SSD 6

icarnegie

curriculum powered by Carnegie Mellon

Lecture 7 (1/2)

Memory Operation & Performance

"640k ought to be enough for anybody.", Bill Gates, 1981

武漢大學

国际软件学院

Lecture 7 Memory Operation and Performance

The Contents in SSD6 cover:

5.1 Memory Systems

5.2 Caches

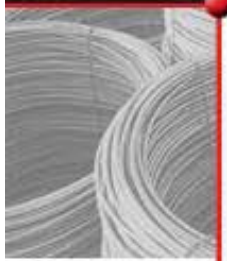
5.3 Virtual Memory (VM)

Exercise 5: Cache Lab

Lecture 7 Why Memory Hierarchy ?

- So far, we rely on a simple model of a computer system as a CPU that executes instructions and a memory system that holds instructions and data for the CPU.
- In practice, a memory system is a hierarchy of storage devices with different capacities, costs, and access times.
- Unwary programmers who assume a flat, uniform memory risk significant performance slowdowns in their programs, while wise programmers who understand the hierarchical nature of memory can produce efficient programs with fast average memory access times.

Lecture 7 Memory Technology

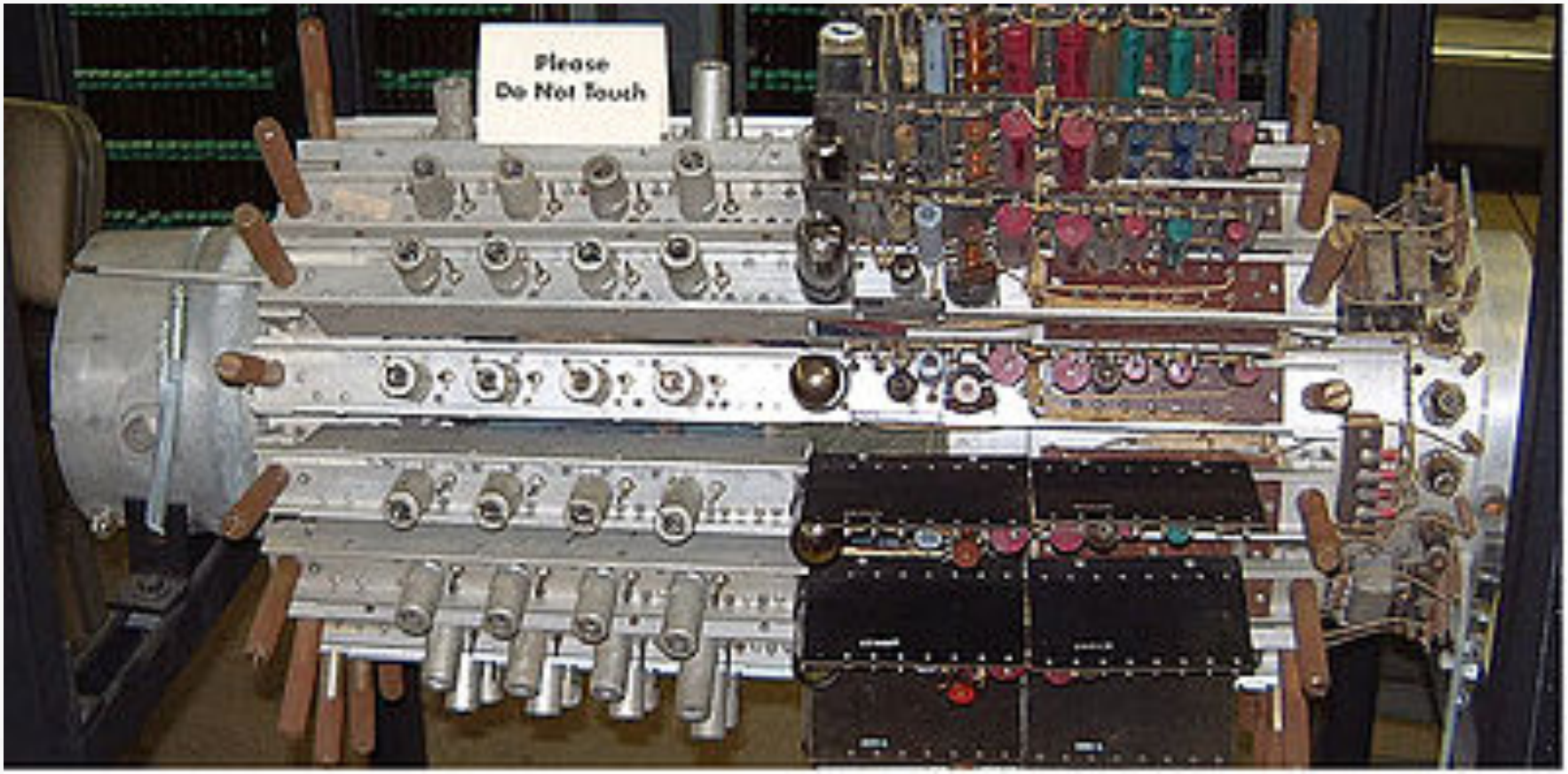




- There are many ways to store a bit of information.
- Current technologies use:
  - semiconductors (memory proper),
  - magnetic plates (hard disks),
  - and reflective puckered surfaces (CDs)
- Computers designed in England in the 1940s, used ***mercury delay lines*** to store bits.

5

Lecture 7 Memory Technology



Lecture 7 Memory Technology

- Static Random Access Memory (SRAM)
- Dynamic Random Access Memory (DRAM)
- Magnetic disks
- Magnetic tapes
- Optical disks

Lecture 7 Memory Technology

Characteristics of DRAM & SRAM memory

	Transistors per bit	Relative access time	Persistent?	Sensitive?	Relative Cost	Applications
SRAM	6	1X	Yes	No	100X	Cache memory
DRAM	1	10X	No	Yes	1X	Main mem, frame buffers

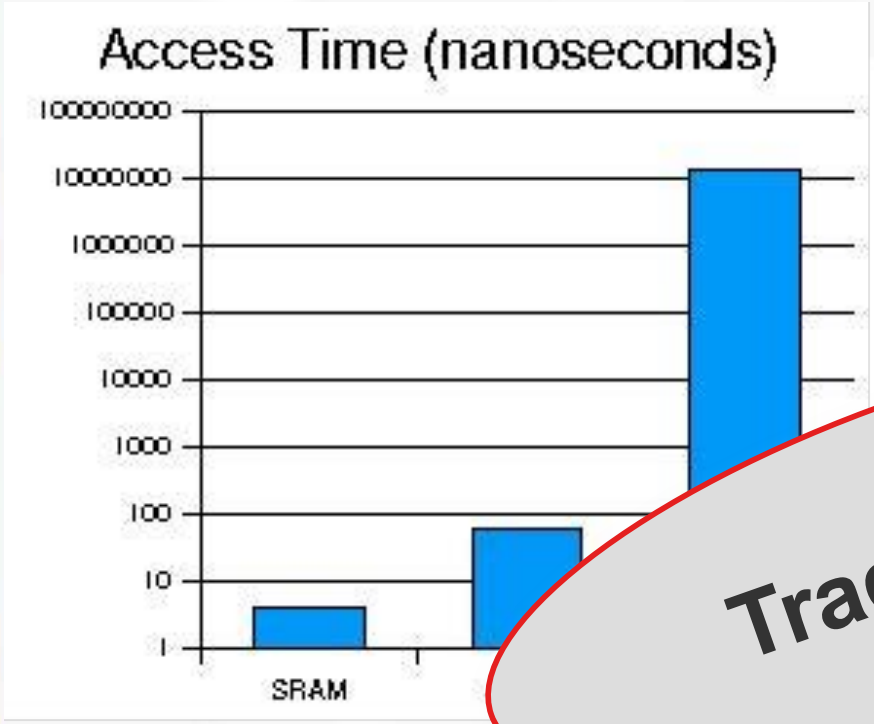
Gordon Moore @ 2004

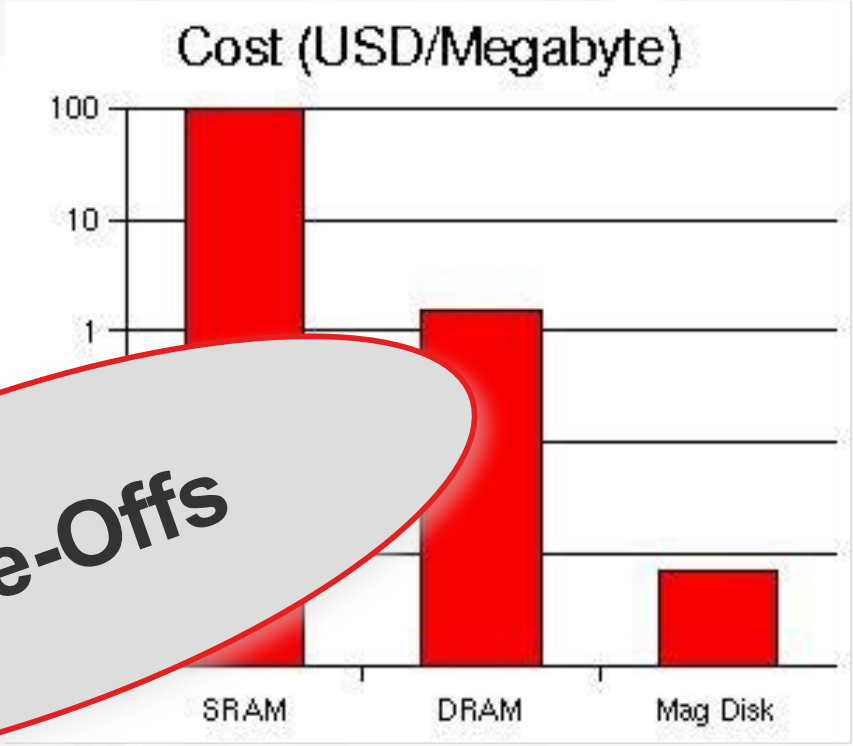
Lecture 7 Memory Technology

Characteristics of DISKs memories

- Disks are workhorse storage devices that hold enormous amounts of data, on the order of tens to hundreds of gigabytes, as opposed to the hundreds or thousands of megabytes in a RAM-based memory.
- However, it takes on the order of milliseconds to read information from a disk, a hundred thousand times longer than from DRAM and a million times longer than from SRAM

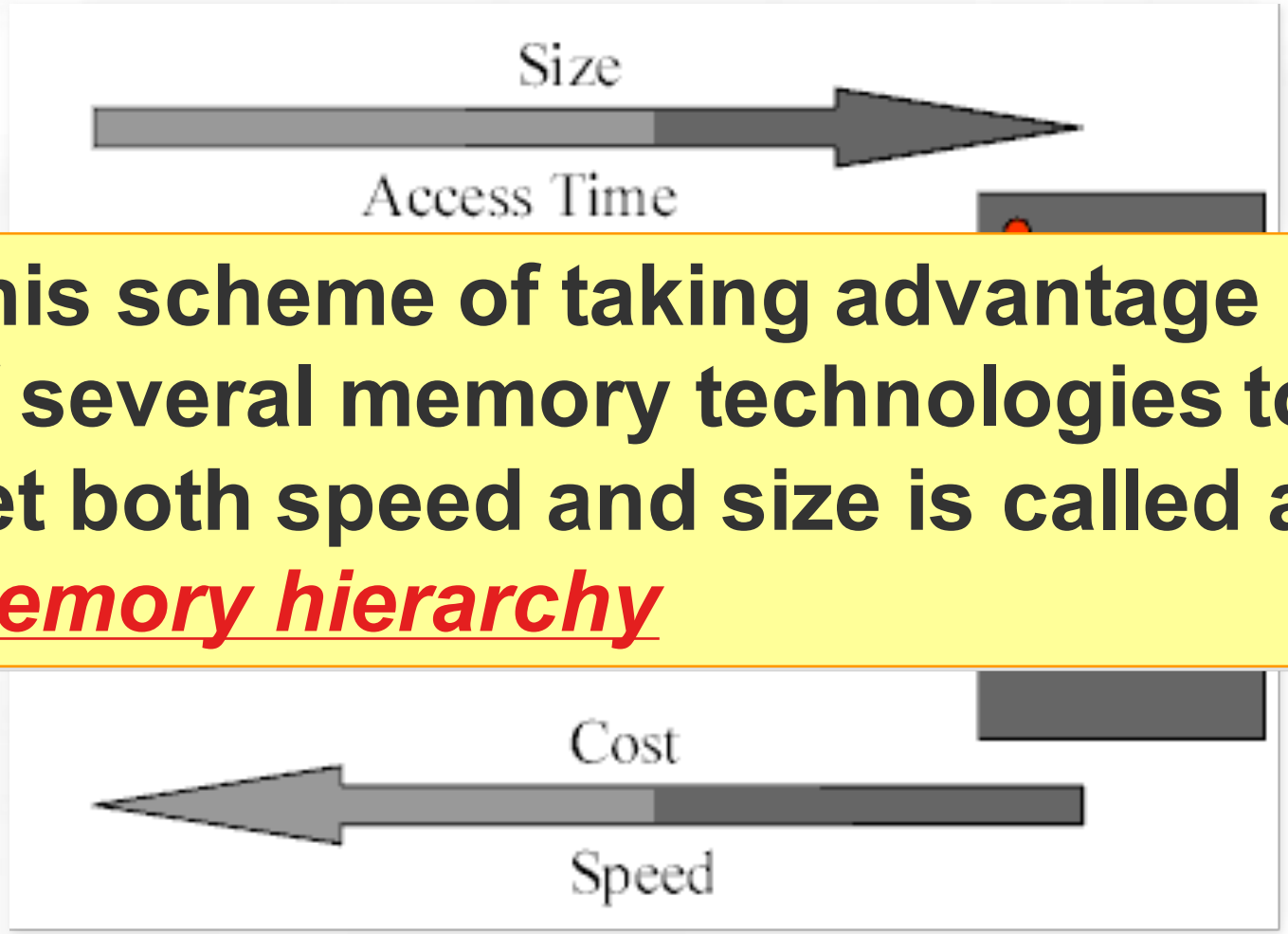
Lecture 7 Memory Technology





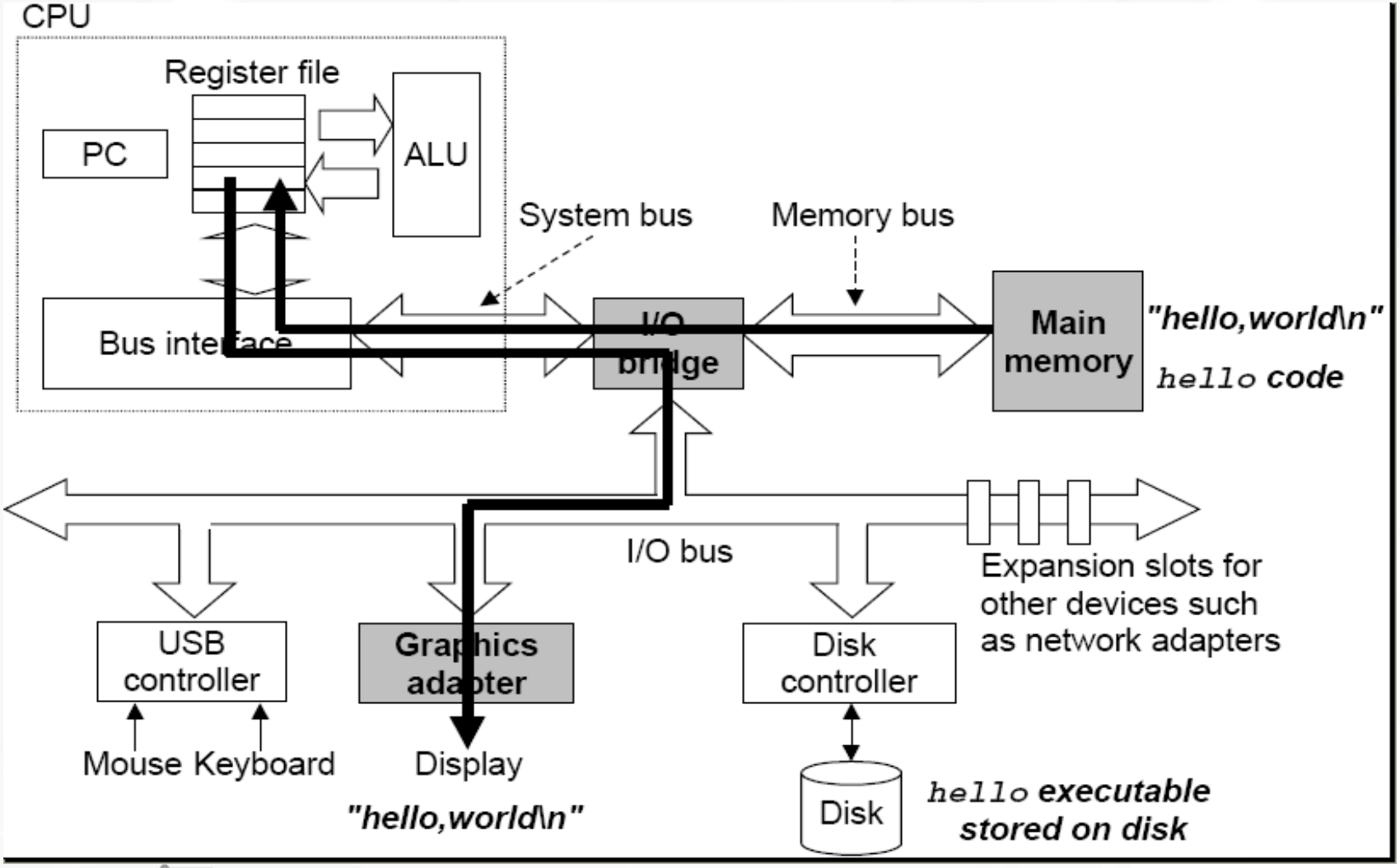
Trade-Offs

Lecture 7 Memory Technology



This scheme of taking advantage of several memory technologies to get both speed and size is called a *memory hierarchy*

Lecture 7 Memory Technology



Lecture 7

Locality of Reference

- Ideal situation:
  - addresses of memory to be accessed by the CPU during execution would be known ahead of time
  - ***prefetch* data**
- Actual situation:
  - Impossible to know exactly which addresses will be accessed by the CPU
  - Compiler cannot by itself decide which data to move from slow to fast memory

Lecture 7 Locality of Reference

GUESS!

- the immediate future will be similar to the immediate past and that memory addresses that have been accessed recently are likely to be accessed again.

——locality of reference

Lecture 7

Locality of Reference

- Spatial and Temporal Locality
  - References to a single address occur close together in time (this is called *temporal* locality).
  - References to addresses that are near to each other occur together in time (this is called *spatial* locality).
- Locality in a Code Fragment

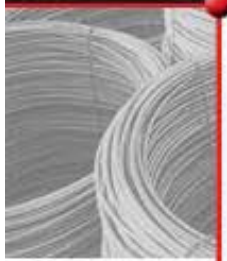
Lecture 7 Locality of Reference

Theory?

- It is just that the typical program does
  - The local variables of your procedures are all accessed when the procedures execute
  - variables tend to be accessed multiple times within their useful lifetime
- Locality of reference is in a statistic sense, rather in a “always” sense.

Lecture 7

Locality of Reference





- It is a property of programs, not of computers
- Program's locality is a property of the program's behavior, and not of the computer

Lecture 7 Locality of Reference

Locality in a Code Fragment

```
1 int sumvec(int v[N])
2 {
3     int i, sum = 0;
4
5     for (i = 0; i < N; i++)
6         sum += v[i];
7     return sum;
8 }
```

(a)

A function with good locality, *N*=8

Address	0	4	8	12	16	20	24	28
Contents	v_0	v_1	v_2	v_3	v_4	v_5	v_6	v_7
Access order	1	2	3	4	5	6	7	8

Lecture 7 Locality of Reference

Locality in a Code Fragment

Another function with good locality, a[2][3]

```
1 int sumarrayrows(int a[M][N])
2 {
3       int i, j, sum = 0;
4
5       for (i = 0; i < M; i++)
6           for (j = 0; j < N; j++)
7               sum += a[i][j];
8       return sum;
9 }
```

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	2	3	4	5	6

Lecture 7 Locality of Reference

Locality in a Code Fragment

```
1 int sumarraycols(int a[M][N])
2 {
3     int i, j, sum = 0;
4
5     for (j = 0; j < N; j++)
6         for (i = 0; i < M; i++)
7             sum += a[i][j];
8     return sum;
9 }
```

A function with poor spatial locality.

