

Jin-Soo Kim  
([jinsoo.kim@snu.ac.kr](mailto:jinsoo.kim@snu.ac.kr))

Systems Software &  
Architecture Lab.  
Seoul National University

Fall 2020

# Cache Optimization

Chap. 5.4



# Cache Performance

- Components of CPU time
  - Program execution cycles include cache hit time
  - Memory stall cycles: mainly from cache misses
- With simplifying assumptions:

*Memory stall cycles*

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

# Cache Performance: Example

- **Given**
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
  - Miss penalty = 100 cycles
  - Base CPI (ideal cache) = 2
  - Load & stores are 36% of instructions
- **Miss cycles per instruction**
  - I-cache:  $0.02 \times 100 = 2$
  - D-cache:  $0.36 \times 0.04 \times 100 = 1.44$
- **Actual CPI =  $2 + 2 + 1.44 = 5.44$** 
  - Ideal CPU is  $5.44/2 = 2.72$  times faster

# Average Access Time

- Hit time is also important for performance
- Average memory access time (AMAT):

$$AMAT = Hit\ time + Miss\ rate \times Miss\ penalty$$

- Example
  - CPU with 1ns clock
  - Hit time = 1 cycle,
  - Miss penalty = 20 cycles
  - I-cache miss rate = 5%
  - $AMAT = 1 + 0.05 \times 20 = 2ns$  (2 cycles per instruction)

# Cache Performance Summary

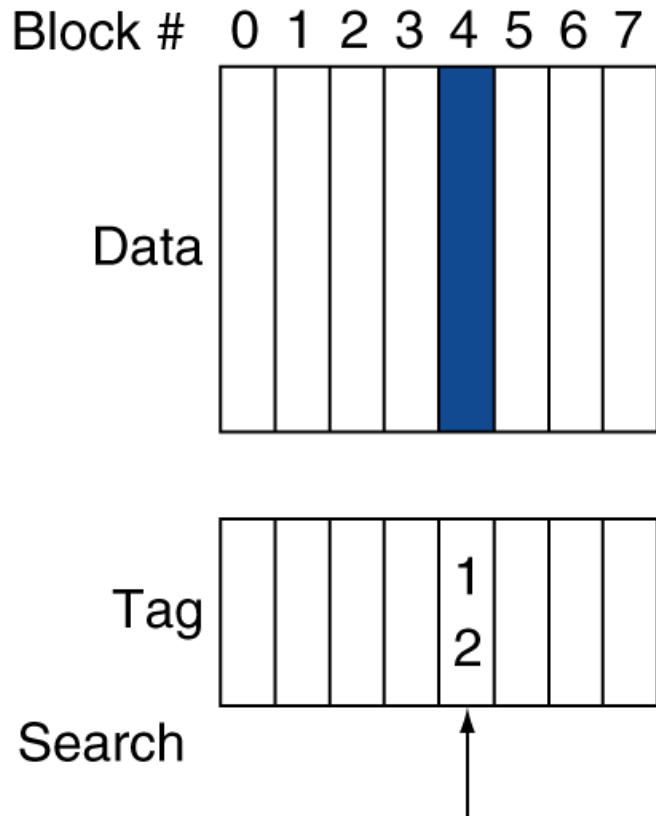
- **When CPU performance increased**
  - Miss penalty becomes more significant
- **Decreasing base CPI**
  - Greater proportion of time spent on memory stalls
- **Increasing clock rate**
  - Memory stalls account for more CPU cycles
- **Can't neglect cache behavior when evaluating system performance**

# Associative Caches

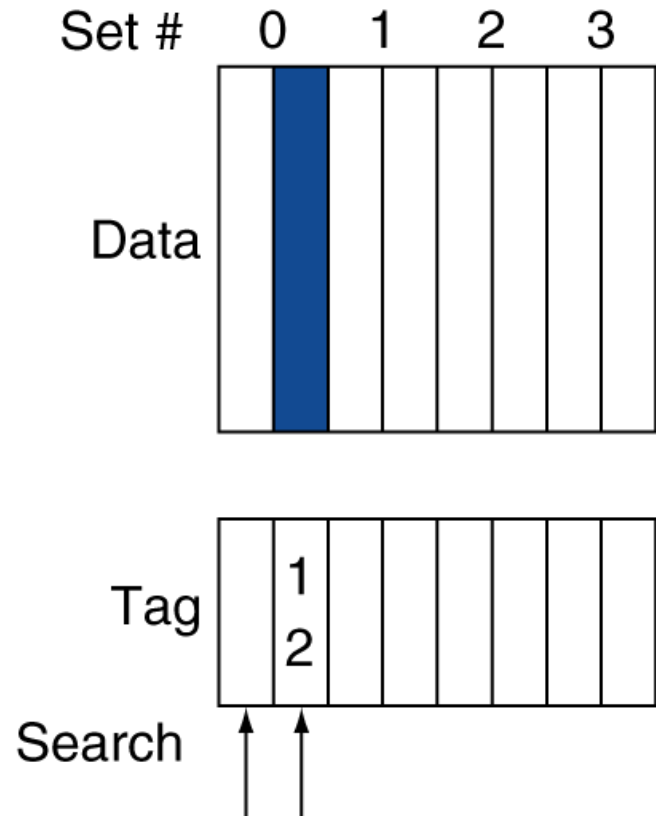
- **Fully associative**
  - Allows a given block to go in any cache entry
  - Requires all entries to be searched at once
  - Comparator per entry (expensive)
  
- ***n*-way set associative**
  - Each set contains *n* entries
  - Block number determines which set:  $Block\_number \bmod \#Sets\_in\_cache$
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

