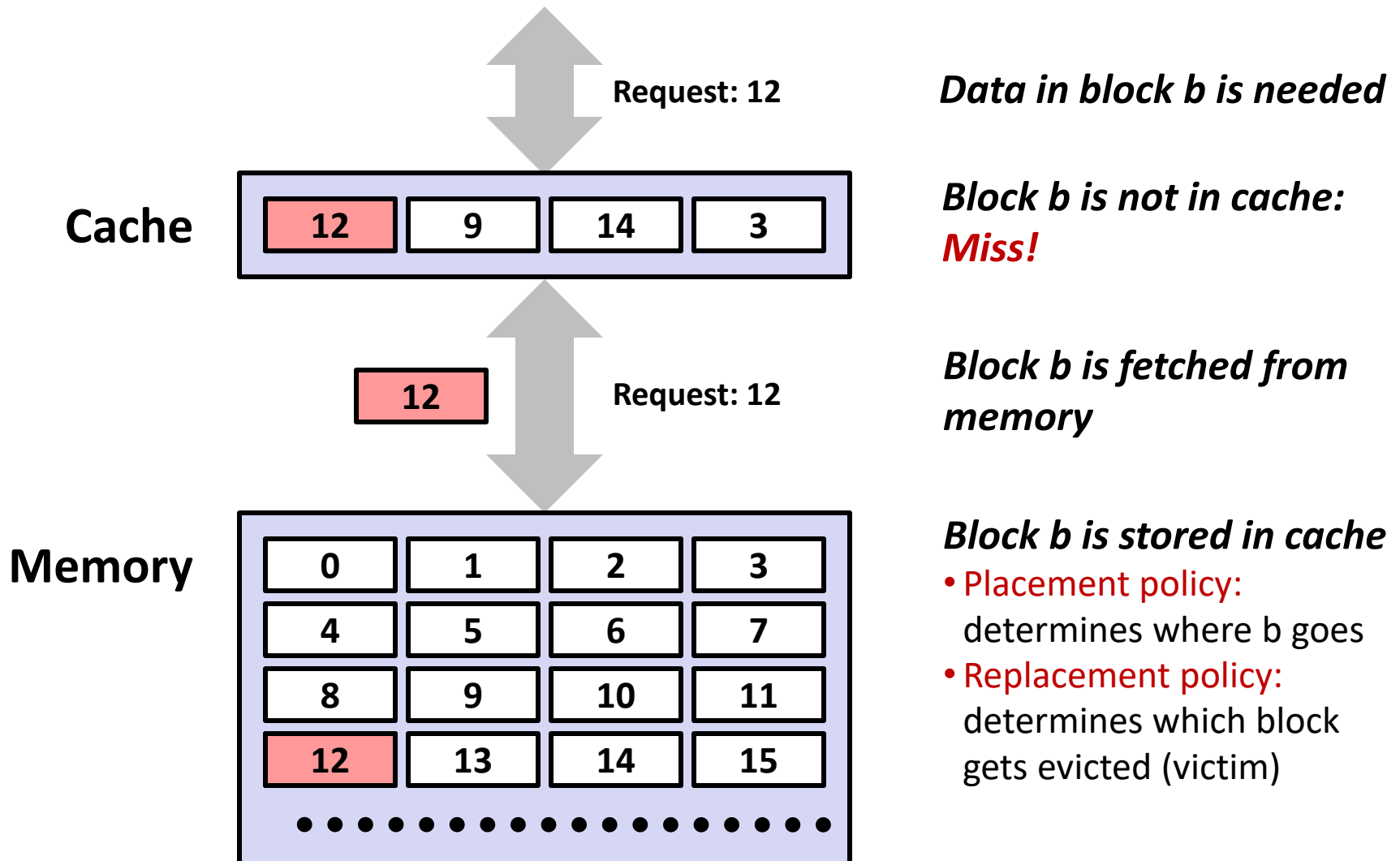# Cache Memories

15-213/14-513/15-513: Introduction to Computer Systems
10th Lecture, September 28, 2023

Today

- **Cache memory organization and operation**   **CSAPP 6.4-6.5**
- **Performance impact of caches**
    - The memory mountain   **CSAPP 6.6.1**
    - Rearranging loops to improve spatial locality   **CSAPP 6.6.2**
    - Using blocking to improve temporal locality   **CSAPP 6.6.3**

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

Cache

| 8 | 9 | 14 | 3 |

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 in cache:
Hit!

General Cache Concepts: Miss

Request: 12

Data in block b is needed

Cache

| 12 | 9 | 14 | 3 |

Block b is not in cache:
Miss!

| 12 |

Request: 12

Block b is fetched from memory

Memory

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

● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ● ●

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

Working Set, Locality, and Caches

- **Working Set: The set of data a program is currently "working on"**
    - Definition of "currently" depends on context, e.g., in this loop
    - Includes accesses to data and instructions

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

- **Caches take advantage of temporal locality by storing recently used data, and spatial locality by copying data in block-sized transfer units**
    - Locality reduces working set sizes
    - Caches are most effective when the working set fits in the cache

Recall: 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.
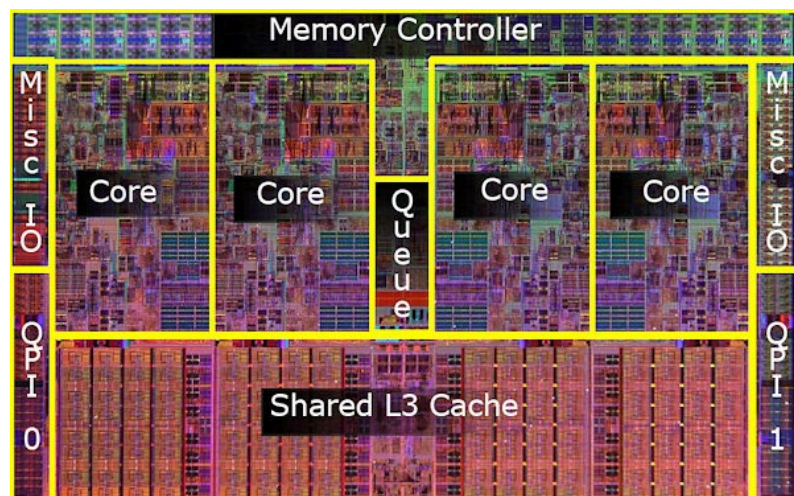
- **Conflict miss**
  - Occurs when the cache is large enough, but too many data objects all map (by the placement policy) to the same limited set of blocks
    - E.g., if the placement policy maps both 0 and 8 to the same block, then referencing 0, 8, 0, 8, 0, 8, ... would miss every time.

CPU Cache Memories

- **CPU 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**
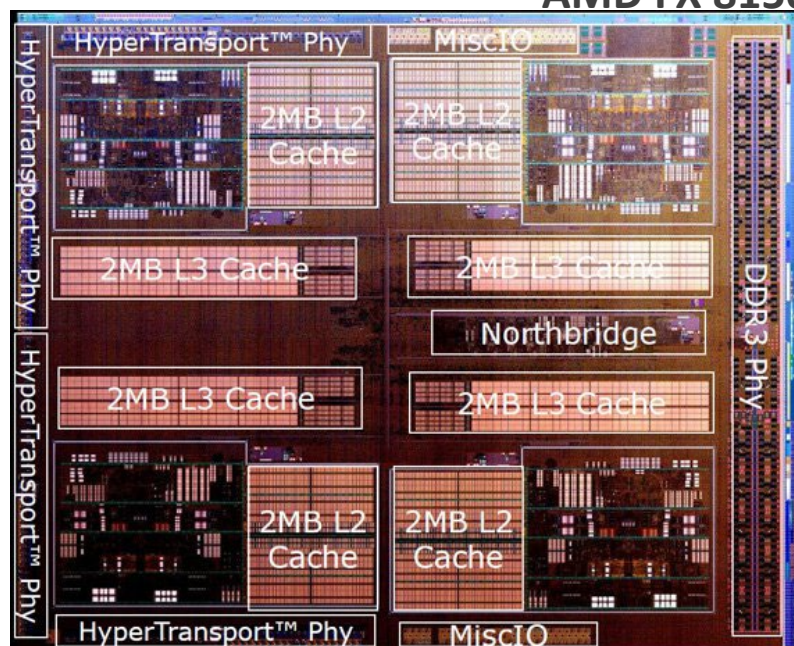- **Typical system structure:**

CPU chip

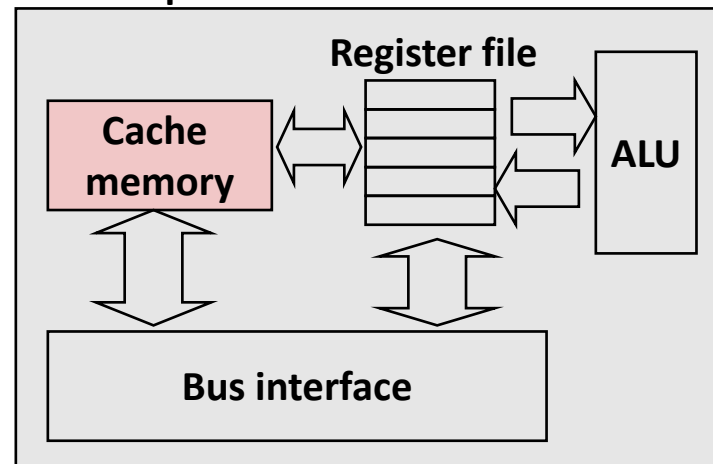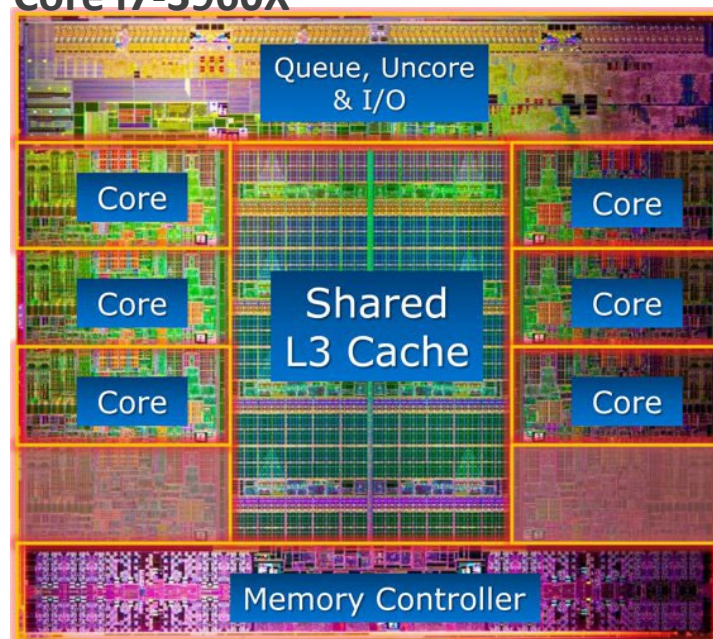Cache memory — Register file — ALU — Bus interface

System bus — Memory bus — I/O bridge — Main memory

What it Really Looks Like

CPU chip

Nehalem

AMD FX 8150

Core i7-3960X

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

12
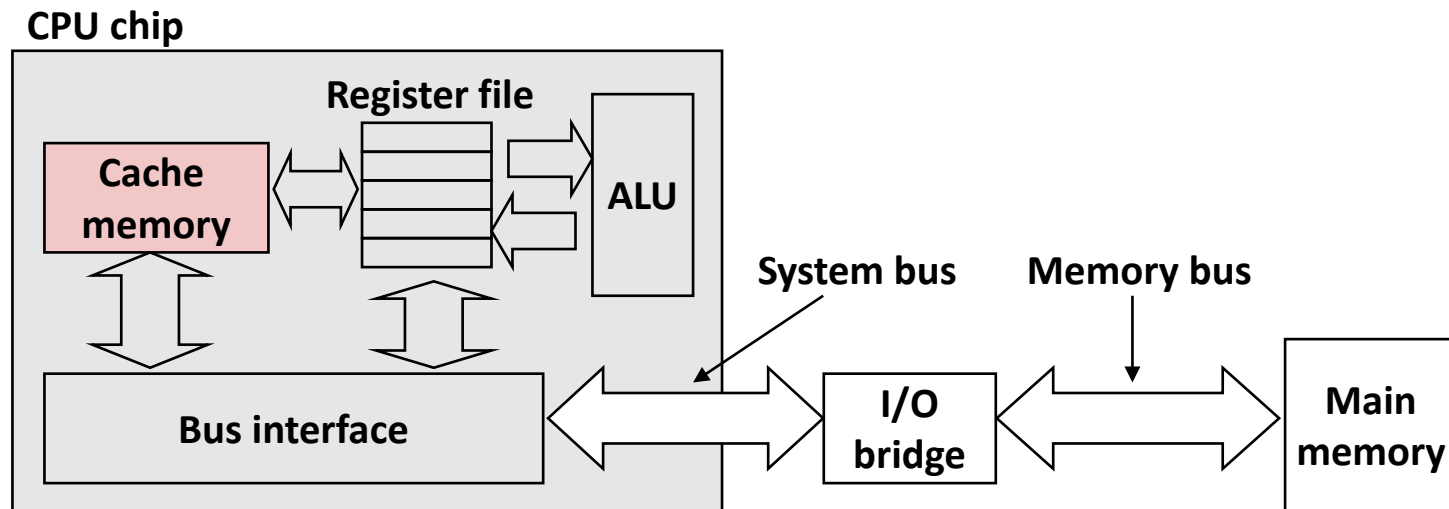
General Cache Organization (S, E, B)

