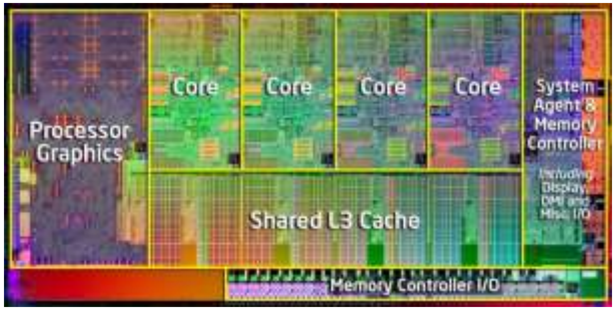



COSE222 Computer Architecture

# Lecture 6. Cache #2

*Prof. Taeweon Suh*

*Computer Science & Engineering*

*Korea University*

# Performance

- Execution time is composed of CPU execution cycles and memory stall cycles

- Assuming that cache hit does not require any extra CPU cycle for execution (that is, the MA stage takes 1 cycle upon a hit),

CPU time = #insts × CPI × T
= clock cycles x T
= (**CPU execution clock cycles** + **Memory-stall clock cycles**) x T

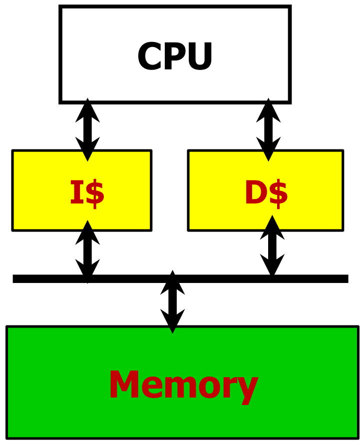
- Memory stall cycles come from cache misses

**Memory stall clock cycles** = Read-stall cycles + Write-stall cycles

- For simplicity, assume that a read miss and a write miss incur the same miss penalty

**Memory stall clock cycles** = (memory accesses/program) x (miss rate) x (miss penalty)

# Impacts of Misses on Performance




- A processor with
  - I-cache miss rate = 2% (**98% hit rate**)
  - Perfect D-cache (**100% hit rate**)
  - Miss penalty = 100 cycles
  - Base CPI with an ideal cache = 1

- What is the execution cycle on the processor?

CPU time = clock cycles x T

= (**CPU execution clock cycles** + **Memory-stall clock cycles**) x T

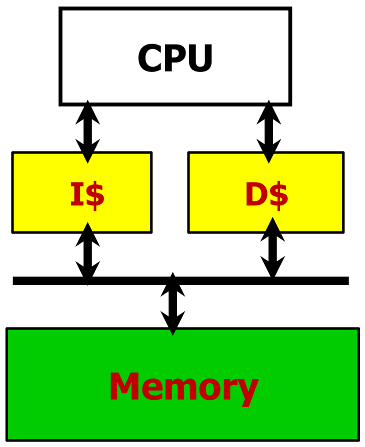
**Memory stall clock cycles** = (memory accesses/program) x (miss rate) x (miss penalty)

  - Assume that # of instructions executed is $n$
  - **CPU execution clock cycles** = $n$
  - **Memory stall clock cycles**
    - I$: $n \times 0.02 \times 100 = 2n$
  - Total execution cycles = $n + 2n = 3n$

**3X slower even with 98% hit rate, compared with ideal case!**

# Impacts of Misses on Performance

- A processor with
  - I-cache miss rate = 2% (**98% hit rate**)
  - D-cache miss rate = 4% (**96% hit rate**)
    - Loads (`lw`) or stores (`sw`) are 36% of instructions
  - Miss penalty = 100 cycles
  - Base CPI with an ideal cache = 1




- What is the execution cycle on the processor?

  CPU time = clock cycles x T

  = (**CPU execution clock cycles** + **Memory-stall clock cycles**) x T

  **Memory stall clock cycles** = (memory accesses/program) x (miss rate) x (miss penalty)

  - Assume that # of instructions executed is `n`
  - **CPU execution clock cycles** = `n`
  - **Memory stall clock cycles**
    - I$: `n` x 0.02 × 100 = 2`n`
    - D$: (`n` x 0.36) × 0.04 × 100 = 1.44`n`
    - Total memory stall cycles = 2`n` + 1.44`n` = 3.44`n`
  - Total execution cycles = `n` + 3.44`n` = 4.44`n`

# Average Memory Access Time (AMAT)

- Hit time is also important for performance
  - Hit time is not always less than 1 CPU cycle
  - For example, the hit latency of Core 2 Duo's L1 cache is 3 cycles!

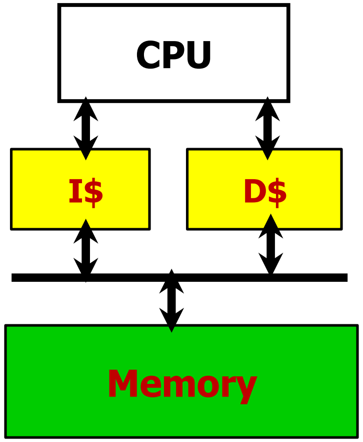
- To capture the fact that the time to access data for both hits and misses affect performance, designers sometimes use average memory access time (AMAT), which is the average time to access memory considering both hits and misses

AMAT = Hit time + Miss rate x Miss penalty

- Example
  - CPU with 1ns clock (1GHz)
  - Hit time = 1 cycle
  - Miss penalty = 20 cycles
  - Cache miss rate = 5%

**AMAT = 1 + 0.05 × 20 = 2ns**

**(2 cycles for each memory access)**



# Improving Cache Performance

- How to increase the cache performance?
  - Simply increase the cache size?
    - Surely, it would improve hit rate
    - But, a larger cache will have a longer access time
    - At some point, the longer access time will beat the improvement in hit rate, leading to a decrease in performance

