Cache Memories

18-213/18-613: Introduction to Computer Systems
10th Lecture, June 6th, 2023

Announcements

- **Lab 3 (attacklab) due Thursday**

- **Lab 4 (cachelab) out Thursday**
    - But will be available tomorrow for those who want it early

- **Homework continues on the usual schedule**

- **Looking ahead: Midterm is not next week, but the week after**
    - Take-home
    - Counts 25% toward midterm grade
    - Counts as a 2x homework toward final grade

Today

- **Cache memory organization and operation**
- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality
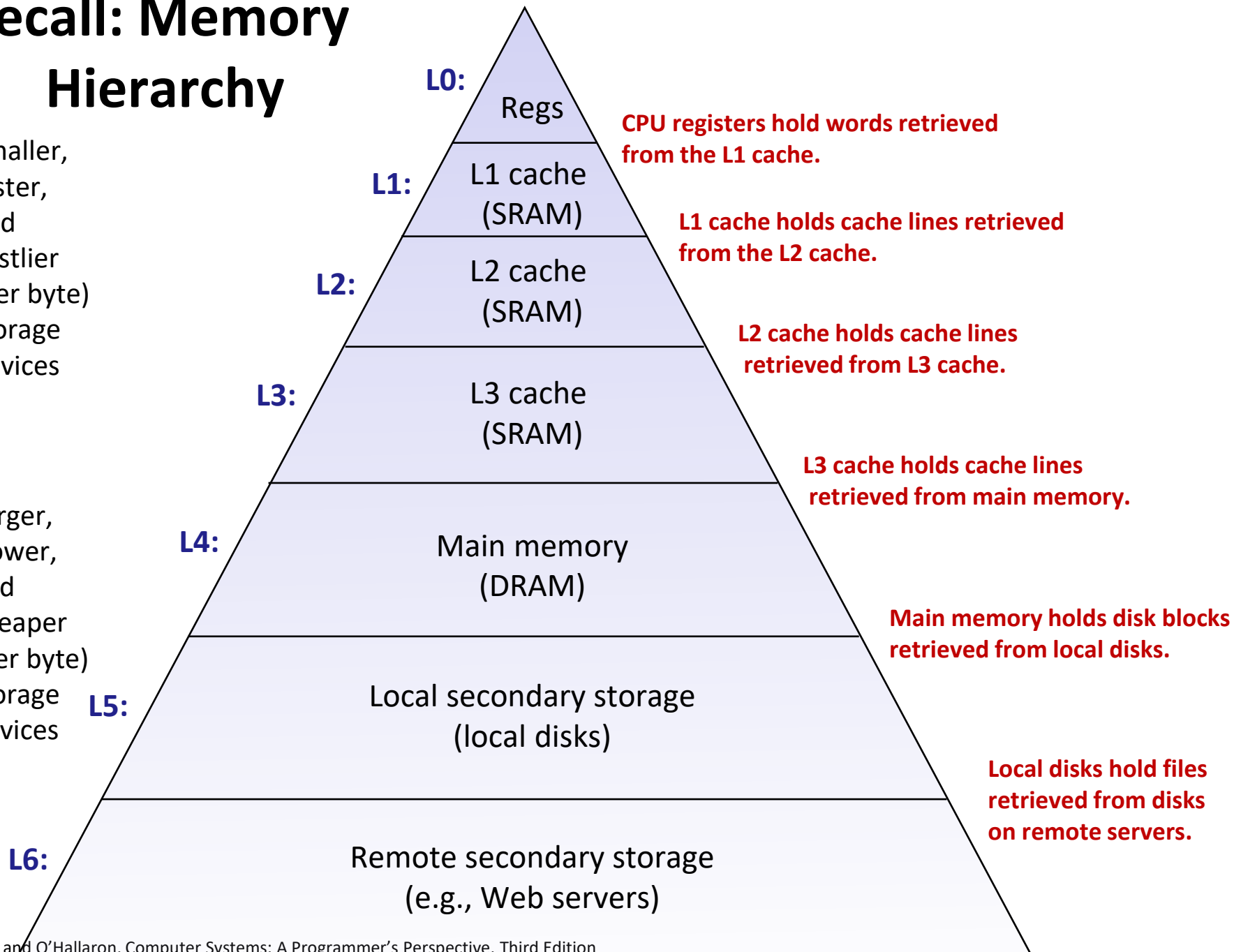
Recall: Locality

- **Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently**

- **Temporal locality:**
    - Recently referenced items are likely to be referenced again in the near future

- **Spatial locality:**
    - Items with nearby addresses tend to be referenced close together in time
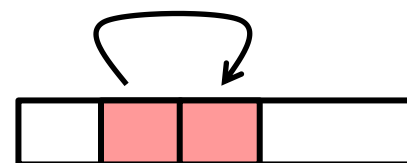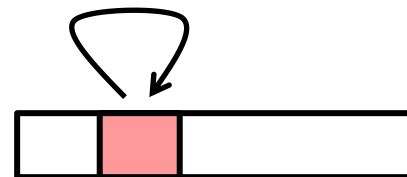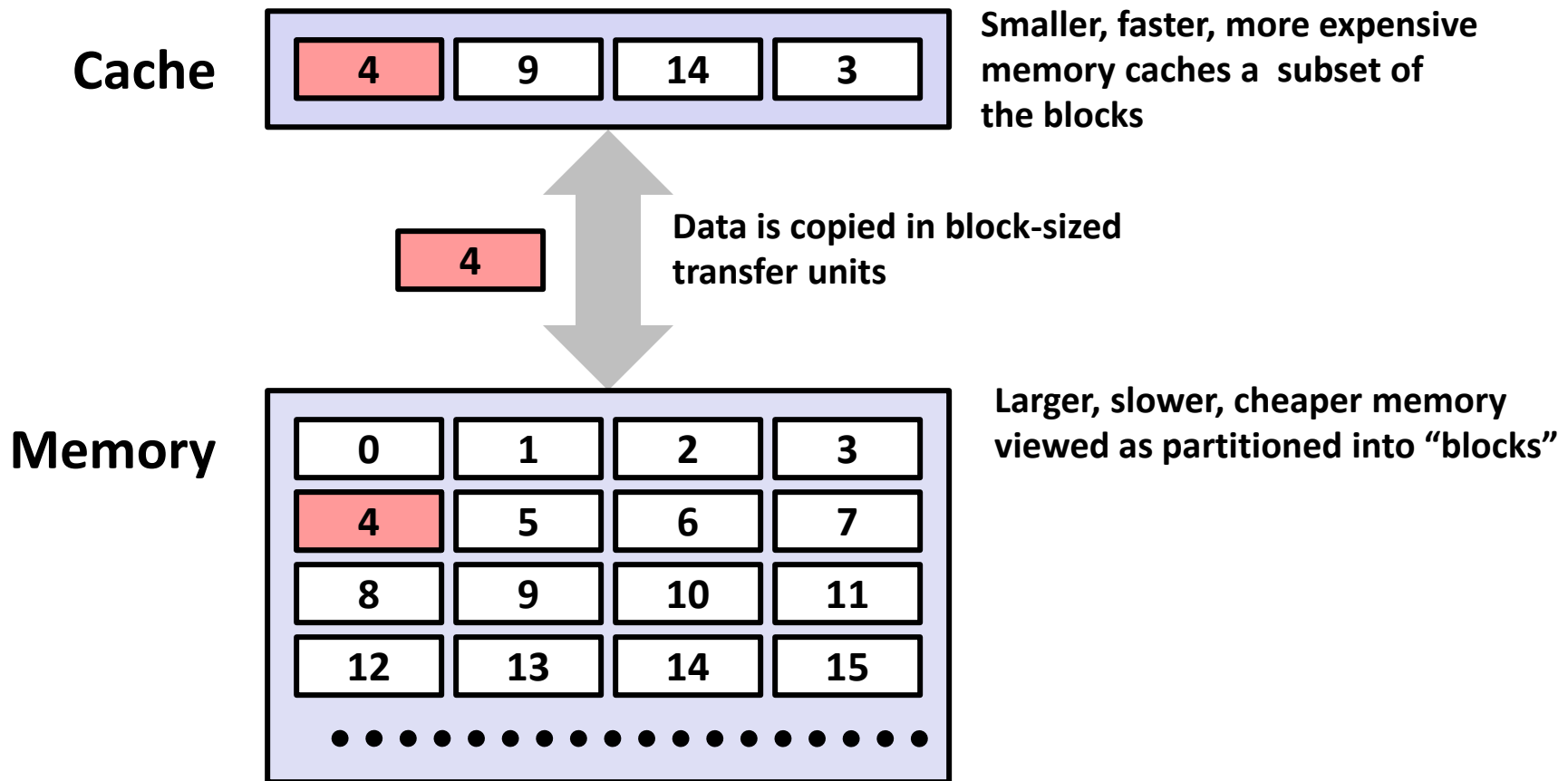
Recall: Memory Hierarchy

Smaller,
faster,
and
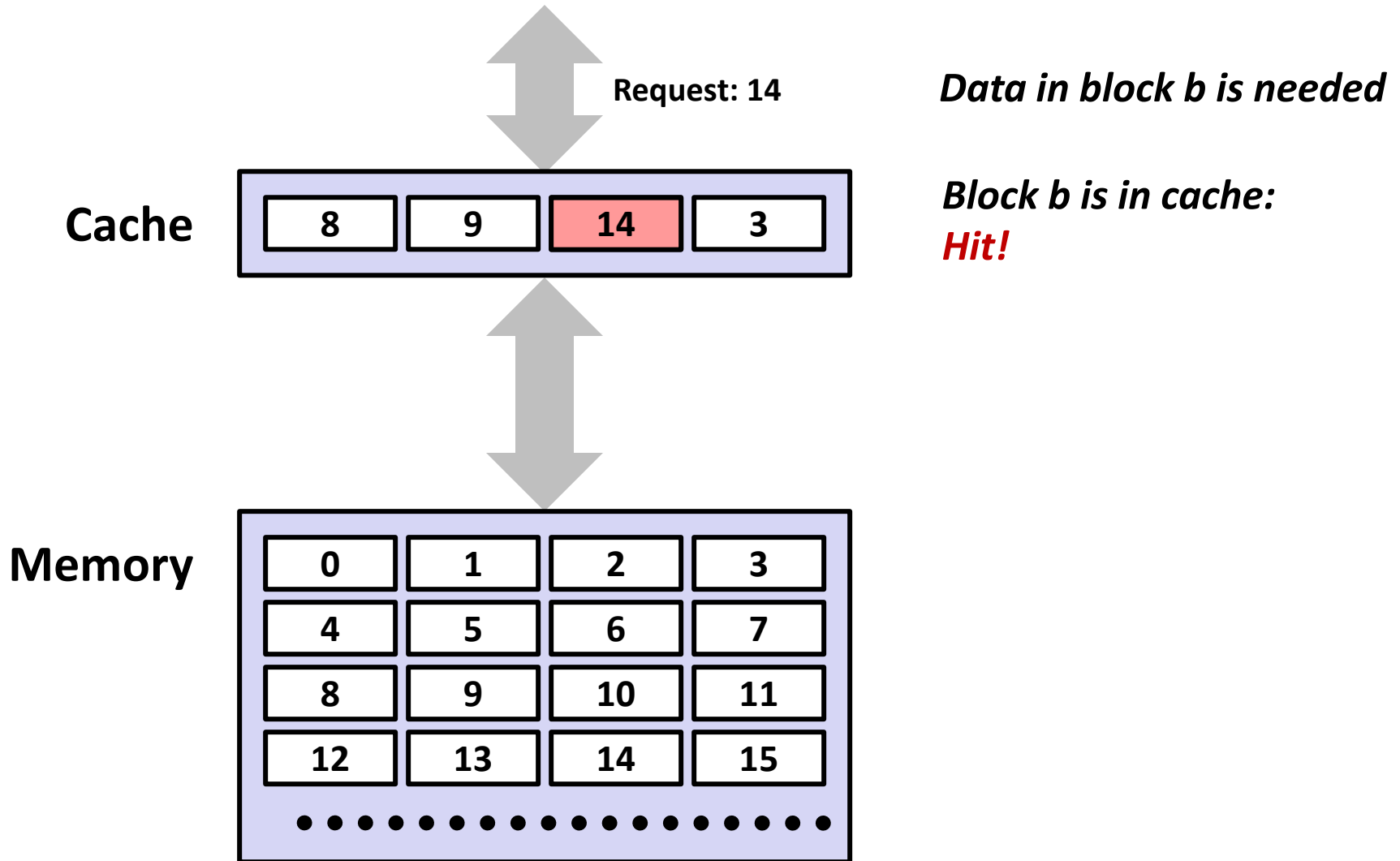costlier
(per byte)
storage
devices

Larger,
slower,
and
cheaper
(per byte)
storage
devices

L0: Regs

L1: L1 cache (SRAM)

L2: L2 cache (SRAM)

L3: L3 cache (SRAM)

L4: Main memory (DRAM)

L5: Local secondary storage (local disks)

L6: Remote secondary storage (e.g., Web servers)

CPU registers hold words retrieved from the L1 cache.

L1 cache holds cache lines retrieved from the L2 cache.

L2 cache holds cache lines retrieved from L3 cache.

L3 cache holds cache lines retrieved from main memory.

Main memory holds disk blocks retrieved from local disks.

Local disks hold files retrieved from disks on remote servers.

Recall: General Cache Concepts

Cache

4	9	14	3

Smaller, faster, more expensive memory caches a subset of the blocks

4

Data is copied in block-sized transfer units

Memory

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

Larger, slower, cheaper memory viewed as partitioned into "blocks"

General Cache Concepts: Hit

Request: 14

Data in block b is needed

Cache

| 8 | 9 | 14 | 3 |

Block b is in cache:
Hit!

Memory

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

General Cache Concepts: Miss

Cache

| 8 | 12 | 14 | 3 |

Request: 12

12   **Request: 12**

Memory

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

Data in block b is needed

Block b is not in cache:
Miss!

