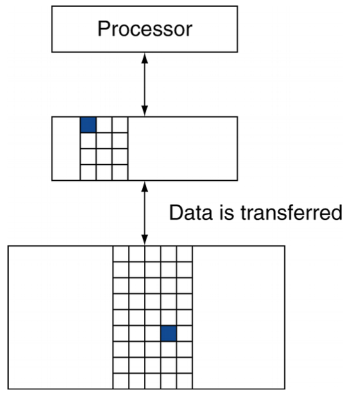
Where are we?

1. Where can a block be placed? **(Q1)**
2. How is a block found? **(Q2)**
3. What block is replaced on a miss? **(Q3)**
4. How are writes handled? **(Q4)**

Memory Hierarchy



Direct mapped cache memory

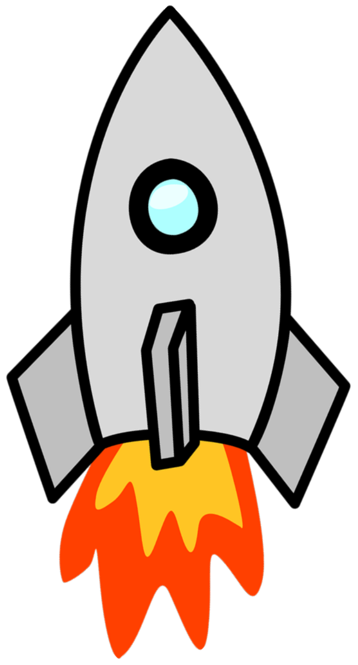
Cache Performance

Improving Cache Performance

Reduce miss rate (??)

Reduce miss (time) penalty (??)

Speed hit access time!



Example

- We have **4-block** caches
  - Direct mapped
  - **Block access sequence**: 0, 8, 0, 6, 8

Block address	Cache index	Hit/miss	Cache content after access			
			0	1	2	3
0	0	miss	**Mem[0]**			
8	0	miss	**Mem[8]**			
0	0	miss	**Mem[0]**			
6	2	miss	Mem[0]		**Mem[6]**	
8	0	miss	**Mem[8]**		Mem[6]	

