Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2022

Cache Optimization

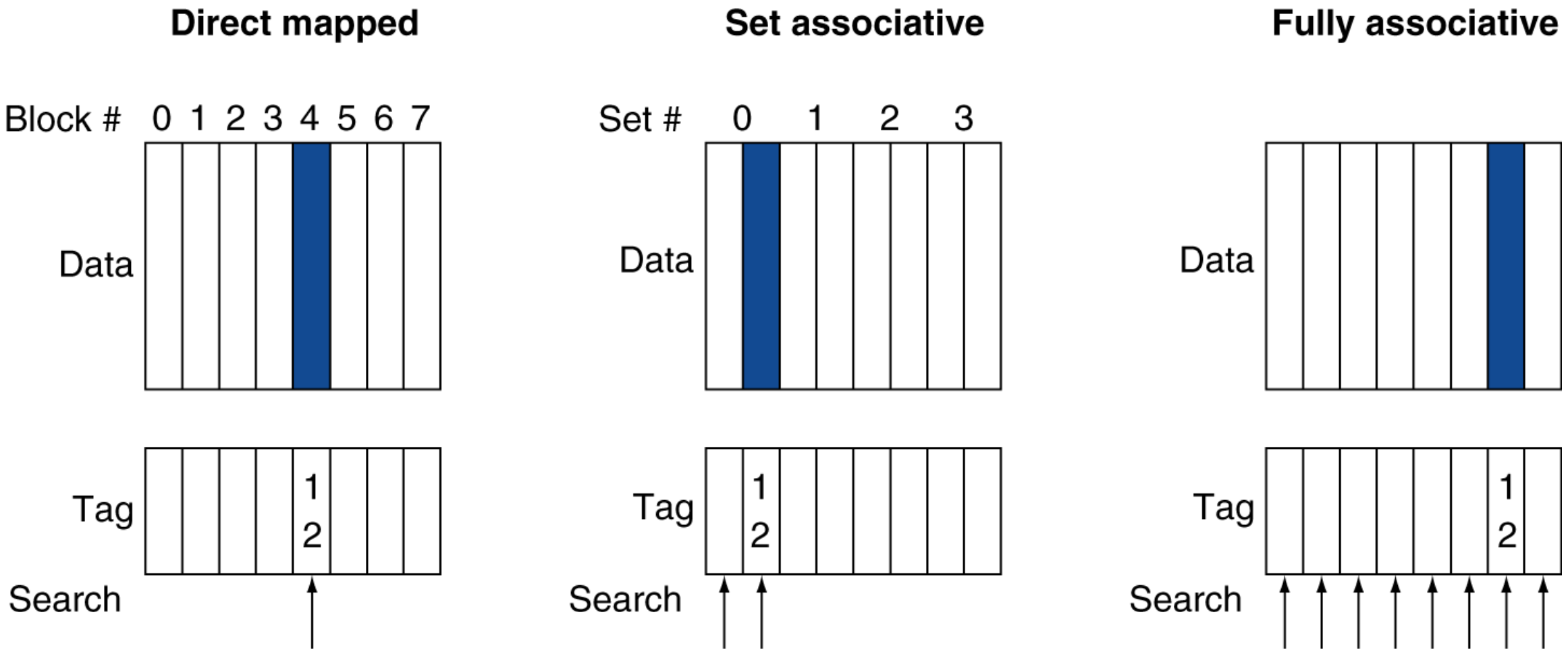
Chap. 5.4

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* modulo *#Sets_in_cache*
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

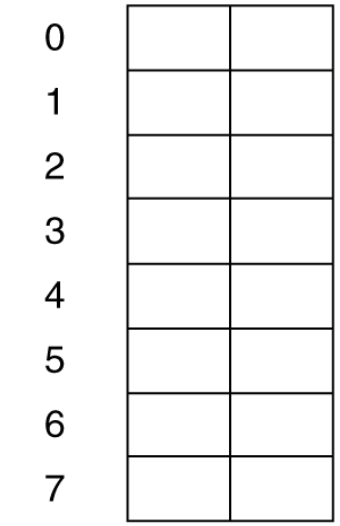
Associative Cache Example



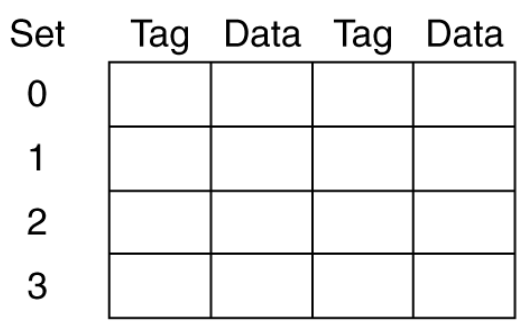
Spectrum of Associativity

- For a cache with 8 entries

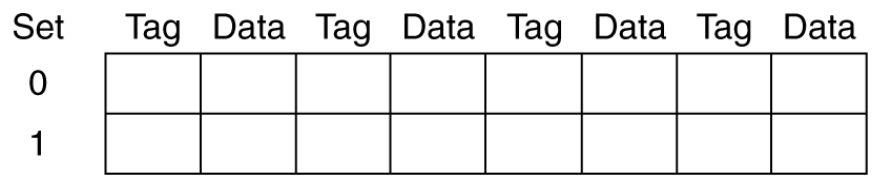
One-way set associative (direct mapped)



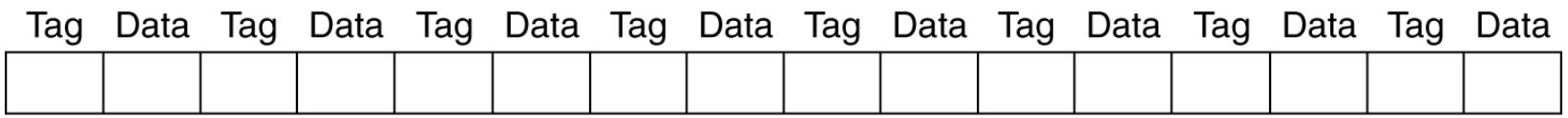
Two-way set associative



Four-way set associative



Eight-way set associative (fully associative)



Associativity Example (1)

- 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]		

