5
4	64	2048	64	16	2	57	1	6

Cache Performance Metrics

Miss Rate

- Fraction of memory references not found in cache (misses/references)
- Typical numbers:
  - 3-10% for L1
  - can be quite small (e.g., < 1%) for L2, depending on size, etc.

Hit Time

- Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)
- Typical numbers:
  - 1 clock cycle for L1
  - 3-8 clock cycles for L2

Miss Penalty

- Additional time required because of a miss
  - Typically 25-100 cycles for main memory

overall cache performance

AMAT (Avg. Mem. Access Time) =
t_hit + prob_miss * penalty_miss

- reduce hit time
- reduce miss penalty
- reduce miss rate

Cache Performance: One Level

AMAT = t_hit + prob_miss * penalty_miss

- L1$ hits in 1 cycle, with 80% hit rate
- main memory hits in 1000 cycles

AMAT = 1 + (1-.8)(1000)
= 201

- L1$ hits in 1 cycle, with 85% hit rate
- main memory hits in 1000 cycles

AMAT = 1 + (1-.85)(1000)
= 151

Cache Performance: Multi-Level

AMATi = t_hiti + prob_missi * penalty_missi

- L1$ hits in 1 cycle, with 50% hit rate
- L2$ hits in 10 cycles, with 75% hit rate
- L3$ hits in 100 cycles, with 90% hit rate
- main memory hits in 1000 cycles

AMAT1 = 1 + (1-.5)(AMAT2)
= 1 + (1-.5)(10 + (1-.75)(AMAT3))
= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(AMATm)))
= 1 + (1-.5)(10 + (1-.75)(100 + (1-.9)(1000)))

= 31

Cache Performance: Multi-Level

AMATi = t_hiti + prob_missi * penalty_missi

- L1$ hits in 1 cycle, with 25% hit rate
- L2$ hits in 6 cycles, with 80% hit rate
- L3$ hits in 60 cycles, with 95% hit rate
- main memory hits in 1000 cycles

AMAT1 = 1 + (1-.25)(AMAT2)
= 1 + (1-.25)(6 + (1-.8)(AMAT3))
= 1 + (1-.25)(6 + (1-.8)(60 + (1-.95)(AMATm)))
= 1 + (1-.25)(6 + (1-.8)(60 + (1-.95)(1000)))

= 22

Layout of C Arrays in Memory

C arrays allocated in row-major order

- **each row in contiguous memory locations**

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d	e	f		o	p

Row Major Order (C)

a	e	i	m	b	f		l	p

Column-major Order (Fortran)

Layout of C Arrays in Memory

C arrays allocated in row-major order

- each row in contiguous memory locations

Stepping through columns in one row:

- ```
  for (i = 0; i < N; i++)
  sum += a[0][i];
  ```
- accesses successive elements
- if block size (B) > 4 bytes, exploit spatial locality
  - compulsory miss rate = 4 bytes / B

Stepping through rows in one column:

- ```
  for (i = 0; i < n; i++)
  sum += a[i][0];
  ```
- accesses distant elements
- no spatial locality!
  - compulsory miss rate = 1 (i.e. 100%)

Writing Cache Friendly Code

Repeated references to variables are good (temporal locality)

Stride-1 reference patterns are good (spatial locality)

Example: Summing all elements of a 2D array

- cold cache, 4-byte words, 4-word cache blocks

```
int sumarrayrows(int a[N][N])
{
    int i, j, sum = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