E = 2^e lines per set

set

line

S = 2^s sets

Cache size

= S x E x B data bytes

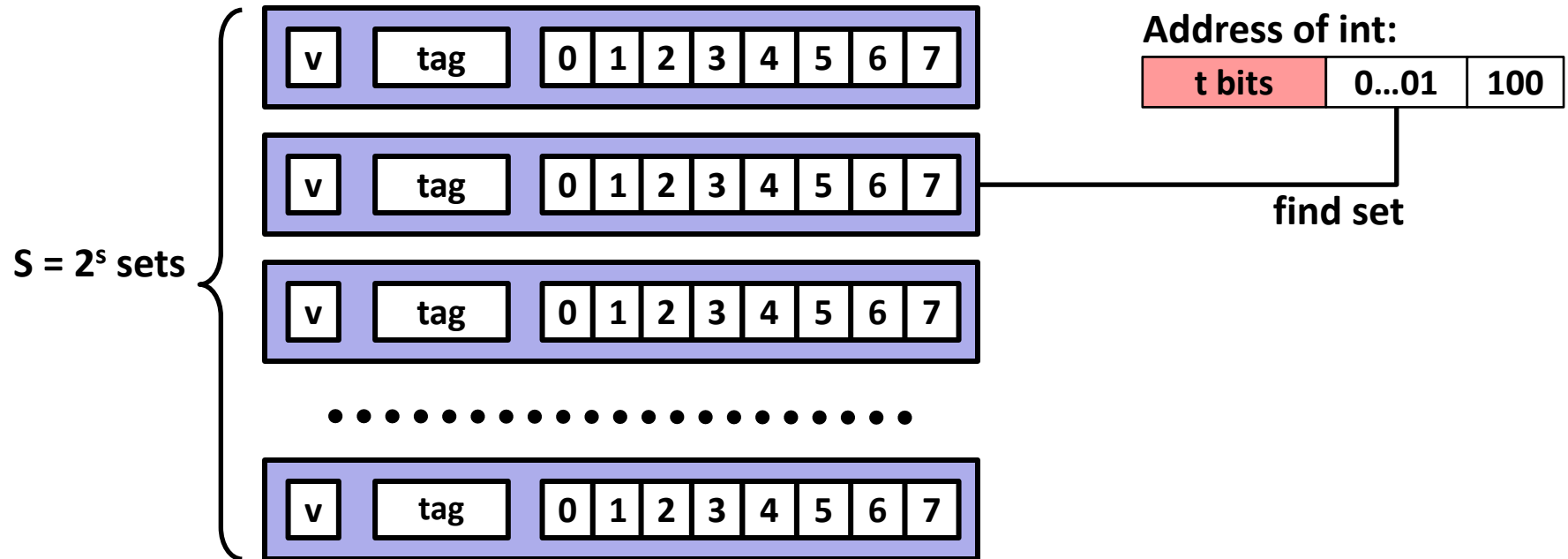| v | tag | 0 | 1 | 2 | ...... | B-1 |

valid bit

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

13
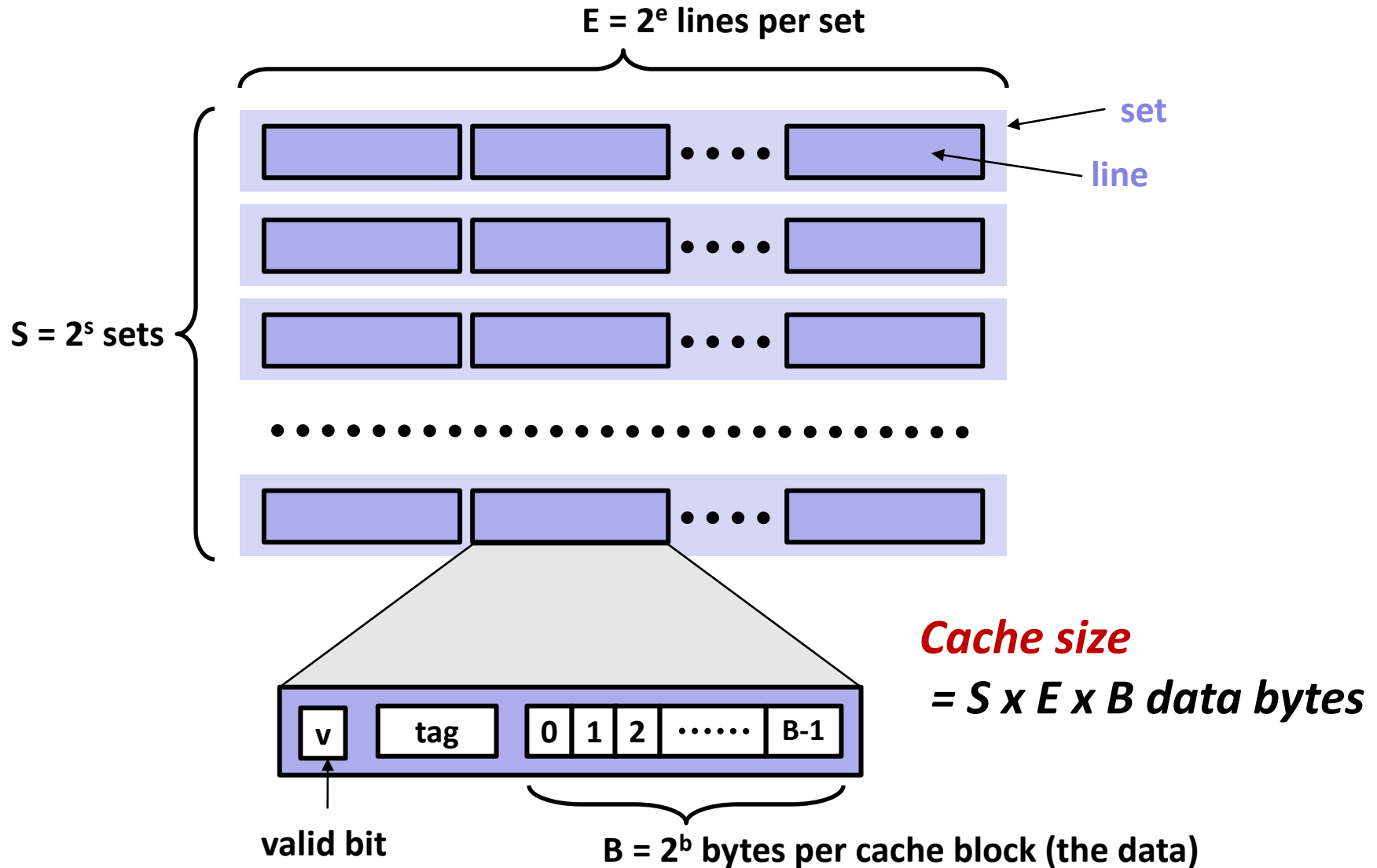
Cache Read
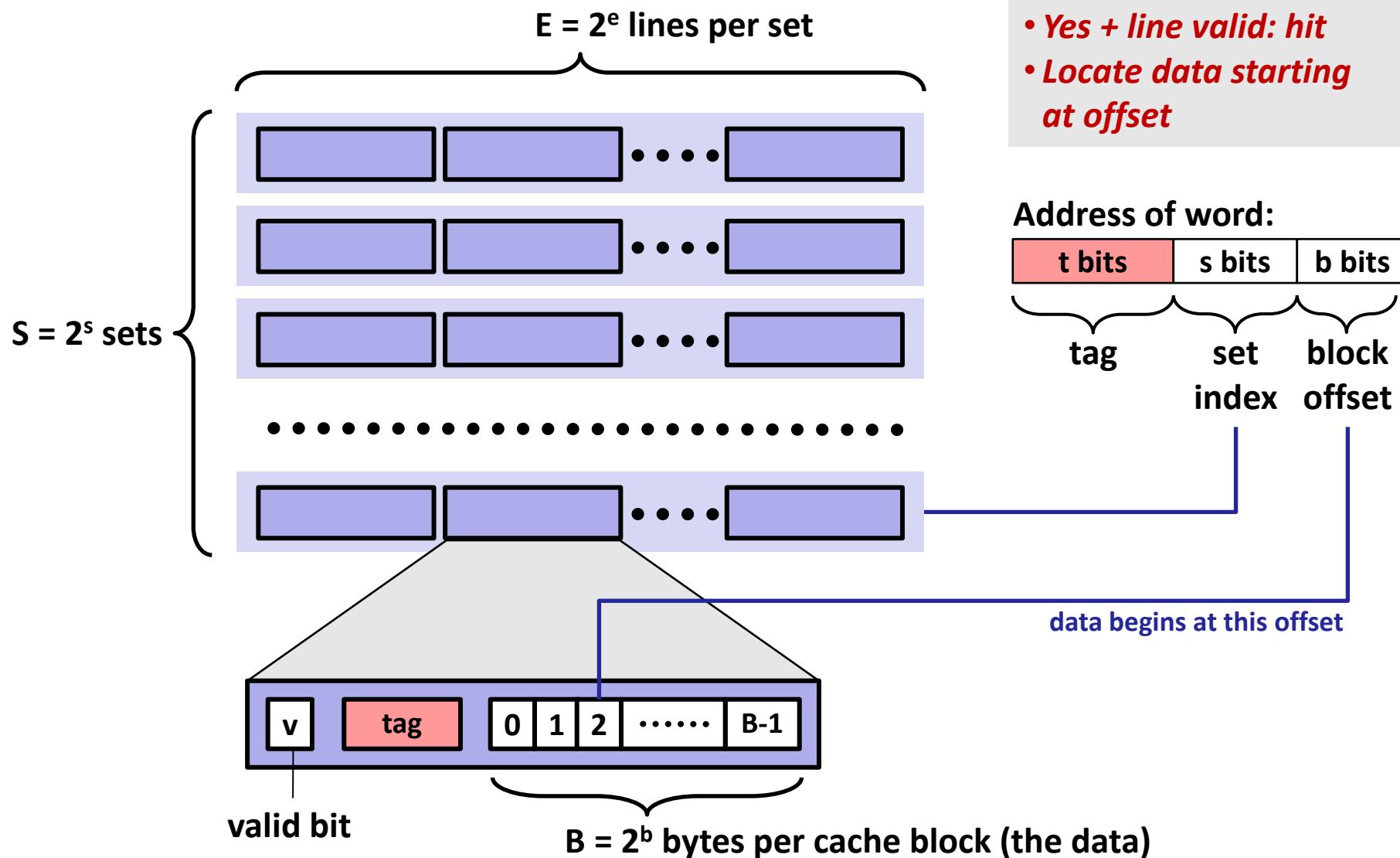
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

$E = 2^e$ **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

Address of int:

t bits	0...01	100

find set

$S = 2^s$ sets

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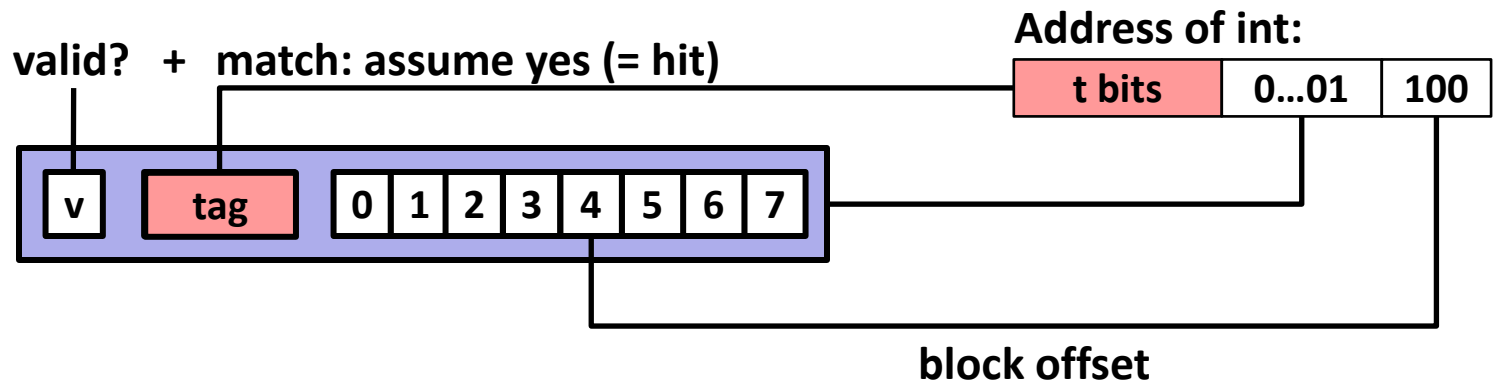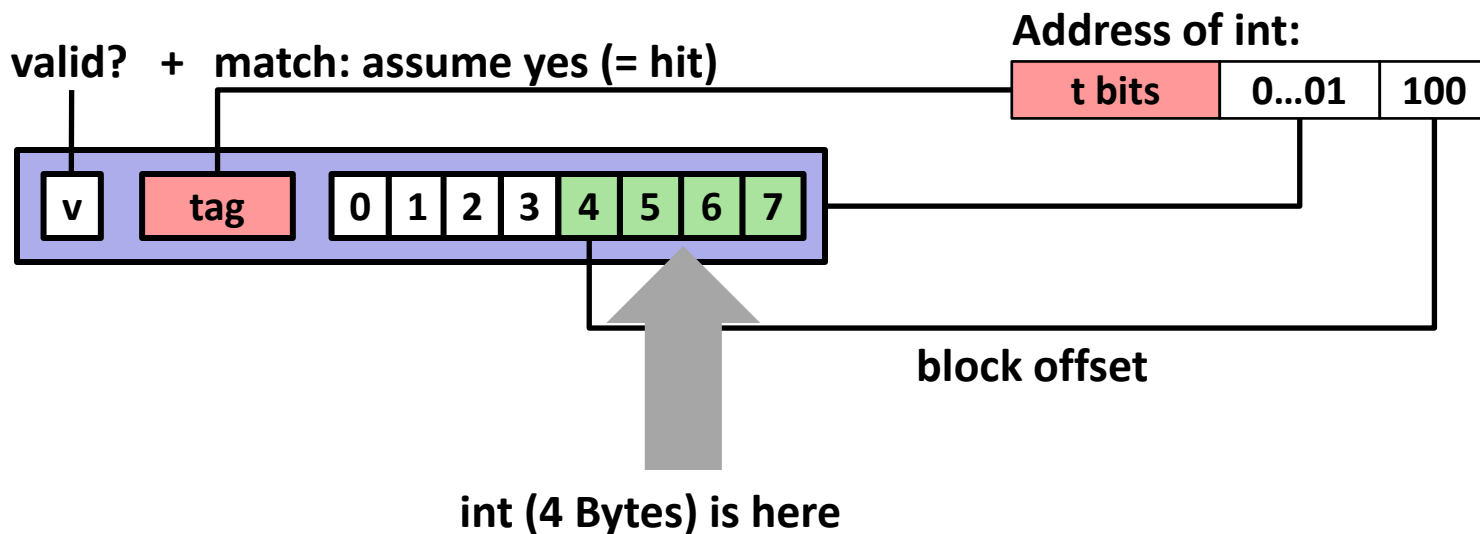
If tag doesn't match (= miss): old line is evicted and replaced

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

2 lines per set

Address of short int:
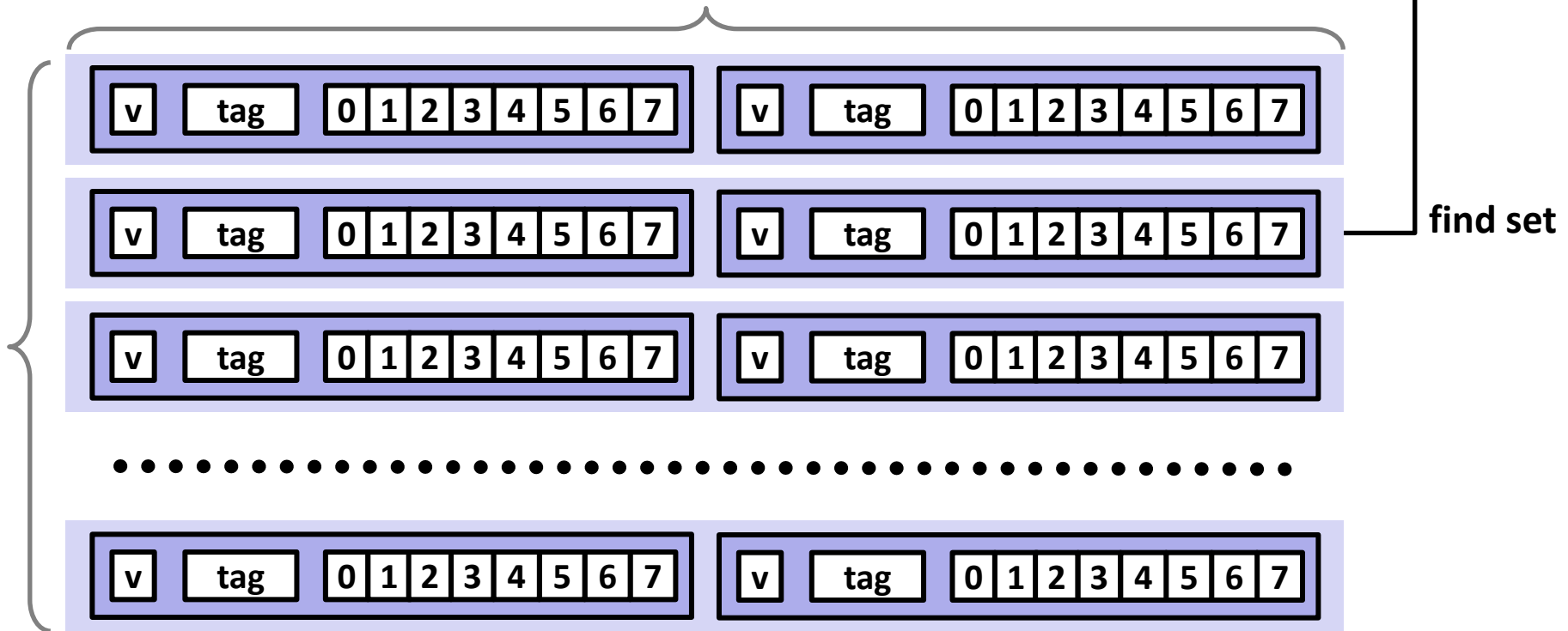
t bits	0...01	100

find set

S sets

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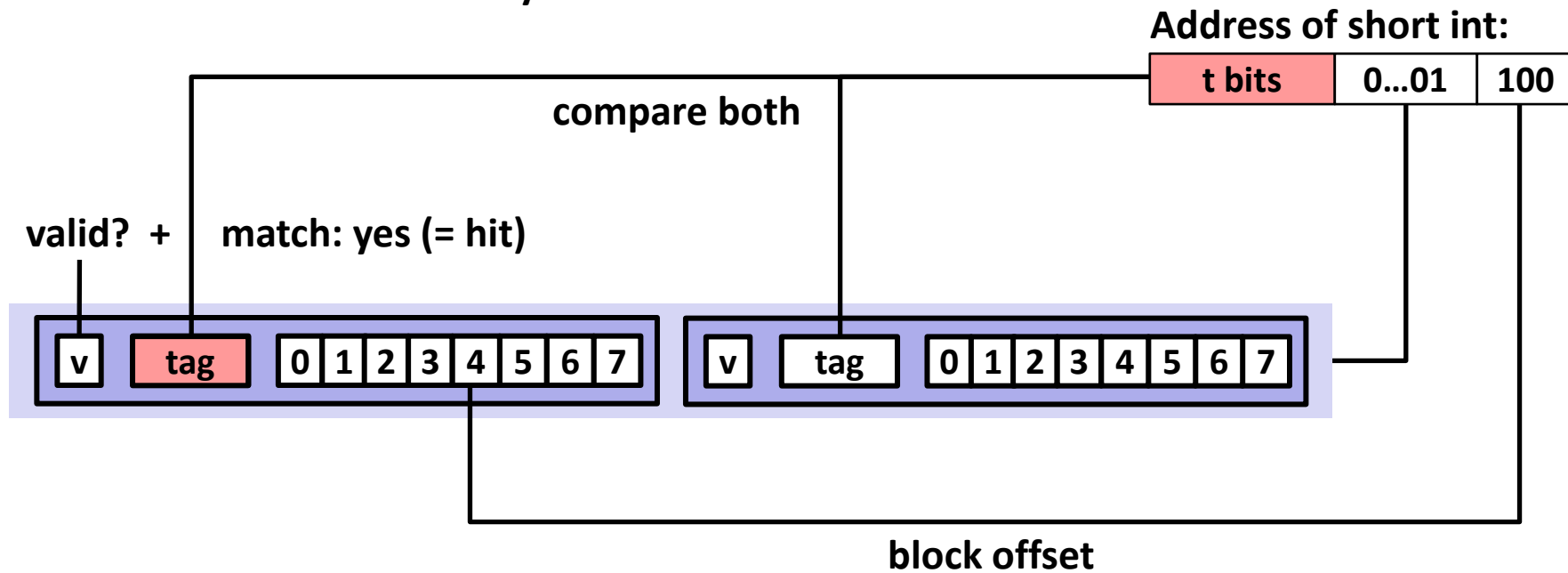
E = 2: Two lines per set