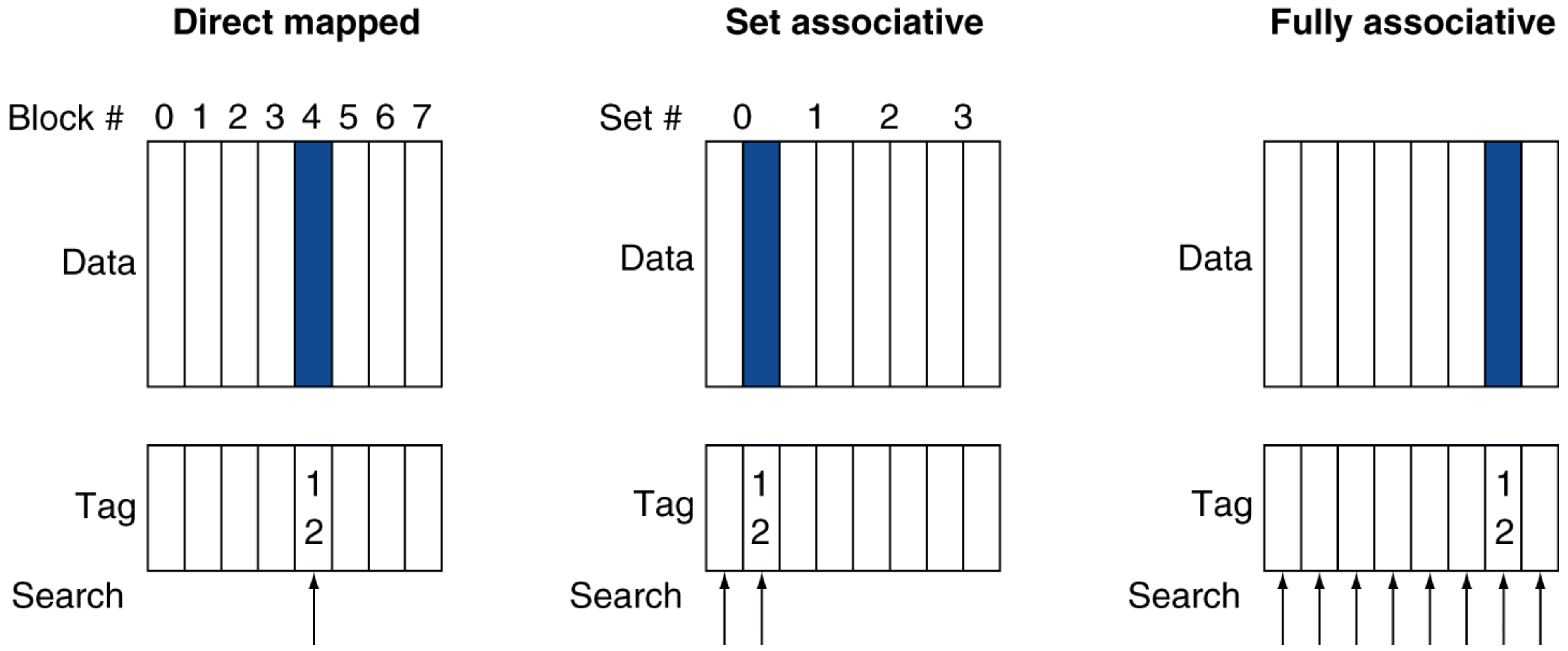
Reducing Miss Rate

Flexible block location to reduce the competition for cache blocks → ***Associative Caches***

- ***Fully associative caches***
  - Allow a given block to go in any cache entry (less competition)
  - Increased search time
    - Search all entries at once to reduce hit time

- ***n-way set-associative caches***
  - 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