```
int sumarraycols(int a[N][N])
{
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < N; i++)
            sum += a[i][j];
    return sum;
}
```

Miss rate = 1/4 = 25%

Miss rate = 100%

Matrix Multiplication Example

Major Cache Effects to Consider

- **Total cache size**
  - Exploit temporal locality and keep the working set small (e.g., by using blocking)
- **Block size**
  - Exploit spatial locality

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** *held in register*

Description:

- **Multiply N x N matrices**
- **O(N3) total operations**
- **Accesses**
  - N reads per source element
  - N values summed per destination
    - » but may be able to hold in register

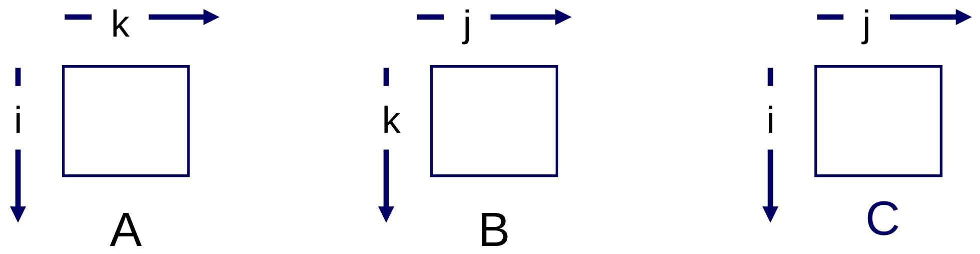
Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32B (big enough for 4 64-bit words)
- Matrix dimension (N) is very large
  - Approximate 1/N as 0.0
- Cache is not even big enough to hold multiple rows

Analysis Method:

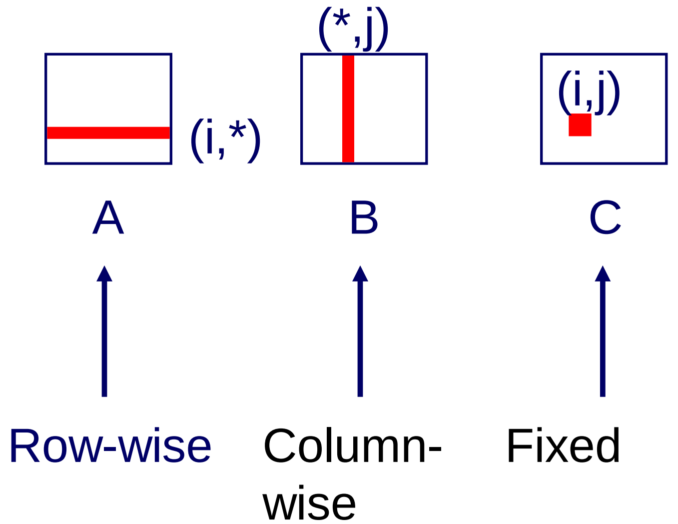
- Look at access pattern of inner loop



Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Inner loop:

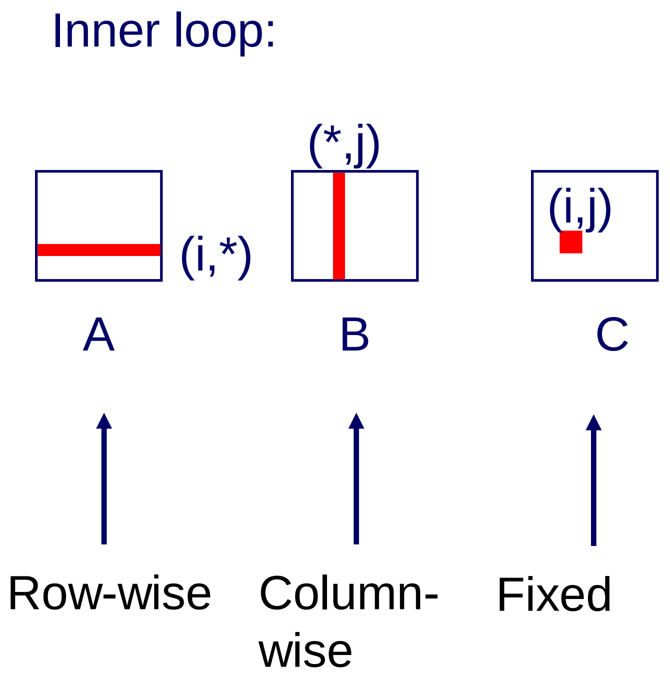


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum
  }
}
```