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
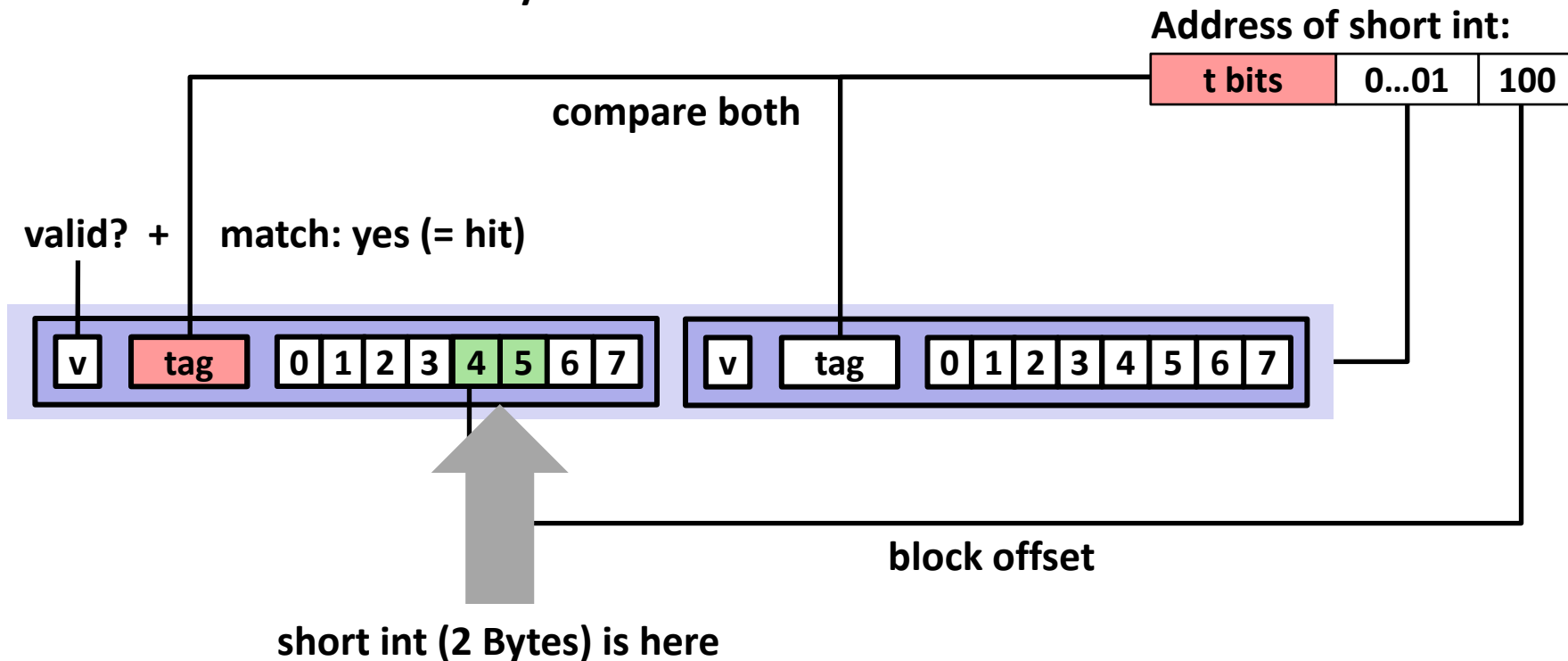
No match or not valid (= miss):

- **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	[0000_2],	miss
1	[0001_2],	hit
7	[0111_2],	miss
8	[1000_2],	miss
0	[0000_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		

What about writes?

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

Cache line diagram:

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

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 has been written to)

- **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**
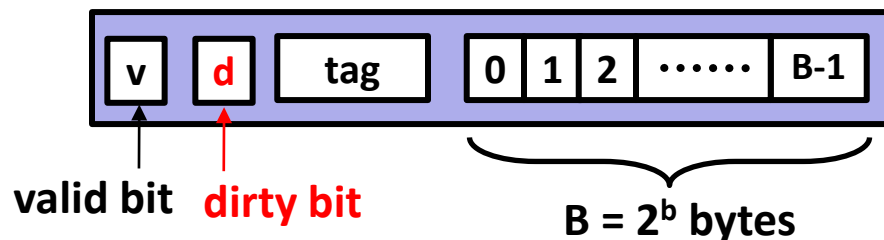
Practical Write-back Write-allocate

- **A write to address X is issued**

- **If it is a hit**
  - Update the contents of block
  - Set dirty bit to 1 (bit is sticky and only cleared on eviction)

- **If it is a miss**
  - Fetch block from memory (per a read miss)
  - Then perform the write operations (per a write hit)
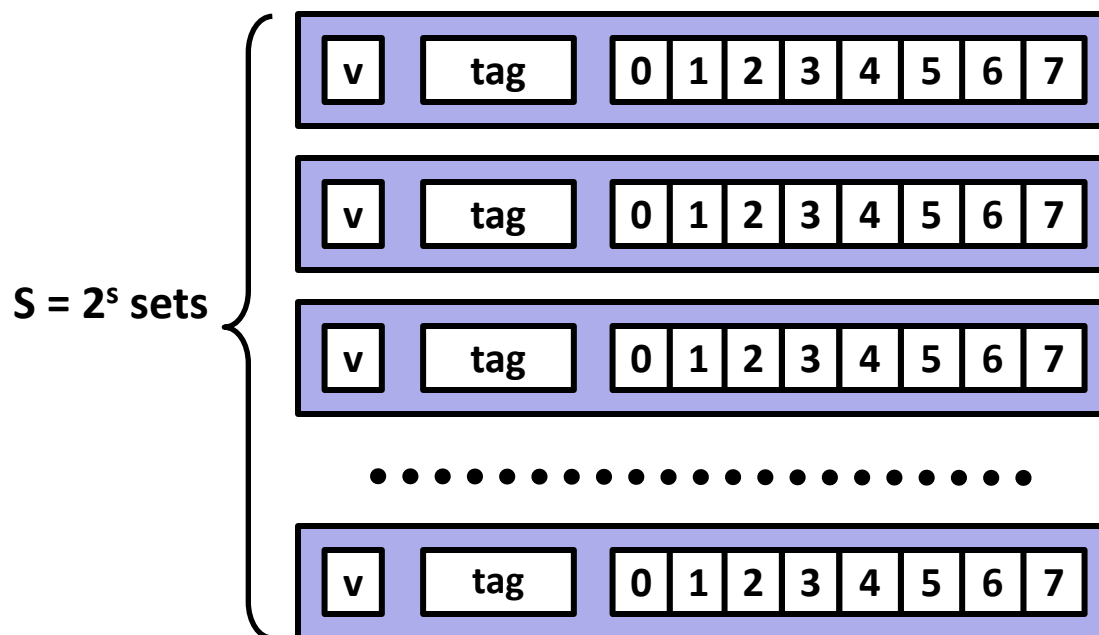
- **If a line is evicted and dirty bit is set to 1**
  - The entire block of 2^b bytes are written back to memory
  - Dirty bit is cleared (set to 0)
  - Line is replaced by new contents

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

valid bit dirty bit

B = 2^b bytes
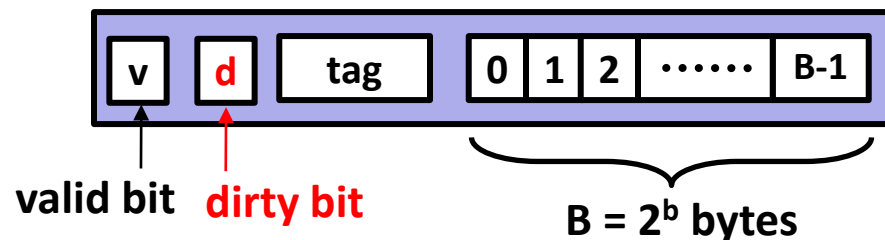
Why Index Using Middle Bits?

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

S = 2^s sets

Standard Method:
Middle bits indexing

Address of int:

| t bits | 0...01 | 100 |
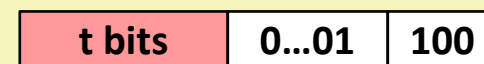
find set

Alternative Method:
High bits indexing
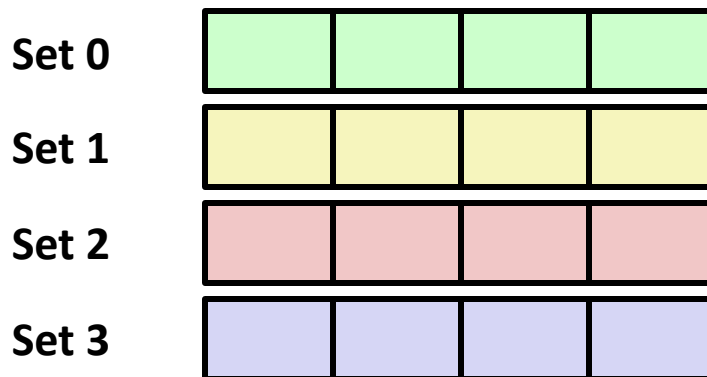
Address of int:

| 1...11 | t bits | 100 |
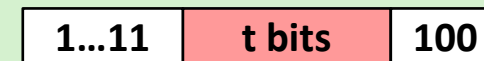
find set

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 Bits 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
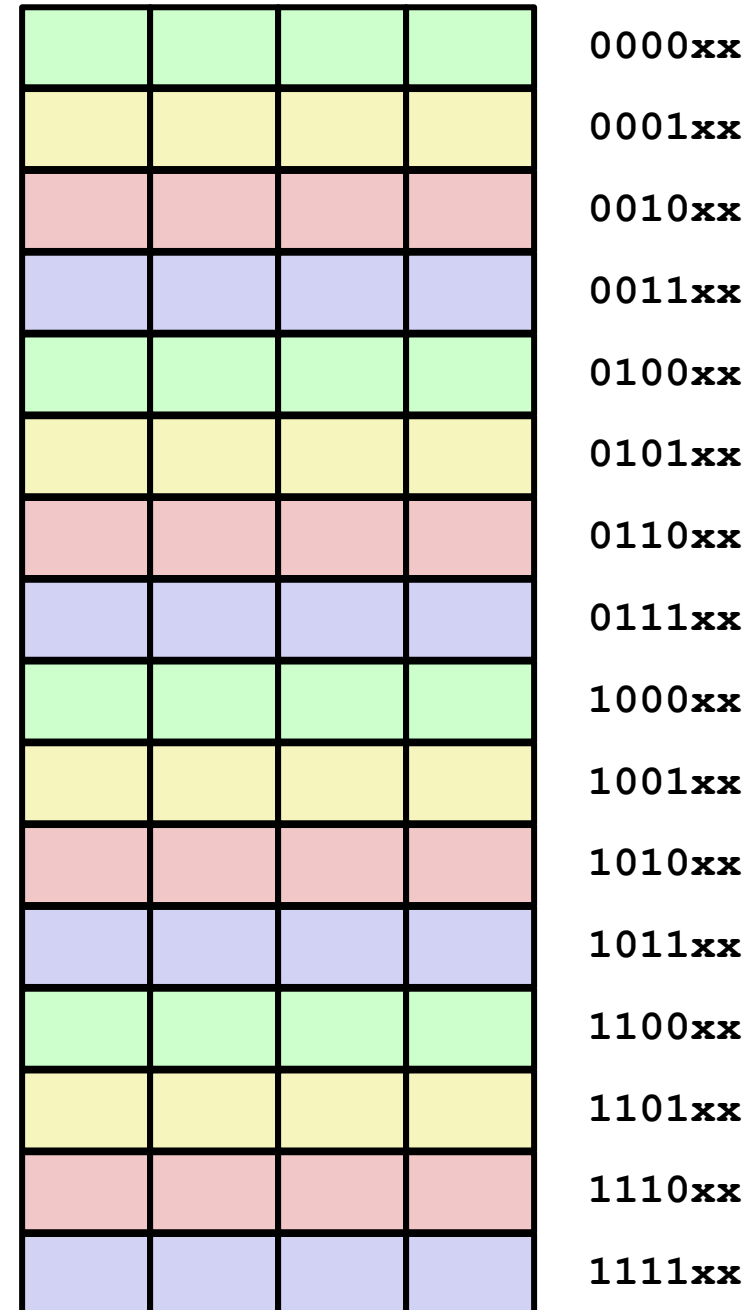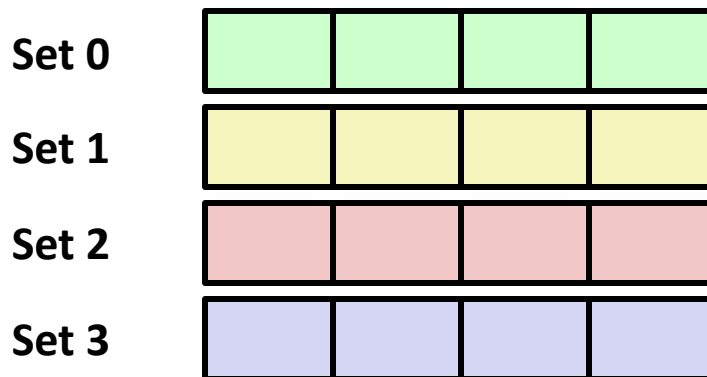
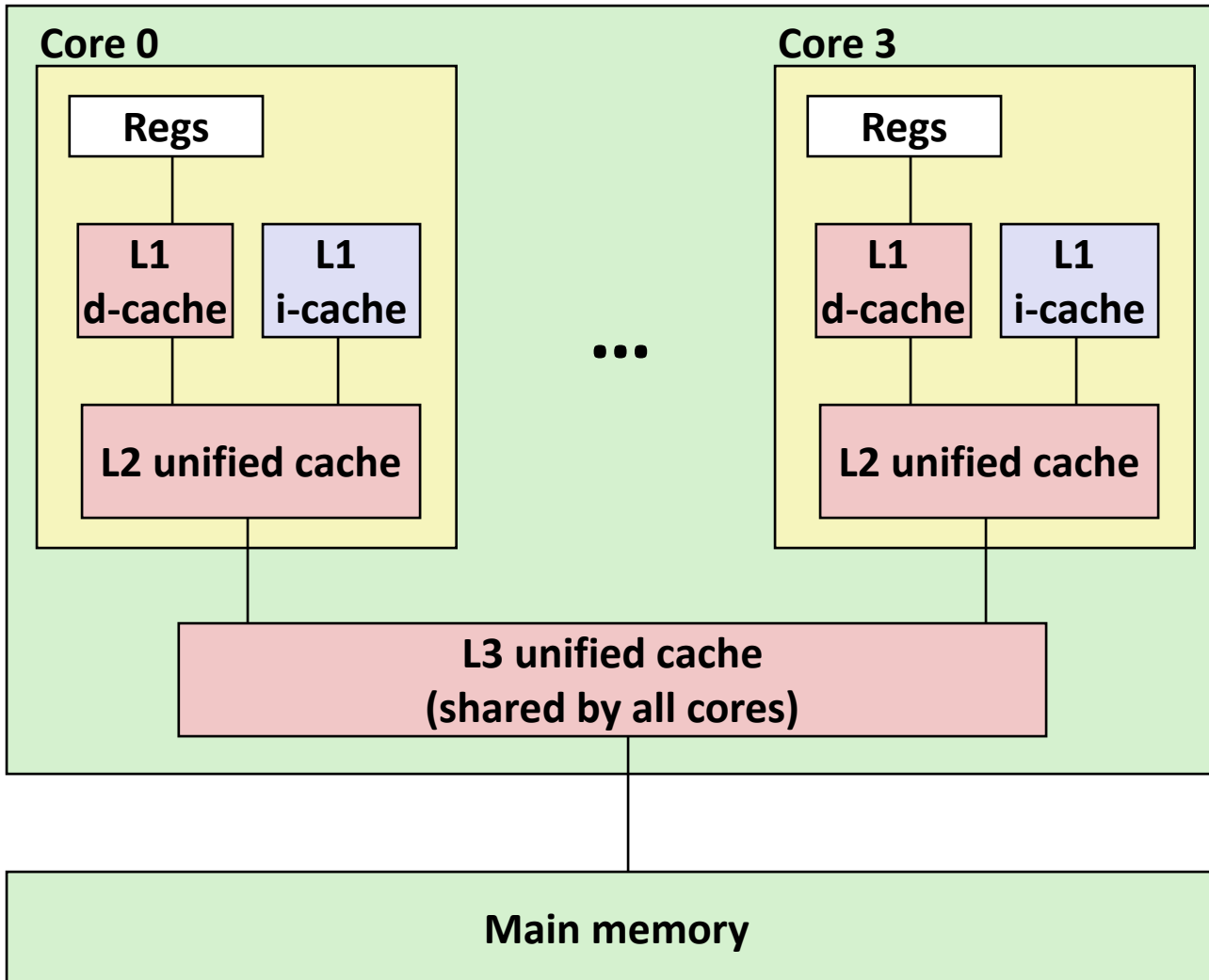# High Bits Indexing