Address	0	4	8	12	16	20
Contents	a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}
Access order	1	3	5	2	4	6

```
1 #define N 1000
2
3 typedef struct {
4     int vel[3];
5     int acc[3];
6 } point;
7
8 point p[N];
```

(a) An array of `structs`.

```
1 void clear1(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++)
7             p[i].vel[j] = 0;
8         for (j = 0; j < 3; j++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear1` function.

```
1 void clear2(point *p, int n)
2 {
3     int i, j;
4
5     for (i = 0; i < n; i++) {
6         for (j = 0; j < 3; j++) {
7             p[i].vel[j] = 0;
8             p[i].acc[j] = 0;
9         }
10    }
11 }
```

(a) The `clear2` function.

```
1 void clear3(point *p, int n)
2 {
3     int i, j;
4
5     for (j = 0; j < 3; j++) {
6         for (i = 0; i < n; i++)
7             p[i].vel[j] = 0;
8         for (i = 0; i < n; i++)
9             p[i].acc[j] = 0;
10    }
11 }
```

(b) The `clear3` function.

Lecture 7 Locality of Reference

Locality in a Code Fragment

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum += array[i];
```

Lecture 7

Locality of Reference

Locality in a Code Fragment

```
sum = 0;
for (i = 0; i < MAX; i++)

    sum += array[i];
```

Address	Instruction
FRAG	sum = 0;
FRAG + 1	i = 0;
FRAG + 2	MAX - i
FRAG + 3	branch FRAG + 7 if negative
FRAG + 4	sum = sum + array[i]
FRAG + 5	i = i + 1
FRAG + 6	jump FRAG + 2

Lecture 7

Locality of Reference

Locality in a Code Fragment

```
sum = 0;
for (i = 0; i < MAX; i++)

    sum += array[i];
```

		Instructions	Data Read	Data Write
1	sum = 0;	FRAG		STACKPTR + X
2	i = 0;	FRAG + 1		STACKPTR + X + 4
3	i < MAX ?	FRAG + 2	STACKPTR + X + 4	
4		FRAG + 3		
5	sum += array[i]	FRAG + 4	STACKPTR + X + 4 STACKPTR + X ARRAY	STACKPTR + X
6	i++	FRAG + 5	STACKPTR + X + 4	STACKPTR + X + 4
7		FRAG + 6		
8	i < MAX ?	FRAG + 2	STACKPTR + X + 4	
9		FRAG + 3		
10	sum += array[i]	FRAG + 4	STACKPTR + X + 4 STACKPTR + X ARRAY + 1	STACKPTR + X
11	i++	FRAG + 5	STACKPTR + X + 4	STACKPTR + X + 4
12		FRAG + 6		
13	i < MAX ?	FRAG + 2	STACKPTR + X + 4	
14		FRAG + 3		

Lecture 7 Locality of Reference

Locality in a Code Fragment

		Instructions	Data Read	Data Write
1	sum = 0;	FRAG		STACKPTR + X
2	i = 0;	FRAG + 1		STACKPTR + X + 4
3	i < MAX ?	FRAG + 2	STACKPTR + X + 4	
4		FRAG + 3		
5	sum += array[i]	FRAG + 4	STACK	
6	i++			
7				
8	i <			
9				
11			TR + X + 4	STACKPTR + X + 4
12		FRAG + 6		
13	i < MAX ?	FRAG + 2	STACKPTR + X + 4	
14		FRAG + 3		

The principle is statistically true!

- The fetch of FRAG + 4 in line 5 predicts the fetch in line 10—
- edicts the
- ses between
- o. —spatial
- es to STACKPTR + X
- hat STACKPTR + X + 4—
- tial locality.
- The references to both STACKPTR + X and STACKPTR + X + 4 predict further repeated references to each—temporal locality.
- The reference to ARRAY predicts later reference to ARRAY + 1—spatial locality.

Lecture 7

Memory Hierarchies

- memory hierarchy
  - Computers use memory technologies of varying size and speed to achieve both acceptable speed and size at an affordable cost
- hit ratio
  - The percentage of accesses for which the prediction is successful
  - Typical hit ratios in today's computers are almost always above 0.90
  - Look at some numerical evidences in our doc.

Lecture 7 Memory Hierarchies

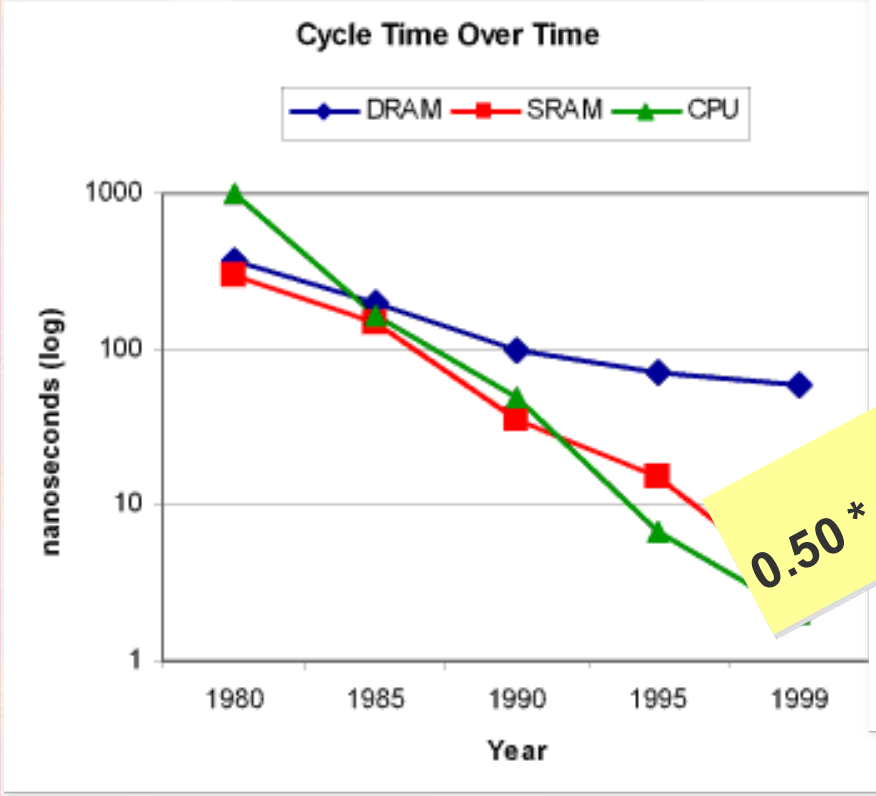
- An Intel Pentium III 600 MHz processor can execute hundreds of millions of instructions per second
- Condition: Each of those instructions must be fetched from memory
- capacity to generate at least 200 million memory references per second
- reference cannot take, on the average, more than about 1/200,000,000 of a second, or 5 nanoseconds
- However, DRAM with an access time of 60 nanoseconds or more

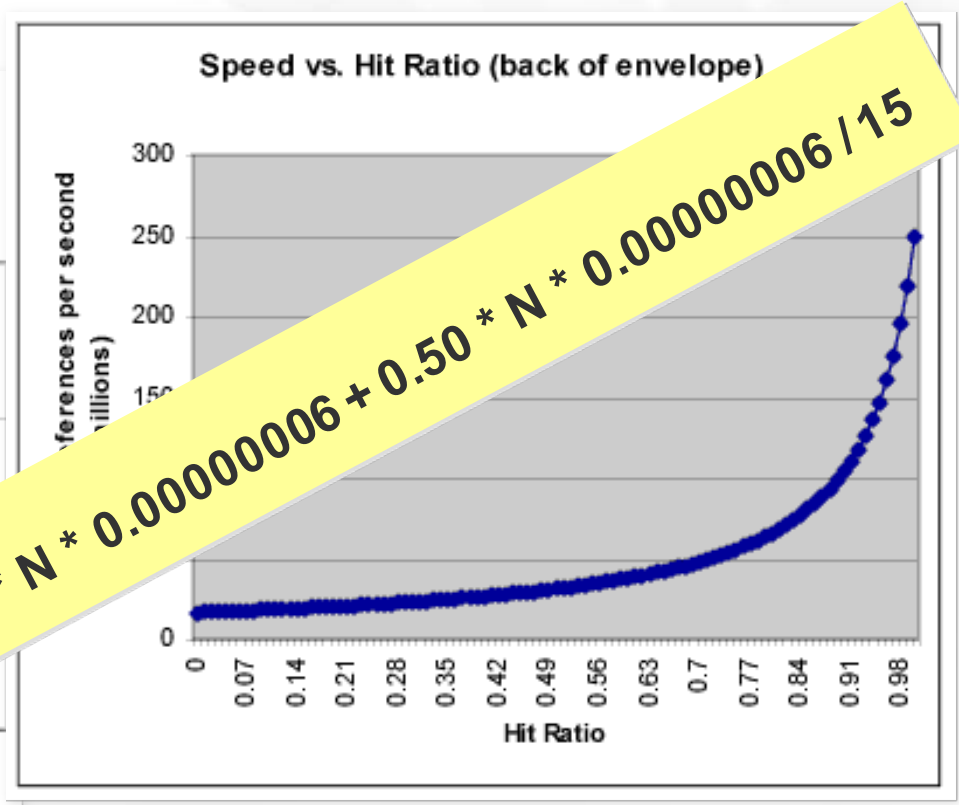
Lecture 7 Memory Hierarchies

- What if we had no memory hierarchy to access DRAM (that is, if the hit ratio were 0.0)?
- Access time to DRAM is 60 nanoseconds
- Complete 1/0.00000006 memory references in one second, or fewer than 17 million
- 9 million instructions per second
- slower than a 50 MHz processor does
- A 600MHz processor operating at the speed of a 50 MHz, if a cache hit ratio is 0.0!

Lecture 7 Memory Hierarchies

The memory bottleneck is increasingly the obstacle to high computer performance





Lecture 7

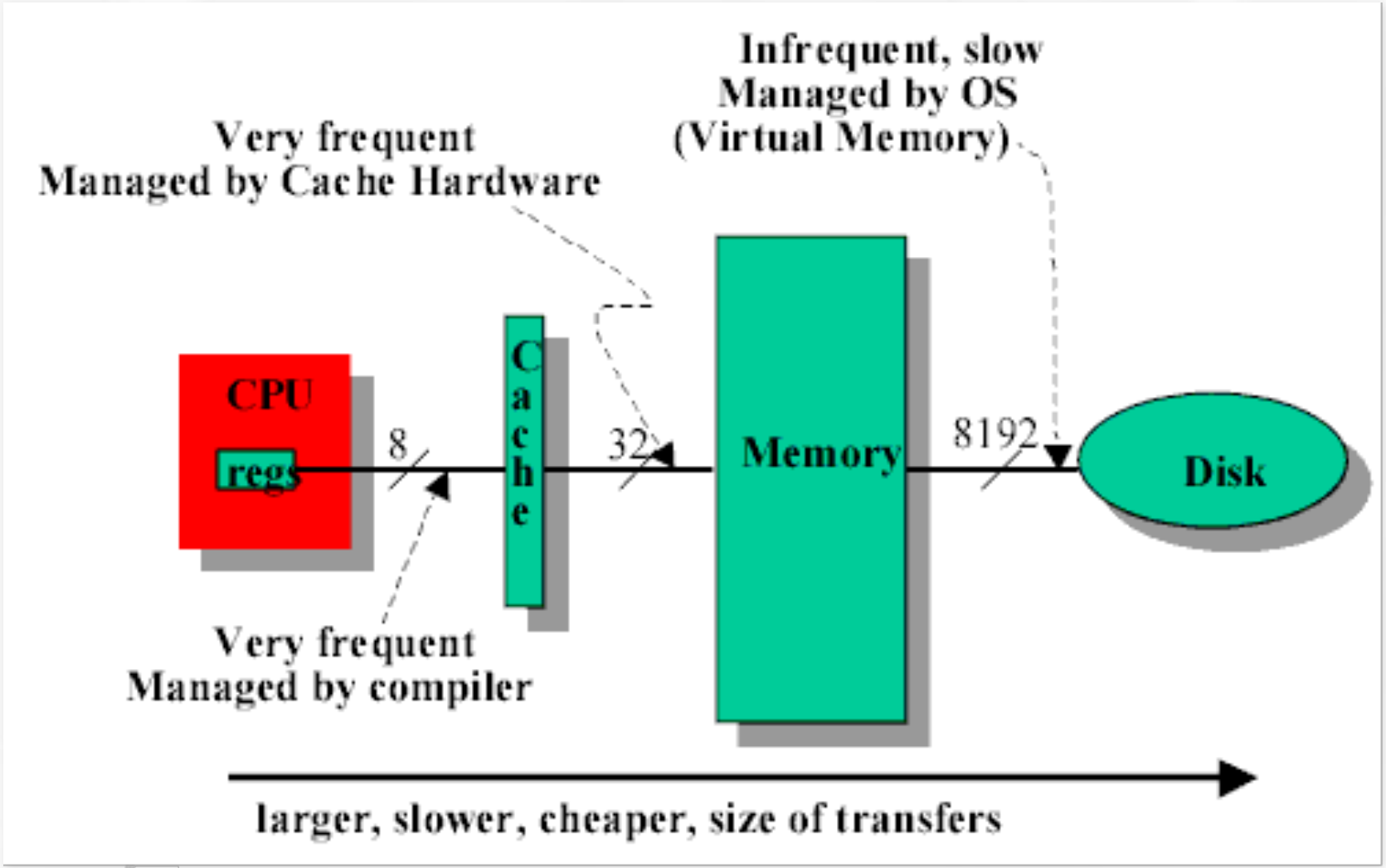
Memory Hierarchies

Design of the Hierarchy

- Typical computer designs have at least four levels in the hierarchy
  - Registers are closest to the CPU, are smallest in size, and are also the fastest memory in the system
  - Furthest from the CPU is the disk, where large inactive parts of programs sleep for long periods of time
  - In between: caches & main memory
    - The policy decisions for these levels are made as the program executes
    - The computer has special cache hardware that moves data from memory to caches and vice versa.

Lecture 7 Memory Hierarchies

Design of the Hierarchy

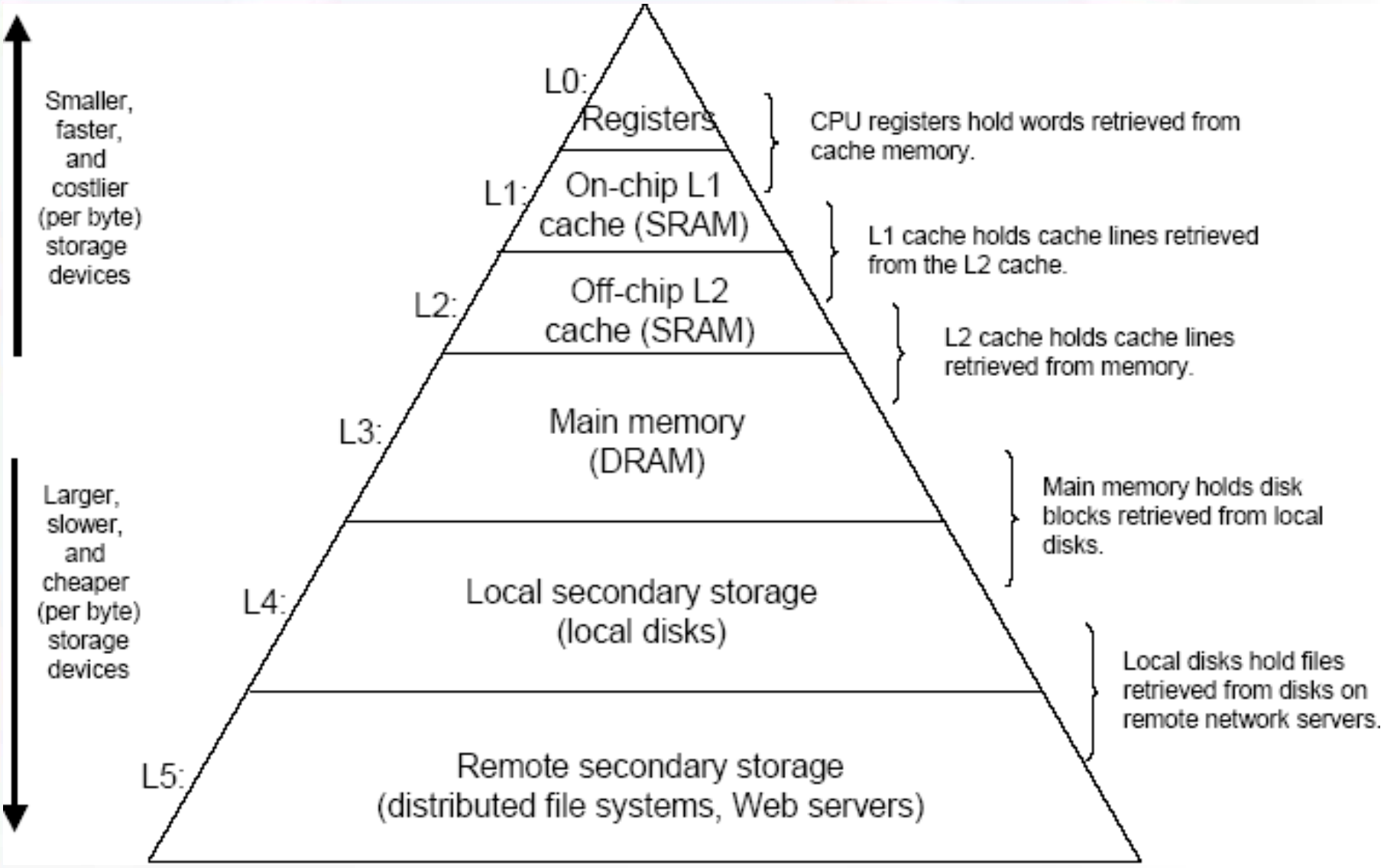


Lecture 7 Memory Hierarchies

Design of the Hierarchy : Strategies

- **Fetch**: When should data be moved up the hierarchy? How much data should be moved in each transaction?
- **Placement**: Where in the smaller memory should the fetched data be stored? How can it be found when requested by the CPU?
- **Replacement**: When newly fetched data finds its location full, which previously resident data should be evicted?
- **Update**: When the "cached" data is changed, should the corresponding data in a lower level of memory be updated immediately or later?

Lecture 7 Memory Hierarchies



Lecture 7 Caches

Special terms:

- Caches: small, fast storage devices that act as staging areas for the data objects stored in larger, slower devices
- Each level in the hierarchy caches data objects from the next lower level
- Miss: any data requested by the CPU that is not in the cache must be immediately fetched.
- Cache lines: the "chunks" in which all transfers to and from the cache are done
- Cache lines are typically in the order of 16 or 32 bytes long

Lecture 7 Caches

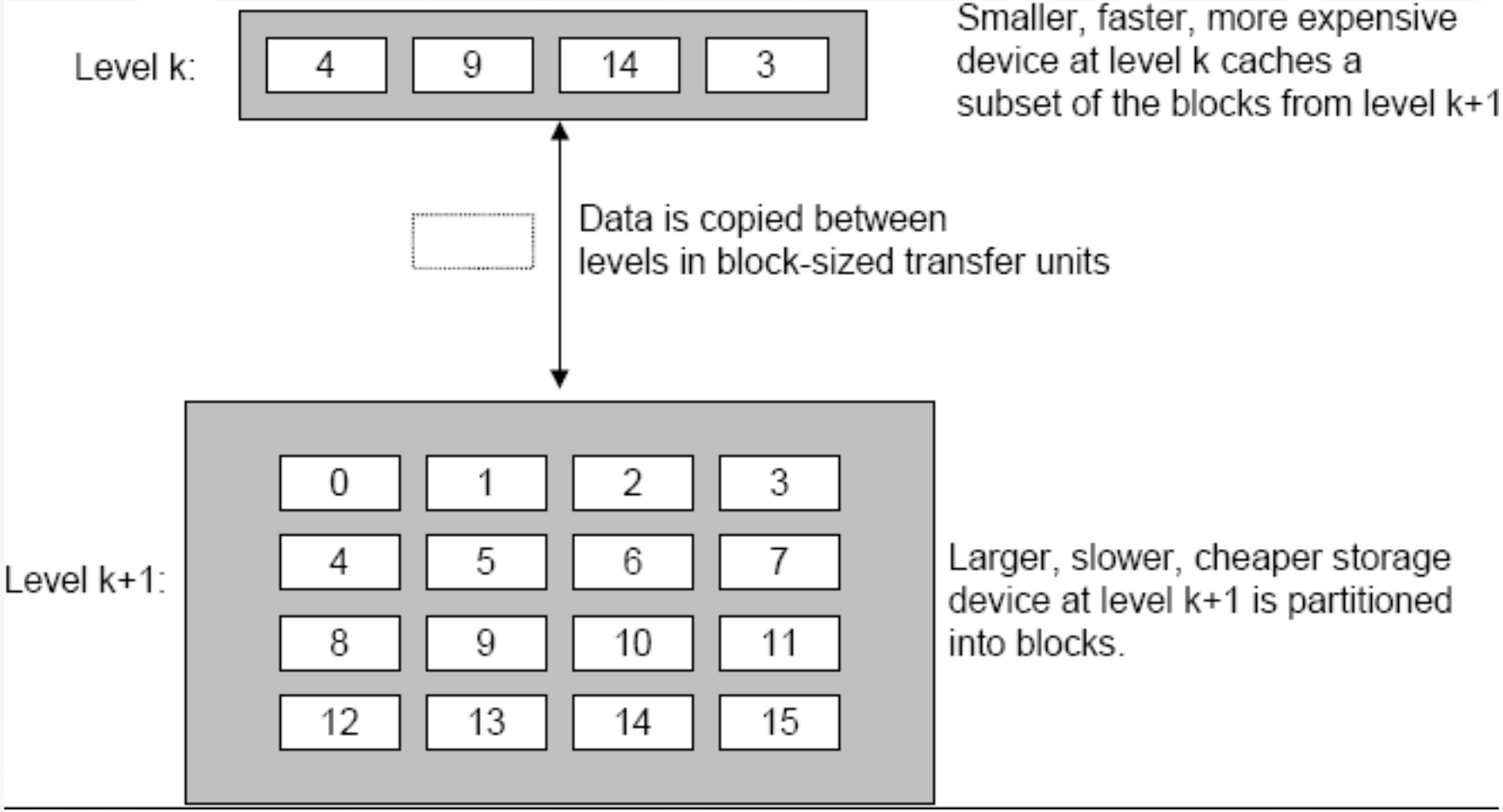
- Caches are used everywhere in modern systems.
- Caches are used in CPU chips, operating systems, distributed file systems, and on the World-Wide Web.
- They are built from and managed by various combinations of hardware and software.

Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware
L1 cache	32-byte block	On-chip L1 cache	1	Hardware
L2 cache	32-byte block	Off-chip L2 cache	10	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.23: **The ubiquity of caching in modern computer systems.** Acronyms: TLB: Translation Lookaside Buffer, MMU: Memory Management Unit, OS: Operating System, AFS: Andrew File System, NFS: Network File System.

Lecture 7 Caches

Special terms:



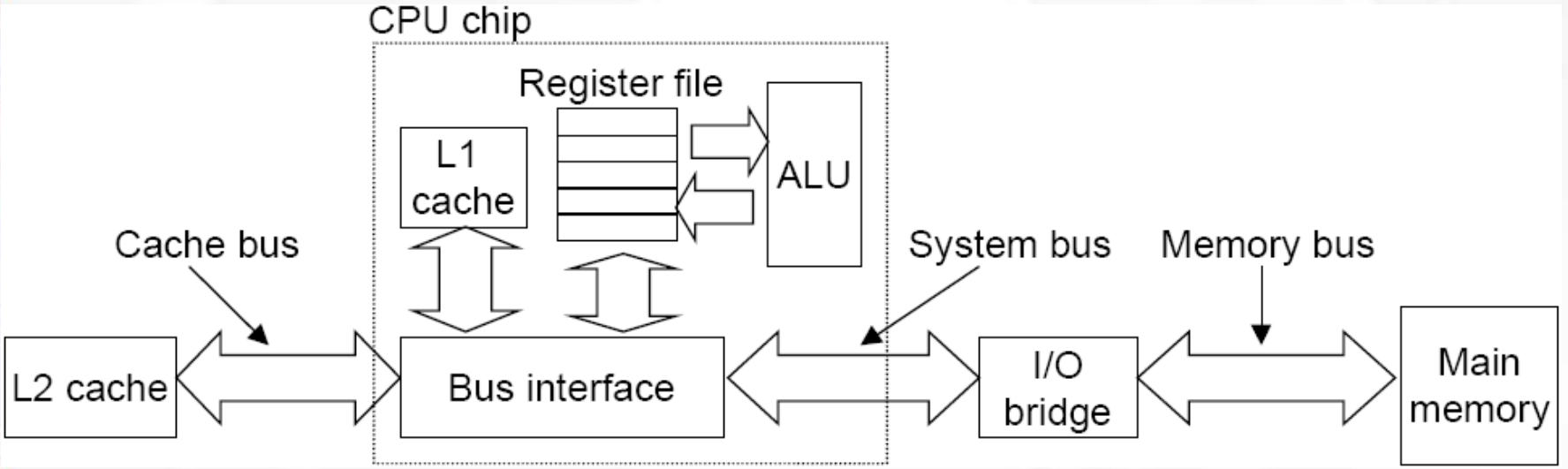
Lecture 7 Caches

Special terms:

- Cache Hits:
- Cache Misses
  - Cold misses, or compulsory misses
  - Conflict misses
  - Capacity misses

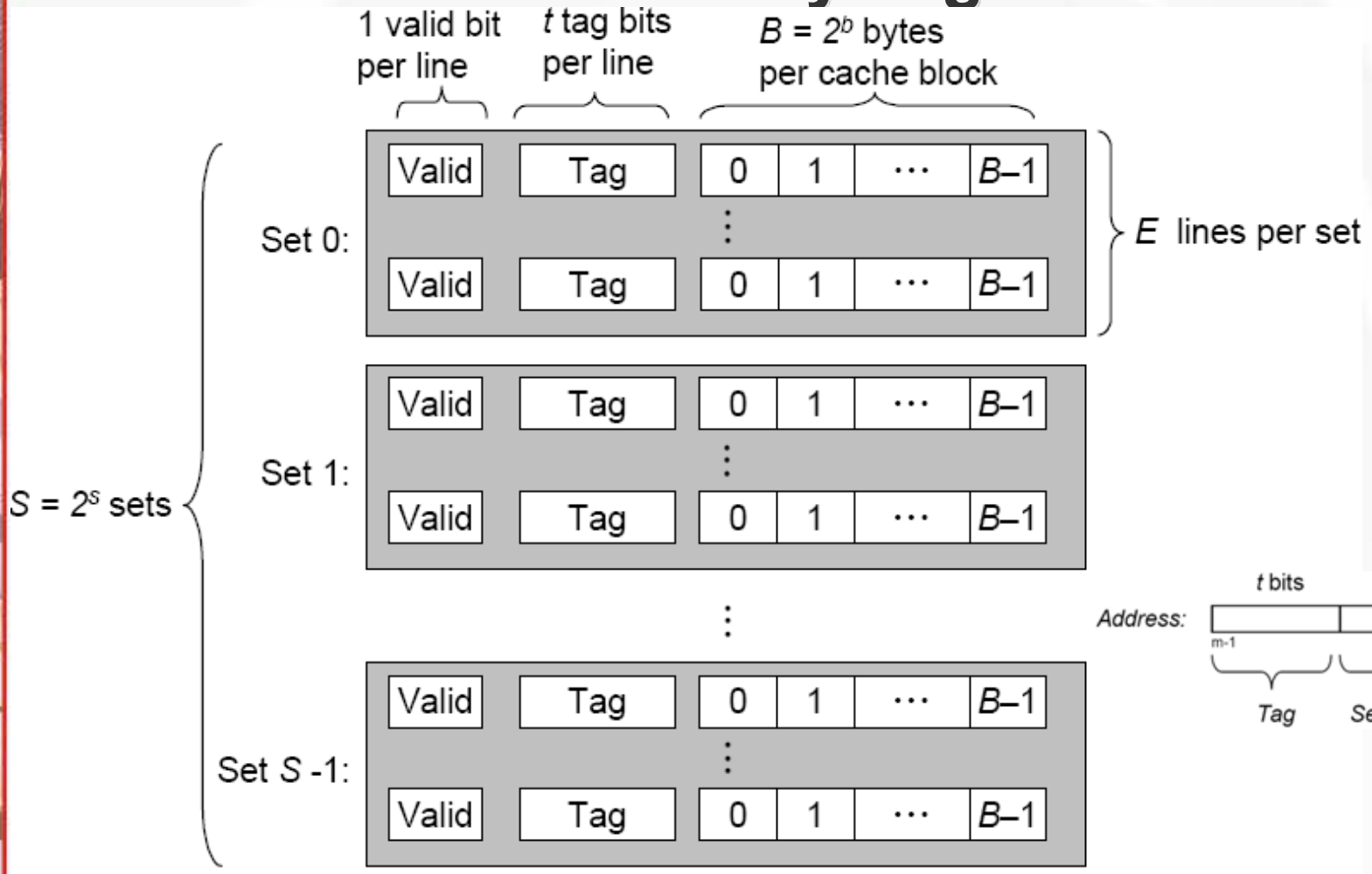
Lecture 7 Cache Memories

Typical bus structure for L1 & L2 caches



Lecture 7 Cache Memories

Generic Cache Memory Organization



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

Lecture 7

Cache Memories

An Example

Fundamental parameters	
Parameters	Description
$S = 2^s$	Number of Sets
E	Number of lines per set
$B = 2^b$	Block size (bytes)
$m = \log_2(M)$	Number of physical address bits

Lecture 7 Cache Memories

An Example

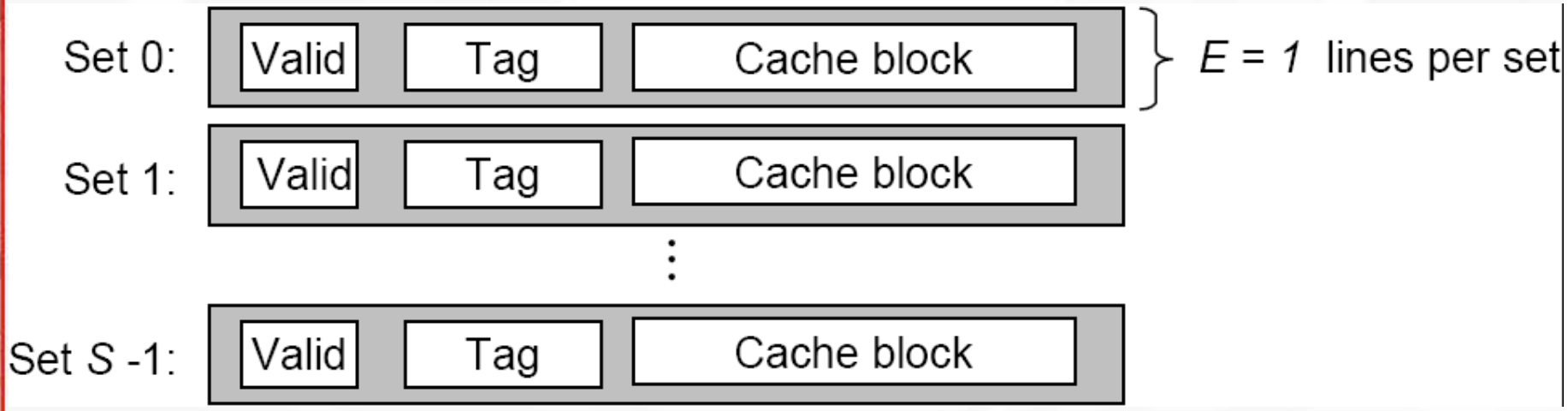
Derived quantities	
Parameters	**Description**
$M = 2^m$	Maximum number of unique mem address
$s = \log_2(S)$	Number of set index bits
$b = \log_2(B)$	Number of block offset bits
$t = m - (s + b)$	Number of tag bits
$C = B \times E \times S$	Cache size (bytes) not including overhead such as the valid and tag bits

Cache	*m*	*C*	*B*	*E*	*S*	*t*	*s*	*b*
1	32	1024	4	1	256	2	8	22
2	32	1024	8	4	32	24	5	3
3	32	1024	32	32	1	27	0	5

Lecture 7

Cache Memories

Direct-Mapped Caches



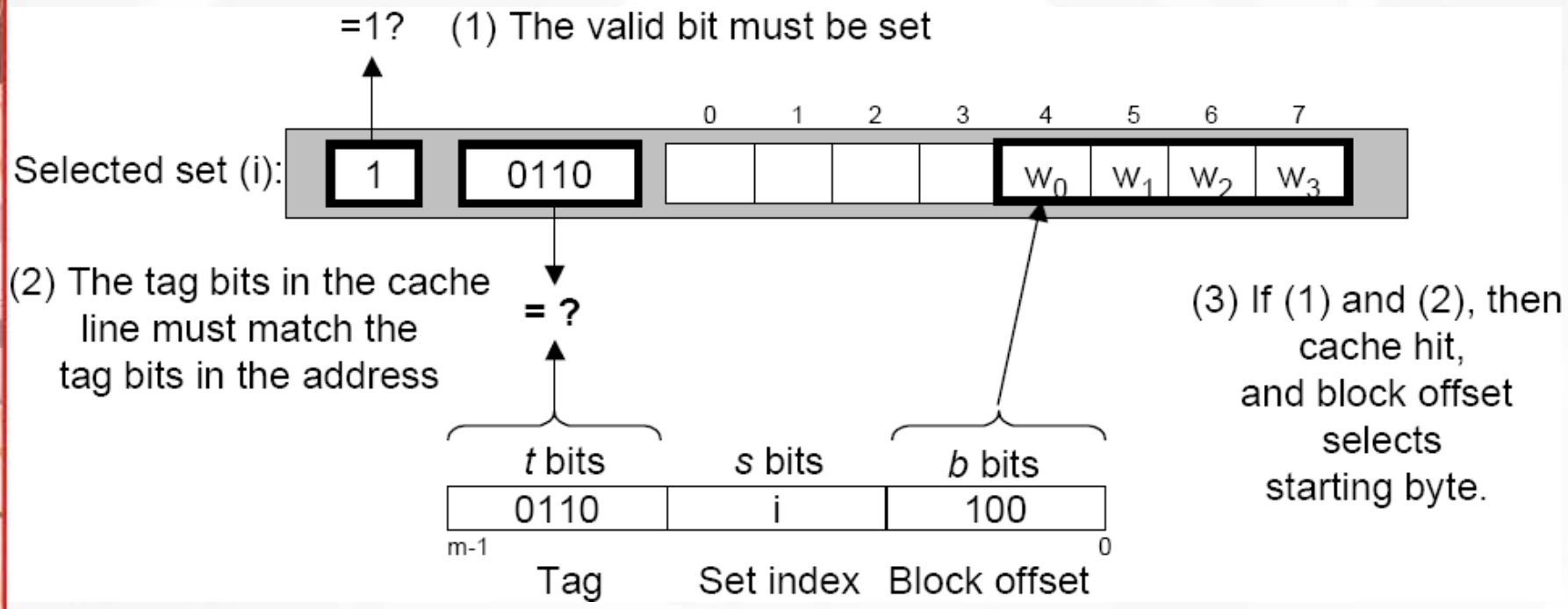
Lecture 7 Cache Memories

Set selection in a direct-mapped cache

Level K **Level K+1**

Lecture 7 Cache Memories

Word selection in Direct-mapped Caches



Lecture 7 Cache Memories

An Example

(S,E,B,m)=(4,1,2,4)

	Address Bits			
Address (decimal)	Tag bits (t=1)	Index bits (s=2)	Offset bits (b=1)	Block number (decimal)
0		0	0	
1		0	1	
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				

Lecture 7 Cache Memories

An Example

(S,E,B,m)=(4,1,2,4)

	Address Bits			
Address (decimal)	Tag bits (t=1)	Index bits (s=2)	Offset bits (b=1)	Block number (decimal)
0	0	00	0	
1	0	00	1	
2	0	01	0	
3	0	01	1	
4	0	10	0	
5	0	10	1	
6	0	11	0	
7	0	11	1	
8				
9				
10				
11				
12				
13				
14				
15				

Lecture 7 Cache Memories

An Example

(S,E,B,m)=(4,1,2,4)

	Address Bits			
Address (decimal)	Tag bits (t=1)	Index bits (s=2)	Offset bits (b=1)	Block number (decimal)
0	0	00	0	0
1	0	00	1	0
2	0	01	0	1
3	0	01	1	1
4	0	10	0	2
5	0	10	1	2
6	0	11	0	3
7	0	11	1	3
8	1	00	0	4
9	1	00	1	4
10	1	01	0	5
11	1	01	1	5
12	1	10	0	6
13	1	10	1	6
14	1	11	0	7
15	1	11	1	7

Lecture 7 Cache Memories

An Example

(S,E,B,m)=(4,1,2,4)

- Block 0 consists of address 0&1, Block 1 consists of address 2&3, ……
- Blocks 0 & 4 both map to set 0, Blocks 1 & 5 both map to set 1, ……
- Blocks that map to the same cache set are uniquely identified by the tag: block 0 has a tag bit of 0, while block 4 has a tag bit of 1, ……

Lecture 7 Cache Memories

An Example

1. Initial status:

*** Empty ***

Set	Valid	Tag	block (0)	block (1)
0	0			
1	0			
2	0			
3	0			

Lecture 7 Cache Memories

An Example

2. Read word at address 0:

***** Cache Miss *****

Set	Valid	Tag	block (0)	block (1)
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

Lecture 7 Cache Memories

An Example

3. Read word at address 1:

***** Cache Hit *****

Set	Valid	Tag	block (0)	block (1)
0	1	0	m[0]	m[1]
1	0			
2	0			
3	0			

Lecture 7 Cache Memories

An Example

4. Read word at address 13:

***** Cache Miss *****

Set	Valid	Tag	block (0)	block (1)
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Lecture 7 Cache Memories

An Example

5. Read word at address 8:

***** Cache Miss *****

Set	Valid	Tag	block (0)	block (1)
0	1	1	m[8]	m[9]
1	0			
2	1	1	m[12]	m[13]
3	0			

Lecture 7 Cache Memories

An Example

6. Read word at address 0:

***** Cache Miss *****

Set	Valid	Tag	block (0)	block (1)
0	1	0	m[0]	m[1]
1	0			
2	1	1	m[12]	m[13]
3	0			

Lecture 7 Cache Memories

Conflict Misses in Direct-Mapped Caches

A fragment Code

```
float dotprod ( float x[8], float y[8])
{
        float sum=0.0;
        int i;
        for ( i=0; i<8; i++)
                sum += x[i] + y[i];
        return sum;
}
```

Lecture 7 Cache Memories

Conflict Misses in Direct-Mapped Caches

```
float dotprod ( float x[8], float y[8])
{
        float sum=0.0;
        int i;
        for ( i=0; i<8; i++)
                sum +=
        return sum;
}
```

Thrashing!
How to solve?

Element	Address	Set i			Set index
x[0]	0				0
x[1]	4	0		36	0
x[2]	8	0	y[2]	40	0
x[3]	12	0	y[3]	44	0
x[4]	16	1	y[4]	48	1
x[5]	20	1	y[5]	52	1
x[6]	24	1	y[6]	56	1
x[7]	28	1	y[7]	60	1

Lecture 7 Cache Memories

Conflict Misses in Direct-Mapped Caches

```
float dotprod ( float x[8], float y[8])
{
        float sum=0.0;
        int i;
        for ( i=0; i<8; i++)
                sum +=
        return sum
}
```

Define x[12] rather than x[8]

Element	Address				Set index
x[0]	0				1
x[1]	4			52	1
x[2]	8			56	1
x[3]	12	0	y[3]	60	1
x[4]	16	1	y[4]	64	0
x[5]	20	1	y[5]	68	0
x[6]	24	1	y[6]	72	0
x[7]	28	1	y[7]	76	0

Lecture 7 Cache Memories

Conflict Misses in Direct-Mapped Caches

Conclusions:

Even through the program has good spatial locality, and we have room in the cache to hold the blocks for both x[i] and y[i], each reference results in a conflict miss because the blocks map to same cache set.

Lecture 7 Cache Memories

Different Caches

Classification	Cost	Hit Rate	Miss Rate
Direct mapped	$5	88%	12%
Fully Associative	$500000	99.8%	0.02%
S-way set Associative	$20~$50	98~99%	1~2%

SSD 6

icarnegie

curriculum powered by Carnegie Mellon

Lecture 7 (2/2)

Memory Operation

&

Performance

武漢大學

国际软件学院

Lecture 7 Memory Operation and Performance

The Contents in SSD6 cover:

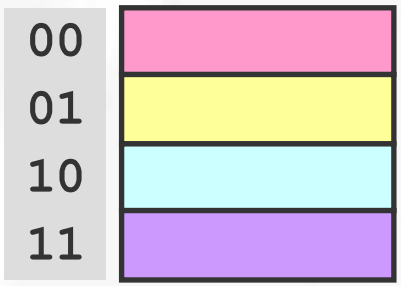
5.1 Memory Systems

5.2 Caches

5.3 Virtual Memory (VM)

Exercise 5: Cache Lab

Lecture 7 Why Use Middle Bits as Index?

