COMP201
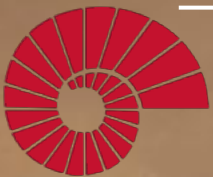
Computer Systems & Programming

Lecture #23 – More Cache Memories

Aykut Erdem // Koç University // Fall 2021

KOÇ UNIVERSITY

Recap

- Cache basics

- Principle of locality

- Cache memory organization and operation

Recap: Cache Parameters

- **Block size (B):** basic unit of transfer between memory and the cache, given in bytes (e.g. 64 B).

- **Cache size (C):** Total amount of data that can be stored in the cache, given in bytes (e.g. 32 KiB).
    - Must be multiple of block size
    - Number of blocks in cache is calculated by C/K

- **Associativity (E):** Number of ways blocks can be stored in a cache set, or how many blocks in each set

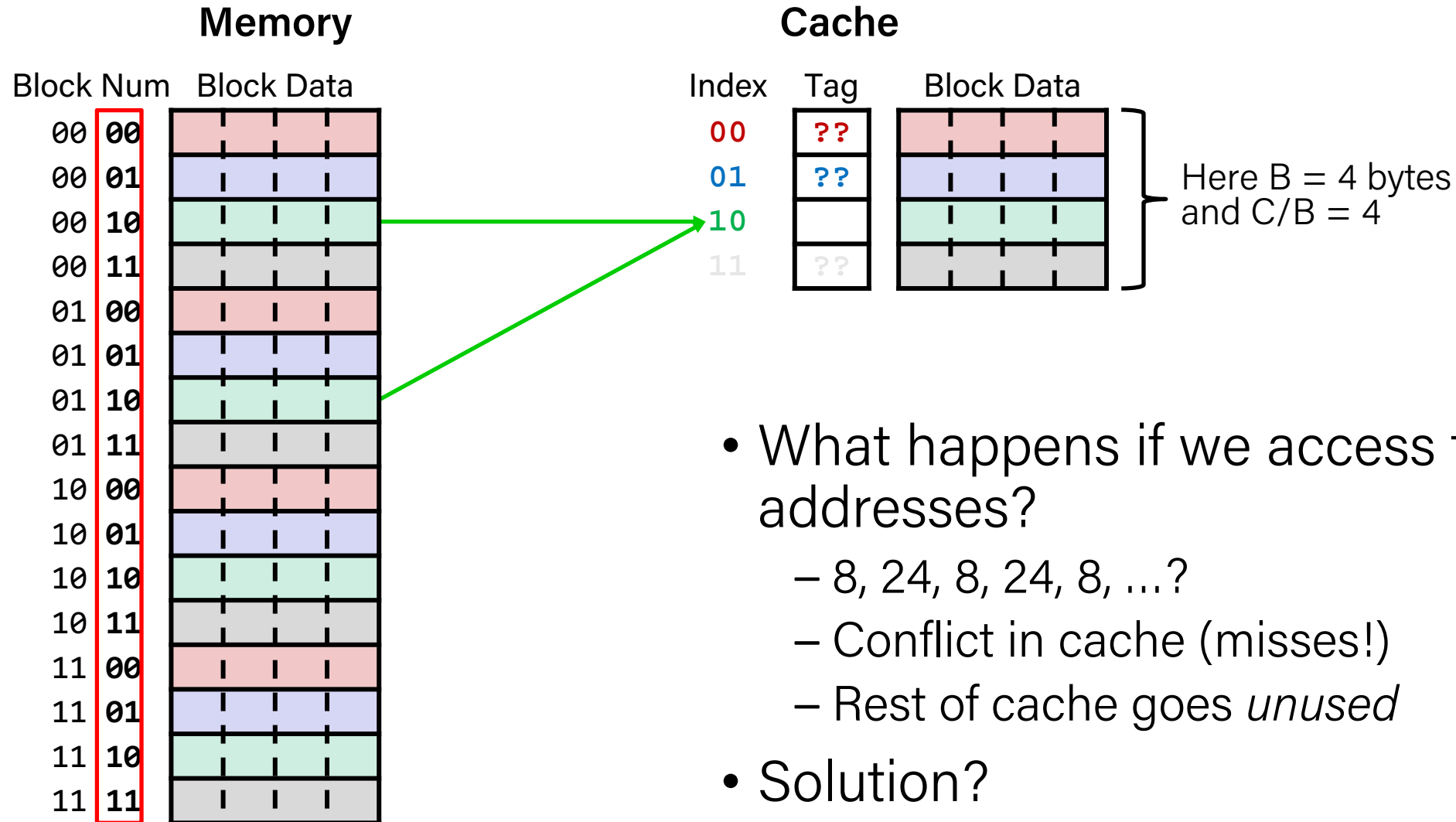- **Number of sets (S):** Number of unique sets that blocks can be placed into in a cache (calculated as C/K/E).

Recap: TIO address breakdown

- TIO address breakdown:

m-bit address:

Tag (**t**)	Index (**s**)	Offset (**b**)
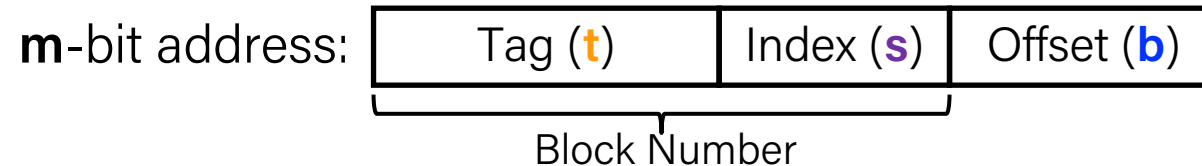
Block Number

- **Index (s)** field tells you where to look in cache
  - Number of bits is determined by number of sets ($\log_2(C/B/E)$)
  - Need enough bits to reference every set in the cache
- **Tag (t)** field lets you check that data is the block you want
  - Rest of the bits not used for index and offset (**m** – **s** – **b**)
- **Offset (b)** field selects specified start byte within block
  - Number of bits is determined by block size ($\log_2(B)$)
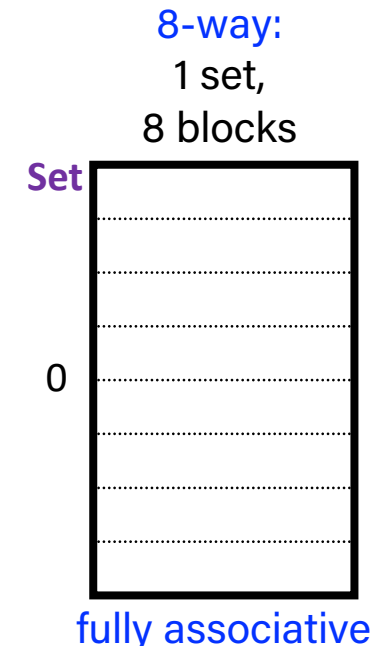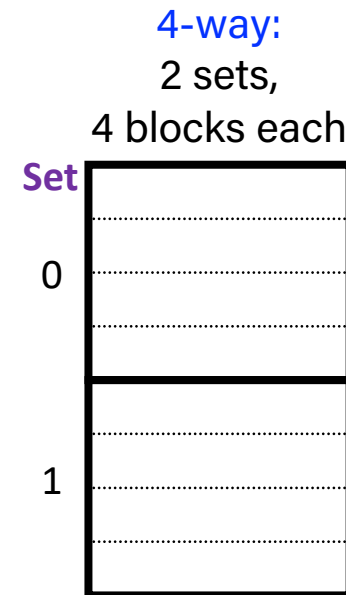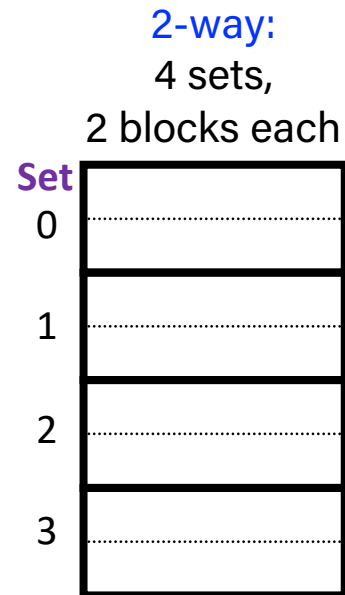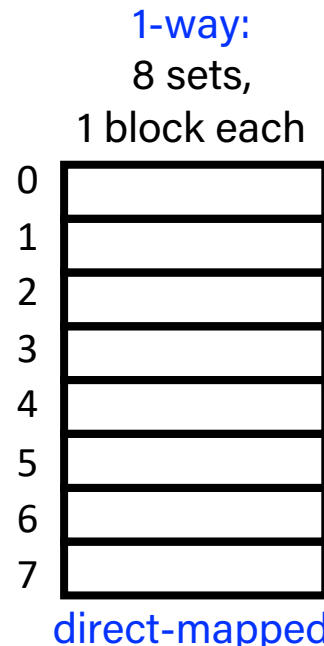 - Need enough bits to reference every byte in a block

Recap: Direct-Mapped Cache Problem

Memory

Block Num   Block Data

00 **00**	
00 **01**	
00 **10**	
00 **11**	
01 **00**	
01 **01**	
01 **10**	
01 **11**	
10 **00**	
10 **01**	
10 **10**	
10 **11**	
11 **00**	
11 **01**	
11 **10**	
11 **11**	

Cache

Index   Tag   Block Data

Index	Tag	Block Data
00	??	
01	??	
10		
11	??	

Here B = 4 bytes
and C/B = 4

- What happens if we access the following addresses?
    - 8, 24, 8, 24, 8, ...?
    - Conflict in cache (misses!)
    - Rest of cache goes *unused*
- Solution?