Block b is fetched from memory

Block b is stored in cache
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

Recall: General Caching Concepts: 3 Types of Cache Misses

- **Cold (compulsory) miss**
  - Cold misses occur because the cache starts empty and this is the first reference to the block.
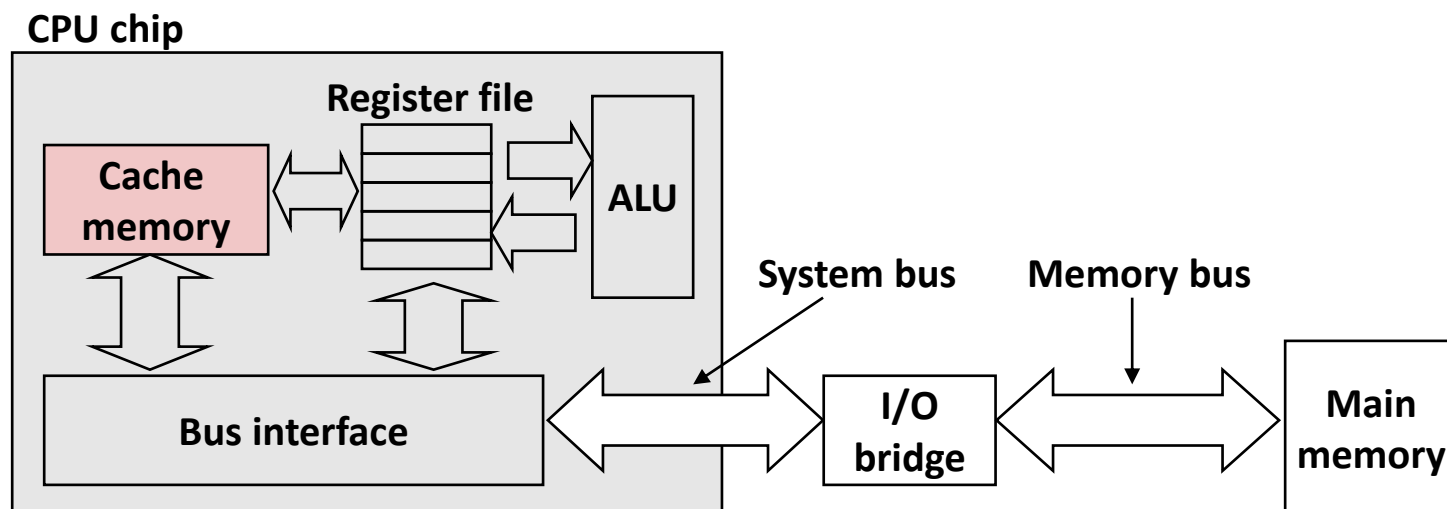
- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache.
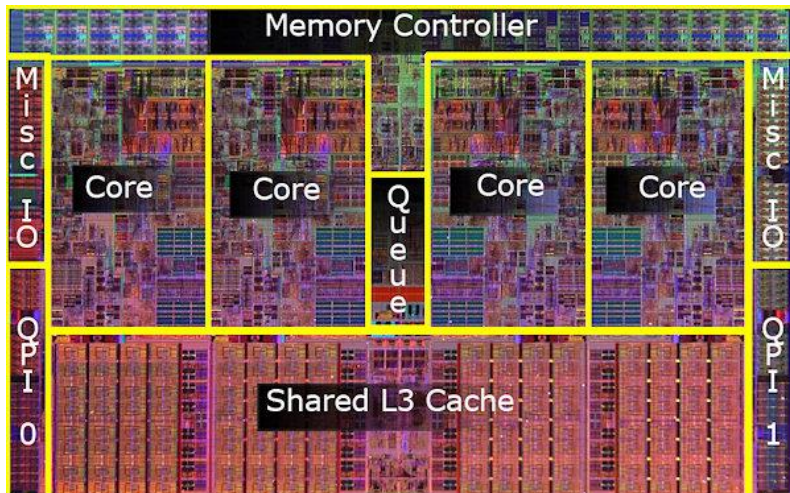
- **Conflict miss**
  - Occurs when the cache is large enough, but too many data objects all map (by the placement policy) to the same limited set of blocks
    - E.g., if the placement policy maps both 0 and 8 to the same block, then referencing 0, 8, 0, 8, 0, 8, ... would miss every time.

Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware**
  - Hold frequently accessed blocks of main memory
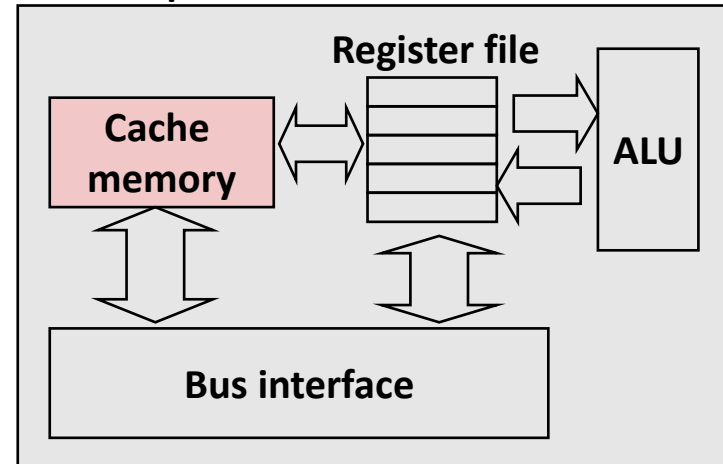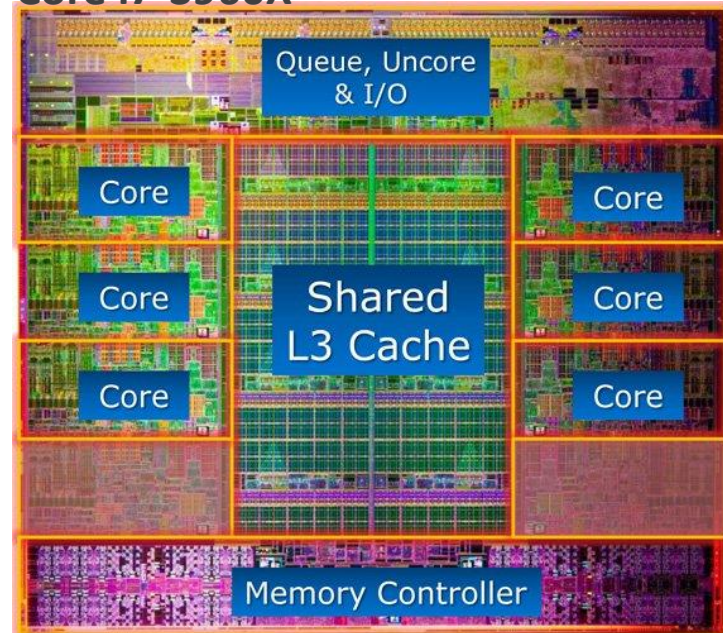- **CPU looks first for data in cache**
- **Typical system structure:**

CPU chip

Cache memory

Register file

ALU

Bus interface

System bus

Memory bus

I/O bridge

Main memory

What it Really Looks Like

Memory Controller

Misc IO | Core | Core | Queue | Core | Core | Misc IO

QPI 0 | Shared L3 Cache | QPI 1

Nehalem

AMD FX 8150

HyperTransport™ Phy | MiscIO

2MB L2 Cache | 2MB L2 Cache

2MB L3 Cache | 2MB L3 Cache

Northbridge

2MB L3 Cache | 2MB L3 Cache

2MB L2 Cache | 2MB L2 Cache

HyperTransport™ Phy | MiscIO

DDR3 Phy

CPU chip

Register file

Cache memory ⟷ | **ALU**

Bus interface

Core i7-3960X

Queue, Uncore & I/O

Core | Shared L3 Cache | Core
Core | | Core
Core | | Core

Memory Controller

What it Really Looks Like (Cont.)

**Intel Tiger Lake
Processor Die**
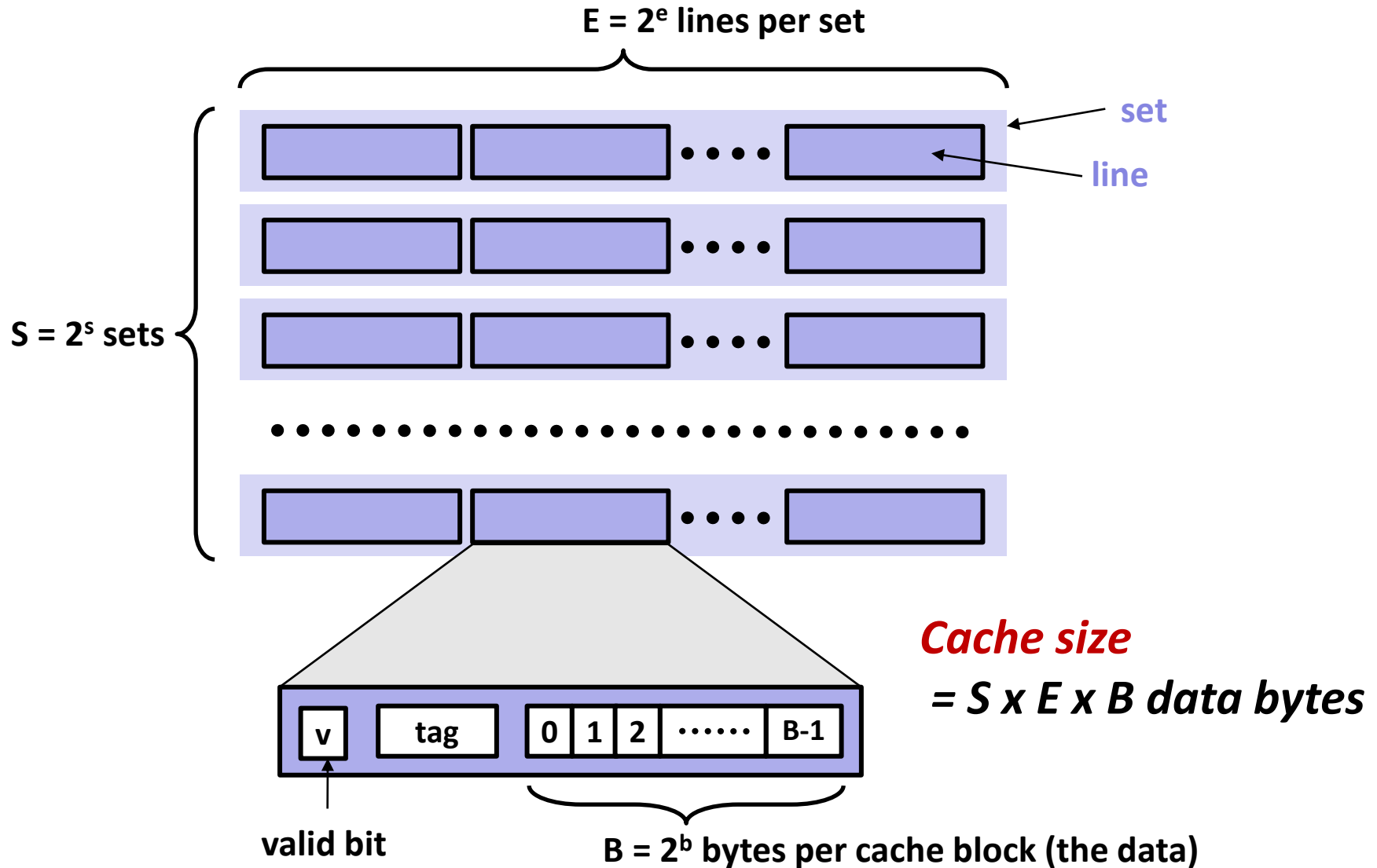**(4 core version)**

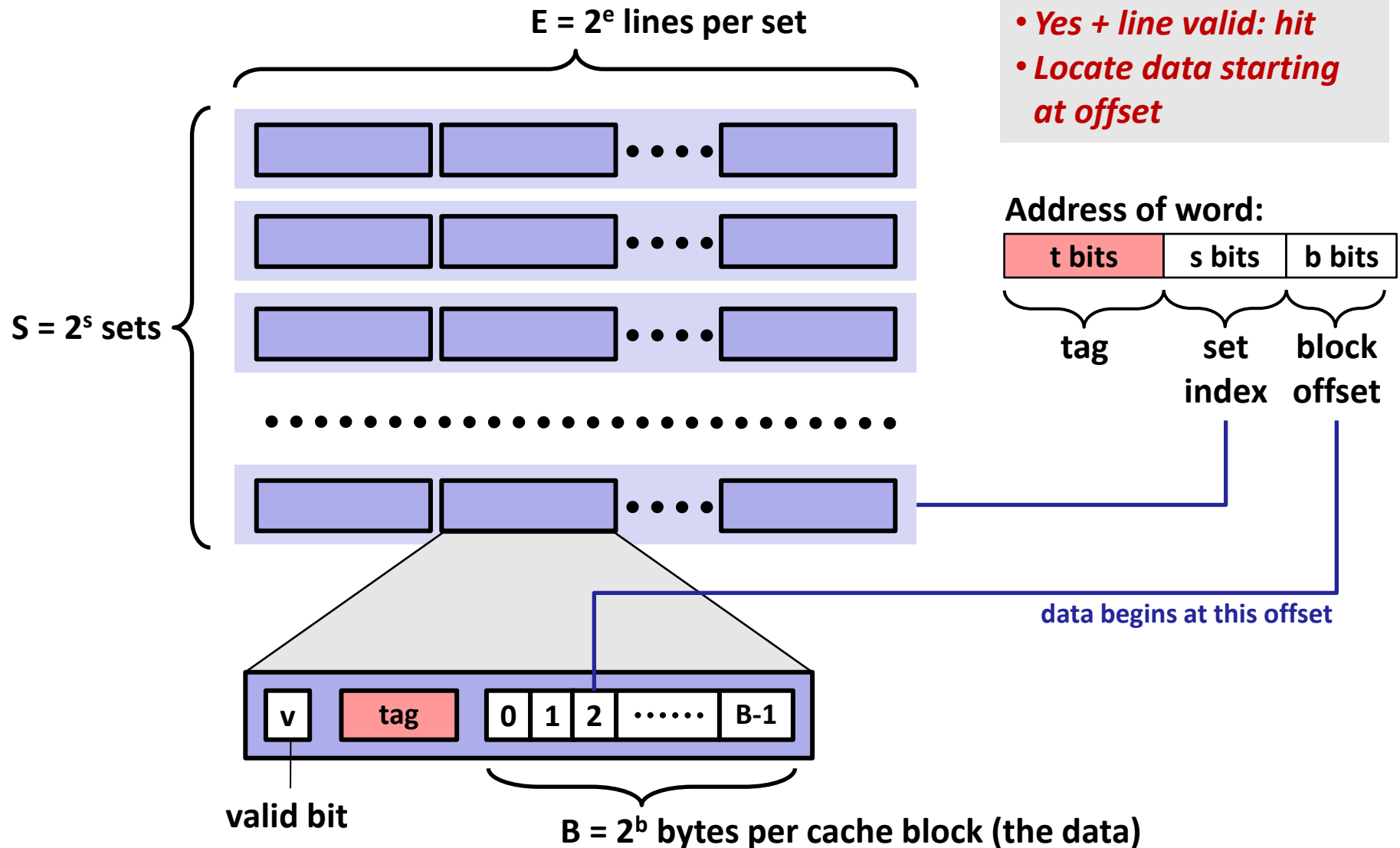**CPU caches**

**L1 per core: 32KB Instruction
+ 48KB Data**

L2 per core: 1.25MB

**Shared L3: 12MB (8 core
version has 24MB)**

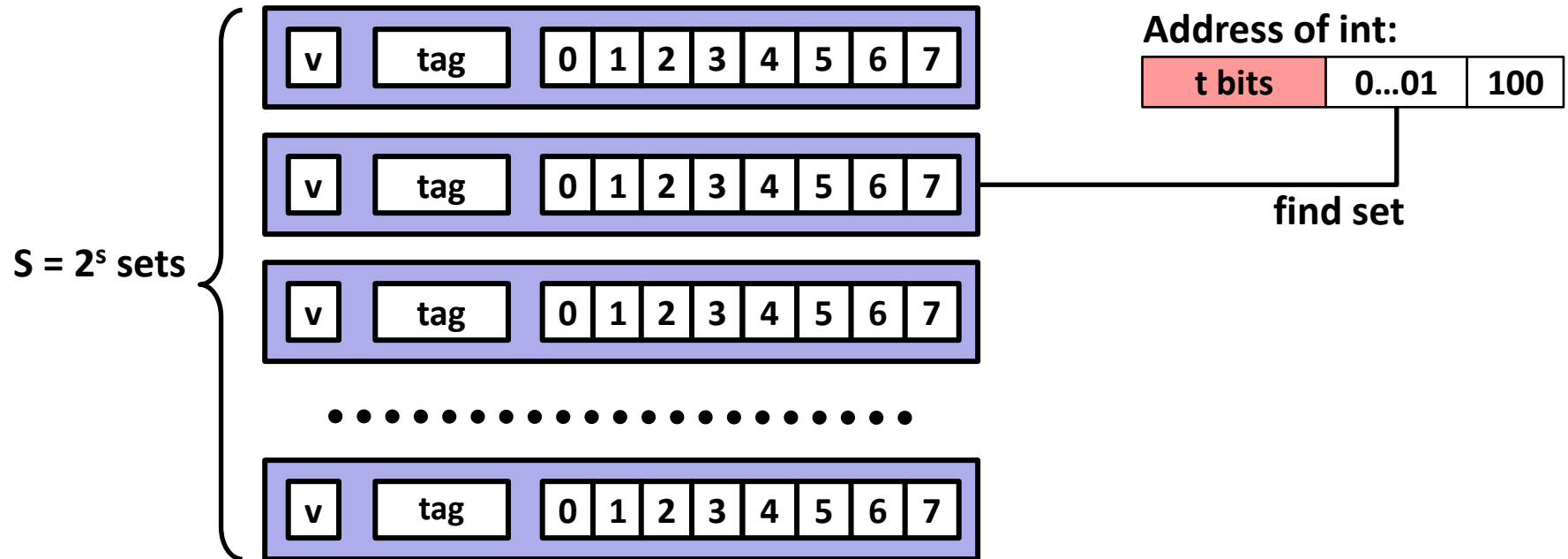# General Cache Organization (S, E, B)