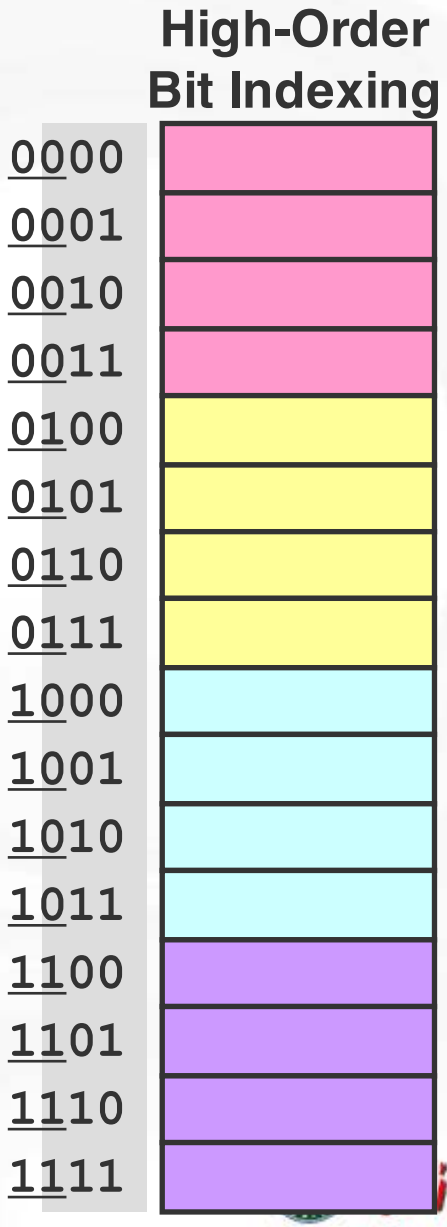


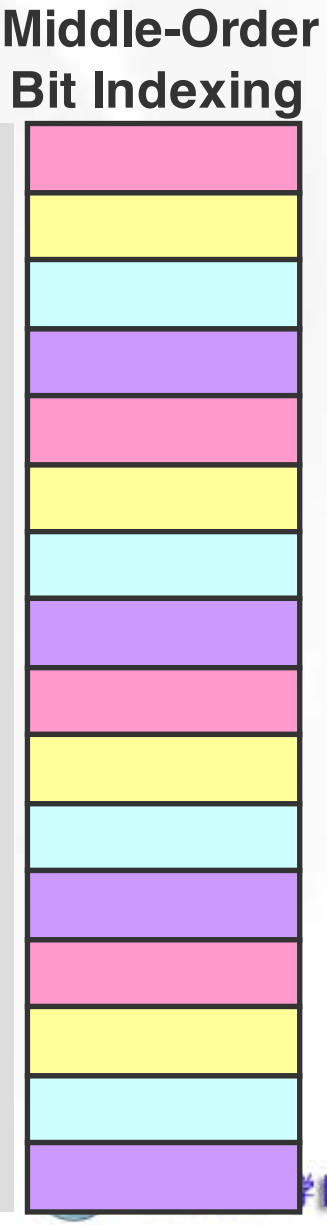
High-Order Bit Indexing

- Adjacent memory lines would map to same cache entry
- Poor use of spatial locality

Middle-Order Bit Indexing

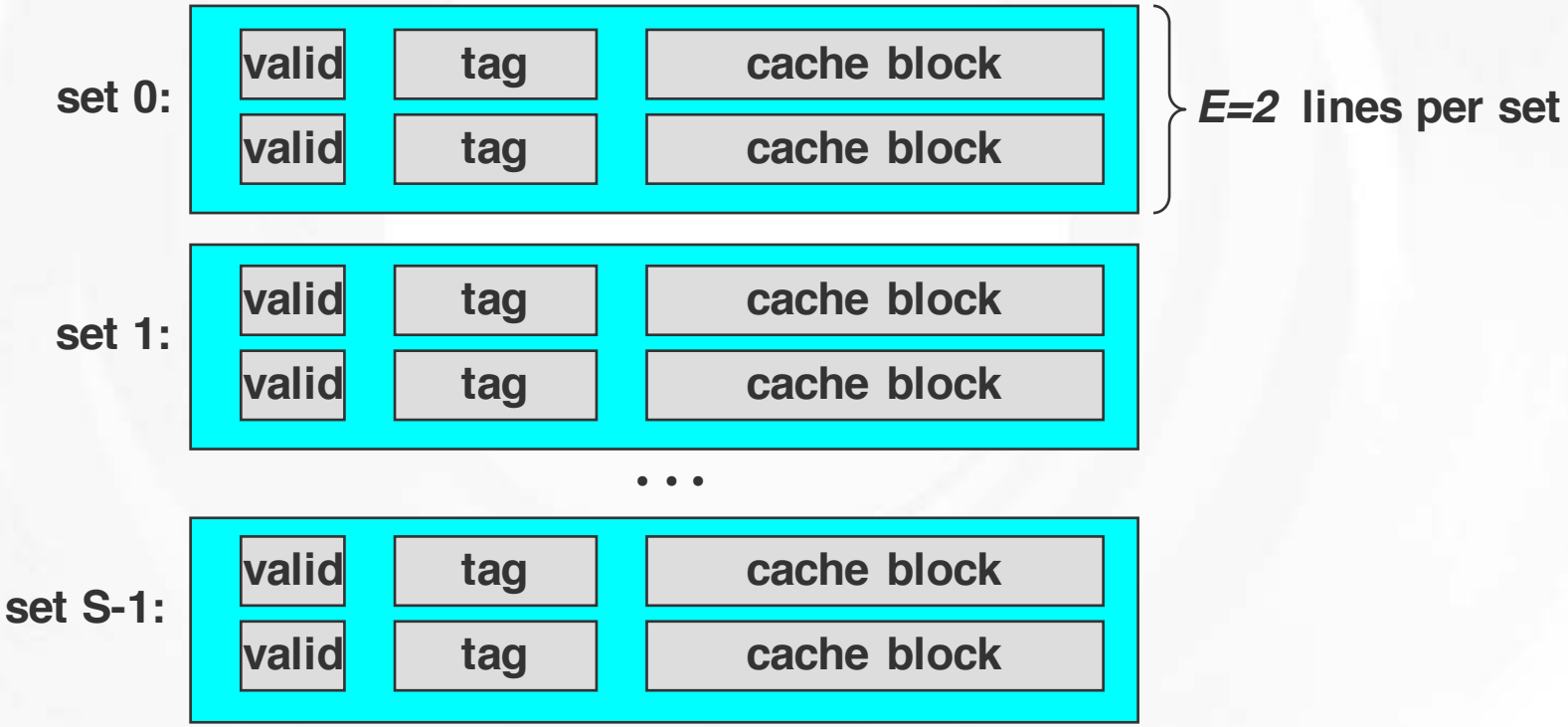
- Consecutive memory lines map to different cache lines
- Can hold C-byte region of address space in cache at one time





Lecture 7 Set Associative Caches

Characterized by more than one line per set



Lecture 7 Set Associative Caches

A description:

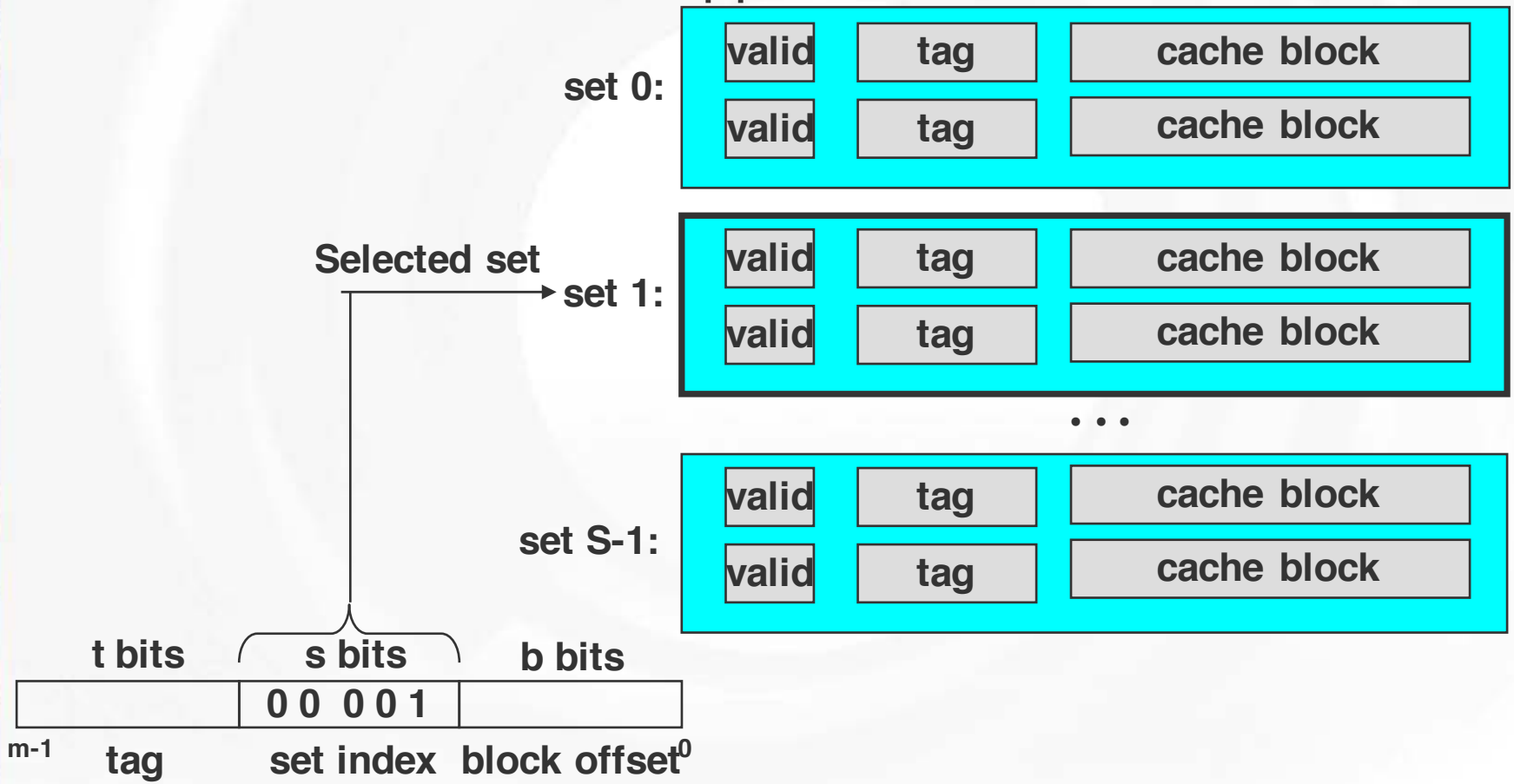
- Suppose we have a system with a CPU, a register file, an L1 cache, and a main memory.
- When the CPU executes an instruction that reads a memory word *w*, it requests the word from the L1 cache.
- If the L… the we have an L1 … xtracts ***w*** and re…
- Otherv… e CPU must wait w… of the block containing ***w*** from the main memory
- When the requested block finally arrives from memory, the L1 cache stores the block in one the its cache lines, extract word ***w*** from the stored block, and returns it to the CPU

1. **Set selection**
2. **Line matching**
3. **Word extraction**

Lecture 7 Accessing Set Associative Caches

Set selection

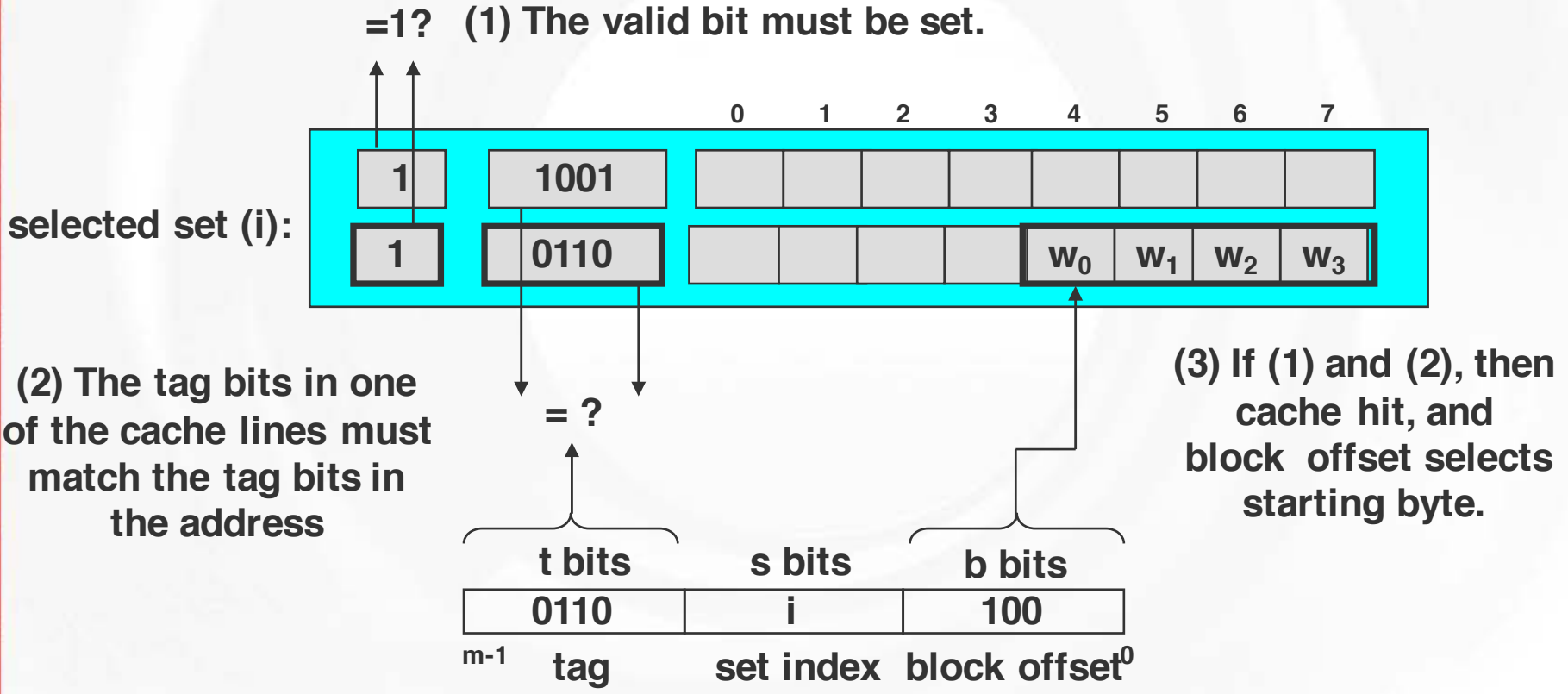
- identical to direct-mapped cache



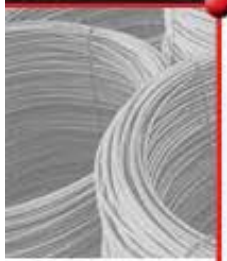
Lecture 7 Accessing Set Associative Caches

Line matching and word selection

- must compare the tag in each valid line in the selected set.



Lecture 7 Writing Cache Friendly Code





- Repeated references to variables are good (temporal locality)
- Stride-1 reference patterns are good (spatial locality)

Examples:

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

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

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

Miss rate = 1/4 = 25%

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

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

Miss rate = 100%

Lecture 7 The Memory Mountain

Read throughput (read bandwidth)

- Number of bytes read from memory per second (MB/s)

Memory mountain

- Measured read throughput as a function of spatial and temporal locality.
- Compact way to characterize memory system performance.

Lecture 7 The Memory Mountain

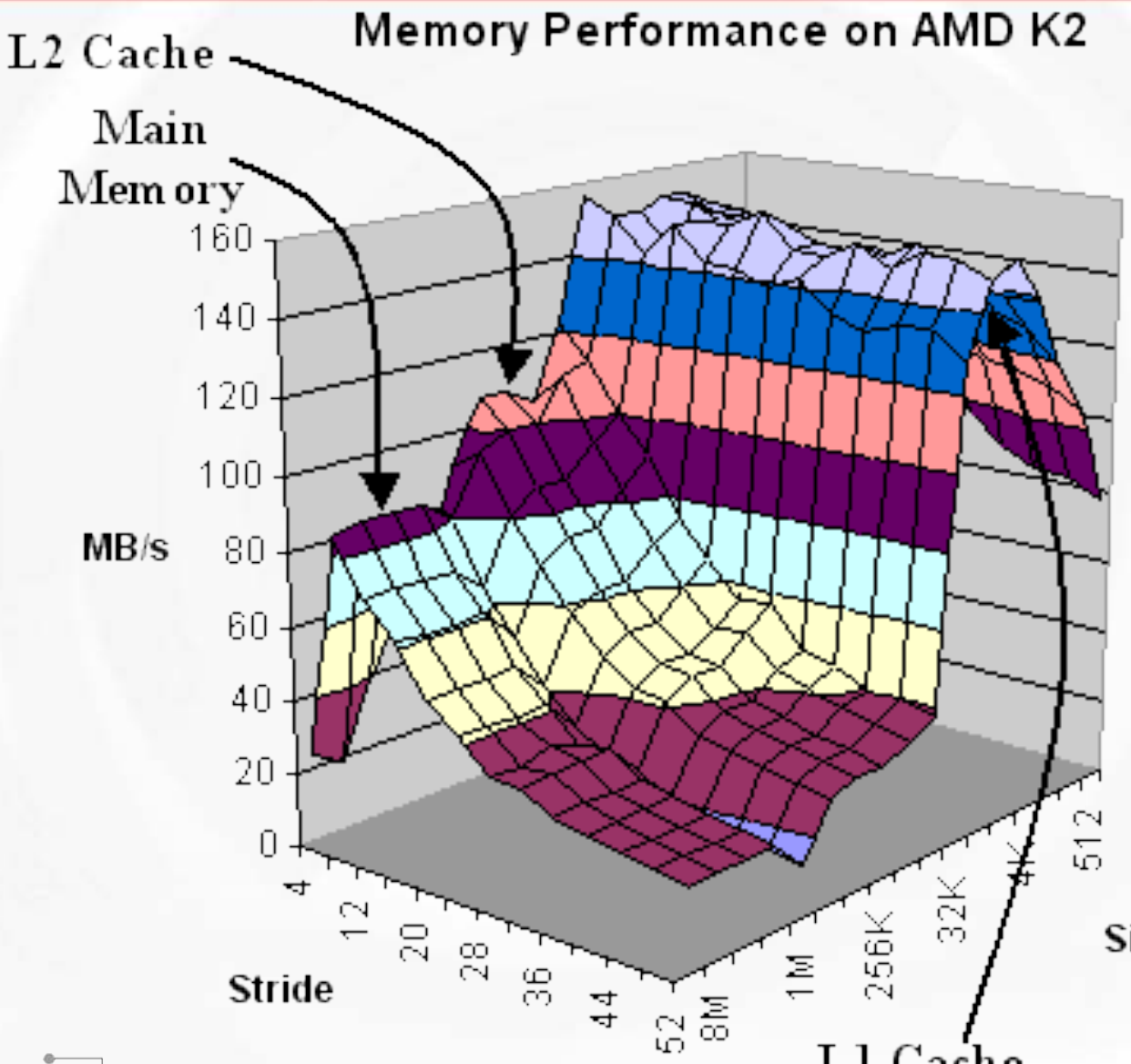
```
/* The test function */
void test(int elems, int stride) {
    int i, result = 0;
    volatile int sink;

    for (i = 0; i < elems; i += stride)
        result += data[i];
    sink = result; /* So compiler doesn't optimize away the loop */
}

/* Run test(elems, stride) and return read throughput (MB/s) */
double run(int size, int stride, double Mhz)
{
    double cycles;
    int elems = size / sizeof(int);

    test(elems, stride);                    /* warm up the cache */
    cycles = fcyc2(test, elems, stride, 0);  /* call test(elems,stride) */
    return (size / stride) / (cycles / Mhz); /* convert cycles to MB/s */
}
```

Lecture 7 The Memory Mountain

```
/* mountain.c - Generate the memory mountain. */
#define MINBYTES (1 << 10)  /* Working set size ranges from 1 KB */
#define MAXBYTES (1 << 23)  /* ... up to 8 MB */
#define MAXSTRIDE 16        /* Strides range from 1 to 16 */
#define MAXELEMS MAXBYTES/sizeof(int)

int data[MAXELEMS];         /* The array we'll be traversing */

int main()
{
    int size;         /* Working set size (in bytes) */
    int stride;       /* Stride (in array elements) */
    double Mhz;       /* Clock frequency */

    init_data(data, MAXELEMS); /* Initialize each element in data to 1 */
    Mhz = mhz(0);              /* Estimate the clock frequency */
    for (size = MAXBYTES; size >= MINBYTES; size >>= 1) {
        for (stride = 1; stride <= MAXSTRIDE; stride++)
            printf("%.1f\t", run(size, stride, Mhz));
        printf("\n");
    }
    exit(0);
}
```