5

Recap: Associativity

- What if we could store data in any place in the cache?
  – More complicated hardware = more power consumed, slower

- So we *combine* the two ideas:
  – Each address maps to exactly one set
  – Each set can store block in more than one way

1-way:
8 sets,
1 block each

2-way:
4 sets,
2 blocks each

4-way:
2 sets,
4 blocks each

8-way:
1 set,
8 blocks

direct-mapped

fully associative

Recap: Cache Lookup Process

- CPU requests data at a given address

- Cache breaks down address into different bit fields
    - Determines offset, index, and tag bits

- Cache checks to see if block containing address is already in the cache
    - Uses index bits to find which set to look in
    - Uses tag bits to make sure the block in the set matches

- If block is in the cache, it's a **cache hit**
    - Data is returned to CPU starting at byte offset

- If block is not in the cache, it's a **cache miss**
    - Block is loaded from memory into the cache, evicting other blocks from the cache if necessary
    - Data is returned to CPU starting at byte offset

Plan for Today

- Cache memory organization and operation

- Writing cache-friendly code

Disclaimer: Slides for this lecture were borrowed from

—Randal E. Bryant and David R. O'Hallaroni's CMU 15-213 class
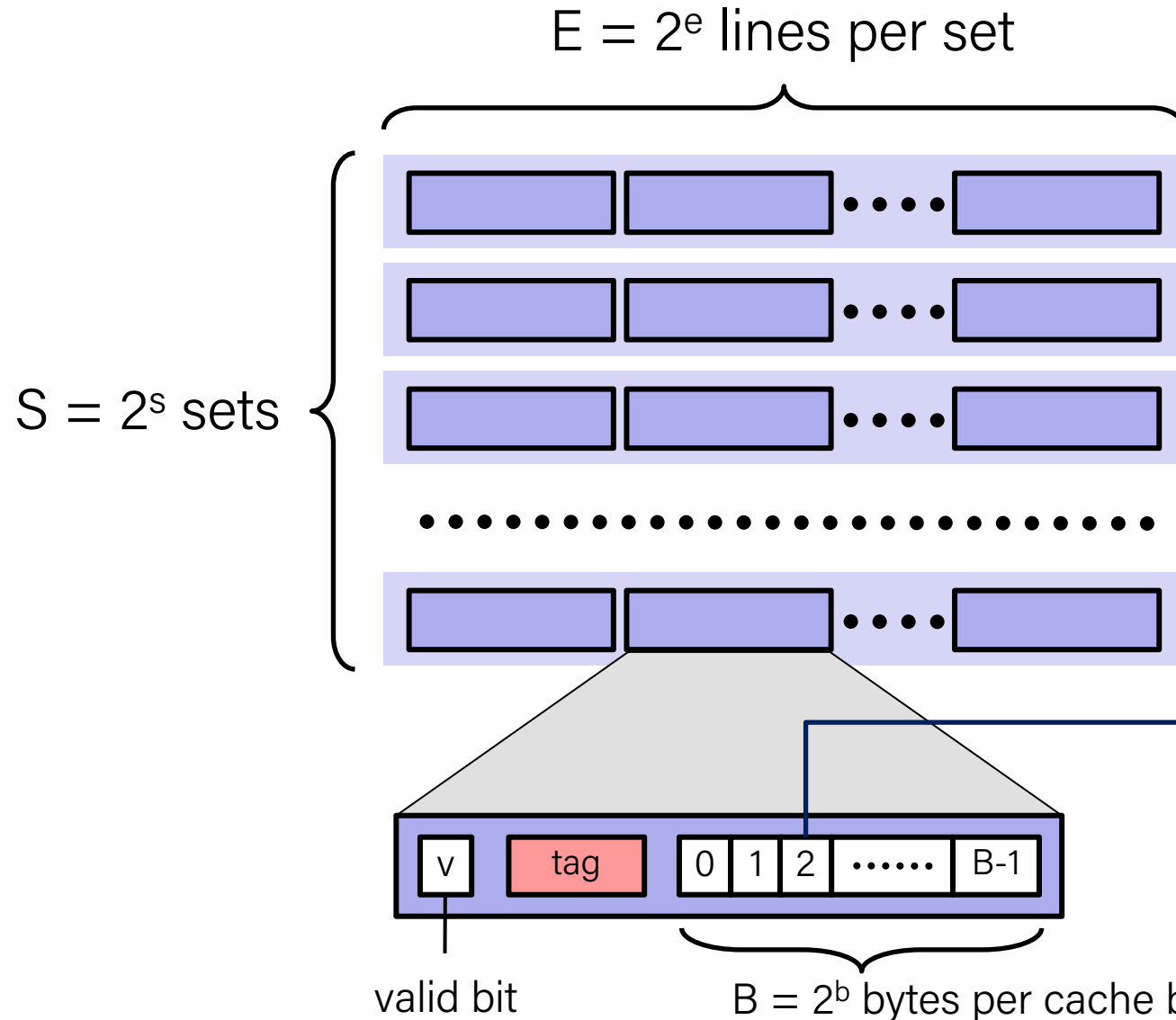
—Porter Jones' UW CSE 351 class

Lecture Plan

- Cache memory organization and operation
- Performance impact of caches
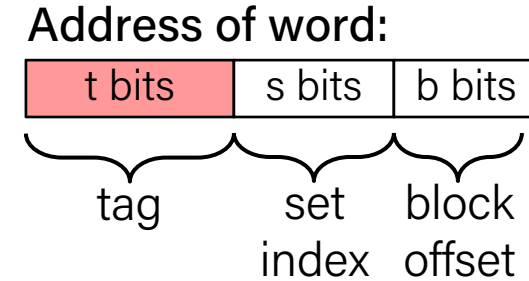
General Cache Organization (S, E, B)

$E = 2^e$ lines per set

$S = 2^s$ sets

set

line

valid bit

$B = 2^b$ bytes per cache block (the data)
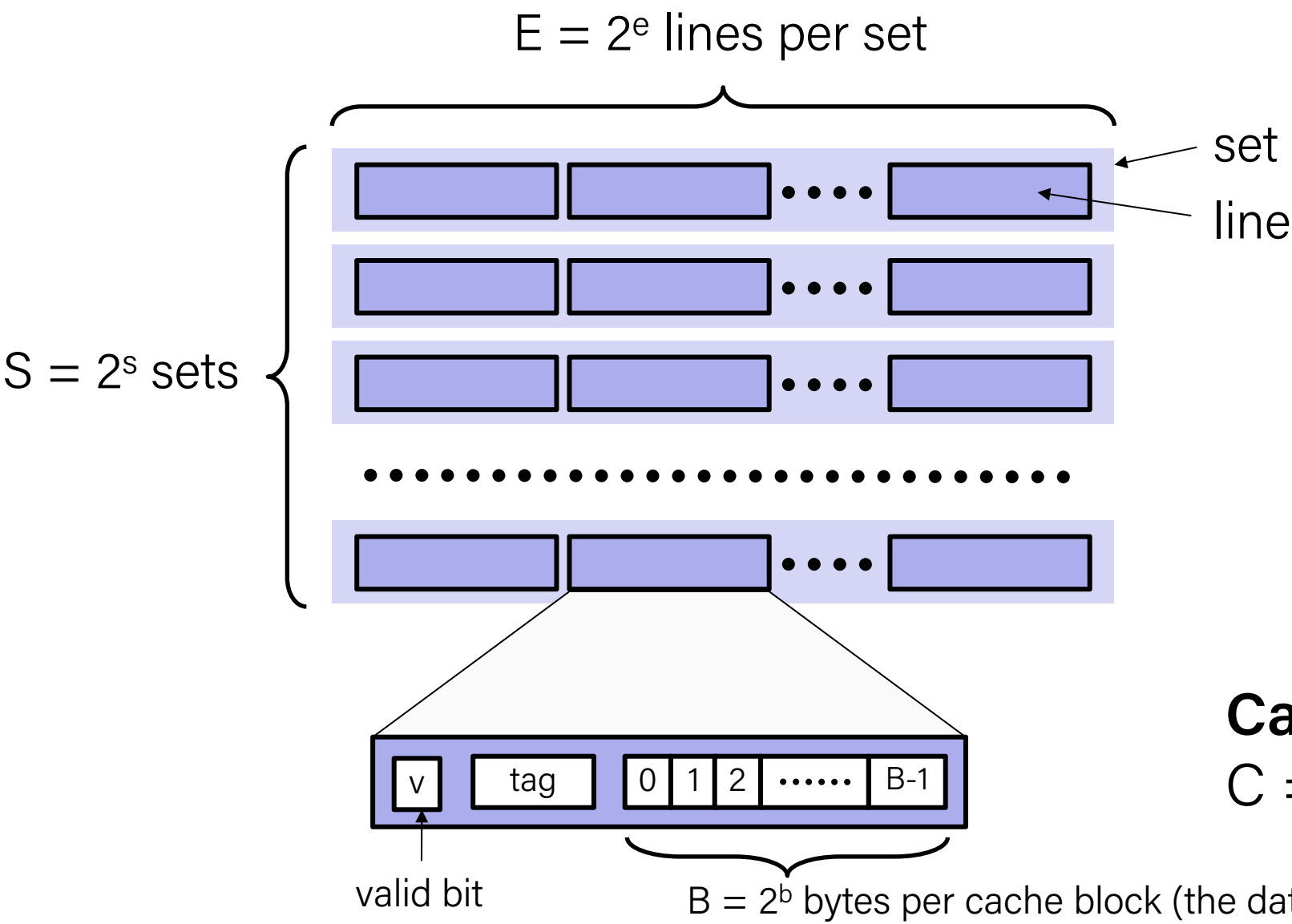
Cache size:
$C = S \times E \times B$ data bytes

Cache Read

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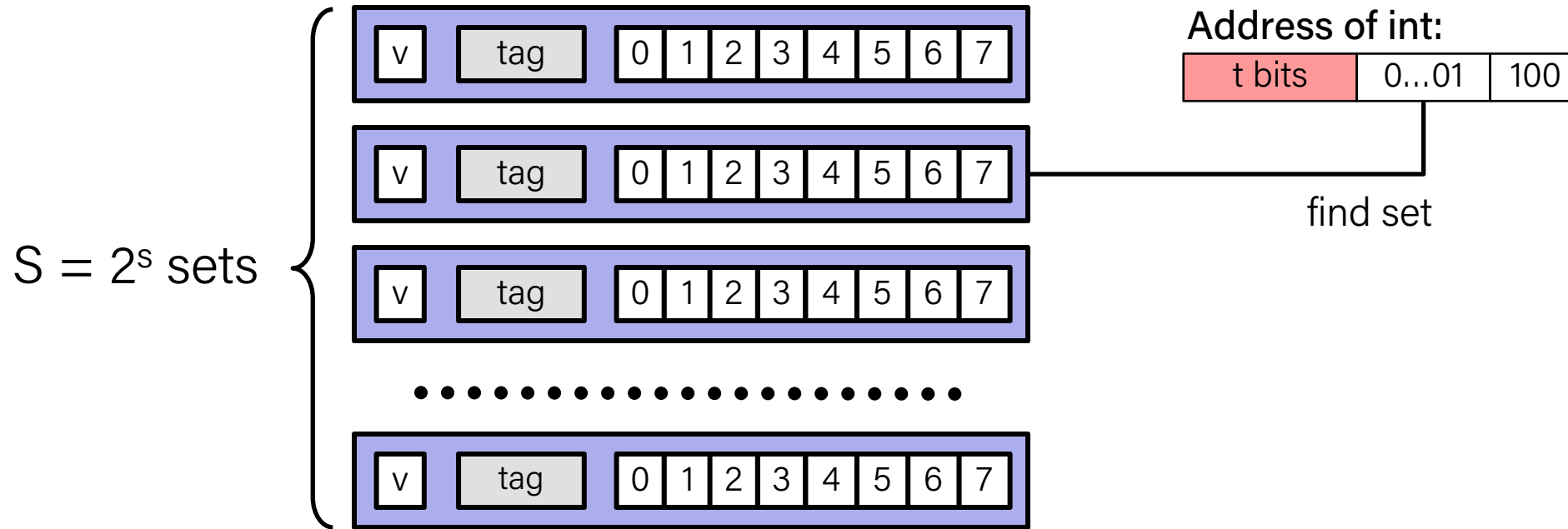- Yes + line valid: hit
- Locate data starting at offset

