Cache and Memory Hierarchy (2)

Lecture 9
November 8th, 2023

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

***Slide credits**: [COD:RV2e] slides from Elsevier Inc.*

Outline

Textbook: COD 5.4-5.5

- **Measuring and Improving Cache Performance**
  - Cache Performance
  - Associative Caches
  - Multilevel Caches
  - Software Optimization via Blocking
- **Dependable Memory Hierarchy**

Cache Performance (1)

- **Components of CPU time**
  - Program execution cycles
    - Includes cache hit time
  - Memory stall cycles
    - Mainly from cache misses
- **With simplifying assumptions:**

$$\text{Memory stall cycles}$$

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

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

Cache Performance (2): An 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 × 100 = 2
  - D-cache: 0.36 × 0.04 × 100 = 1.44
- **Actual CPI = 2 + 2 + 1.44 = 5.44**
  - Ideal CPU is 5.44/2 =2.72 times faster

Cache Performance (3): Average Access Time

- **Hit time is also important for performance**
- **Average memory access time (AMAT)**
  - AMAT = Hit time + Miss rate × Miss penalty
- **Example**
  - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - AMAT = 1 + 0.05 × 20 = 2ns
    - 2 cycles per instruction

Cache Performance (4): Summary

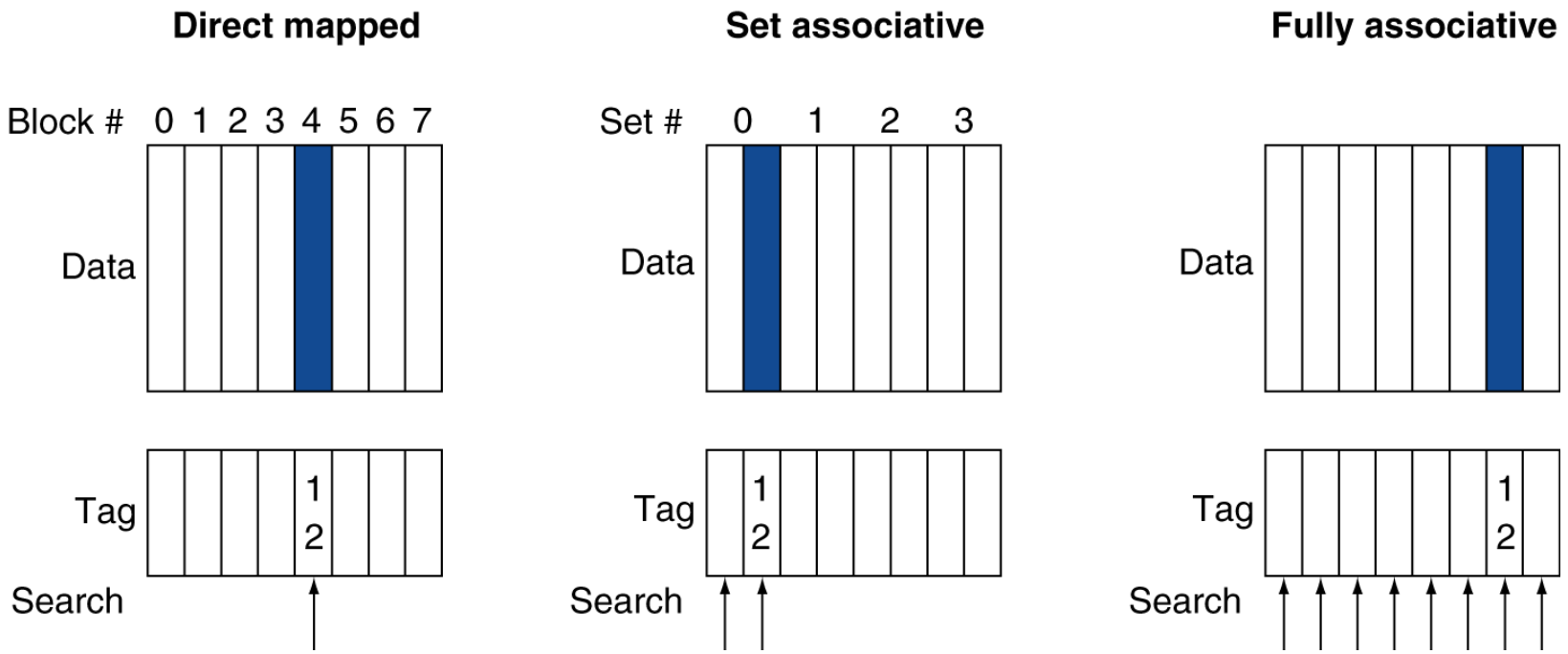
- **When CPU performance increased**
  - Miss penalty becomes more significant
- **Decreasing base CPI**
  - Greater proportion of time spent on memory stalls
- **Increasing clock rate**
  - Memory stalls account for more CPU cycles
- **Can’t neglect cache behavior when evaluating system performance**

