CSC 252: Computer Organization
Spring 2018: Lecture 17

Instructor: Yuhao Zhu

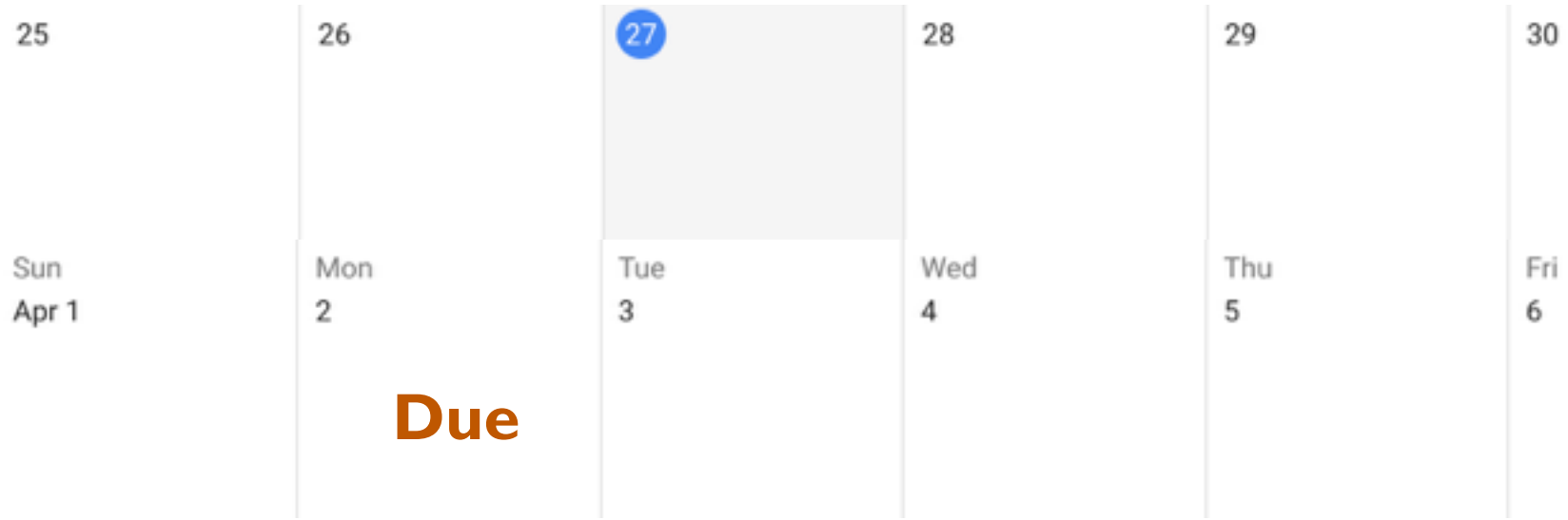
Department of Computer Science
University of Rochester

Action Items:

- **Programming Assignment 4 is out**
- **Cache Problem Set is out (no turn-in)**

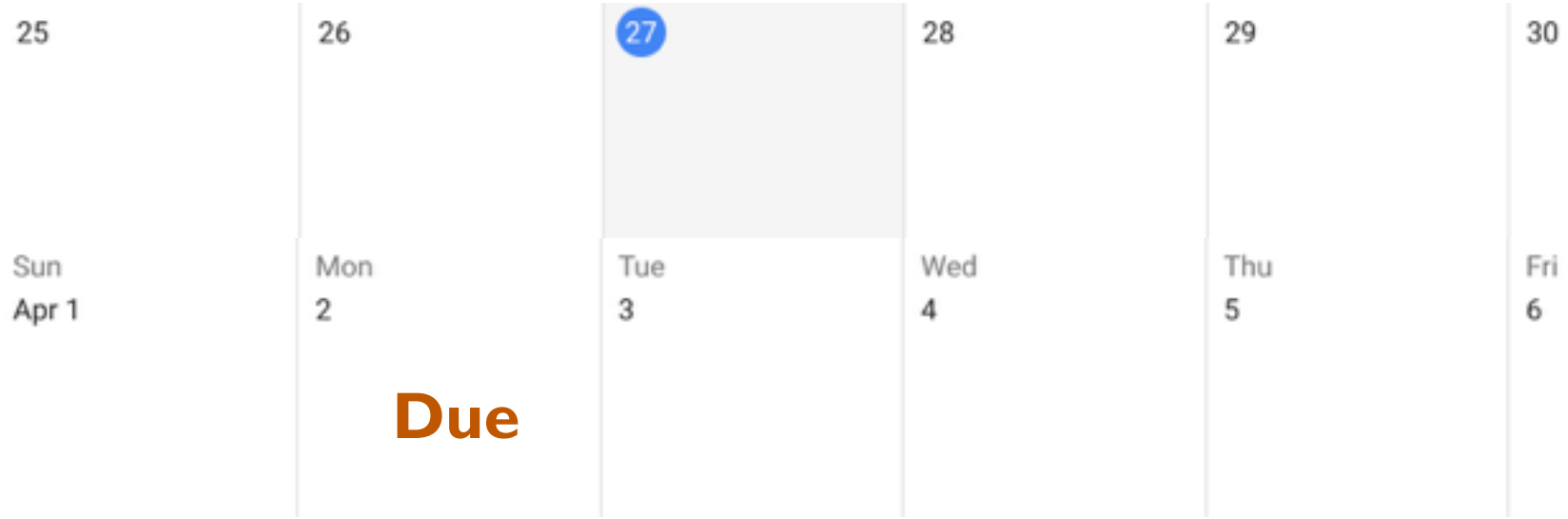
Announcement

- Programming Assignment 4 is out
  - Main assignment due on 11:59pm, **Monday, April 2**.



Announcement

- Programming Assignment 4 is out
  - Main assignment due on 11:59pm, **Monday, April 2**.
- I have put up a Problem Set to help you understand cache better
  - No turn-in required. Solutions to be released soon
  - Get a preview of final exam problems!!!



Announcement

Announcement

- Mid-term:
  - Mid-term grades will be updated after the class
  - You can get your exams from the TAs after the class
  - Solutions will be posted very soon

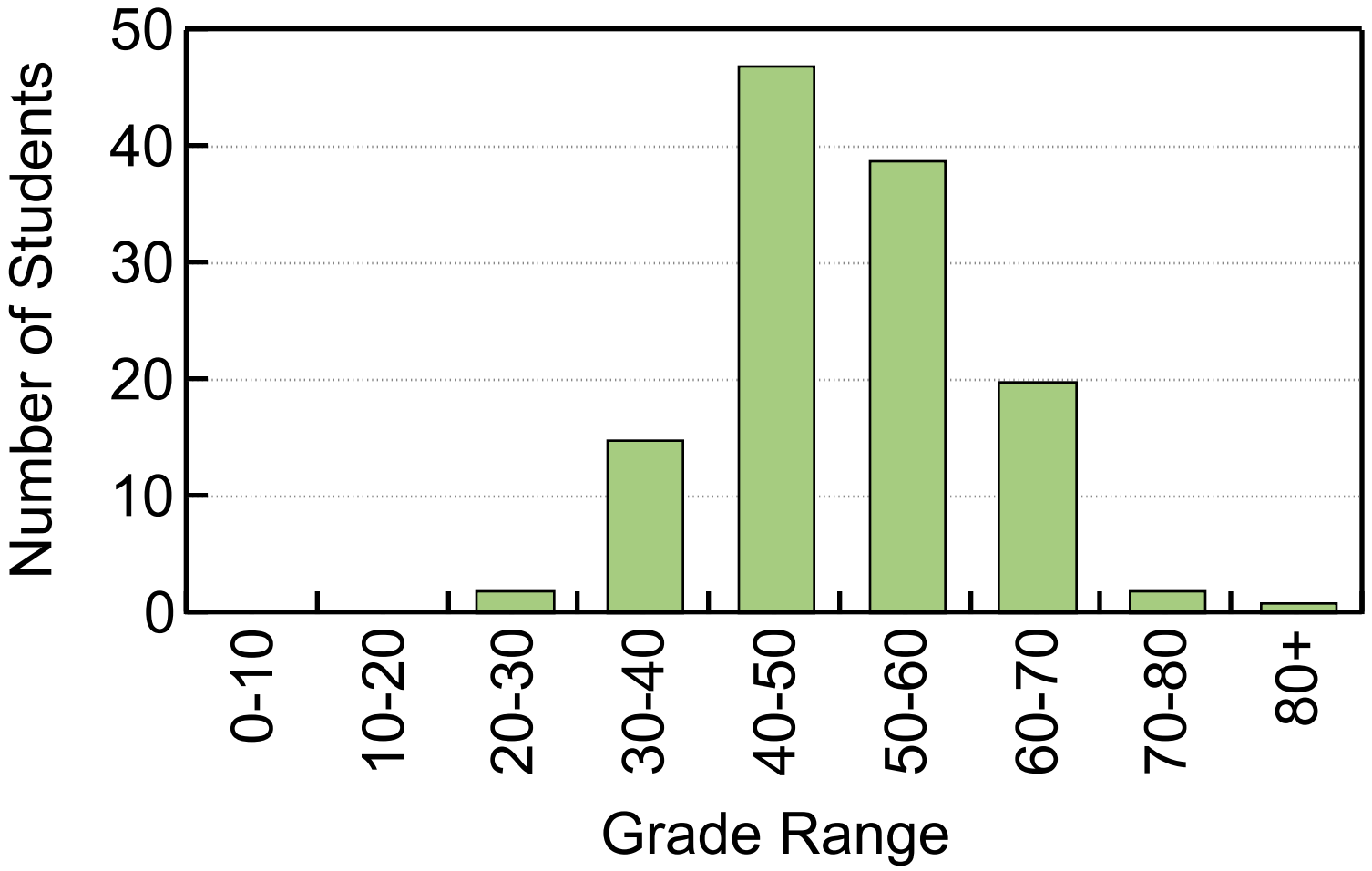
Announcement

- Mid-term:
  - Mid-term grades will be updated after the class
  - You can get your exams from the TAs after the class
  - Solutions will be posted very soon
- Statistics:
  - Average: 50.7; Standard-deviation: 10.0
  - Random guess: $1/\infty = 0$
  - Just writing down “I don’t know”: 15

Announcement

- Mid-term:
  - Mid-term grades will be updated after the class
  - You can get your exams from the TAs after the class
  - Solutions will be posted very soon
- Statistics:
  - Average: 50.7; Standard-deviation: 10.0
  - Random guess: $1/\infty = 0$
  - Just writing down “I don’t know”: 15
- So overall encouraging results

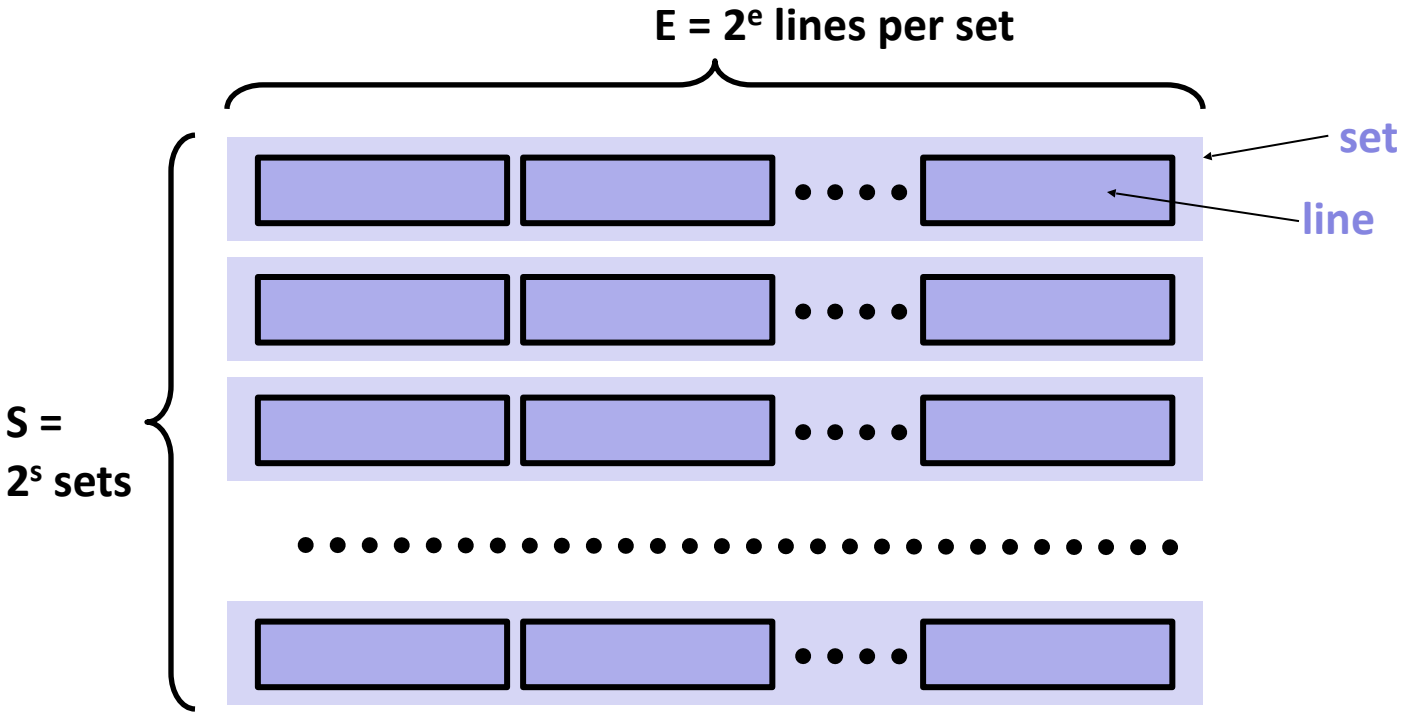
Announcement



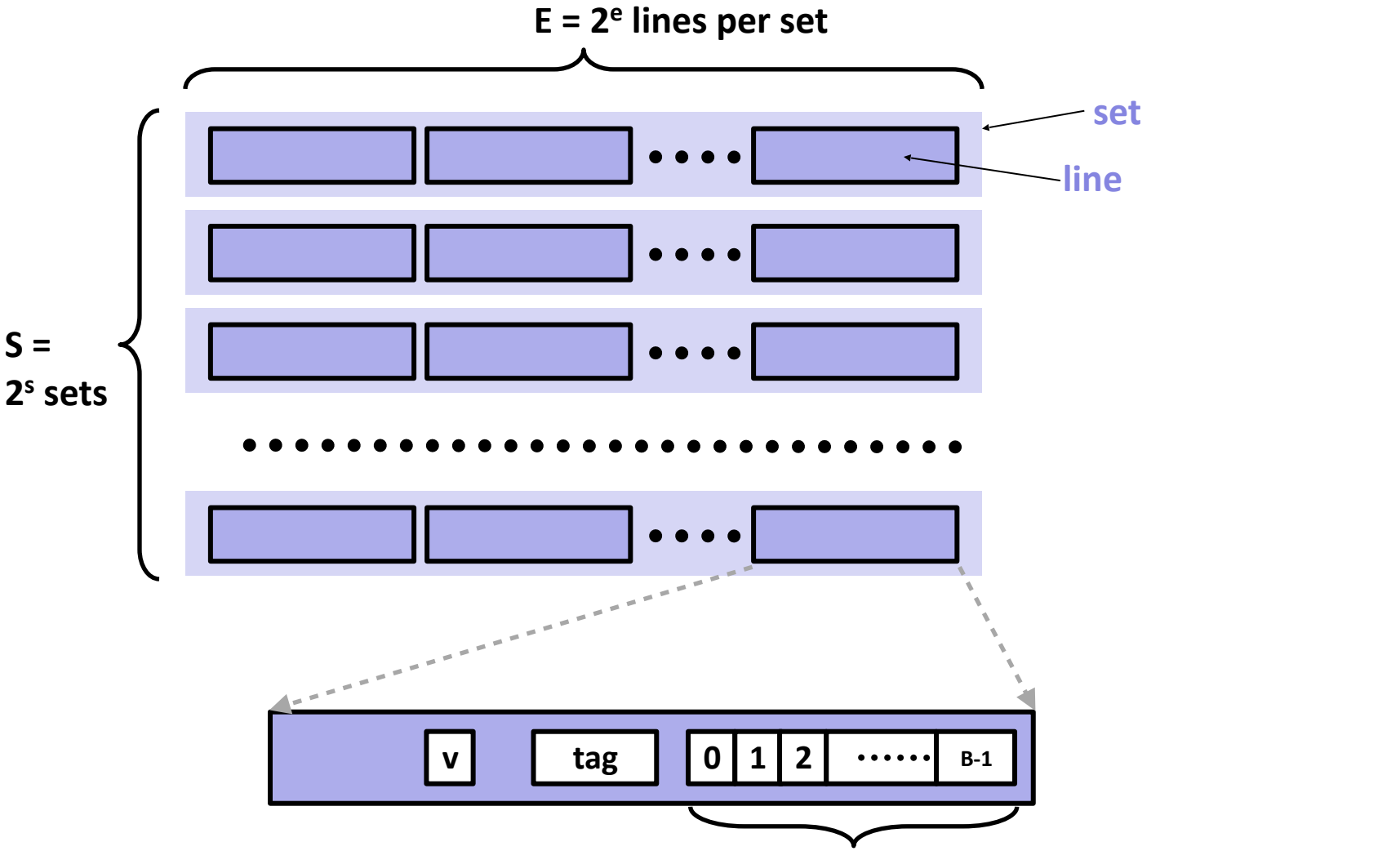
Today

- Review: Cache memory organization and operation
- Performance impact of caches
  - Analytical Model
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