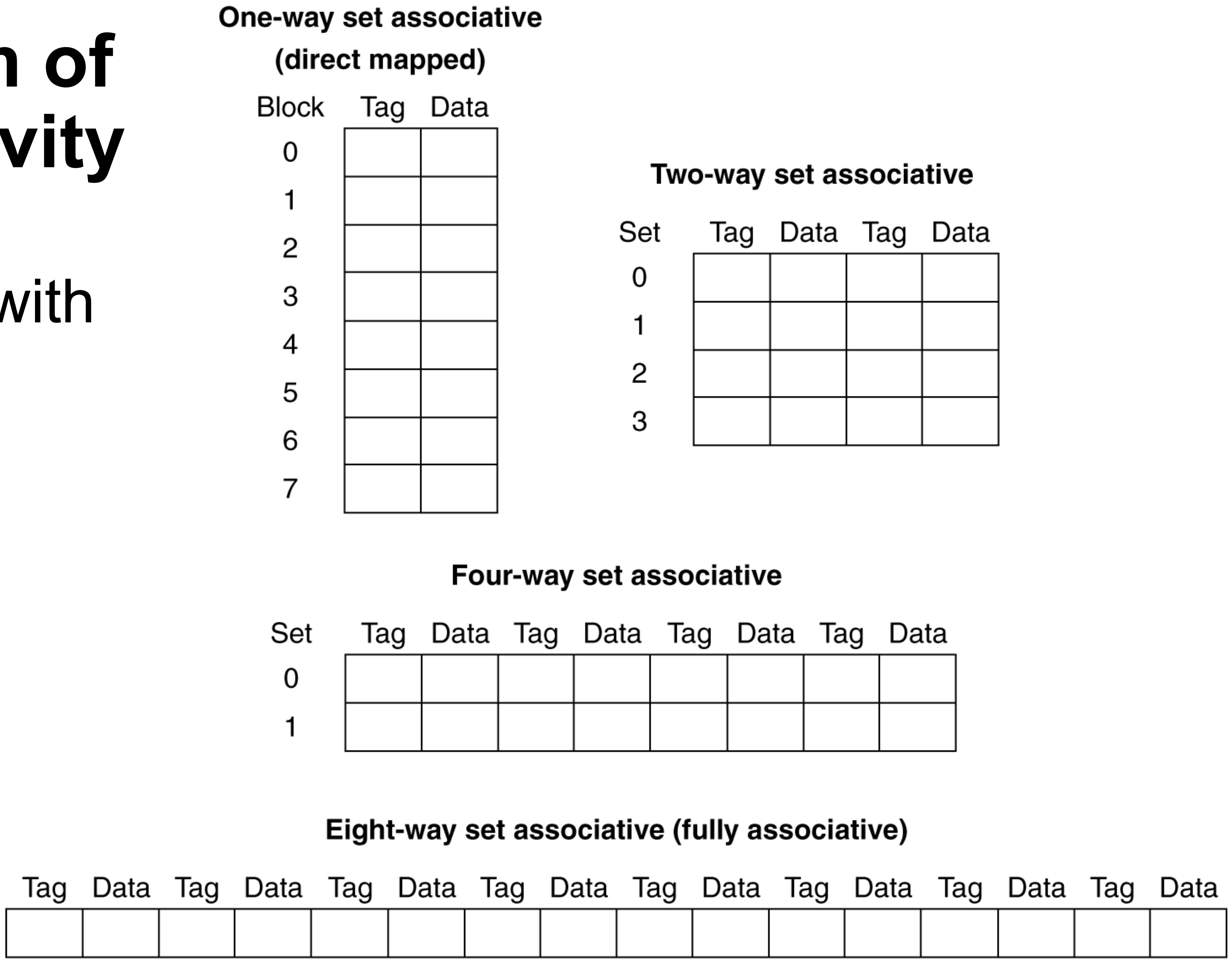


Block position = (Block #) modulo (# of blocks in the cache)

Set position = (Block #) modulo (# of sets in the cache)

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

- 2-way set associative

Block address	Cache index	Hit/miss	Cache content after access			
			Set 0		**Set 1**	
0	0	miss	**Mem[0]**			
8	0	miss	Mem[0]	**Mem[8]**		
0	0	hit	**Mem[0]**	Mem[8]		
6	0	miss	Mem[0]	**Mem[6]**		
8	0	miss	**Mem[8]**	Mem[6]		

