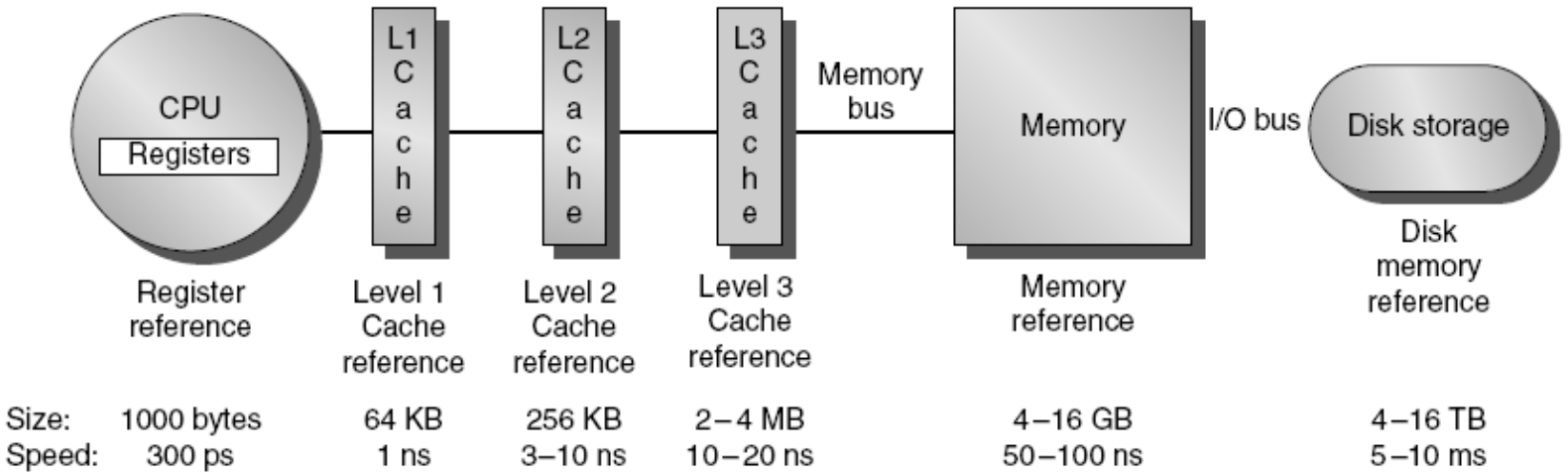
EEL 5764 Computer Architecture

Sandip Ray
Department of Electrical and Computer Engineering
University of Florida

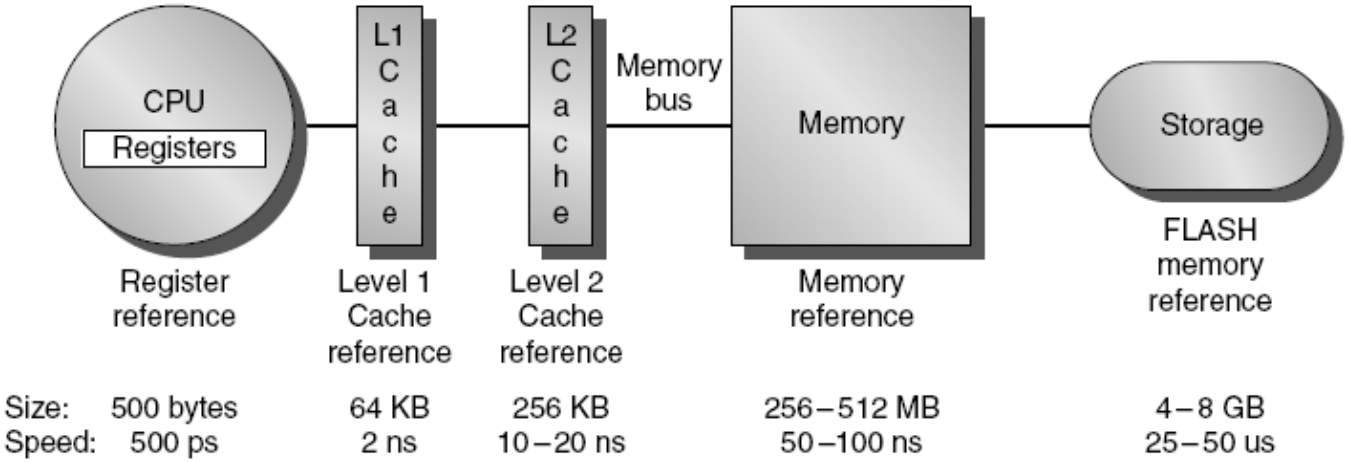
Lecture 8-9:

- Memory Hierarchy (Contd.)
- Introduction to Cache Optimizations

Memory Hierarchy



(a) Memory hierarchy for server



(b) Memory hierarchy for a personal mobile device

Cache Misses

- On cache *hit*, CPU proceeds normally
- On cache *miss*
  - → Stall the CPU
  - → Fetch a block from next level of mem. hierarchy
  - → Instruction cache miss
    - ➤ Restart instruction fetch
  - → Data cache miss
    - ➤ Complete data access
- Note that speculative and multithreaded processors may execute other instructions during a miss
  - → Reduces performance impact of misses