- We'll explore two different techniques

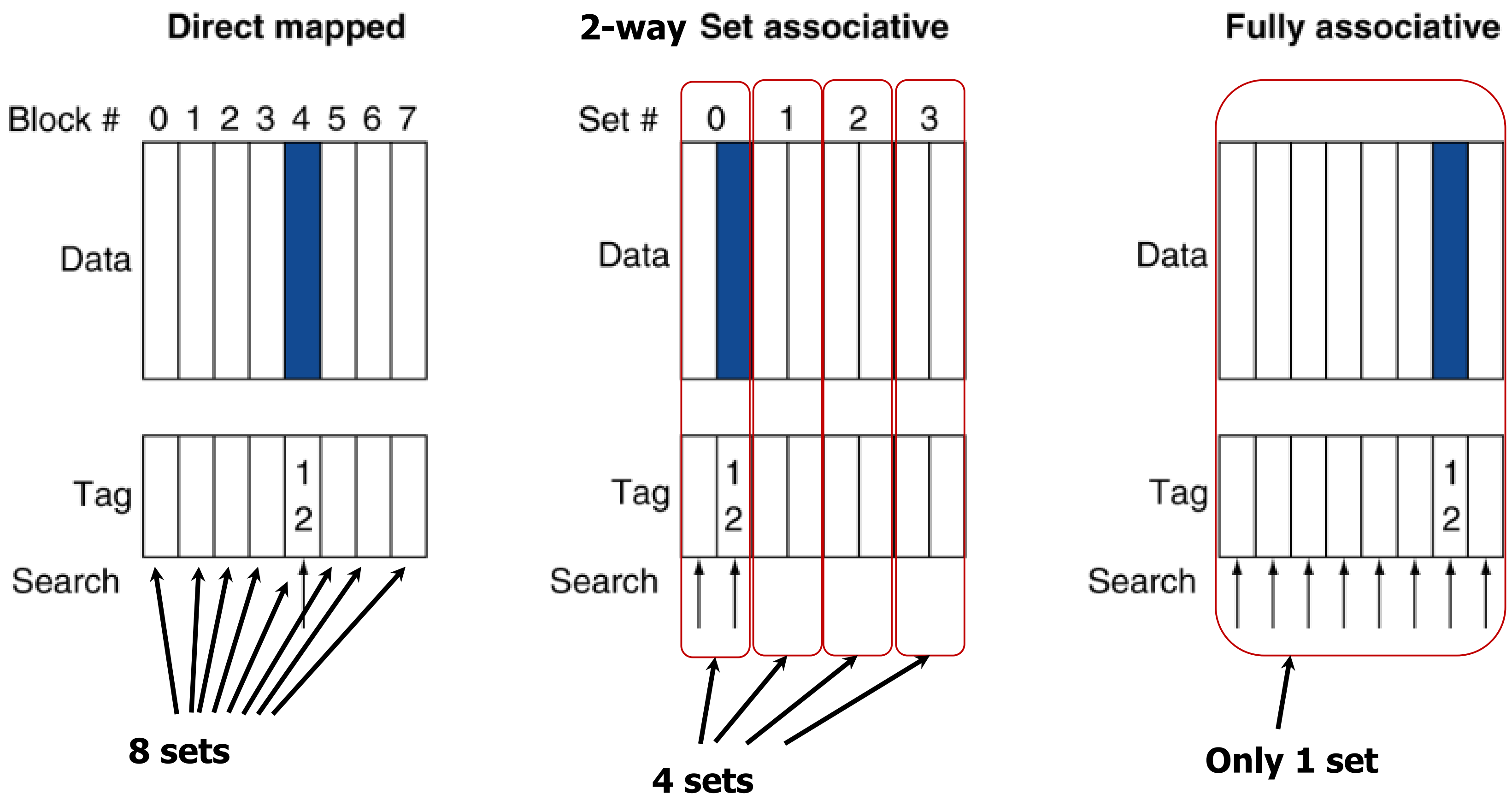
$$\text{AMAT} = \text{Hit time} + \textbf{Miss rate} \times \textbf{Miss penalty}$$

  - The first technique reduces the **miss rate** by reducing the probability that different memory blocks will contend for the same cache location
  - The second technique reduces the **miss penalty** by adding additional cache levels to the memory hierarchy

# Reducing Cache Miss Rates (#1)

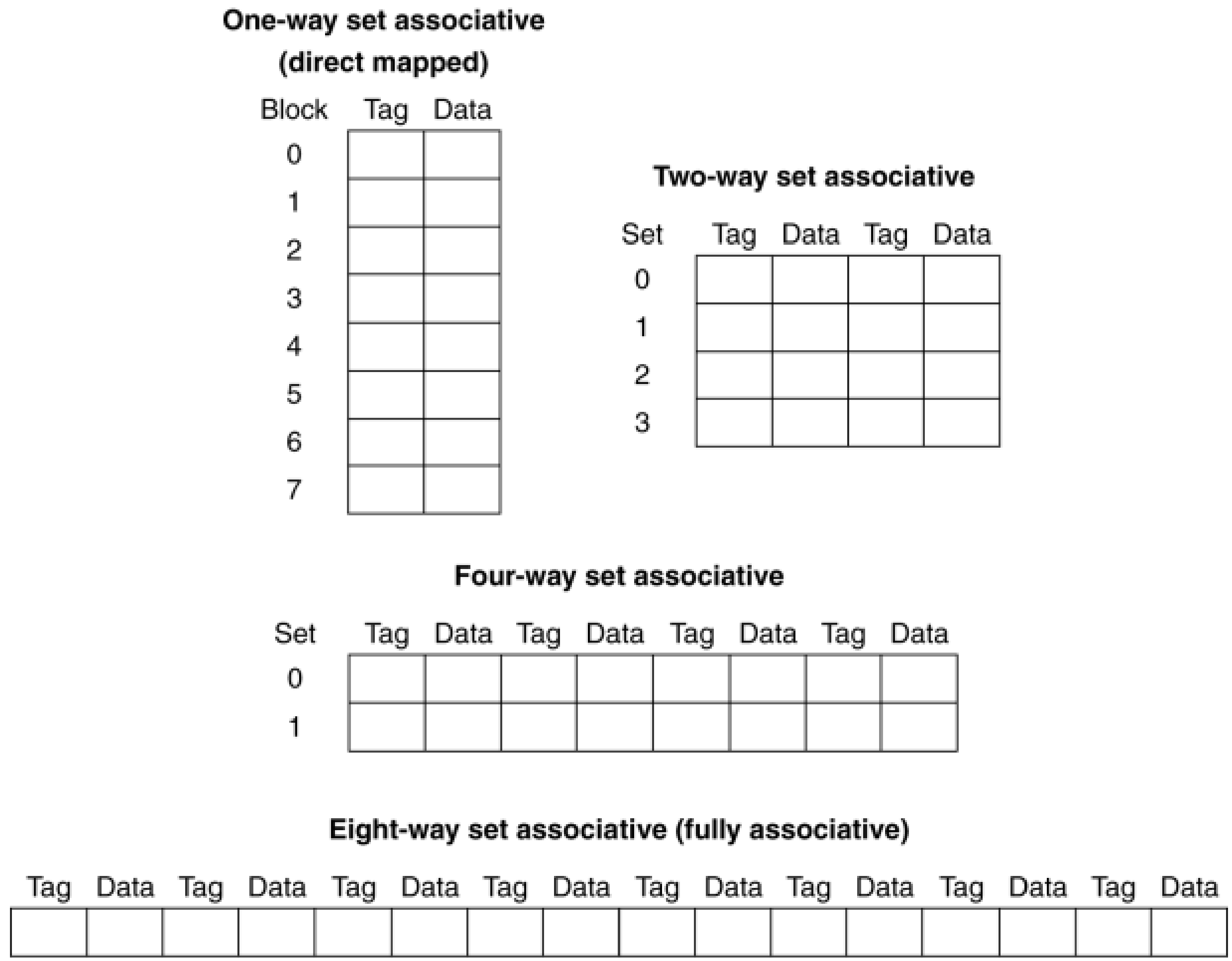
- Alternatives for more flexible block placement
  - Direct mapped cache maps a memory block to exactly one location in cache
  - Fully associative cache allows a memory block to be mapped to any cache block
  - n-way set associative cache divides the cache into sets, each of which consists of n different locations (ways)
    - A memory block is mapped to a unique set (specified by the index field) and can be placed in any way of that set (so, there are n choices)