Associativity Example

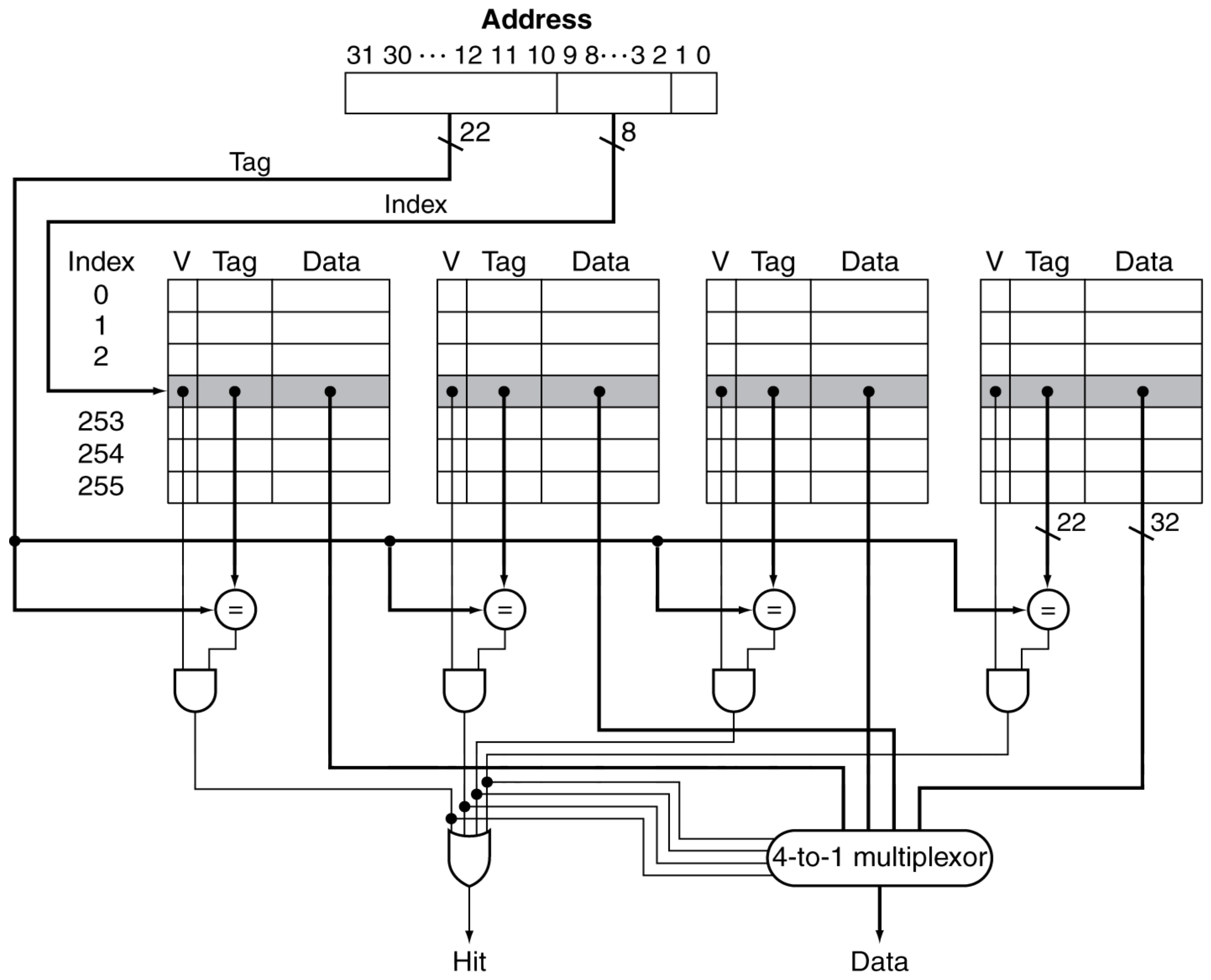
Fully associative

Block address		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 with 64KB D-cache, 16-word blocks, SPEC2000
  - 1-way: 10.3%
  - 2-way: 8.6%
  - 4-way: 8.3%
  - 8-way: 8.1%
- Performance cost tradeoff
  - Better performance requires additional hardware for speeding hit time