$E = 2^e$ lines per set

set

line

$S = 2^s$ sets

Cache size

= S x E x B data bytes

| v | tag | 0 | 1 | 2 | ...... | B-1 |

valid bit

$B = 2^b$ bytes per cache block (the data)

Cache Read

*   *Locate set*
*   *Check if any line in set has matching tag*
*   *Yes + line valid: hit*
*   *Locate data starting at offset*

$E = 2^e$ lines per set

$S = 2^s$ sets

Address of word:

t bits	s bits	b bits

tag — set index — block offset

data begins at this offset

v	tag	0	1	2	......	B-1

valid bit

$B = 2^b$ bytes per cache block (the data)

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size B=8 bytes

S = 2^s sets

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Address of int:

| t bits | 0…01 | 100 |

find set

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size B=8 bytes

Address of int:

valid?   +   match: assume yes (= hit)

| t bits | 0...01 | 100 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

Example: Direct Mapped Cache (E = 1)

Direct mapped: One line per set
Assume: cache block size B=8 bytes

valid? + **match: assume yes (= hit)**

Address of int:

| t bits | 0…01 | 100 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

int (4 Bytes) is here
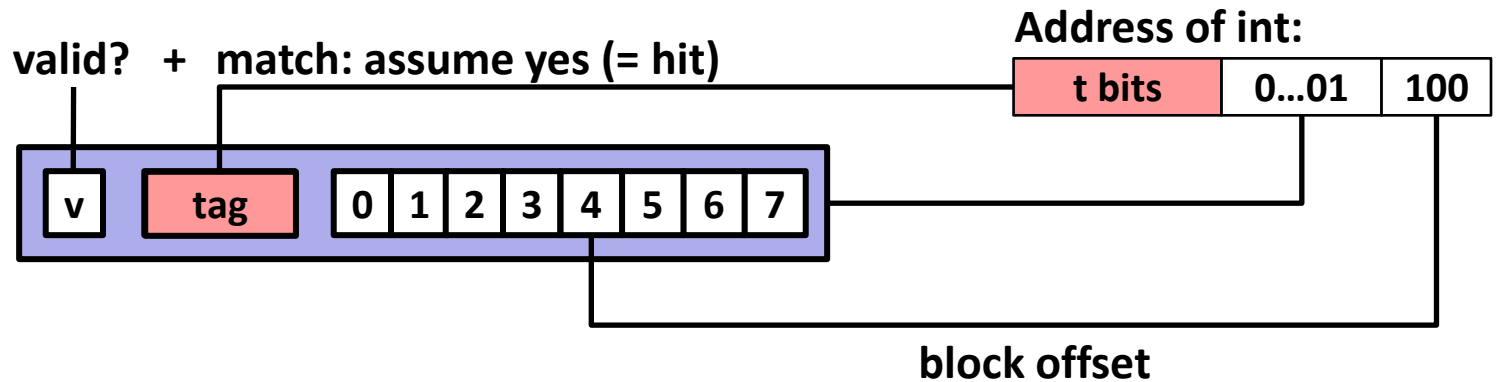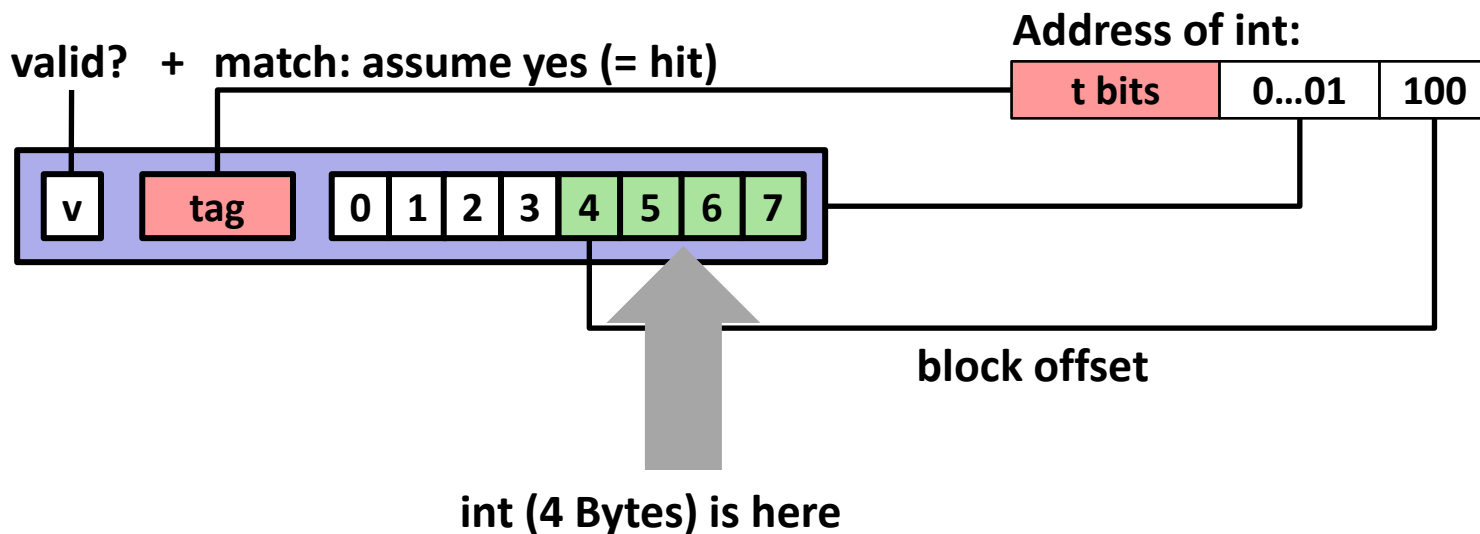
If tag doesn't match (= miss): old line is evicted and replaced

Direct-Mapped Cache Simulation

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

4-bit addresses (address space size M=16 bytes)
S=4 sets, E=1 Blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[0000$_2$],	miss	(cold)
1	[0001$_2$],	hit	
7	[0111$_2$],	miss	(cold)
8	[1000$_2$],	miss	(cold)
0	[0000$_2$]	miss	(conflict)

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1	0		
Set 2	0		
Set 3	1	0	M[6-7]

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set
Assume: cache block size B=8 bytes

Address of short int:

| t bits | 0...01 | 100 |

2 lines per set

find set

S sets

E-way Set Associative Cache (Here: E = 2)
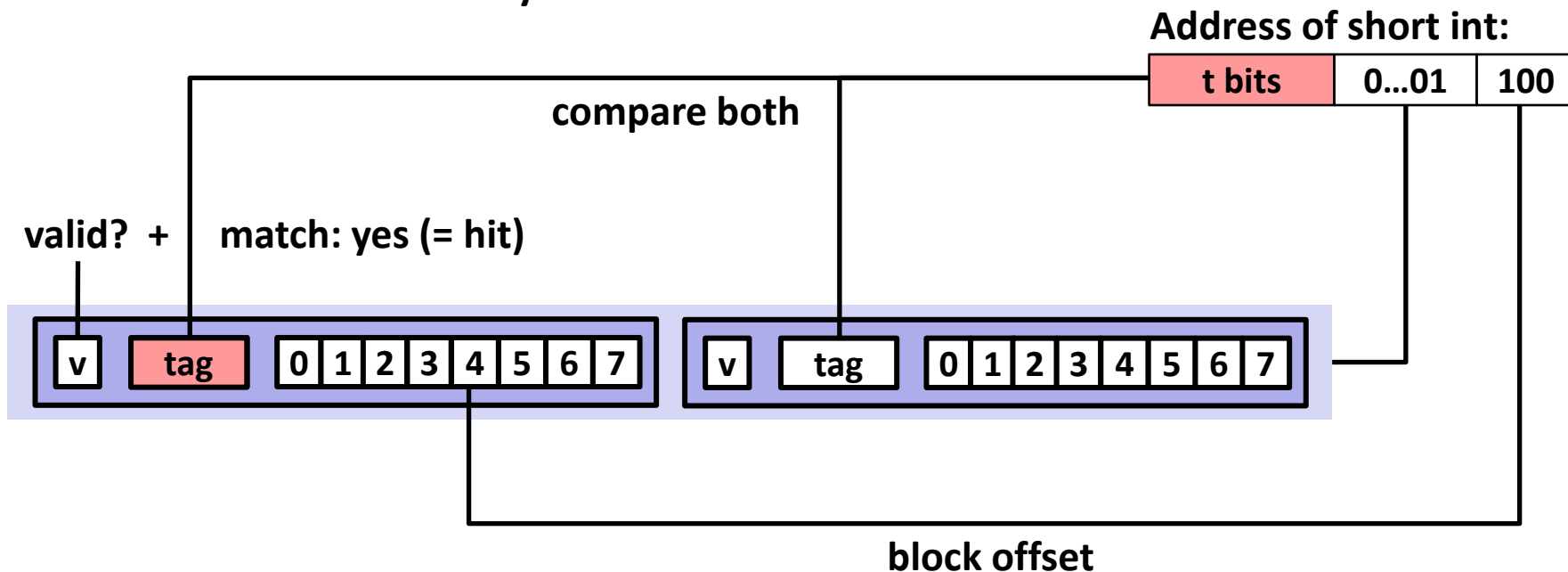
E = 2: Two lines per set

Assume: cache block size B=8 bytes

Address of short int:

t bits	0...01	100

compare both

valid? +   match: yes (= hit)

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size B=8 bytes

Address of short int:

| t bits | 0...01 | 100 |

compare both

valid? +   match: yes (= hit)

| v | **tag** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

block offset

short int (2 Bytes) is here
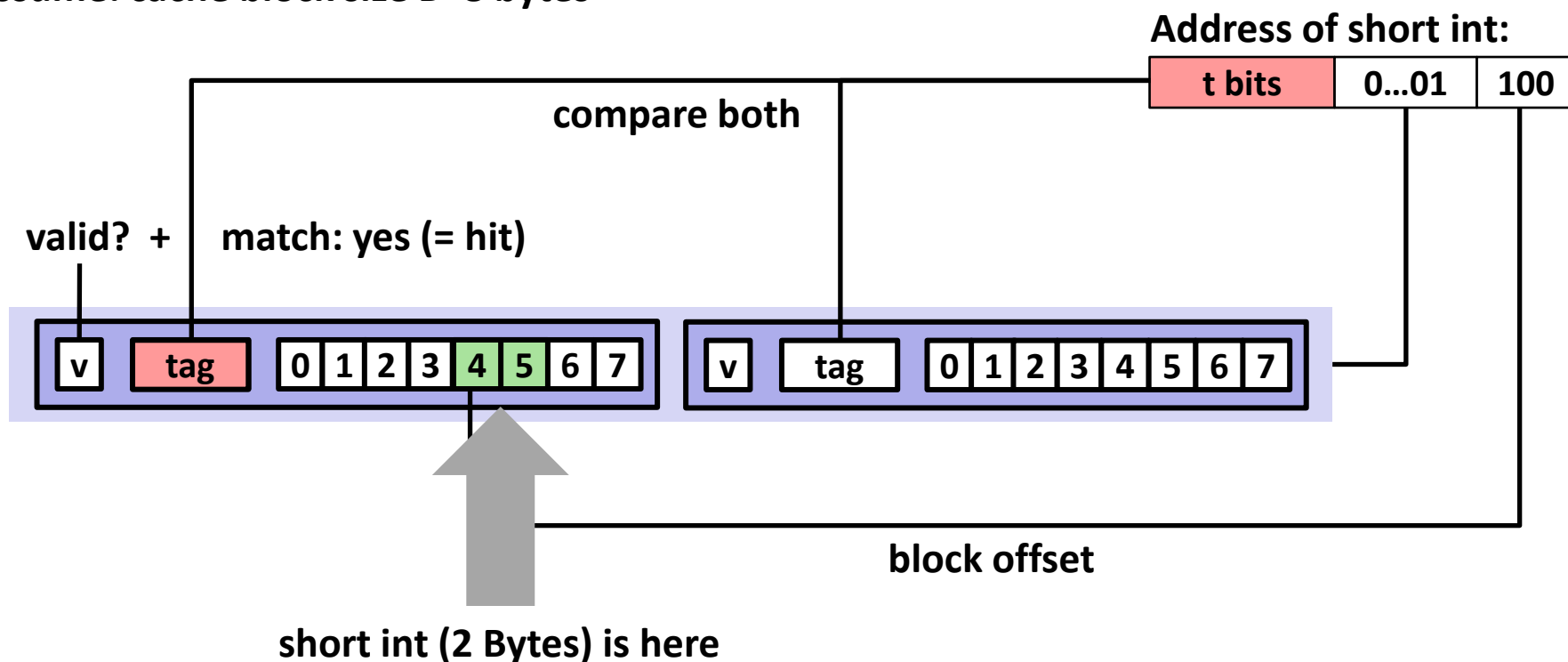
No match or not valid (= miss):

- **One line in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU), ...**

2-Way Set Associative Cache Simulation

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

4-bit addresses (M=16 bytes)
S=2 sets, E=2 blocks/set, B=2 bytes/block

Address trace (reads, one byte per read):

0	[**000**$\underline{0}$$_2$],	miss
1	[**00**$\underline{0}$**1**$_2$],	hit
7	[**01**$\underline{1}$**1**$_2$],	miss
8	[**10**$\underline{0}$**0**$_2$],	miss
0	[**000**$\underline{0}$$_2$]	hit

	v	**Tag**	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0		

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

What about writes?

- **Multiple copies of data exist:**
  - L1, L2, L3, Main Memory, Disk

valid bit   dirty bit $B = 2^b$ bytes

- **What to do on a write-hit?**
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Each cache line needs a dirty bit (set if data differs from memory)

- **What to do on a write-miss?**
  - Write-allocate (load into cache, update line in cache)
    - Good if more writes to the location will follow
  - No-write-allocate (writes straight to memory, does not load into cache)

- **Typical**
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate**
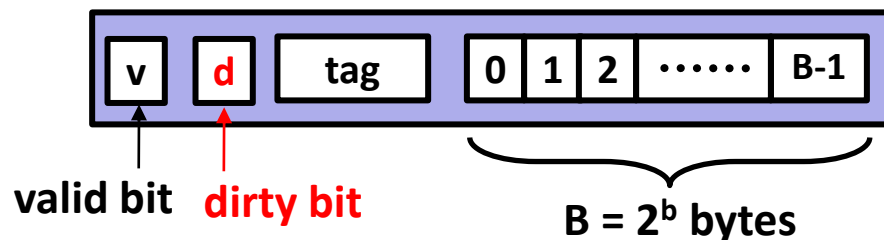
Why Index Using Middle Bits?