# Associative Cache Example



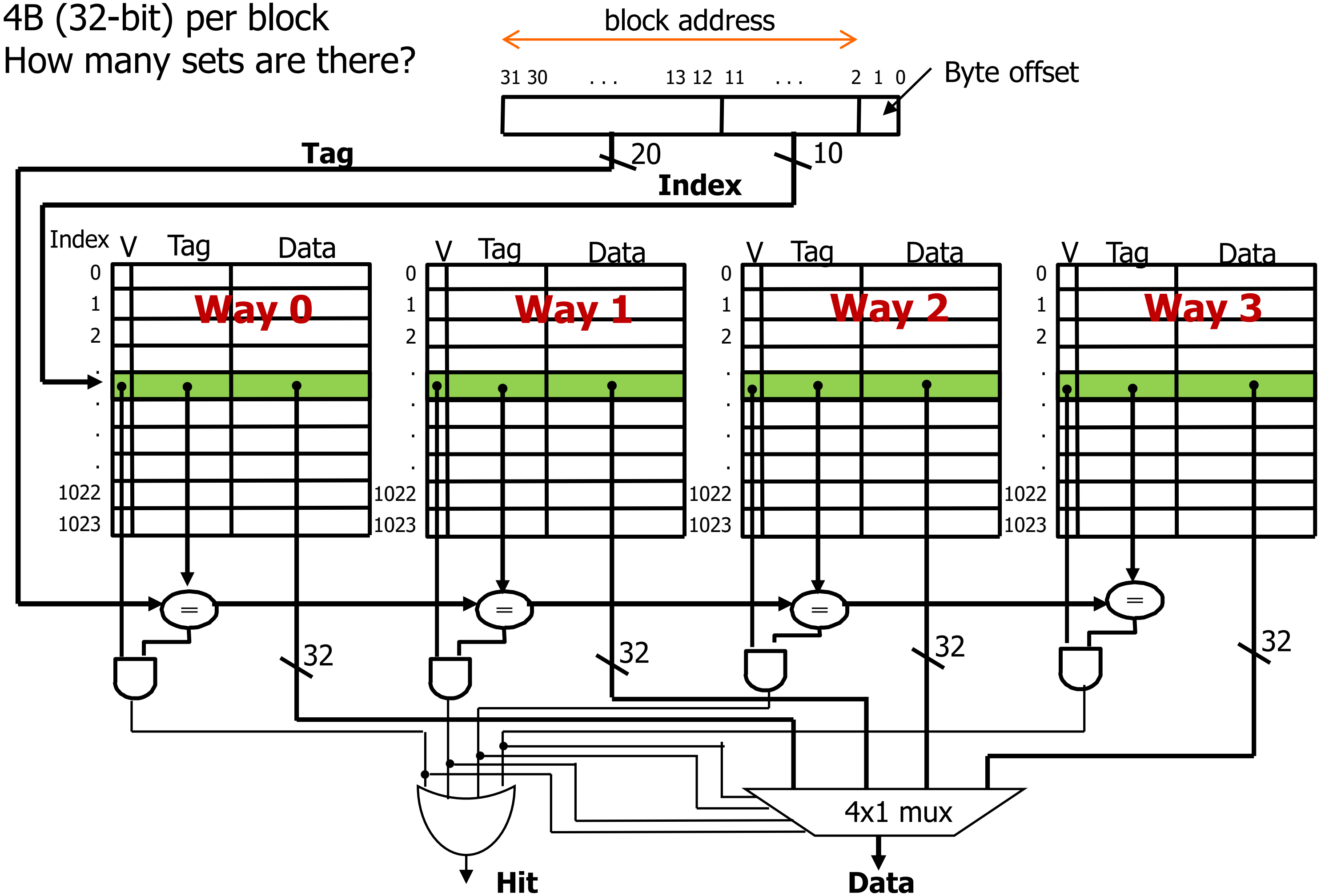
# Spectrum of Associativity

- For a cache with 8 entries



# 4-Way Set Associative Cache

- 16KB cache
  - 4-way set associative
  - 4B (32-bit) per block
  - How many sets are there?



# Associative Caches

- **Fully associative**
  - Allow a memory block to go in any cache entry
  - Requires all cache entries to be searched
  - Comparator per entry (expensive)

- ***n*-way set associative**
  - Each set contains *n* entries
  - (Block address) % (#sets in cache) determines which set to search
  - Search all *n* entries in a indexed set
  - *n* comparators (less expensive)

# Associativity Example

- Assume that there are 3 small different caches with 4 one-word blocks, and CPU generates 8-bit address to cache
  - Direct mapped
  - 2-way set associative
  - Fully associative

- Find the number of misses for each cache organization given the following sequence of block addresses: 0, 8, 0, 6, 8
  - We assume that the sequence is for all memory reads (`lw`)

# Associativity Example

- Direct mapped cache

block address

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| tag | | | | index | | | |

tag

0, 8, 0, 6, 8

| Byte address | Block address | Cache Index | Hit or Miss? |
|---|---|---|---|
| b'0000 0000 (0x00) | b'00 0000 (0x00) | 0 | miss |
| b'0010 0000 (0x20) | b'00 1000 (0x08) | 0 | miss |
| b'0000 0000 (0x00) | b'00 0000 (0x00) | 0 | miss |
| b'0001 1000 (0x18) | b'00 0110 (0x06) | 2 | miss |
| b'0010 0000 (0x20) | b'00 1000 (0x08) | 0 | miss |

| | index | V | Tag | Data |
|---|---|---|---|---|
| 8 | 0 | 1 | 0010 | Mem[8] |
| | 1 | 0 | | |
| 6 | 2 | 1 | 0001 | Mem[6] |
| | 3 | 0 | | |

# Associativity Example (Cont)

- 2-way set associative

block address

| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|
| tag | | | | | index | | |

tag

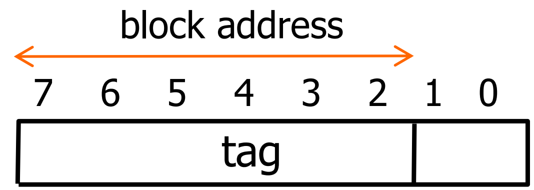
0, 8, 0, 6, 8

| Byte address | Block address | Cache Index | Hit or Miss? |
|---|---|---|---|
| b’0000 0000 (0x00) | b’00 0000 (0x00) | 0 | miss |
| b’0010 0000 (0x20) | b’00 1000 (0x08) | 0 | miss |
| b’0000 0000 (0x00) | b’00 0000 (0x00) | 0 | hit |
| b’0001 1000 (0x18) | b’00 0110 (0x06) | 0 | miss |
| b’0010 0000 (0x20) | b’00 1000 (0x08) | 0 | miss |

8

| index | V | Tag | Data | V | Tag | Data |
|---|---|---|---|---|---|---|
| 0 | 1 | 00100 | Mem[8] | 1 | 00011 | Mem[6] |
| 1 | 0 | | | 0 | | |
| | way #0 | | | way #1 | | |