- **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

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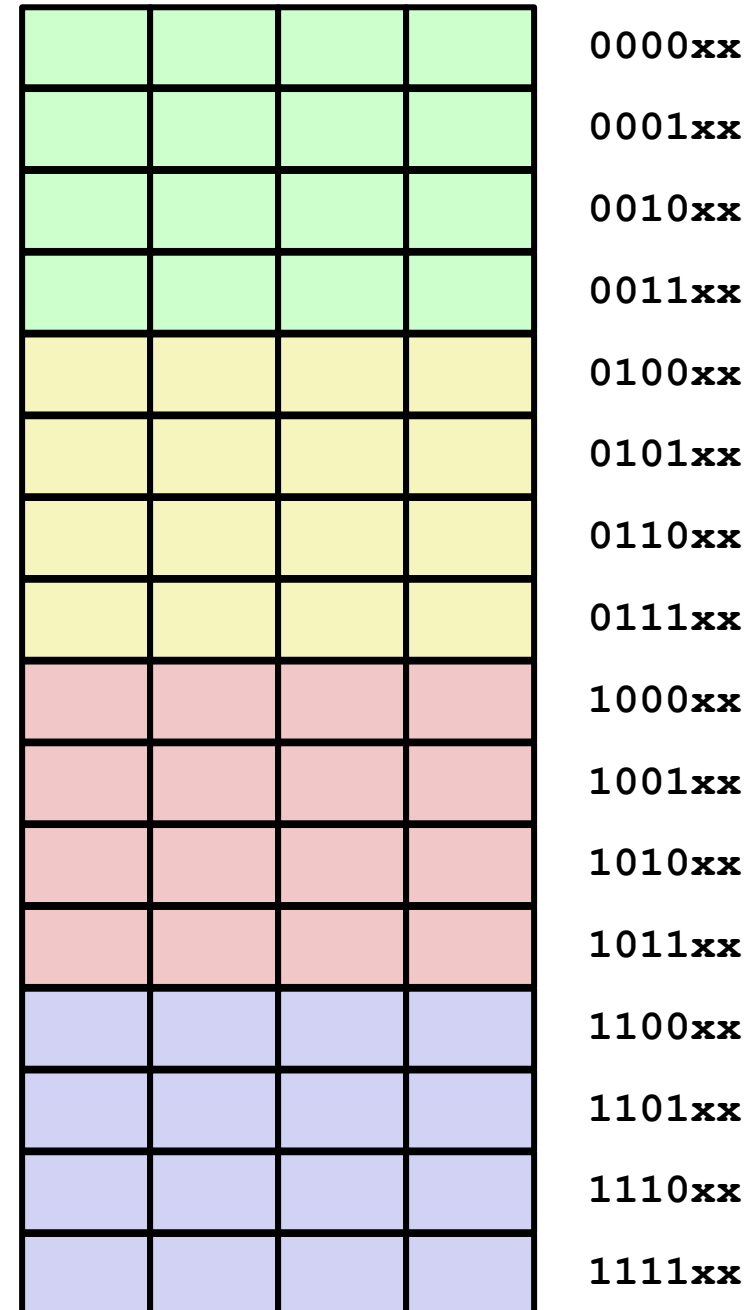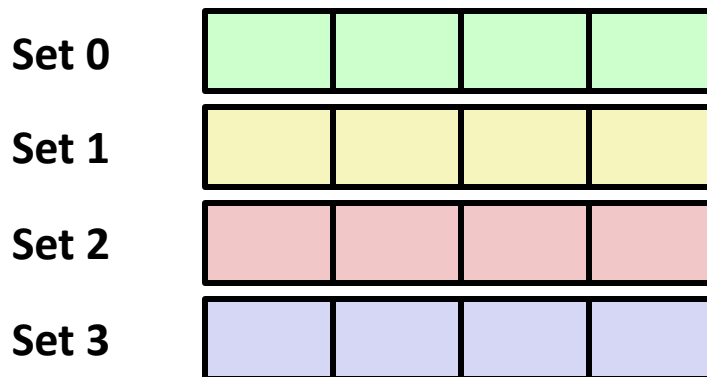
Intel Core i7 Cache Hierarchy

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.

Cache Performance Metrics

- **Miss Rate**
  - Fraction of memory references not found in cache (misses / accesses) = 1 – hit rate
  - Typical numbers (in percentages):
    - 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:
    - 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!)

Let's think about those numbers

- **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!

Canvas Quiz: 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
```

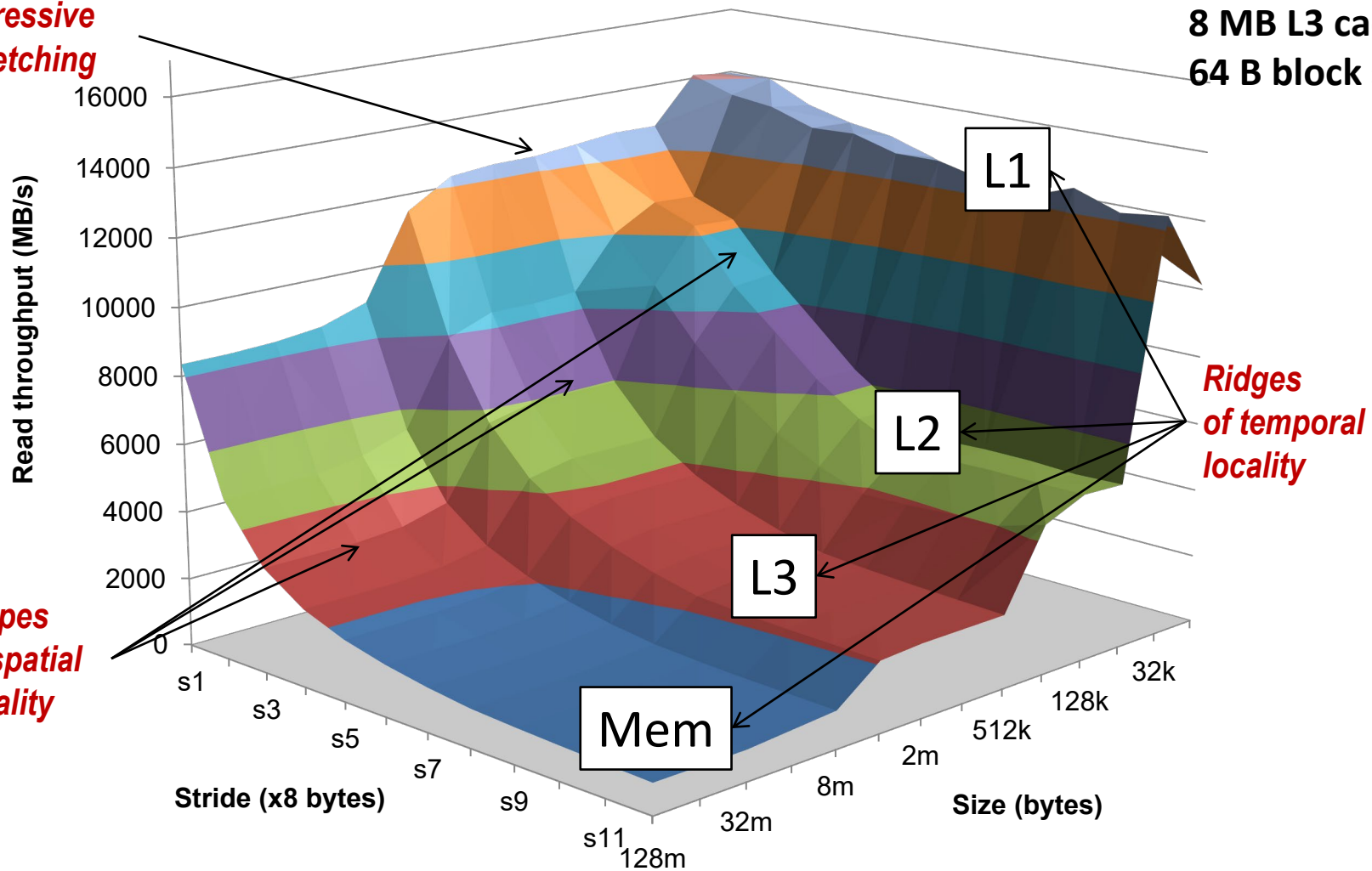Call `test()` with many combinations of `elems` and `stride`.

For each `elems` and `stride`:

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

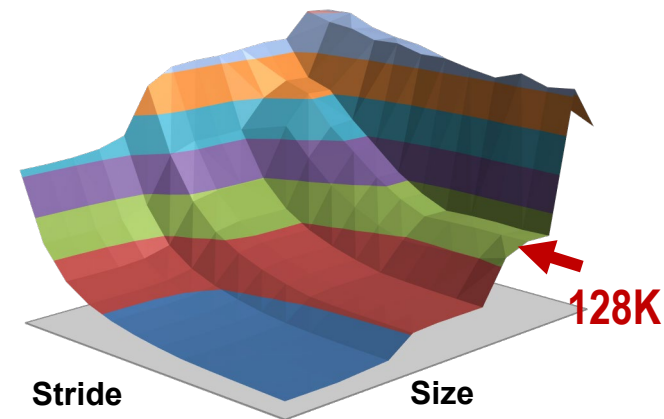2. Call `test()` again and measure the read throughput(MB/s)

The Memory Mountain

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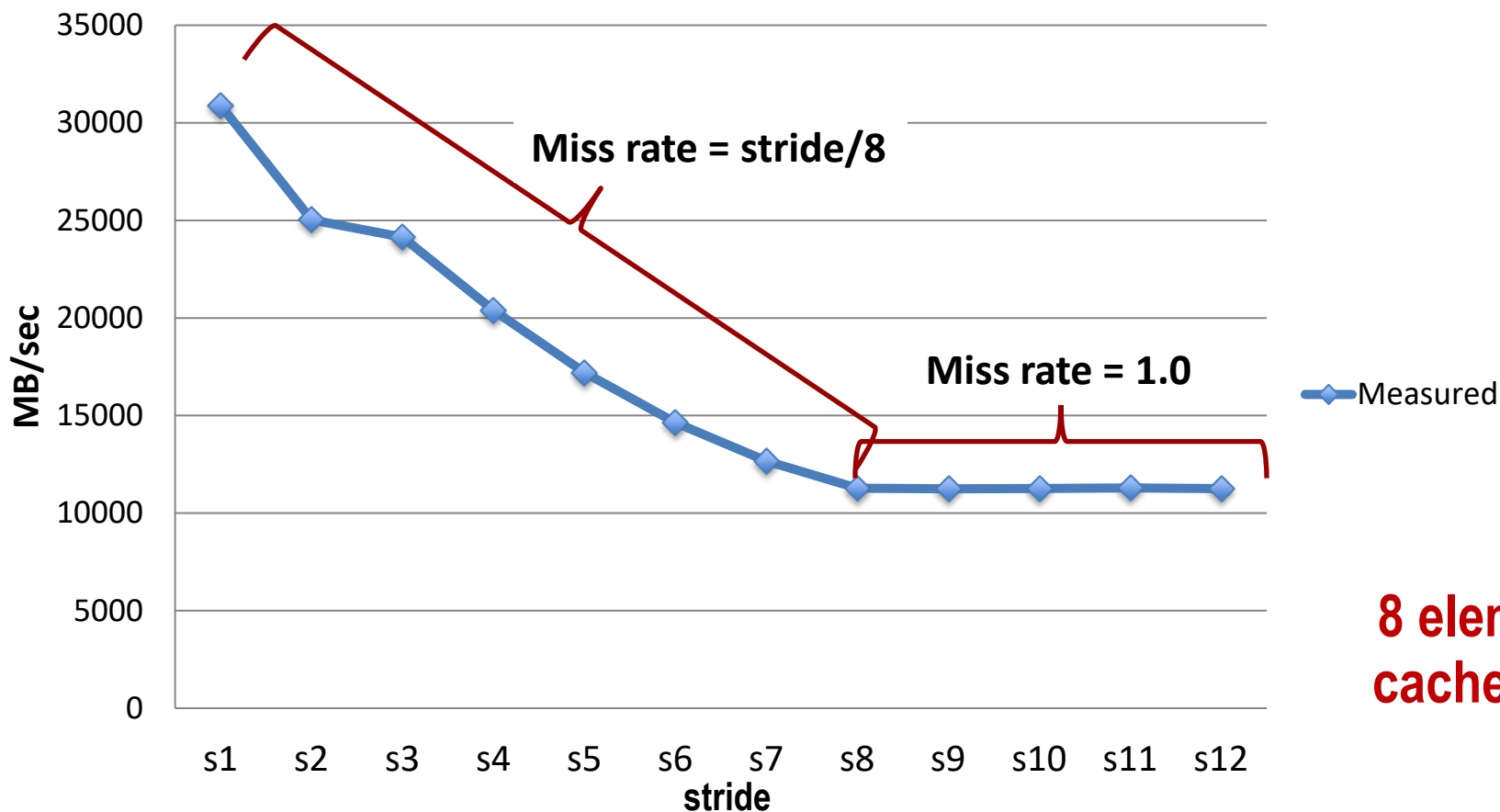**256 KB L2 cache**
8 MB L3 cache
64 B block size

Aggressive prefetching

Ridges of temporal locality

Slopes of spatial locality

L1

L2

L3

Mem

Read throughput (MB/s)

16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes)

s1 s3 s5 s7 s9 s11

32k 128k 512k 2m 8m 32m 128m

Size (bytes)

Closer Look at Stride Effects

Throughput for size = 128K

128K

Stride **Size**

Miss rate = stride/8

Miss rate = 1.0

Measured

8 elems per cache block

(y-axis: MB/sec; values 0, 5000, 10000, 15000, 20000, 25000, 30000, 35000)
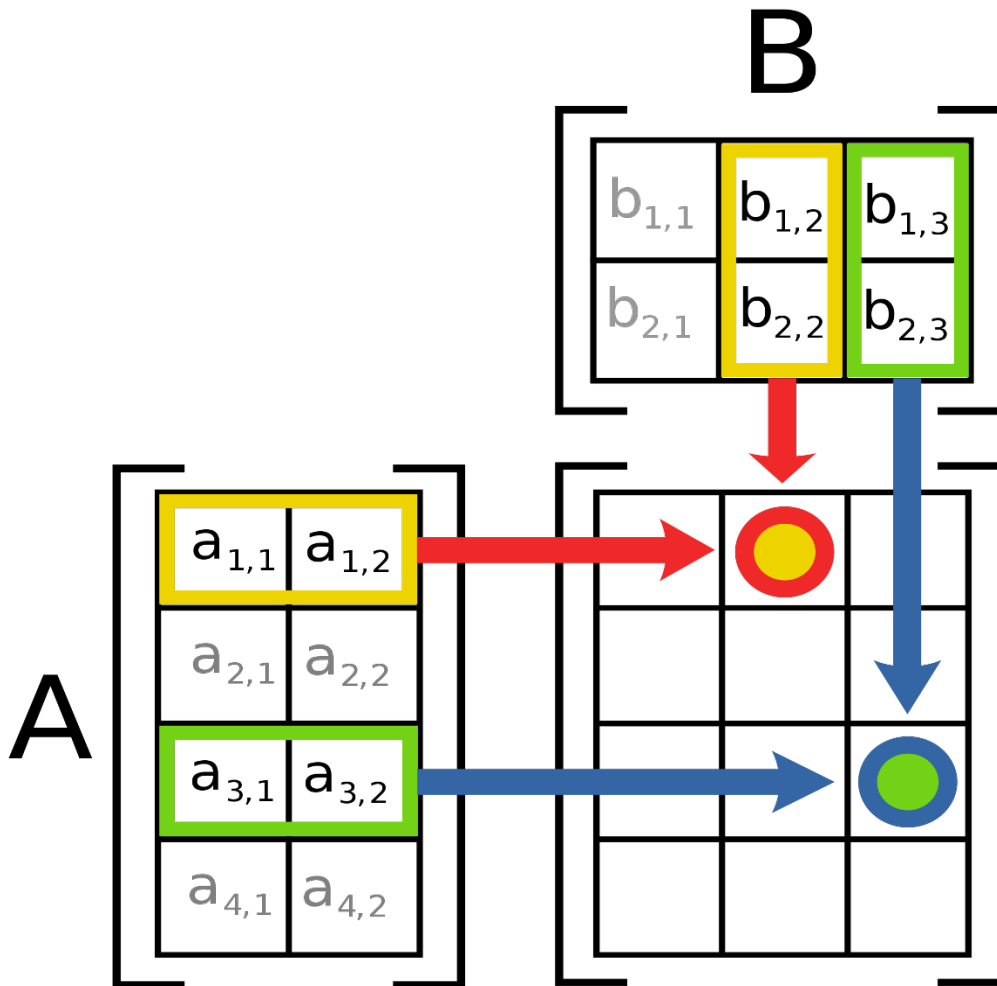
(x-axis: stride; s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12)

Today

- **Cache organization and operation**

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

Remember matrix multiplication

Out[i, j] =
   dot product(A[i, ..], B[..,j])
= sum (a[i, 0] * b[0, j],
          a[i, 1] * b[1, j],
               ...
          a[i, n] * b[n, j])

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