Direct mapped: One line per set
Assume: cache block size 8 bytes

$S = 2^s$ **sets**

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

• •

| v | tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Standard Method:
Middle bit indexing

Address of int:

| t bits | 0…01 | 100 |

find set

Alternative Method:
High bit indexing

Address of int:

| 1…11 | t bits | 100 |

find set
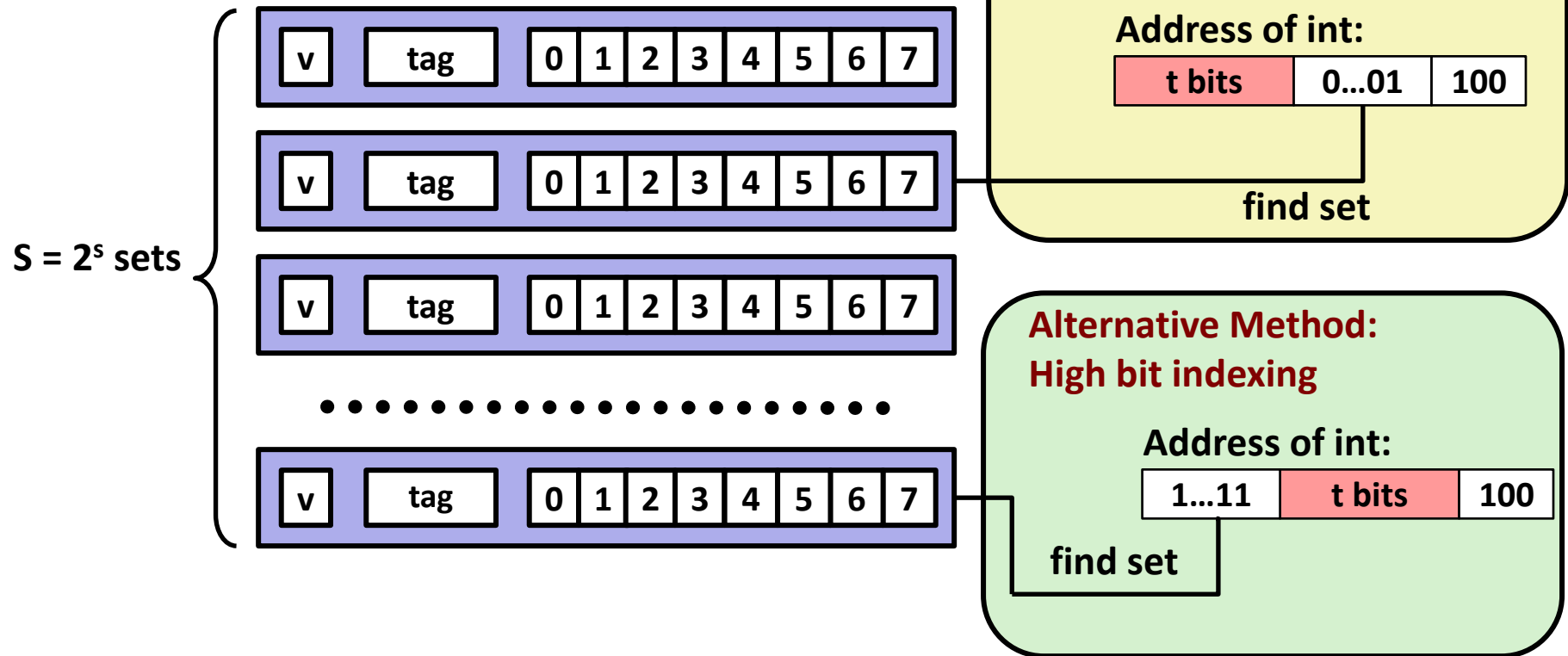
Illustration of Indexing Approaches

- **64-byte memory**
  - 6-bit addresses
- **16 byte, direct-mapped cache**
- **Block size = 4. (Thus, 4 sets; why?)**
- **2 bits tag, 2 bits index, 2 bits offset**
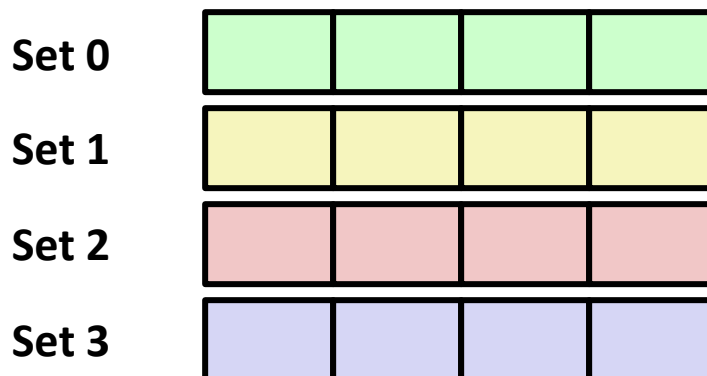
Set 0

Set 1

Set 2

Set 3

0000**xx**
0001**xx**
0010**xx**
0011**xx**
0100**xx**
0101**xx**
0110**xx**
0111**xx**
1000**xx**
1001**xx**
1010**xx**
1011**xx**
1100**xx**
1101**xx**
1110**xx**
1111**xx**

Middle Bit Indexing

- **Addresses of form TTSSBB**

  - **TT**   Tag bits
  - **SS**   Set index bits
  - **BB**   Offset bits

- **Makes good use of spatial locality**

Set 0

Set 1

Set 2

Set 3

0000xx

0001xx

0010xx

0011xx

0100xx

0101xx

0110xx

0111xx

1000xx

1001xx

1010xx

1011xx

1100xx

1101xx

1110xx

1111xx
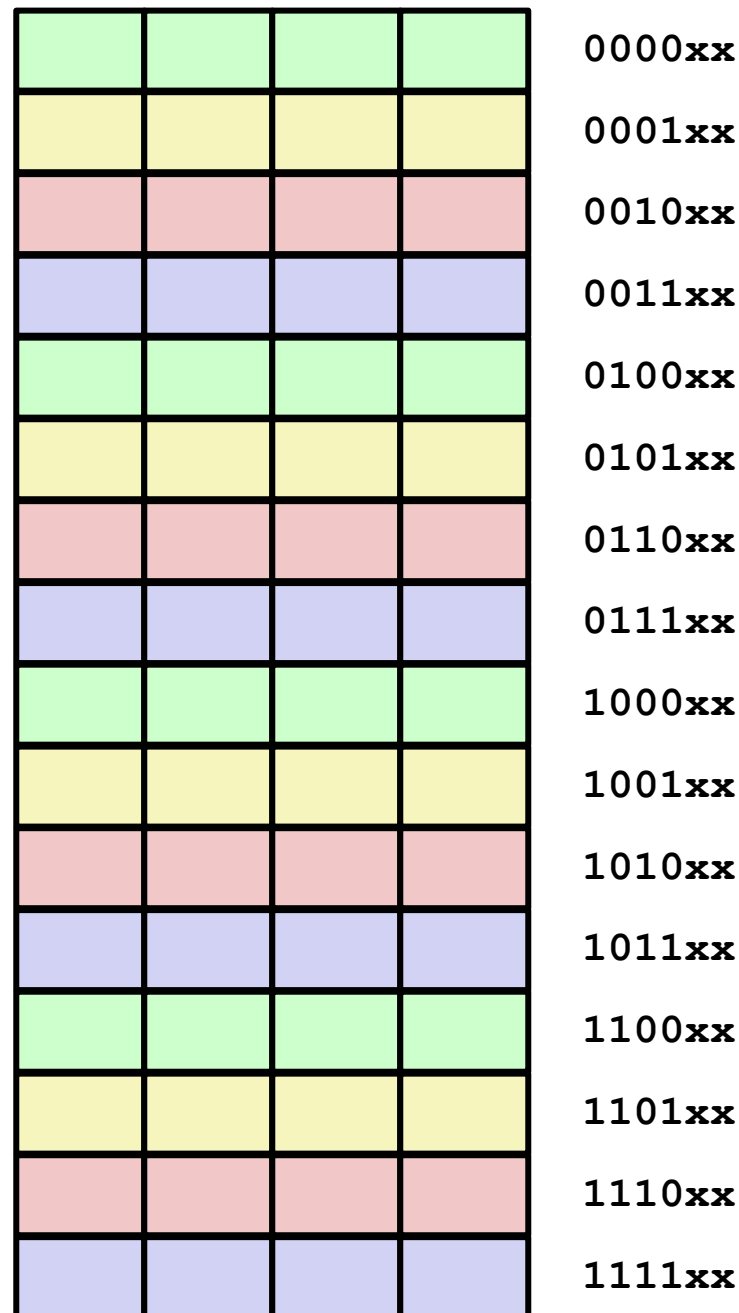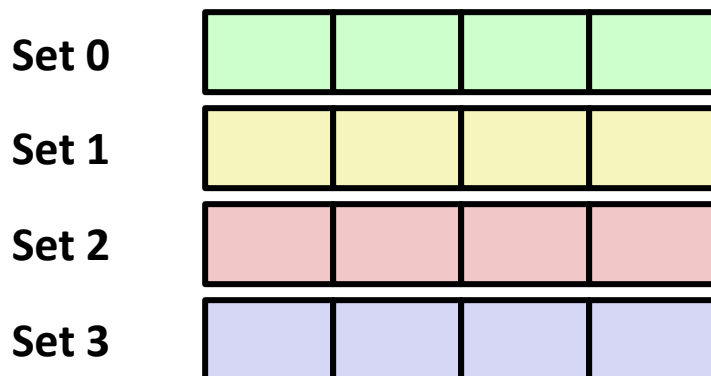
High Bit Indexing

- **Addresses of form SSTTBB**
    - **SS**   Set index bits
    - **TT**   Tag bits
    - **BB**   Offset bits

- **Program with high spatial locality would generate lots of conflicts**

Set 0				
Set 1				
Set 2				
Set 3				

				0000**xx**
				0001**xx**
				0010**xx**
				0011**xx**
				0100**xx**
				0101**xx**
				0110**xx**
				0111**xx**
				1000**xx**
				1001**xx**
				1010**xx**
				1011**xx**
				1100**xx**
				1101**xx**
				1110**xx**
				1111**xx**
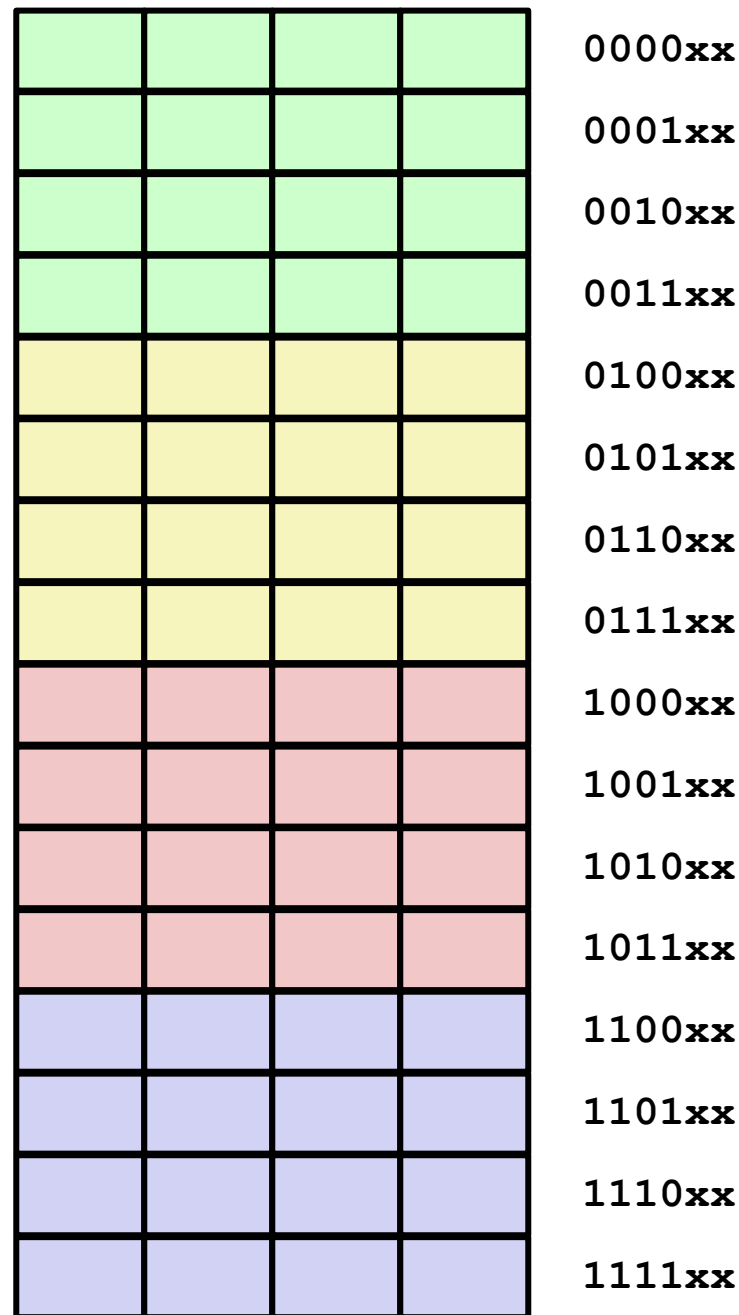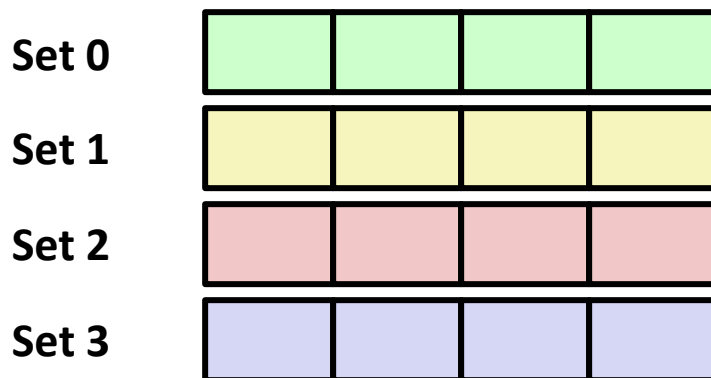
Intel Core i7 Cache Hierarchy