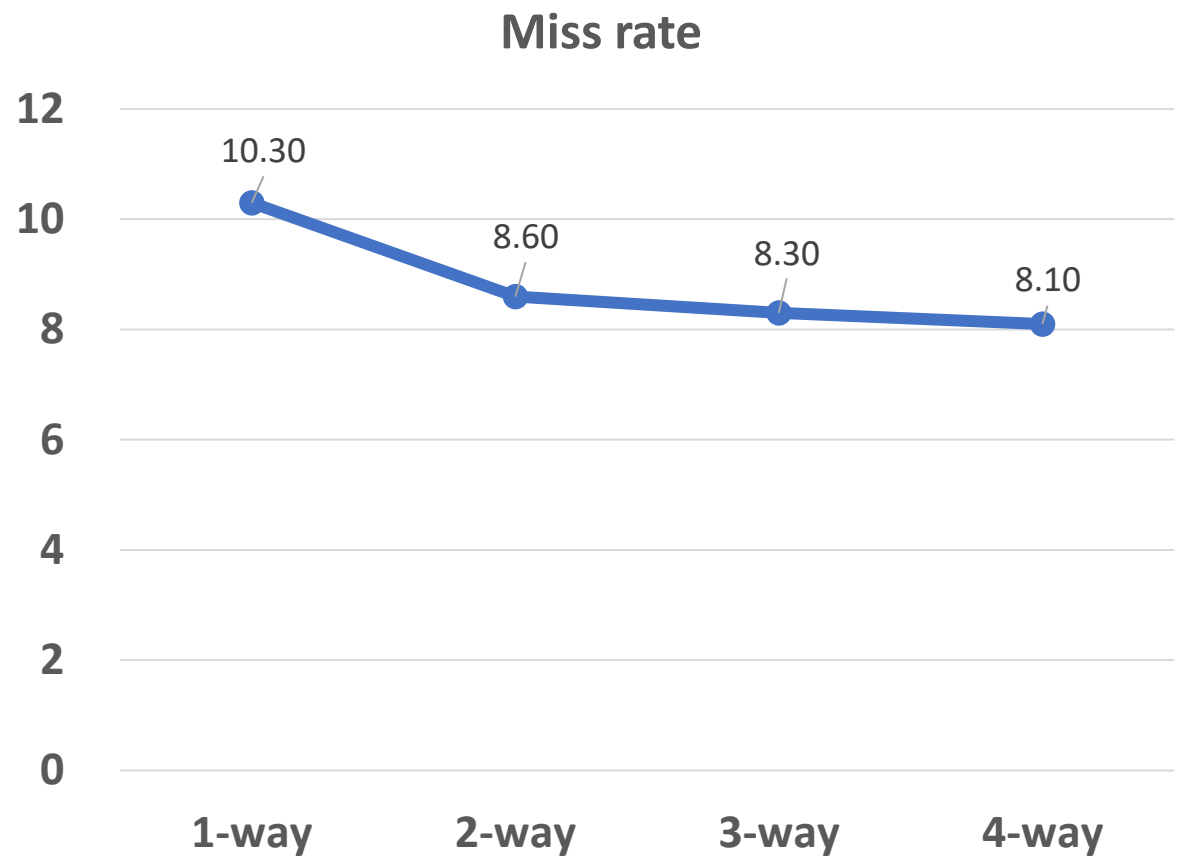
- Fully associative

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]	

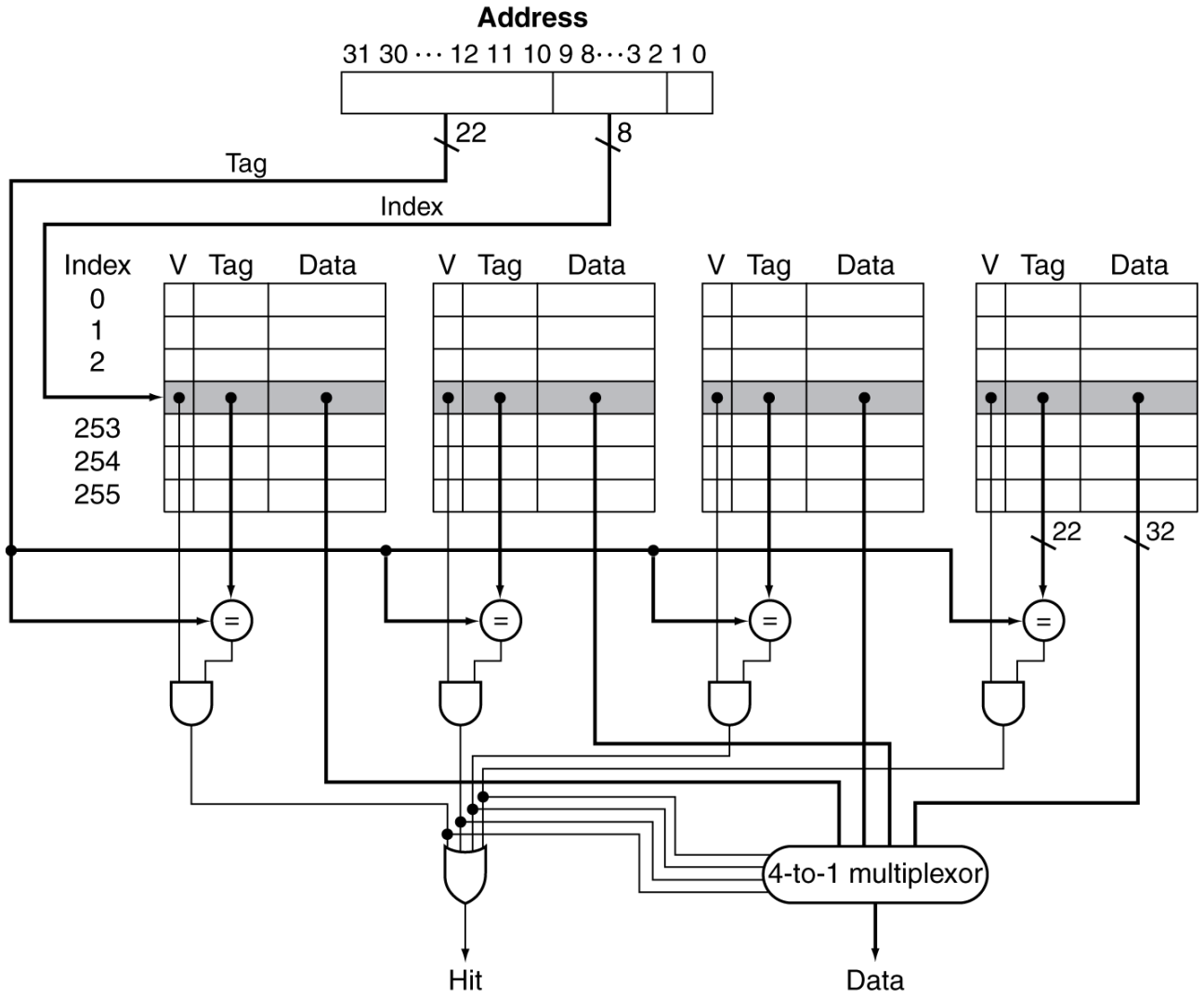
How Much Associativity?