# Associativity Example (Cont)

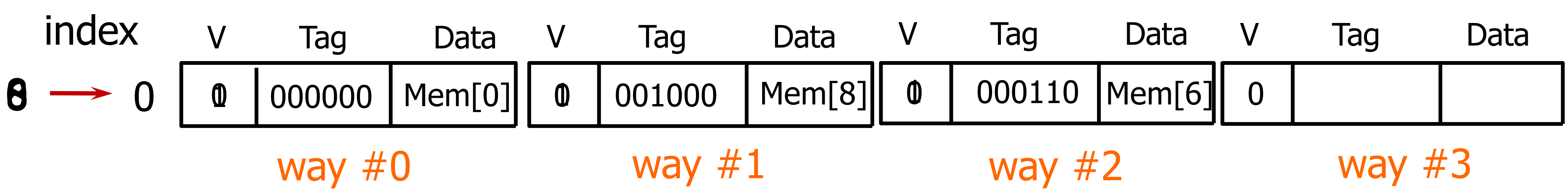



- Fully associative

0, 8, 0, 6, 8

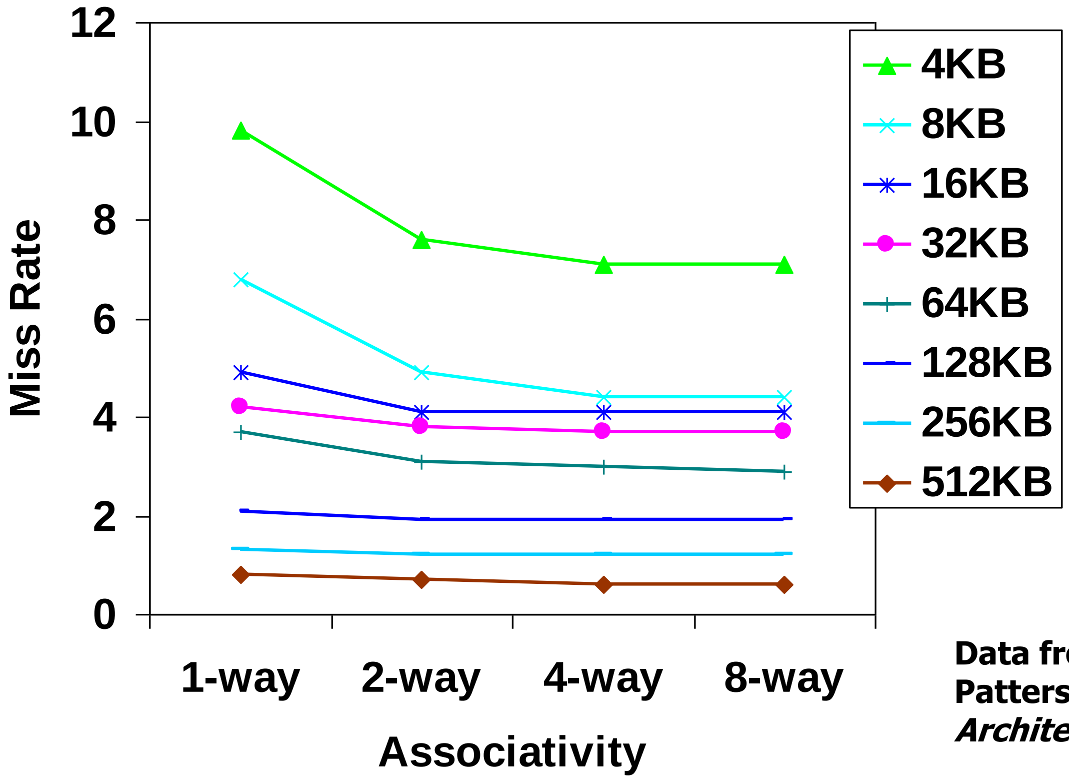
tag

| Byte address | Block address | Cache Index | Hit or Miss? |
|---|---|---|---|
| b’0000 0000 (0x00) | b’00 0000 (0x00) | 0 | miss |
| b’0010 0000 (0x20) | b’00 1000 (0x08) | 0 | miss |
| b’0000 0000 (0x00) | b’00 0000 (0x00) | 0 | hit |
| b’0001 1000 (0x18) | b’00 0110 (0x06) | 0 | miss |
| b’0010 0000 (0x20) | b’00 1000 (0x08) | 0 | hit |



# Benefits of Set Associative Caches

- The choice of direct mapped or set associative depends on the cost of a miss versus the cost of implementation

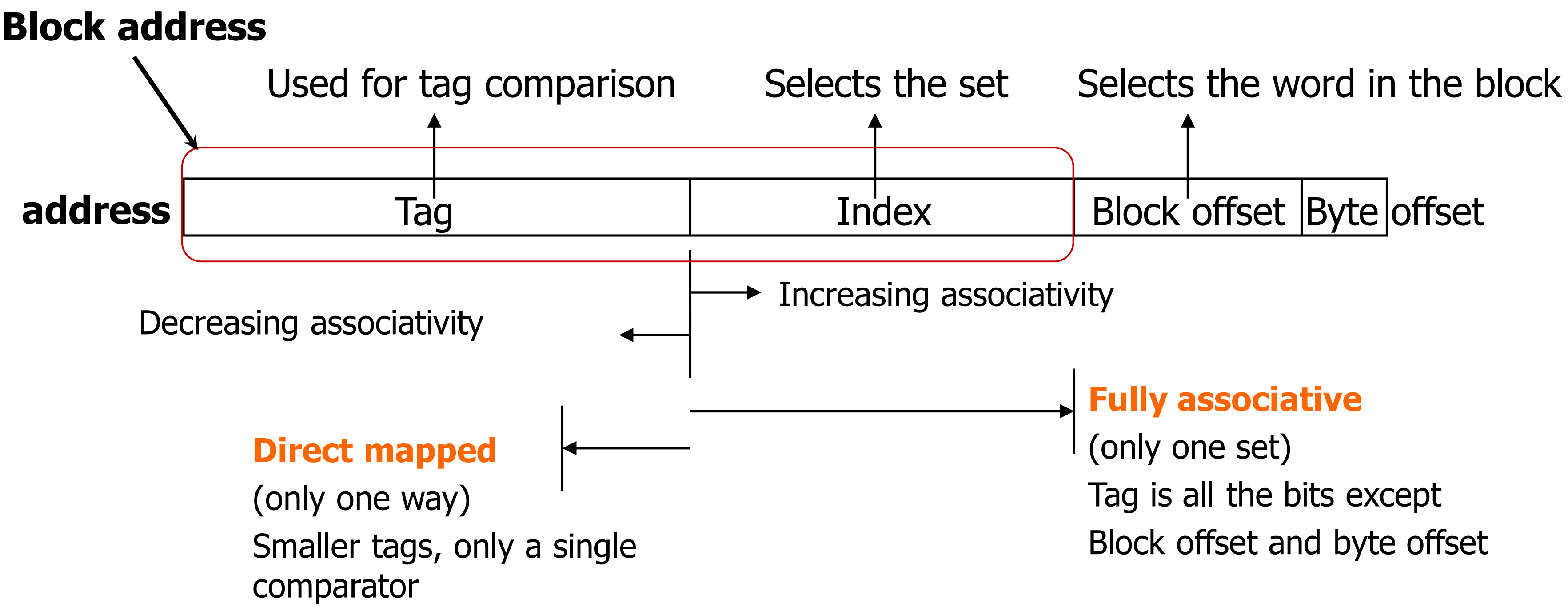



**Data from Hennessy & Patterson, *Computer Architecture*, 2003**

**Largest gains are achieved when going from direct mapped to 2-way (more than 20% reduction in miss rate)**

# Range of Set Associative Caches

- For a fixed size cache, the increase by a factor of 2 in associativity doubles the number of blocks per set (the number of ways) and halves the number of sets
  - Example: An increase of associativity from 4 ways to 8 ways decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