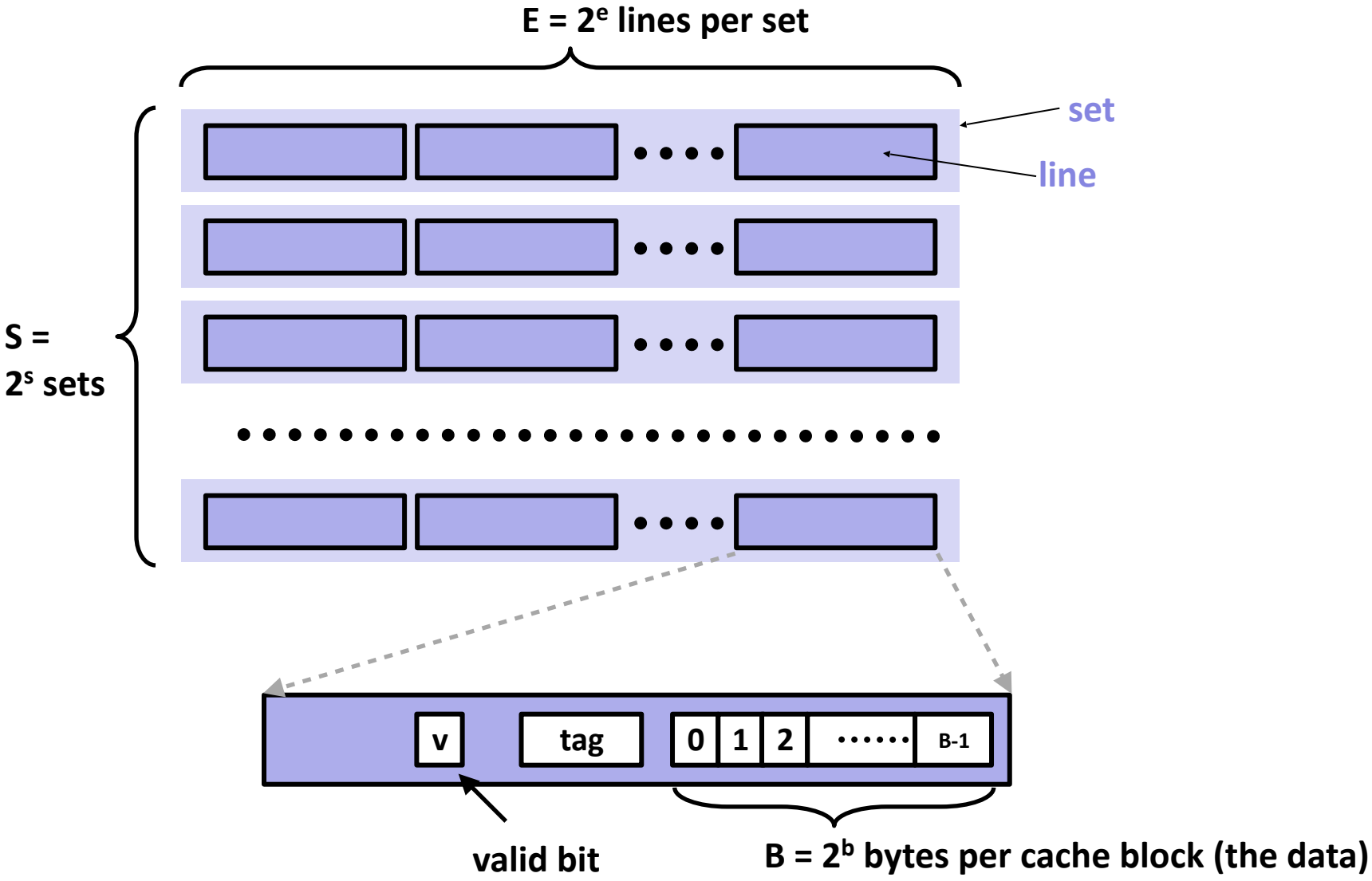
General Cache Organization (S, E, B)



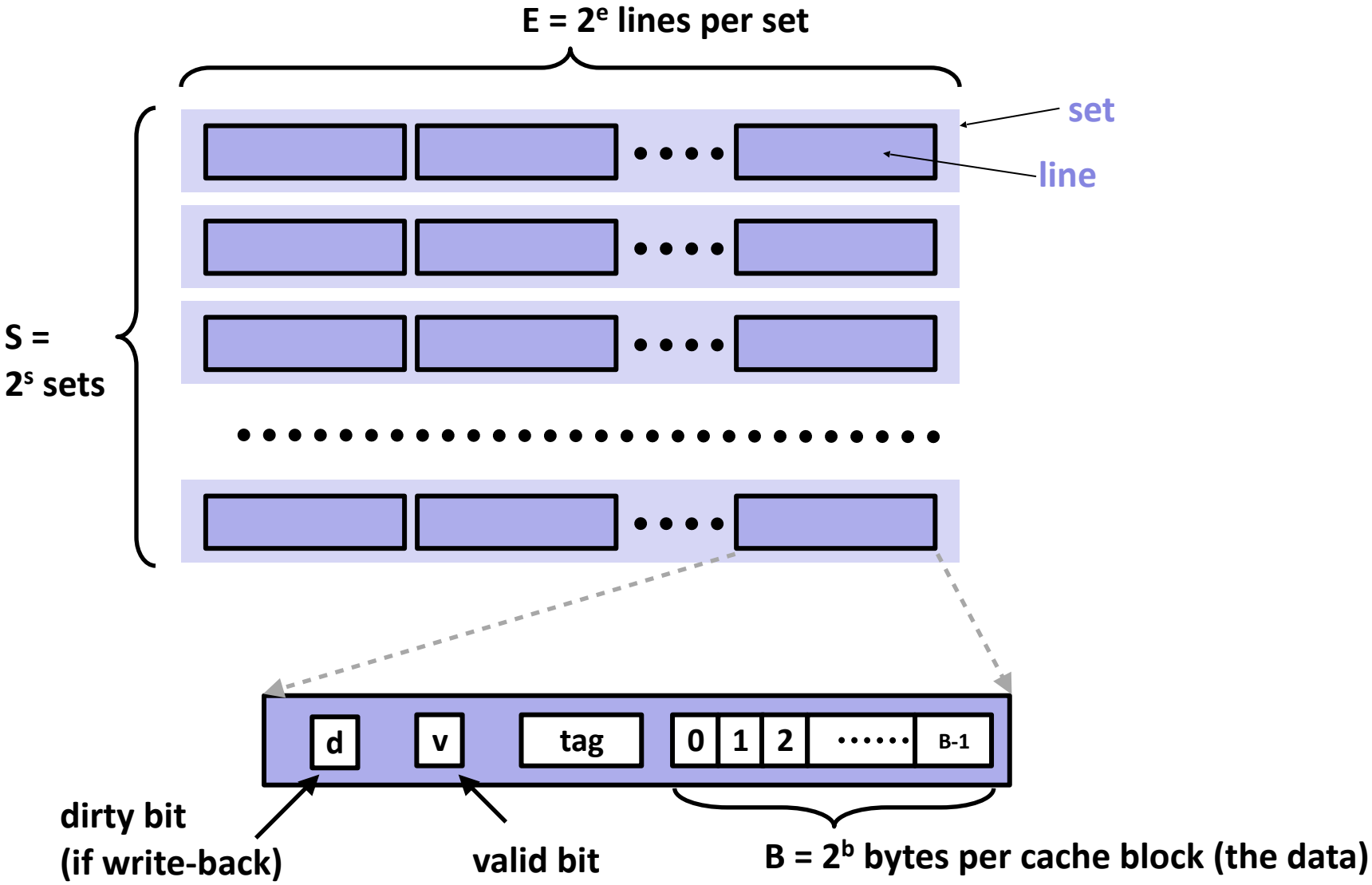
General Cache Organization (S, E, B)



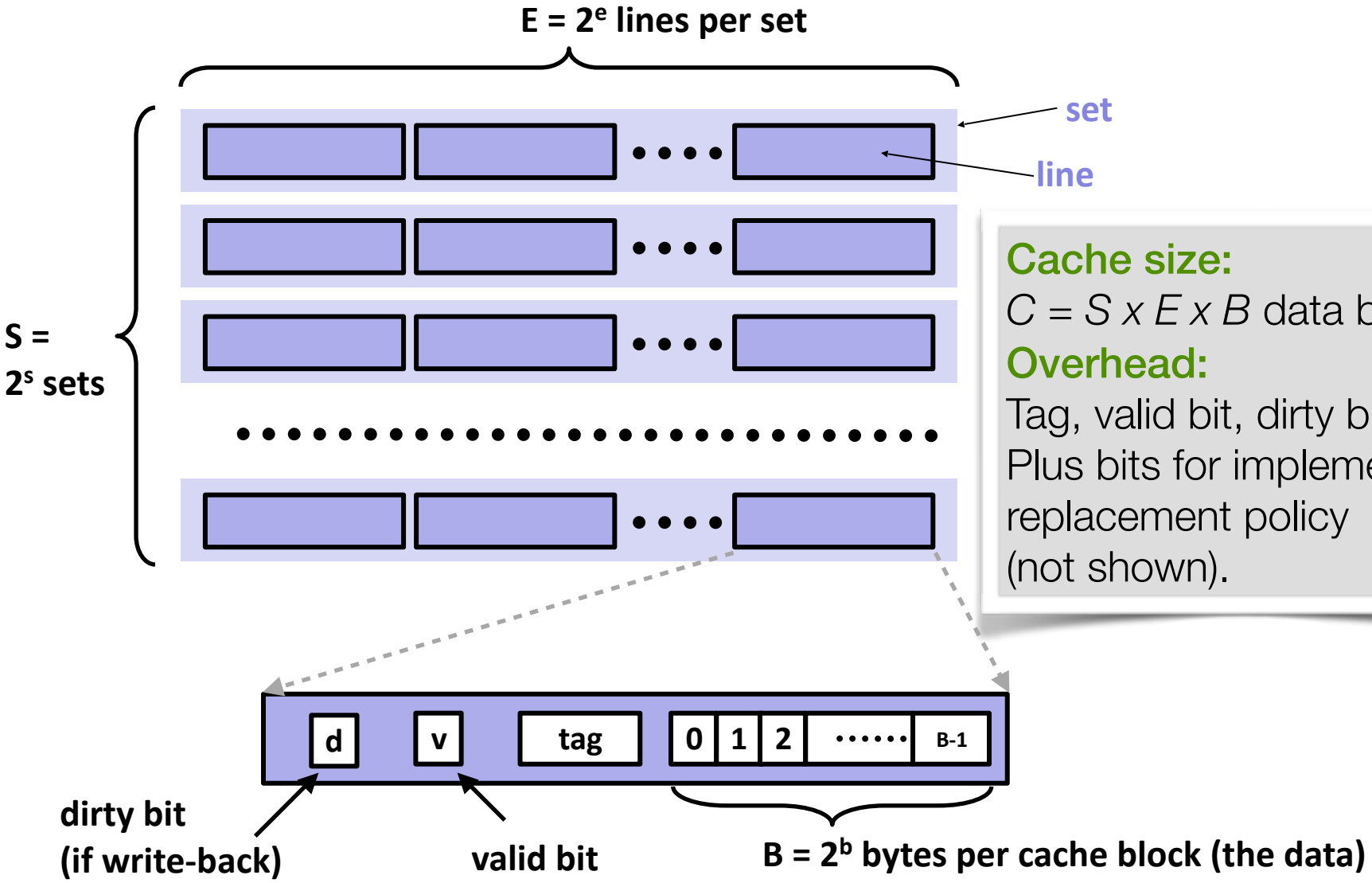
General Cache Organization (S, E, B)



General Cache Organization (S, E, B)



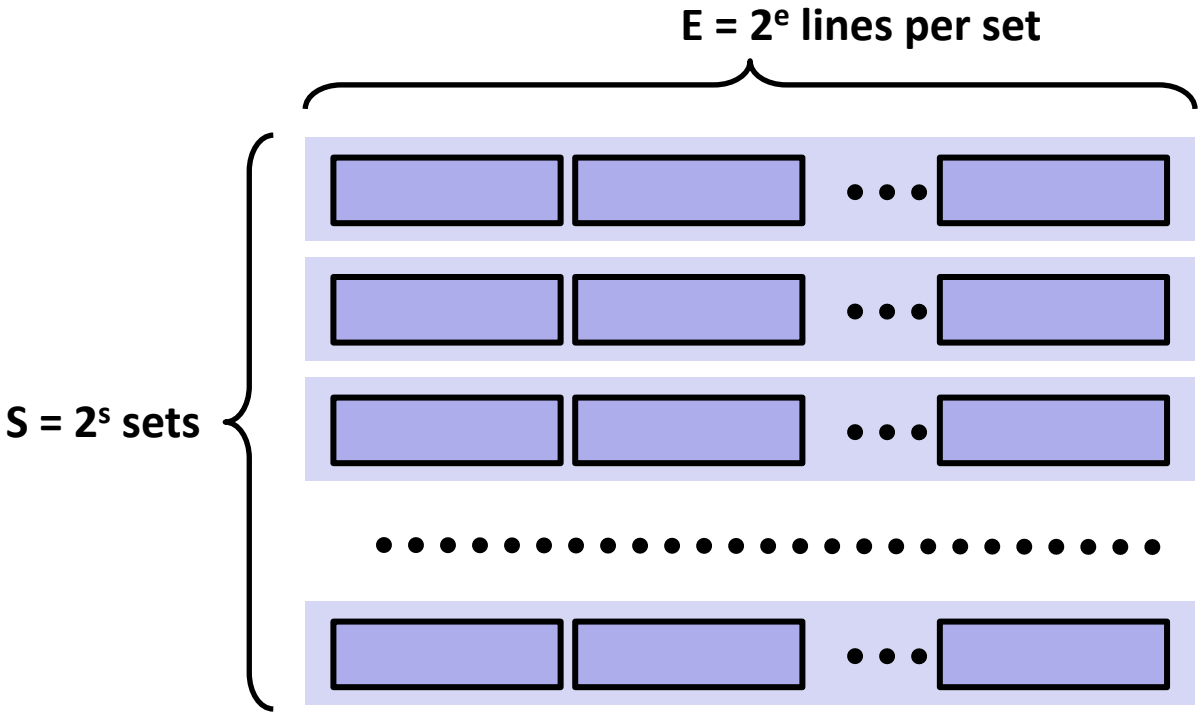
General Cache Organization (S, E, B)

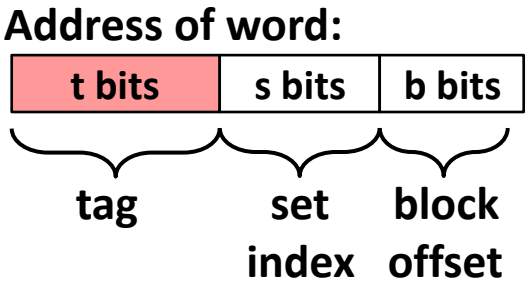


Cache size:
C = S x E x B data bytes

Overhead:
Tag, valid bit, dirty bit.
Plus bits for implementing replacement policy (not shown).