Set Associative Cache Organization

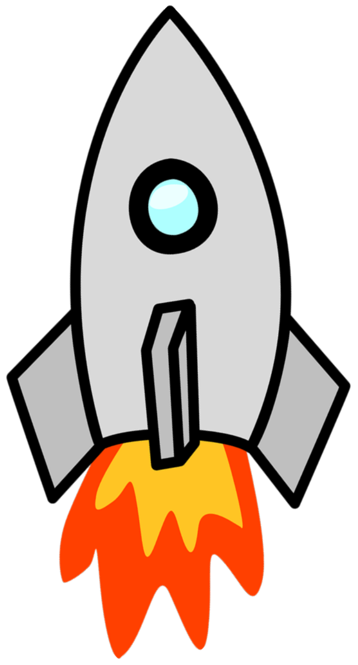


Replacement Policy

- 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

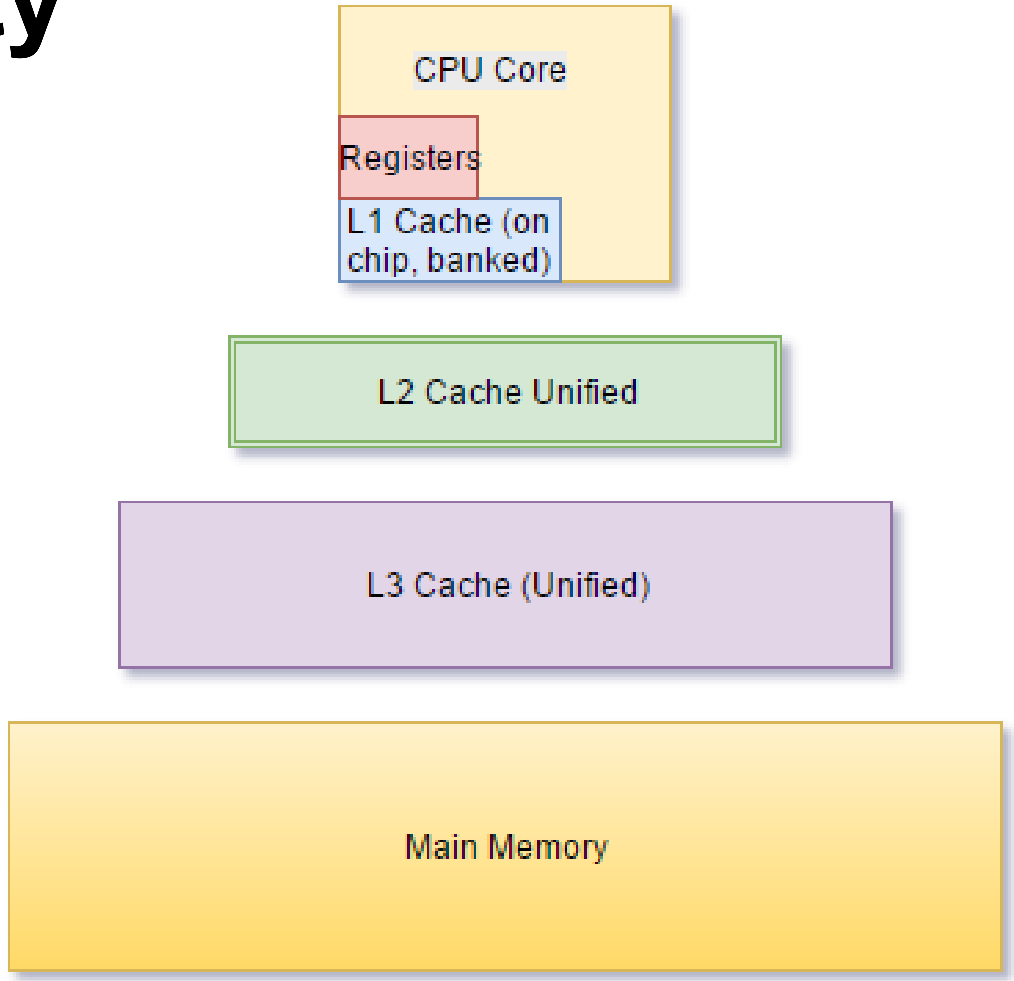
Improving Cache Performance

Reduce miss rate (**Associative cache**)
Reduce miss (time) penalty (??)
Speed hit access time!



Reducing the miss penalty

- Approach: ***Multi-level Cache***
  - Level-1 cache (Primary cache): service the CPU, Small, but fast
  - Level-2 cache: services misses from primary cache, still faster than main memory
-



Multilevel Cache Example

- Given
  - CPU base CPI = 1, clock rate = 4GHz
  - Miss rate/instruction = 2%
  - Main memory access time = 100ns
- With just a primary cache
  - Miss penalty = 100ns/0.25ns = 400 cycles
  - Effective CPI = Base CPI + Memory-stall cycles per inst.
    = 1 + 0.02 × 400 = 9