# Associative Cache Example

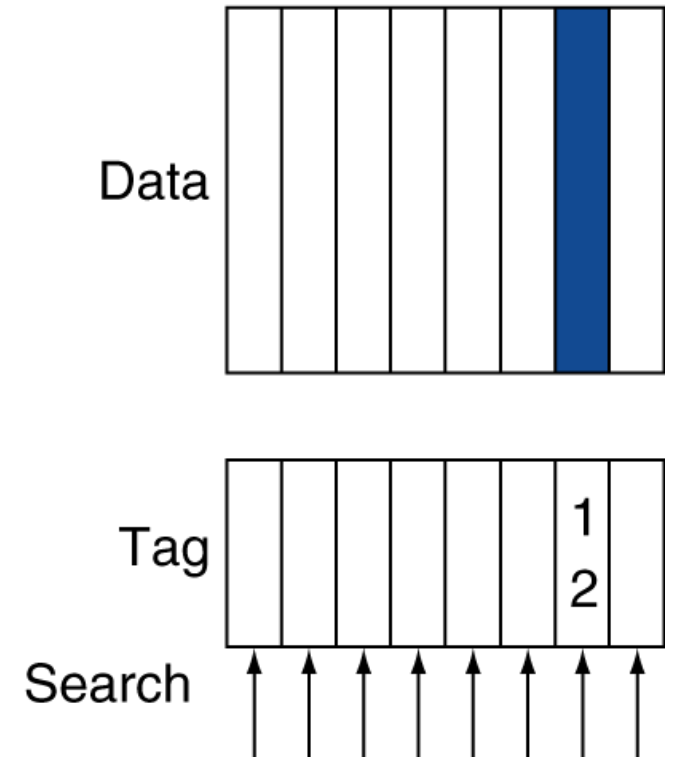
**Direct mapped**



**Set associative**



**Fully associative**



# Spectrum of Associativity

- For a cache with 8 entries

## One-way set associative (direct mapped)

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

## Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

## Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

## Eight-way set associative (fully associative)

Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data



# Associativity Example (I)

- Compare 4-block caches
  - Direct mapped, 2-way set associative, fully associative
  - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped

Block address	Cache index	Hit / Miss	Cache content after access			
			0	1	2	3
0	$0 \% 4 = 0$	Miss	Mem[0]			
8	$8 \% 4 = 0$	Miss	Mem[8]			
0	$0 \% 4 = 0$	Miss	Mem[0]			
6	$6 \% 4 = 2$	Miss	Mem[0]		Mem[6]	
8	$8 \% 4 = 0$	Miss	Mem[8]			

# Associativity Example (2)

- 2-way set associative

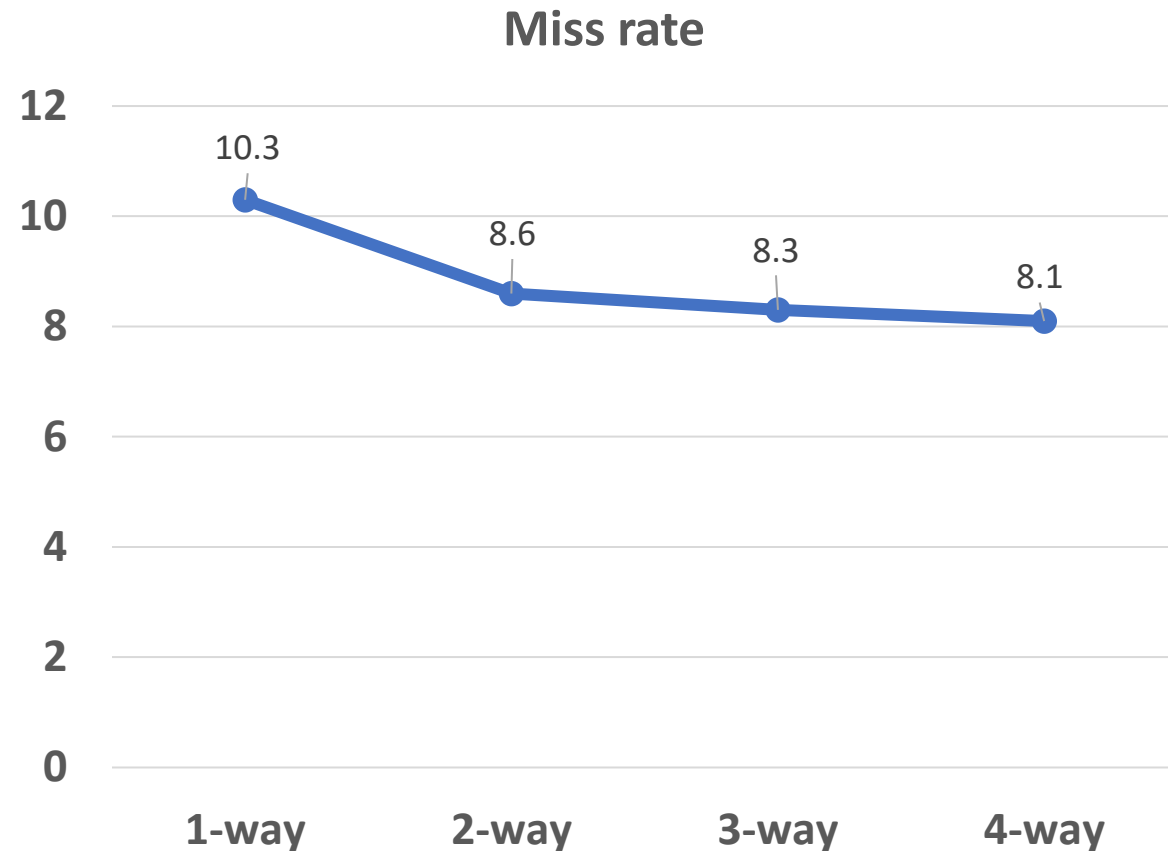
Block address	Cache index	Hit / Miss	Cache content after access			
			Set 0		Set 1	
0	$0 \% 2 = 0$	Miss	Mem[0]			
8	$8 \% 2 = 0$	Miss	Mem[0]	Mem[8]		
0	$0 \% 2 = 0$	Hit	Mem[0]	Mem[8]		
6	$6 \% 2 = 0$	Miss	Mem[0]	Mem[6]		
8	$8 \% 2 = 0$	Miss	Mem[8]	Mem[6]		