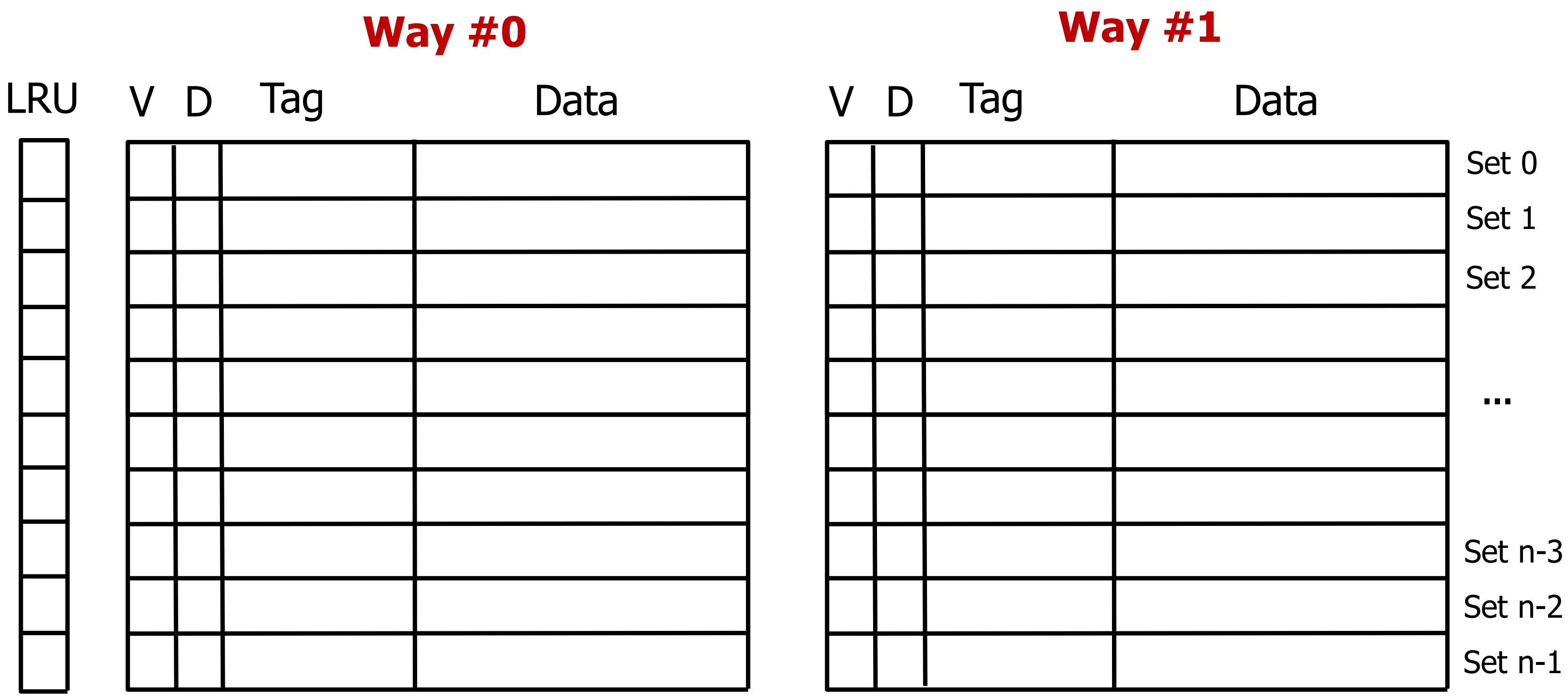


# Replacement Policy

- Upon a miss, which way's block should be picked for replacement?
  - Direct mapped cache has no choice
  - Set associative cache prefers 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
      - Hardware should keep track of when each way's block was used relative to the other blocks in the set
      - Simple for 2-way (takes one bit per set), manageable for 4-way, too hard beyond that
    - Random
      - Gives approximately the same performance as LRU with high-associativity cache

# Example Cache Structure with LRU

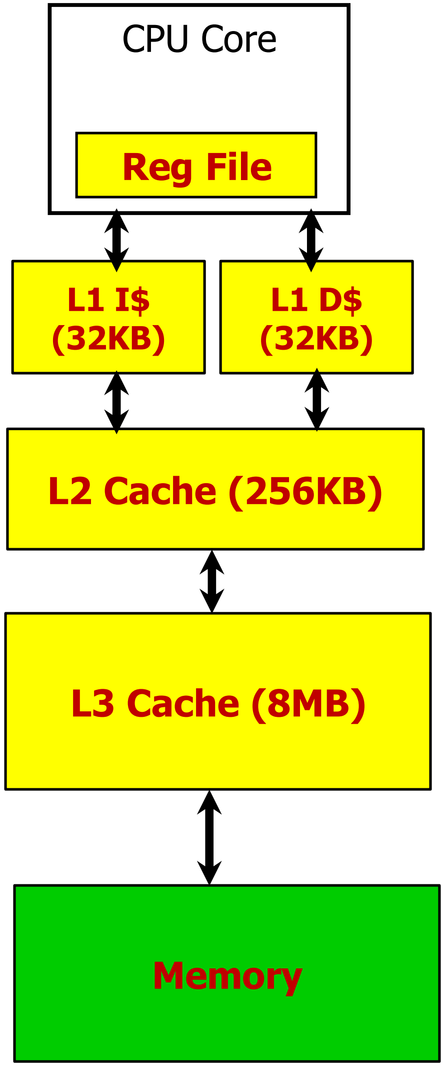
- 2-way set associative **data** cache



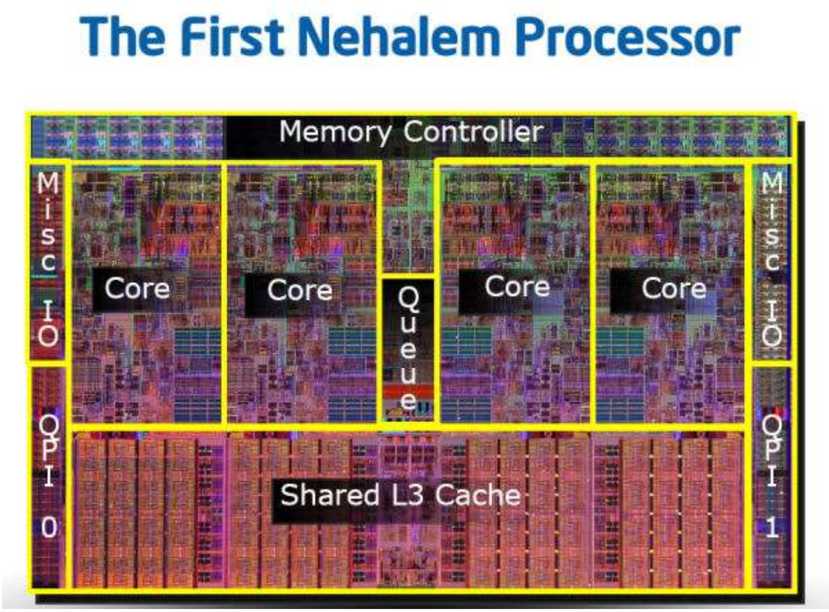
# Reducing Cache Miss Penalty (#2)