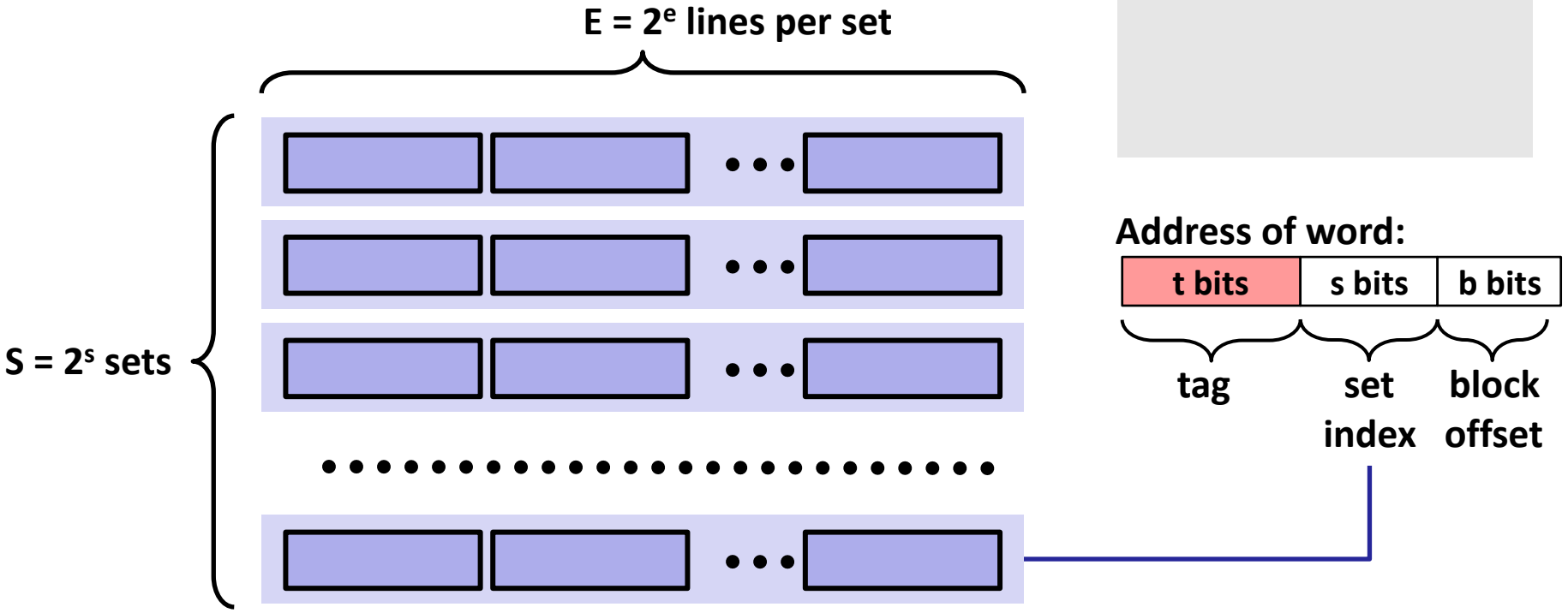
Cache Access





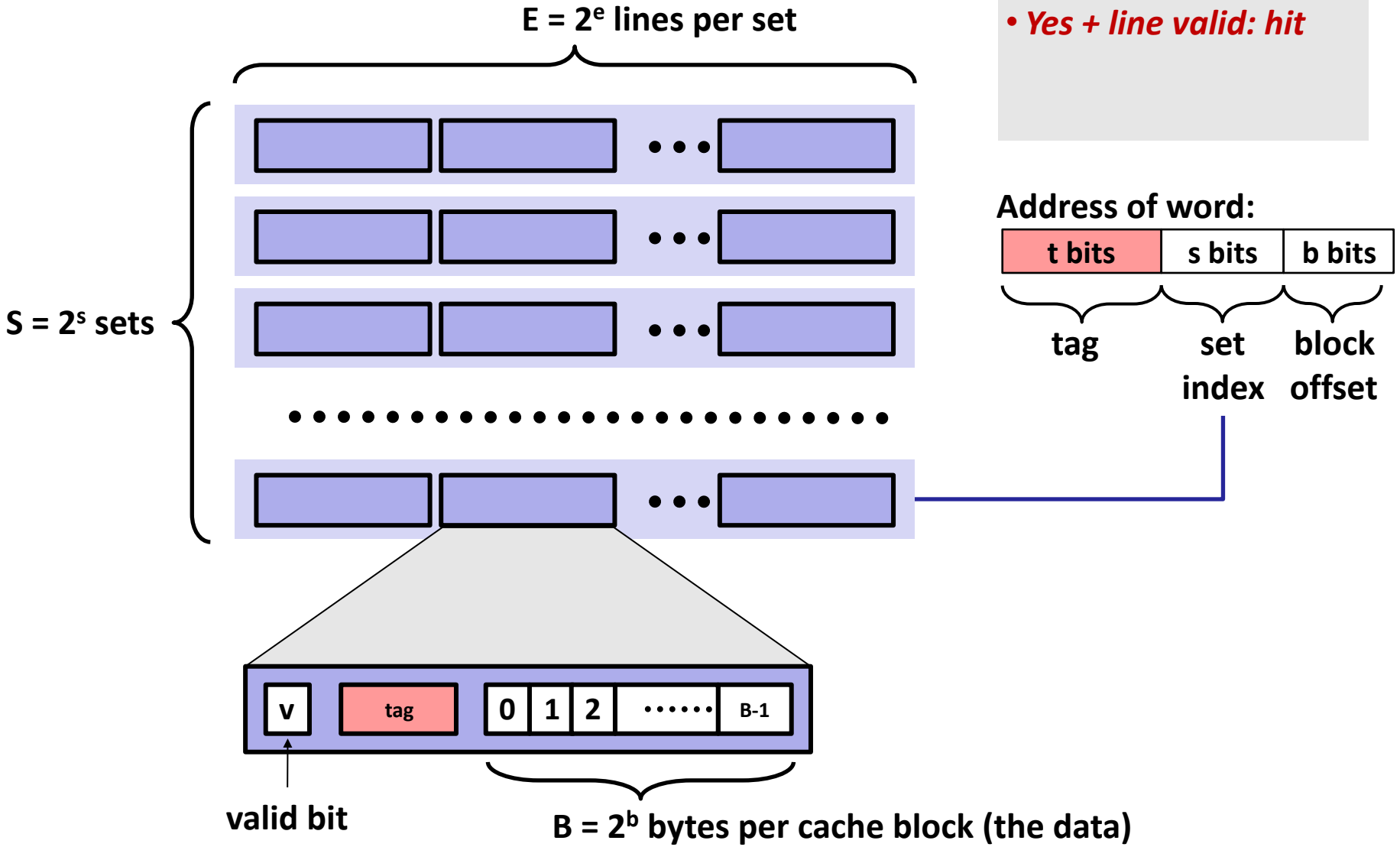
Cache Access

- *Locate set*



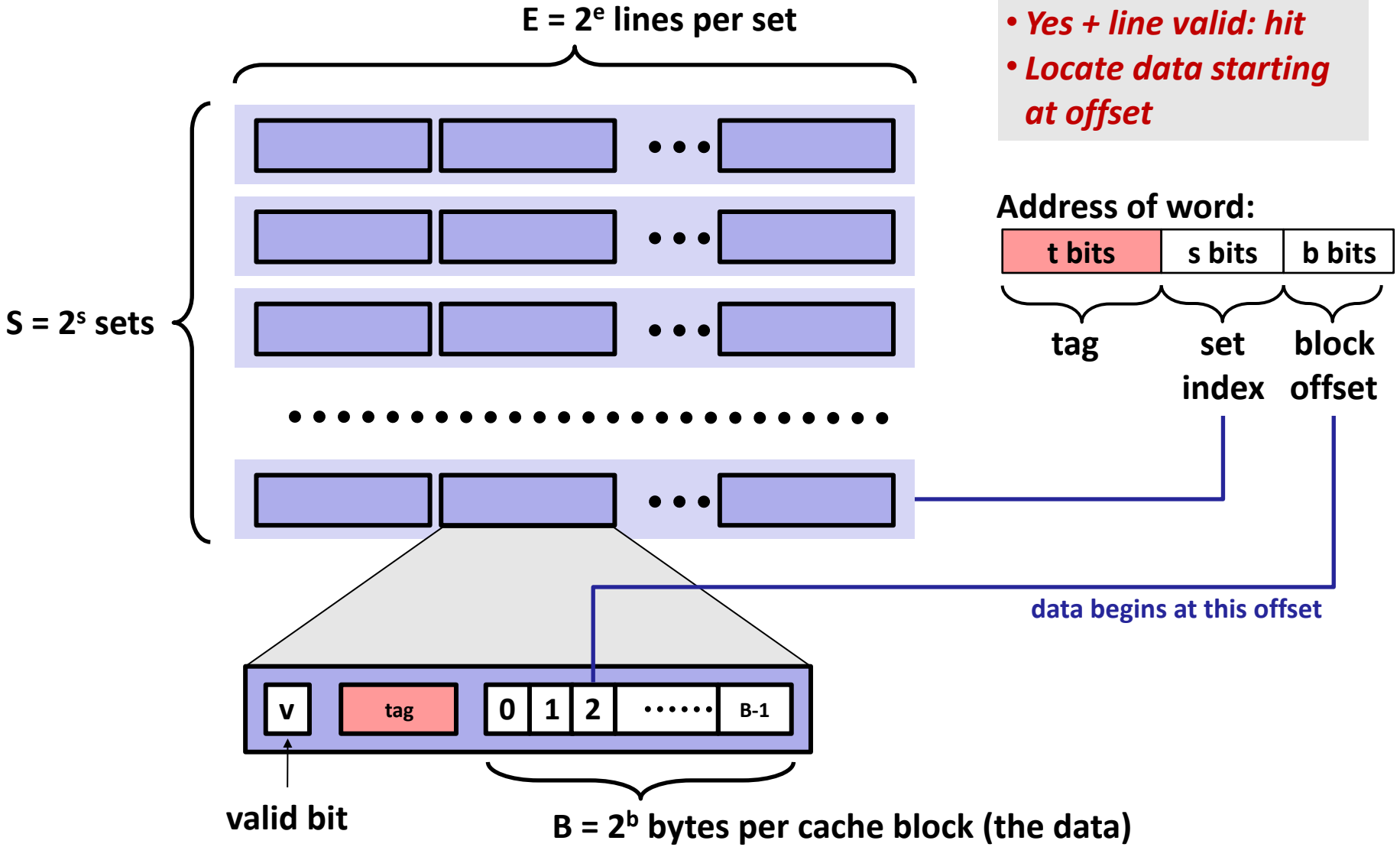
Cache Access

- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*



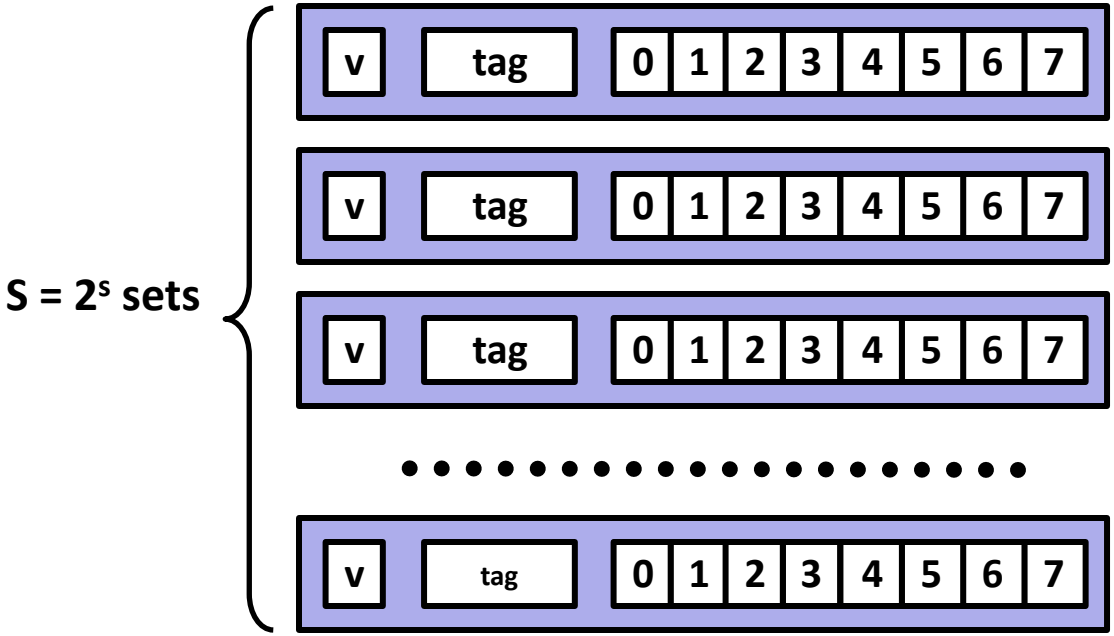
Cache Access

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



Example: Direct Mapped Cache

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

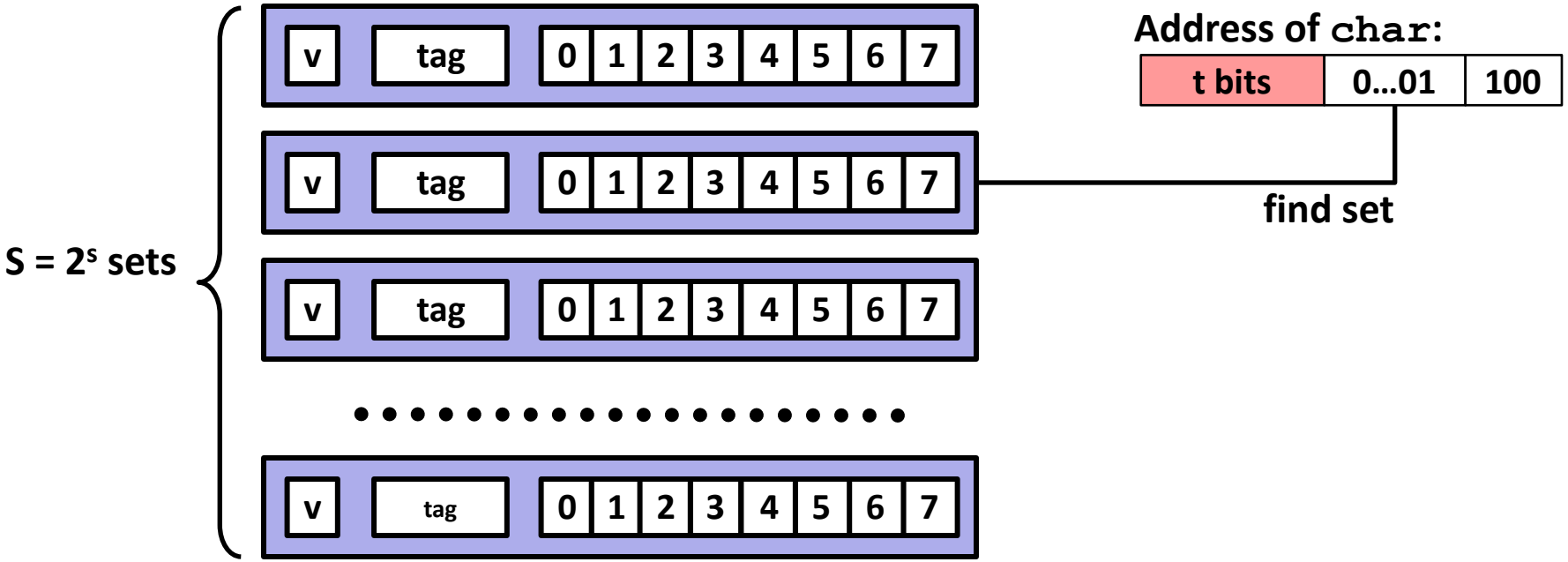


Address of `char`:

t bits	0…01	100

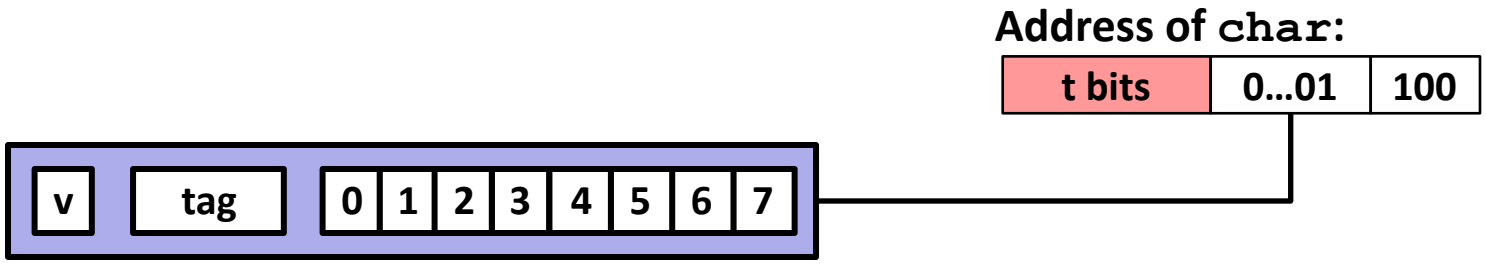
Example: Direct Mapped Cache

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



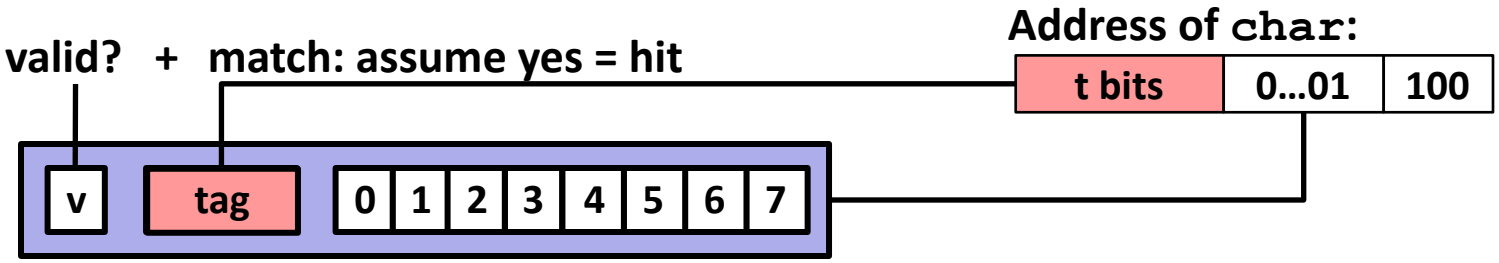
Example: Direct Mapped Cache

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



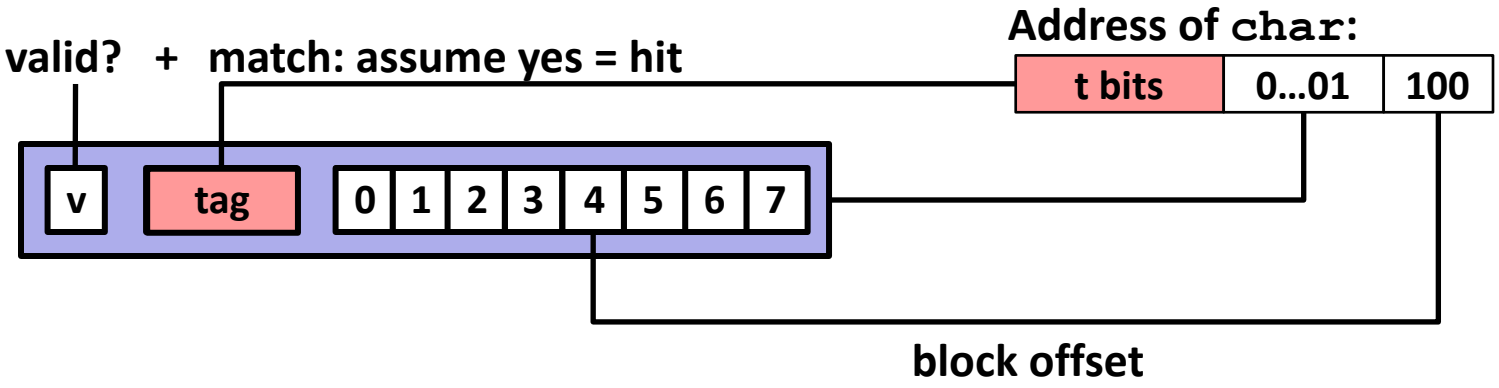
Example: Direct Mapped Cache

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



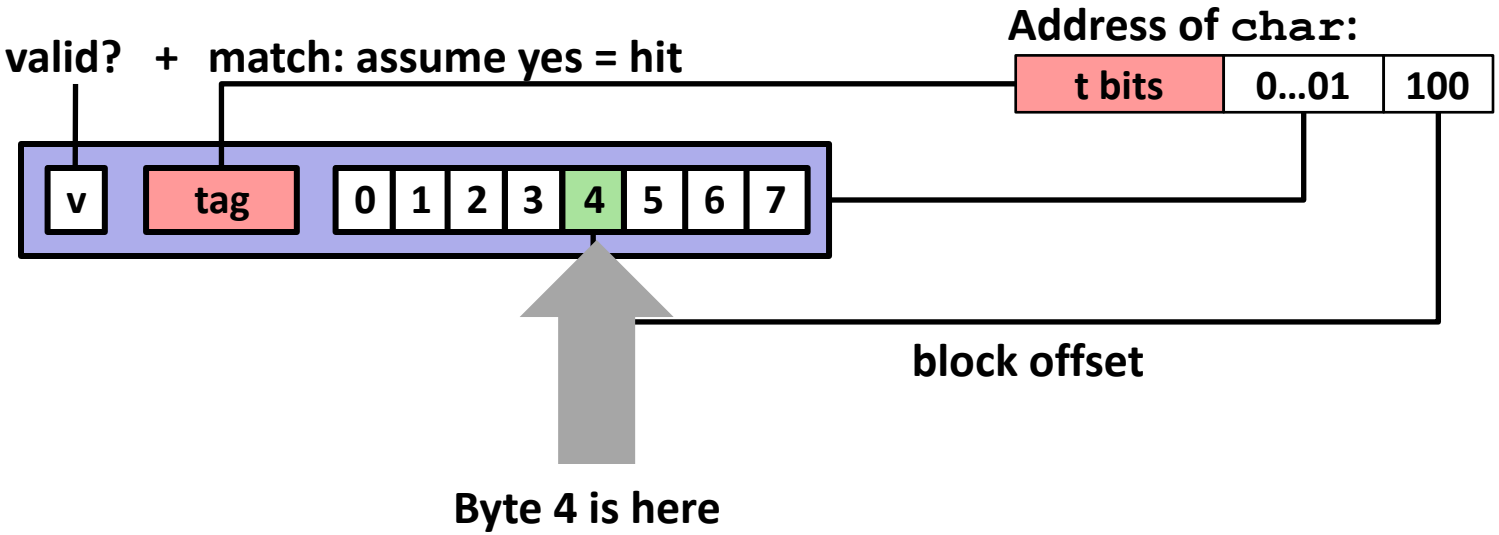
Example: Direct Mapped Cache

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



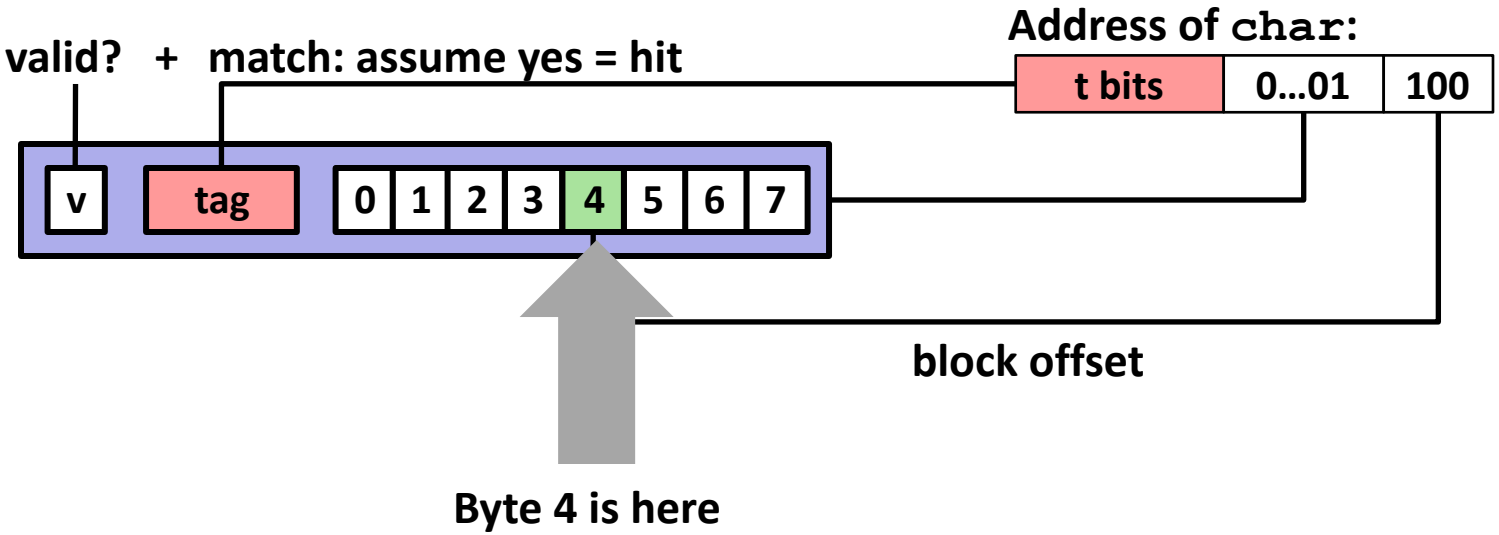
Example: Direct Mapped Cache

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



Example: Direct Mapped Cache

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



If tag doesn't match: old line is evicted and replaced

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0 **[0$\underline{00}$0$_2$],**
1 **[0$\underline{00}$1$_2$],**
7 **[0$\underline{11}$1$_2$],**
8 **[1$\underline{00}$0$_2$],**
0 **[0$\underline{00}$0$_2$]**

	v	Tag	Line
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	0	?	?
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3			

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	
0	**[0$\underline{00}$0$_2$]**	

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

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	

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

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	

	v	Tag	Line
Set 0	1	1	M[8-9]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

	v	Tag	Line
Set 0	1	1	M[8-9]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

Direct-Mapped Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
B=2 bytes/line, S=4 sets, E=1 line/set

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

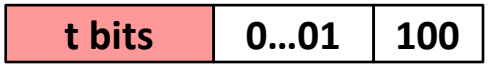
	v	Tag	Line
Set 0	1	0	M[0-1]
Set 1			
Set 2			
Set 3	1	0	M[6-7]

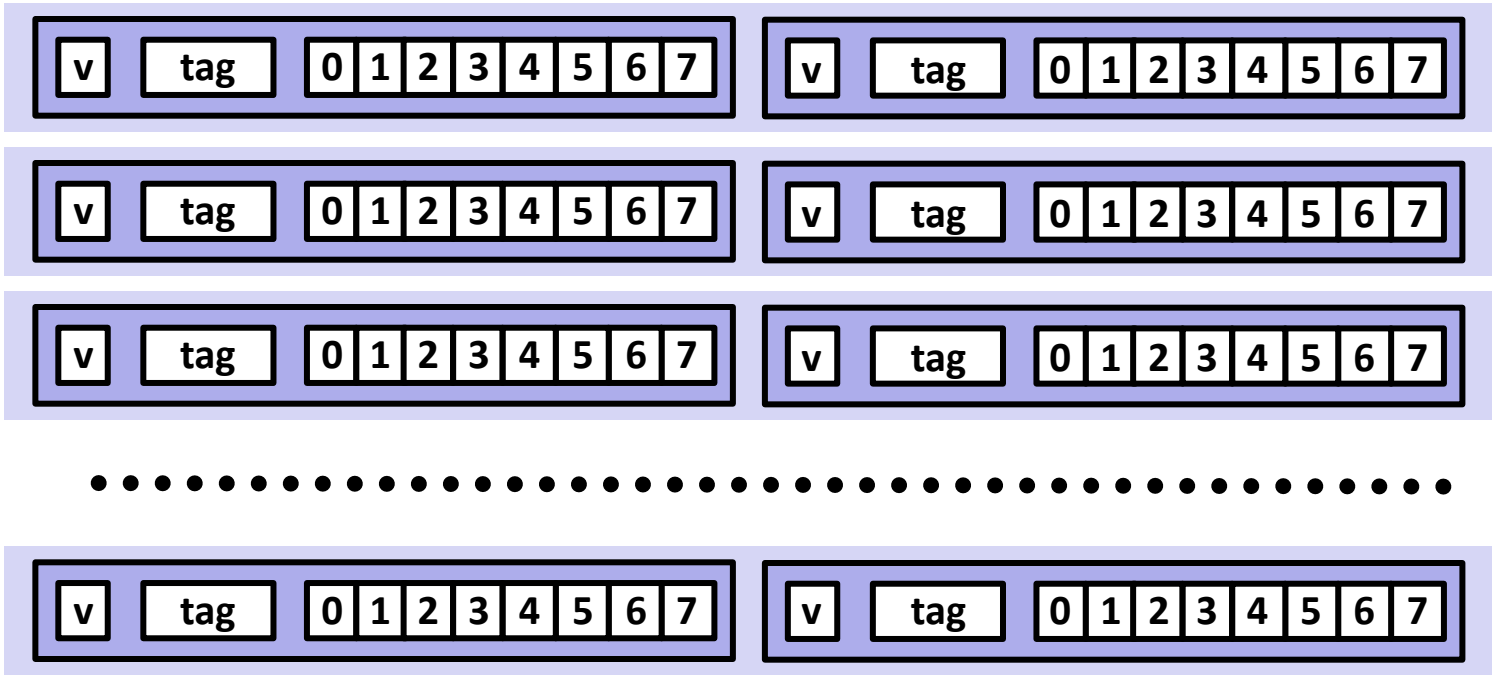
E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

Assume: cache block size 8 bytes

Address of `short int`:

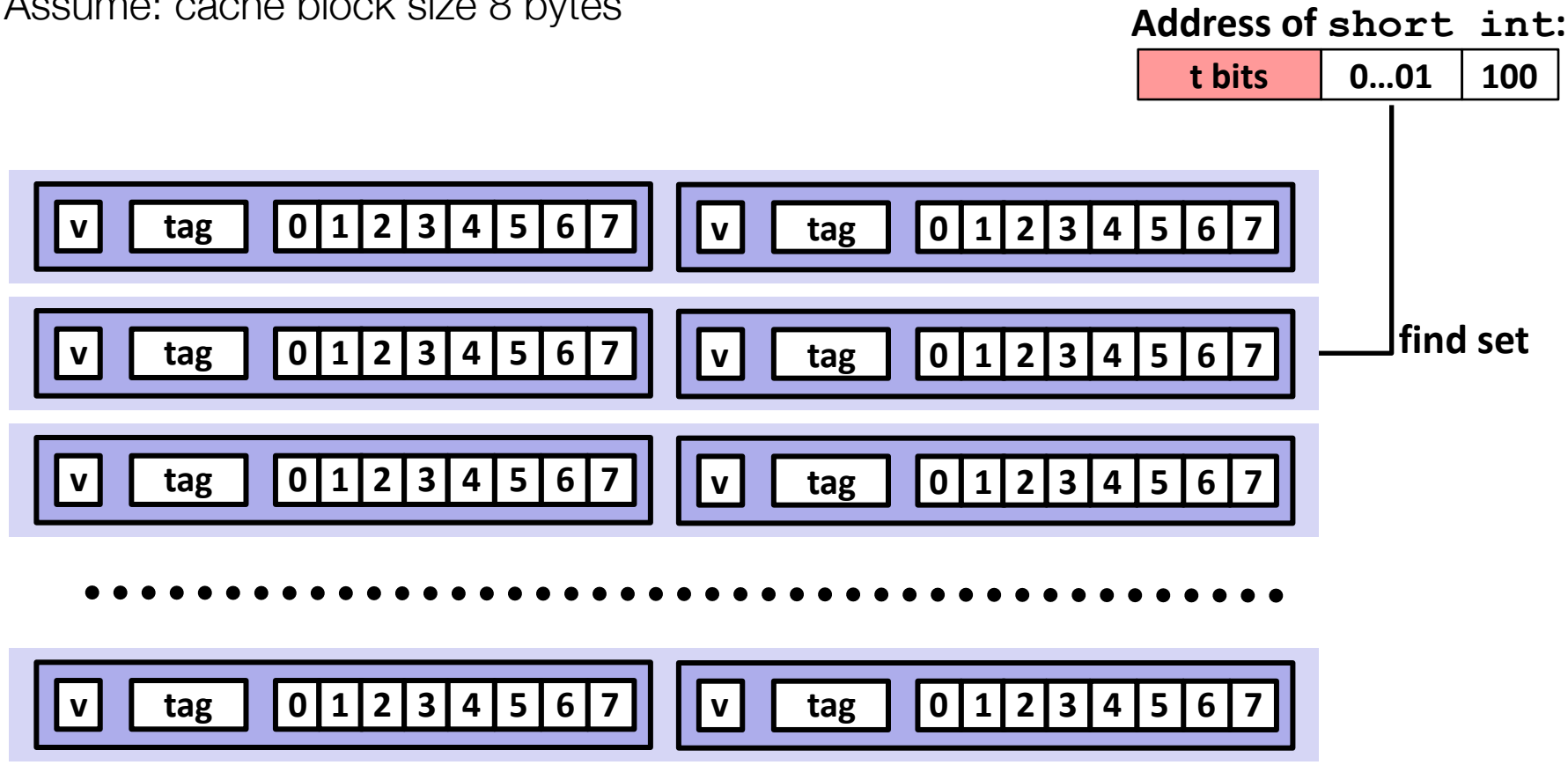




E-way Set Associative Cache (Here: E = 2)

E = 2: Two lines per set

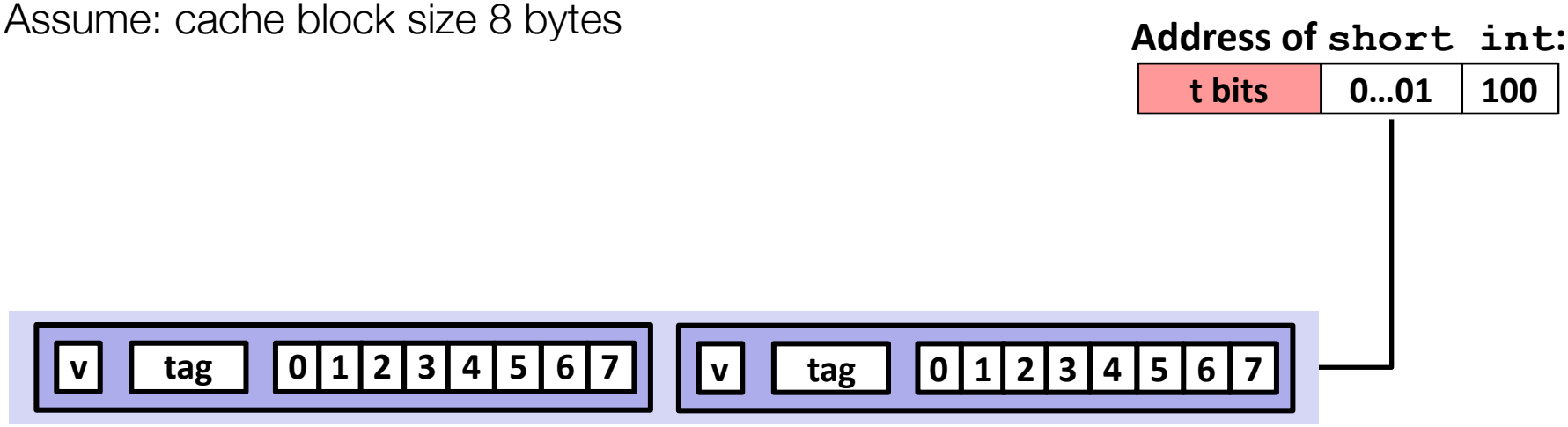
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

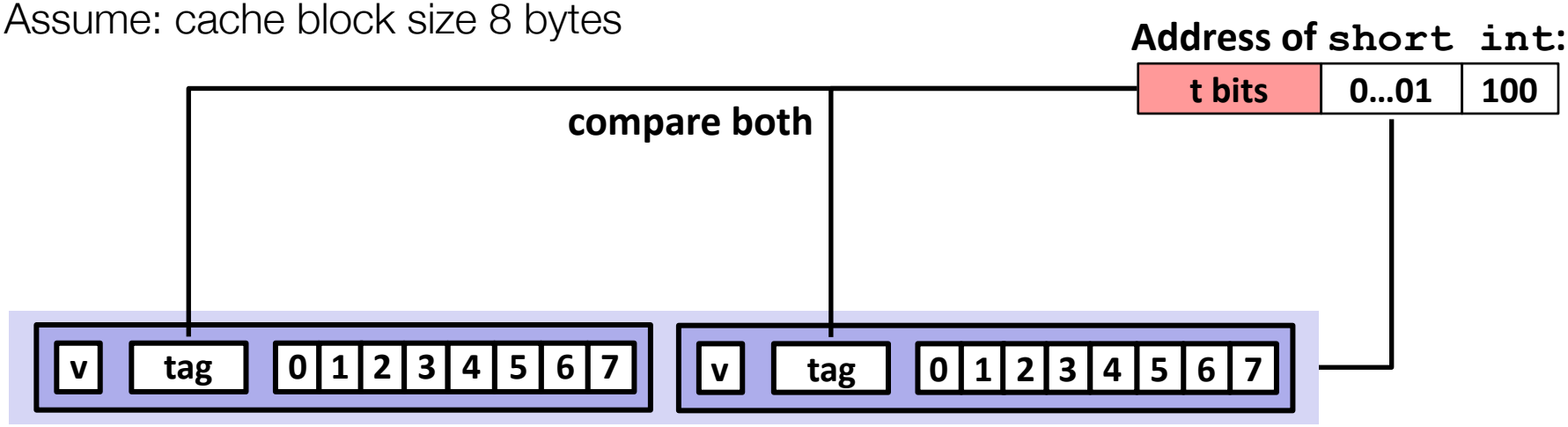
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

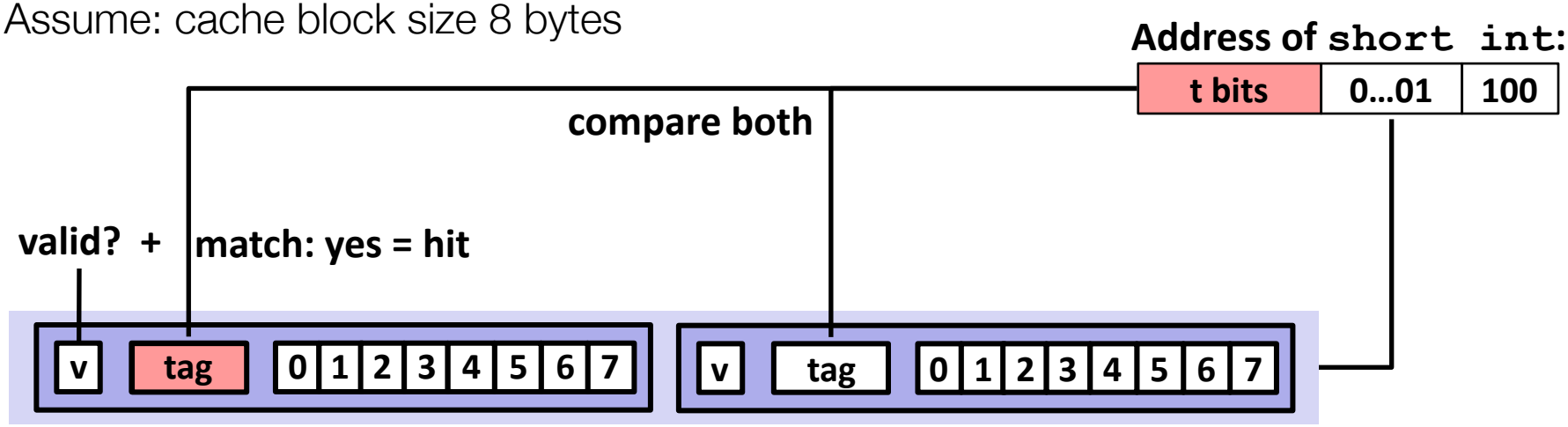
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

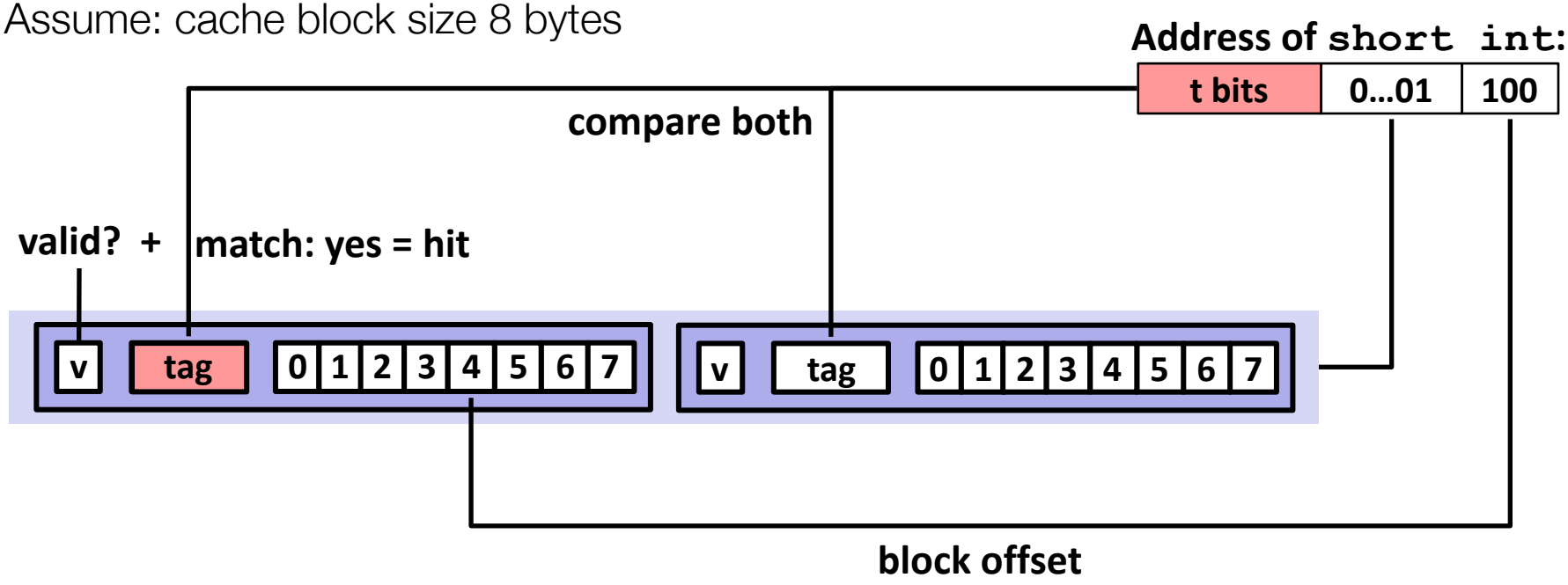
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

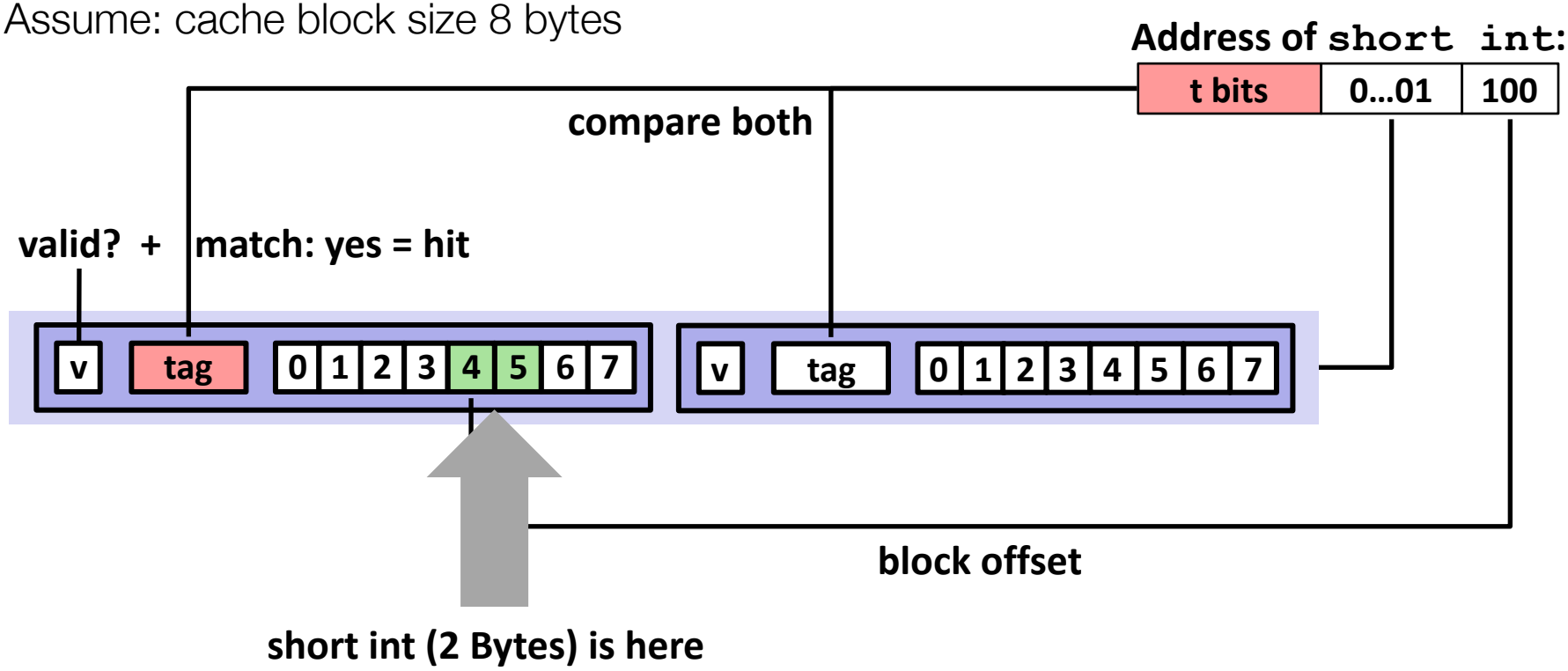
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