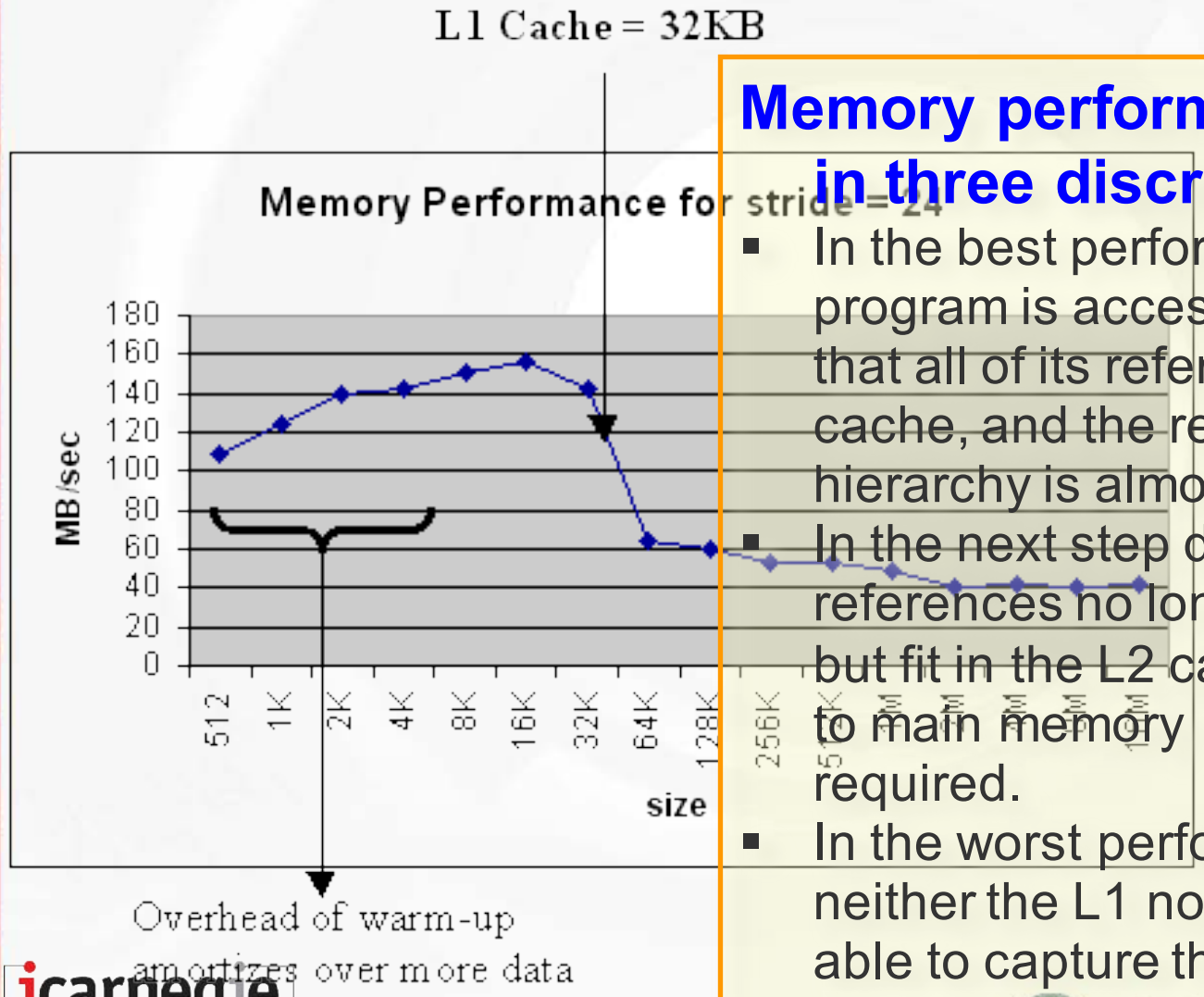
Lecture 7

The Memory Mountain



Lecture 7 A Slope of Spatial Locality

Slice through memory mountain with size=32KB

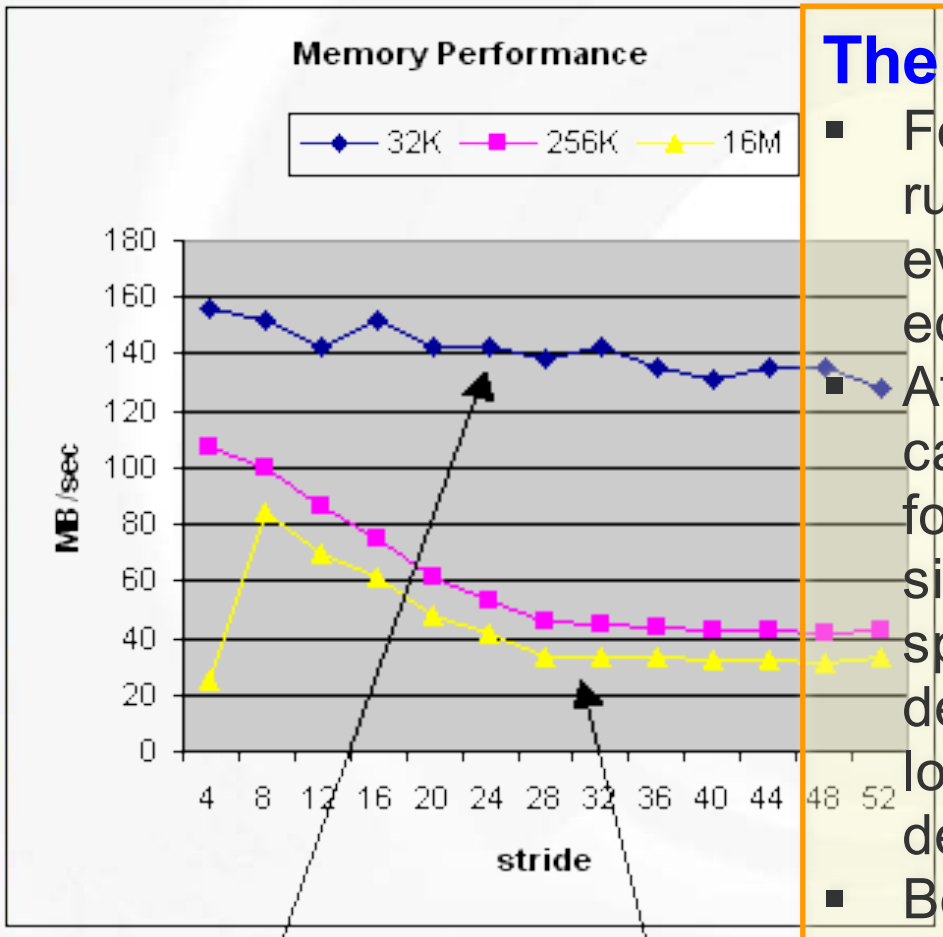


Memory performance comes in three discrete steps:

- In the best performing step, the program is accessing so little data that all of its references fit in the L1 cache, and the rest of the hierarchy is almost never required.
- In the next step down, the references no longer fit in the L1 but fit in the L2 cache, and access to main memory is almost never required.
- In the worst performing region, neither the L1 nor the L2 cache is able to capture the poor locality of the program.

Lecture 7 A Slope of Spatial Locality

Three slices from the surface of the mountain



The effects of the stride:

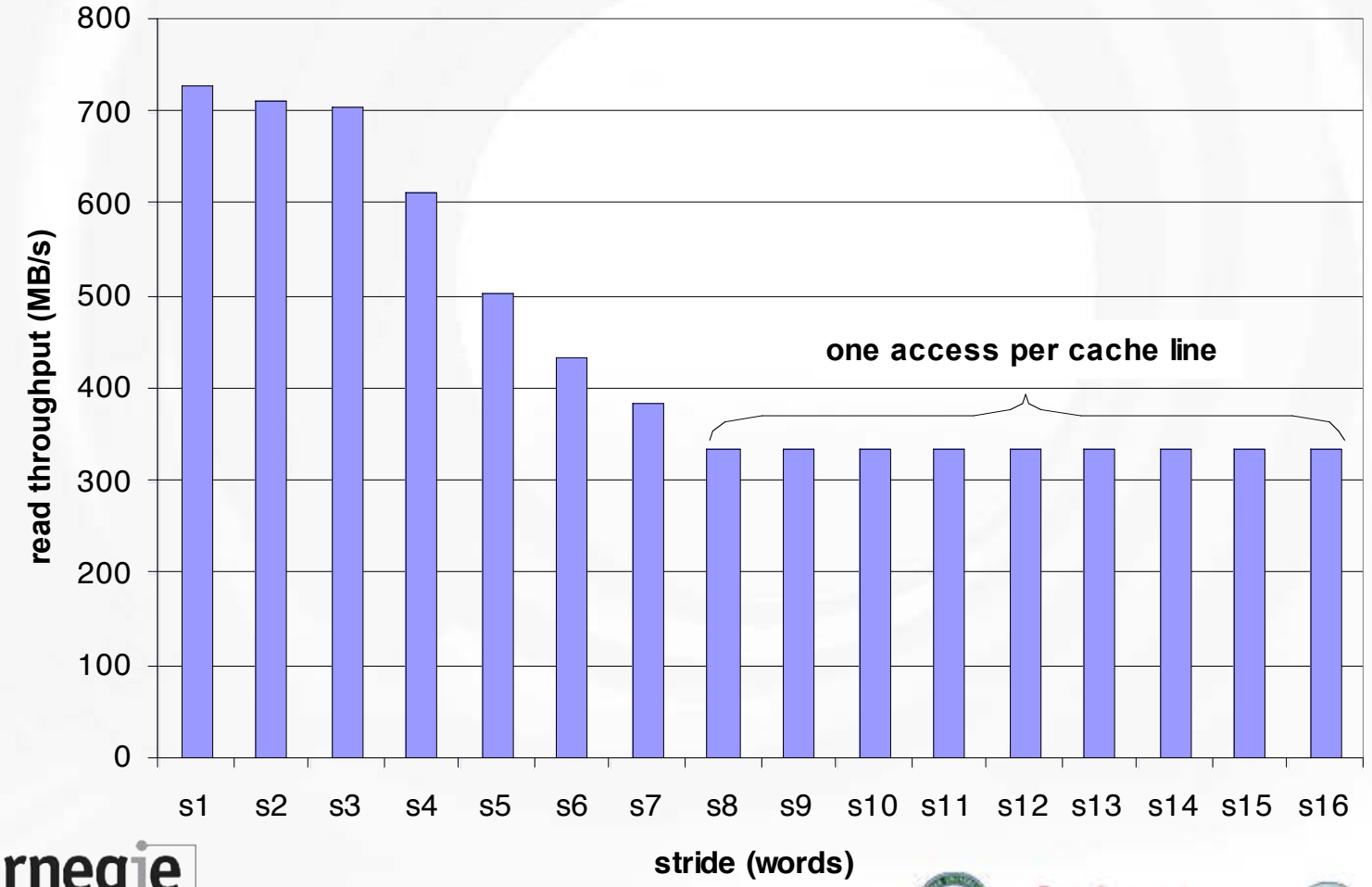
- For N = 32K, the benchmark is running fully in the L1 cache, so everything runs more or less equally fast.
- At N = 256K, the L1 is unable to capture the temporal locality, but for strides below its cache line size, it is still able to help with the spatial locality, though decreasingly so as the spatial locality of the benchmark decreases.
- Beyond the size of its cache line, however, it doesn't do any more good and the memory performance flattens out.

Lecture 7

A Slope of Spatial Locality

Slice through memory mountain with size=256KB

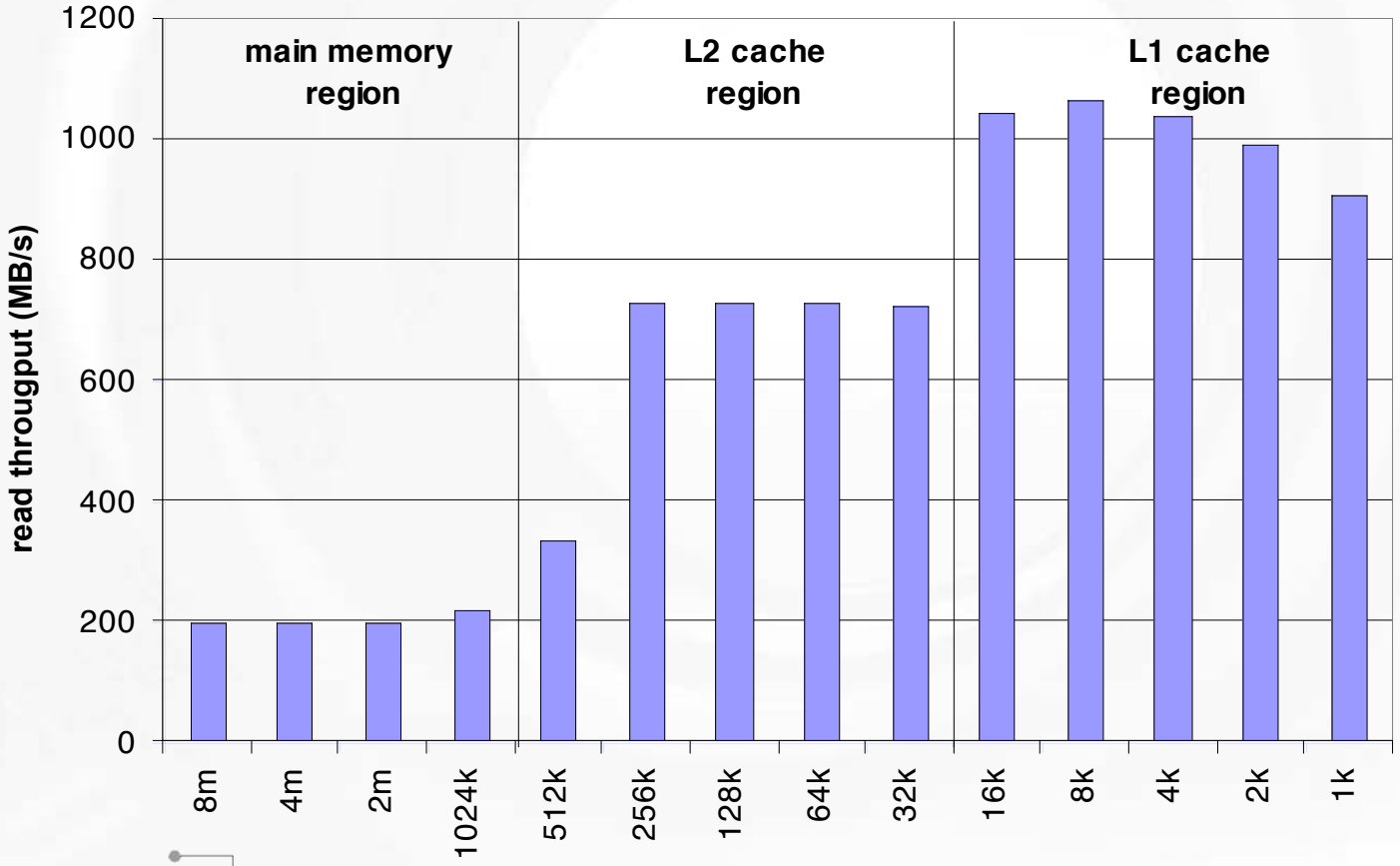
- shows cache block size.



Lecture 7 Ridges of Temporal Locality

Slice through the memory mountain with stride=1

- illuminates read throughputs of different caches and memory



Lecture 7

Cache-aware Programming

- Knowing how caches are designed helps us write better programs.
- Often, programs are designed assuming that the access time to one address or another is the same. But in fact caches make some memory references much faster than others.
- A program that may appear to be very efficient according to the usual analysis may, in fact, turn out to exhibit very poor locality of reference, and, therefore, poor memory performance.
- To be solved :::: thinking about the memory hierarchy of the computer or computers involved!
- Remark :::: Cache-aware programming is to increase the locality of a program's memory references, either spatially or temporally.

Lecture 7

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

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

```
/* 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;
  }
}
```

Lecture 7 An Example

```
int sumvec ( int v[N])
{
  int i, sum=0;
    for (i=0; i<N; i++)
      sum += v[i];
  return sum;
}
```

V[i]	i=0	i=1	i=2	i=3	i=4	i=5	i=6	i=7
hit/miss	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1[m]	2[h]	3[h]	4[h]	5[m]	6[h]	7[h]	8[h]
i=1	9[m]	10[h]	11[h]	12[h]	13[m]	14[h]	15[h]	16[h]
i=2	17[m]	18[h]	19[h]	20[h]	21[m]	22[h]	23[h]	24[h]
i=3	25[m]	26[h]	27[h]	28[h]	29[m]	30[h]	31[h]	32[h]

```
int sumarrayrows ( int a[M][N])
{
  int i,j; sum=0;
    for (i=0; i<M; i++)
      for (j=0; j<N; j++
        sum += a[i][j];
  return sum;
}
```

Lecture 7 An Example

a[i][j]	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
i=0	1[m]	2[m]	3[m]	4[m]	5[m]	6[m]	7[m]	8[m]
i=1	9[m]	10[m]	11[m]	12[m]	13[m]	14[m]	15[m]	16[m]
i=2	17[m]	18[m]	19[m]	20[m]	21[m]	22[m]	23[m]	24[m]
i=3	25[m]	26[m]	27[m]	28[m]	29[m]	30[m]	31[m]	32[m]

Inversion of *i* & *j*

```
int sumarrayrows ( int a[M][N])
{
  int i,j; sum=0;
    for (j=0; i<N; j++)
      for (i=0; i<M; i++ )
          sum += a[i][j];
  return sum;
}
```

Lecture 7 An Example

```
typedef int array[2][2]
void transposel(array dst, array src)
{
  int i,j;
  for (i=0; i<2; i++)
      for (j=0; j<2; j++){
         dst[j][i]= src[i][j];
      }
  return sum;
}
```

dst array

	Col 0	Col 1
Row 0	m	m
Row 1	m	m

src array

	Col 0	Col 1
Row 0	m	m
Row 1	m	h

Assume:

- sizeof (int) = 4 ;
- L1 data cache, direct-mapped, with a block size of 8 bytes;
- The cache has a total size of 16 data bytes, initially empty;

Lecture 7

An Example

```
struct algae_position {
   int x;
   int y;
 };

struct algae_position grid [16][16];
int total_x = 0; total_y = 0;
int i,j;

for (i=0; i<16; i++) {
      for (j=0; j<16; j++){
         total_x += grid[i][j].x;
      }
}
  for (i=0; i<16; i++) {
      for (j=0; j<16; j++){
         total_y += grid[i][j].y;
      }
}
```

Questions:

- What's the total number of reads?
- What's the total number of reads that miss in the cache?
- What's the miss rate?

Assume:

- sizeof (int) = 4 ;
- direct-mapped, with a block size of 16bytes;
- The cache has a total size of 1024 data bytes;

Lecture 7 An Example

```
struct algae_position {
   int x;
   int y;
 };

struct algae_position grid [16][16];
int total_x = 0; total_y = 0;
int i,j;

for (i=0; i<16; i++) {
      for (j=0; j<16; j++){
         total_x += grid[i][j].x;
         total_y += grid[i][j].y;
      }
}
```

Questions:

- What's the total number of reads?
- What's the total number of reads that miss in the cache?
- What's the miss rate?
- What would the miss rate be if the cache were twice as big?