```
/* 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
```
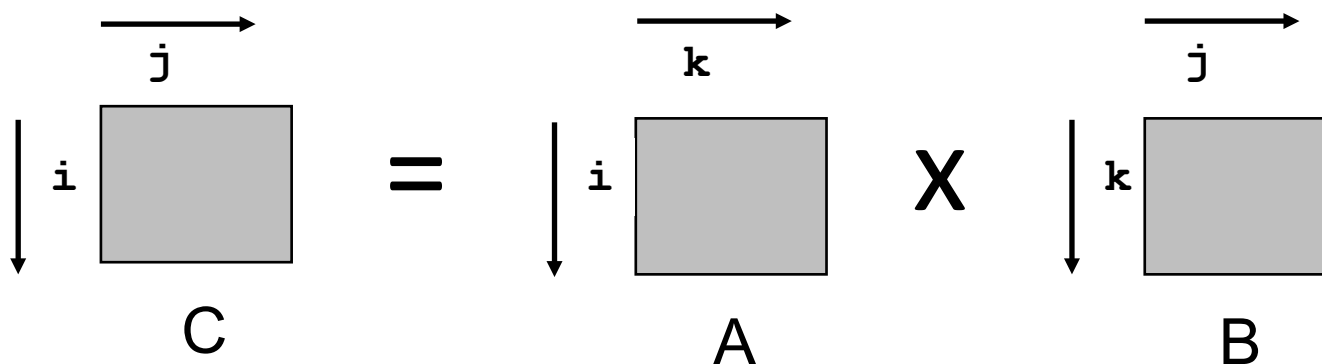
*Variable **sum** held in register*

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

Layout of C Arrays in Memory (review)

- **C arrays allocated in row-major order**
  - each row in contiguous memory locations
  - a[i][j] = a[i*N + j] where N is the number of columns

- **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%)
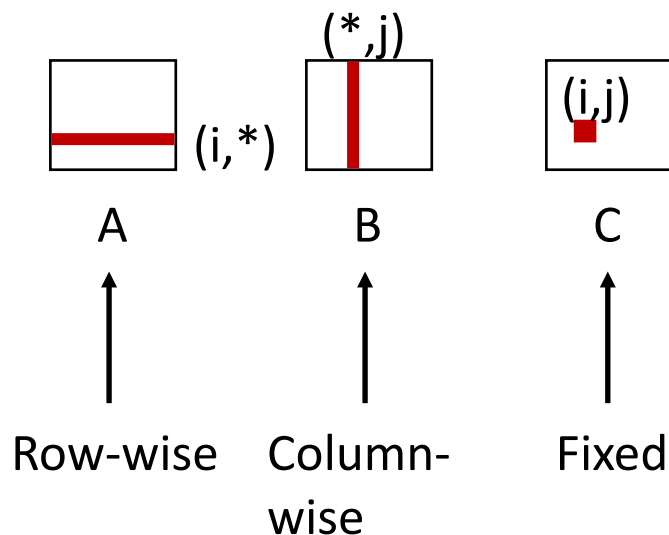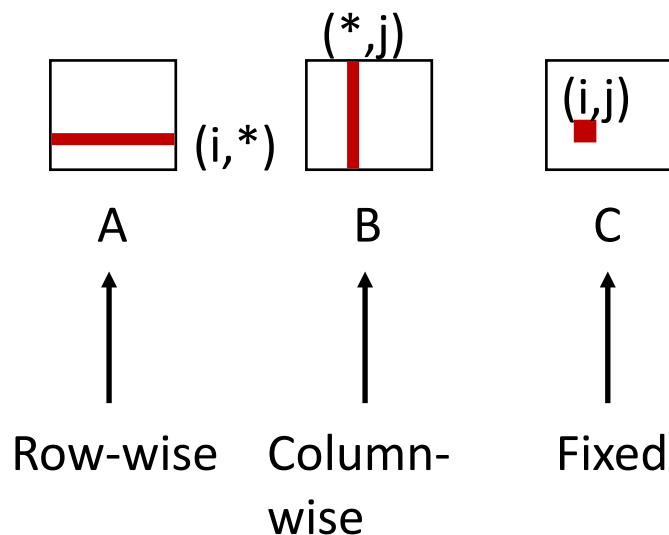
Matrix Multiplication (`ijk`)

```
/* 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:

$(*,j)$

$(i,*)$

A   B   C

(i,j)

Row-wise   Column-wise   Fixed

<u>Miss rate for inner loop iterations:</u>

<u>A</u>   <u>B</u>   <u>C</u>

Block size = 32B (four doubles)

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:

	(i,*)	(*,j)	(i,j)
A	B	C
Row-wise	Column-wise	Fixed

Miss rate for inner loop iterations:

A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

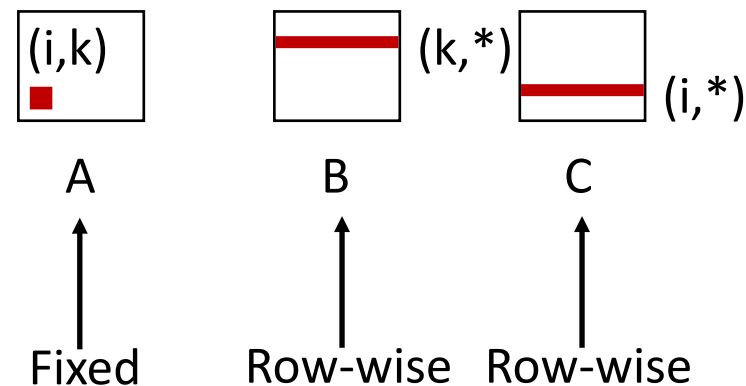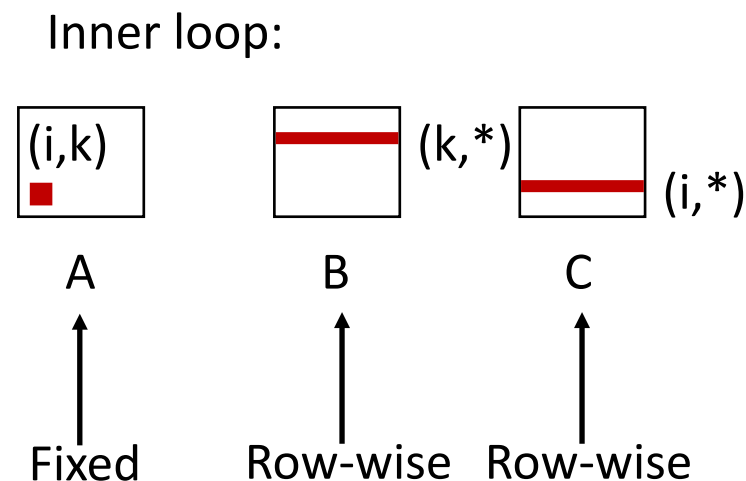# Matrix Multiplication (`kij`)

```
/* 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

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)

A

B

(k,*)

C

(i,*)

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)
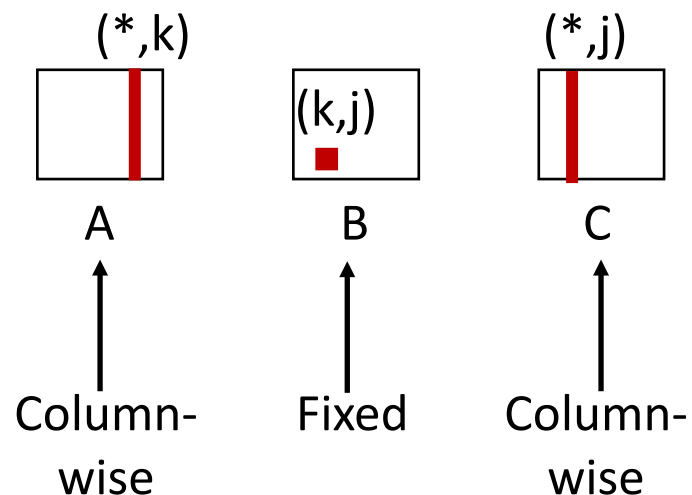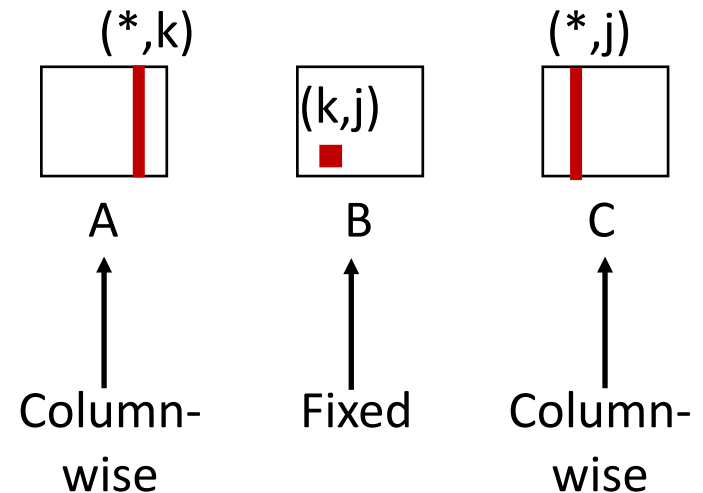
Matrix Multiplication (`jki`)

```
/* 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)   (*,j)

(k,j)

A   B   C

Column-   Fixed   Column-
wise   wise

Miss rate for inner loop iterations:

A   B   C

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)   (*,j)

A   B   C

                (k,j)

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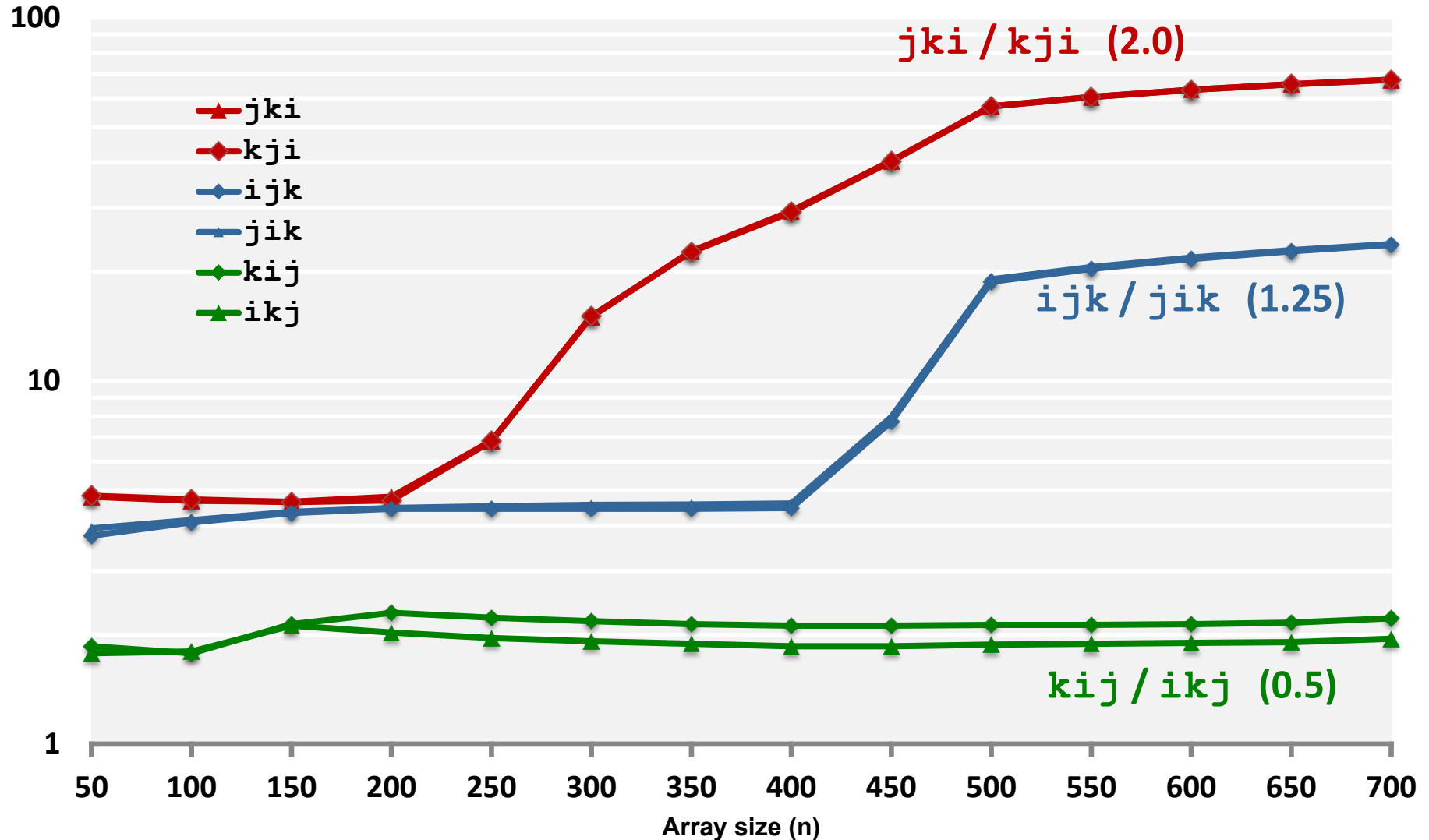
jki (& kji):
- 2 loads, 1 store
- avg misses/iter = **2.0**

Core i7 Matrix Multiply Performance

Cycles per inner loop iteration

- jki
- kji
- ijk
- jik
- kij
- ikj

jki / kji (2.0)

ijk / jik (1.25)

kij / ikj (0.5)

Array size (n)

Today

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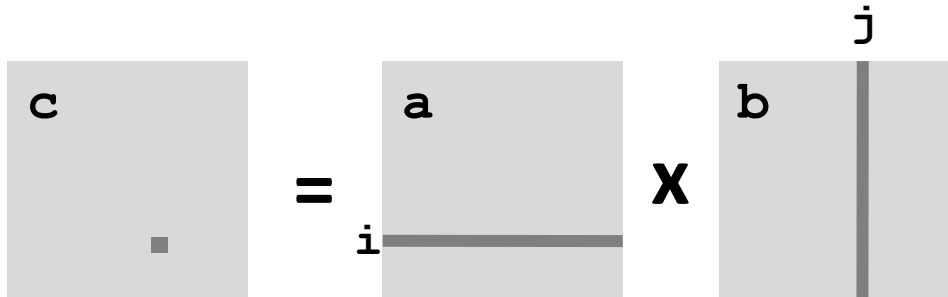 - Using blocking to improve temporal locality