- Increased associativity decreases miss rate
  - But with diminishing returns

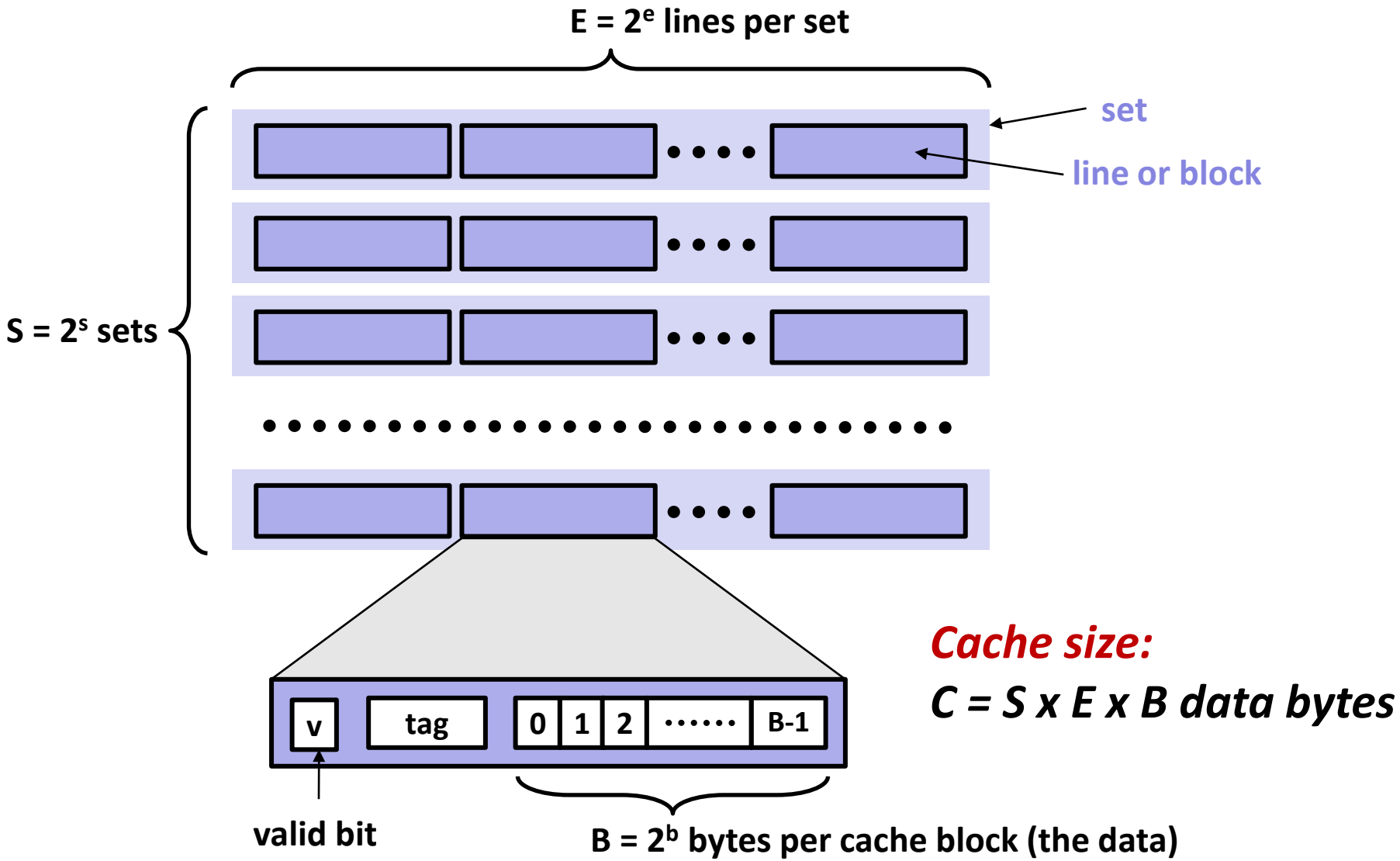
- Simulation of a system
  - 64KB D-cache
  - 16-word blocks
  - SPEC2000