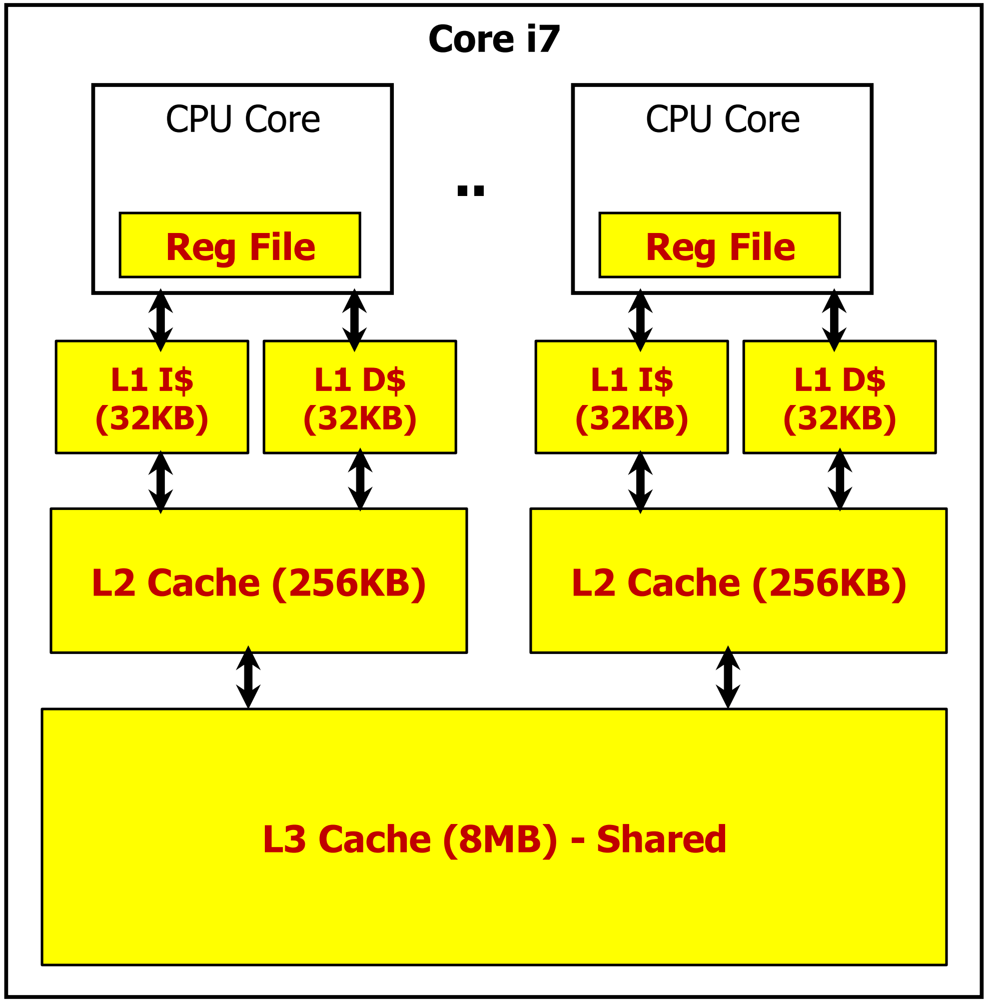
- Use multiple levels of caches
  - Primary cache (Level-1 or L1) attached to CPU
    - Small, but fast
  - Level-2 (L2) cache services misses from primary cache
    - Larger, slower, but faster than L3
  - Level-3 (L3) cache services misses from L2 cache
    - Largest, slowest, but faster than main memory
  - Main memory services L3 cache misses

- Advancement in semiconductor technology allows enough room on the die for L1, L2, and, L3 caches
  - L2 and L3 caches are typically unified, meaning that it holds both instructions and data



# Core i7 Example



# Multilevel Cache Performance Example

- Given
  - Base CPI = 1 (Ideal case)
  - Clock frequency = 4GHz
  - Main memory access time = 100ns
- With just L1 cache
  - L1 $ access time = 0.25 ns (1 cycle)
  - L1 miss penalty
    - 100ns/0.25ns = 400 cycles
  - Miss rate/instruction = 2%
  - CPI
    - $1 + 0.02 \times 400 =$ **9 CPI**
      (= 98% x 1 CPI + 2% x (1 + 400))
- Now add L2 cache
  - L2 $ access time = 5ns
  - Global miss rate to main memory = 0.5%
    (i.e., L2 miss rate is 25% (= 0.5% / 2% x 100)
- L1 miss with L2 hit
  - L1 miss penalty = 5ns/0.25ns = 20 cycles
- L1 miss with L2 miss
  - Extra penalty = 400 cycles
- CPI
  - $1 + 0.02 \times 20 + 0.005 \times 400 =$ **3.4 CPI**
    (= 1 + 0.02 x (75% x 20 + 25% x (20+400)))

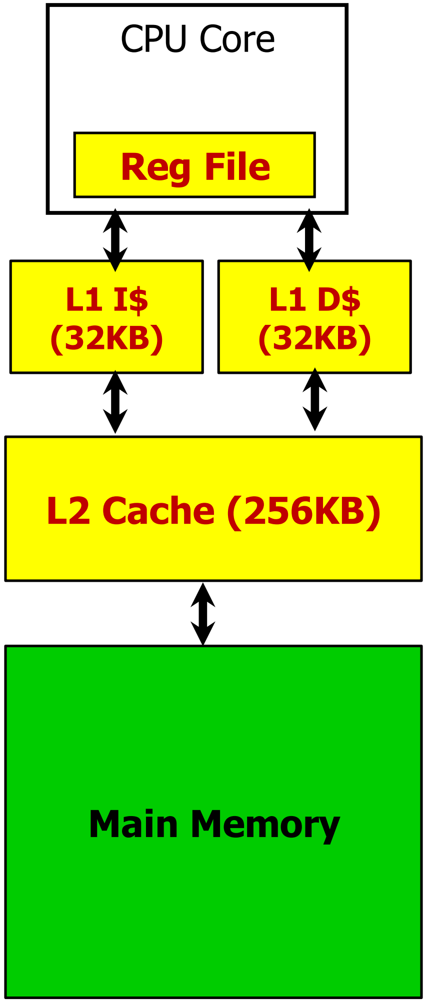
**Performance ratio = 9/3.4 = 2.6**

**Addition of L2 cache in this example achieves 2.6x speedup**

# Multilevel Cache Design Considerations

AMAT = Hit time + **Miss rate** x **Miss penalty**

- Design considerations for L1 and L2 caches are very different
  - L1 should focus on minimizing hit time in support of a shorter clock cycle
    - L1 is smaller (i.e., faster) but have higher miss rate
  - L2 (L3) should focus on reducing miss rate and reducing L1 miss penalty
    - High associativity and large cache size
    - For the L2 (L3) cache, hit time is less important than miss rate
    - Miss penalty of L1 cache is significantly reduced by the presence of L2 and L3 caches






Korea Univ

# Two Machines' Cache Parameters

| | Intel Nehalem | AMD Barcelona |
|---|---|---|
| **L1 cache** organization & size | Split I$ and D$; 32KB for each per core; 64B blocks | Split I$ and D$; 64KB for each per core; 64B blocks |
| L1 associativity | **4-way** (I), **8-way** (D) set assoc.; ~LRU replacement | **2-way** set assoc.; LRU replacement |
| L1 write policy | write-back, write-allocate | write-back, write-allocate |
| **L2 cache** organization & size | Unified; 256KB (0.25MB) per core; 64B blocks | Unified; 512KB (0.5MB) per core; 64B blocks |
| L2 associativity | **8-way** set assoc.; ~LRU | **16-way** set assoc.; ~LRU |
| L2 write policy | write-back | write-back |
| L2 write policy | write-back, write-allocate | write-back, write-allocate |
| **L3 cache** organization & size | Unified; 8192KB (8MB) shared by cores; 64B blocks | Unified; 2048KB (2MB) shared by cores; 64B blocks |
| L3 associativity | **16-way** set assoc. | **32-way** set assoc.; evict block shared by fewest cores |
| L3 write policy | write-back, write-allocate | write-back; write-allocate |

# Software Techniques

- Misses occur if sequentially accessed array elements come from different cache lines
  - To avoid the cache misses, try to exploit the cache structure in programming to reduce the miss rate

- Code optimizations (No hardware change!)
  - Rely on programmers or compilers
  - **Loop interchange**
    - In nested loops, outer loop becomes inner loop and vice versa
  - **Loop blocking**
    - Partition large array into smaller blocks, thus fitting the accessed array elements into cache size
    - Enhance the reuse of cache blocks

# Cache-friendly Code

- In **C**, a 2-dimensional array is stored according to **row-major ordering**
  - Example: `int A[10][8]`

**memory**

| |
|---|
| … |
| A[2][1] |
| A[2][0] |
| A[1][7] |
| A[1][6] |
| A[1][5] |
| A[1][4] |
| A[1][3] |
| A[1][2] |
| A[1][1] |
| A[1][0] |
| A[0][7] |
| A[0][6] |
| A[0][5] |
| A[0][4] |
| A[0][3] |
| A[0][2] |
| A[0][1] |
| A[0][0] |
| … |