$E = 2^e$ 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 8 bytes

S = 2^s sets

Address of int:

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

find set

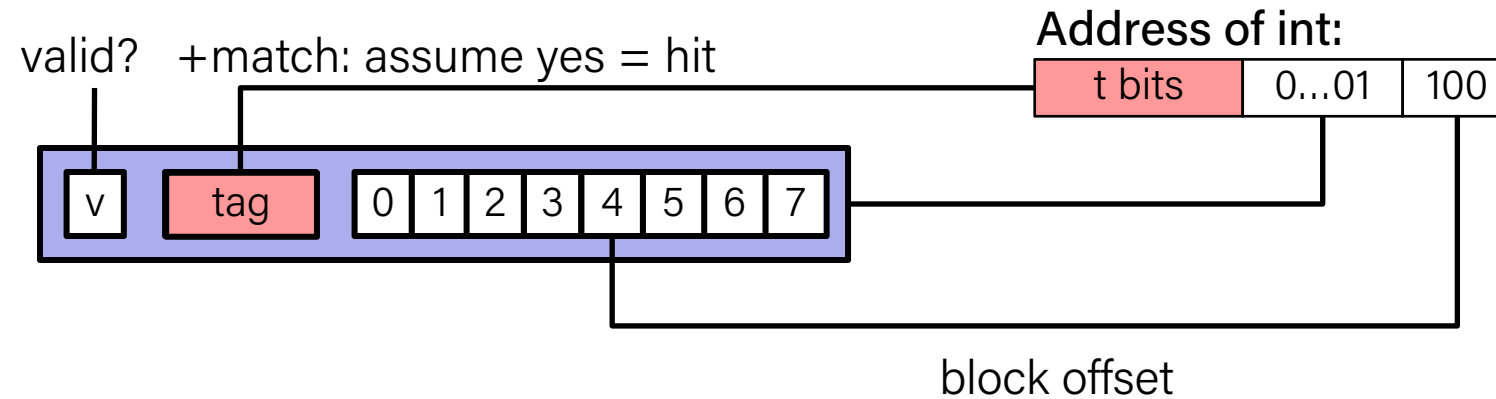# Example: Direct Mapped Cache (E = 1)

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

Example: Direct Mapped Cache (E = 1)
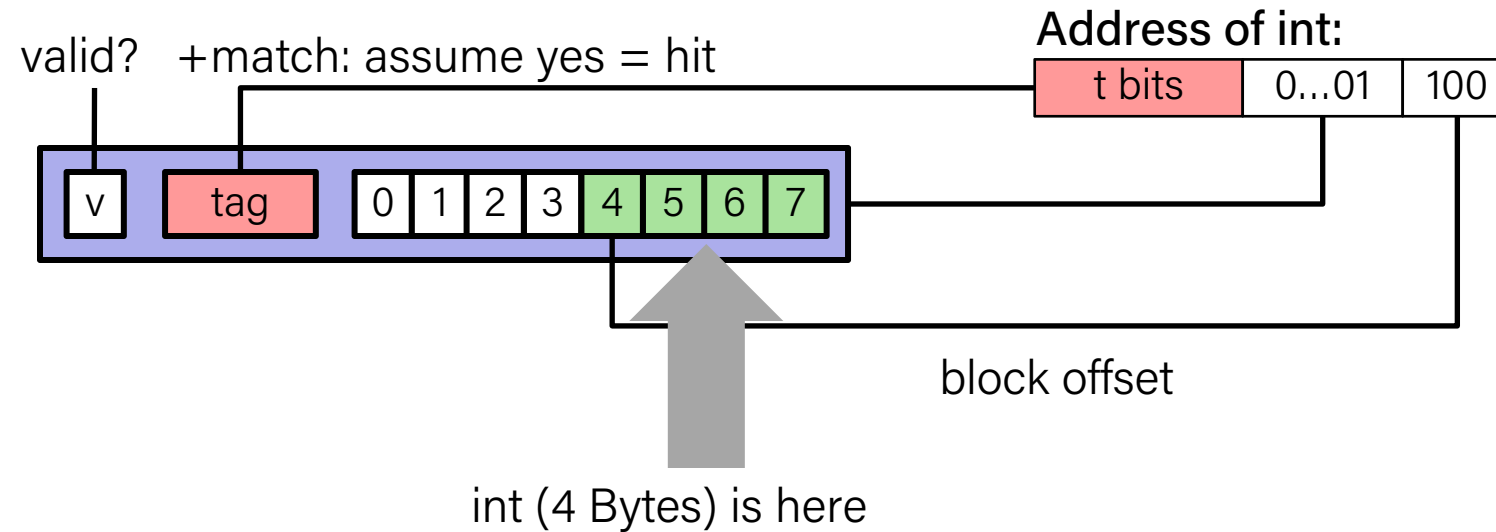
Direct mapped: One line per set
Assume: cache block size 8 bytes

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: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1   s=2   b=1

x	xx	x

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

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]	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1			
Set 2			
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 8 bytes

Address of short int:

t bits	0...01	100

| 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 | find set

| 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 |

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

E = 2: Two lines per set
Assume: cache block size 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 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 |

short int (2 Bytes) is here

block offset
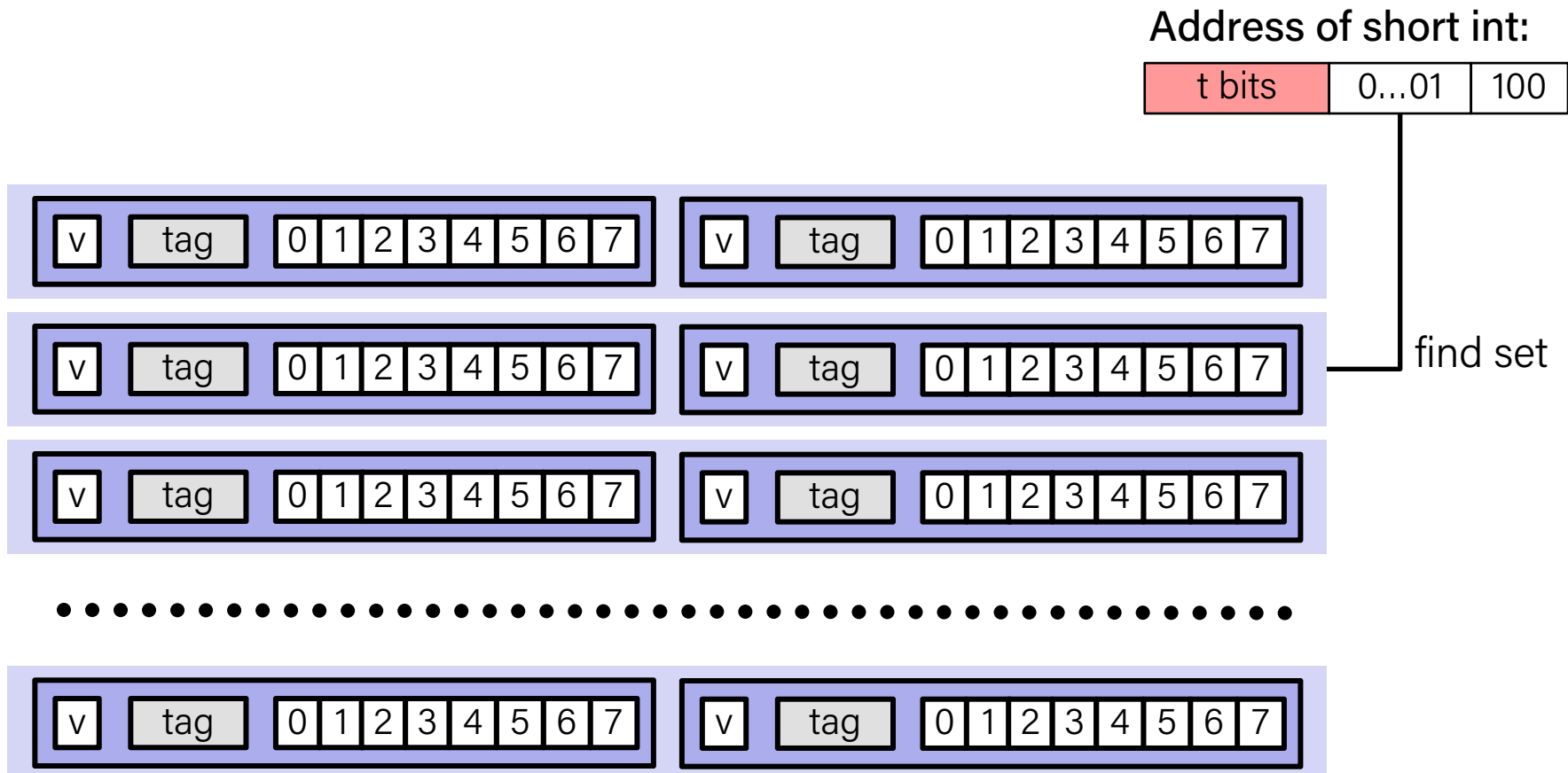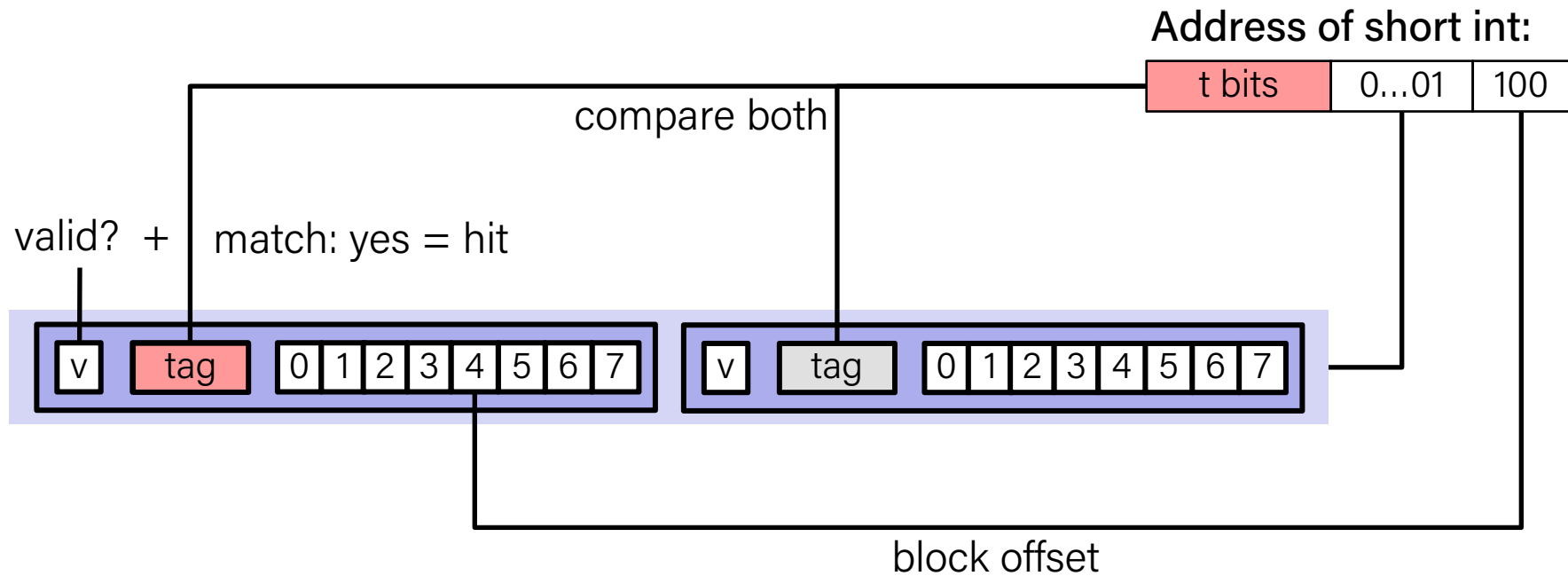
No match:
- One line in set is selected for eviction and replacement
- Replacement policies: random, least recently used (LRU), ...