Set Associativity Cache Organization



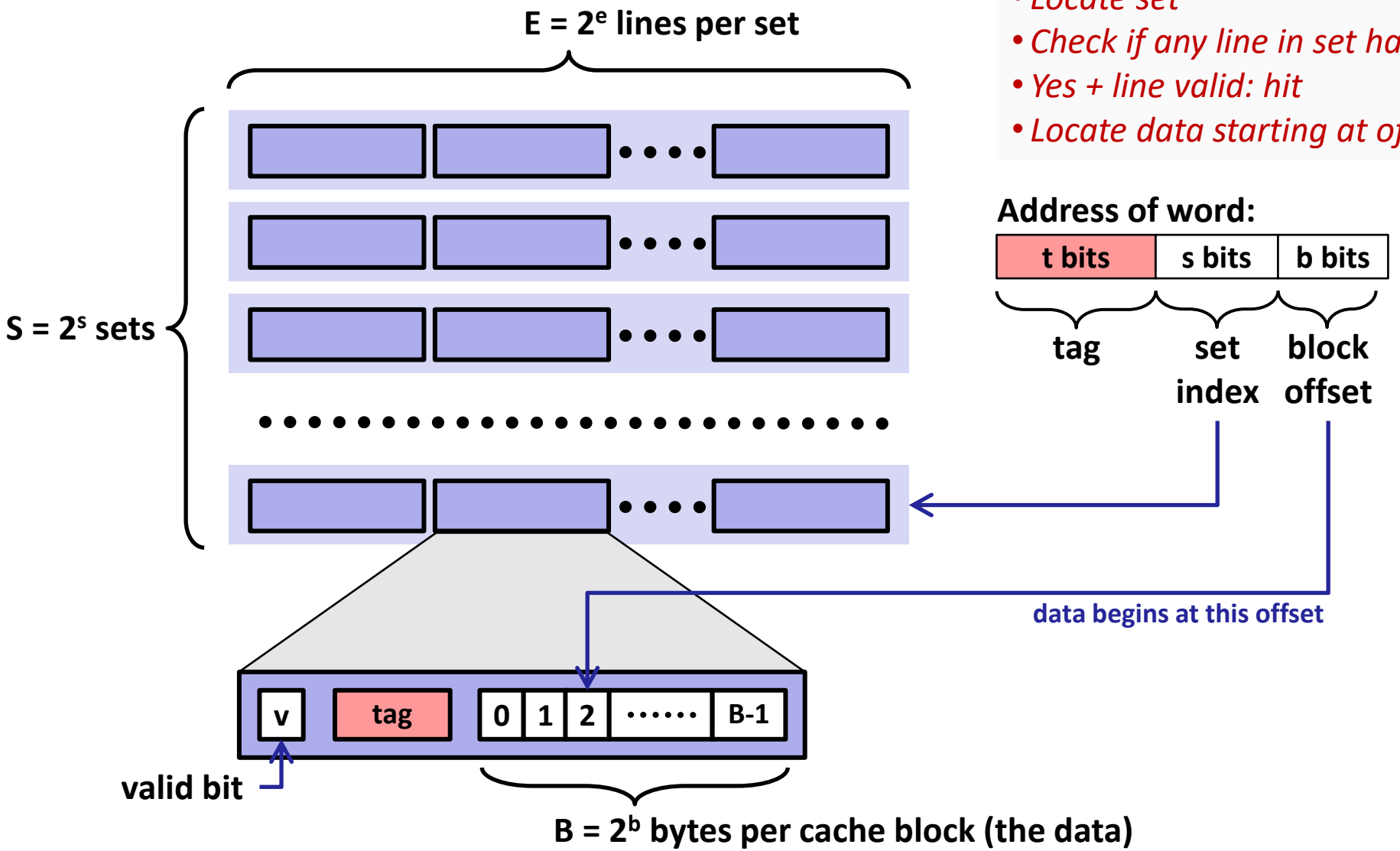
General Cache Organization (S, E, B)



Cache size:

C = S x E x B data bytes

Cache Read



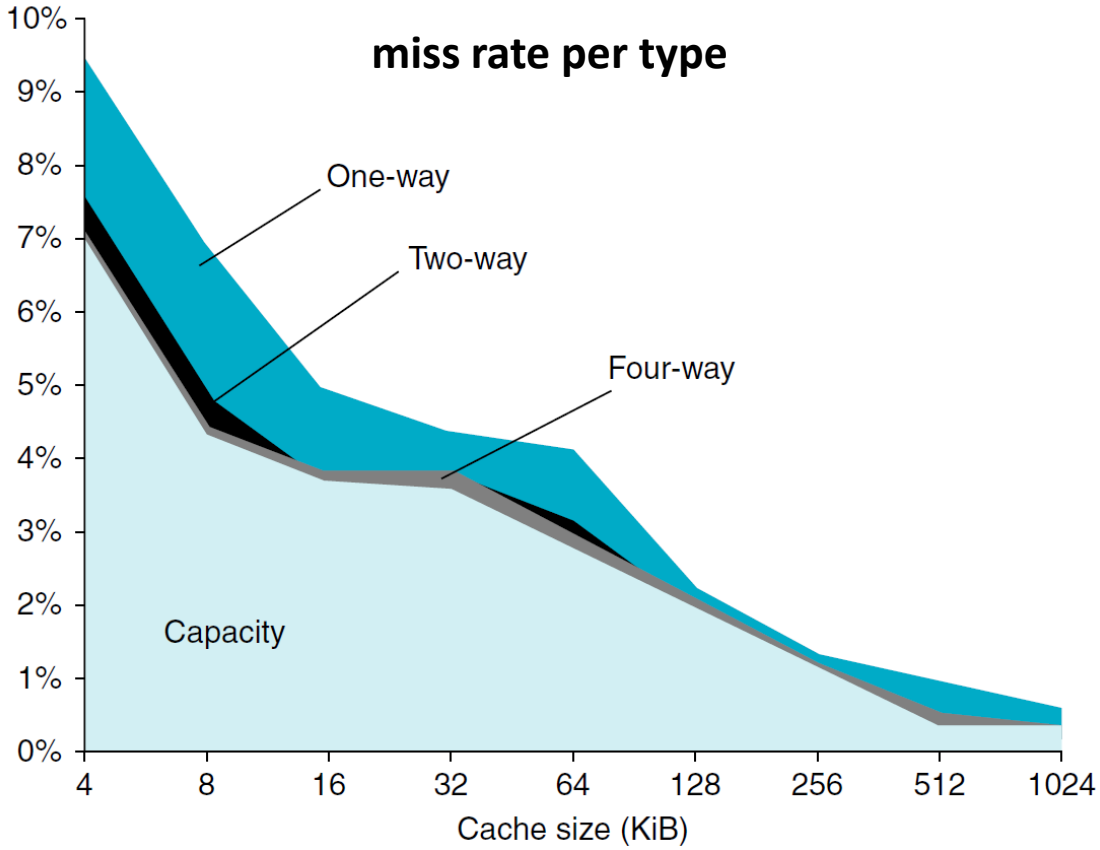
- *Locate set*
- *Check if any line in set has matching tag*
- *Yes + line valid: hit*
- *Locate data starting at offset*

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

Sources of Misses (Three C's)

- Compulsory misses (or cold-start misses)
  - First access to a block
- Capacity misses
  - Due to finite cache size
  - A replaced block is later accessed again
- Conflict misses (or collision misses)
  - In a non-fully associative cache
  - Due to competition for entries in a set
  - Would not occur in a fully associative cache of the same total size