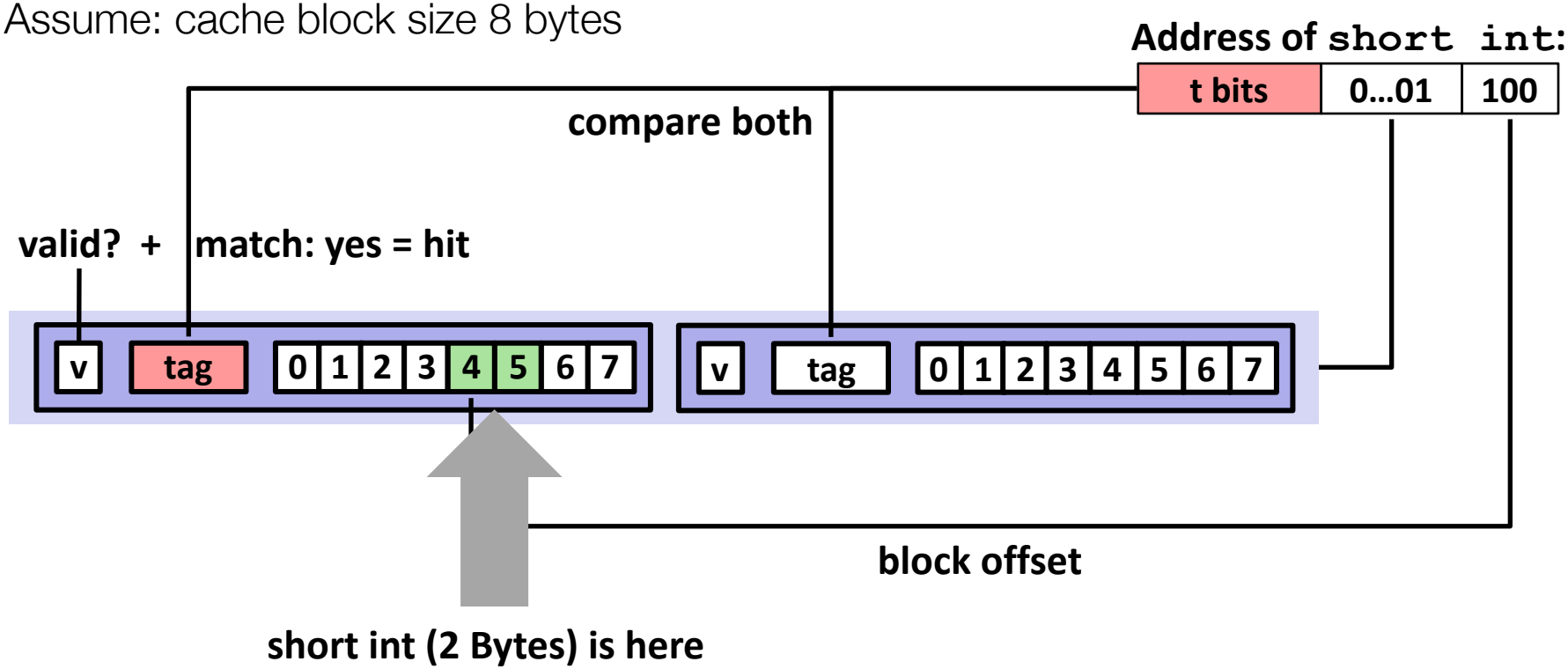
Assume: cache block size 8 bytes



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

E = 2: Two lines per set

Assume: cache block size 8 bytes



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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0 **[00$\underline{0}$0$_2$],** miss

1 **[00$\underline{0}$1$_2$],**

7 **[01$\underline{1}$1$_2$],**

8 **[10$\underline{0}$0$_2$],**

0 **[00$\underline{0}$0$_2$]**

	v	Tag	Block
Set 0	0	?	?
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

0 **[00$\underline{0}$0$_2$],** miss

1 **[00$\underline{0}$1$_2$],**

7 **[01$\underline{1}$1$_2$],**

8 **[10$\underline{0}$0$_2$],**

0 **[00$\underline{0}$0$_2$]**

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	0		
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

	v	Tag	Block
Set 0	1	00	M[0-1]
	0		
Set 1	1	01	M[6-7]
	0		

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

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

2-Way Set Associative Cache Simulation

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

4-bit address space, i.e., Memory = 16 bytes
S=2 sets, E=2 blocks/set

Address trace (reads, one byte per read):

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

Today

- Review: Cache memory organization and operation
- Performance impact of caches
  - Analytical Model
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

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.

Cache Performance Metrics

- Hit Time
  - Time to deliver a line in the cache to the processor
    - includes time to determine whether the line is in the cache
  - Typical numbers:
    - 1~4 clock cycle for L1
    - 5~10 clock cycles for L2

Cache Performance Metrics

- 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

Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate

Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles

Let’s think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles

    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles

    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles
- **Think of it as reducing the miss rate from 3% to 1% (3X improvement) rather than improving hit rate**

Let's think about those numbers

- Huge difference between a hit and a miss
  - Could be 100x, if just L1 and main memory
- Compare 97% hit rate with 99% hit rate
  - Assume:
    cache hit time of 1 cycle
    miss penalty of 100 cycles
  - Average access time:

    97% hit rate: 1 cycle + 0.03 * 100 cycles = 4 cycles

    99% hit rate: 1 cycle + 0.01 * 100 cycles = 2 cycles
- **Think of it as reducing the miss rate from 3% to 1% (3X improvement) rather than improving hit rate**
- Improving hit rate by even a little bit helps overall speed a lot

Writing Cache Friendly Code

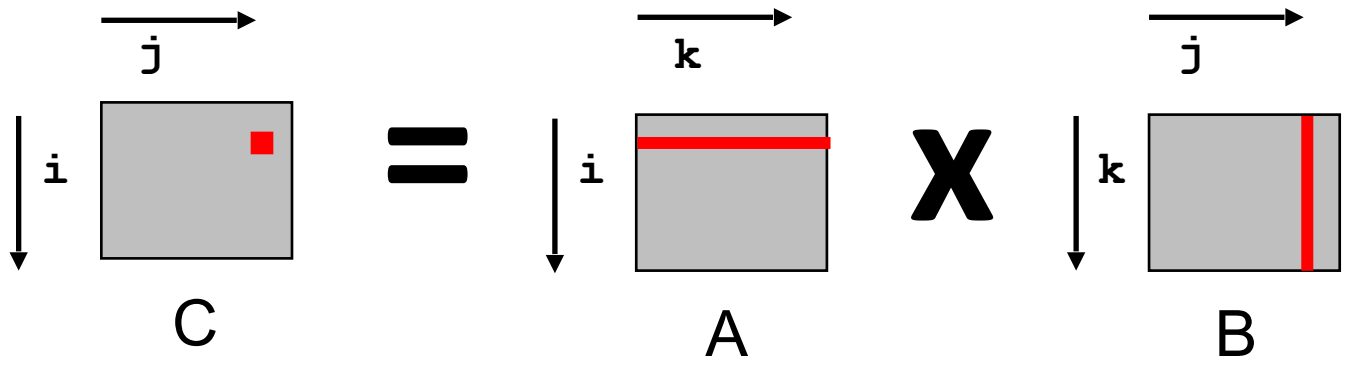
- Make the common case go fast
  - Inner loops get executed most often. So focus on those

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

Today

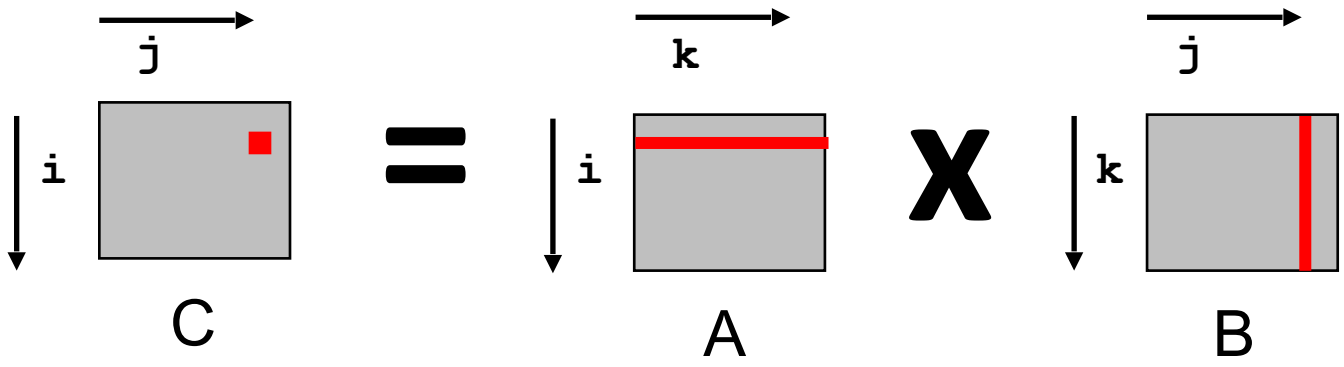
- Review: Cache memory organization and operation
- Performance impact of caches
  - Analytical Model
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Matrix Multiplication Example



Matrix Multiplication Example

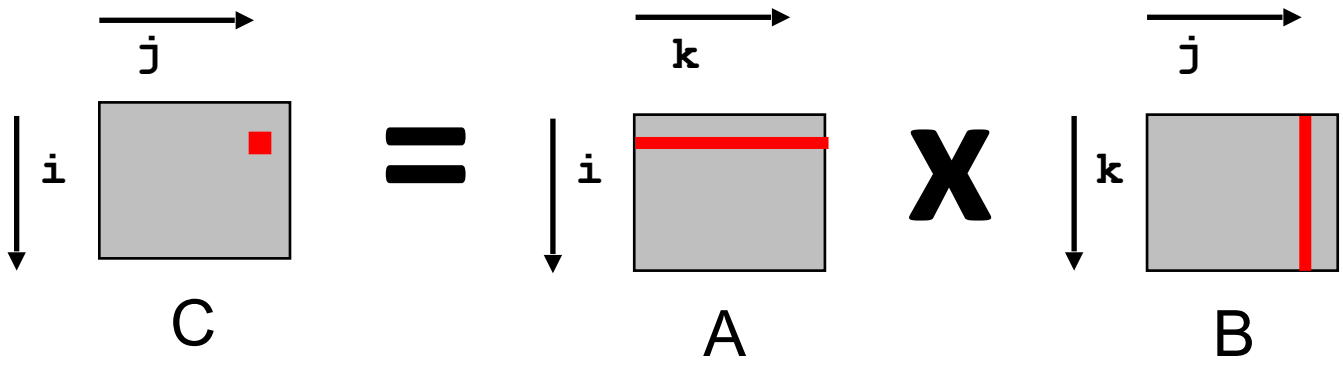
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```



Matrix Multiplication Example

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

Variable ***sum*** held in register

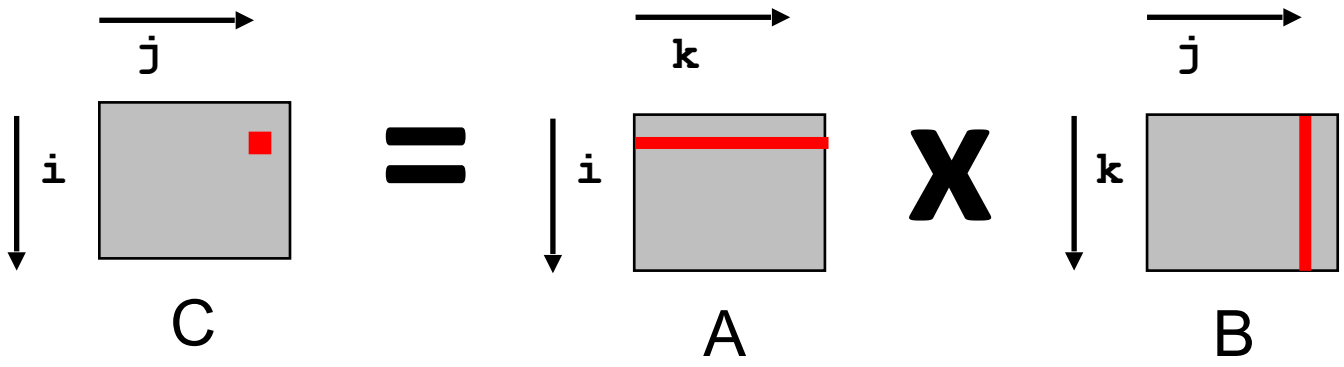


Matrix Multiplication Example

- Multiply N x N matrices
- Matrix elements are doubles (8 bytes)
- $O(N^3)$ total operations

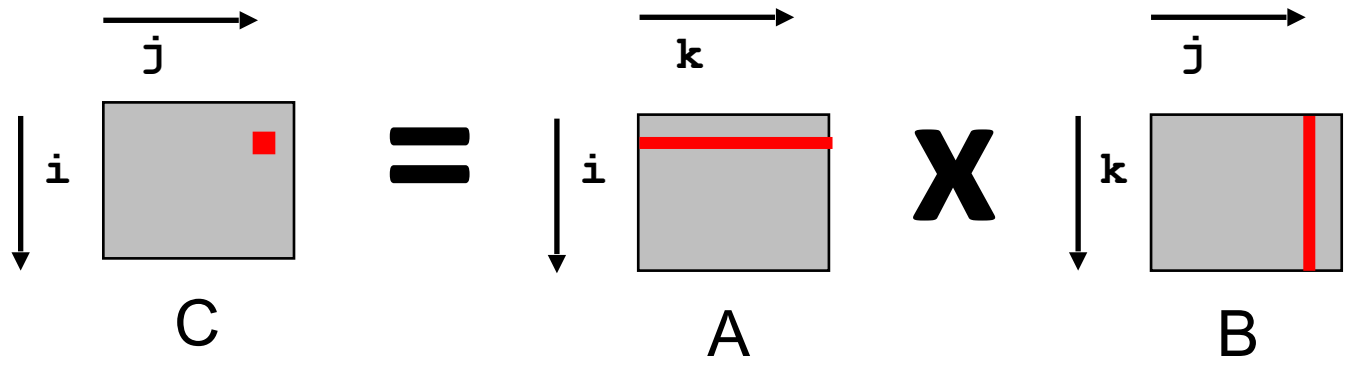
```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

Variable ***sum*** held in register



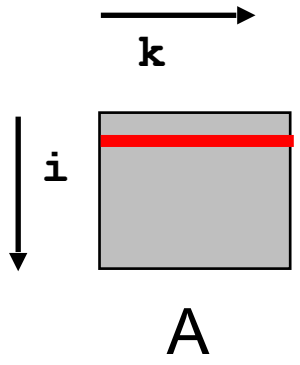
Miss Rate Analysis for Matrix Multiply

- 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



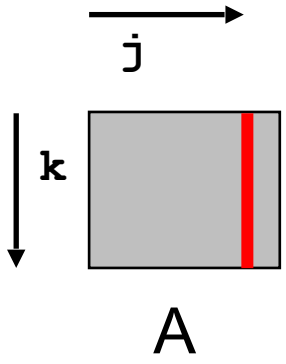
Layout of C Arrays in Memory (review)

- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - accesses successive elements
  - cache line size (32) > size of an element (8 bytes), exploiting spatial locality!
    - miss rate = 8 / 32 = 25%



Layout of C Arrays in Memory (review)

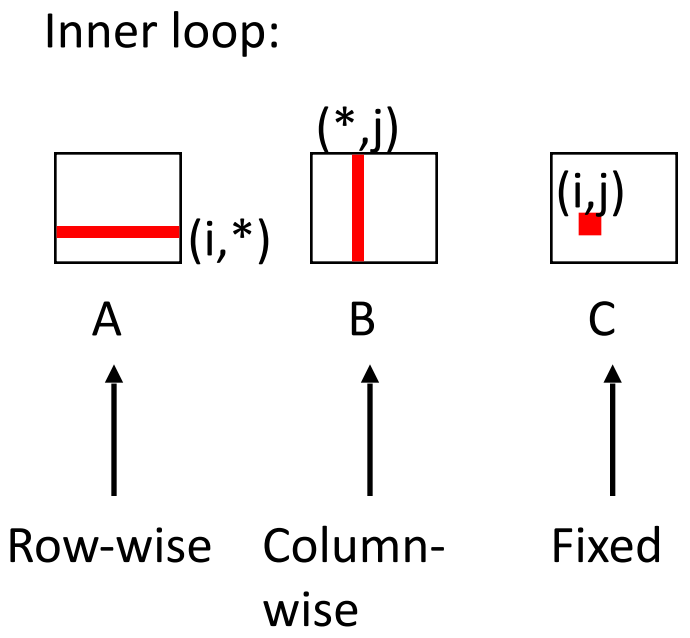
- C arrays allocated in row-major order
  - each row in contiguous memory locations
- Stepping through rows in one column:
  - `for (i = 0; j < n; j++)`
    `sum += a[i][0];`
  - accesses distant elements
  - no spatial locality!
    - miss rate = 1 (i.e. 100%)



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



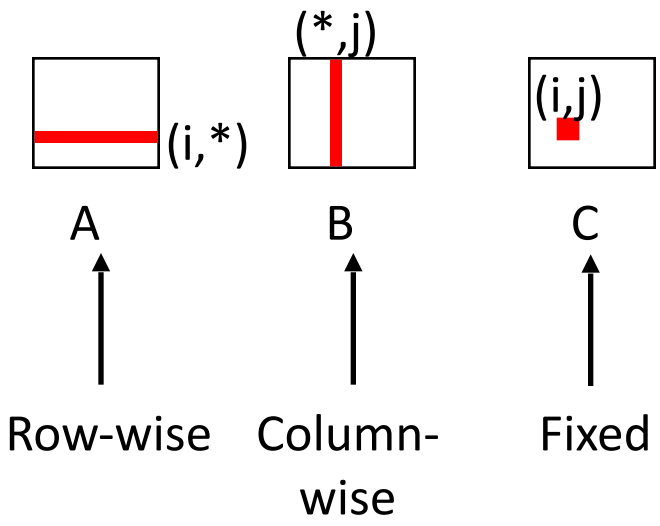
Misses per inner loop iteration:

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
  }
}
```