Associative Caches (1)

- **Fully associative**
  - Allow 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 (# of sets in cache)
  - Search all entries in a given set at once
  - *n* comparators (less expensive)

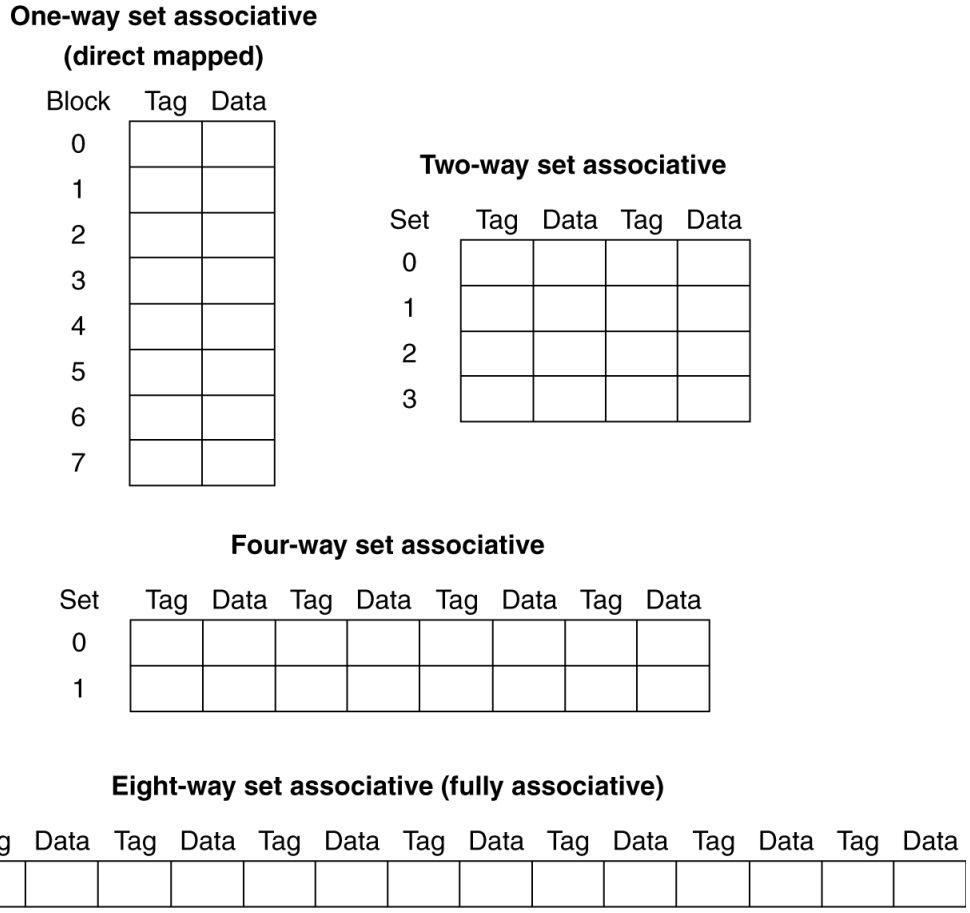
Associative Caches (2)

- **Example**



Associative Caches (3)

- **Spectrum of Associativity**
  - For a cache with 8 entries



Associative Caches (4): An Example

- **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	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]	

Associative Caches (5): An 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]		