2-Way Set Associative Cache Simulation

t=2 s=1 b=1

XX	X	X

M=16 byte addresses, B=2 bytes/block,
S=2 sets, E=2 blocks/set

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

- What to do on a write-hit?
  - Write-through (write immediately to memory)
  - Write-back (defer write to memory until replacement of line)
    - Need a dirty bit (line different from memory or not)

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

- Typical
  - Write-through + No-write-allocate
  - Write-back + Write-allocate

Write-back, Write Allocate Example

Note: While unrealistic, this example assumes that all requests have offset 0 and are for a block's worth of data.

Cache:

Valid	Dirty	Tag	Block Contents
1	0	G	0xBEEF
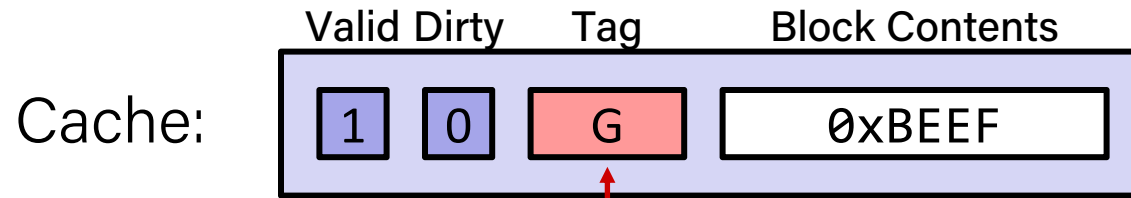
There is only one set in this tiny cache,
so the tag is the entire block number!

Memory:
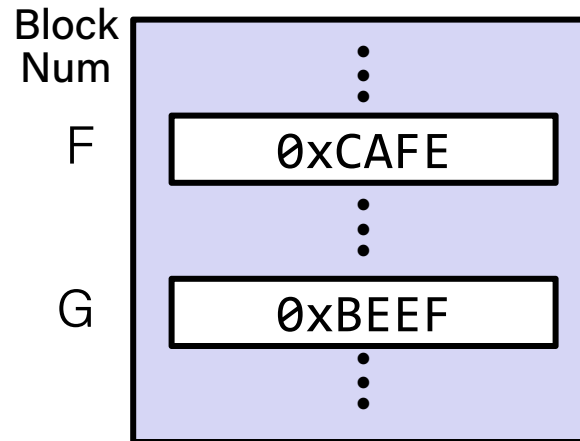
Block Num

F	0xCAFE
G	0xBEEF

Write-back, Write Allocate Example

1) `mov $0xFACE, (F)`

Write Miss!

Cache:

Valid	Dirty	Tag	Block Contents
1	0	G	0xBEEF

Step 1: Bring `F` into cache

Memory:

Block Num

F	0xCAFE
G	0xBEEF

22

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
    Write Miss

Valid Dirty   Tag   Block Contents

Cache:
| 1 | 0 | F | 0xCAFE |

Step 1: Bring `F` into cache

Memory:

Block Num

F   0xCAFE

G   0xBEEF

Step 2: Write `0xFACE` to cache only and set the dirty bit

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
   Write Miss

Valid Dirty   Tag   Block Contents

Cache: | 1 | 1 | F | 0xFACE |

Step 1: Bring `F` into cache

Memory:

Block Num

F   0xCAFE

G   0xBEEF

Step 2: Write `0xFACE` to cache only and set the dirty bit

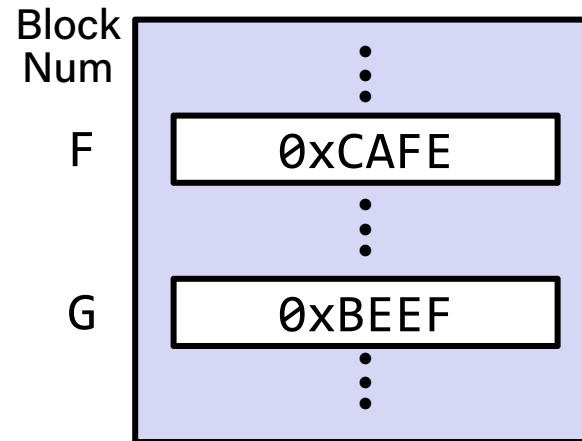# Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
     Write Miss

(2) `mov $0xFEED, (F)`
     Write Hit!

Cache:

Valid	Dirty	Tag	Block Contents
1	1	F	0xFACE
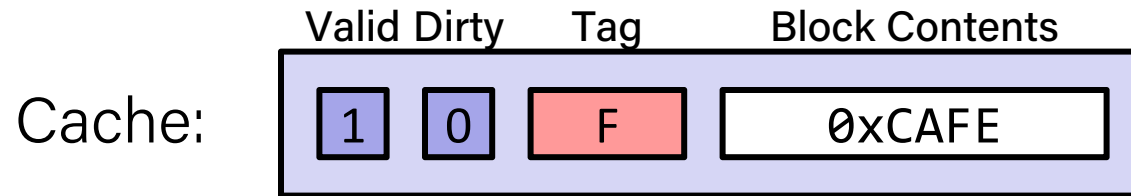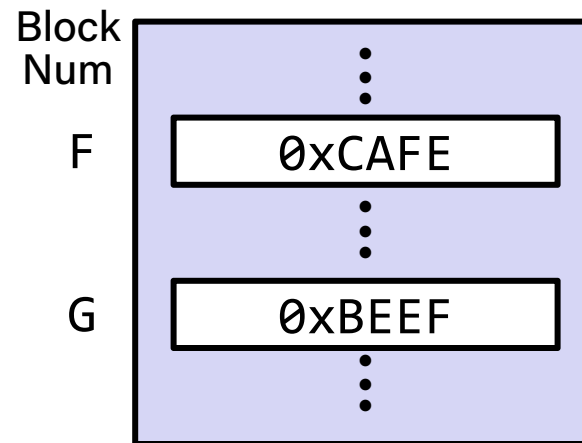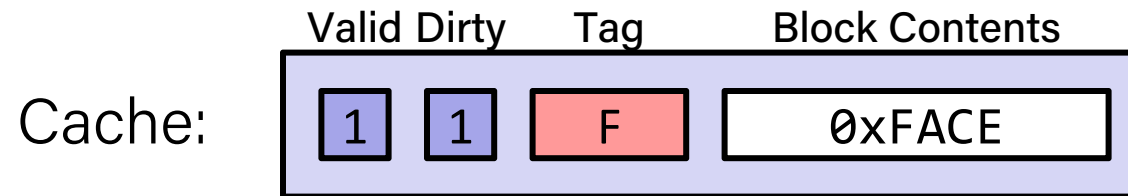
Step: Write `0xFEED` to cache only (and set the dirty bit)

Memory:
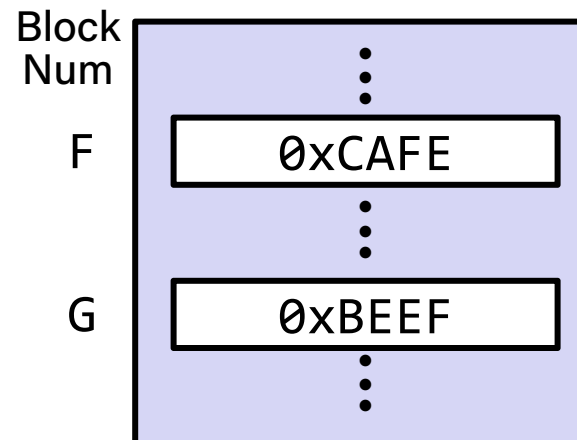
Block Num	
⋮	
F	0xCAFE
⋮	
G	0xBEEF
⋮	

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
    Write Miss

(2) `mov $0xFEED, (F)`
    Write Hit!

Cache:

Valid	Dirty	Tag	Block Contents
1	1	F	0xFEED

Memory:

Block Num
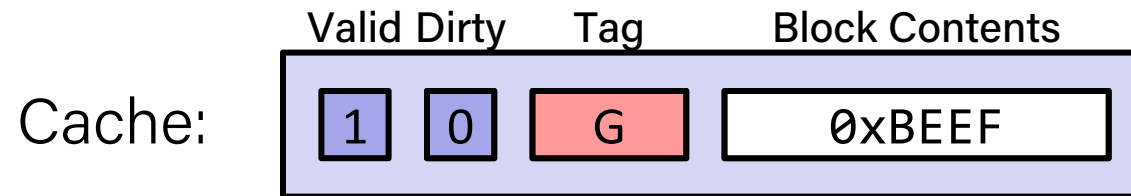
F	0xCAFE
G	0xBEEF

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
     Write Miss

(2) `mov $0xFEED, (F)`
     Write Hit!