**Data cache**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … | … |
| A[3][7] | A[3][6] | A[3][5] | A[3][4] | A[3][3] | A[3][2] | A[3][1] | A[3][0] |
| A[2][7] | A[2][6] | A[2][5] | A[2][4] | A[2][3] | A[2][0] | A[2][1] | A[2][0] |
| A[1][7] | A[1][6] | A[1][5] | A[1][4] | A[1][3] | A[1][2] | A[1][1] | A[1][0] |
| A[0][7] | A[0][6] | A[0][5] | A[0][4] | A[0][3] | A[0][2] | A[0][1] | A[0][0] |

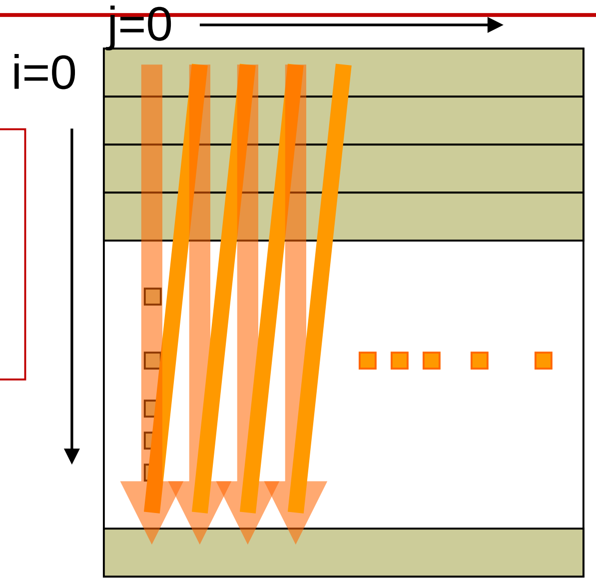
A Cache Line (block)

```
for (i=0 ; i<n; i++)
    for (j=0 ; j<n; j++)
        sum += a[i][j];
```

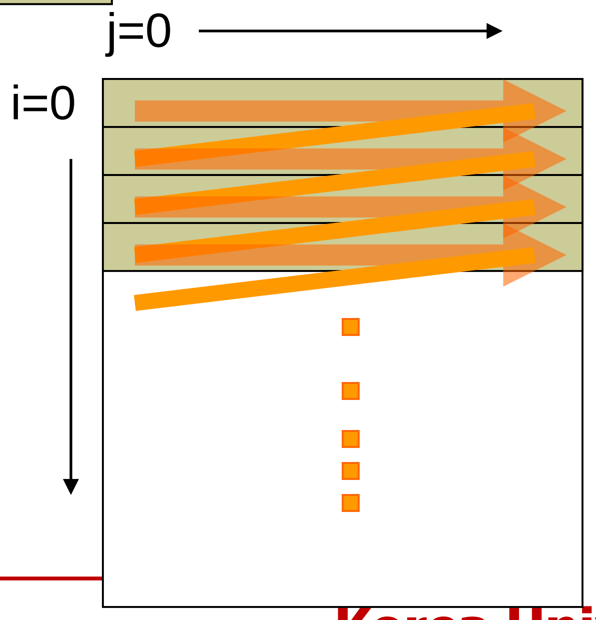
# Loop Interchange

**Row-major ordering**

```
/* Before */
for (j=0; j<100; j++)
 for (i=0; i<5000; i++)
   x[i][j] = 2*x[i][j]
```




**What is the worst that could happen?**




```
/* After */
for (i=0; i<5000; i++)
 for (j=0; j<100; j++)
   x[i][j] = 2*x[i][j]
```

**Improved cache efficiency**

# Example (Loop Interchange)

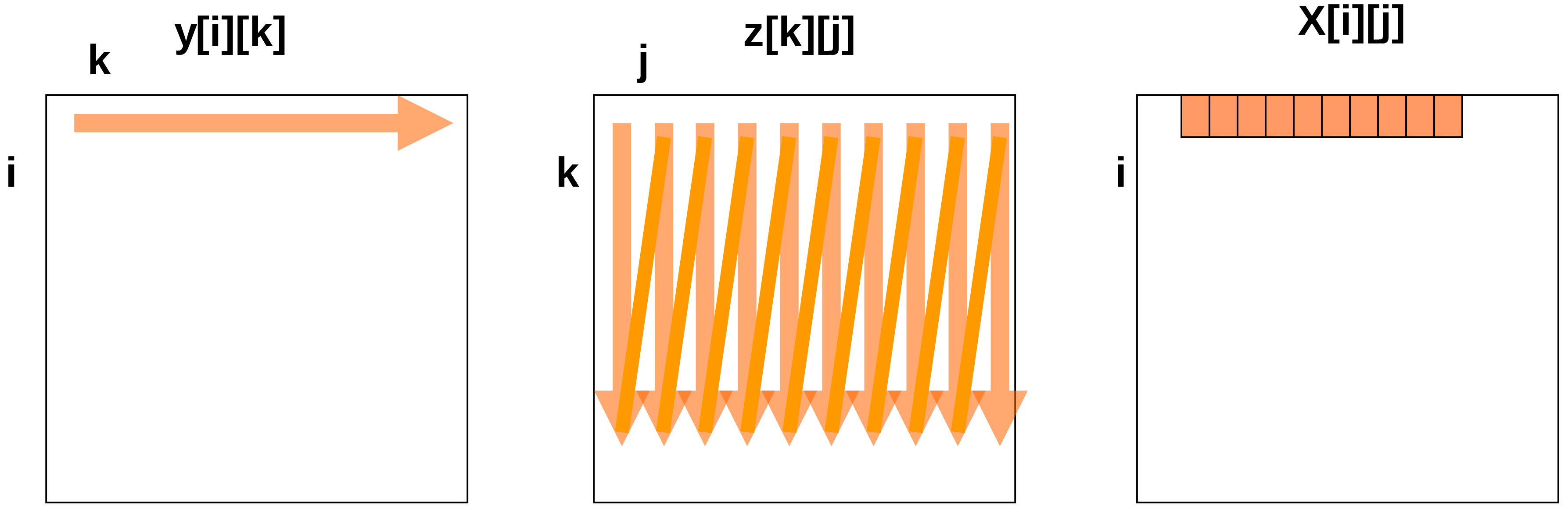
- Run the program on Ubuntu
  - To compile the program, use `gcc`
    - `%gcc loop_interchange.c -o loop_interchange`
  - To measure the execution time, use the `time` utility
    - `%time ./loop_interchange`
  - To collect cache statistics, use the `perf` utility
    - `%perf stat ./loop_interchange`
    - `%perf stat -e cache-misses ./loop_interchange`
  - Repeat the above steps by exchanging the `for` loops in red

```
#include <stdio.h>
#include <stdlib.h>

#define n 1024  // 1K
#define m 1024  // 1K

int main()
{
   int  i, j, count;
   long long int ** a; // 2-dimential array
   long long int sum;

   for (i=0; i<n ; i++)
         a = malloc(n*sizeof(long long int *));

   for (i=0; i<m ; i++)
         a[i] = (long long int *) malloc(m*sizeof(long long int));


   for (i=0; i<n; i++)
     for (j=0; j<m; j++)
        a[i][j] = i+j ; // initialization with some numbers

   sum = 0;

   for (count=0; count <1000; count++) {

      for (j=0; j<m; j++)
        for (i=0; i<n; i++)
            sum += a[i][j] ;

   }

   printf("sum is %lld, 0x%llx\n", sum, sum);

   return 0;
}
```

# Backup

# Why Loop Blocking?

```
/* Before */
for (i=0; i<N; i++)
 for (j=0; j<N; j++)
   for (k=0; k<N; k++)
      x[i][j] = y[i][k]*z[k][j];
```