Processor package
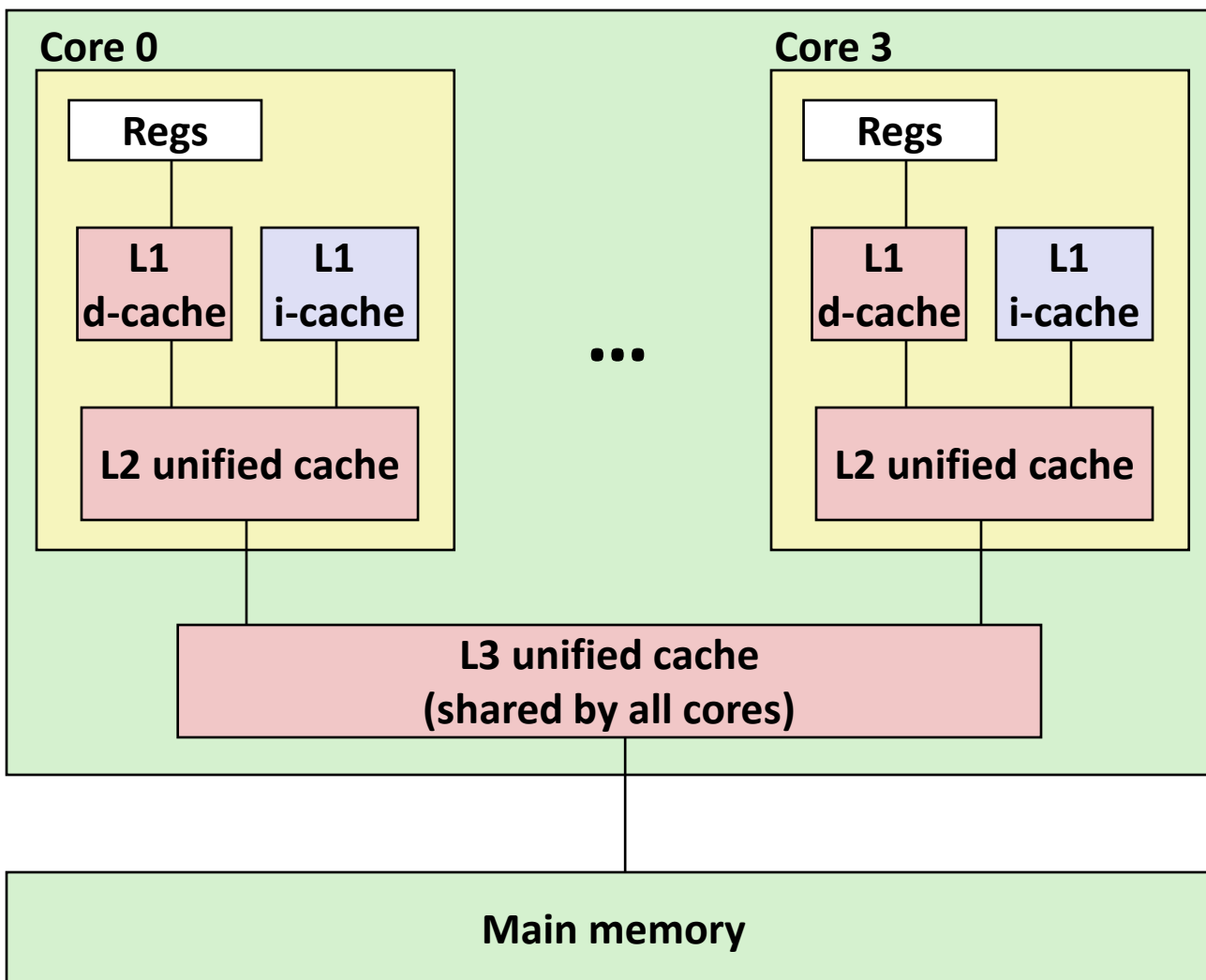
L1 i-cache and d-cache:
   32 KB, 8-way,
   Access: 4 cycles

L2 unified cache:
   256 KB, 8-way,
   Access: 10 cycles

L3 unified cache:
   8 MB, 16-way,
   Access: 40-75 cycles
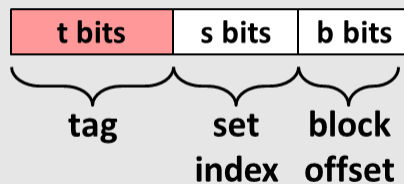
Block size: 64 bytes for all caches.

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B =
S =   , s =
E =   , e =
C =

E = 2^e lines per set

set
line

S = 2^s sets

v | tag | 0 1 2 .... B-1

valid bit

Cache size:
C = S x E x B data bytes

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:

t bits	s bits	b bits
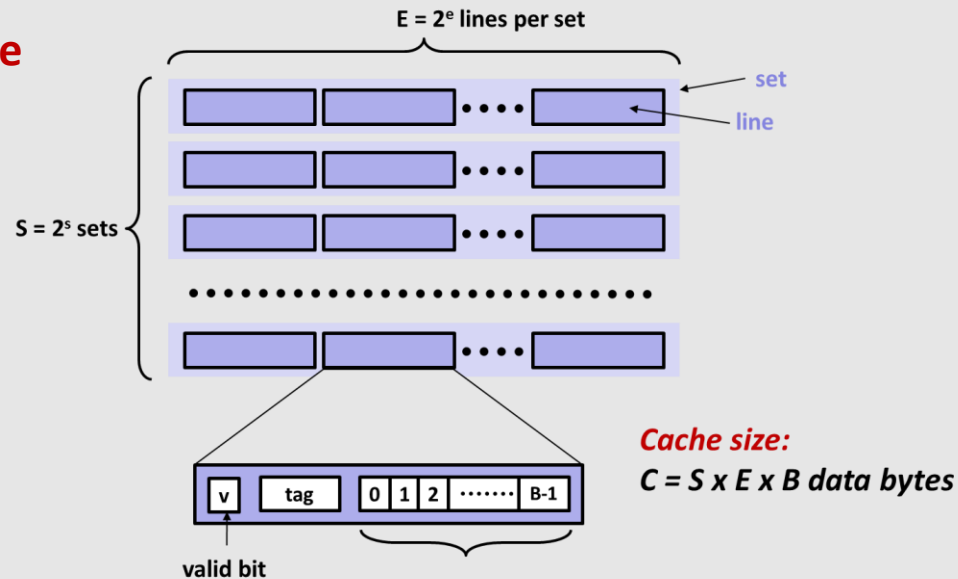
tag   set index   block offset

Block offset: . bits
Set index: . bits
Tag: . bits

Stack Address:
0x00007f7262a1e010

Block offset: **0x??**
Set index: **0x??**
Tag: **0x??**

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768

$E = 2^e$ lines per set

$S = 2^s$ sets

set
line

Cache size:
C = S x E x B data bytes

v | tag | 0 | 1 | 2 | .... | B-1

valid bit

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:

t bits	s bits	b bits

tag | set index | block offset

Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

Stack Address:
0x00007f7262a1e010

Block offset: **0x??**
Set index: **0x??**
Tag: **0x??**

Example: Core i7 L1 Data Cache

32 kB 8-way set associative
64 bytes/block
47 bit address range

B = 64
S = 64, s = 6
E = 8, e = 3
C = 64 x 64 x 8 = 32,768

E = 2^e lines per set

S = 2^s sets

set
line

v | tag | 0 | 1 | 2 | .... | B-1

valid bit

Cache size:
C = S x E x B data bytes

Hex	Decimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Address of word:

t bits	s bits	b bits

tag — set index — block offset

Block offset: 6 bits
Set index: 6 bits
Tag: 35 bits

Stack Address:
0x00007f7262a1e010

0000 0001 0000

Block offset: 0x10
Set index: 0x0
Tag: 0x7f7262a1e

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory accesses not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (as %):
    - 3-10% for L1
    - can be quite small (e.g., < 1%) for L2, depending on size, etc.

- **Hit Time**
  - Time to deliver a cached block to the processor
    - includes time to determine whether line is in cache
  - Typical numbers:
    - 4 clock cycle for L1
    - 10 clock cycles for L2

- **Miss Penalty**
  - Additional time required because of a miss
    - typically 50-200 cycles for main memory (Trend: increasing!)

How Bad Can a Few Cache Misses Be?

- **Huge difference between a hit and a miss**
  - Could be 100x, if just L1 and main memory

- **Would you believe 99% hits is twice as good as 97%?**
  - Consider this simplified example:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

  - Average access time:
    97% hits: 1 cycle + 0.03 x 100 cycles = **4 cycles**
    99% hits: 1 cycle + 0.01 x 100 cycles = **2 cycles**

- **This is why "miss rate" is used instead of "hit rate"**

Writing Cache Friendly Code

- **Make the common case go fast**
  - Focus on the inner loops of the core functions

- **Minimize the misses in the inner loops**
  - Repeated references to variables are good (temporal locality)
  - Stride-1 reference patterns are good (spatial locality)

Key idea: Our qualitative notion of locality is quantified through our understanding of cache memories

Quiz Time!

Check out:

Canvas > Day 10 – Cache Memories

Today

- **Cache organization and operation**

- **Performance impact of caches**

  - The memory mountain

  - Rearranging loops to improve spatial locality

  - Using blocking to improve temporal locality

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.

Memory Mountain Test Function

```c
long data[MAXELEMS];  /* Global array to traverse */

/* test - Iterate over first "elems" elements of
 *        array "data" with stride of "stride",
 *        using 4x4 loop unrolling.
 */
int test(int elems, int stride) {
    long i, sx2=stride*2, sx3=stride*3, sx4=stride*4;
    long acc0 = 0, acc1 = 0, acc2 = 0, acc3 = 0;
    long length = elems, limit = length - sx4;

    /* Combine 4 elements at a time */
    for (i = 0; i < limit; i += sx4) {
        acc0 = acc0 + data[i];
        acc1 = acc1 + data[i+stride];
        acc2 = acc2 + data[i+sx2];
        acc3 = acc3 + data[i+sx3];
    }

    /* Finish any remaining elements */
    for (; i < length; i++) {
        acc0 = acc0 + data[i];
    }
    return ((acc0 + acc1) + (acc2 + acc3));
}
```

mountain/mountain.c

Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

1. Call `test()` once to warm up the caches.

2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain

Core i7 Haswell
2.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

L1

L2

L3

Mem

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition
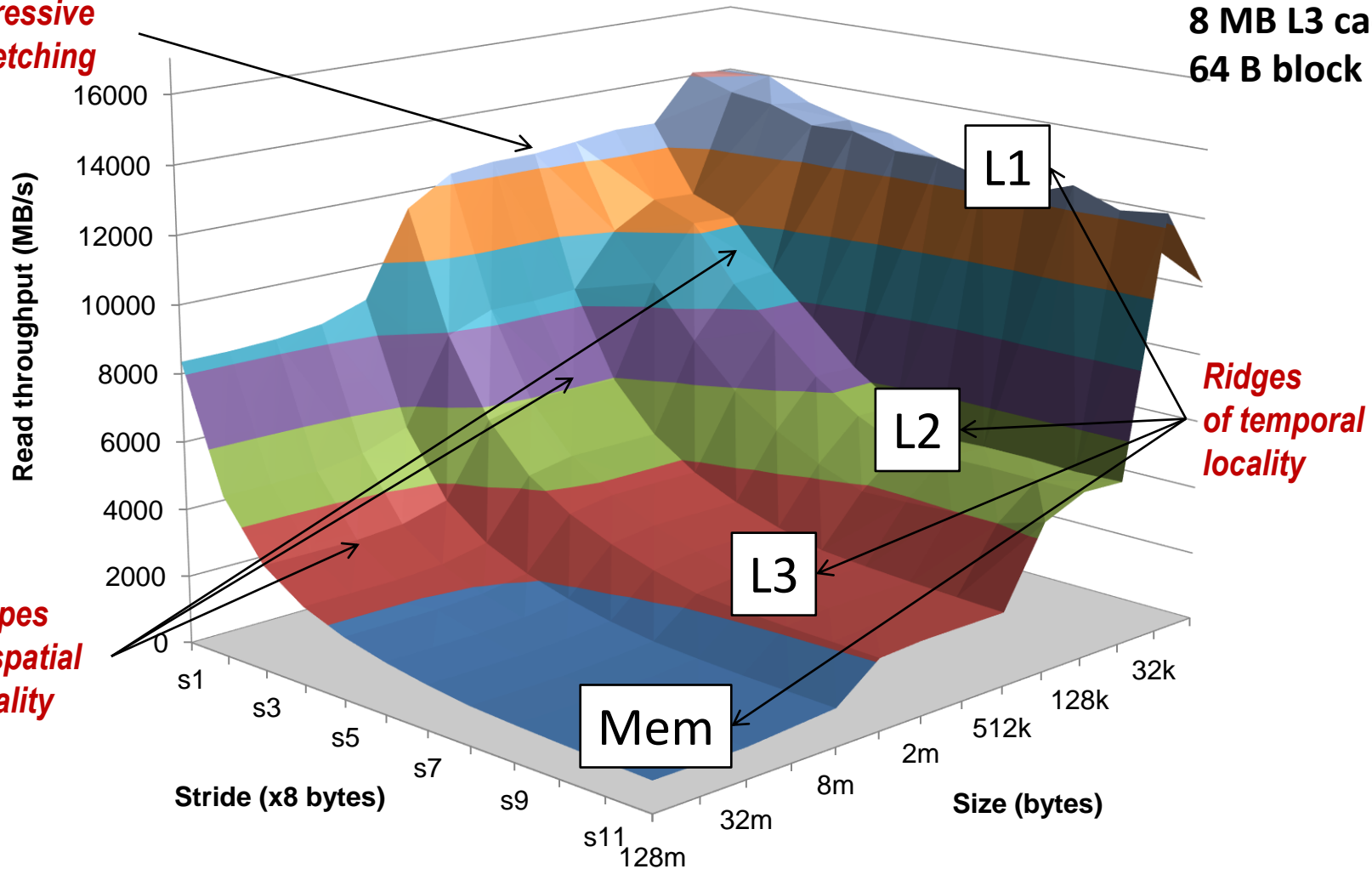
Closer Look at Stride Effects