- Fully associative

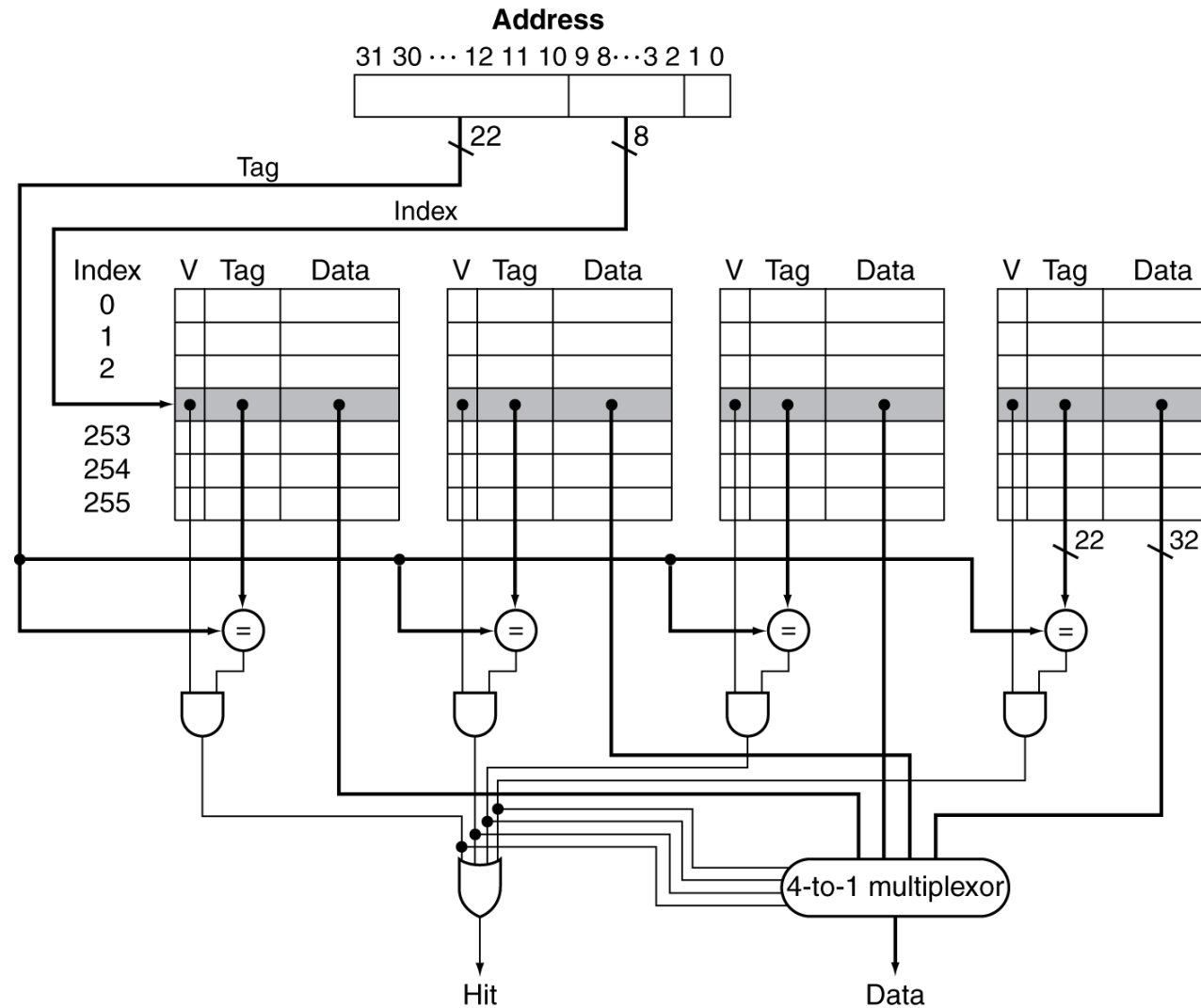
Block address	Cache index	Hit / Miss	Cache content after access			
0	-	Miss	Mem[0]			
8	-	Miss	Mem[0]	Mem[8]		
0	-	Hit	Mem[0]	Mem[8]		
6	-	Miss	Mem[0]	Mem[8]	Mem[6]	
8	-	Hit	Mem[0]	Mem[8]	Mem[6]	

# How Much Associativity?

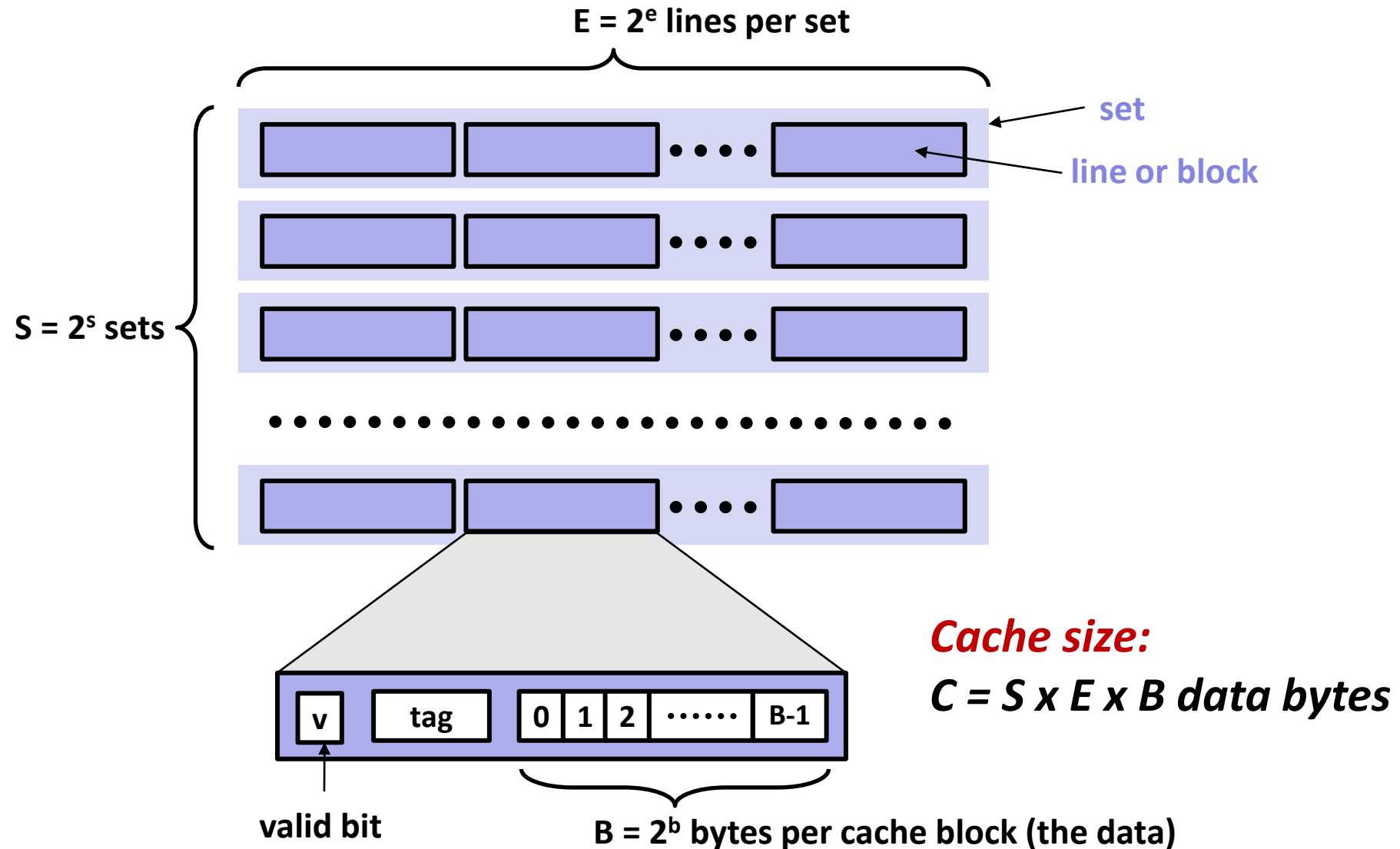
- Increased associativity decreases miss rate
  - But with diminishing returns
- Simulation of a system
  - 64KB D-cache
  - 16-word blocks
  - SPEC2000