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];
}
```
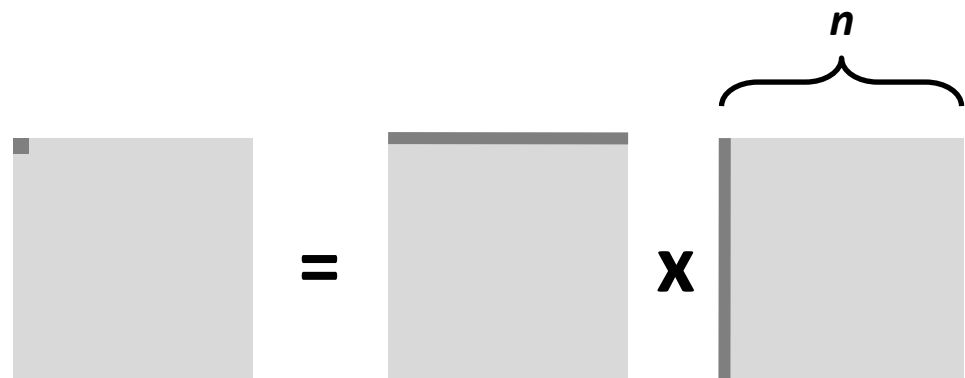
Cache Miss Analysis

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

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

  - Afterwards in cache: (schematic)
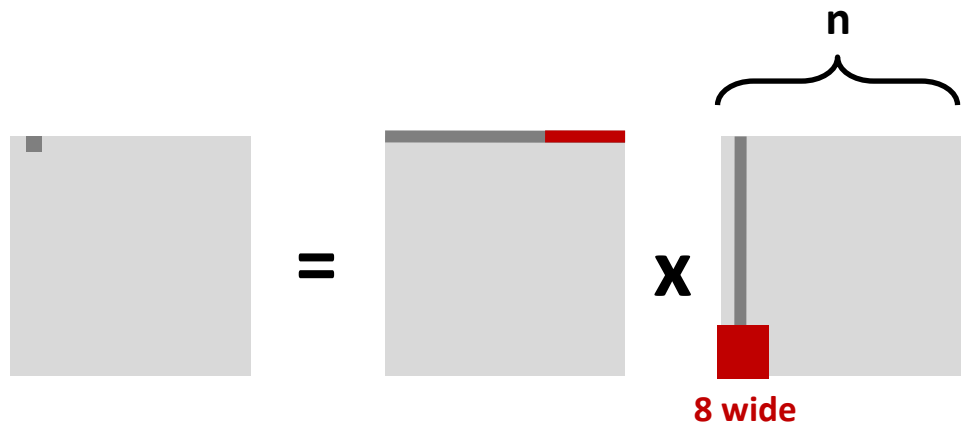
n

= X

= X

8 wide

Cache Miss Analysis

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

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

$$= \quad \times$$

n

8 wide

- **Total misses:**
  - $9n/8 \; n^2 = (9/8) \; n^3$
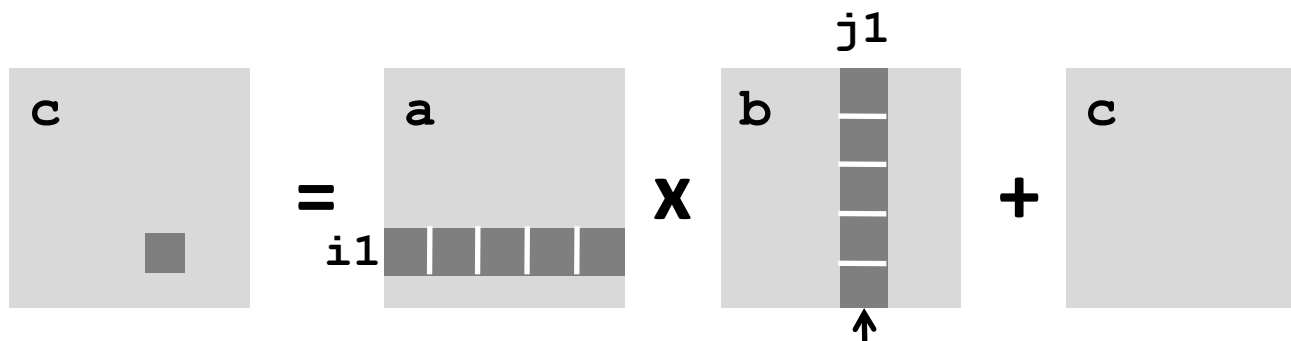
Blocked Matrix Multiplication