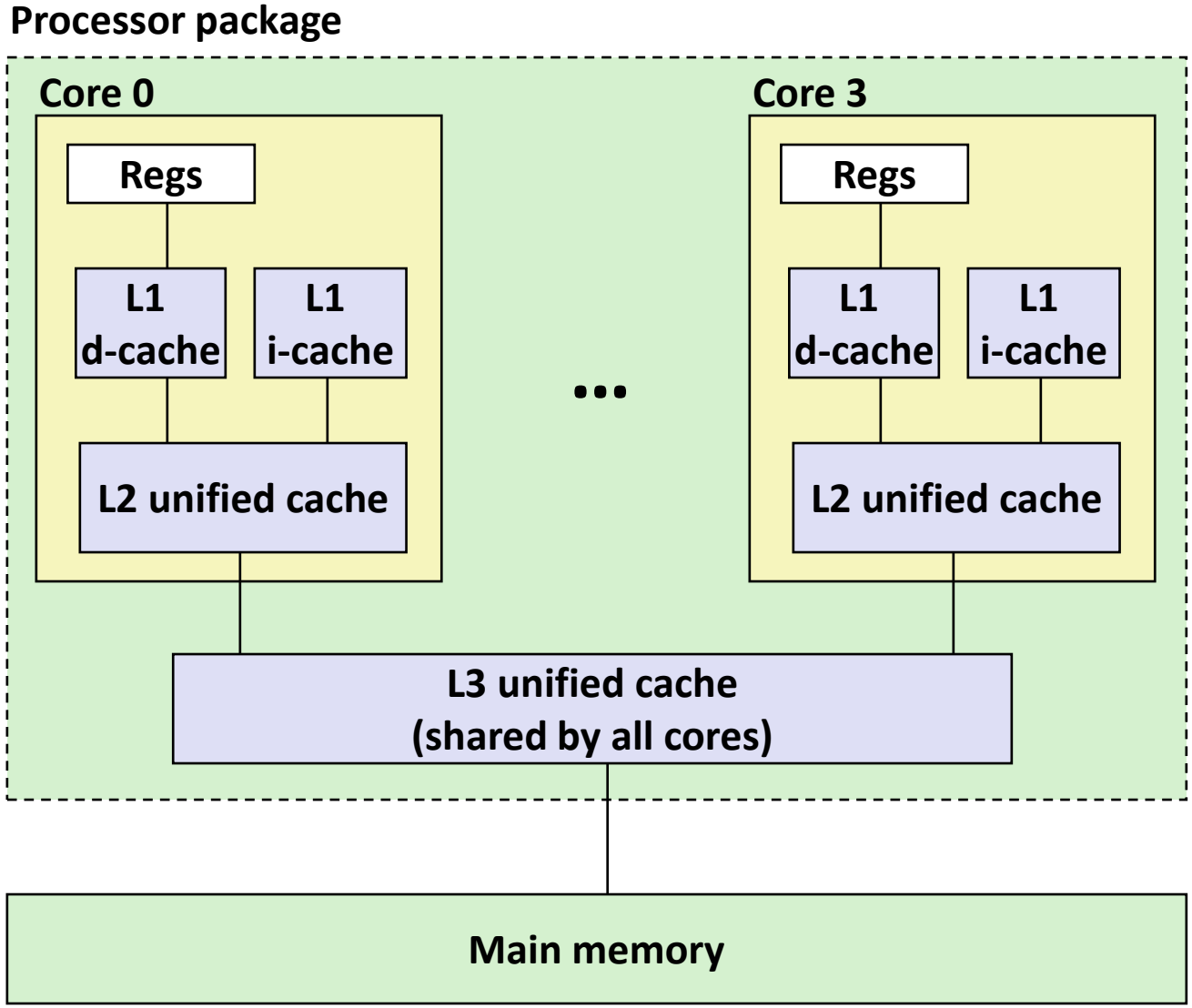
Multilevel Caches

- 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 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

Cache Performance

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

$$Memory\ stall\ cycles$$

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

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

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 x 100 = 2
  - D-cache: 0.36 x 0.04 x 100 = 1.44
- Actual CPI = 2 + 2 + 1.44 = 5.44
  - Ideal CPU is 5.44/2 = 2.72 times faster

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 = 100ns / 0.25ns = 400 cycles
  - Effective CPI = 1 + 0.02 x 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 = 5ns / 0.25ns = 20 cycles
- Primary miss with L2 miss
  - Extra penalty = 400 cycles

- CPI = 1 + 0.02 x 20 + 0.005 x 400 = 3.4
- Performance ratio = 9 / 3.4 = 2.6

Average Access Time

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

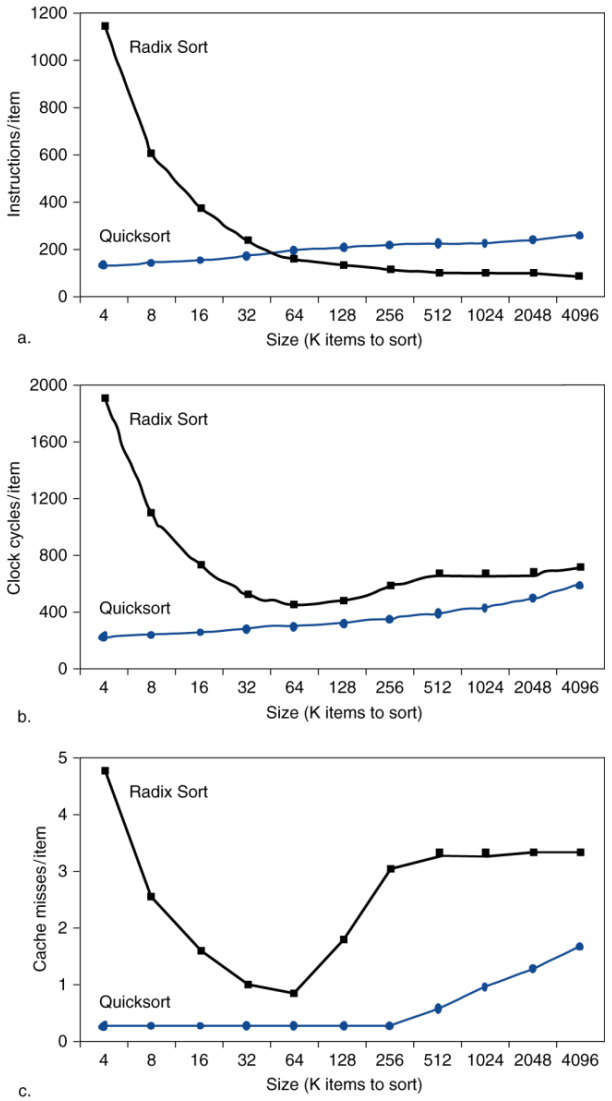
$$AMAT = Hit\ time + Miss\ rate \times Miss\ penalty$$

- Example
  - CPU with 1ns clock
  - Hit time = 1 cycle,
  - Miss penalty = 20 cycles
  - Misses per instruction: 5%
  - AMAT = 1 + 0.05 x 20 = 2ns (2 cycles per instruction)