(3) `mov (G), %ax`
     Read Miss!

Cache:

	Valid	Dirty	Tag	Block Contents
	1	1	F	0xFEED
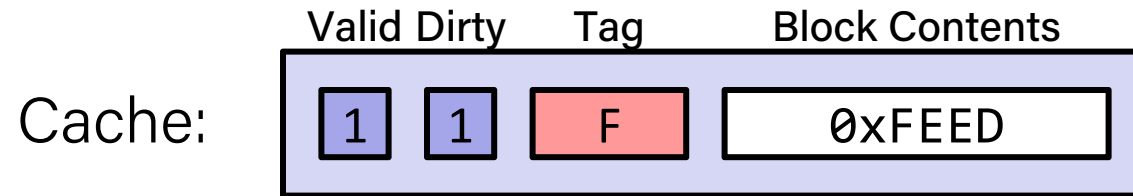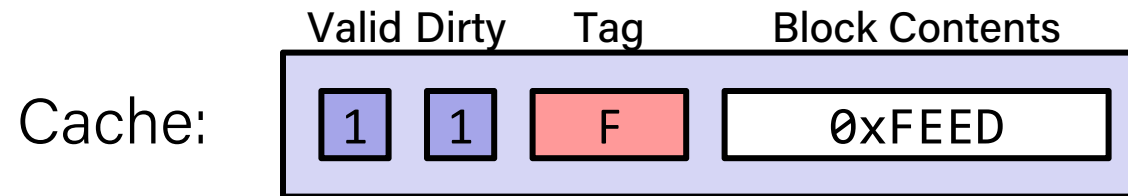
Step 1: Write **F** back to memory since it is dirty

Memory:

Block Num

F | 0xCAFE

G | 0xBEEF

Write-back, Write Allocate Example

(1) `mov $0xFACE, (F)`
Write Miss

(2) `mov $0xFEED, (F)`
Write Hit!

(3) `mov (G), %ax`
Read Miss!

Cache:

Valid	Dirty	Tag	Block Contents
1	0	G	0xBEEF

Memory:

Block Num

F	0xFEED
G	0xBEEF

<u>Step 1</u>: Write `F` back to memory since it is dirty
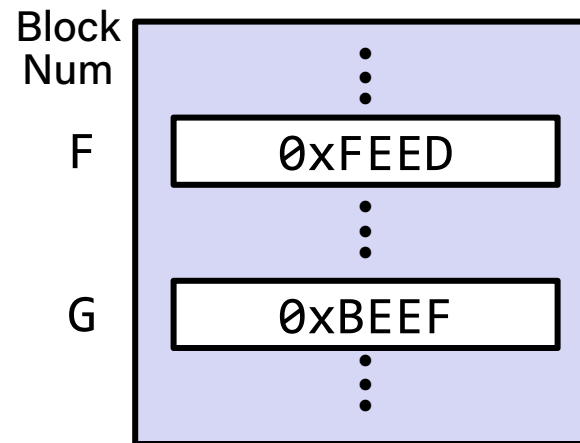
<u>Step 2</u>: Bring `G` into the cache so that we can copy it into `%ax`

28

Cache Simulator

https://courses.cs.washington.edu/courses/cse351/cachesim

Polling Question

- Which of the following cache statements is FALSE?

  A. We can reduce compulsory misses by decreasing our block size

  B. We can reduce conflict misses by increasing associativity

  C. A write-back cache will save time for code with good temporal locality on writes

  D. A write-through cache will always match data with the memory hierarchy level below it

  E. We're lost...

Polling Question

- Which of the following cache statements is FALSE?

A. **We can reduce compulsory misses by decreasing our block size**
   smaller block size pulls fewer bytes into cache on a miss

B. We can reduce conflict misses by increasing associativity
   more options to place blocks before evictions occur

C. A write-back cache will save time for code with good temporal locality on writes
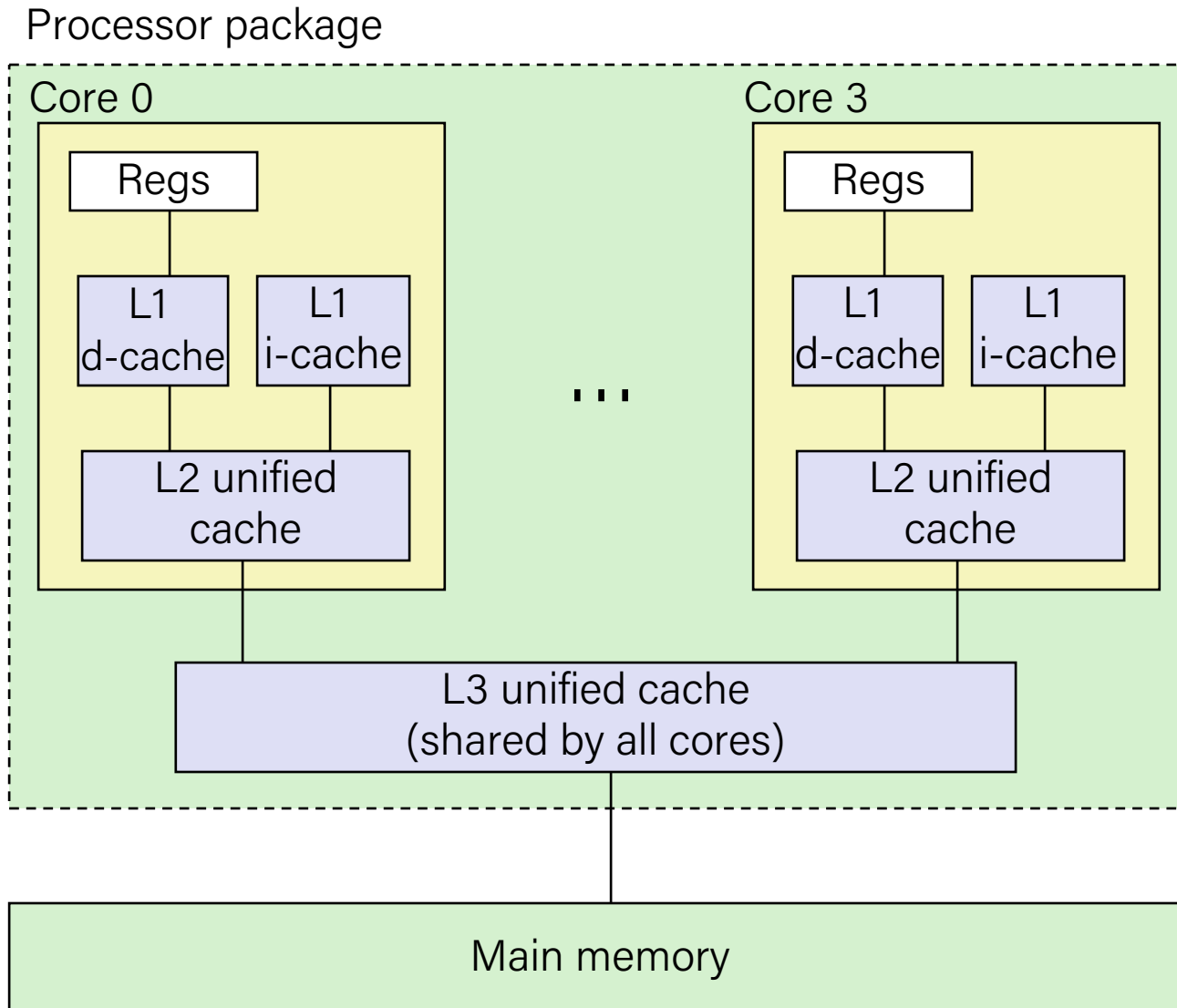   yes, its main goal is data consistency

D. A write-through cache will always match data with the memory hierarchy level below it
   frequently-used blocks rarely get evicted, so fewer write-backs

E. We're lost…

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.