```
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+=B)
        for (j = 0; j < n; j+=B)
            for (k = 0; k < n; k+=B)
                /* B x B mini matrix multiplications */
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
                                                    matmult/bmm.c
```
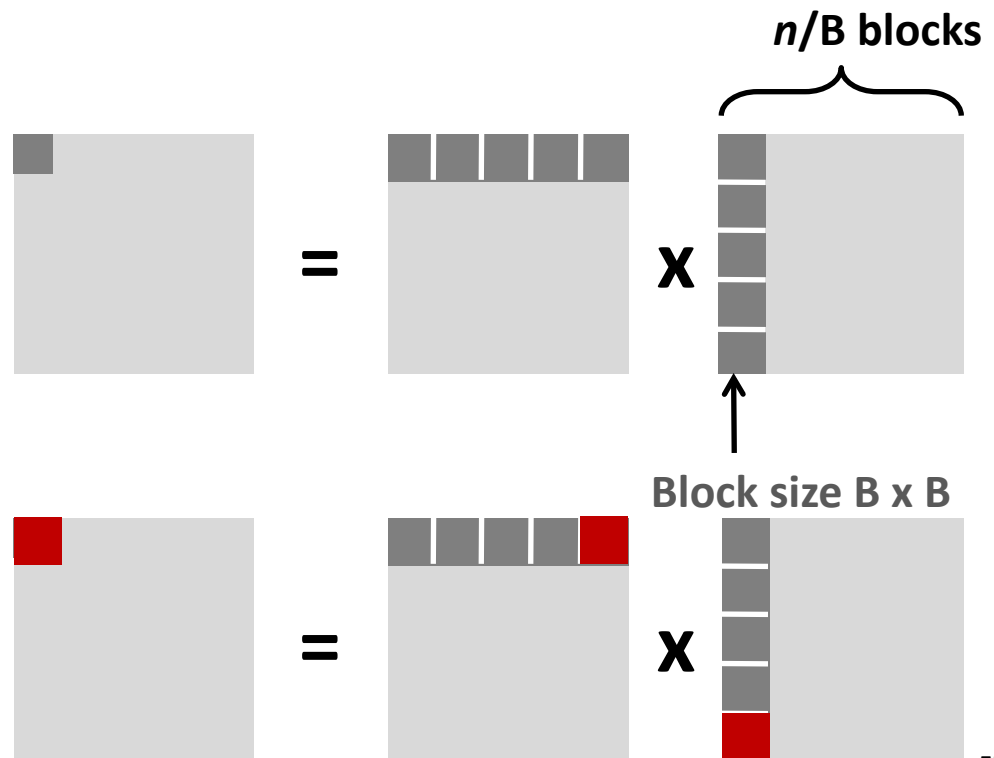

Block size B x B

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks ▪ fit into cache: $3B^2 < C$

- **First (block) iteration:**
  - B*B/8 misses for each block
  - $2n/B \times B^2/8 = nB/4$ (omitting matrix c)

n/B blocks

= X

Block size B x B

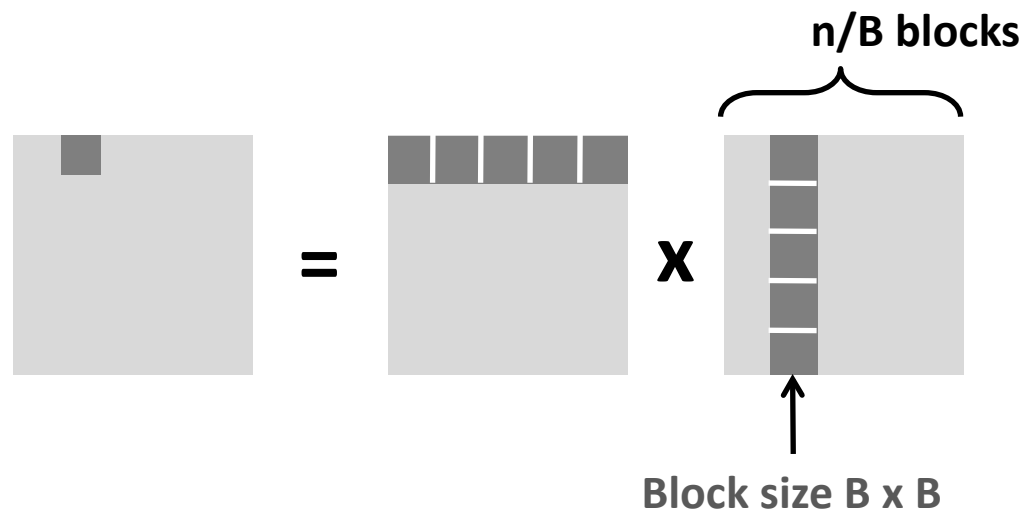 - Afterwards in cache (schematic)

= X

Cache Miss Analysis

- **Assume:**
  - Cache block = 8 doubles
  - Cache size C << *n* (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

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

n/B blocks

= **X**

Block size B x B

- **Total misses:**
  - $nB/4 * (n/B)^2 = n^3/(4B)$

Blocking Summary

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

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

- **Use largest block size B, such that B satisfies $3B^2 < C$**
  - Fit three blocks in cache! Two input, one output.
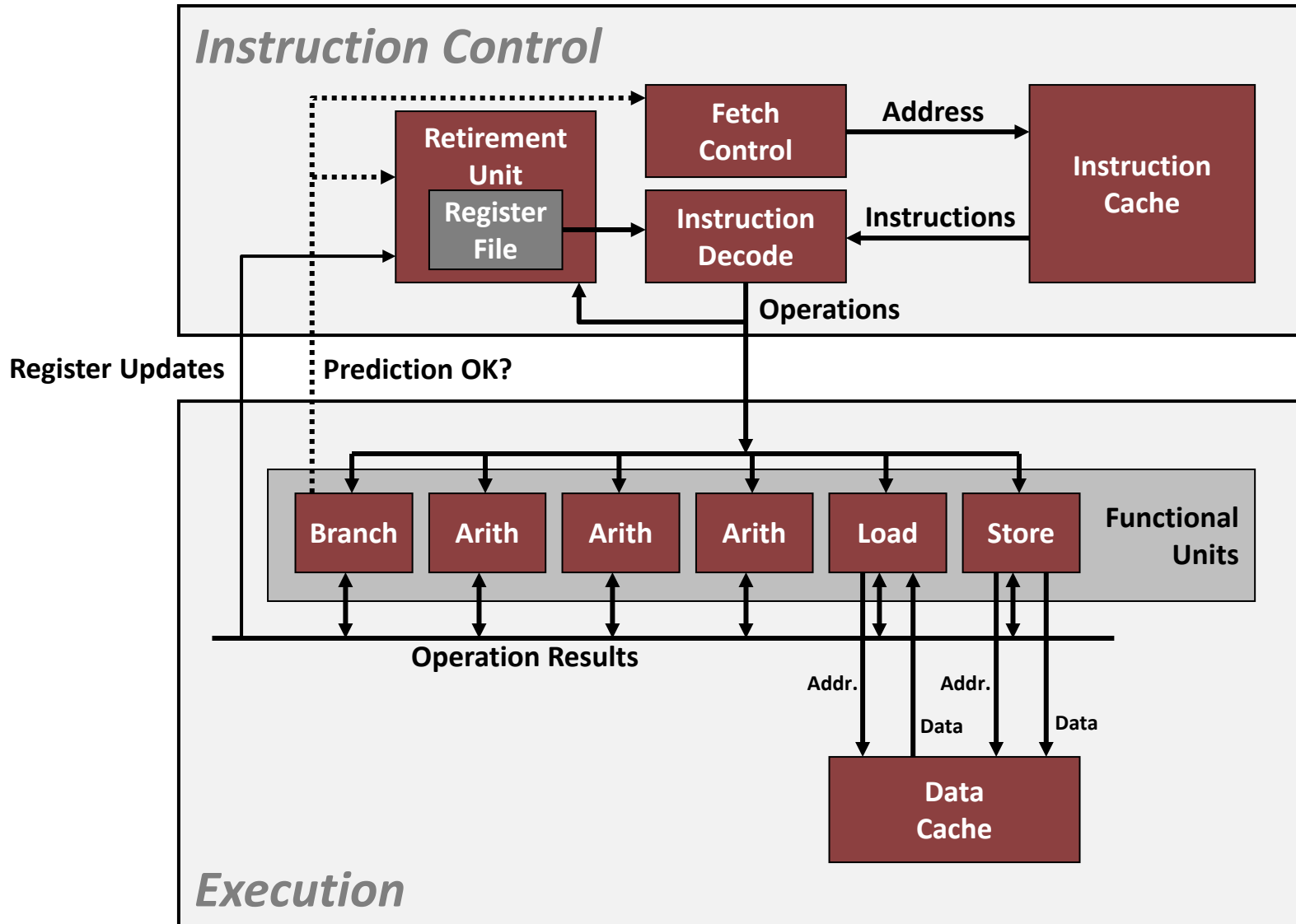
- **Reason for dramatic difference:**
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used O(n) times!
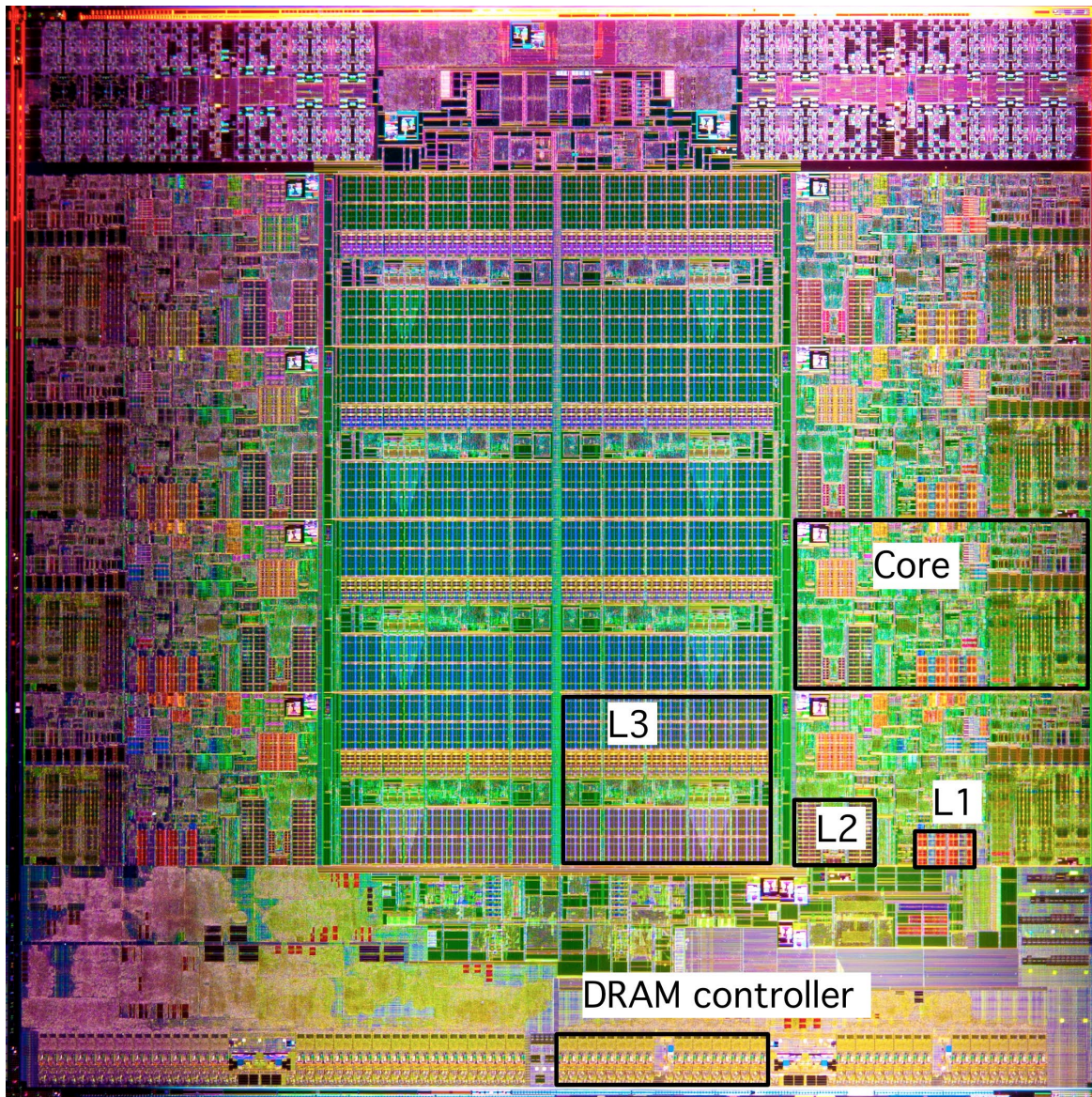 - But program has to be written properly

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

Recall: Modern CPU Design

Instruction Control

- Retirement Unit
  - Register File
- Fetch Control → **Address** → Instruction Cache
- Instruction Decode ← **Instructions** ← Instruction Cache
- Register File → Instruction Decode
- Instruction Decode → **Operations**

Register Updates | **Prediction OK?**

Execution

Functional Units

| Branch | Arith | Arith | Arith | Load | Store |

Operation Results

Addr. | Addr.
Data | Data

Data Cache

What it Really Looks Like (Cont.)

Intel Sandy Bridge Processor Die

L1: 32KB Instruction + 32KB Data
L2: 256KB
L3: 3–20MB

What it Really Looks Like (Cont.)

10nm ESF/Intel 7 Alder Lake die shot (~209mm²) from Intel via Andreas Schilling on Twitter:
https://twitter.com/aschilling/status/1453391035577495553

Die shot interpretation by Locuza, October 2021
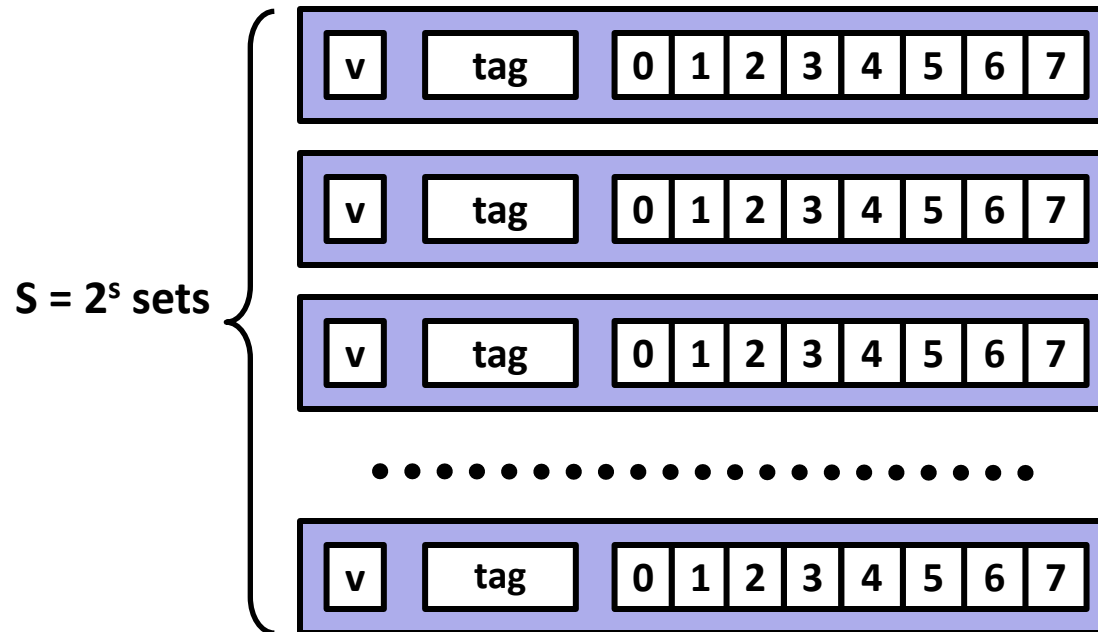
Intel Alder Lake (2021)
8 P-cores + 8 E-cores

L1 caches per P-core: 32KB Instruction & 48KB Data
L1 caches per E-core: 64KB Instruction & 32KB Data
L2 caches: 1.25MB per P-core, 2MB per four E-cores
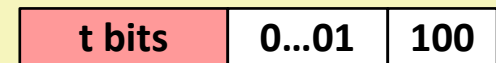**L3 cache: 30MB shared among all cores**

Why Index Using Middle Bits?

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

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

S = 2^s sets

| 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:
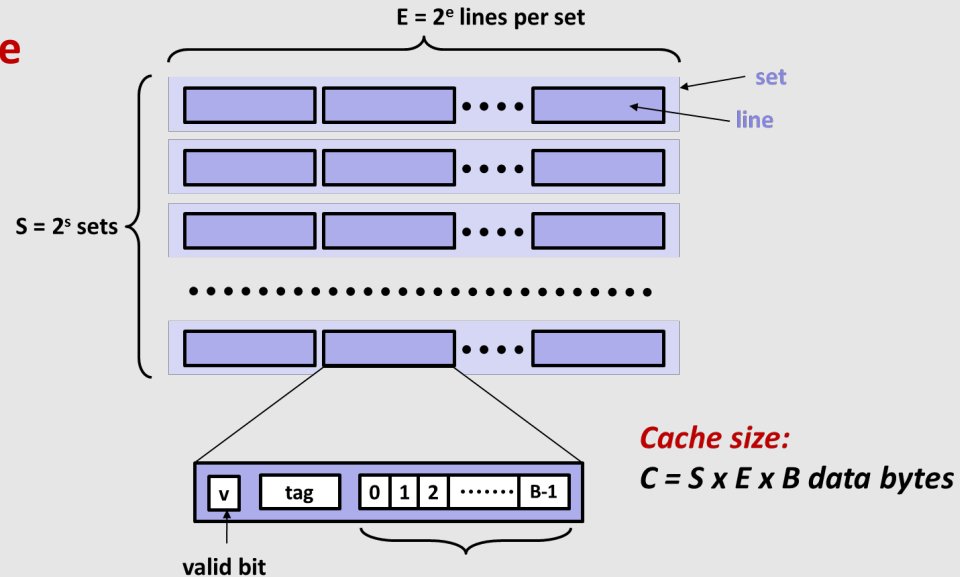
| 1…11 | t bits | 100 |
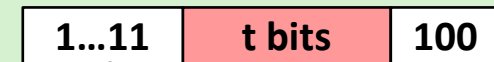
find set

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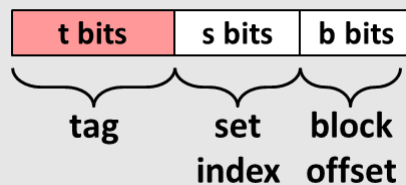
Block offset: . bits
Set index: . bits
Tag: . bits

Stack Address:
0x00007f7262a1e010

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

Example: Core i7 L1 Data Cache
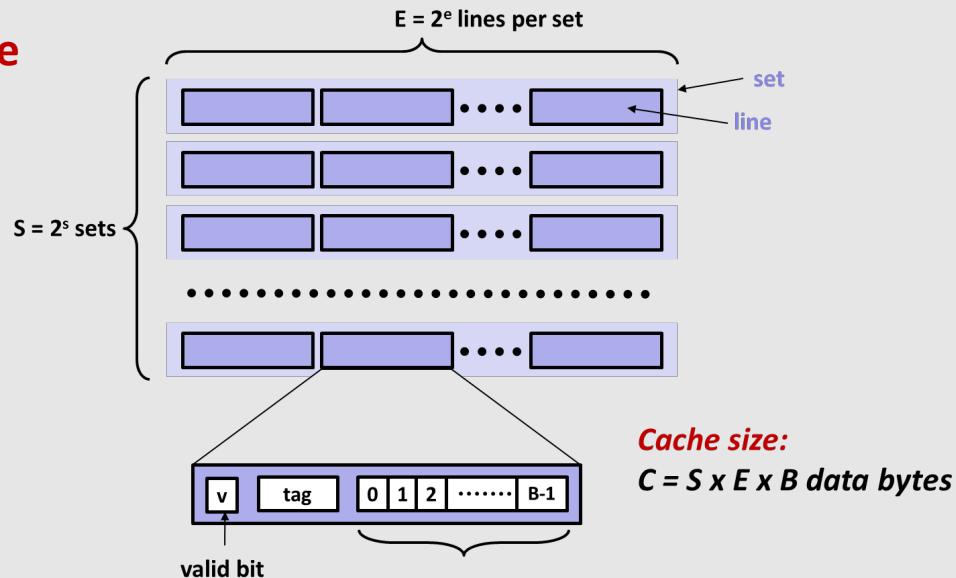
32 kB 8-way set associative
64 bytes/block
47 bit address range

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

$E = 2^e$ lines per set

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: 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

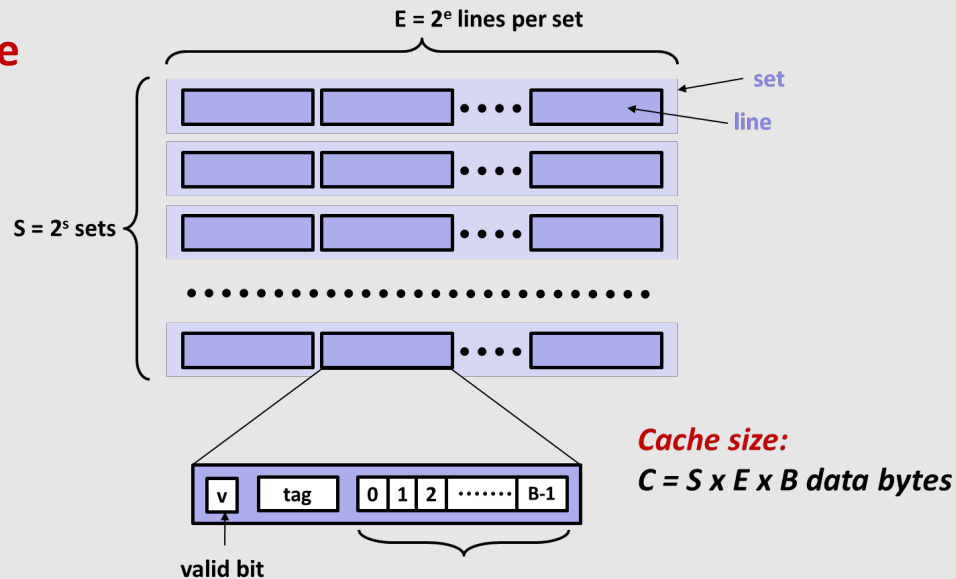**32 kB 8-way set associative**
64 bytes/block
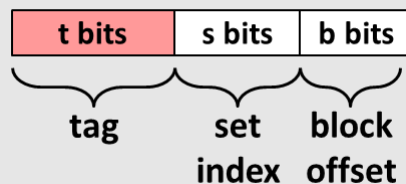**47 bit address range**

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

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

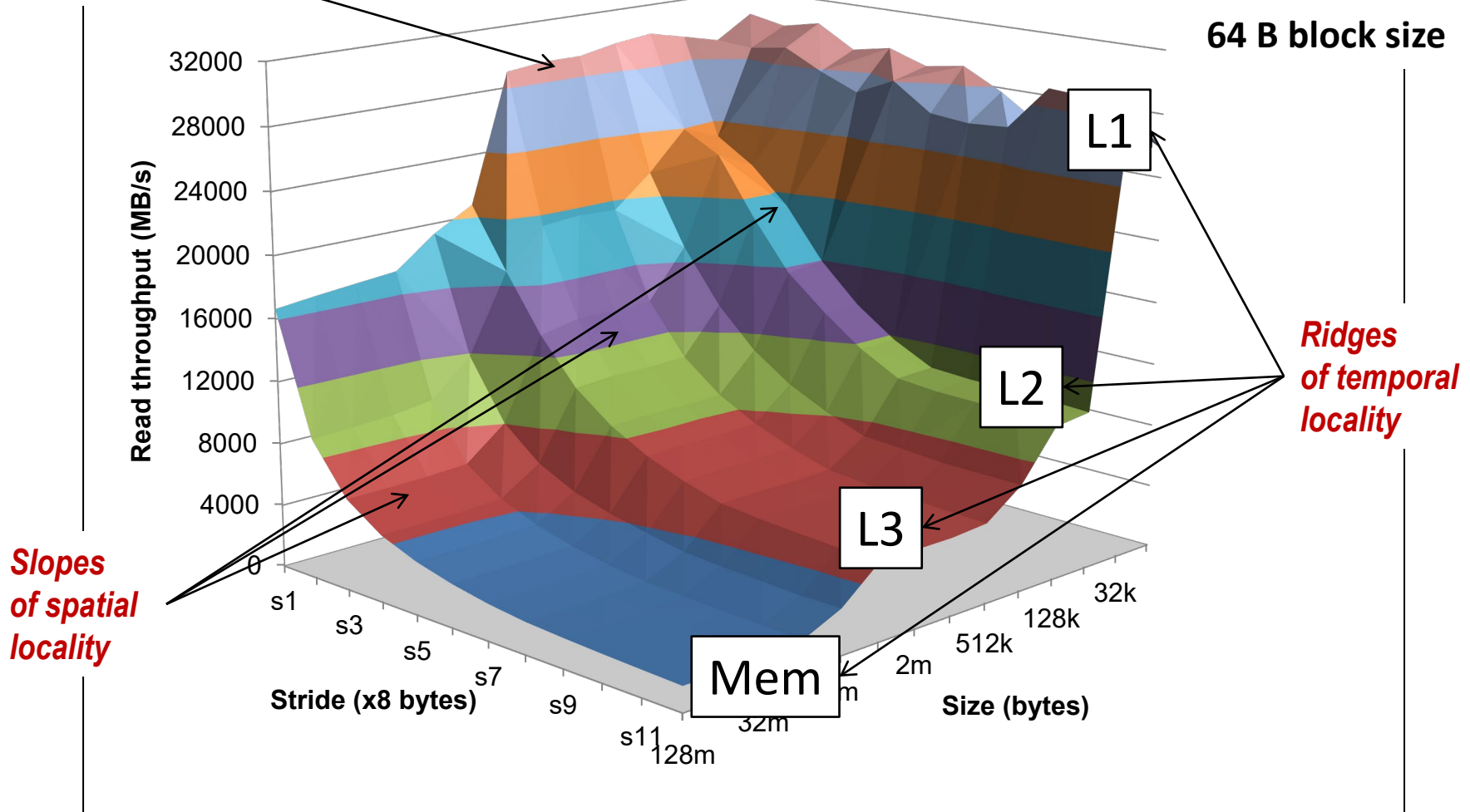**Block offset: 0x10**
**Set index: 0x0**
Tag: 0x7f7262a1e

The Memory Mountain

Core i5 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Aggressive prefetching

Slopes of spatial locality

Ridges of temporal locality

L1

L2

L3

Mem

Read throughput (MB/s)

Stride (x8 bytes)

Size (bytes)

Cache Capacity Effects from Memory Mountain

Core i7 Haswell
3.1 GHz
32 KB L1 d-cache
256 KB L2 cache
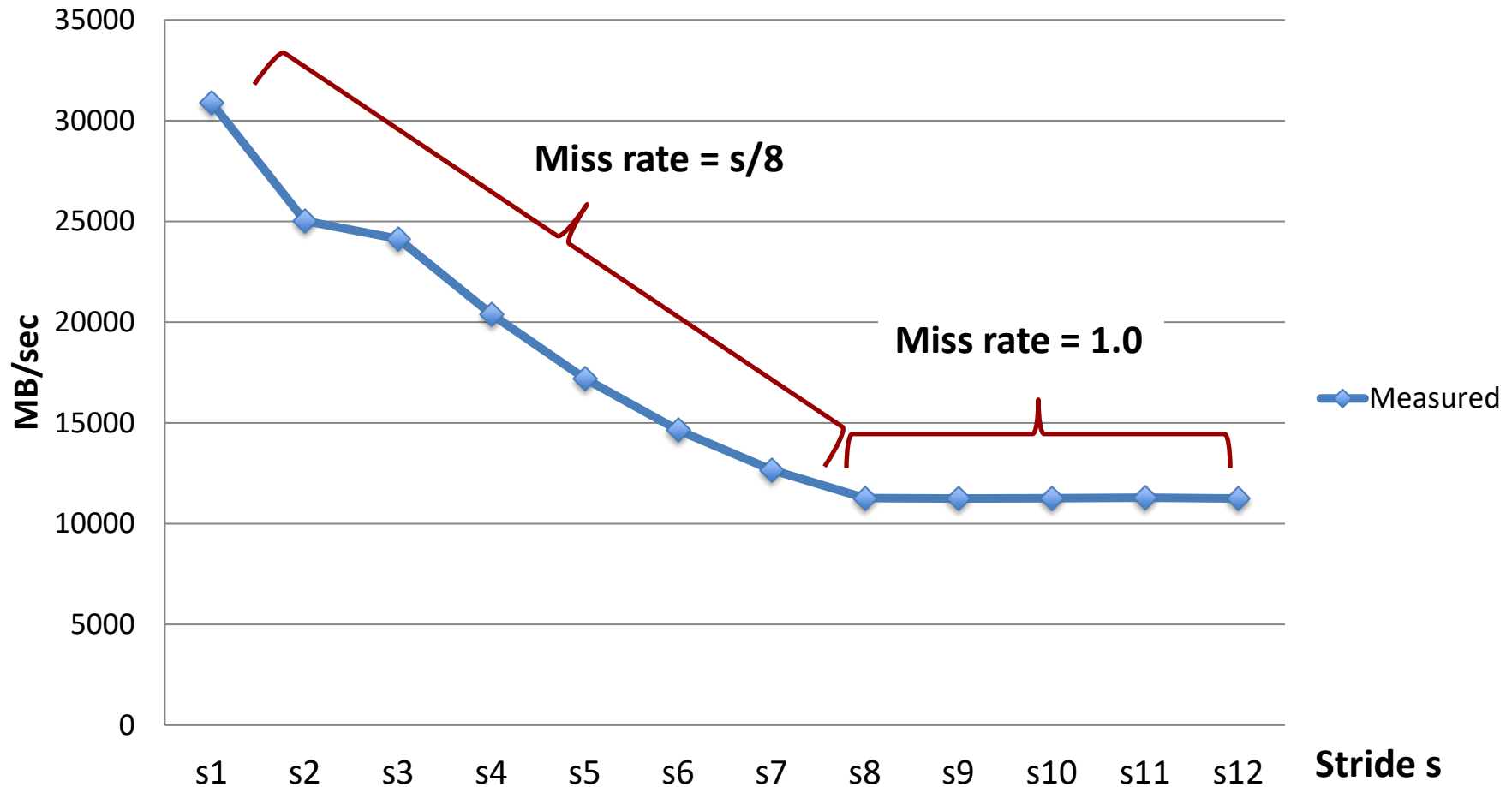8 MB L3 cache
64 B block size
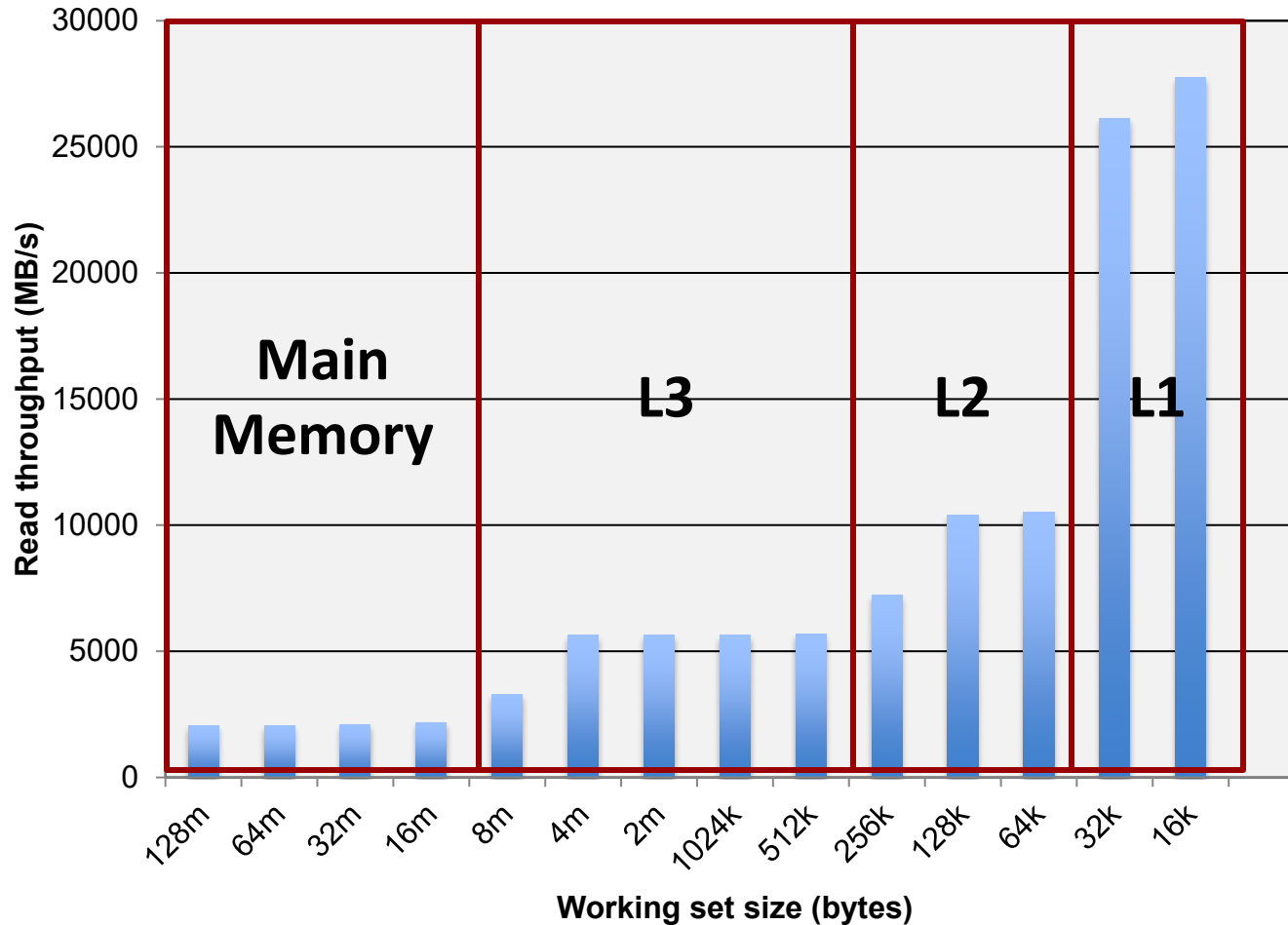
Slice through memory mountain with stride=8

Cache Block Size Effects from Memory Mountain

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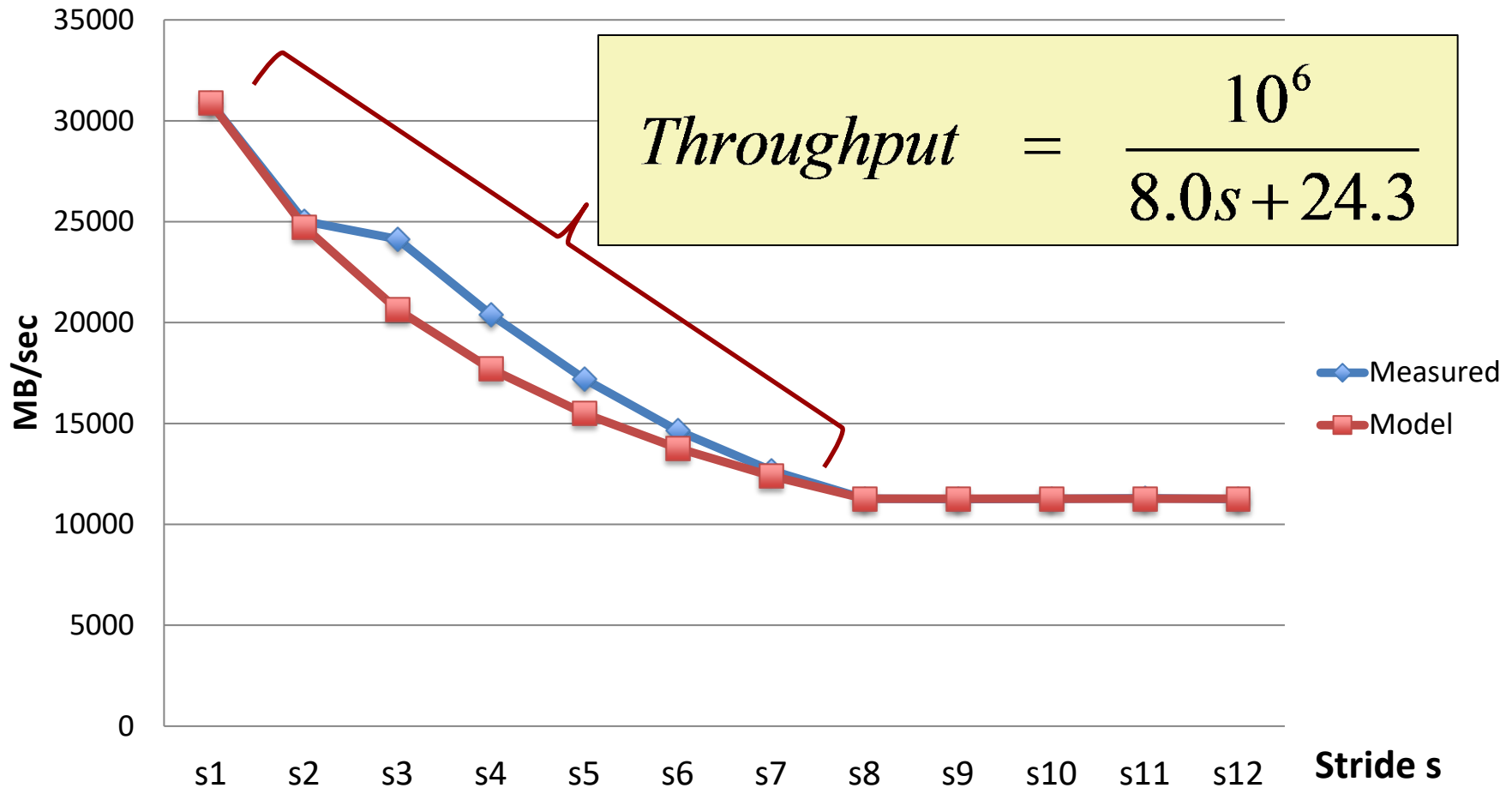32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache
64 B block size

Throughput for size = 128K
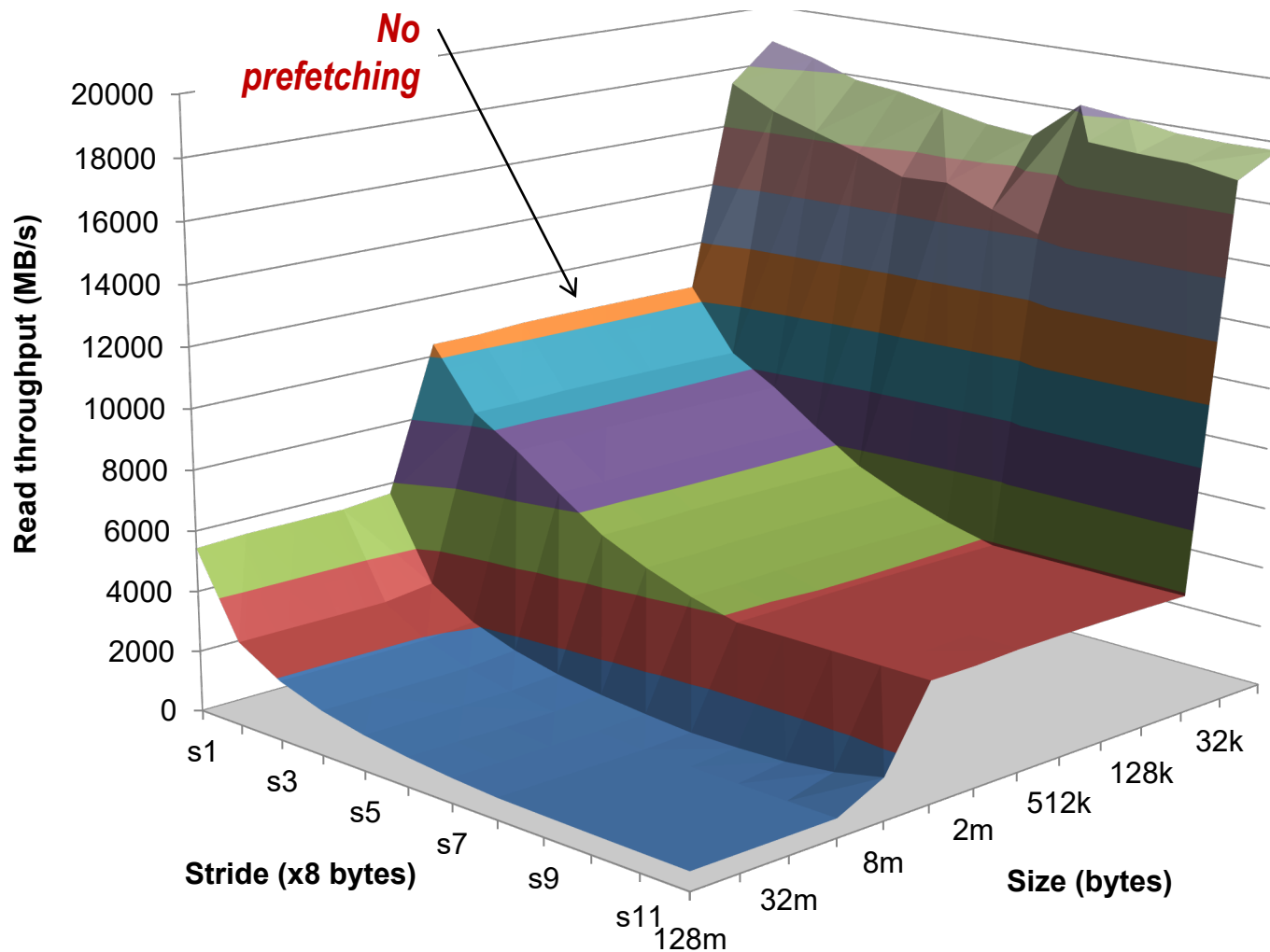
Modeling Block Size Effects from Memory Mountain

**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

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

Measured
Model

Stride s

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) — axis: 0, 2000, 4000, 6000, 8000, 10000, 12000, 14000, 16000, 18000, 20000

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

Size (bytes) — 32k, 128k, 512k, 2m, 8m, 32m, 128m
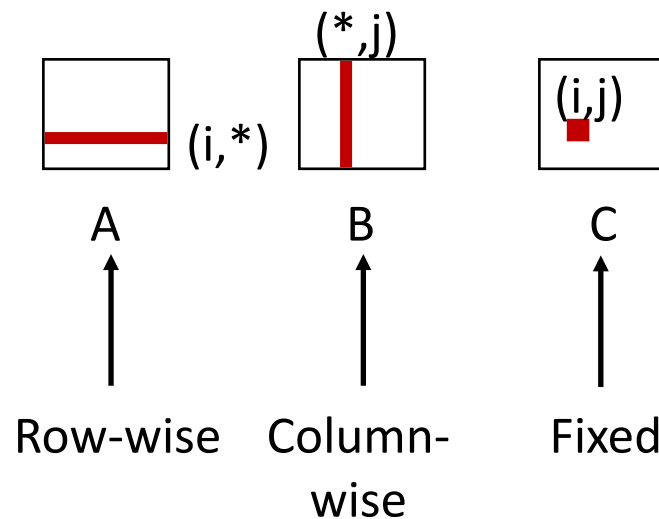
Matrix Multiplication (`jik`)