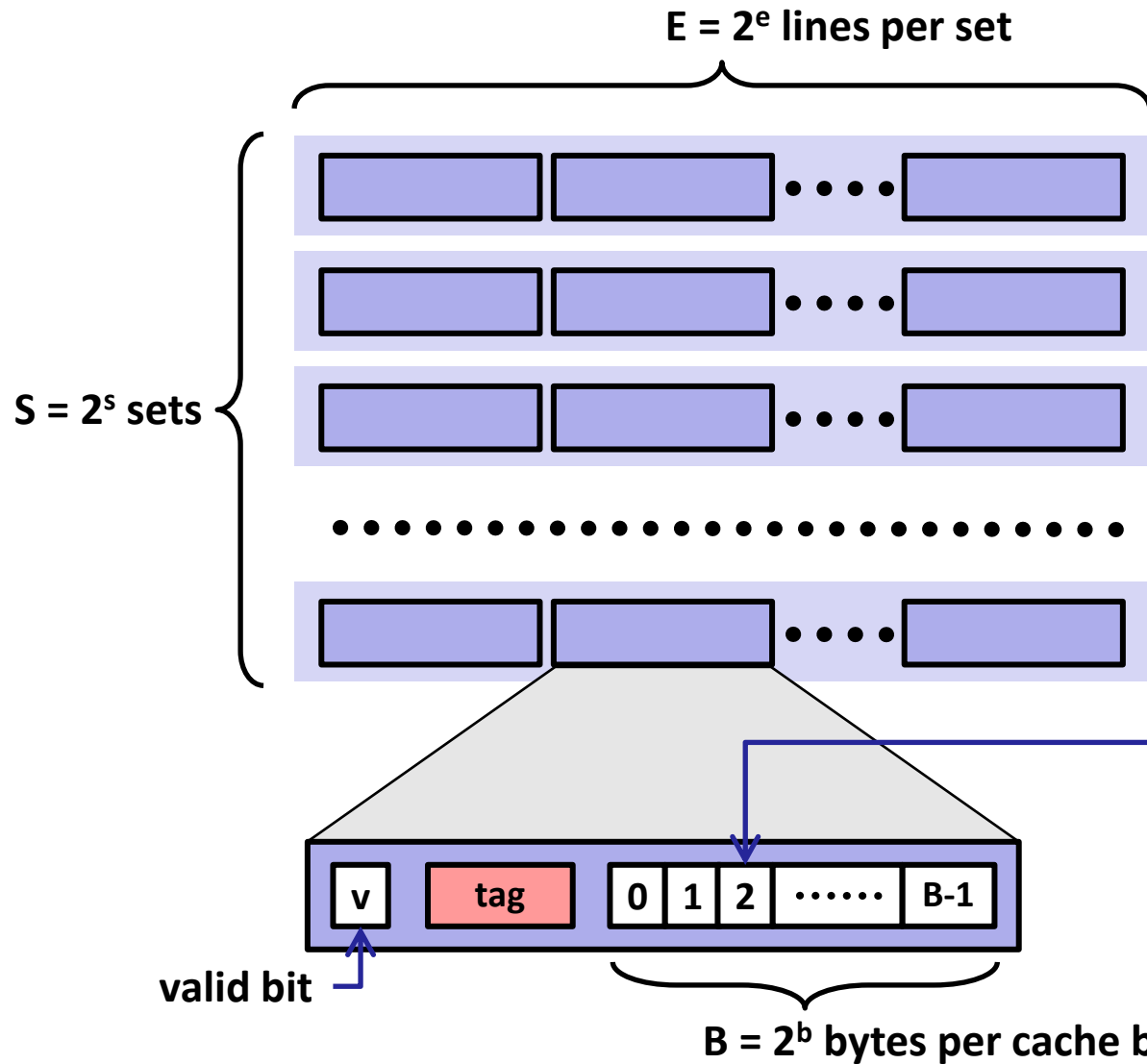
# Set Associativity Cache Organization



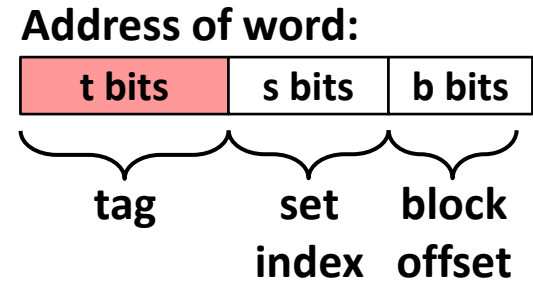
# General Cache Organization (S, E, B)