- **Fully associative**

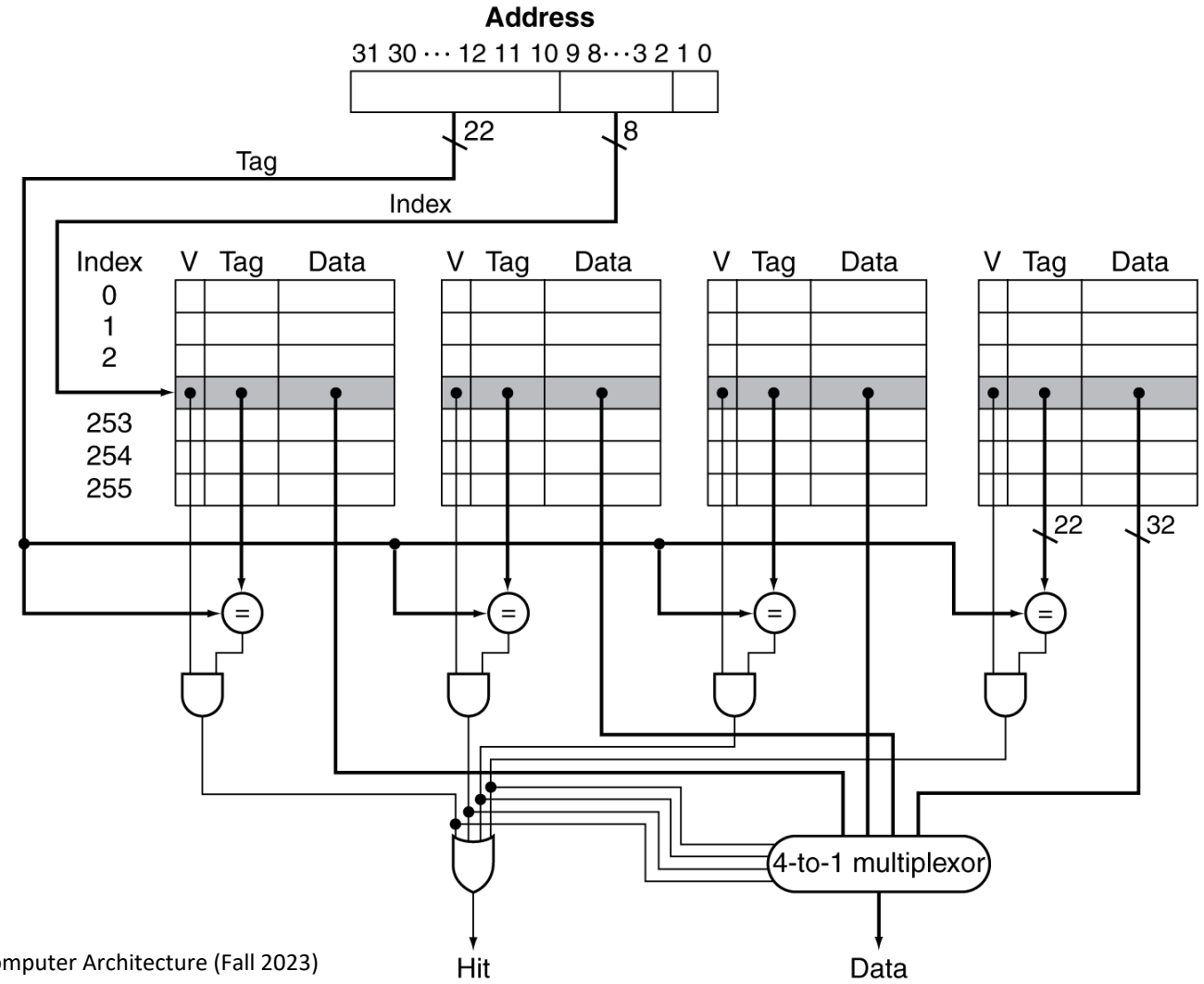
Block address		Hit/miss	Cache content after access			
0		miss	Mem[0]			
8		miss	Mem[0]	Mem[8]		
0		hit	Mem[0]	Mem[8]		
6		miss	Mem[0]	Mem[8]	Mem[6]	
8		hit	Mem[0]	Mem[8]	Mem[6]	

Associative Caches (6)

- **Question: 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%

Associative Caches (7)

- **Set associative cache organization**



Associative Caches (8)