128K

Stride   Size

Throughput for size = 128K

Miss rate = stride/8

Miss rate = 1.0

Measured

MB/sec

35000
30000
25000
20000
15000
10000
5000
0

s1   s2   s3   s4   s5   s6   s7   s8   s9   s10   s11   s12

stride
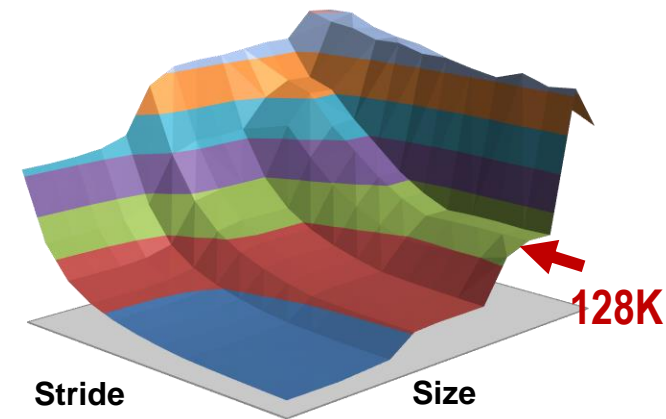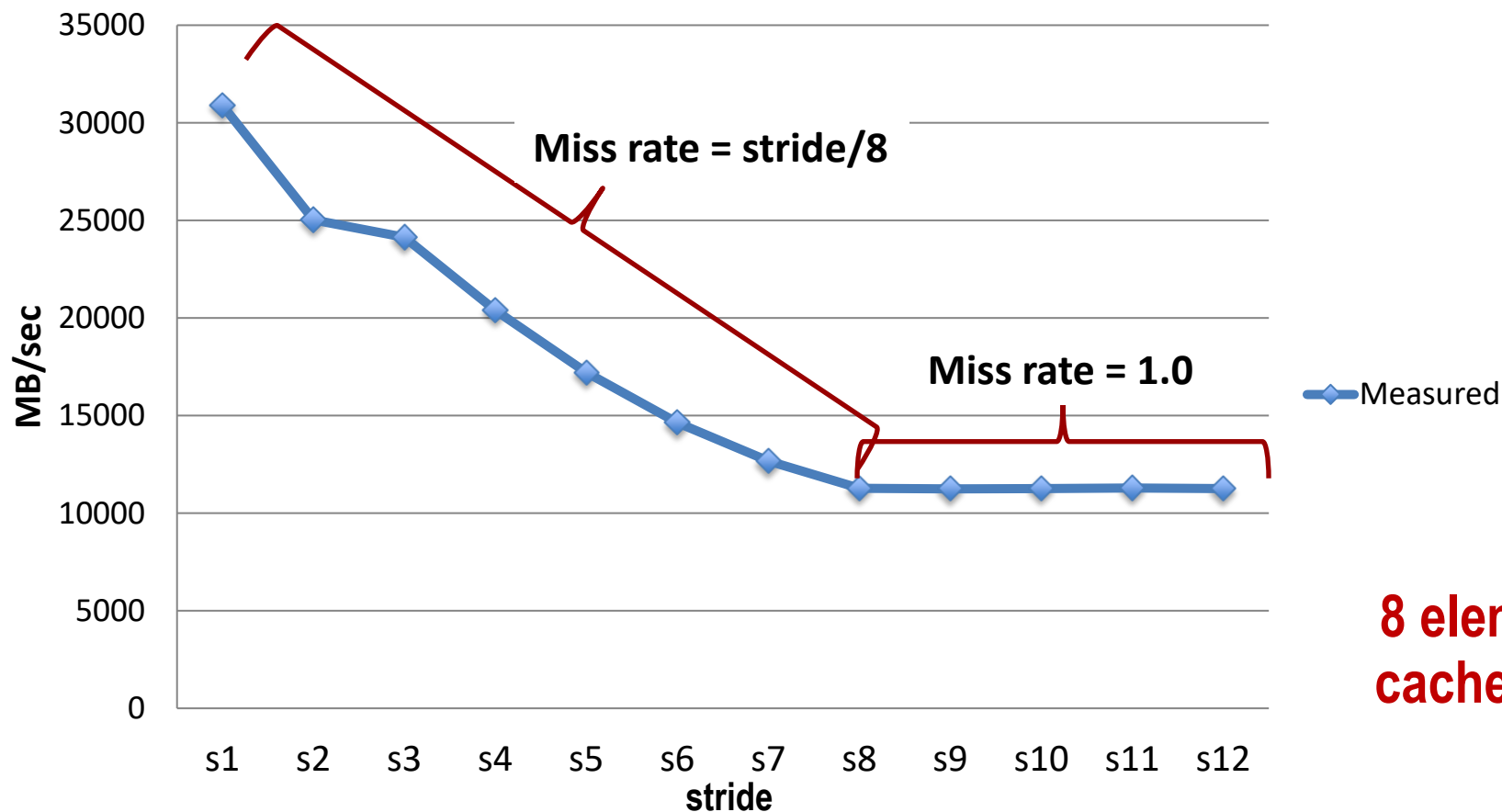
8 elems per cache block

Today

- **Cache organization and operation**

- **Performance impact of caches**
    - The memory mountain
    - Rearranging loops to improve spatial locality
    - Using blocking to improve temporal locality

Matrix Multiplication Example

- **Description:**
  - Multiply *N* x *N* matrices
  - Matrix elements are doubles (8 bytes)
  - $O(N^3)$ total operations
  - *N* reads per source element
  - *N* values summed per destination
   - but may be able to hold in register

*Variable **sum** held 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;
  }
}                          matmult/mm.c
```

Miss Rate Analysis for Matrix Multiply

- **Assume:**
  - Block size = 32B (big enough for four doubles)
  - 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

C = A X B
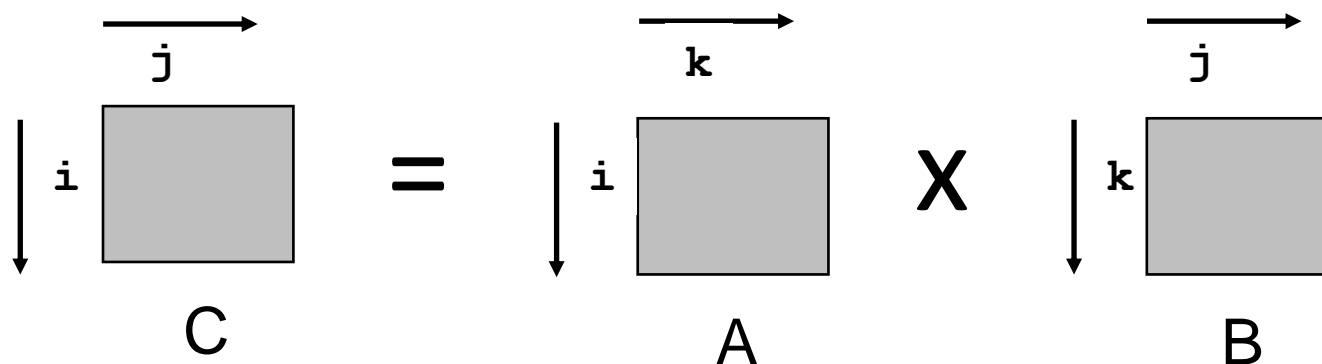
Layout of C Arrays in Memory (review)

- **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) > sizeof(a_{ij}) bytes, exploit spatial locality
    - miss rate = sizeof(a_{ij}) / B
- **Stepping through rows in one column:**
  - ```
    for (i = 0; i < n; i++)
        sum += a[i][0];
    ```
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)

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;
  }
}                        matmult/mm.c
```

Inner loop:

(*,j)

(i,*)   (i,j)

A   B   C

Row-wise   Column-   Fixed
           wise
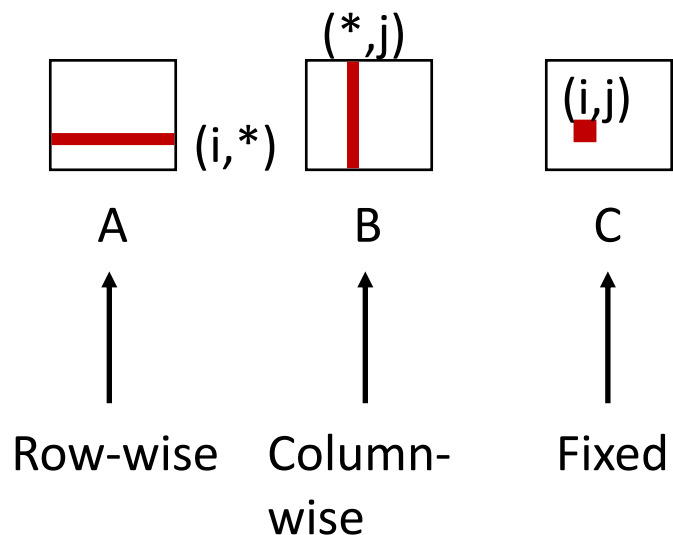
Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

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
  }
}                              matmult/mm.c
```

Inner loop:

	(*,j)	(i,j)
(i,*)		
A	B	C
Row-wise	Column-wise	Fixed
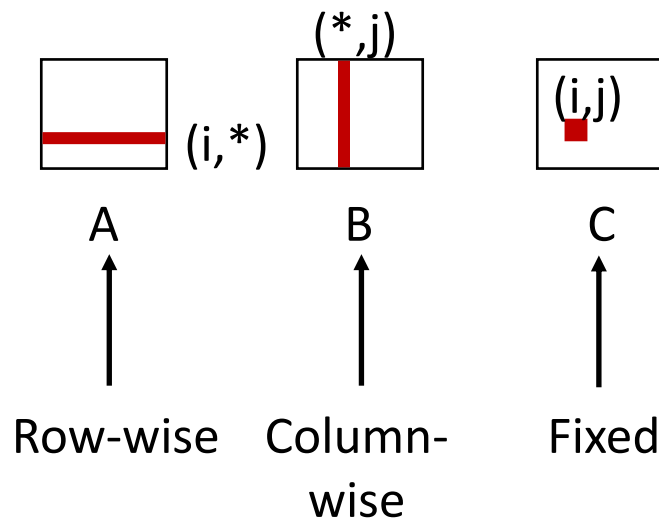
Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Same analysis as `ijk`

Block size = 32B (four doubles)

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];
  }
}                          matmult/mm.c
```

Inner loop:

(i,k)   (k,*)   (i,*)
  A   B   C
Fixed   Row-wise Row-wise
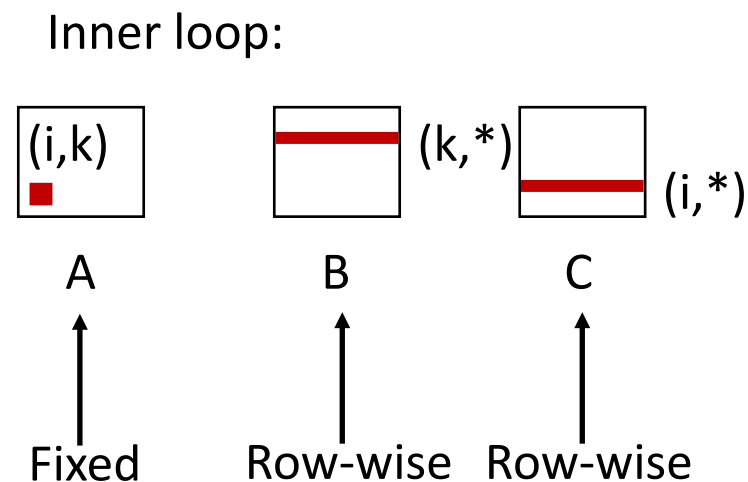
Miss rate for inner loop iterations:

A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)

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;
  }
}
                        matmult/mm.c
```

Inner loop:

(*,k)   (k,j)   (*,j)

A   B   C

Column-   Fixed   Column-
wise   wise
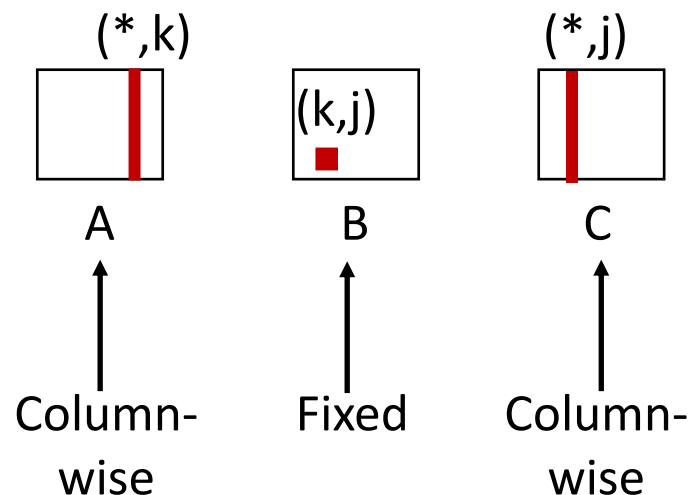
Miss rate for inner loop iterations:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)

Summary of Matrix Multiplication

```
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;
  }
}
```

ijk (& jik):
- 2 loads, 0 stores
- avg misses/iter = **1.25**

```
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];
 }
}
```

kij (& ikj):
- 2 loads, 1 store
- avg misses/iter = **0.5**

```
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;
 }
}
```

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

Core i7 Matrix Multiply Performance

Cycles per inner loop iteration
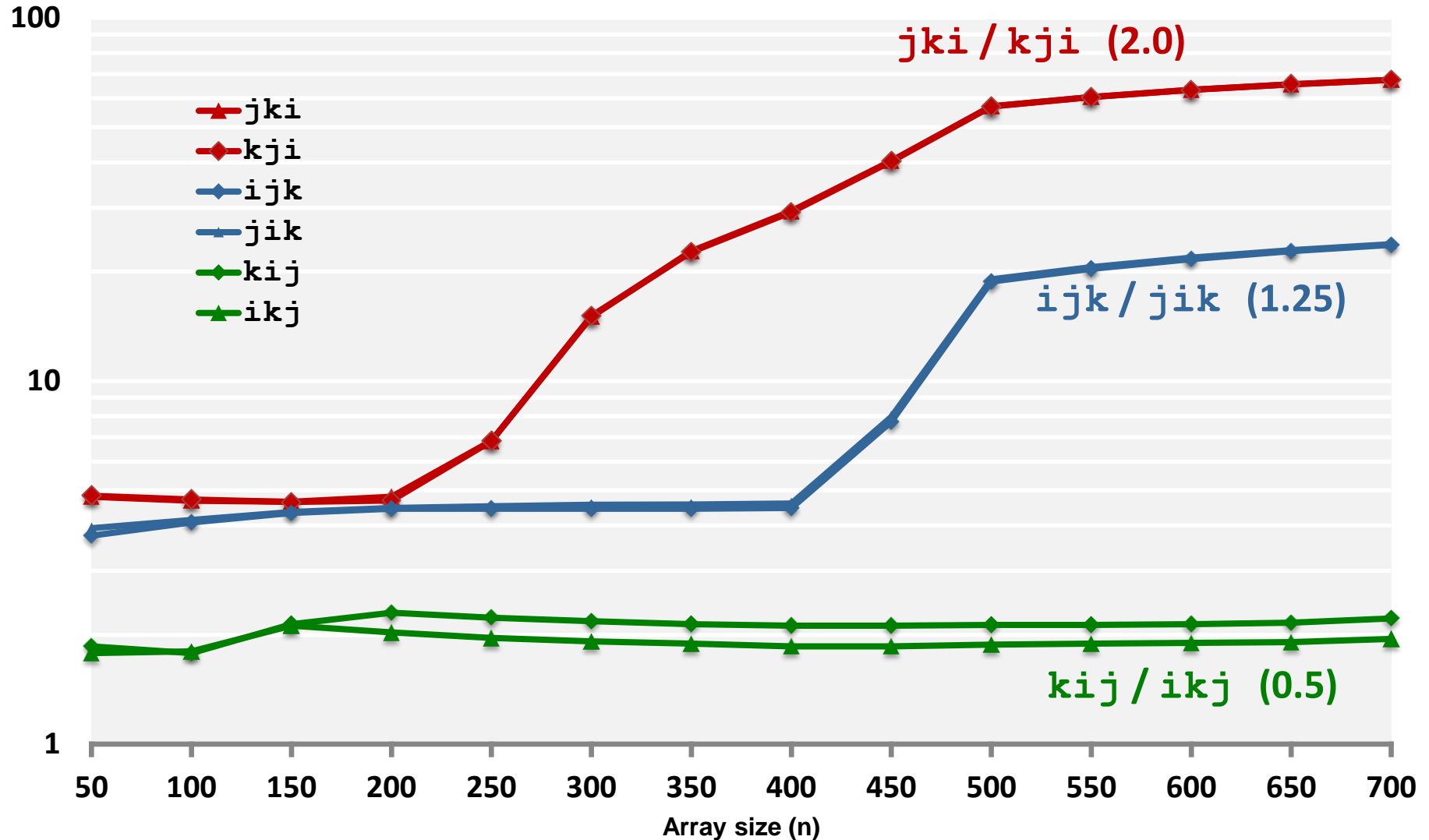
Legend:
- jki
- kji
- ijk
- jik
- kij
- ikj

jki / kji (2.0)

ijk / jik (1.25)

kij / ikj (0.5)

Array size (n)