# Cache Read



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



data begins at this offset

# Replacement Policy

- Direct mapped: no choice
- Set associative
  - Prefer non-valid entry, if there is one
  - Otherwise, choose among entries in the set
- Least-recently used (LRU)
  - Choose the one unused for the longest time
  - Simple for 2-way, manageable for 4-way, too hard beyond that
- Random
  - Gives approximately the same performance as LRU for high associativity

# Multilevel Caches

- Primary cache attached to CPU (Level-1 or L1)
  - Small, but fast
- Level-2 (L2) cache services misses from primary cache
  - Larger, slower, but still faster than main memory
- Main memory services L2 cache misses
- Some high-end systems include L3 cache



# Multilevel Cache Example

- **Given**
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate / instruction = 2%
  - Main memory access time = 100ns
  
- **With just primary cache**
  - Miss penalty =  $100\text{ns} / 0.25\text{ns} = 400$  cycles
  - Effective CPI =  $1 + 0.02 \times 400 = 9$

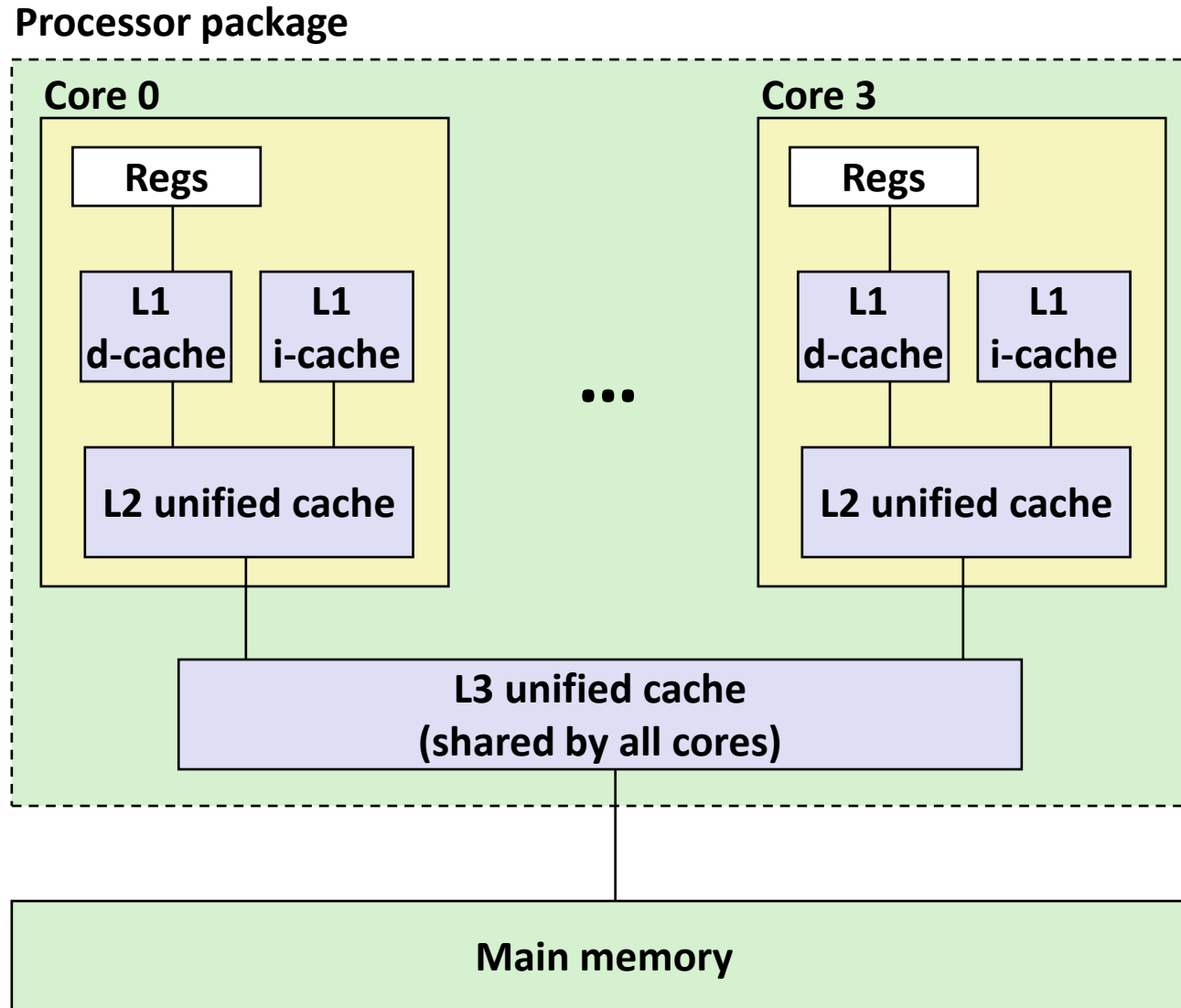
# Multilevel Cache Example (cont'd)

- Now add L2 cache
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L2 hit
  - Penalty =  $5\text{ns} / 0.25\text{ns} = 20$  cycles
- Primary miss with L2 miss
  - Extra penalty = 400 cycles
- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio =  $9 / 3.4 = 2.6$

# Multilevel Cache Considerations

- **Primary cache**
  - Focus on minimal hit time
- **L2 cache**
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- **Results**
  - L1 cache is usually smaller than a single-level cache
  - L1 block size may be smaller than L2 block size
  - L2 cache is much larger than in a single-level cache
  - L2 cache uses higher associativity for reducing miss rates