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



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

Writing Cache Friendly Code

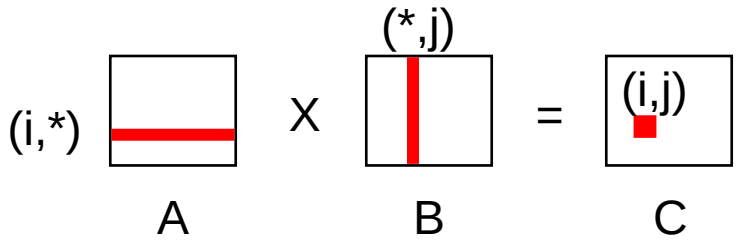
- Make the common case go fast
  - Focus on the inner loops of the core functions

- 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 (1)

- Description
  - Multiply N x N matrices
  - $O(N^3)$ 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;
  }
}
```

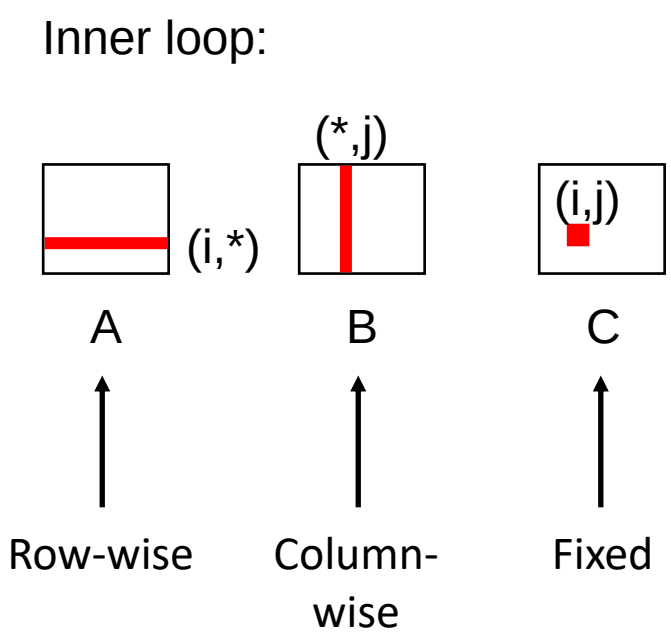
*Variable **sum** held in register*

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

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;
  }
}
```



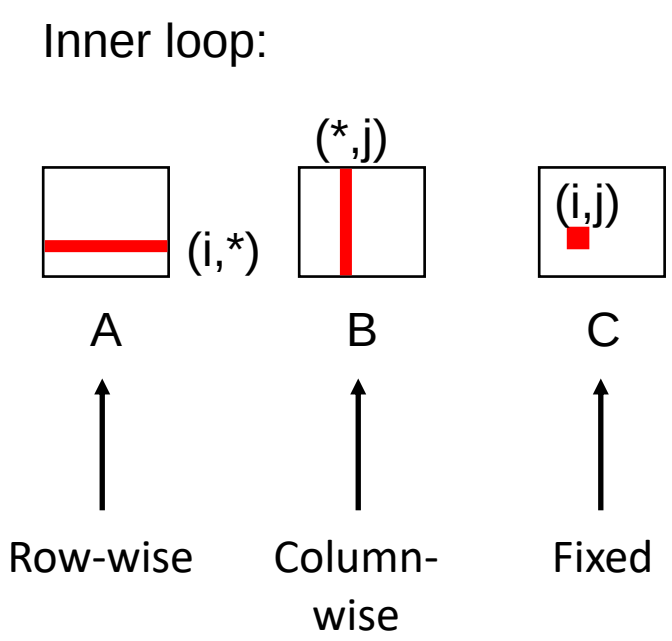
Misses per Inner Loop Iteration:

A	B	C
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
  }
}
```



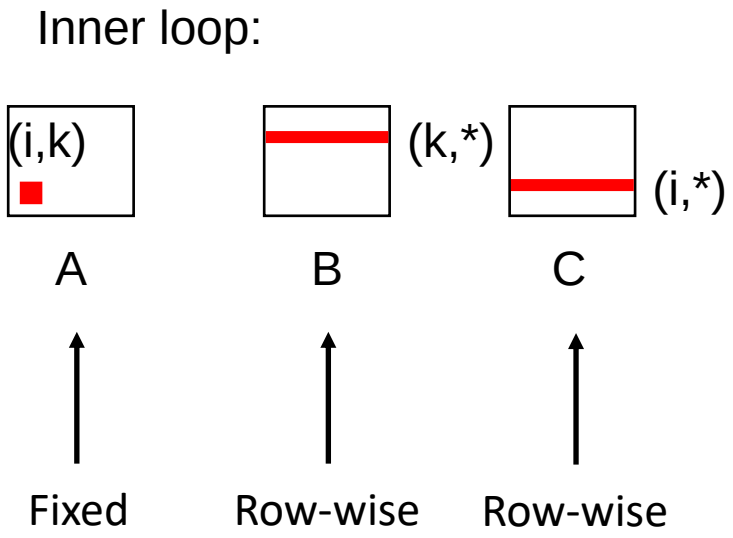
Misses per Inner Loop Iteration:

A	B	C
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];
  }
}
```



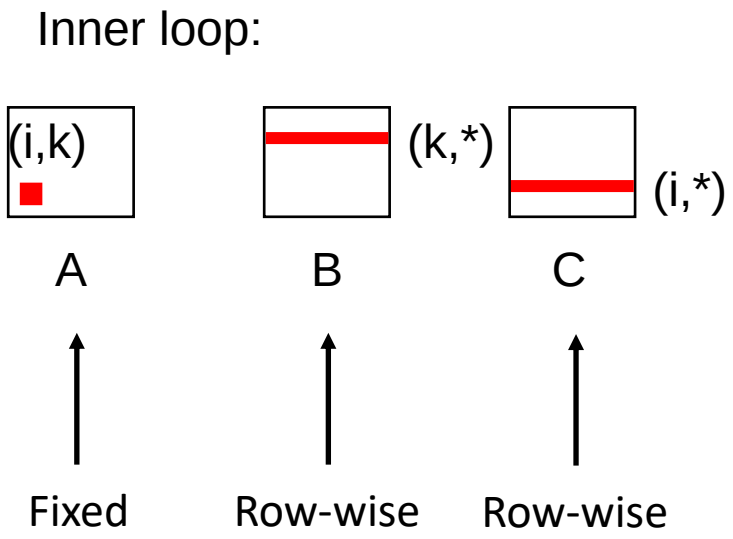
Misses per Inner Loop Iteration:

A	B	C
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];
  }
}
```



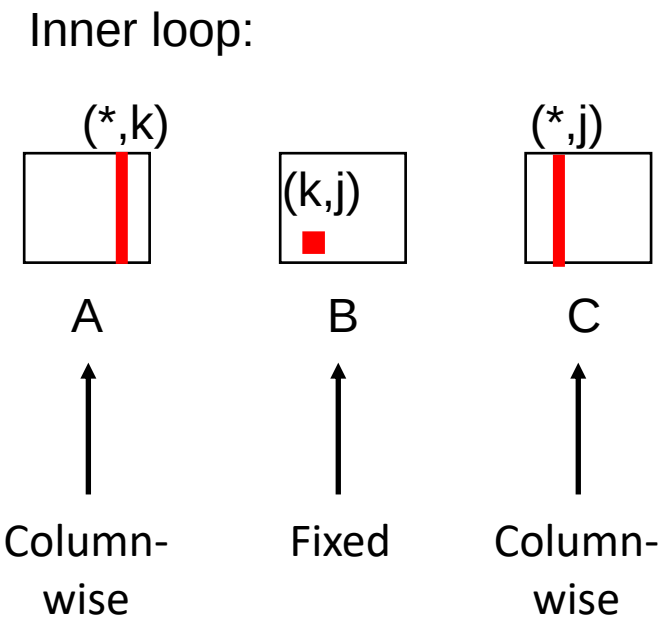
Misses per Inner Loop Iteration:

A	B	C
0.0	0.25	0.25

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;
  }
}
```



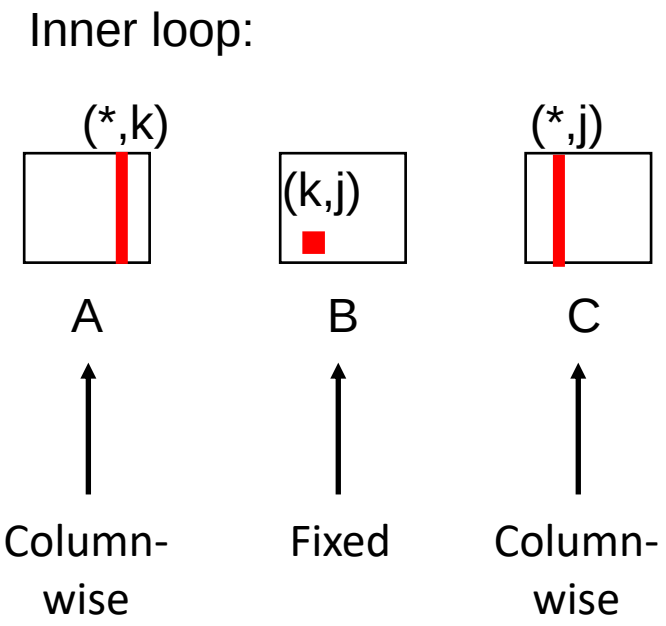
Misses per Inner Loop Iteration:

A	B	C
1.0	0.0	1.0

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:

A	B	C
1.0	0.0	1.0

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;
 }
}
```

A	B	C
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];
 }
}
```

A	B	C
0.0	0.25	0.25

jki (& kji):

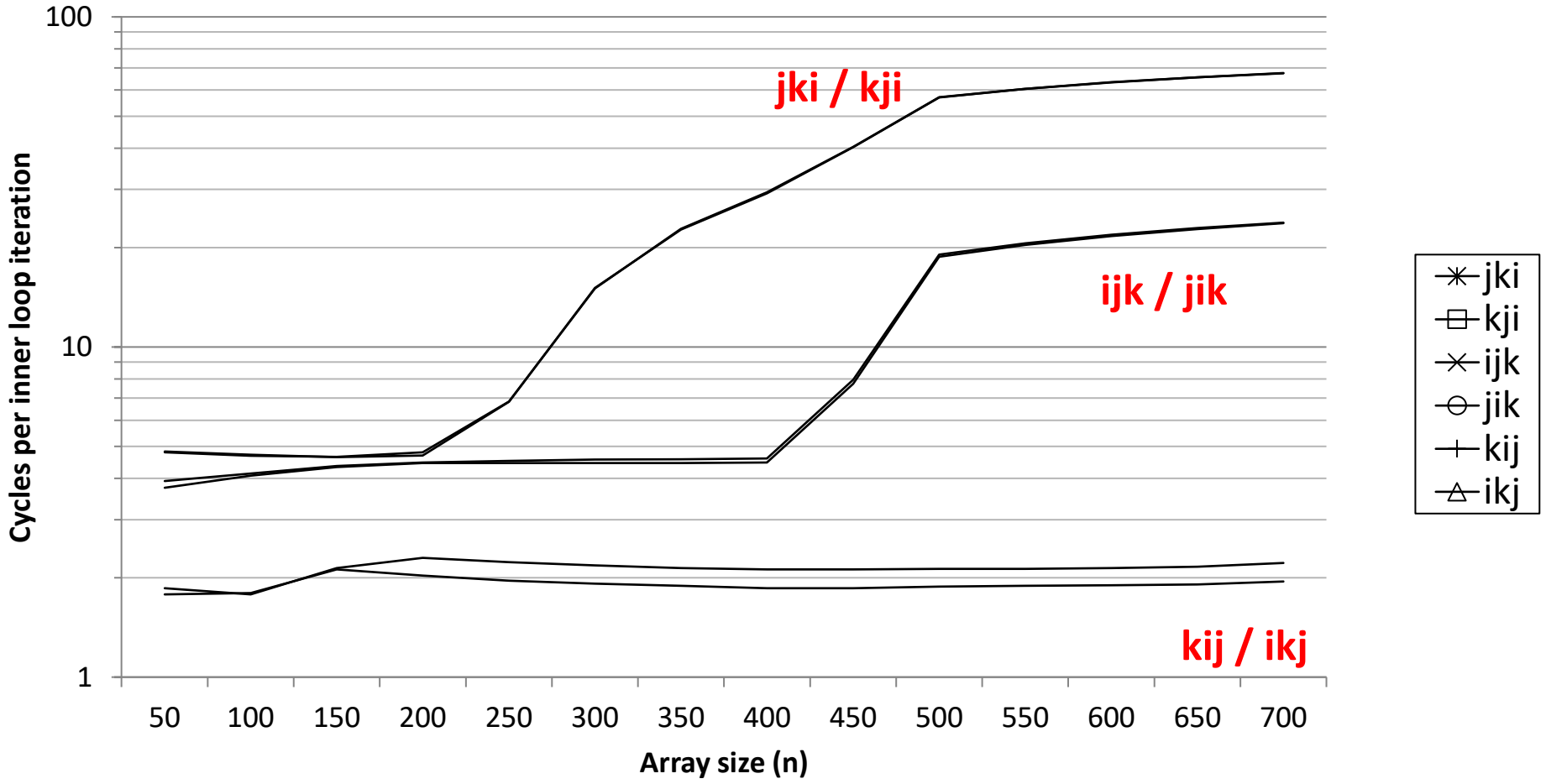
- 2 loads, 1 store
- misses/iter = **2.0**

```
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;
 }
}
```