matmult/mm.c



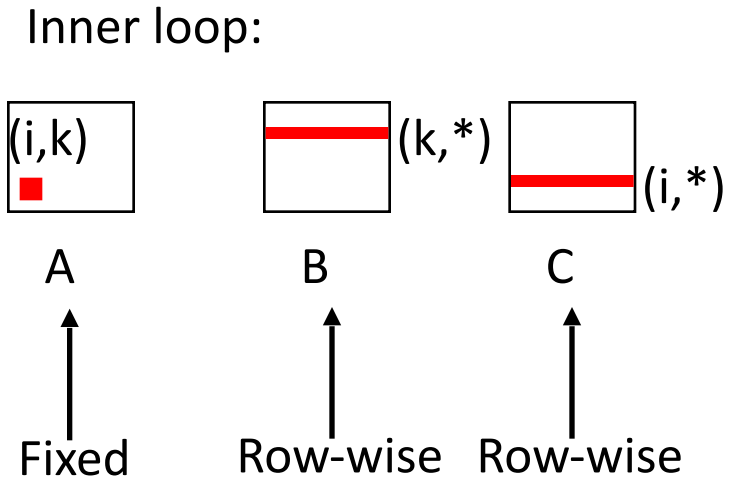
Misses per inner loop iteration:

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];
  }
}
```

matmult/mm.c



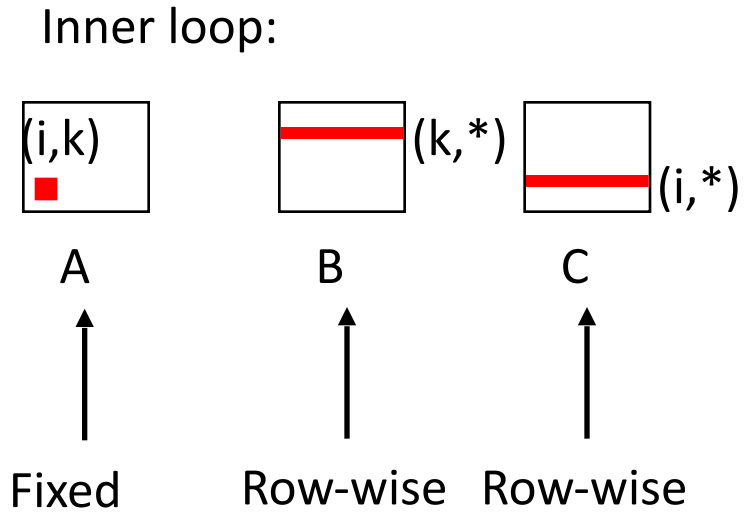
Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

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

matmult/mm.c



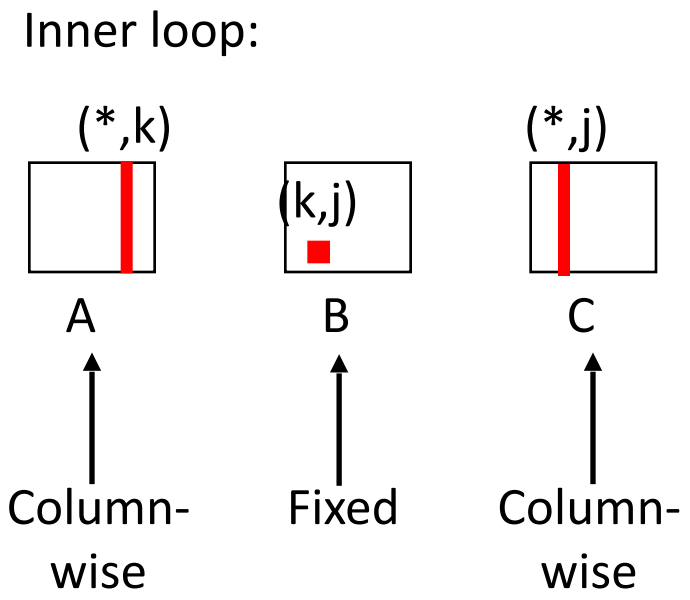
Misses per inner loop iteration:

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;
  }
}
```

matmult/mm.c



Misses per inner loop iteration:

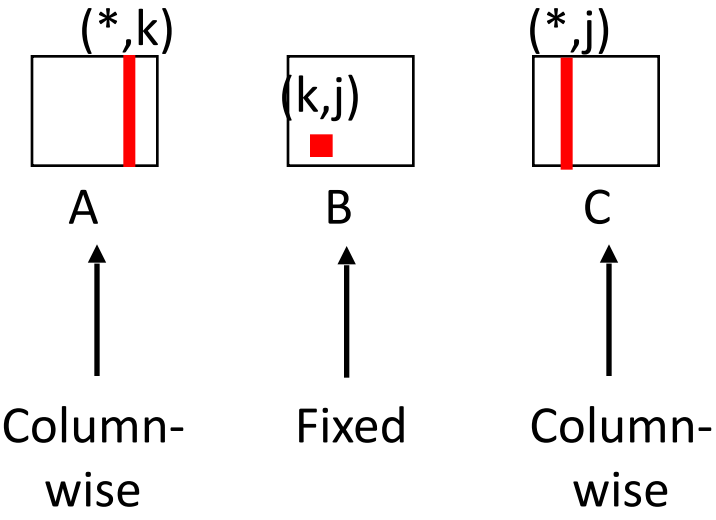
A	B	C
1.0	0.0	1.0

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;
  }
}
```

matmult/mm.c

Inner loop:



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;
 }
}
```

ijk (& jik):

- 2 loads, 0 stores
- 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):

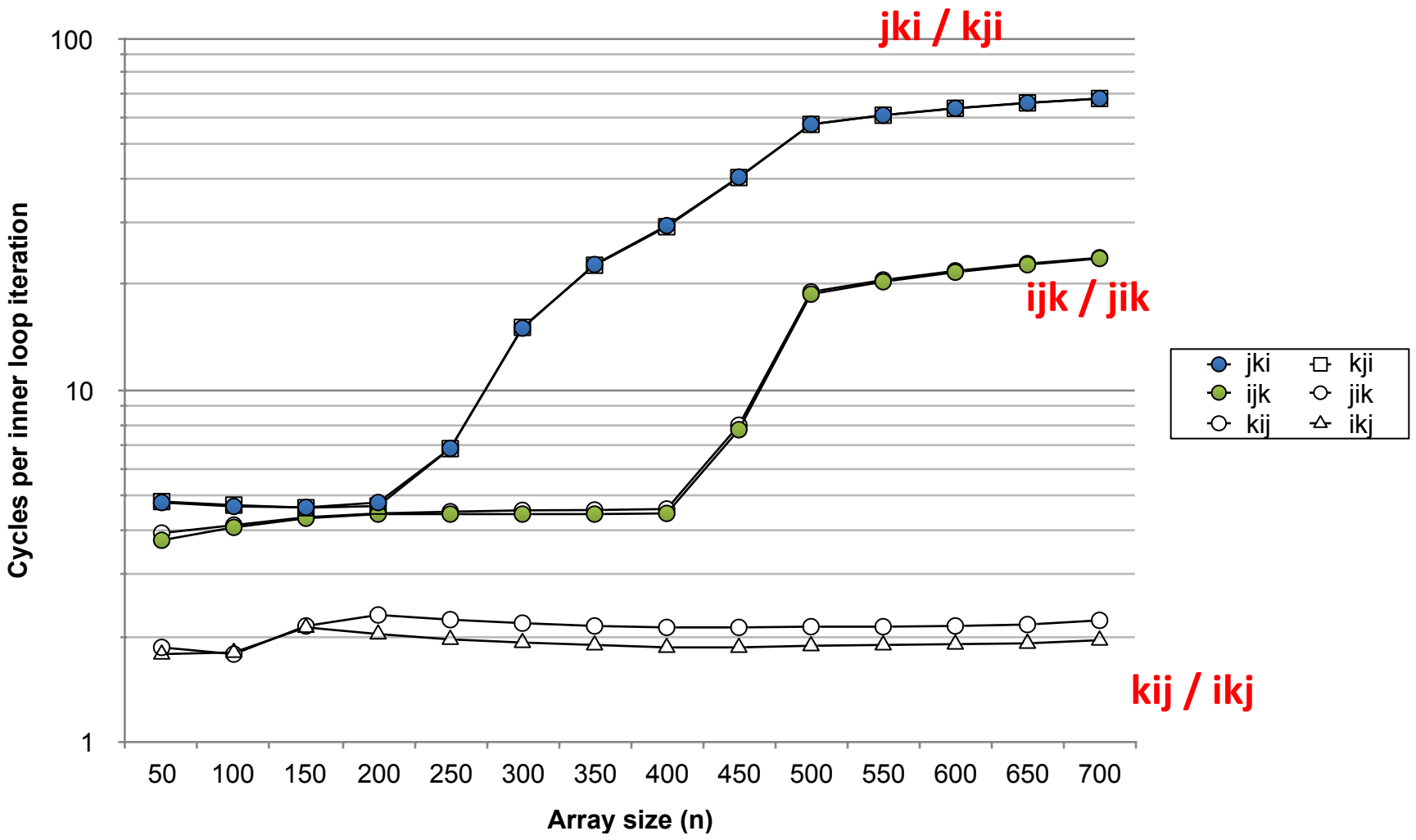
- 2 loads, 1 store
- misses/iter = **0.5**

```
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
- misses/iter = **2.0**

Core i7 Matrix Multiply Performance

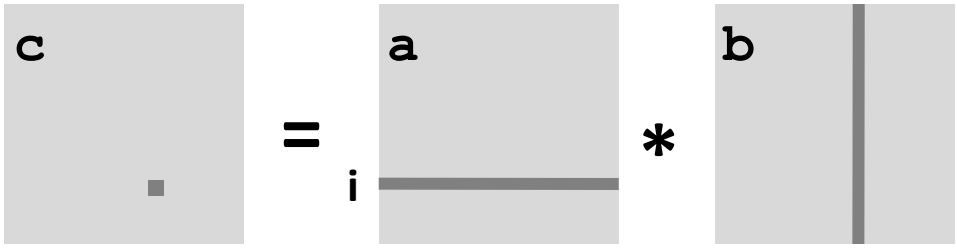


Today

- Review: Cache memory organization and operation
- Performance impact of caches
  - Analytical Model
  - Rearranging loops to improve spatial locality
  - Using blocking to improve temporal locality

Example: Matrix Multiplication

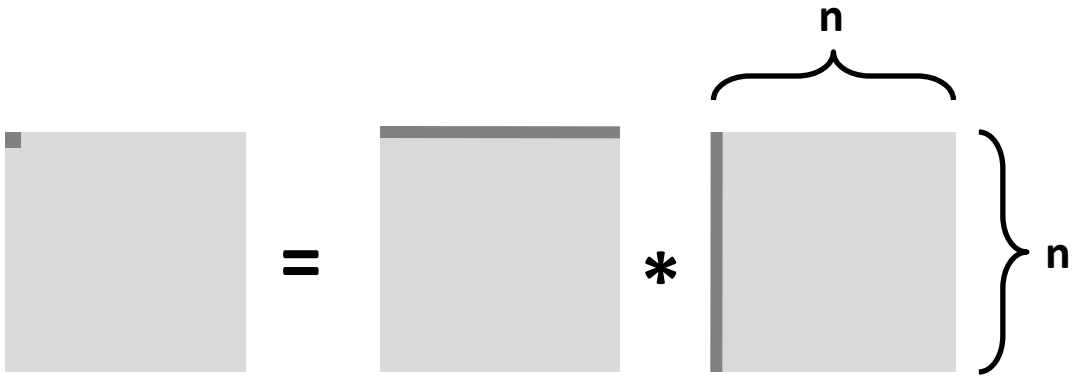
```
c = (double *) calloc(sizeof(double), n*n);

/* 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)

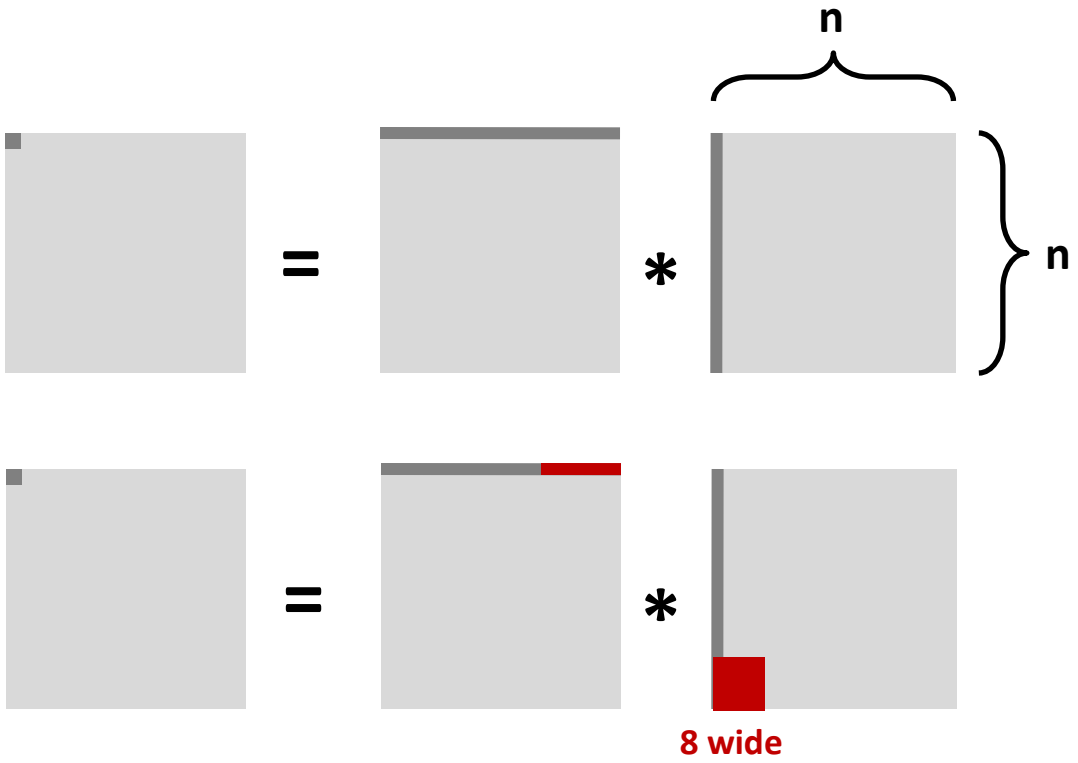
- First iteration:



Cache Miss Analysis

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

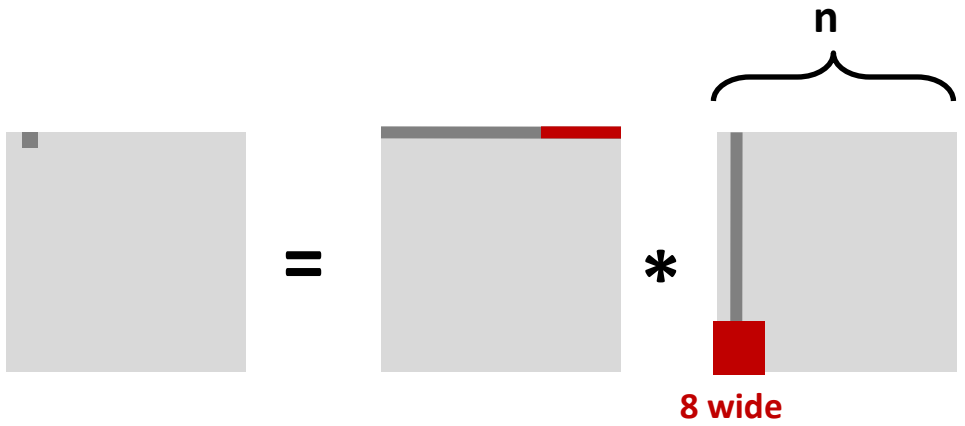
- First iteration:
  - n/8 + n = 9n/8 misses



Cache Miss Analysis

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

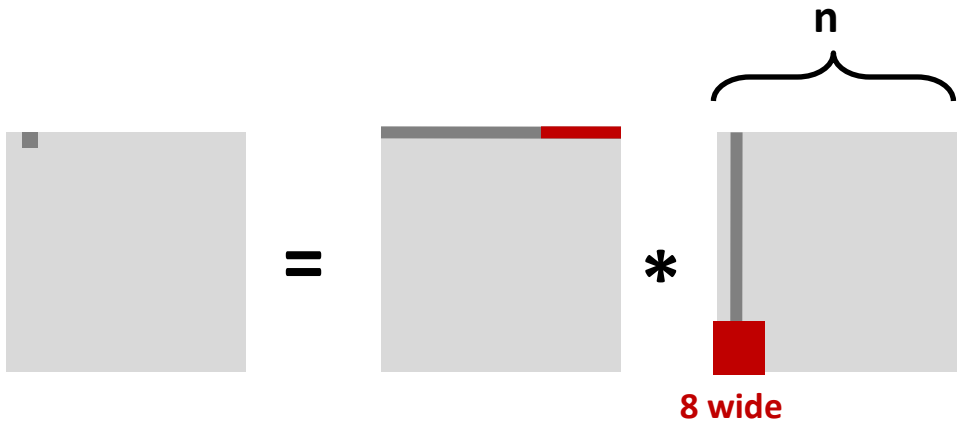
- Second iteration:



Cache Miss Analysis

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

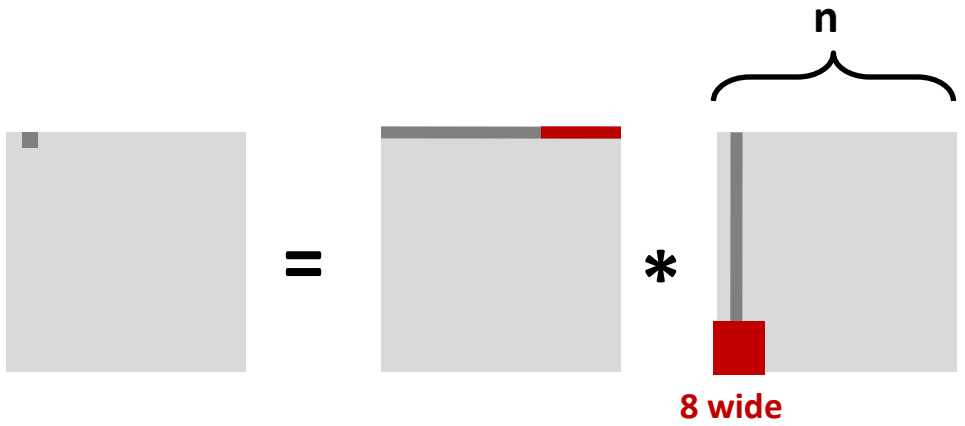
- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses



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

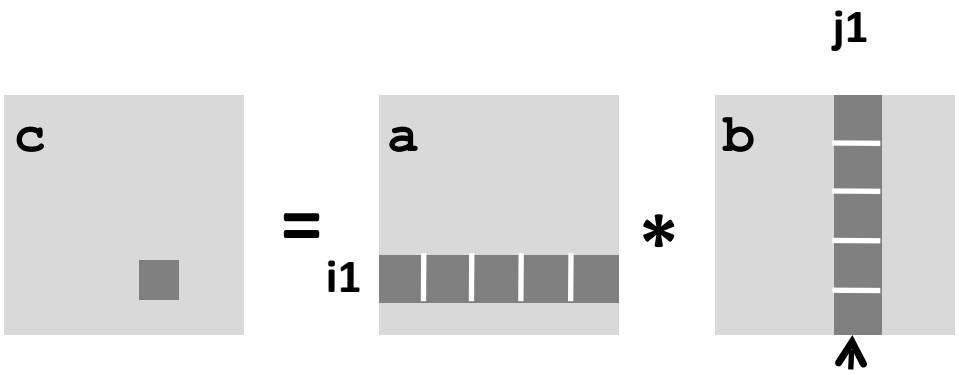


- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

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];
}
```

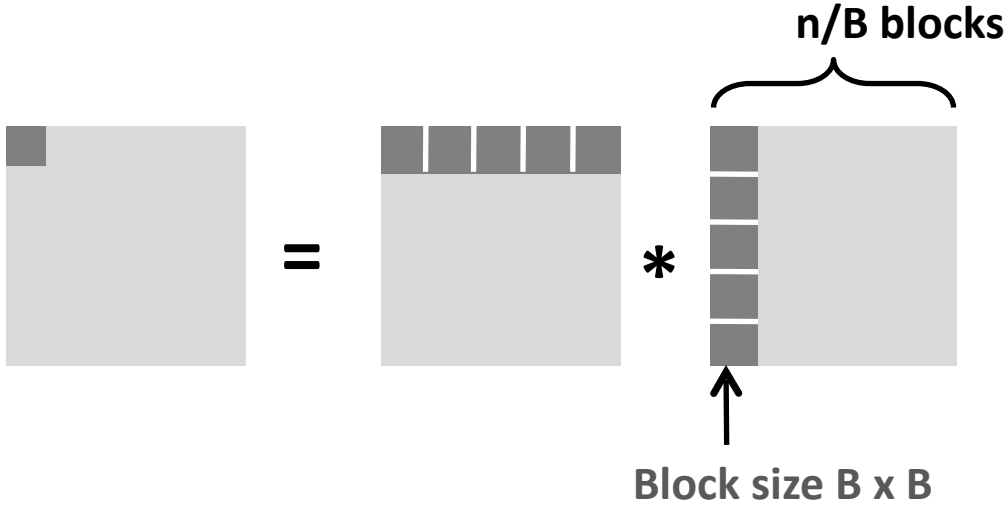
matmult/bmm.c



Cache Miss Analysis

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

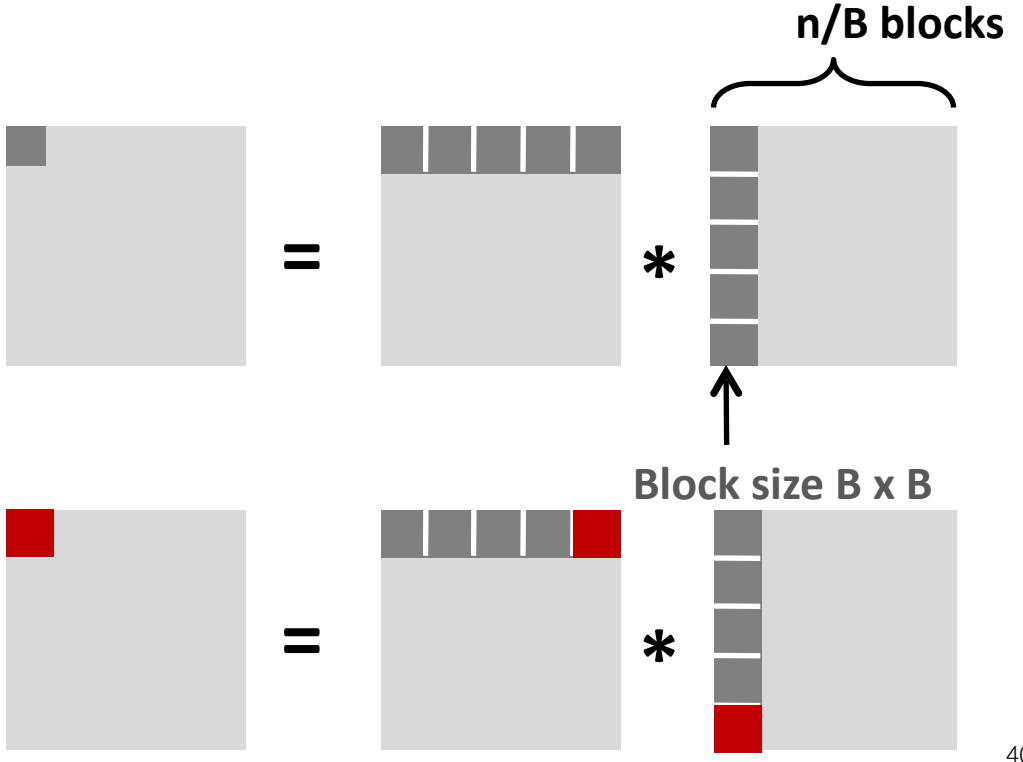
- First (block) iteration:



Cache Miss Analysis

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

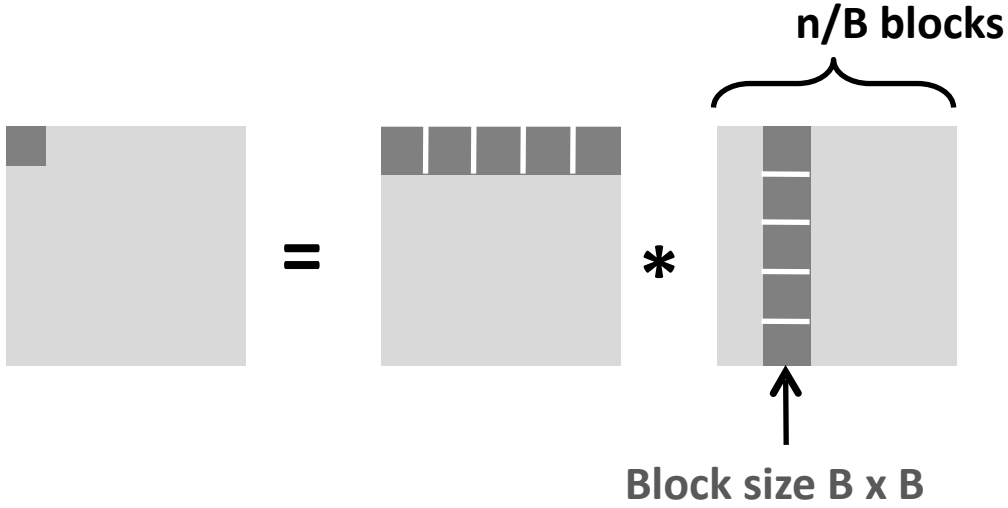
- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$



Cache Miss Analysis

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

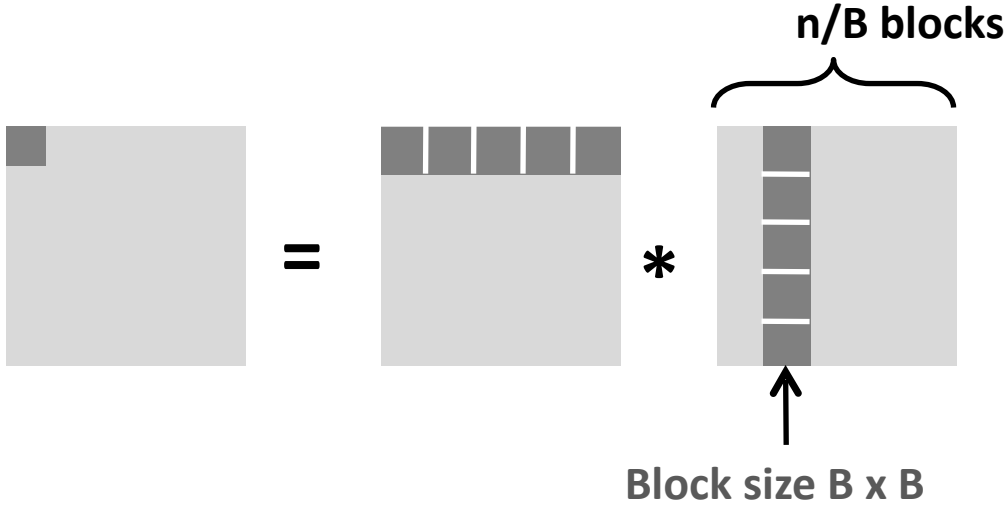
- Second (block) iteration:



Cache Miss Analysis

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

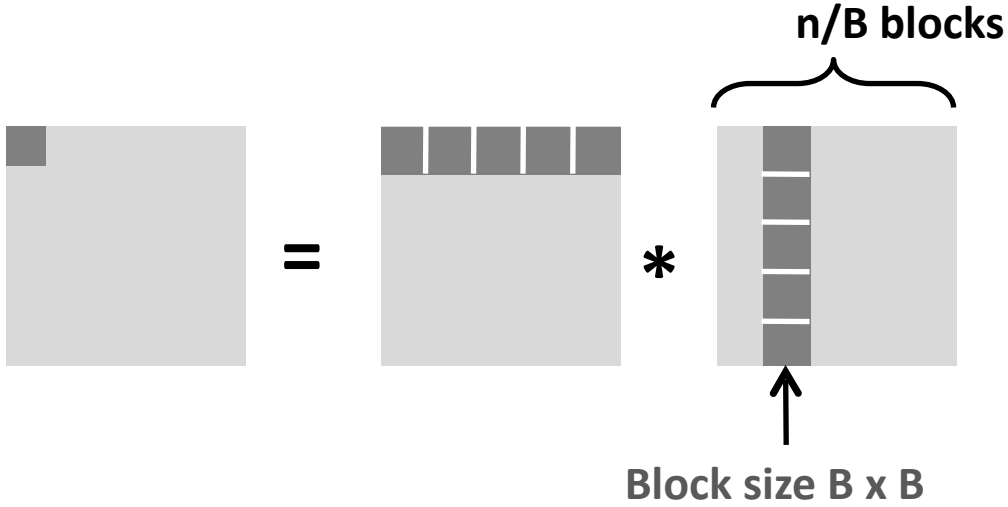
- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$



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

Blocking Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

Blocking Summary

- No blocking: (9/8) * n^3
- Blocking: 1/(4B) * n^3

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

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

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 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.