Lecture Plan

- Cache memory organization and operation

- Performance impact of caches

Lecture Plan

- Cache basics

- Principle of locality

- Cache memory organization and operation

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

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

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
 *        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));
}
```

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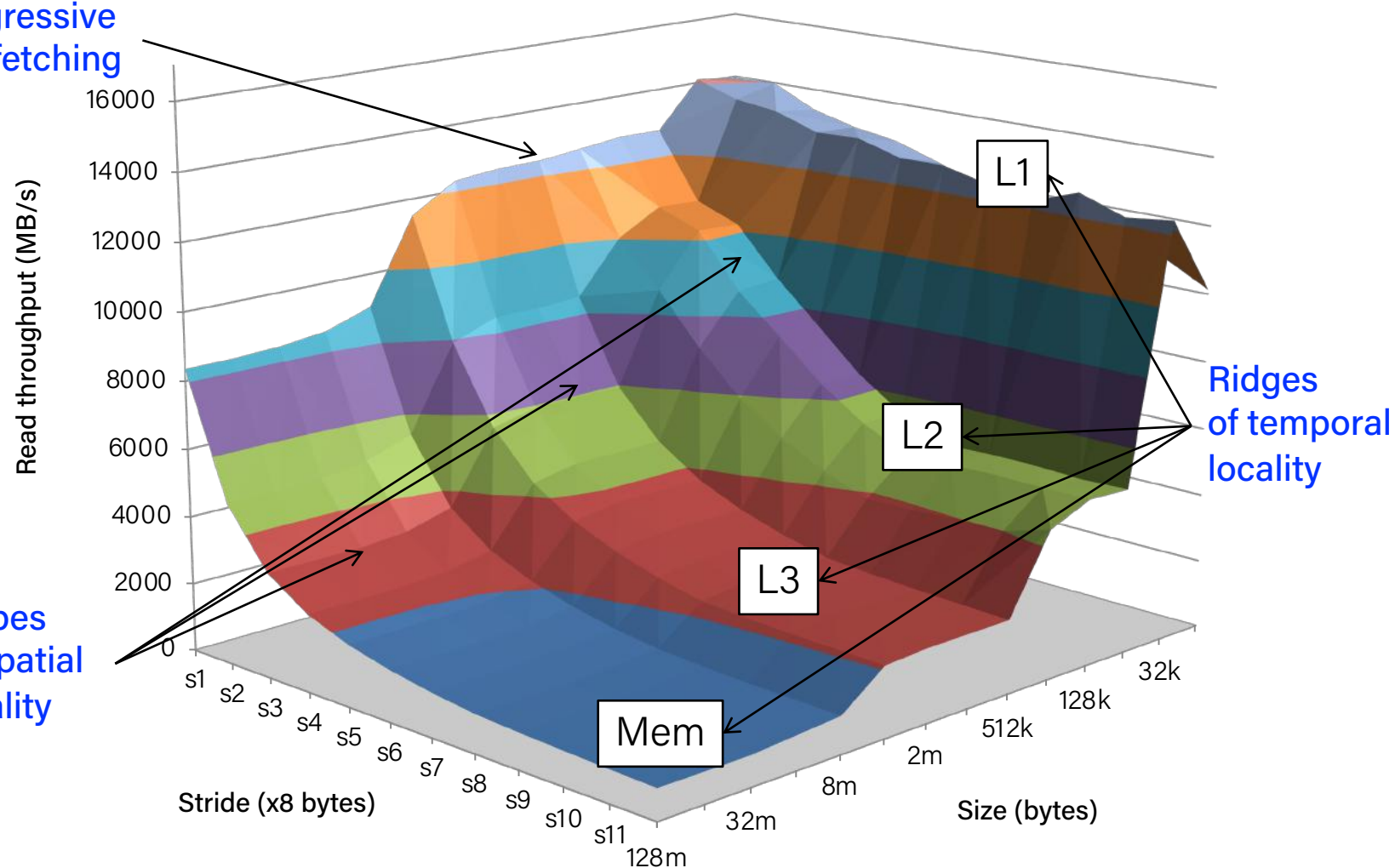
For each `elems` and `stride`:

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

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

mountain/mountain.c

The Memory Mountain


Aggressive prefetching

Slopes of spatial locality

Ridges of temporal locality

L1

L2

L3

Mem

Read throughput (MB/s)

16000
14000
12000
10000
8000
6000
4000
2000
0

Stride (x8 bytes)

s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11

Size (bytes)

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

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

38

Lecture Plan

- Cache basics

- Principle of locality

- Cache memory 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

C   A   B

c_{ij}

a_{i*}

b_{*j}

$$c_{ij} = \sum_{k=1}^{n} a_{ij} \times b_{kj}$$
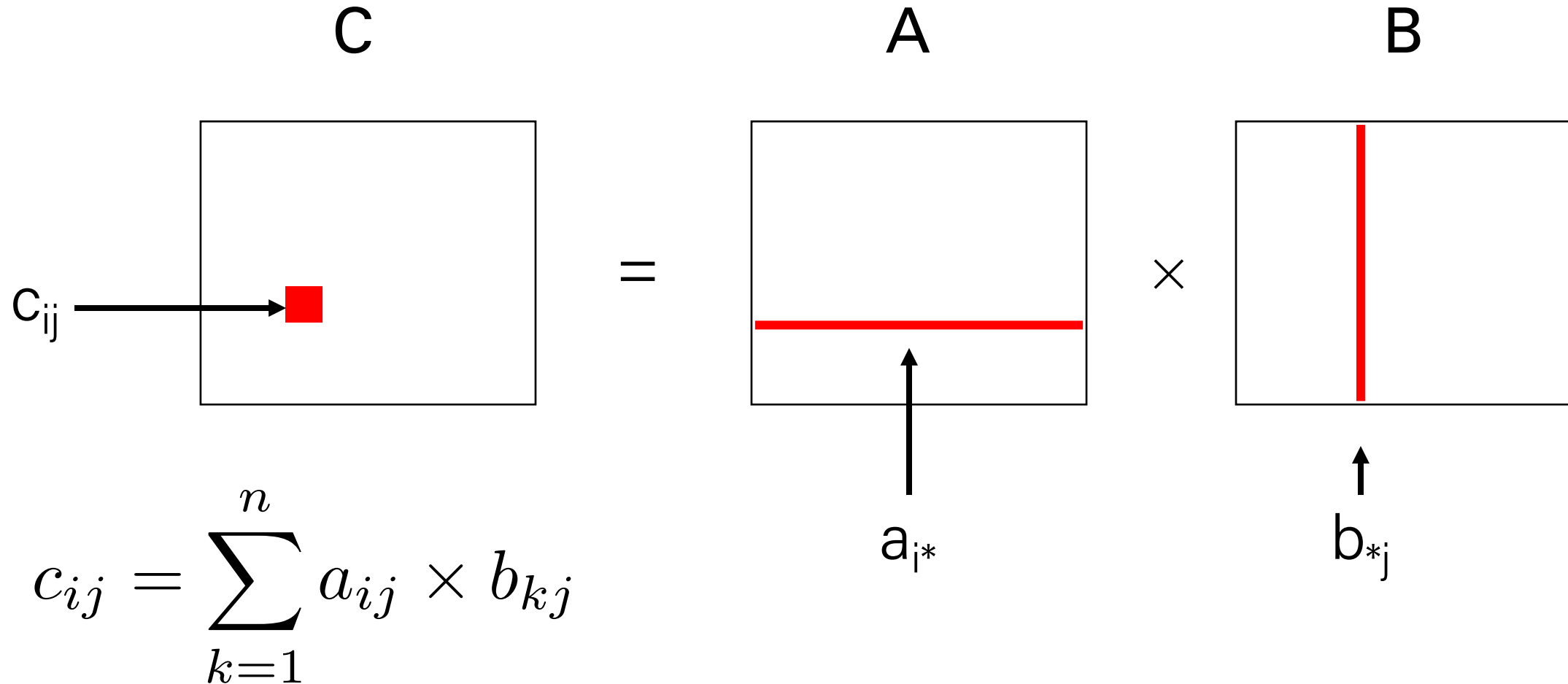
Matrix Multiplication Example

- 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

```c
/* 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 (<u>big enough for four doubles</u>)
  - 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

$$C = A \times B$$
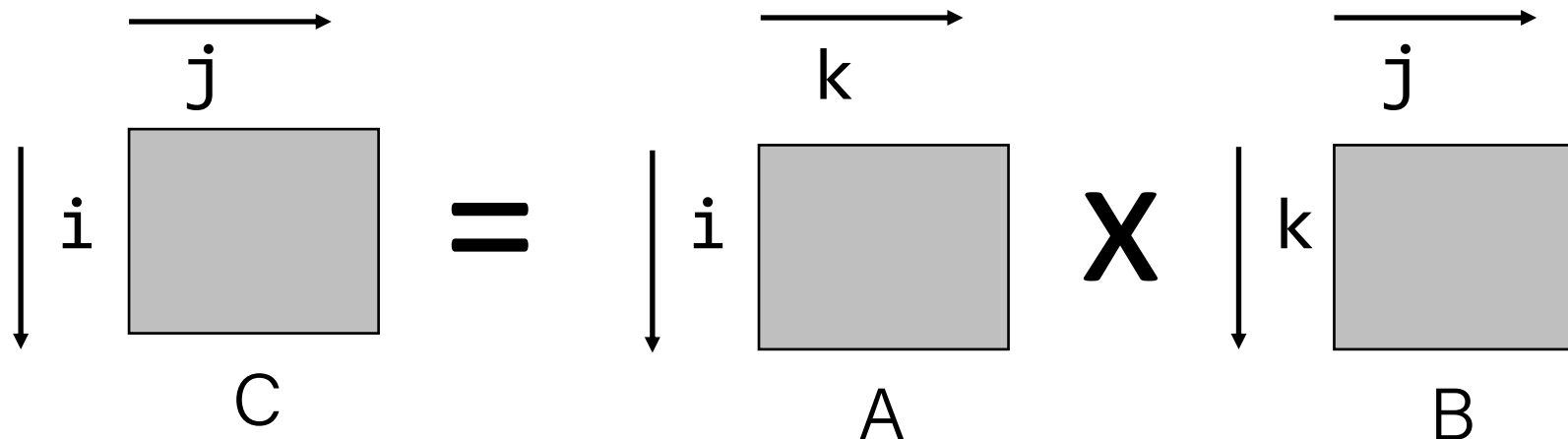
Layout of C Arrays in Memory (review)

- 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(aij) bytes, exploit spatial locality

    miss rate = sizeof(aij) / 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:

	A	B	C
	Row-wise	Column-wise	Fixed

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0
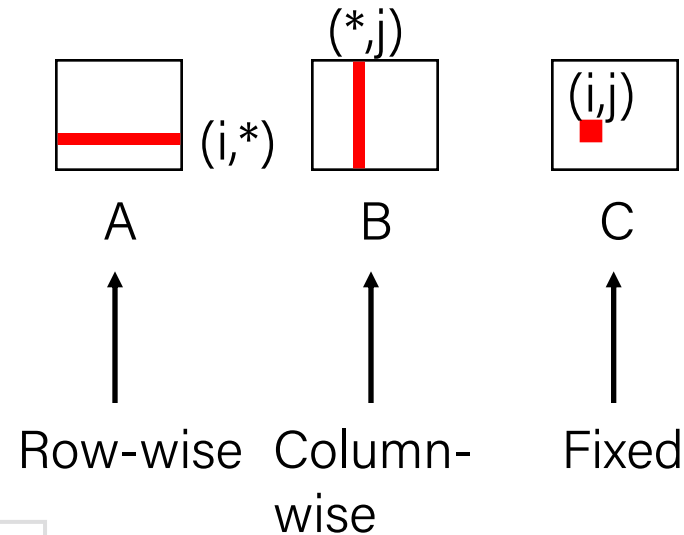
Matrix Multiplication (jik)

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

Inner loop:

Row-wise   Column-wise   Fixed

matmult/mm.c

Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0
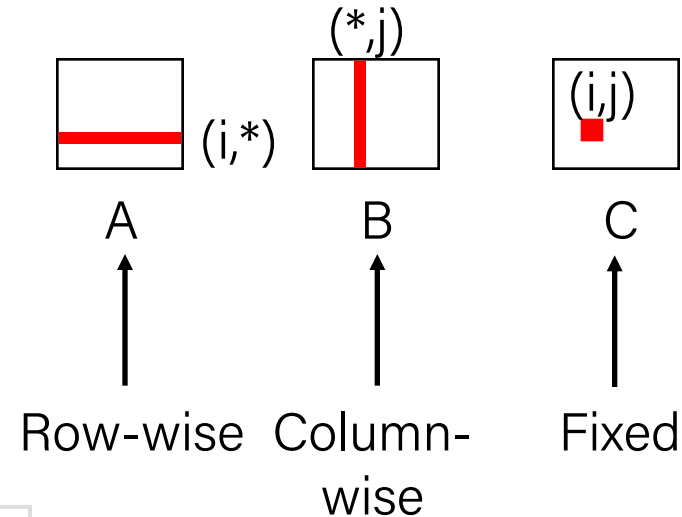
Matrix Multiplication (kij)