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

51

Today

- **Cache organization and operation**

- **Performance impact of caches**
  - The memory mountain
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

```
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```
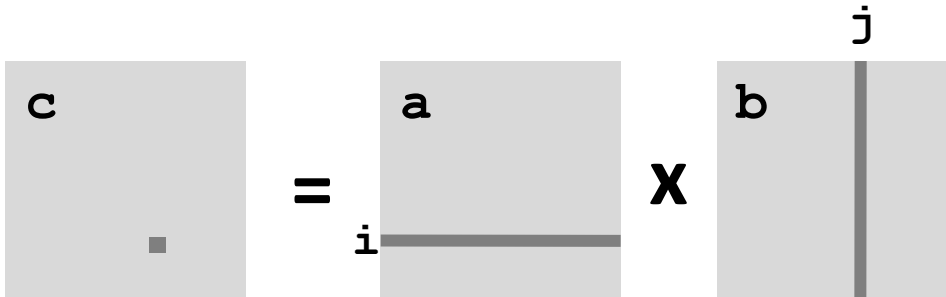
Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache line = 8 doubles
  - Cache size C << n (much smaller than n)

- **First iteration:**
  - $n/8 + n = 9n/8$ misses

  - Afterwards in cache:
    (schematic)
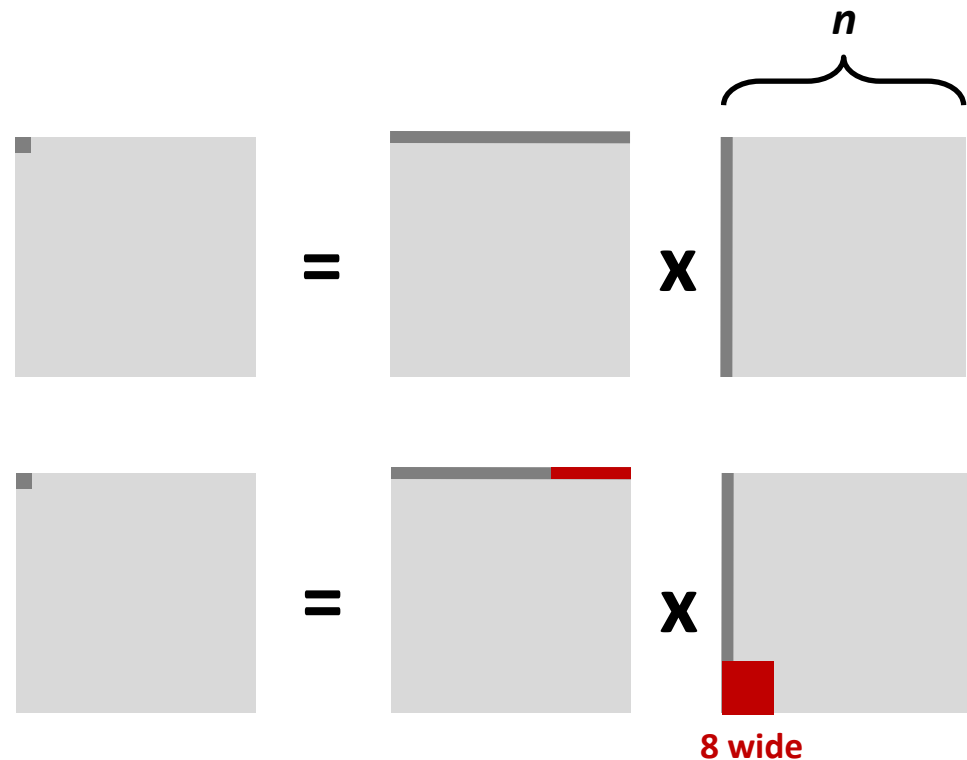
n

= X

= X

8 wide

Cache Miss Analysis

- **Assume:**
  - Matrix elements are doubles
  - Cache line = 8 doubles
  - Cache size C << n (much smaller than n)

- **Second iteration:**
  - Again:
    $n/8 + n = 9n/8$ misses

$$= \quad X$$

n

8 wide
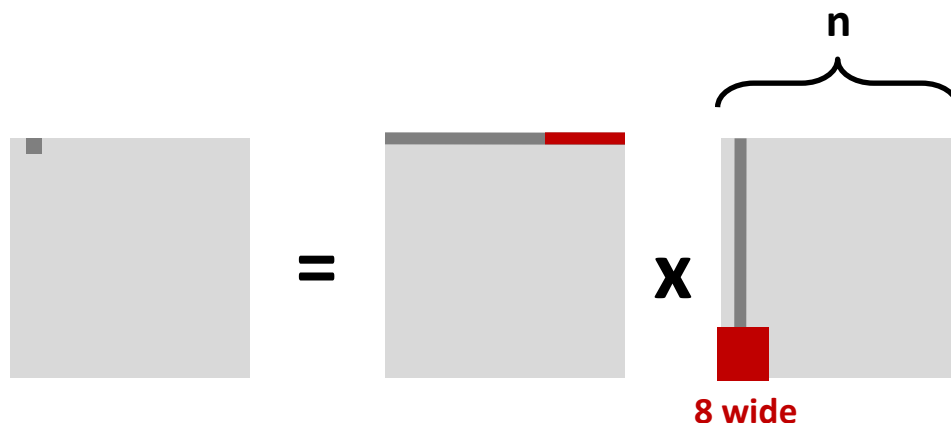
- **Total misses:**
  - $9n/8 \ n^2 = (9/8) \ n^3$

Blocked Matrix Multiplication

```c
c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b  */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i+=L)
        for (j = 0; j < n; j+=L)
            for (k = 0; k < n; k+=L)
                /* L x L mini matrix multiplications */
                for (i1 = i; i1 < i+L; i1++)
                    for (j1 = j; j1 < j+L; j1++)
                        for (k1 = k; k1 <  k+L; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```
matmult/bmm.c
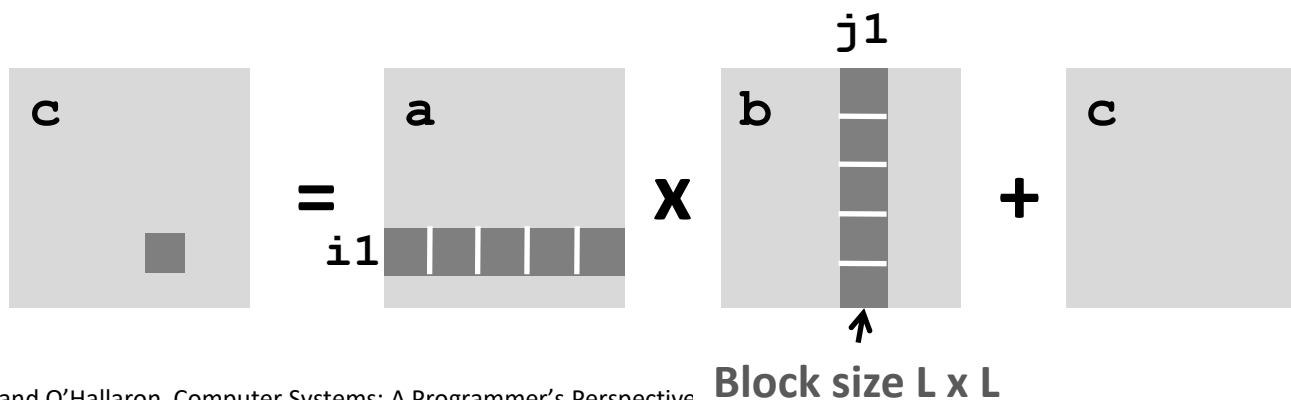
Block size L x L

Cache Miss Analysis

- **Assume:**
  - Cache line = 8 doubles. Blocking size L ≥ 8
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3L^2 < C$

- **First (block) iteration:**
  - **Misses per block:** $L^2/8$
  - **Blocks per Iteration:** $2n/L$ (omitting matrix c)
  - **Misses per Iteration:** $2n/L \times L^2/8 = nL/4$
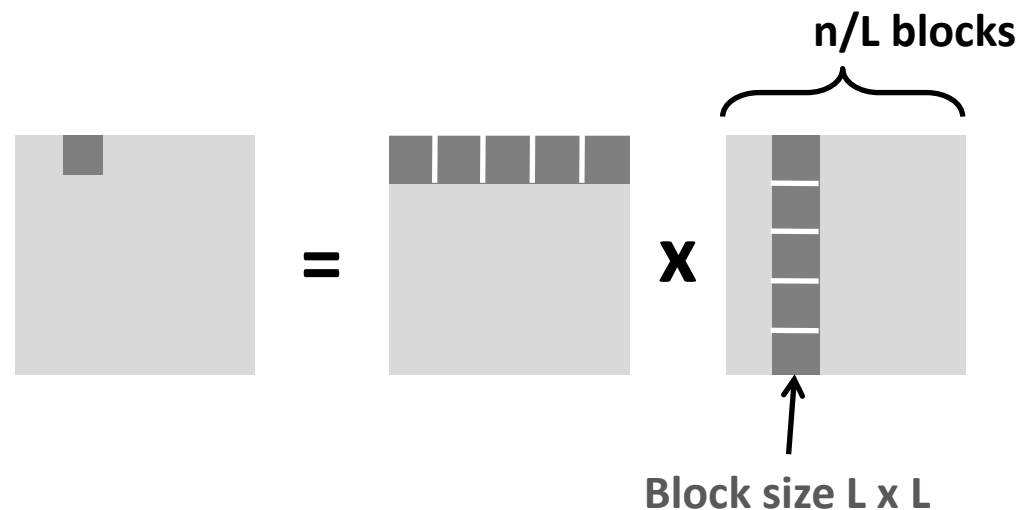
  - Afterwards in cache (schematic)
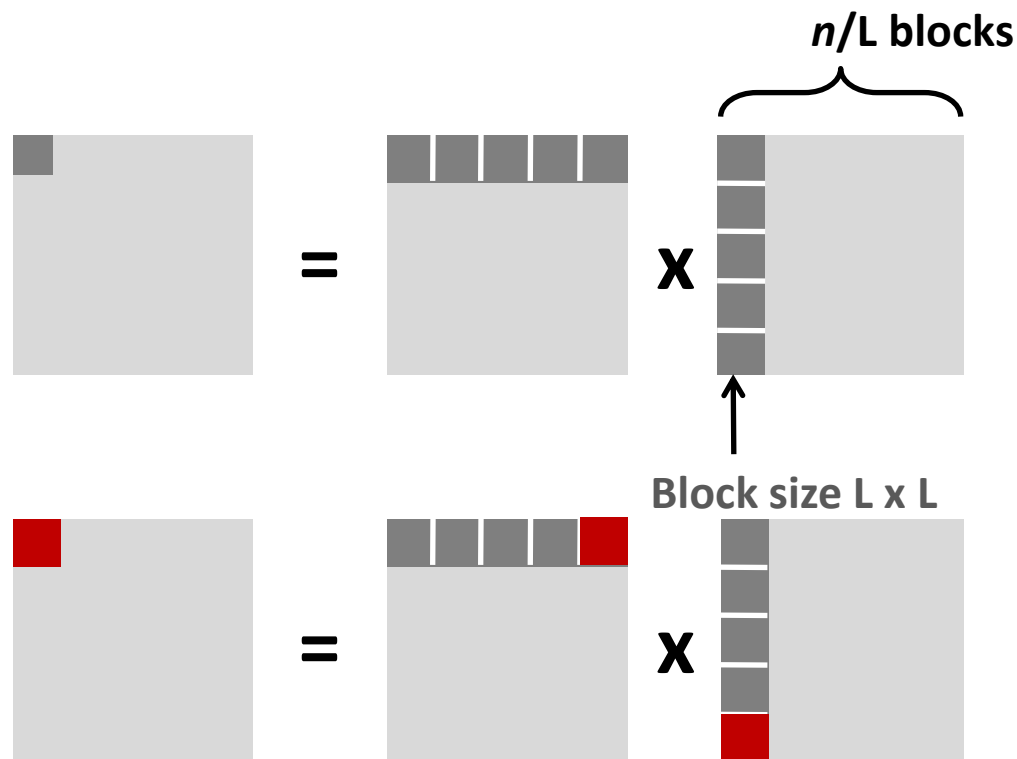
n/L blocks

Block size L x L

Cache Miss Analysis

- **Assume:**
  - Cache line = 8 doubles. Blocking size L ≥ 8
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3L^2 < C$

- **Second (block) iteration:**
  - Same misses as first iteration
  - $2n/L \times L^2/8 = nL/4$

n/L blocks

= **X**

Block size L x L

- **Total misses:**
  - $nL/4$ misses per iteration x $(n/L)^2$ iterations = $n^3/(4L)$ misses

Blocking Summary

- **No blocking: (9/8) n^3 misses**

- **Blocking: (1/(4L)) n^3 misses**

- **Use largest block size L, such that L satisfies $3L^2 < C$**
  - Fit three blocks in cache! Two input, one output.

- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
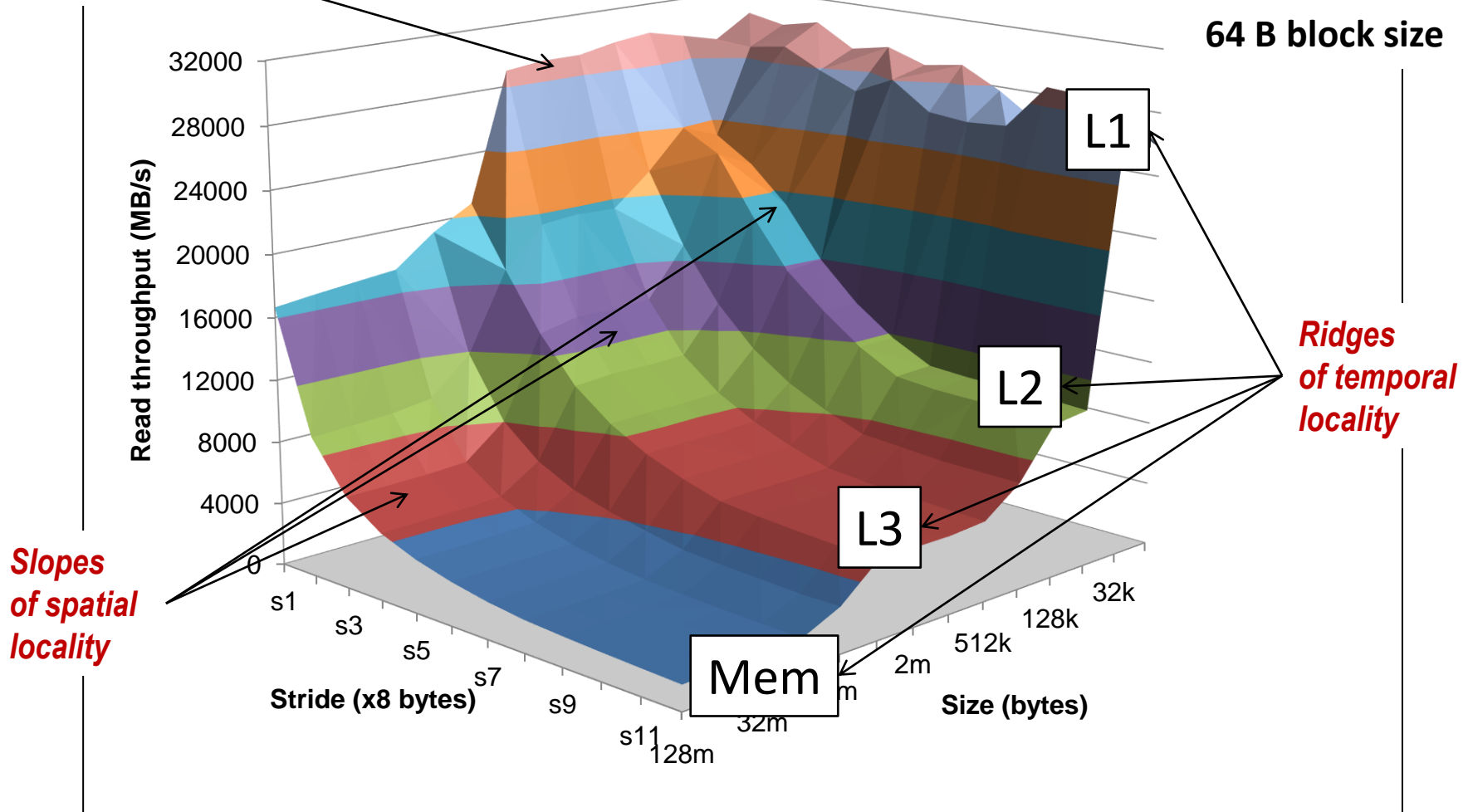 - But program has to be written properly