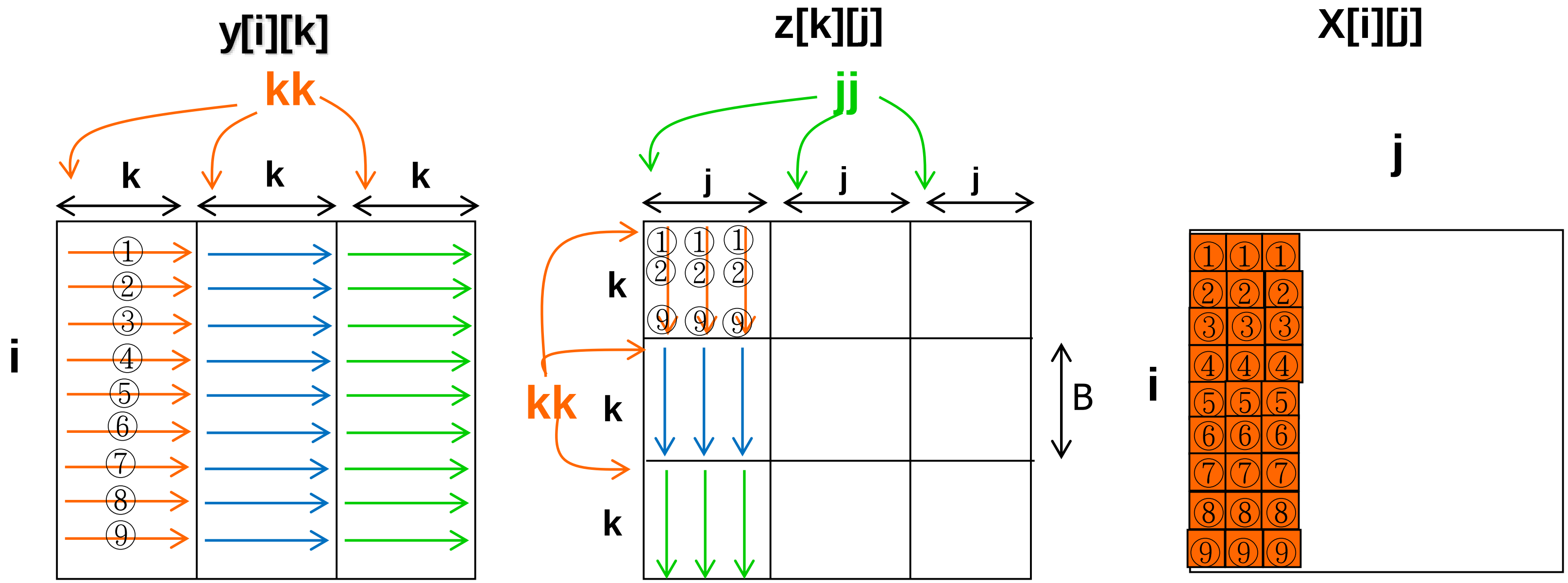



**Does not exploit locality!**

# Loop Blocking (Loop Tilting)

- Partition the loop's iteration space into many smaller chunks and ensure that the data stays in the cache until it is reused

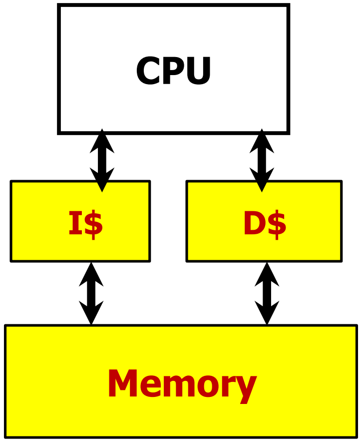
```
/* After */
for (jj=0; jj<N; jj=jj+B)   // B: blocking factor
for (kk=0; kk<N; kk=kk+B)
for (i=0; i<N; i++)
   for (j=jj; j< min(jj+B,N); j++)
     for (k=kk; k< min(kk+B,N); k++)
          x[i][j] += y[i][k]*z[k][j];
```



# Two Machines' Cache Parameters

| | Intel P4 | AMD Opteron |
|---|---|---|
| **L1 organization** | Split I$ and D$ | Split I$ and D$ |
| L1 cache size | 8KB for D$, 96KB for trace cache (~I$) | 64KB for each of I$ and D$ |
| L1 block size | 64 bytes | 64 bytes |
| L1 associativity | 4-way set assoc. | 2-way set assoc. |
| L1 replacement | ~ LRU | LRU |
| L1 write policy | write-through | write-back |
| **L2 organization** | Unified | Unified |
| L2 cache size | 512KB | 1024KB (1MB) |
| L2 block size | 128 bytes | 64 bytes |
| L2 associativity | 8-way set assoc. | 16-way set assoc. |
| L2 replacement | ~LRU | ~LRU |
| L2 write policy | write-back | write-back |

# Impacts of Cache Performance




- A processor with
  - I-cache miss rate = 2%
  - D-cache miss rate = 4%
    - Loads (`lw`) or stores (`sw`) are 36% of instructions
  - Miss penalty = 100 cycles
  - Base CPI with an ideal cache = 2
- What is the execution cycle on the processor?

CPU time = CPU clock cycles x T

= (**CPU execution clock cycles** + **Memory-stall clock cycles**) x T

**Memory stall clock cycles** = (memory accesses/program) x (miss rate) x (miss penalty)

  - Assume that # of instructions executed is I
  - **CPU execution clock cycles** = 2I
  - **Memory stall clock cycles**
    - I$: I x 0.02 × 100 = 2I
    - D$: (I x 0.36) × 0.04 × 100 = 1.44I
    - Total memory stall cycles = 2I + 1.44I = 3.44I
  - Total execution cycles = 2I + 3.44I = 5.44I

# Impacts of Cache Performance (Cont)

- How much faster a processor with a perfect cache?
  - Execution cycle time of the CPU with a perfect cache = 2I
  - Thus, the CPU with the perfect cache is 2.72x (=5.44I/2I) faster!

- If you design a new computer with CPI = 1, how much is the system with the perfect cache faster?
  - Total execution cycles = 1I + 3.44I = 4.44I
  - Therefore, the CPU with the perfect cache is 4.44x faster!
  - Then, the amount of execution time spent on memory stalls have risen from 63% (=3.44/5.44) to 77% (= 3.44/4.44)

- Thoughts
  - 2% of instruction cache misses with the miss penalty of 100 cycle increases CPI by 2!
  - Data cache misses also increase CPI dramatically
  - **As CPU is made faster, the amount of time spent on memory stalls will take up an increasing fraction of the execution time**

# Example (Loop Interchange)

- Run the program on Ubuntu
  - To compile the program, use `gcc`
    - `%gcc loop_interchange.c -o loop_interchange`
  - To measure the execution time, use the `time` utility
    - `%time ./loop_interchange`
  - To collect cache statistics, use the `perf` utility
    - `%perf stat -e LLC-load-misses -e LLC-store-misses ./loop_interchange`
  - Repeat the above steps by exchanging the `for` loops in red

```c
#include <stdio.h>

#define n 1024  // 1K
#define m 1024  // 1K

int main()
{
   int  i, j, count;
   long long int a[n][m];
   long long int sum;

   for (i=0; i<n; i++)
     for (j=0; j<m; j++)
        a[i][j] = i+j ;

   sum = 0;

   for (count=0; count <1000; count++) {

      for (j=0; j<m; j++)
        for (i=0; i<n; i++)
            sum += a[i][j] ;

   }

   printf("sum is %lld, 0x%llx\n", sum, sum);

   return 0;
}
```