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

Inner loop:

| | (i,k) | (k,*) | (i,*) |
| A | B | C |

Fixed   Row-wise Row-wise

matmult/mm.c

Misses per inner loop iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25
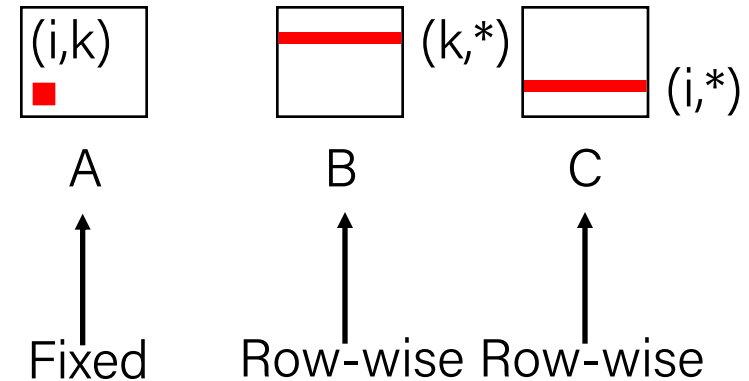
Matrix Multiplication (ikj)

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

Inner loop:

A   B   C

Fixed   Row-wise Row-wise

matmult/mm.c

Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25
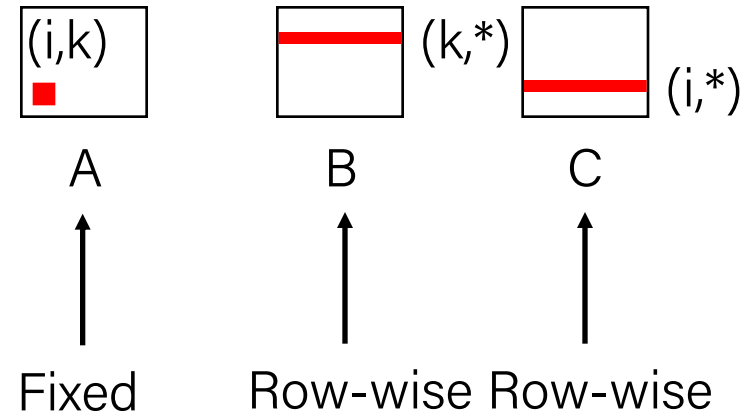
Matrix Multiplication (jki)

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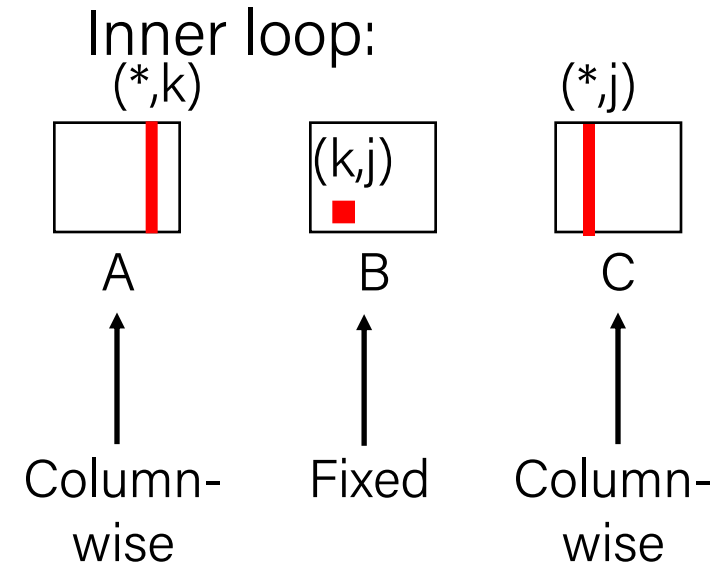
Inner loop:

matmult/mm.c

Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

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

Inner loop:

matmult/mm.c
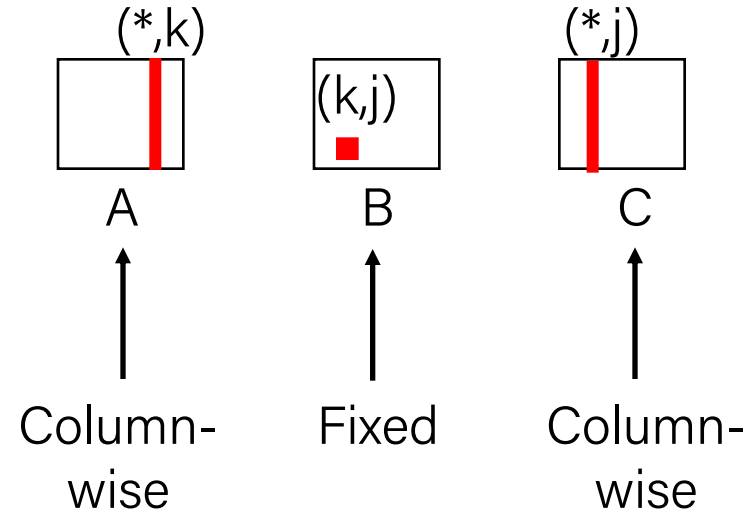
Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