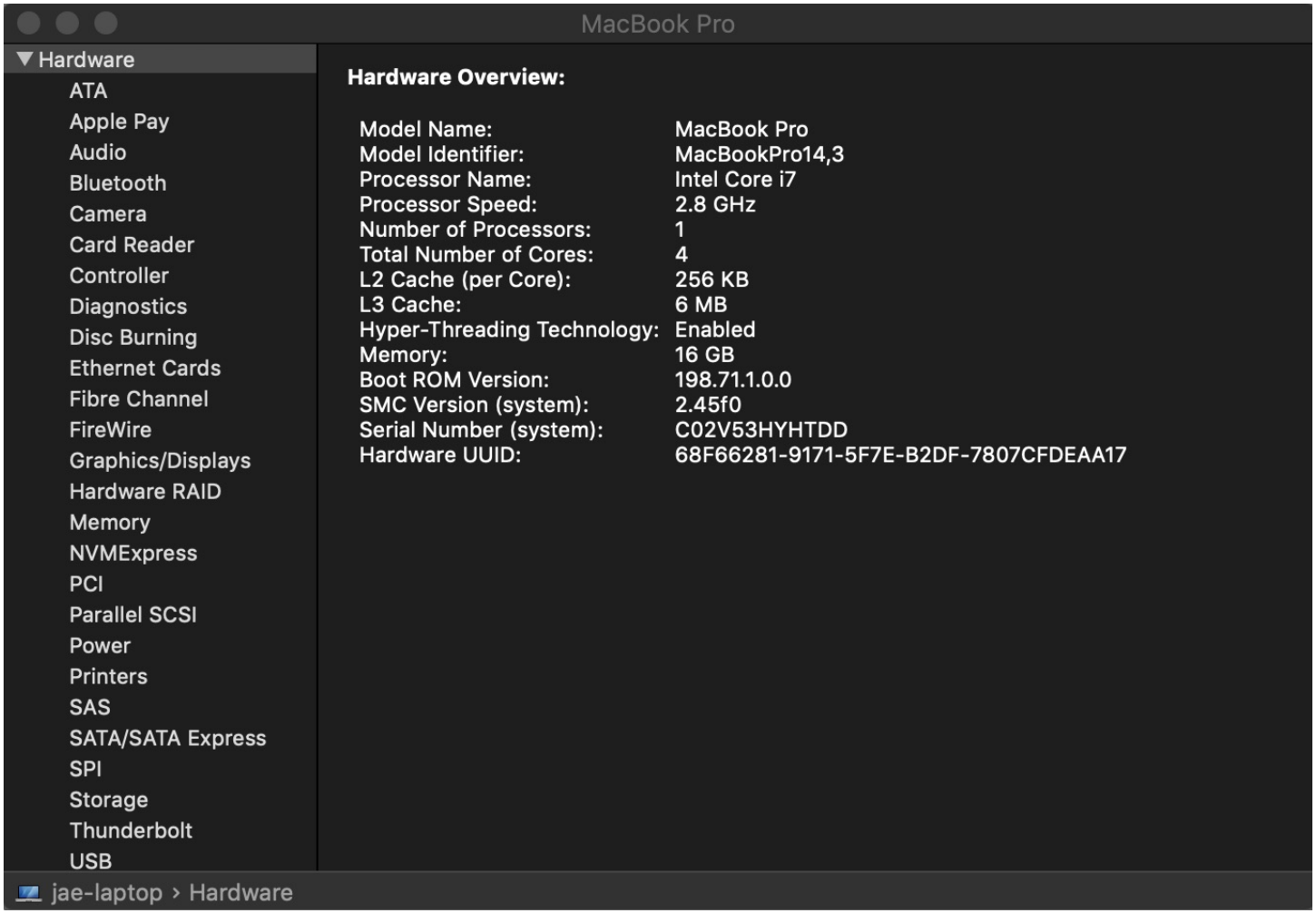
- **Replacement policy**
  - Direct mapped: no choice
  - Set associative
    - Prefer non-valid entry, if there is one
    - Otherwise, choose among entries in the set
  - Least-recently used (LRU)
    - Choose the one unused for the longest time
      - Simple for 2-way, manageable for 4-way, too hard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

Multilevel Caches (1)

- **Primary cache attached to CPU**
  - 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**
- **High-end systems include L3 cache**

Multilevel Caches (1)

- **Example: My laptop (Macbook Pro)**



Multilevel Caches (2)

- **Multilevel cache example**
  - 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 × 400 = 9

Multilevel Caches (3)

- **Example (cont.): 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 × 20 + 0.005 × 400 = 3.4**
- **Performance ratio = 9/3.4 = 2.6**

Multilevel Caches (4)