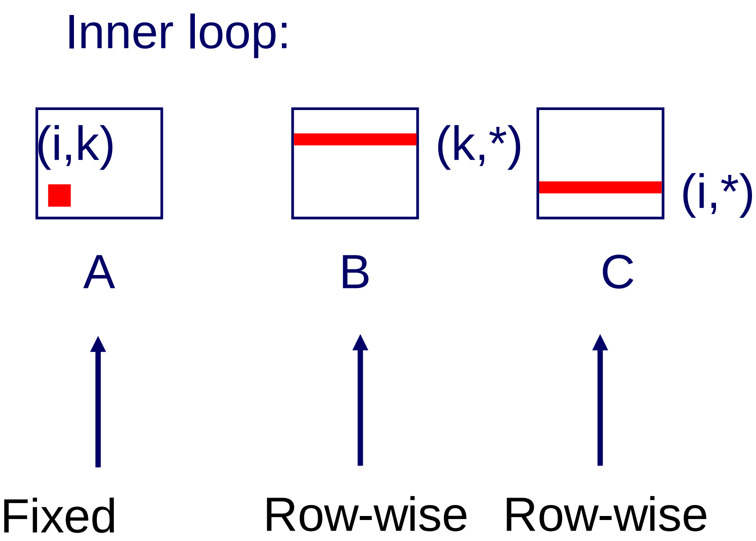


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```



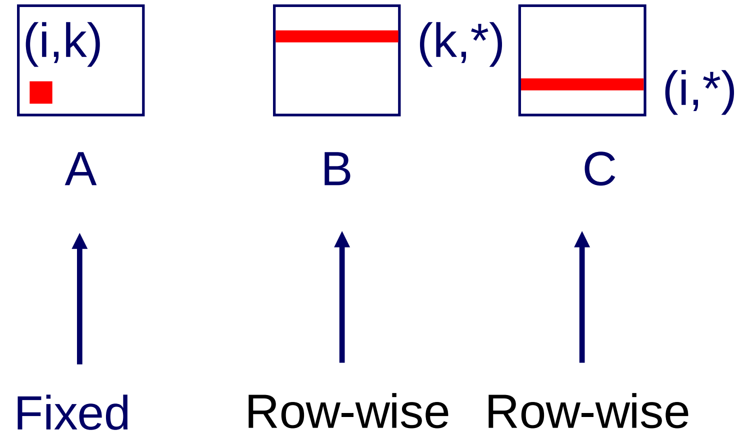
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

Inner loop:



Misses per Inner Loop Iteration:

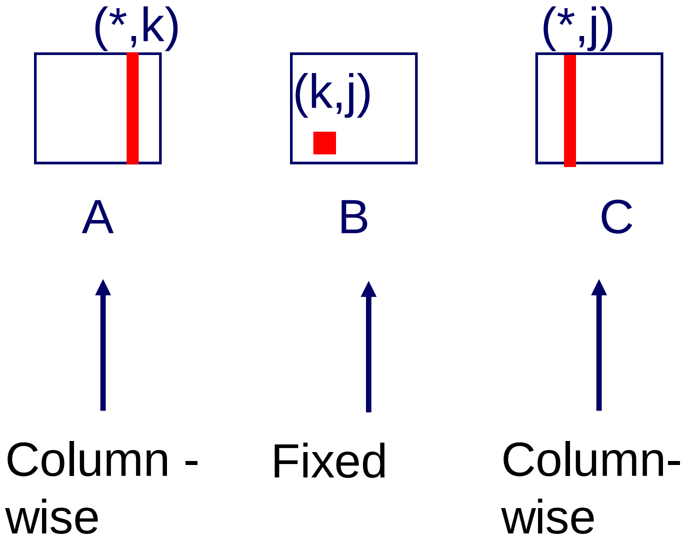
A	B	C
0.0	0.25	

0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



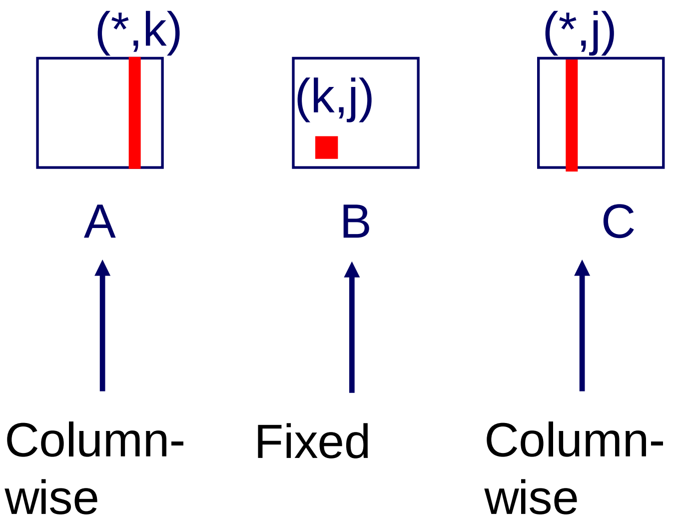
Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

Inner loop:



Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

```
for (i=0; i<n; i++)  {
   for (j=0; j<n; j++) {
       sum = 0.0;
       for (k=0; k<n; k++)
           sum += a[i][k] * b[k]
[j];
       c[i][j] = sum;
   }
}
```

kij (& ikj):

- 2 loads, 1 store
- misses/iter = **0.5**

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

   }
}
```

jki (& kji):

- 2 loads, 1 store
- misses/iter = **2.0**

```
for (j=0; j<n; j++) {
   for (k=0; k<n; k++) {
       r = b[k][j];
       for (i=0; i<n; i++)
           c[i][j] += a[i][k] * r;
   }
}
```