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;
  }
}
```

```
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];
 }
}
```

```
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;
 }
}
```
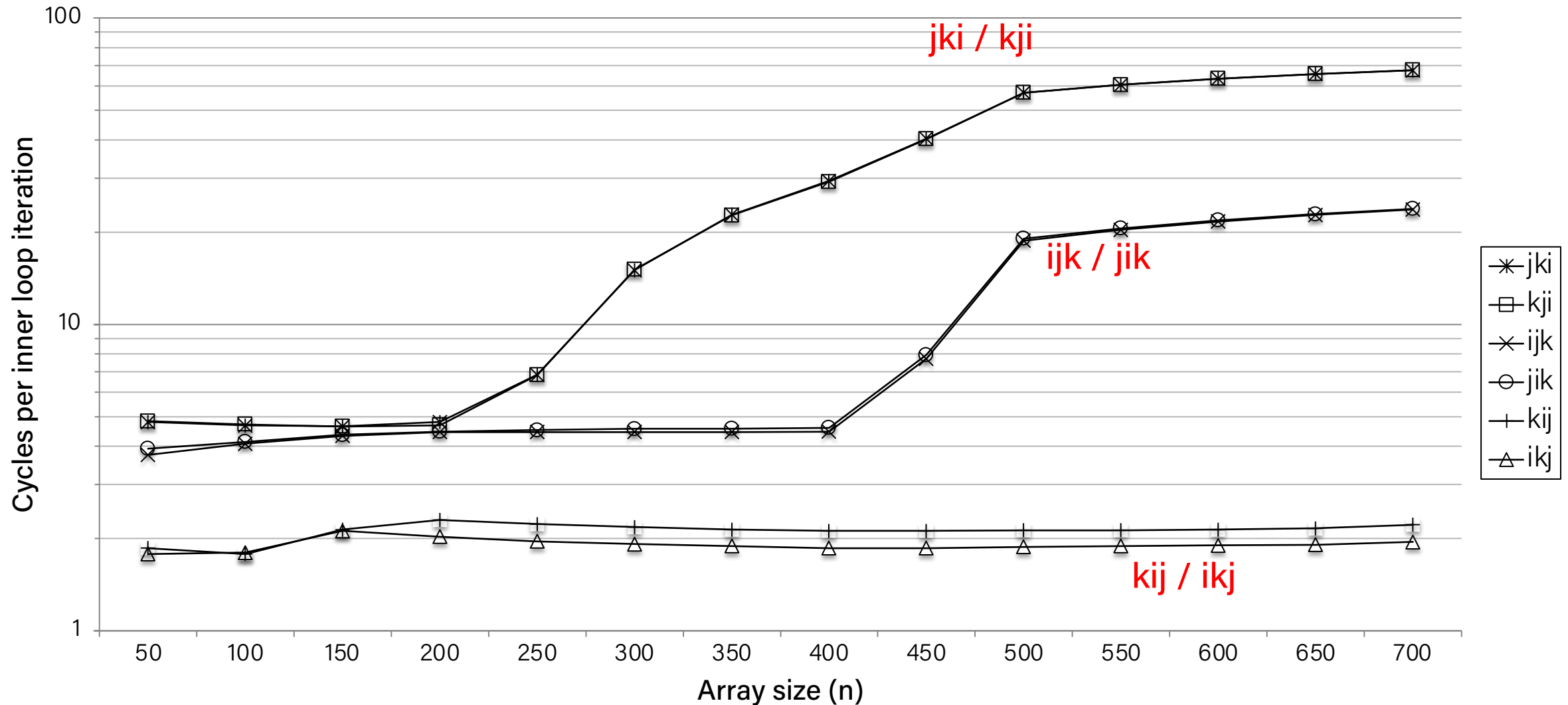
ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

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

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

Core i7 Matrix Multiply Performance

Lecture Plan

- Cache basics

- Principle of locality

- Cache memory 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 = 9n/8 misses
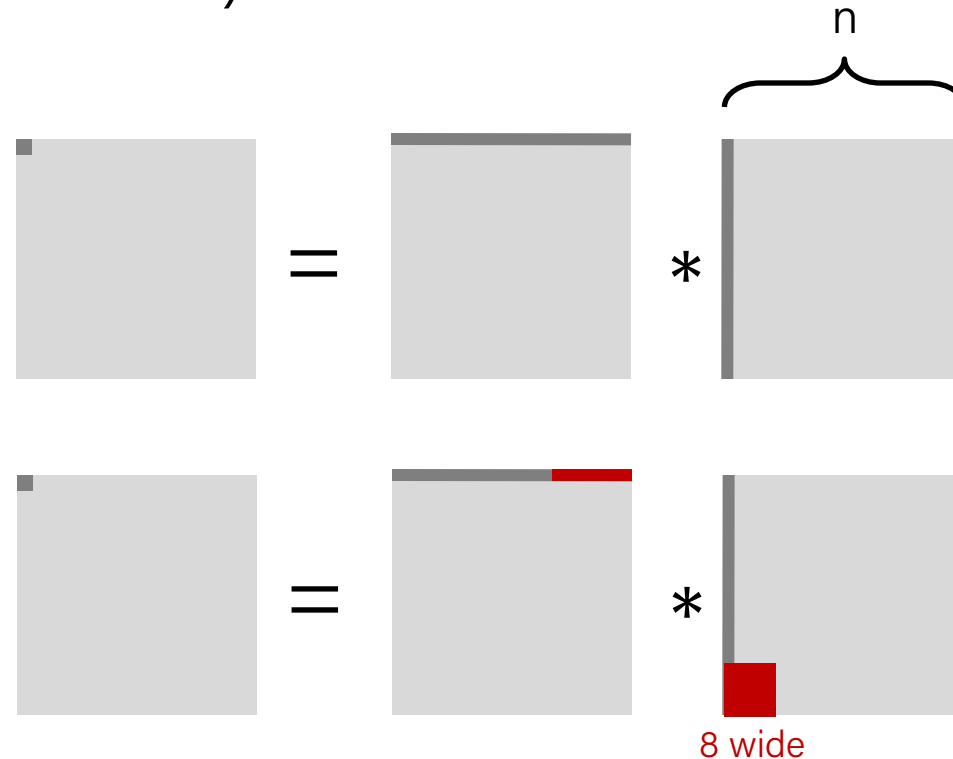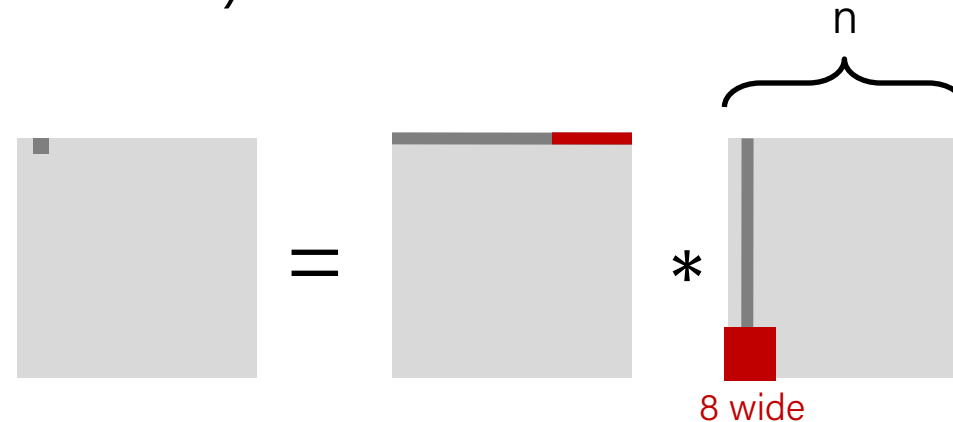
  - Afterwards in cache: (schematic)

Cache Miss Analysis

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

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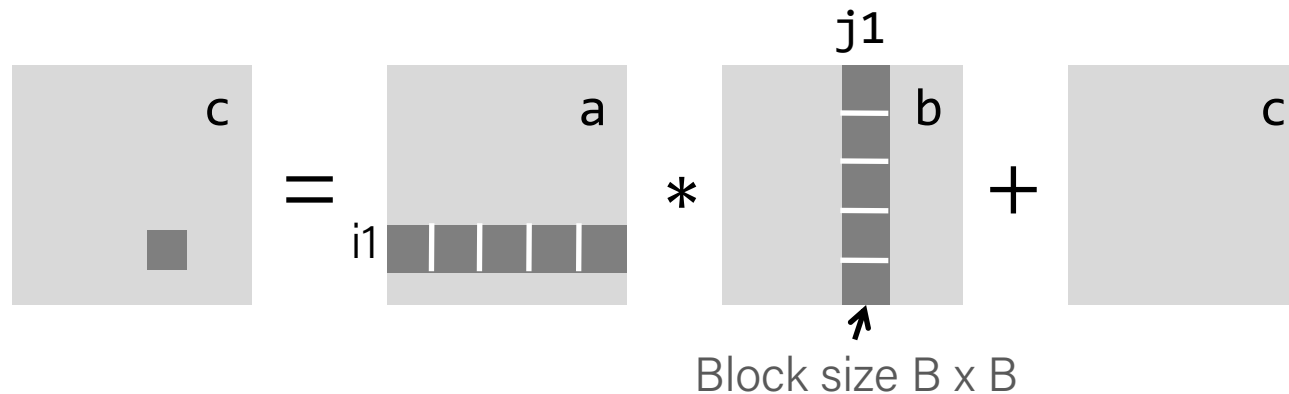 - Again:
    n/8 + n = 9n/8 misses

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

n

= *

8 wide

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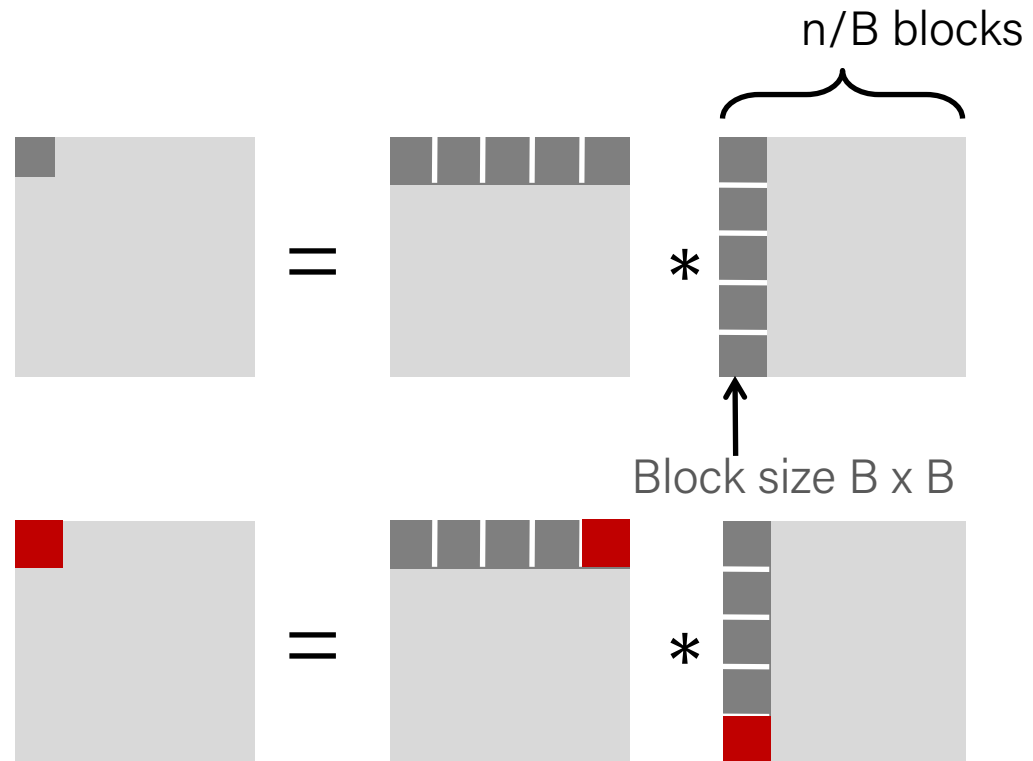
matmult/bmm.c

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

  - Afterwards in cache (schematic)

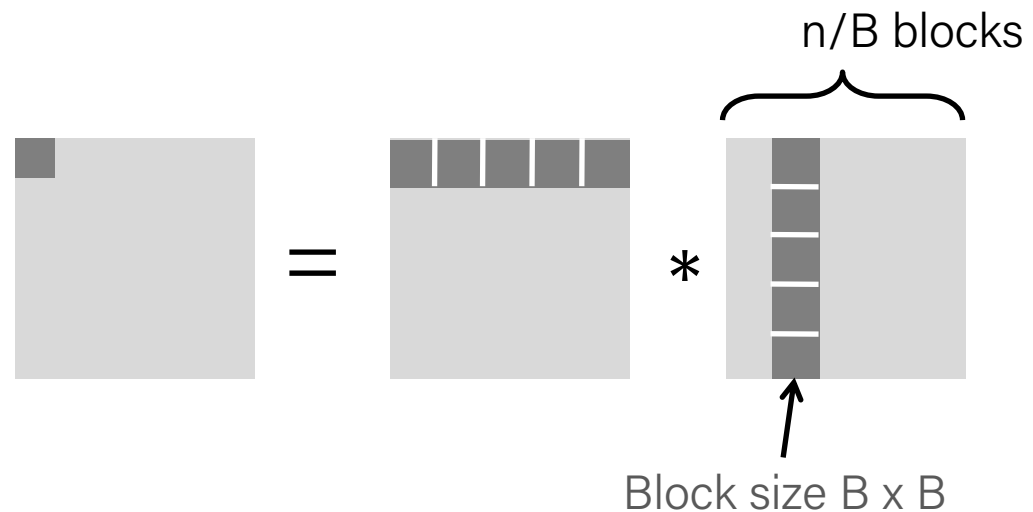n/B blocks

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: $3B^2 < C$

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

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

n/B blocks

= *

Block size B x B

Blocking Summary

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

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

- **Suggest largest possible block size B, but limit $3B^2 < C$!**

- **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
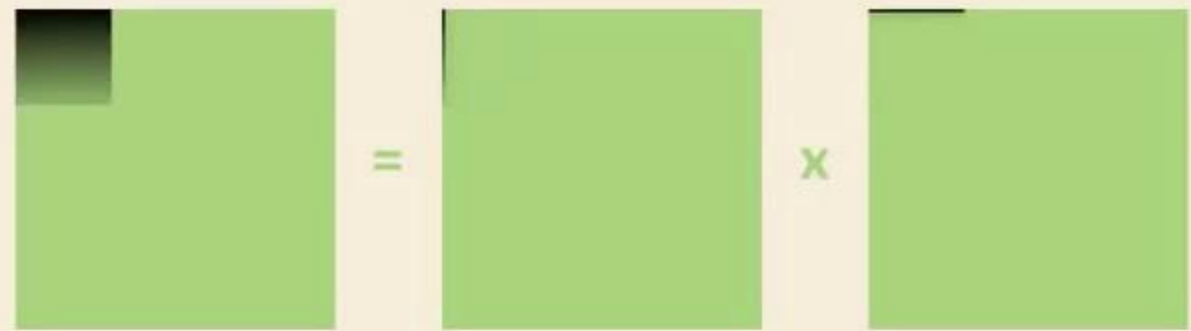
Naïve vs. Blocked Matrix Multiplication

Naïve Multiplication

Blocked Multiplication

≈ 1,020,000 cache misses

≈ 90,000 cache misses

Video credit: Vasily Volkov

Recap

- 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 with sequentially with stride 1.
  - Try to maximize temporal locality by using a data object as often as possible once it's read from memory.

Next time: Debugging and Design