Assume:

- sizeof (int) = 4 ;
- direct-mapped, with a block size of 16bytes;
- The cache has a total size of 1024 data bytes;

Lecture 7

A Practical Example





Perf-sol-4-cache.html

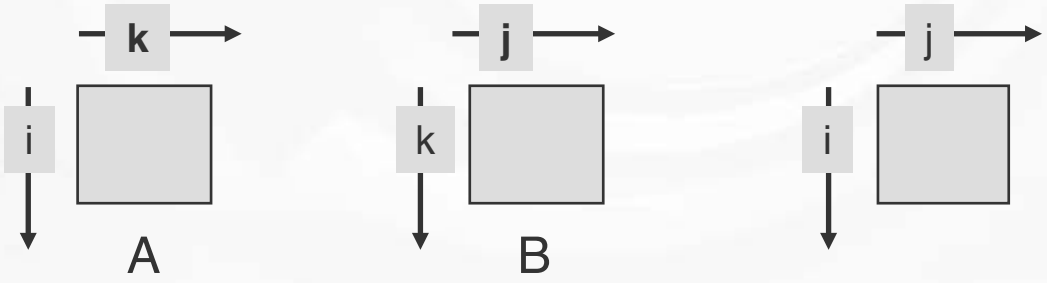
Lecture 7 Miss Rate Analysis for Matrix Multiply

Assume:

- Line size = 32Byte (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



Lecture 7

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:

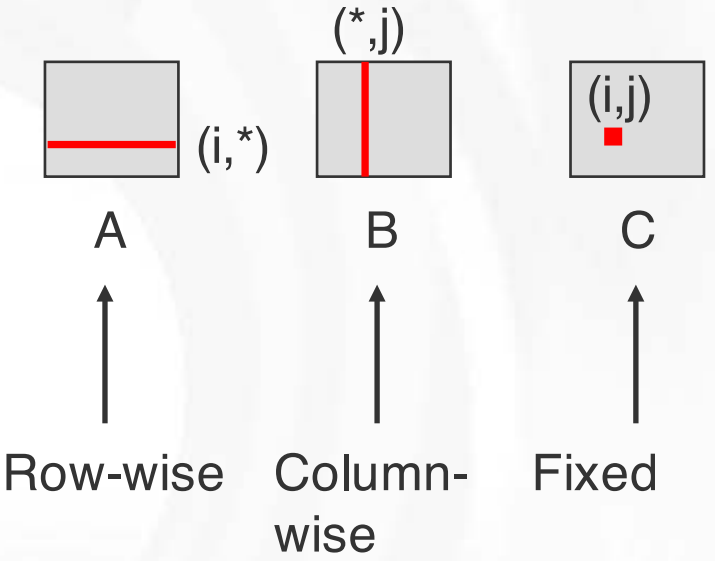
-
```
for (i = 0; i < n; i++)
  sum += a[i][0];
```
- accesses distant elements
- no spatial locality!
  - **compulsory miss rate = 1 (i.e. 100%)**

Lecture 7

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:



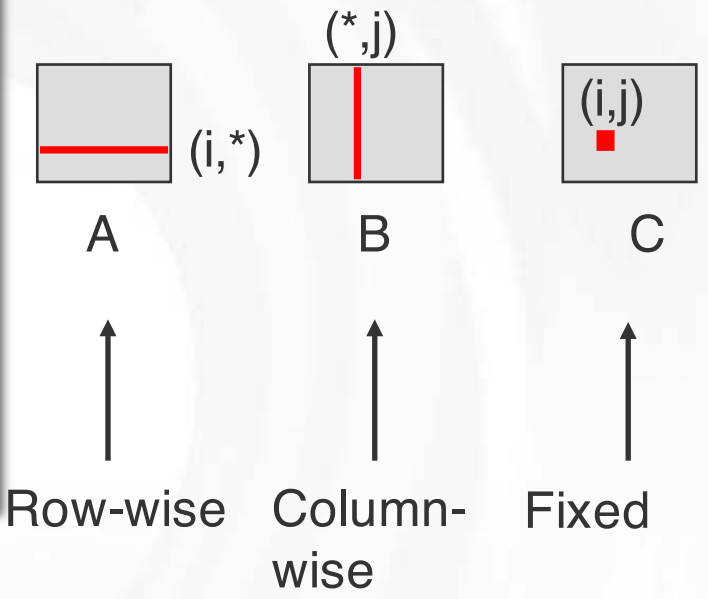
Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Lecture 7 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
  }
}
```

Inner loop:

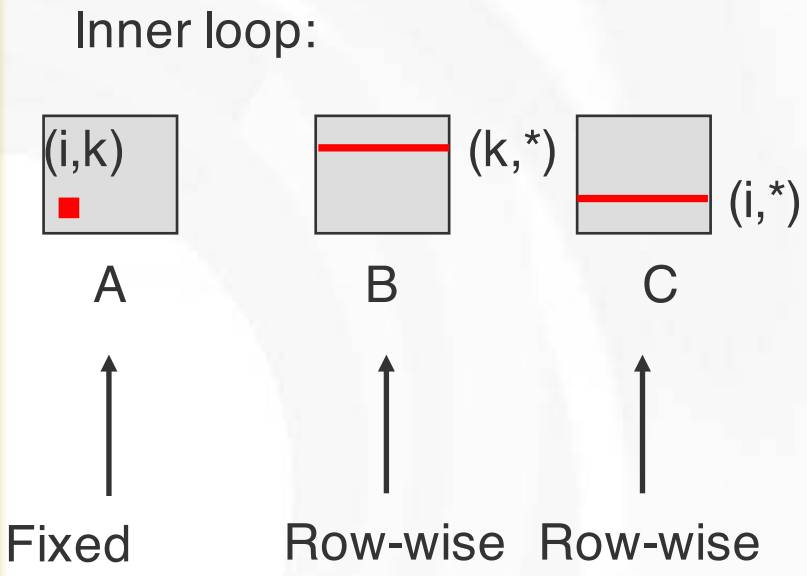


Misses per Inner Loop Iteration:

A	B	C
0.25	1.0	0.0

Lecture 7 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];
  }
}
```



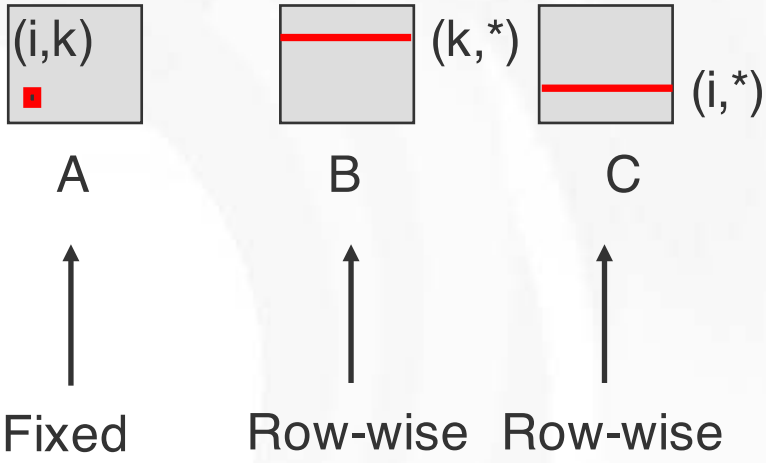
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Lecture 7 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:



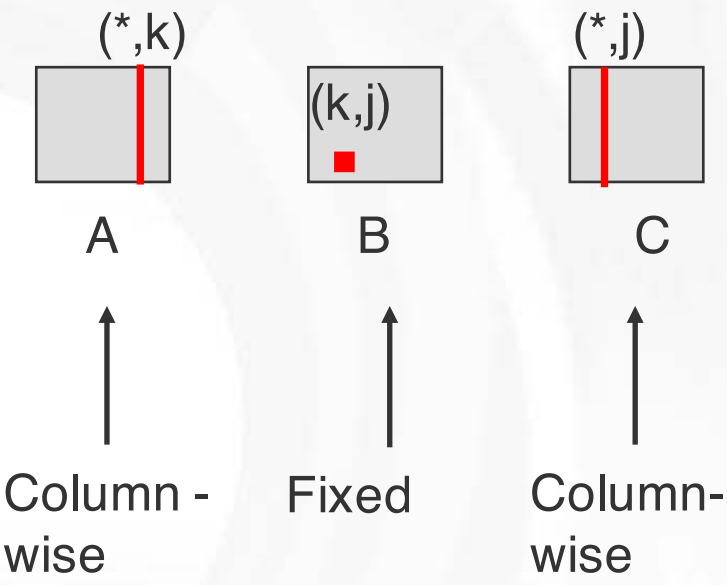
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

Lecture 7 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:

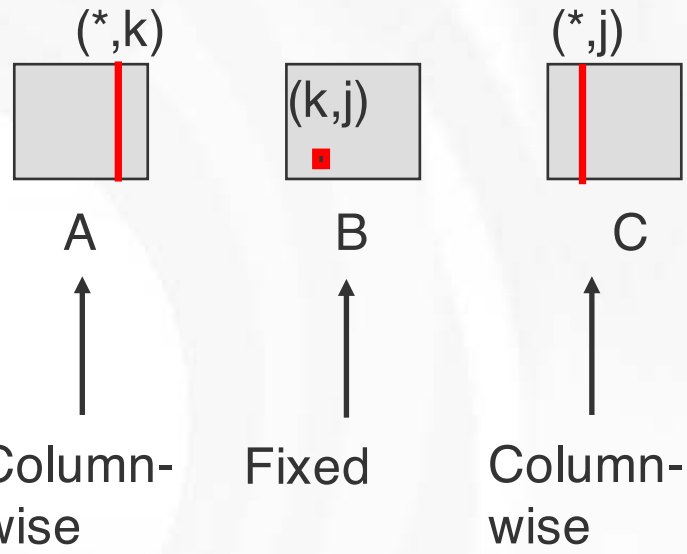
A	B	C
1.0	0.0	1.0

Lecture 7

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

Lecture 7 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];
  }
}
```

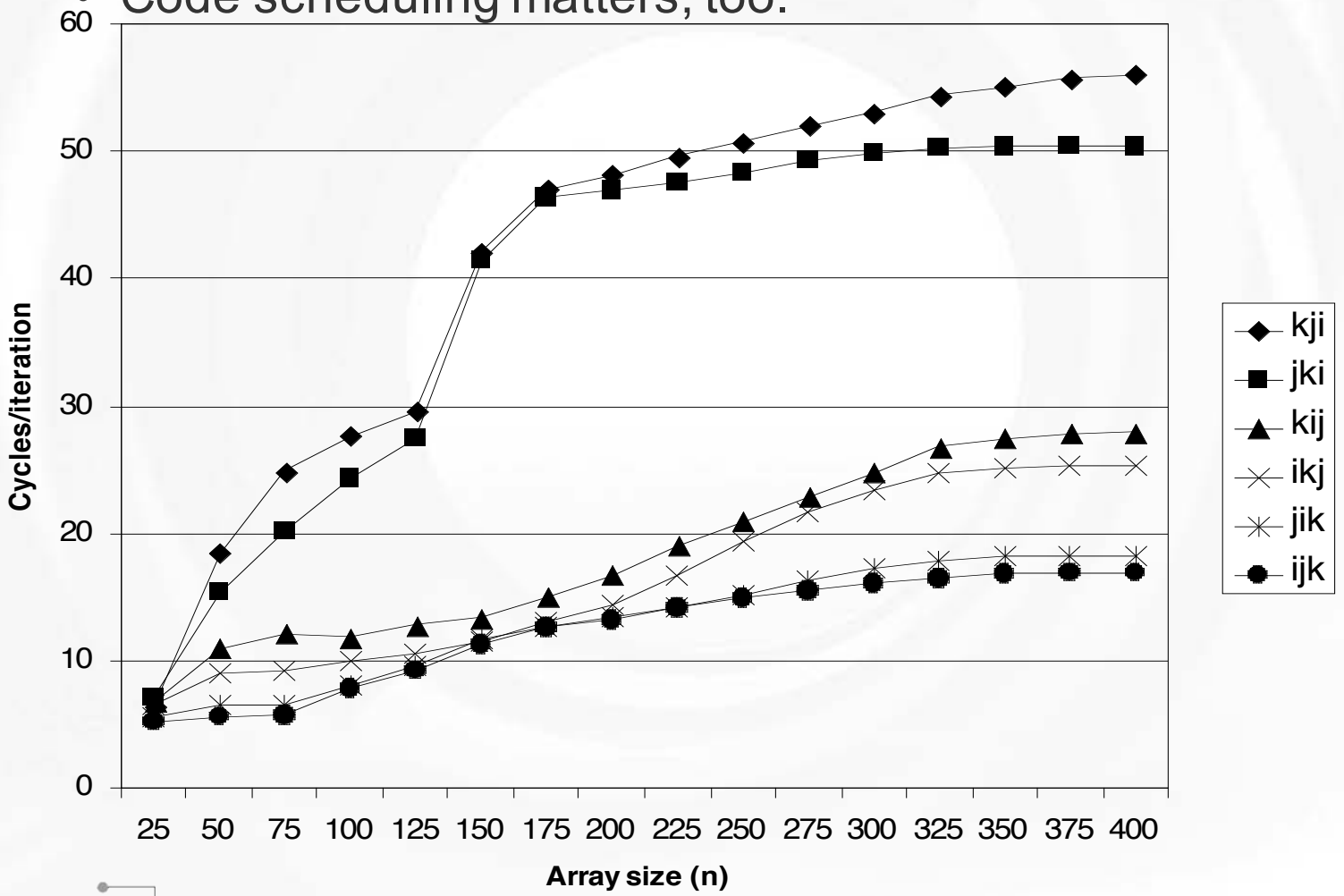
jki (& kji):
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
  }
}
```

Lecture 7 Pentium Matrix Multiply Performance

Miss rates are helpful but not perfect predictors.

- Code scheduling matters, too.



Lecture 7 Improving Temporal Locality by Blocking

Example: Blocked matrix multiplication

- "block" (in this context) does not mean "cache block".
- Instead, it mean a sub-block within the matrix.
- Example: N = 8; sub-block size = 4

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Key idea: Sub-blocks (i.e., $\mathbf{A_{xy}}$) can be treated just like scalars.

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Lecture 7

Blocked Matrix Multiply (bijk)