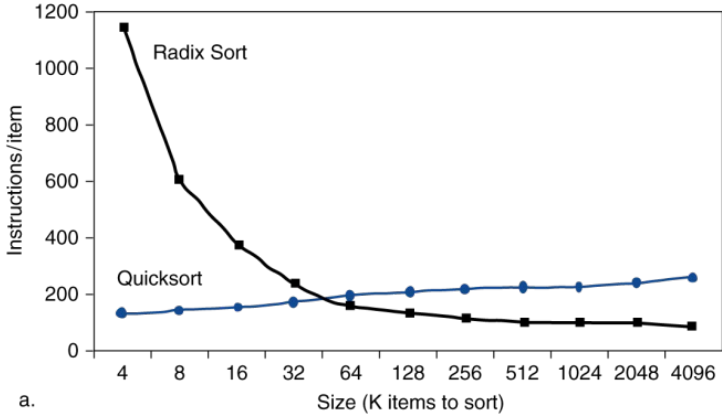
- **Multilevel cache considerations**
  - Primary cache
    - Focus on minimal hit time
  - L2 cache
    - Focus on low miss rate to avoid main memory access
    - Hit time has less overall impact
  - Results
    - L1 cache usually smaller than a single cache
    - L1 block size smaller than L2 block size

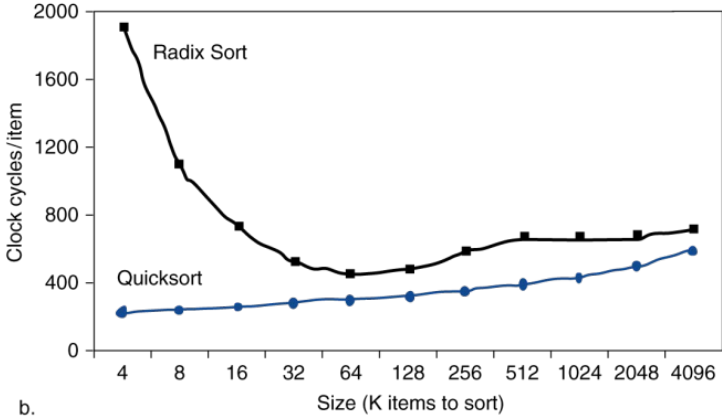
Interactions with Advanced CPUs

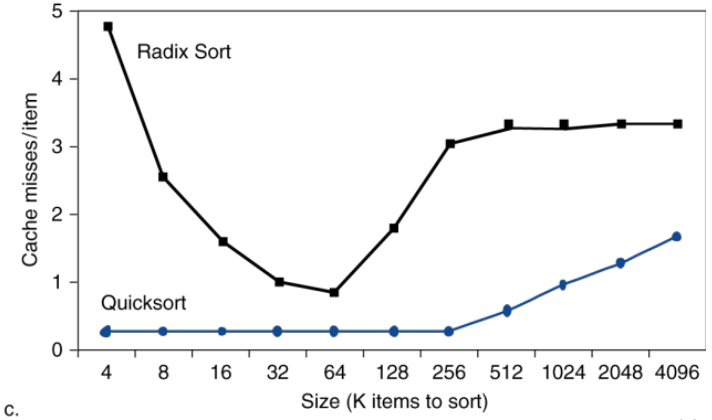
- **Out-of-order CPUs can execute instructions during cache miss**
  - Pending store stays in load/store unit
  - Dependent instructions wait in reservation stations
    - Independent instructions continue
- **Effect of miss depends on program data flow**
  - Much harder to analyse
  - Use system simulation

Interactions with Software

- **Misses depend on memory access patterns**
  - Algorithm behavior
  - Compiler optimization for memory access