A	B	C
1.0	0.0	1.0

Core i7 Matrix Multiplication Performance

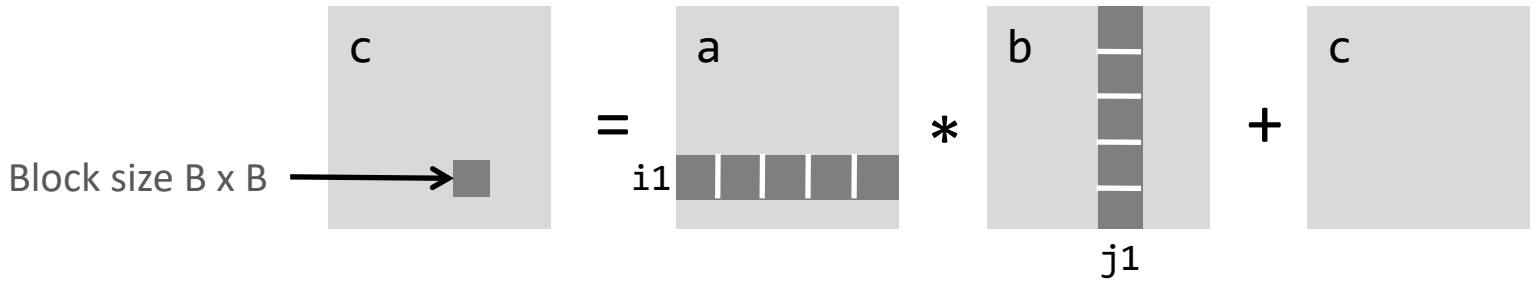
- Performance in Core i7



Blocked Matrix Multiplication

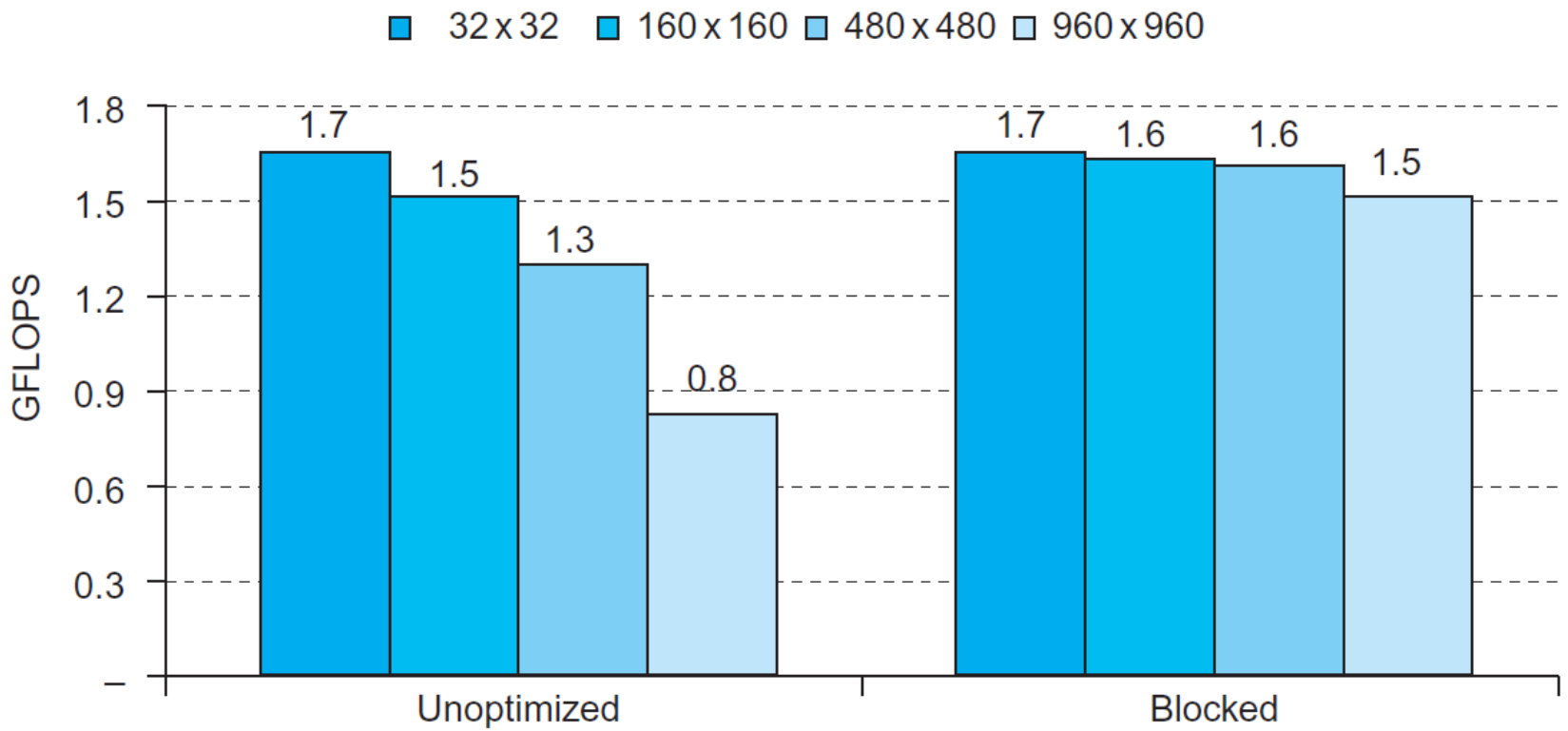
- Three ■ blocks fit into cache:

```
/* 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];
}
```



DGEMM with Cache Blocking

- Double precision GEneral Matrix Multiply
  - Intel Core i7 (Sandy Bridge)
  - 32x32 matrix: 8KB
  - All three matrices (24KB) fit in the 32KB L1 data cache



Summary

- Cache memories can have significant performance impact

- You ~~can~~ *should* 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