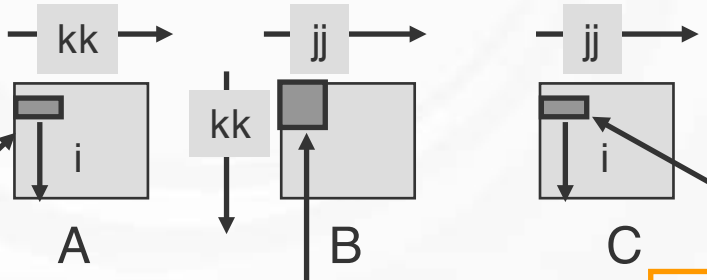
```
for (jj=0; jj<n; jj+=bsize) {
  for (i=0; i<n; i++)
    for (j=jj; j < min(jj+bsize,n); j++)
      c[i][j] = 0.0;
  for (kk=0; kk<n; kk+=bsize) {
    for (i=0; i<n; i++) {
      for (j=jj; j < min(jj+bsize,n); j++) {
        sum = 0.0
        for (k=kk; k < min(kk+bsize,n); k++) {
          sum +=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

Lecture 7 Blocked Matrix Multiply Analysis

- Innermost loop pair multiplies a *1 X bsize* sliver of *A* by a *bsize X bsize* block of *B* and accumulates into *1 X bsize* sliver of *C*
- Loop over *i* steps through *n* row slivers of *A* & *C*, using same *B*

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

Innermost Loop Pair



row sliver accessed ***bsize*** times

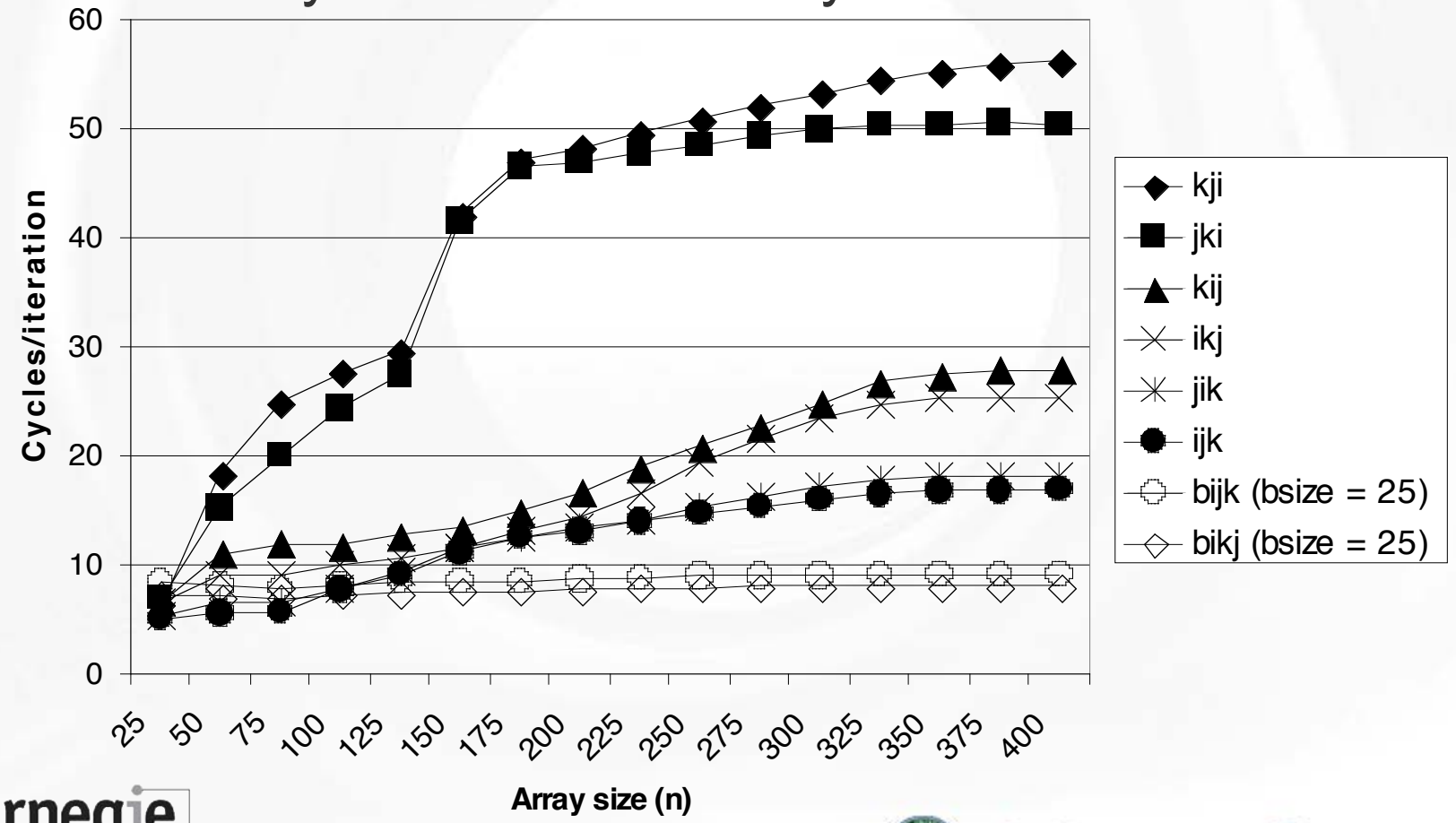
block reused ***n*** times in succession

Update successive elements of sliver

Lecture 7 Pentium Blocked Matrix Multiply Performance

Blocking (bijk and bikj) improves performance by a factor of two over unblocked versions (ijk and jik)

- relatively insensitive to array size.



Lecture 7 Concluding Observations

Programmer can optimize for cache performance

- How data structures are organized
- How data are accessed
  - Nested loop structure
  - Blocking is a general technique

All systems favor “cache friendly code”

- Getting absolute optimum performance is very platform specific
  - Cache sizes, line sizes, associativities, etc.
- Can get most of the advantage with generic code
  - Keep working set reasonably small (temporal locality)
  - Use small strides (spatial locality)

SSD
6

icarnegie
curriculum powered by Carnegie Mellon

Lecture 7 (2/2 cont.)

"640k ought to be enough for anybody.", Bill Gates, 1981

武漢大學

国际软件学院

Lecture 7

Motivations for Virtual Memory

Use Physical DRAM as a Cache for the Disk

- Address space of a process can exceed physical memory size
- Sum of address spaces of multiple processes can exceed physical memory

Simplify Memory Management

- Multiple processes resident in main memory.
  - Each process with its own address space
- Only “active” code and data is actually in memory
  - Allocate more memory to process as needed.

Provide Protection

- One process can’t interfere with another.
  - because they operate in different address spaces.
- User process cannot access privileged information
  - different sections of address spaces have different permissions.

Lecture 7 Motivation #1: DRAM a "Cache" for Disk

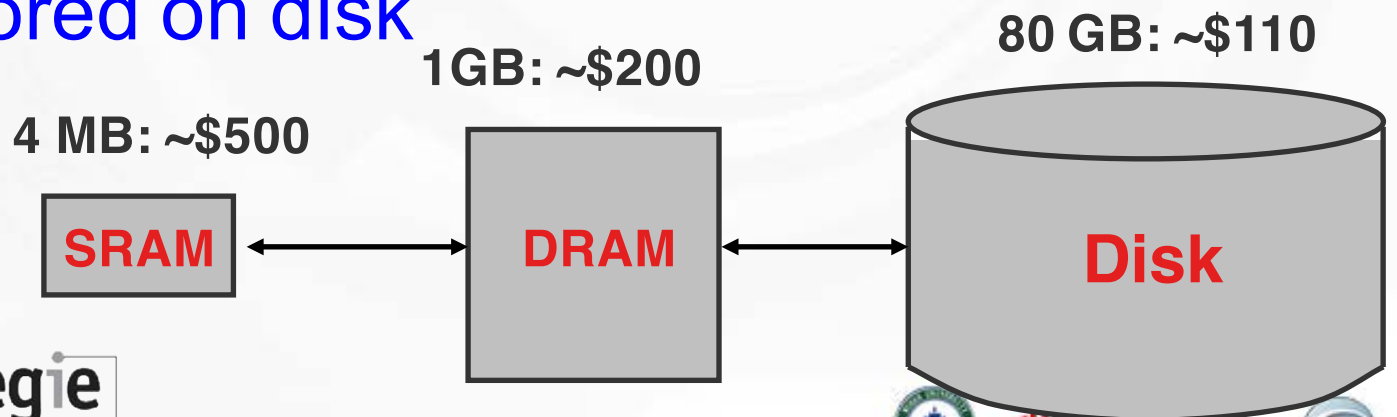
Full address space is quite large:

- 32-bit addresses: ~4,000,000,000 (4 billion) bytes
- 64-bit addresses: ~16,000,000,000,000,000,000 (16 quintillion) bytes

Disk storage is ~300X cheaper than DRAM storage

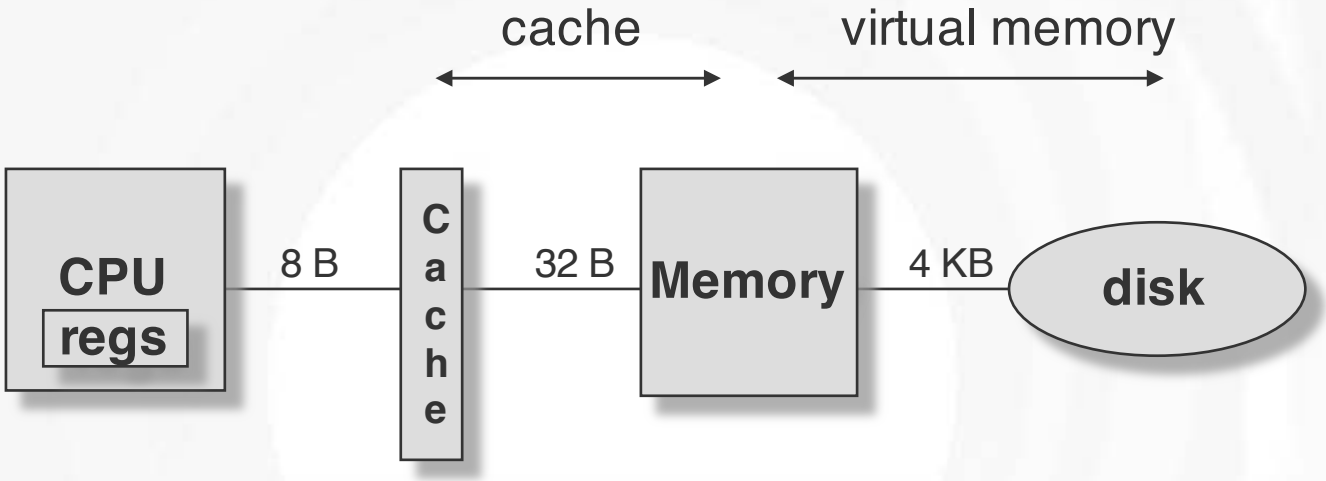
- 80 GB of DRAM: ~ $33,000
- 80 GB of disk: ~ $110

To access large amounts of data in a cost-effective manner, the bulk of the data must be stored on disk



Lecture 7

Levels in Memory Hierarchy



	Register	Cache	Memory	Disk Memory
size:	32 B	32 KB-4MB	1024 MB	100 GB
speed:	1 ns	2 ns	30 ns	8 ms
$/Mbyte:		$125/MB	$0.20/MB	$0.001/MB
line size:	8 B	32 B	4 KB	