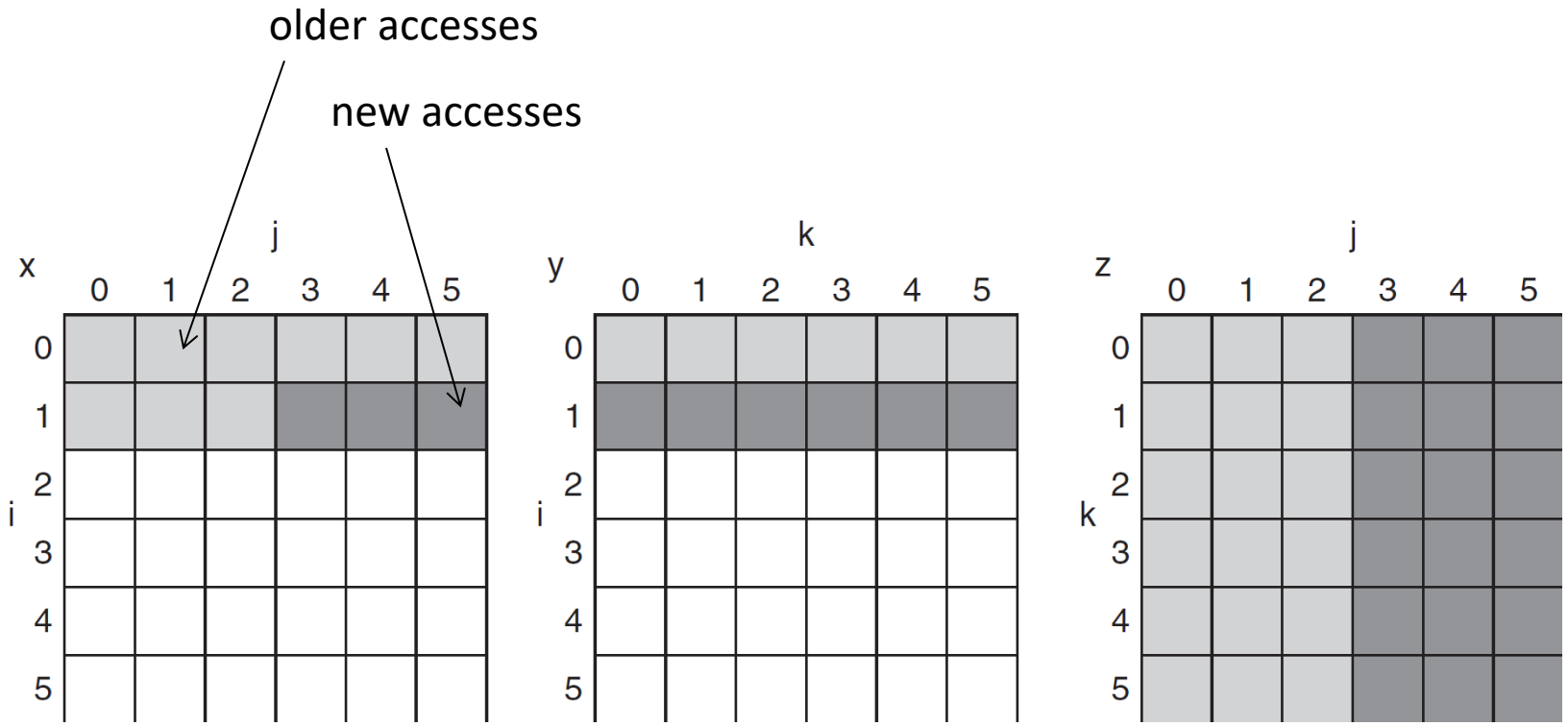
Software Optimization via Blocking (1)

- Goal: maximize accesses to data before it is replaced
- Consider inner loops of DGEMM:

```
for (int j = 0; j < n; ++j)
{
  double cij = C[i*n+j];
  for( int k = 0; k < n; k++ )
    cij += A[i*n+k] * B[k*n+j];
  C[i*n+j] = cij;
}
```

Software Optimization via Blocking (2)