# Intel Core i7 Cache Hierarchy



## 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:

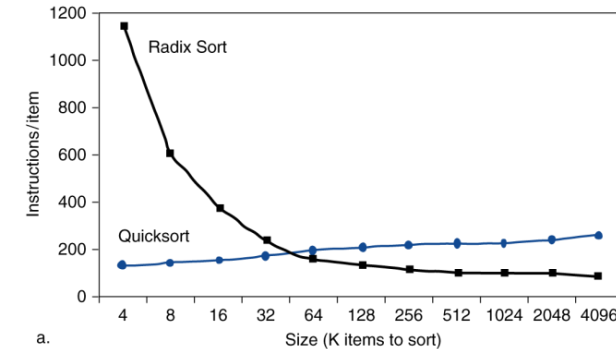
8 MB, 16-way,  
Access: 40-75 cycles

## Block size:

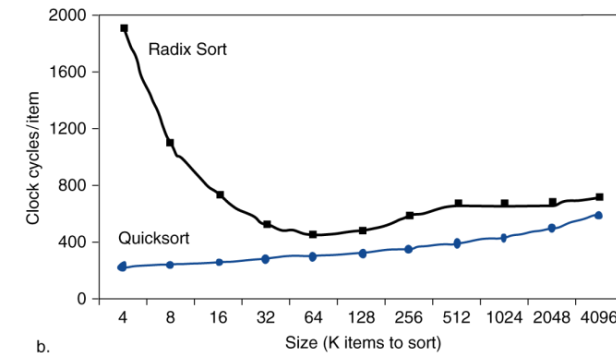
64 bytes for all caches

# Interactions with Software

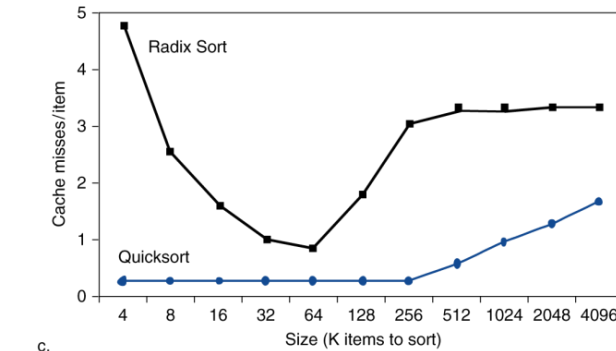
- Misses depend on memory access patterns
  - Algorithm behavior
  - Compiler optimization for memory access
- Standard algorithmic analysis often ignores the impact of the memory hierarchy



a.



b.



c.

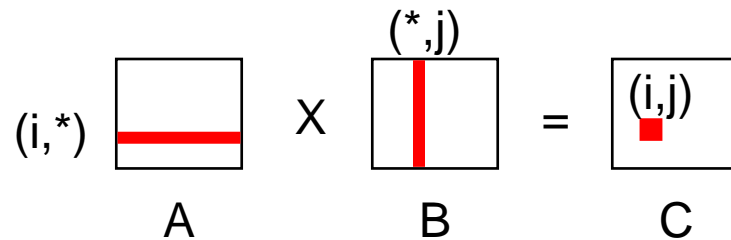
# Writing Cache Friendly Code

- 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

# Matrix Multiplication (I)

## ■ Description

- Multiply  $N \times N$  matrices
- $O(N^3)$  total operations



## ■ Assumptions

- Line size = 32 bytes (big enough for 4 64-bit words)
- Matrix dimension (N) is very large

```
/* 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;  
    }  
}
```

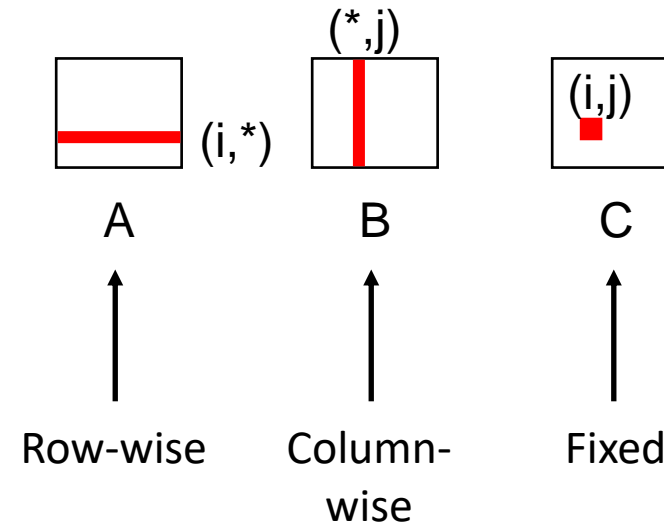
*Variable sum held in register*

# Matrix Multiplication (2)

- Matrix multiplication (ijk)

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

Inner loop:



## Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

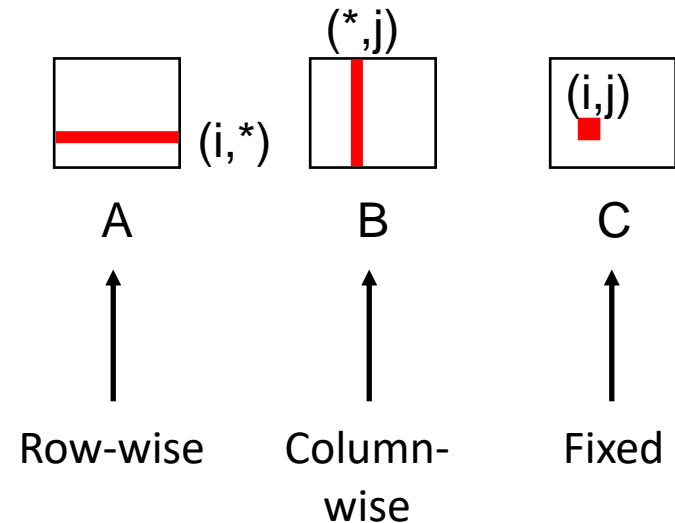


# Matrix Multiplication (3)

- Matrix multiplication (jik)

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

Inner loop:



### Misses per Inner Loop Iteration:

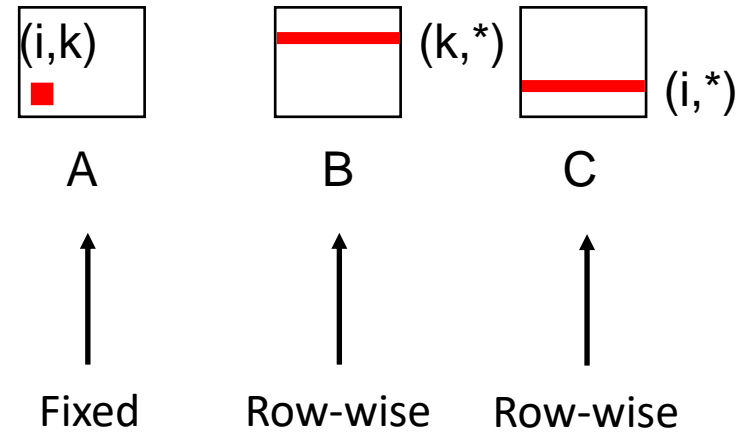
<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