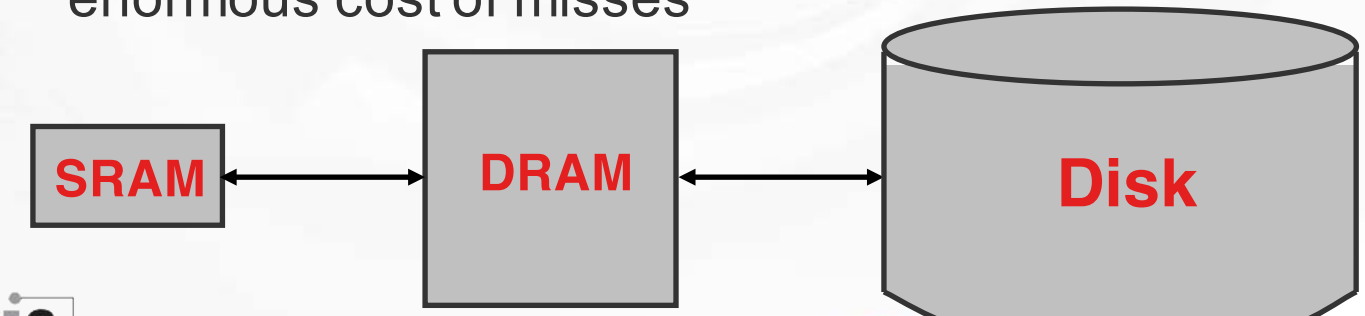
larger, slower, cheaper

Lecture 7

DRAM vs. SRAM as a "Cache"

DRAM vs. disk is more extreme than SRAM vs. DRAM

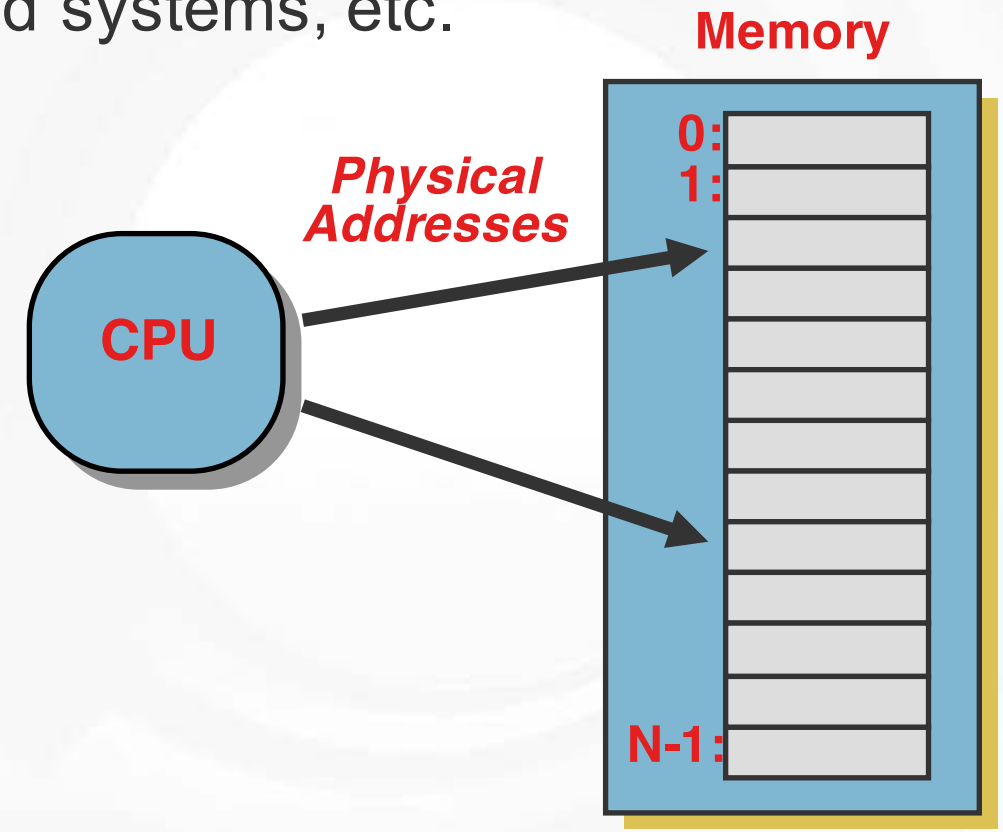
- Access latencies:
  - DRAM ~10X slower than SRAM
  - Disk ~**100,000X** slower than DRAM
- Importance of exploiting spatial locality:
  - First byte is ~**100,000X** slower than successive bytes on disk
    - vs. ~4X improvement for page-mode vs. regular accesses to DRAM
- Bottom line:
  - Design decisions made for DRAM caches driven by enormous cost of misses



Lecture 7 A System with Physical Memory Only

Examples:

- most Cray machines, early PCs, nearly all embedded systems, etc.

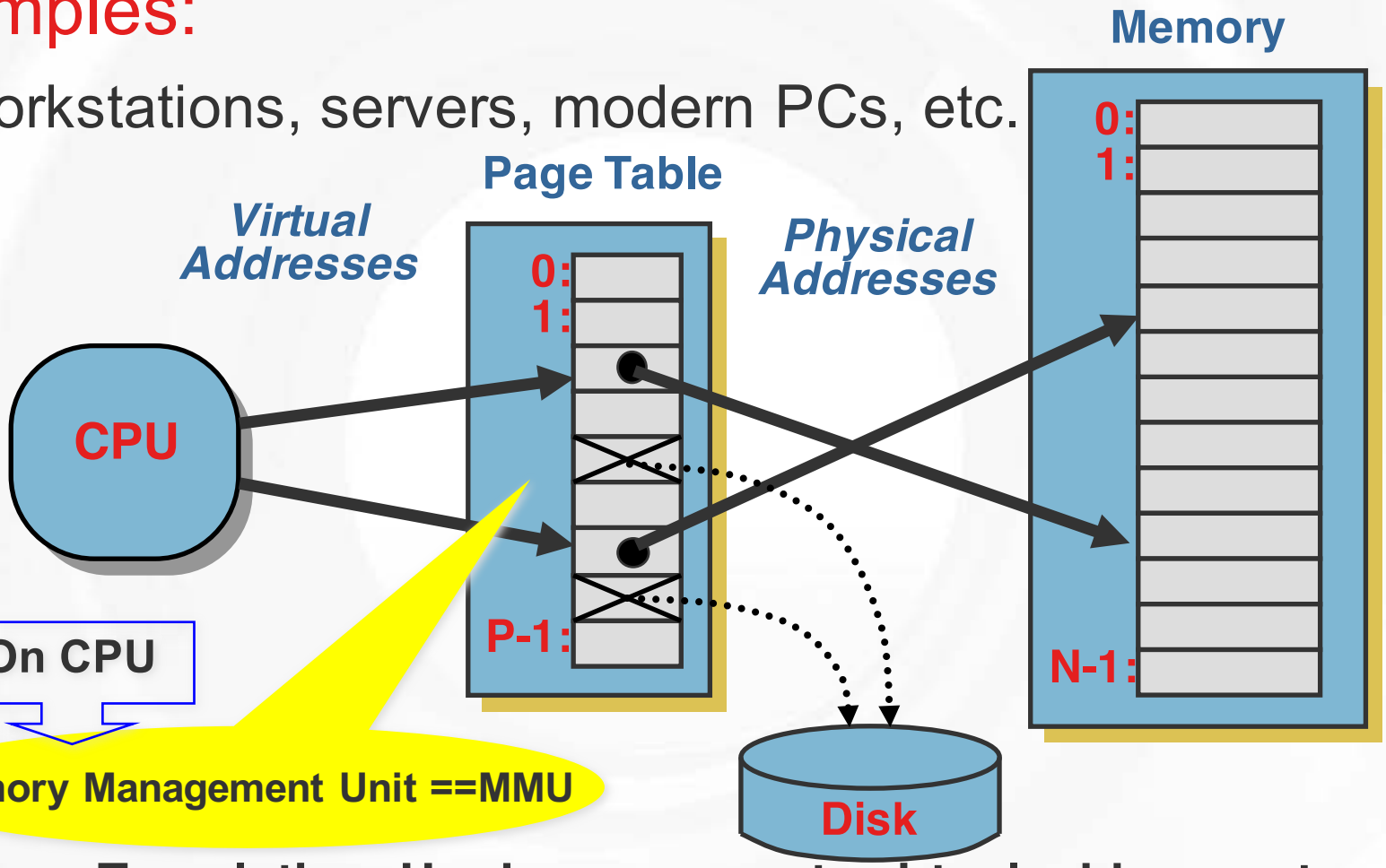


- **Addresses generated by the CPU correspond directly to bytes in physical memory**

Lecture 7 A System with Virtual Memory

Examples:

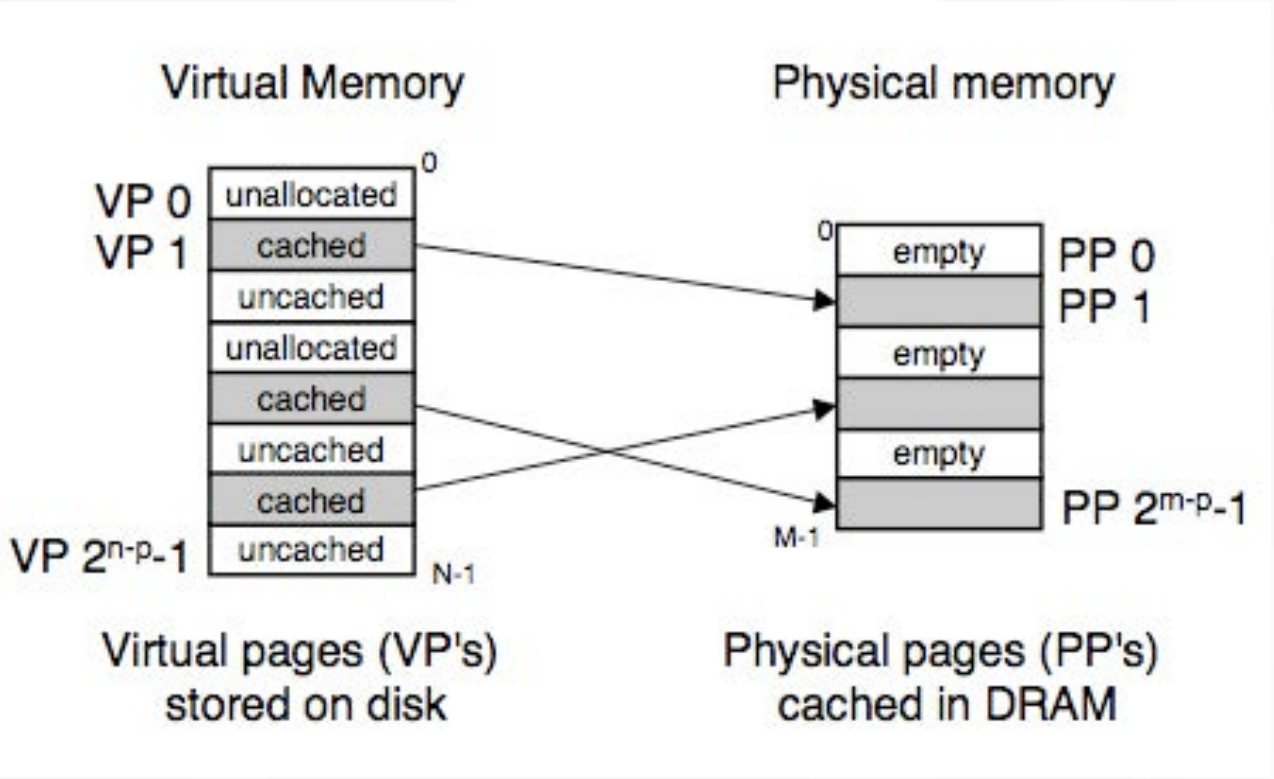
- workstations, servers, modern PCs, etc.



- **Address Translation: Hardware converts virtual addresses to physical addresses via OS-managed lookup table (page table)**

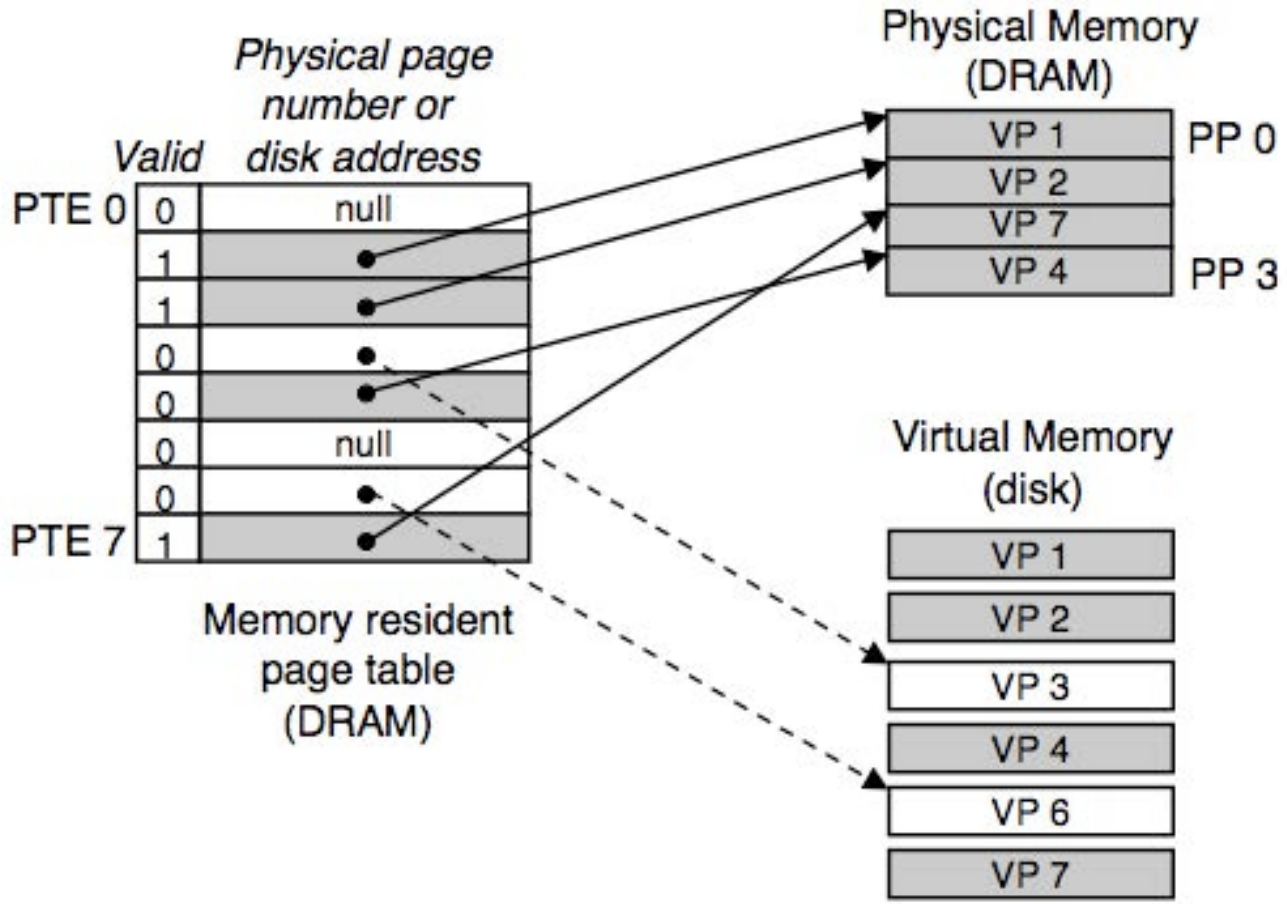
Lecture 7 A System with Virtual Memory

How a VM system uses main memory as a cache



Lecture 7 A System with Virtual Memory

Page table

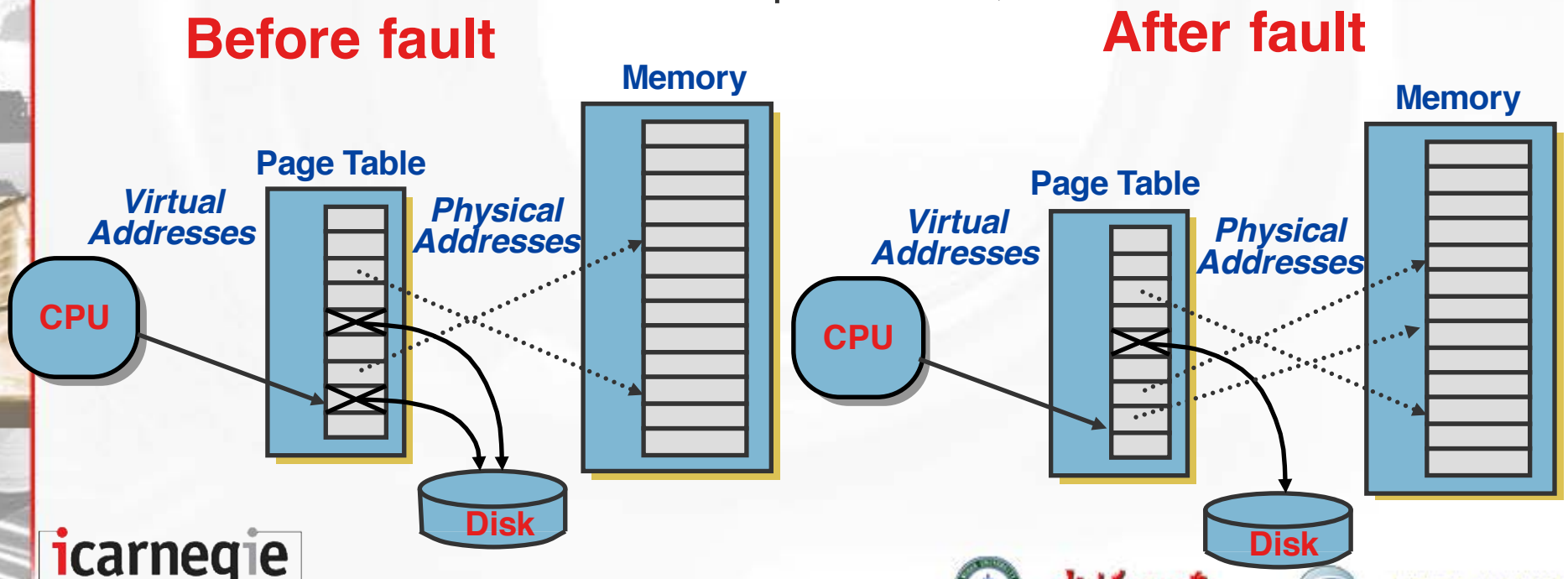


Lecture 7

Page Faults (like "Cache Misses")

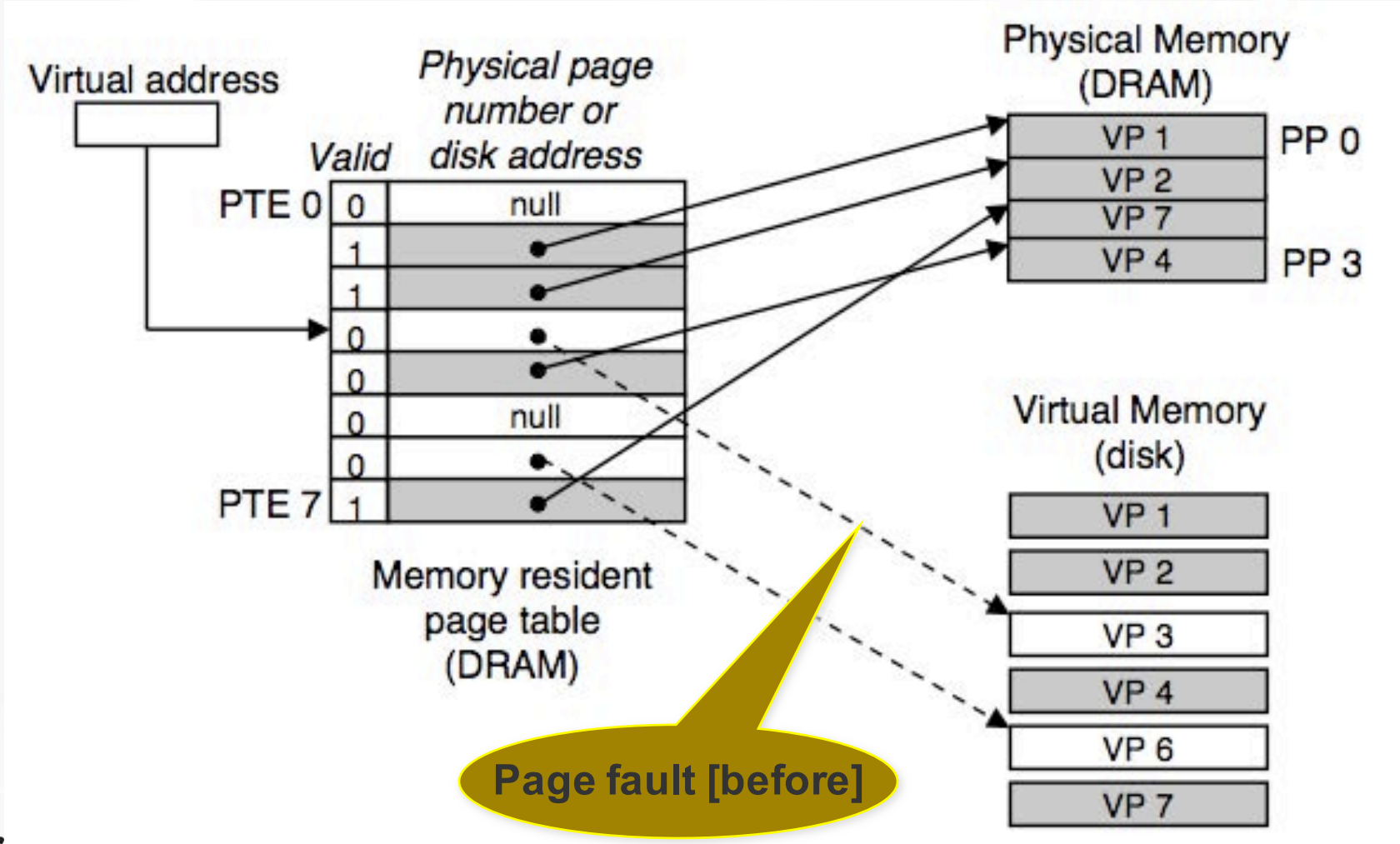
What if an object is on disk rather than in memory?

- Page table entry indicates virtual address not in memory
- OS exception handler invoked to move data from disk into memory
  - current process suspends, others can resume
  - OS has full control over placement, etc.



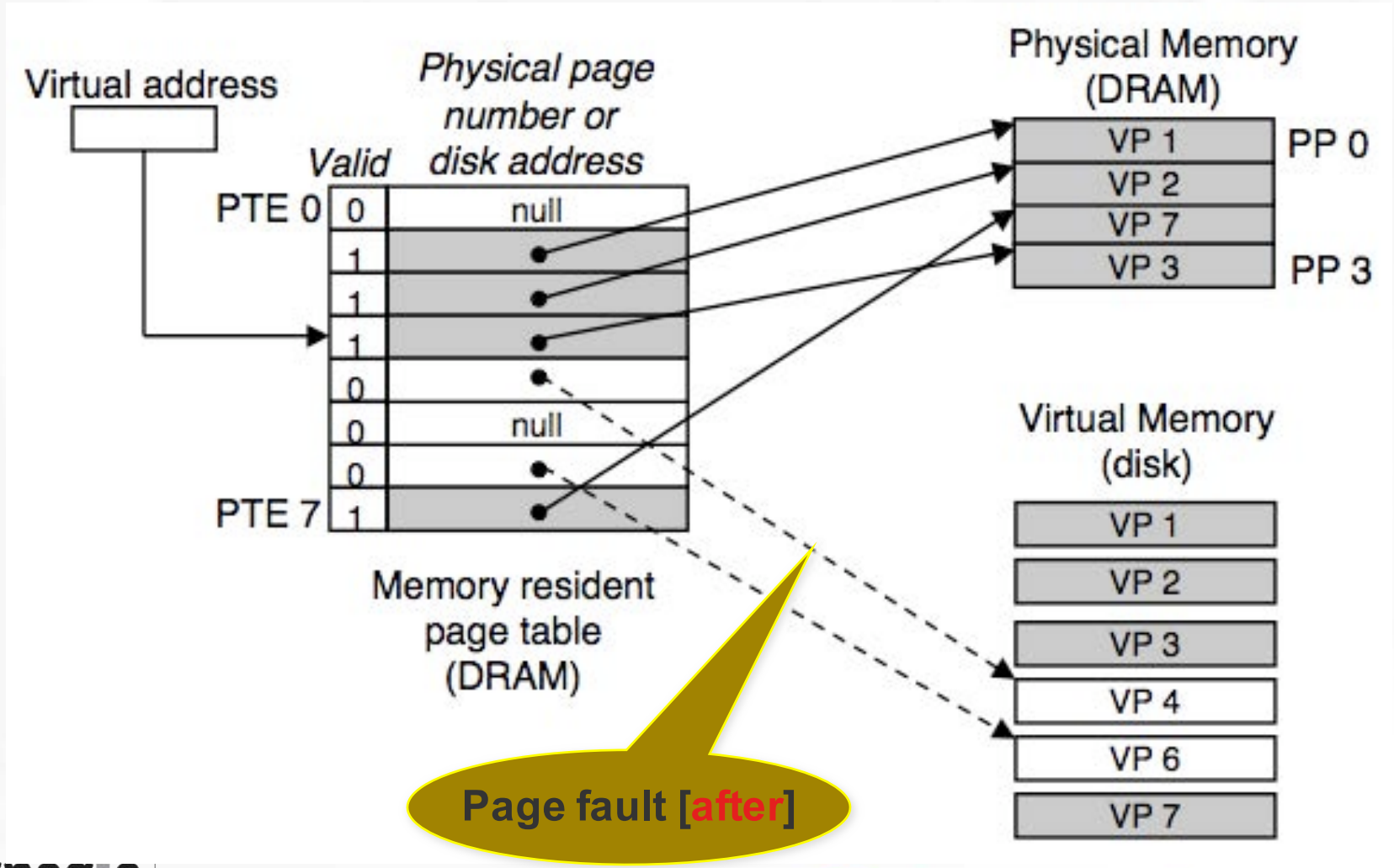
Lecture 7 Page Faults (like "Cache Misses")

What if an object is on disk rather than in memory?



Lecture 7 Page Faults (like "Cache Misses")

What if an object is on disk rather than in memory?



Lecture 7 Locality Again

Locality to the Rescue Again

Usually, total number of pages that programs reference during an entire run might exceed the total size of physical memory

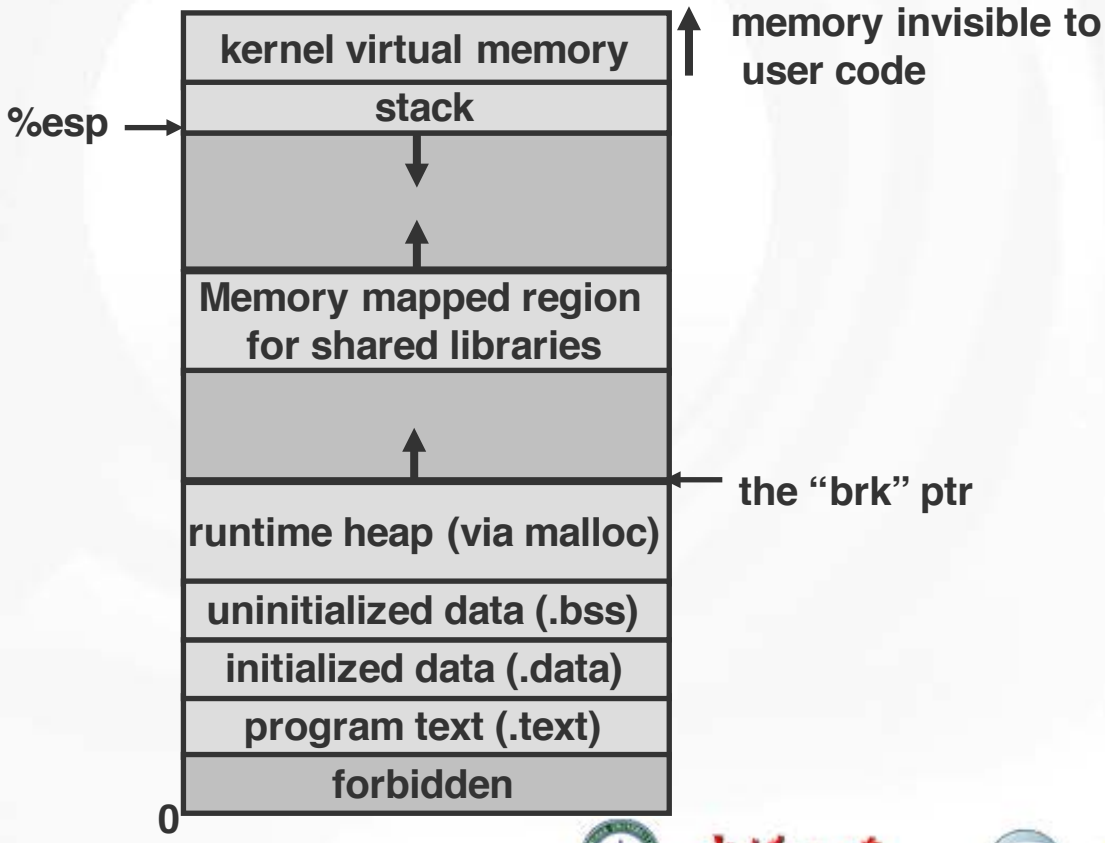
The principle of locality promises that at any point in time they will tend to work on a smaller set of *active pages* known as the *working set* or *resident set*

Lecture 7 Motivation #2: Memory Management

Multiple processes can reside in physical memory.

How do we resolve address conflicts?

- what if two processes access something at the same address?



Linux/x86 process memory image

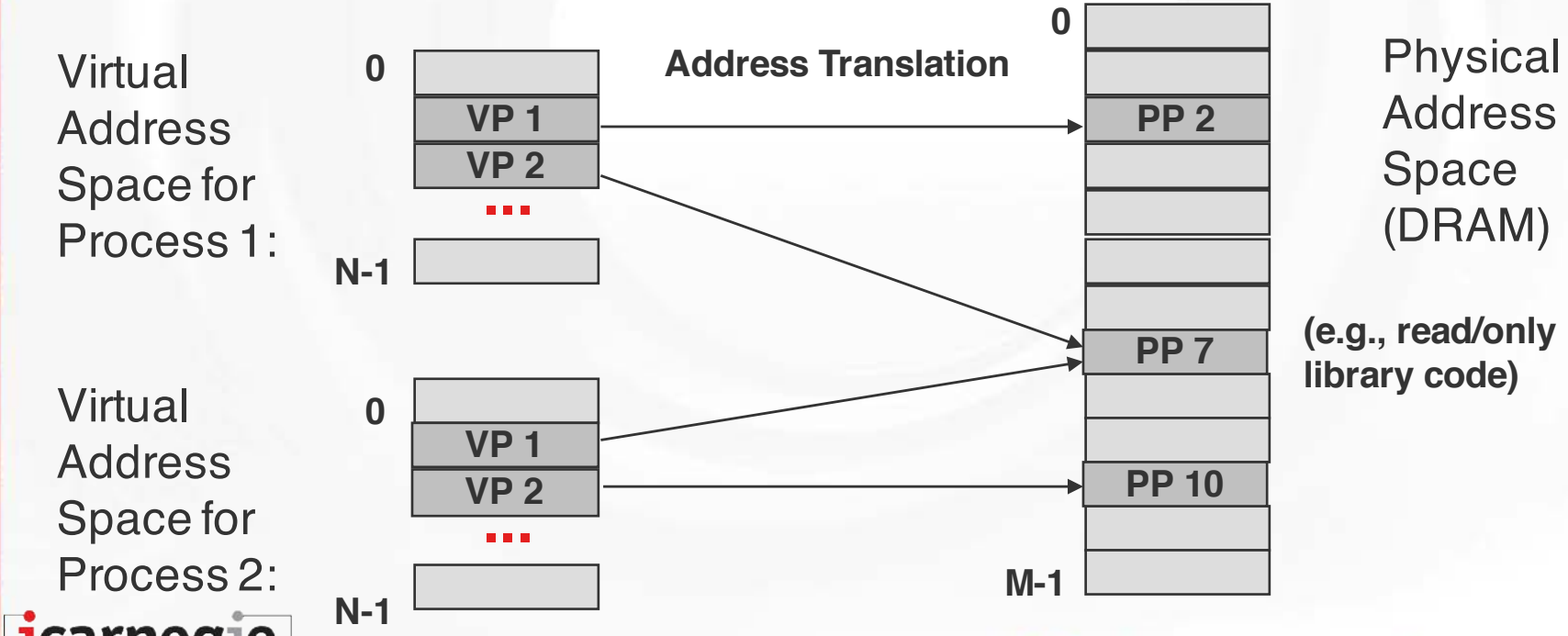
Lecture 7 Solution: Separate Virt. Addr. Spaces

Virtual and physical address spaces divided into equal-sized blocks

- blocks are called “pages” (both virtual and physical)

Each process has its own virtual address space

- operating system controls how virtual pages as assigned to physical memory

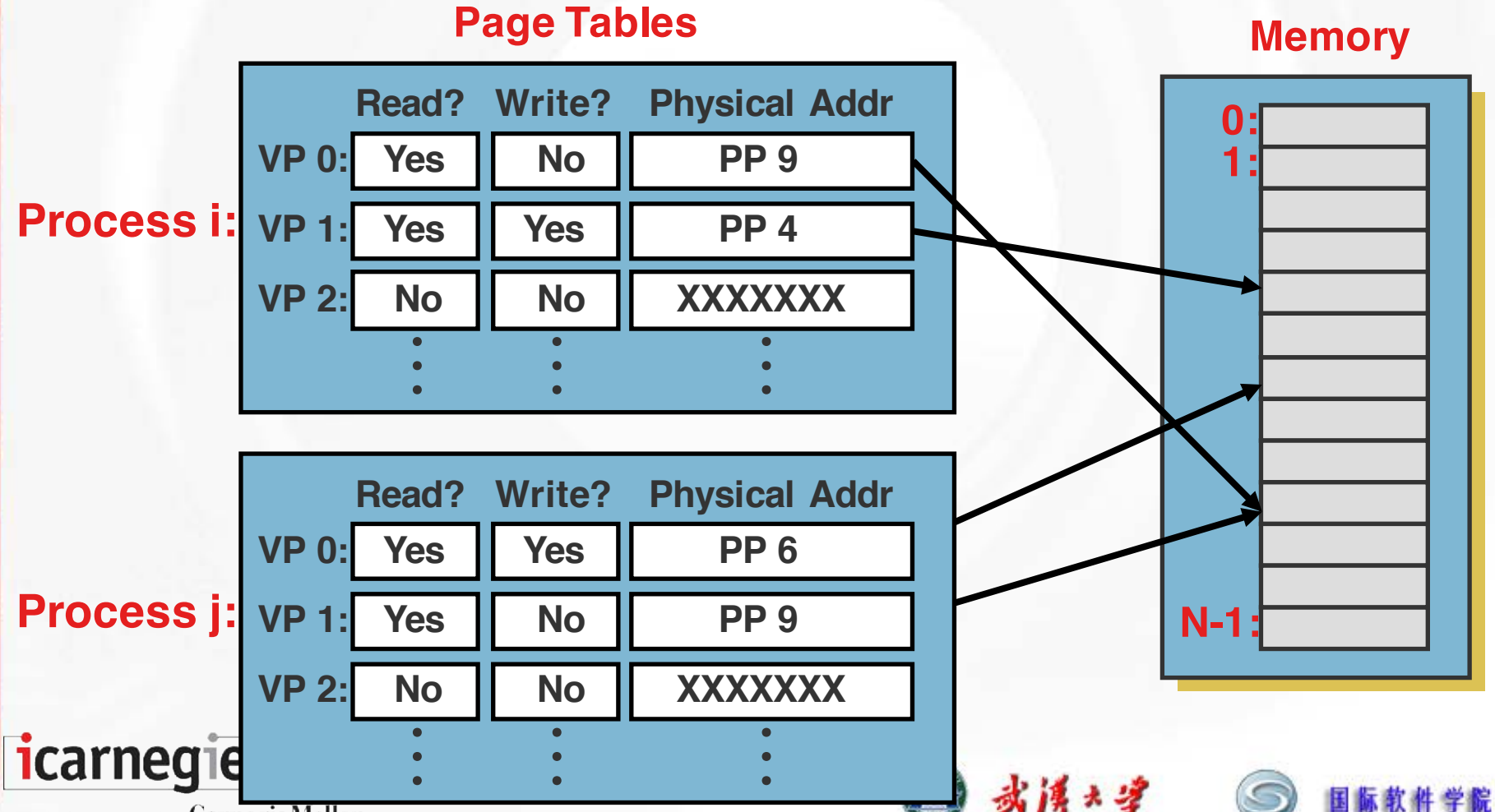


Lecture 7

Motivation #3: Protection

Page table entry contains access rights information

- hardware enforces this protection (trap into OS if violation occurs)



Lecture 7

End of This Lecture

Excise 5

Cache Lab: Improving Program Locality