- **DGEMM access pattern: C, A, and B arrays**



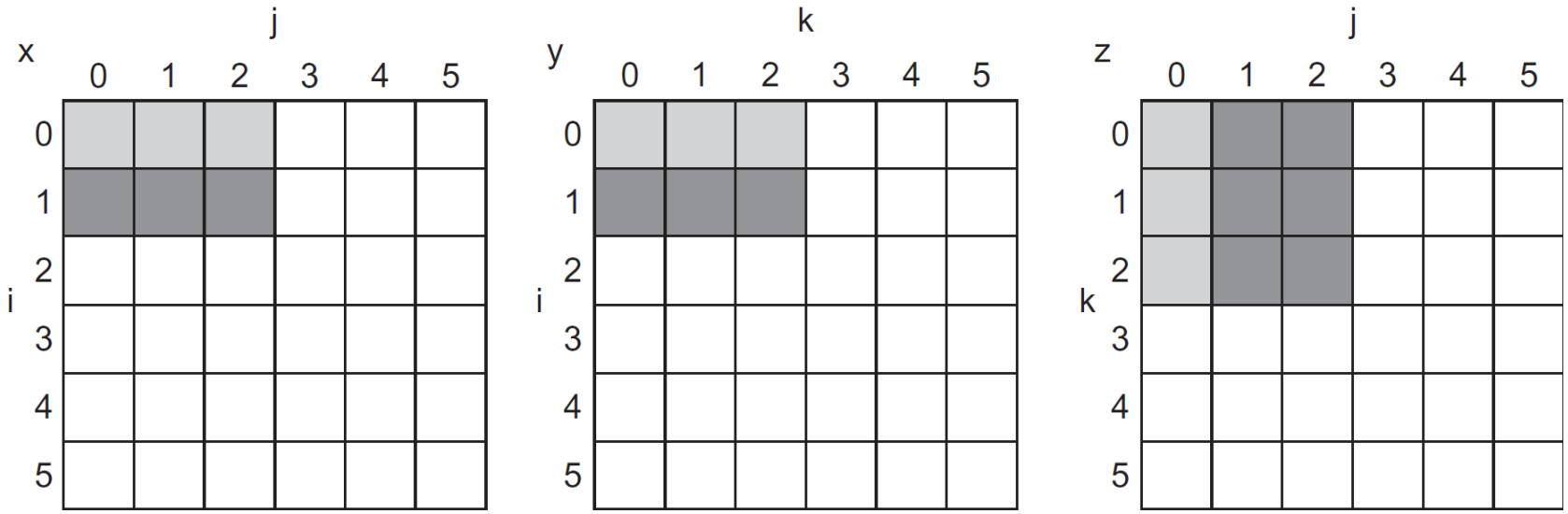
Software Optimization via Blocking (3)

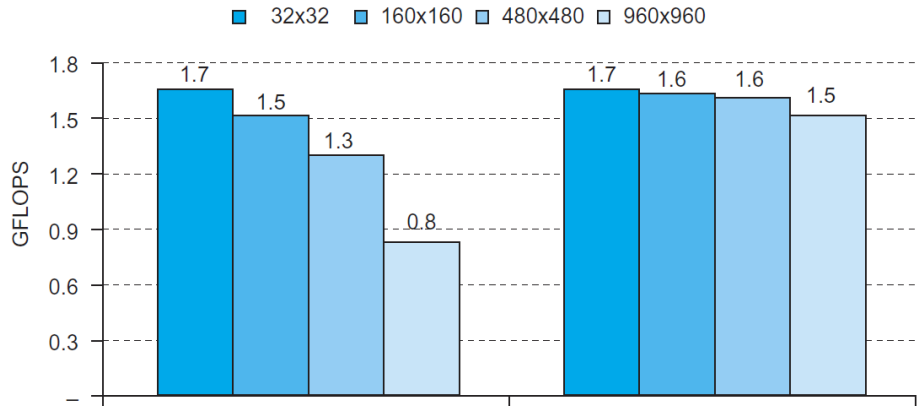
- Cache blocked DGEMM

```
1 #define BLOCKSIZE 32
2 void do_block (int n, int si, int sj, int sk, double *A, double
3 *B, double *C)
4 {
5  for (int i = si; i < si+BLOCKSIZE; ++i)
6   for (int j = sj; j < sj+BLOCKSIZE; ++j)
7   {
8    double cij = C[i*n+j];/* cij = C[i][j] */
9    for( int k = sk; k < sk+BLOCKSIZE; k++ )
10    cij += A[i*n+k] * B[k*n+j];/* cij+=A[i][k]*B[k][j] */
11   C[i*n+j] = cij;/* C[i][j] = cij */
12  }
13 }
14 void dgemm (int n, double* A, double* B, double* C)
15 {
16  for ( int sj = 0; sj < n; sj += BLOCKSIZE )
17   for ( int si = 0; si < n; si += BLOCKSIZE )
18    for ( int sk = 0; sk < n; sk += BLOCKSIZE )
19     do_block(n, si, sj, sk, A, B, C);
20 }
```

Software Optimization via Blocking (4)