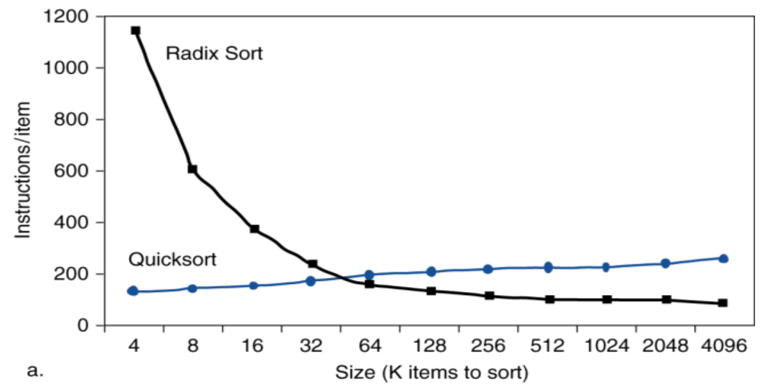
Example (cont.)

- **Now add L-2 cache**
  - Access time = 5ns
  - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
  - Penalty = 5ns/0.25ns = 20 cycles
- Total CPI = 1 + primary stall + secondary stall
  - CPI = $1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = 9/3.4 = 2.6
  - Faster by a factor of 2.6 with secondary cache

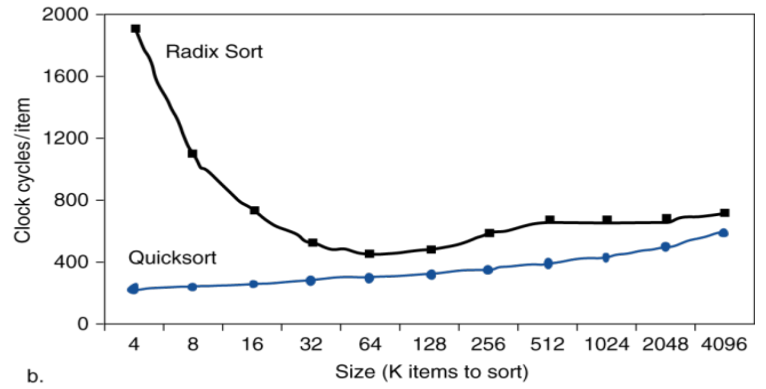
Multilevel Cache Considerations

- Primary cache
  - Focus on minimal hit time
- L-2 cache
  - Focus on low miss rate to avoid main memory access
  - Hit time has less overall impact
- Results
  - L-1 cache usually smaller than a single cache
  - L-1 block size smaller than L-2 block size

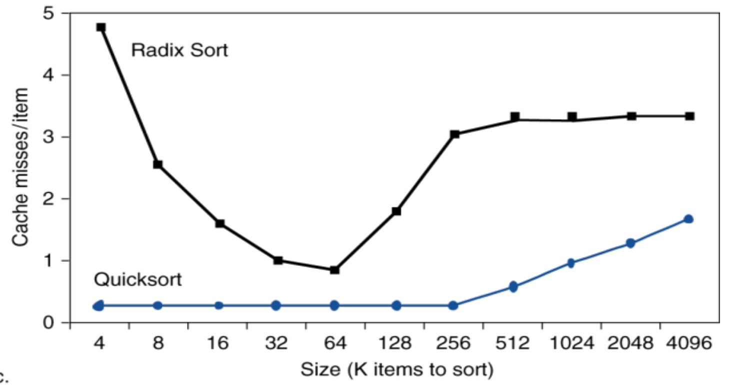
Interactions with Software



a.



b.