# Matrix Multiplication (4)

- 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:



### Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

# Matrix Multiplication (5)

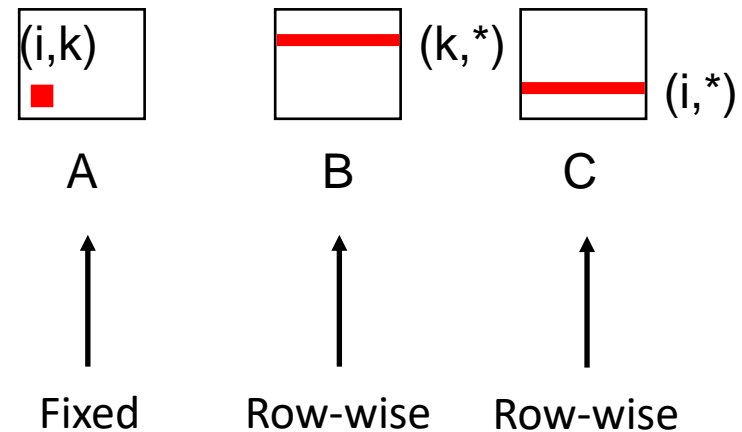
- Matrix multiplication (ikj)

```
/* 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];  
  }  
}
```

## Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

Inner loop:



# Matrix Multiplication (6)

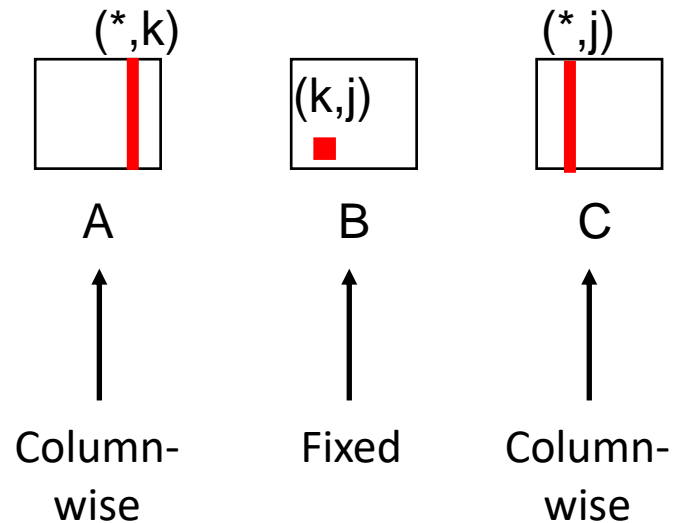
- Matrix multiplication (jki)

```
/* 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;
  }
}
```

### Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Inner loop:



# Matrix Multiplication (7)

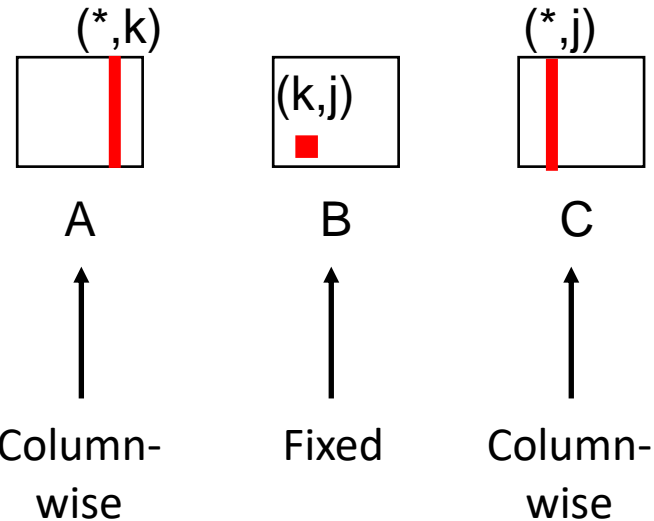
- Matrix multiplication (kji)

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

## Misses per Inner Loop Iteration:

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

Inner loop:



# Matrix Multiplication (8)

## ■ Summary

### ijk (& jik):

- 2 loads, 0 stores
- misses/iter = **1.25**

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

<u>A</u>	<u>B</u>	<u>C</u>
0.25	1.0	0.0

### kij (& ikj):

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

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

<u>A</u>	<u>B</u>	<u>C</u>
0.0	0.25	0.25

### jki (& kji):

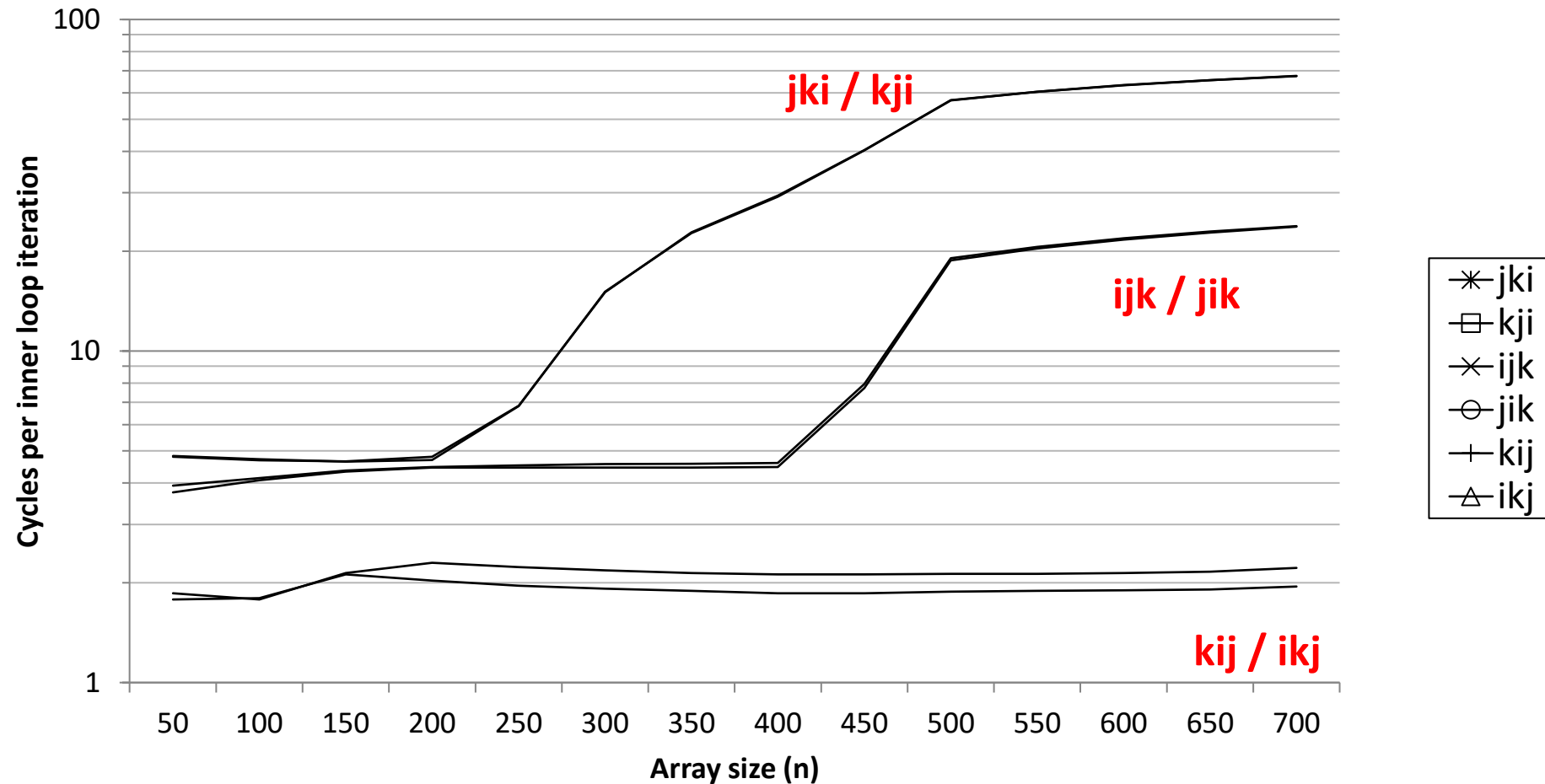
- 2 loads, 1 store
- misses/iter = **2.0**

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

<u>A</u>	<u>B</u>	<u>C</u>
1.0	0.0	1.0

# Core i7 Matrix Multiplication Performance

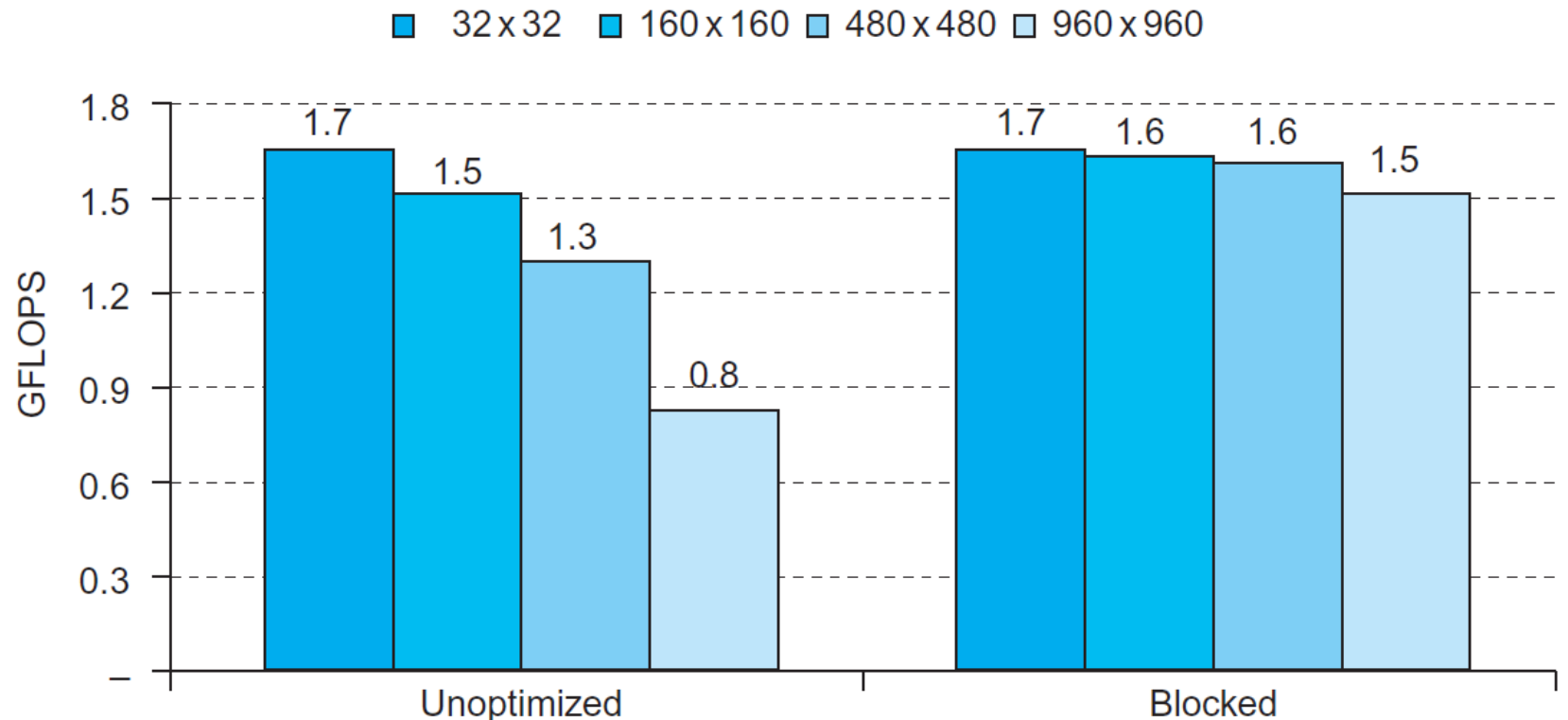
- Performance in Core i7



# DGEMM with Cache Blocking

- Double precision GEMM

- Intel Core i7 (Sandy Bridge)
- 32x32 matrix: 8KB
- All three matrices (24KB) fit in the 32KB L1 data cache
- See Figure 5.21





# Summary

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