- **Blocked DGEMM access pattern**



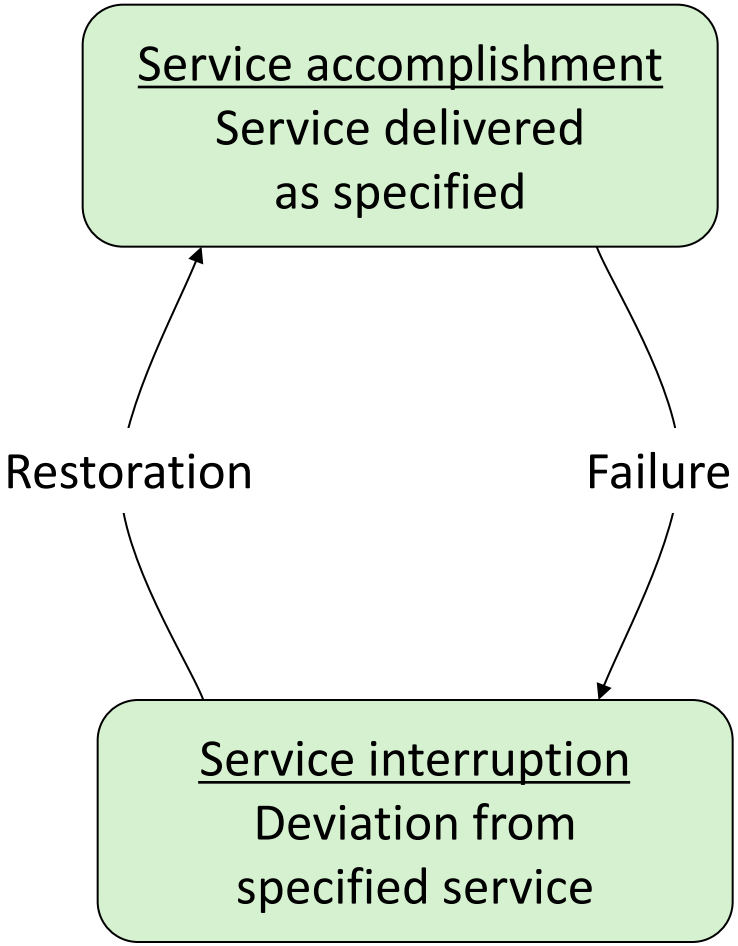


Outline

Textbook: COD 5.4-5.5

- Measuring and Improving Cache Performance
  - Cache Performance
  - Associative Caches
  - Multilevel Caches
  - Software Optimization via Blocking
- **Dependable Memory Hierarchy**

Dependability (1)



- **Fault: failure of a component**
  - May or may not lead to system failure

Dependability (2): Measures

- **Reliability: mean time to failure (MTTF)**
- **Service interruption: mean time to repair (MTTR)**
- **Mean time between failures**
  - MTBF = MTTF + MTTR
- **Availability = MTTF / (MTTF + MTTR)**
- **Improving Availability**
  - Increase MTTF: fault avoidance, fault tolerance, fault forecasting
  - Reduce MTTR: improved tools and processes for diagnosis and repair

Dependability (3): Hamming SEC Code

- **Hamming distance**
  - Number of bits that are different between two bit patterns
- **Minimum distance = 2 provides single bit error detection**
  - E.g. parity code
- **Minimum distance = 3 provides single error correction, 2 bit error detection**

Dependability (4): Encoding SEC

- **To calculate Hamming code:**
  - Number bits from 1 on the left
  - All bit positions that are a power 2 are parity bits
  - Each parity bit checks certain data bits:

Bit position		1	2	3	4	5	6	7	8	9	10	11	12
Encoded date bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8
Parity bit coverate	p1	X		X		X		X		X		X	
	p2		X	X			X	X			X	X	
	p4				X	X	X	X					X
	p8								X	X	X	X	X

Dependability (5): Decoding SEC

- **Value of parity bits indicates which bits are in error**
  - Use numbering from encoding procedure
  - Examples
    - Parity bits = 0000 indicates no error
    - Parity bits = 1010 indicates bit 10 was flipped

Dependability (6): SEC/DED Code

- **Add an additional parity bit for the whole word (p_n)**
- **Make Hamming distance = 4**
- **Decoding:**
  - Let H = SEC parity bits
    - H even, p_n even, no error
    - H odd, p_n odd, correctable single bit error
    - H even, p_n odd, error in p_n bit
    - H odd, p_n even, double error occurred
- **Note: ECC DRAM uses SEC/DED with 8 bits protecting each 64 bits**