```
/* 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:

A — Row-wise
B — Column-wise
C — Fixed

Misses per inner loop iteration:

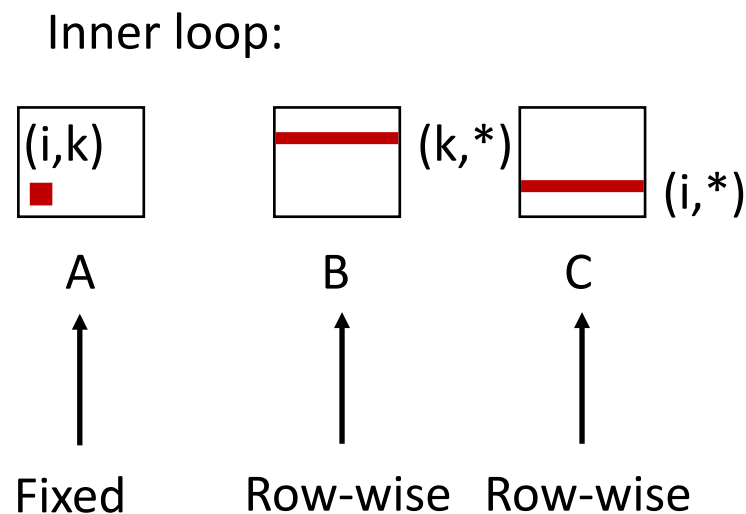| A	B	C
0.25	1.0	0.0

Block size = 32B (four doubles)

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:

(i,k)

A   B   C

Fixed   Row-wise   Row-wise

(k,*)   (i,*)

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

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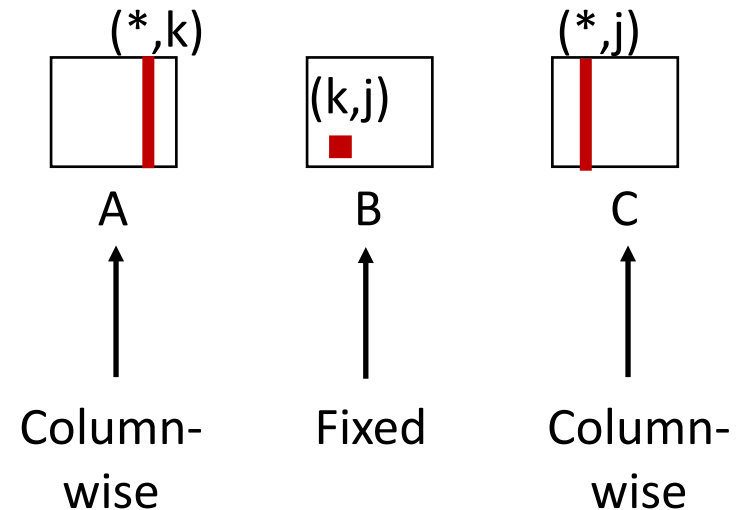
Matrix Multiplication (`kji`)

```
/* 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)