Cache Summary

- **Cache memories can have significant performance impact**

- **You can write your programs to exploit this!**
  - Focus on the inner loops, where bulk of computations and memory accesses occur.
  - Try to maximize spatial locality by reading data objects sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.
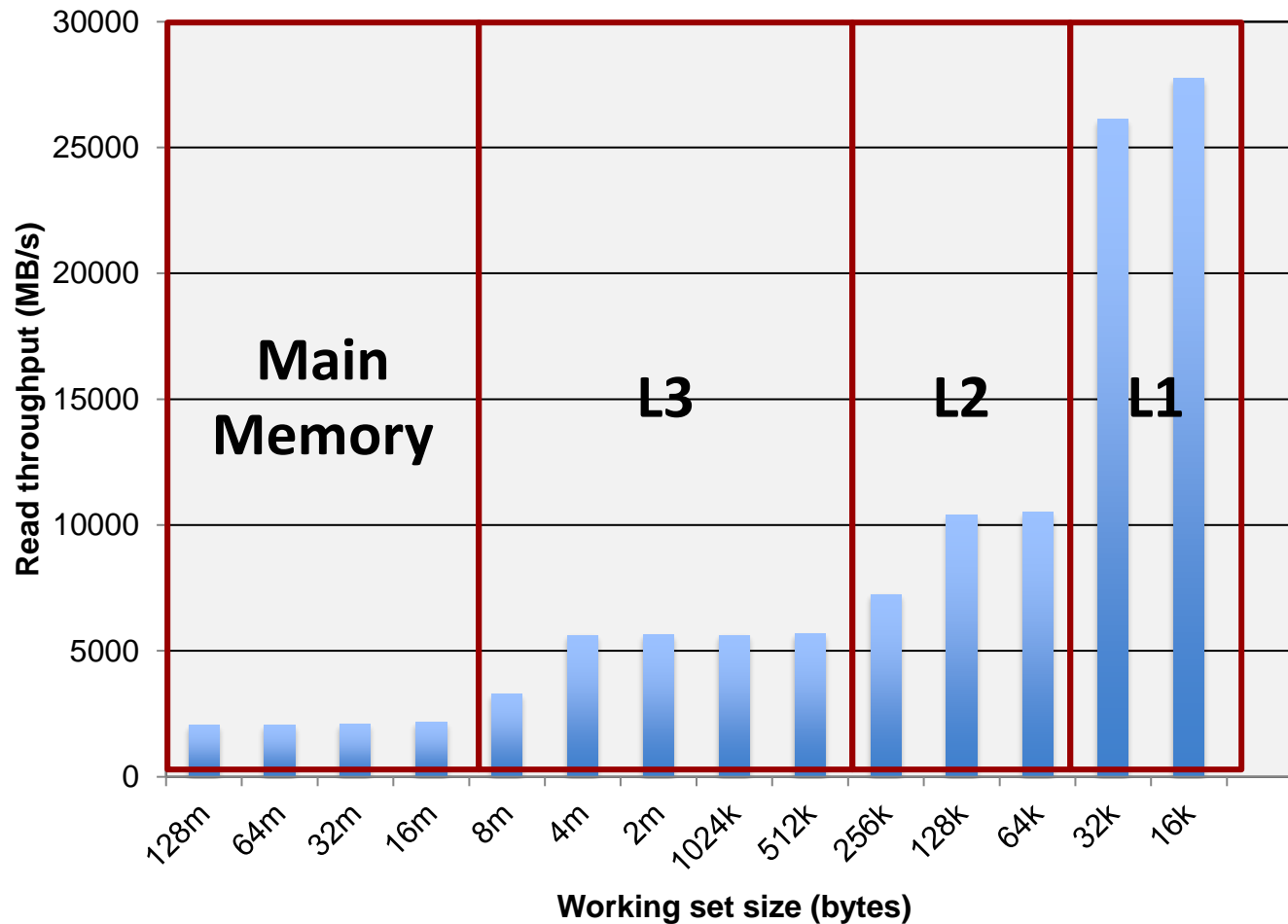
Supplemental slides

Cache Capacity Effects from Memory Mountain

Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

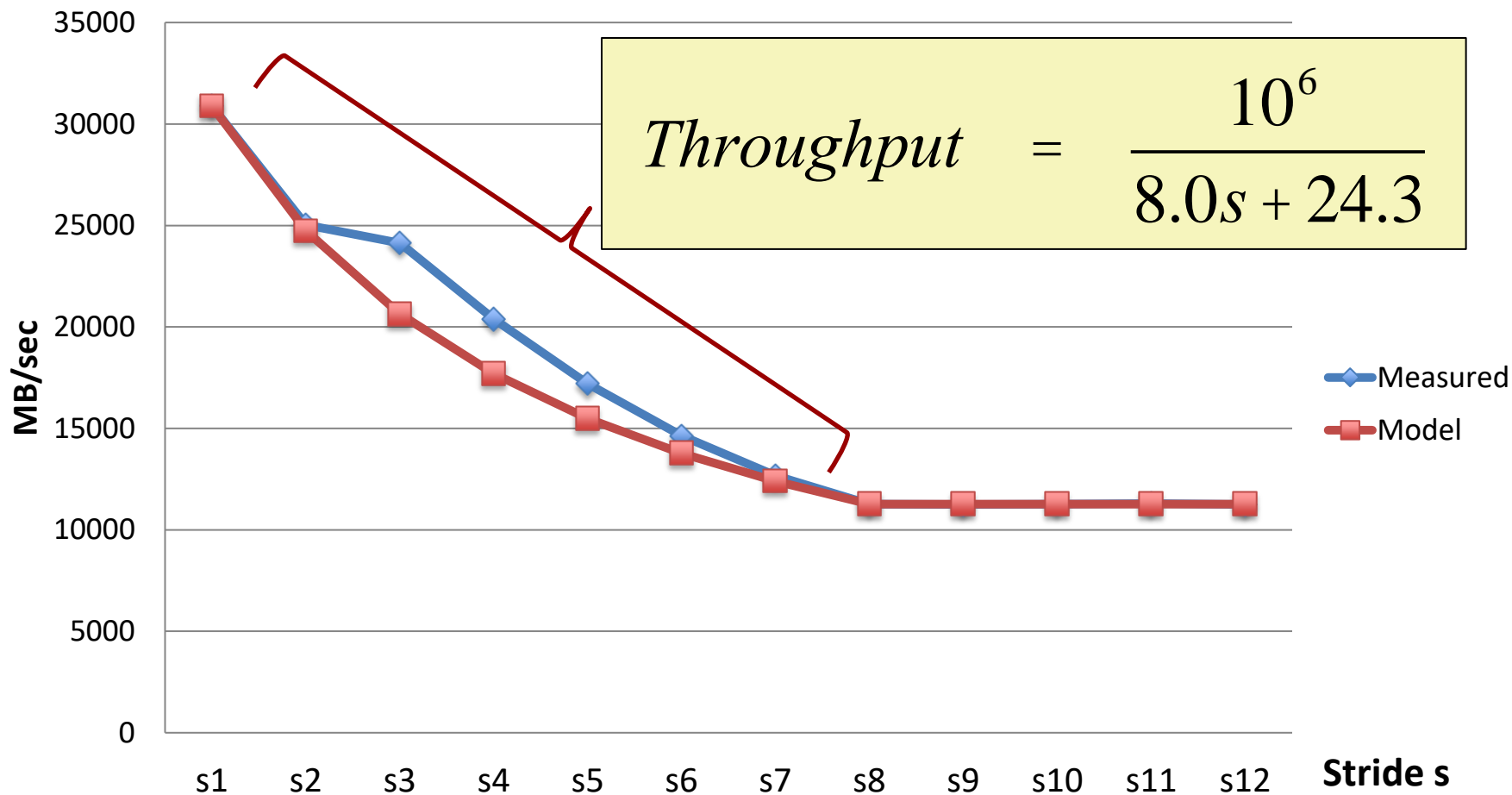Slice through memory mountain with stride=8
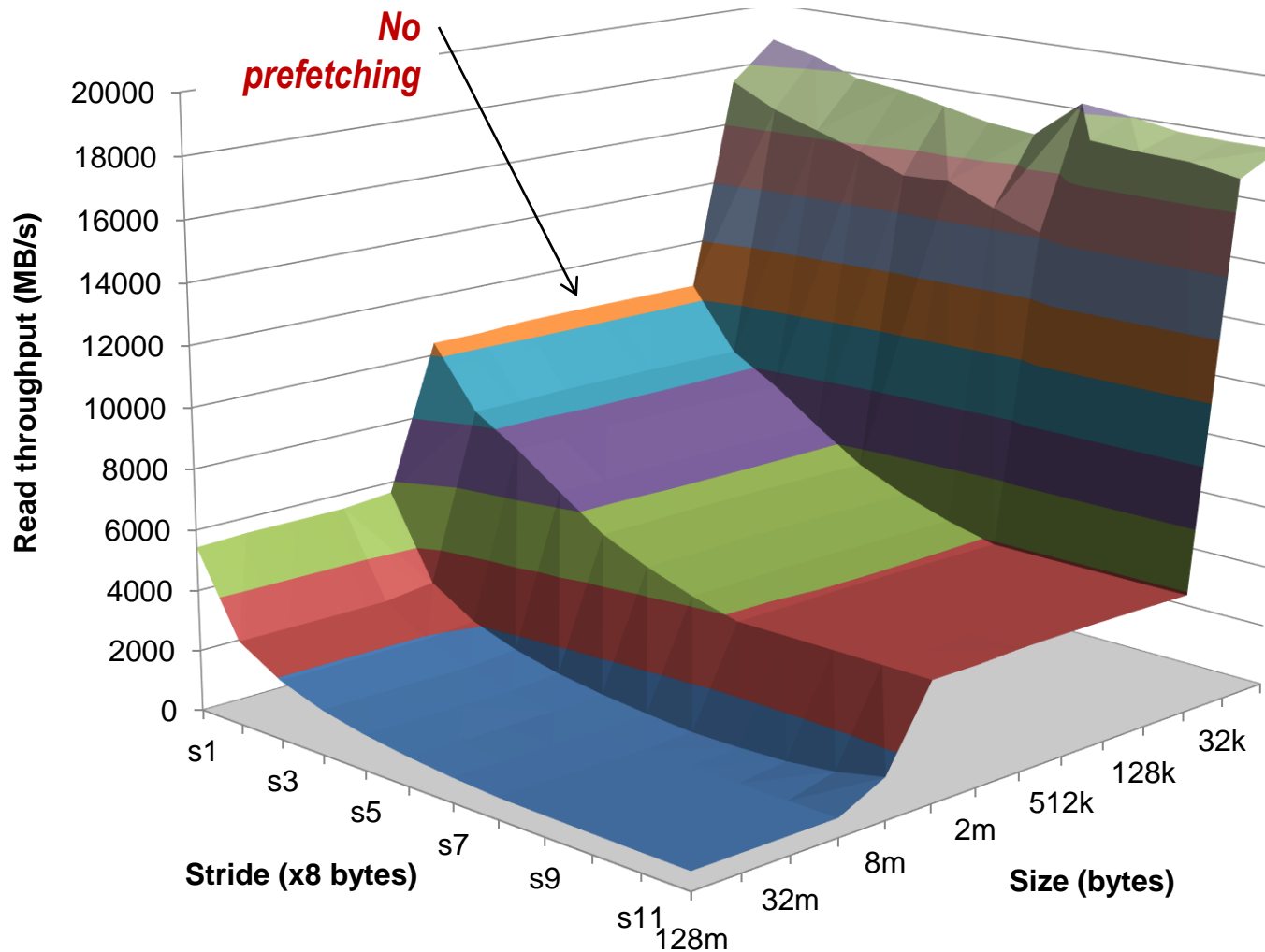
Modeling Block Size Effects from Memory Mountain

Core i7 Haswell
2.26 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Throughput for size = 128K

$$Throughput = \frac{10^6}{8.0s + 24.3}$$

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

2008 Memory Mountain

Core 2 Duo
2.4 GHz
32 KB L1 d-cache
6MB L2 cache
64 B block size

No prefetching

Read throughput (MB/s)

20000
18000
16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes): s1, s3, s5, s7, s9, s11

Size (bytes): 32k, 128k, 512k, 2m, 8m, 32m, 128m
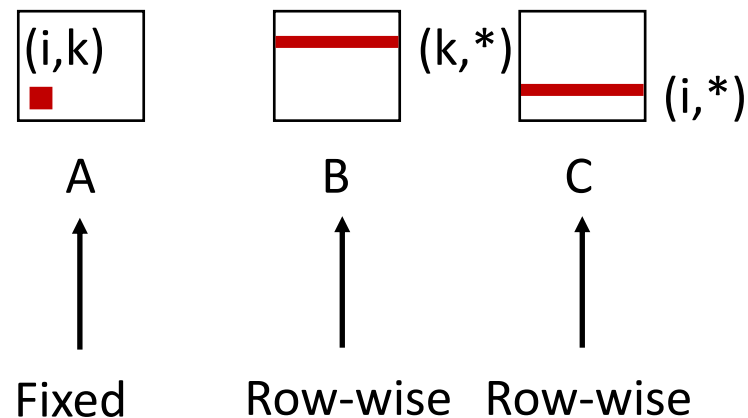
Matrix Multiplication (`ikj`)

```
/* 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];
  }
}
                        matmult/mm.c
```

Inner loop:

A	B	C
Fixed	Row-wise	Row-wise

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Block size = 32B (four doubles)
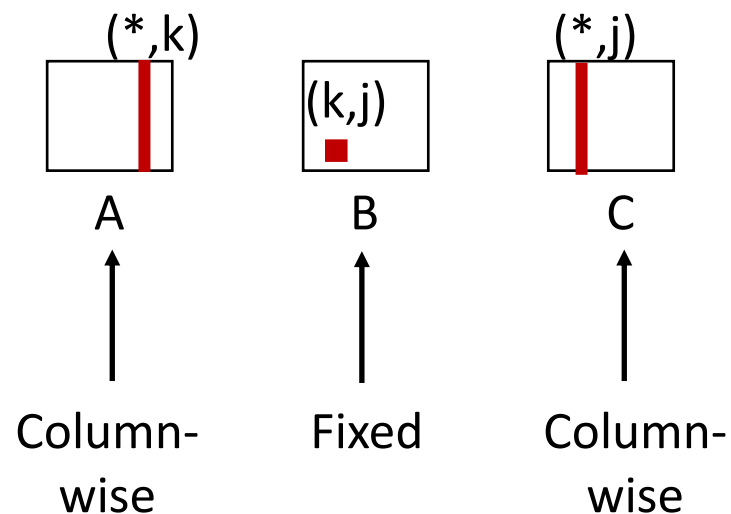
Matrix Multiplication (`kji`)

```
/* 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;
  }
}
                            matmult/mm.c
```

Inner loop:

(*,k)   (k,j)   (*,j)

A   B   C

Column-   Fixed   Column-
wise   wise

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Block size = 32B (four doubles)