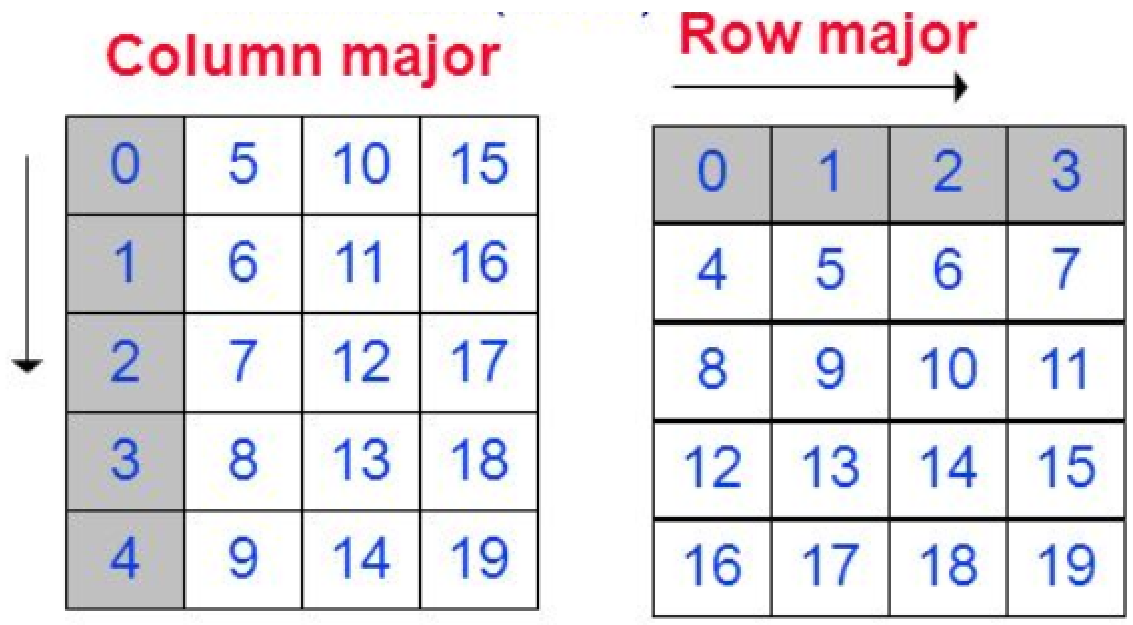


c.

- Misses depend on memory access patterns
  - Algorithm behaviour
  - Compiler optimization for memory access

Matrix storage

- String two-dimensional date in one dimensional (linear) memory



Column major

0	5	10	15
1	6	11	16
2	7	12	17
3	8	13	18
4	9	14	19

Row major

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19

FP Example: Array Multiplication

- MIPS code:

```
   li   $t1, 32       # $t1 = 32 (row size/loop end)
   li   $s0, 0        # i = 0; initialize 1st for loop
L1: li   $s1, 0        # j = 0; restart 2nd for loop
L2: li   $s2, 0        # k = 0; restart 3rd for loop
   sll  $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
   addu $t2, $t2, $s1 # $t2 = i * size(row) + j
   sll  $t2, $t2, 3   # $t2 = byte offset of [i][j]
   addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
   l.d  $f4, 0($t2)   # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5   # $t0 = k * 32 (size of row of z)
   addu $t0, $t0, $s1 # $t0 = k * size(row) + j
   sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
   addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
   l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
   …
```

FP Example: Array Multiplication

```
…
sll  $t0, $s0, 5        # $t0 = i*32 (size of row of y)
addu  $t0, $t0, $s2     # $t0 = i*size(row) + k
sll   $t0, $t0, 3       # $t0 = byte offset of [i][k]
addu  $t0, $a1, $t0     # $t0 = byte address of y[i][k]
l.d   $f18, 0($t0)      # $f18 = 8 bytes of y[i][k]
mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
addiu $s2, $s2, 1       # $k k + 1
bne   $s2, $t1, L3      # if (k != 32) go to L3
s.d   $f4, 0($t2)       # x[i][j] = $f4
addiu $s1, $s1, 1       # $j = j + 1
bne   $s1, $t1, L2      # if (j != 32) go to L2
addiu $s0, $s0, 1       # $i = i + 1
bne   $s0, $t1, L1      # if (i != 32) go to L1
```

Reading

- Sections 5.4