Cache Misses

- *Miss rate*
  - → Fraction of cache access that result in a miss

- Causes of misses
  - → *Compulsory*
    - ➤ First reference to a block
  - → *Capacity*
    - ➤ Blocks discarded and later retrieved
  - → *Conflict*
    - ➤ Program makes repeated references to multiple addresses from different blocks that map to the same location in the cache
    - ➤ Only happen in direct-mapped or set associative caches

Measuring Cache Performance

- **CPU time = (CPU cycles + mem stall cycles) * cycle time**
  - → CPU cycles = instruction cycles + cache hit time
  - → Memory stall cycles from cache misses

$$\text{Memory stall cycles} = \text{Number of misses} \times \text{Miss penalty}$$

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

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

Stall cycles per instruction

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 × 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 - 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 = 2 cycle, miss penalty = 20 cycles, I-cache miss rate = 5%
  - →AMAT = 2 + 0.05 × 20 = 3 cycles

Question: why not AMAT = hit time * (1-Miss rate) + Miss rate * Miss penalty?

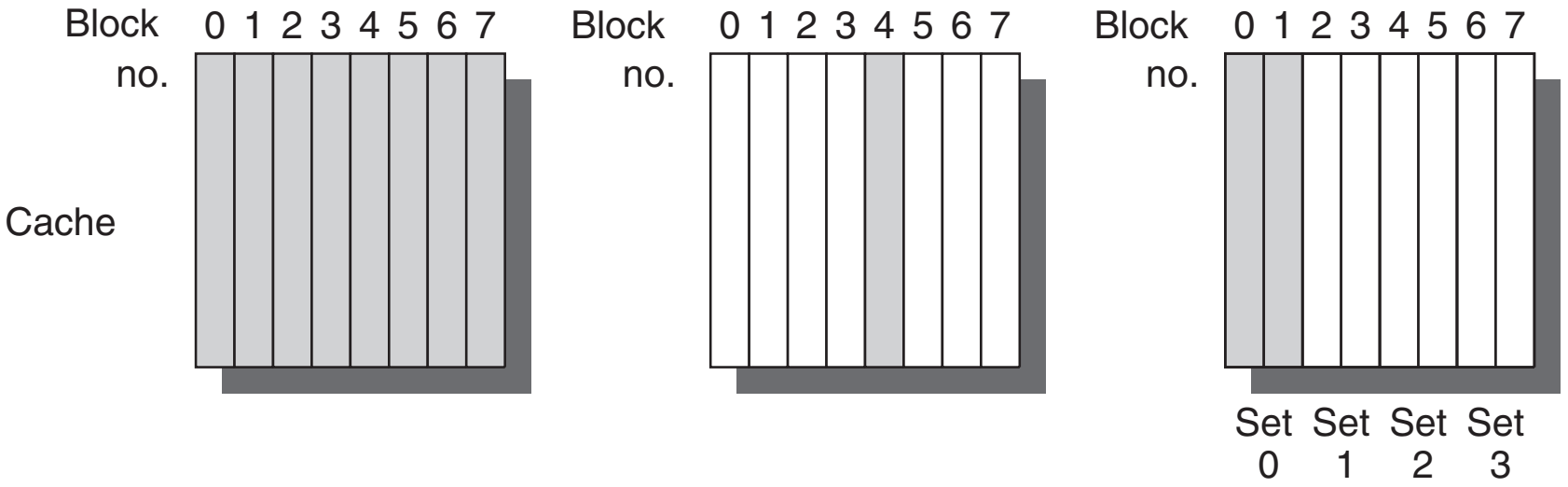
Performance Summary

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

Associative Caches

- Reduce misses due to conflicts.
- *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 address) MOD (#Sets in cache)
  - → Search all entries in a given set at once
  - → *n* comparators (less expensive)
  - → Direct-mapped = 1-way associative

Associative Cache Example



Size of Tags vs Associativity

- Increasing associativity requires
  - →More tag bits per cache block
  - →More comparators, each of which is more complex
- The choice among direct, set-associative and fully-associative mapping in any memory hierarchy will depend on
  - →cost of miss vs cost of implementing associativity, both in time and in extra hardware

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 otherwise
    - ➤ FIFO approximates LRU.
- Random
  - →Gives approximately the same performance as LRU for high associativity

Write Policy – Write-Through

- Update cache and memory
- Easier to implement
- Writes take longer – wait for mem update to complete
  - → e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
    - ➤ Effective CPI = 1 + 0.1×100 = 11
- Solution: **write buffer**
  - → Holds data waiting to be written to memory
  - → CPU continues immediately
    - ➤ Only stalls on write if write buffer is already full
  - → Write buffer is freed when a write to memory is finished

Write Policy – Write-Back

- Just update the block in cache
  - →Keep track of whether each block is dirty - dirty bits
  - →Cache and memory would be inconsistent
- When a dirty block is replaced
  - →Write it back to memory
- Write speed is faster
  - →One memory update for multiple writes.
  - →Power saving.
- Write buffer can also be used

Write Allocation

- What should happen on a write miss?
  - →Write allocate
  - →No write allocate
- Alternatives for write-through
  - →Allocate on miss: fetch the block
  - →Write around: don't fetch the block
- For write-back
  - →Usually fetch the block
  - →Act like read misses

Cache Performance – Review

$$\text{Average memory access time} = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$$

Any optimization should consider its impact on all three factors

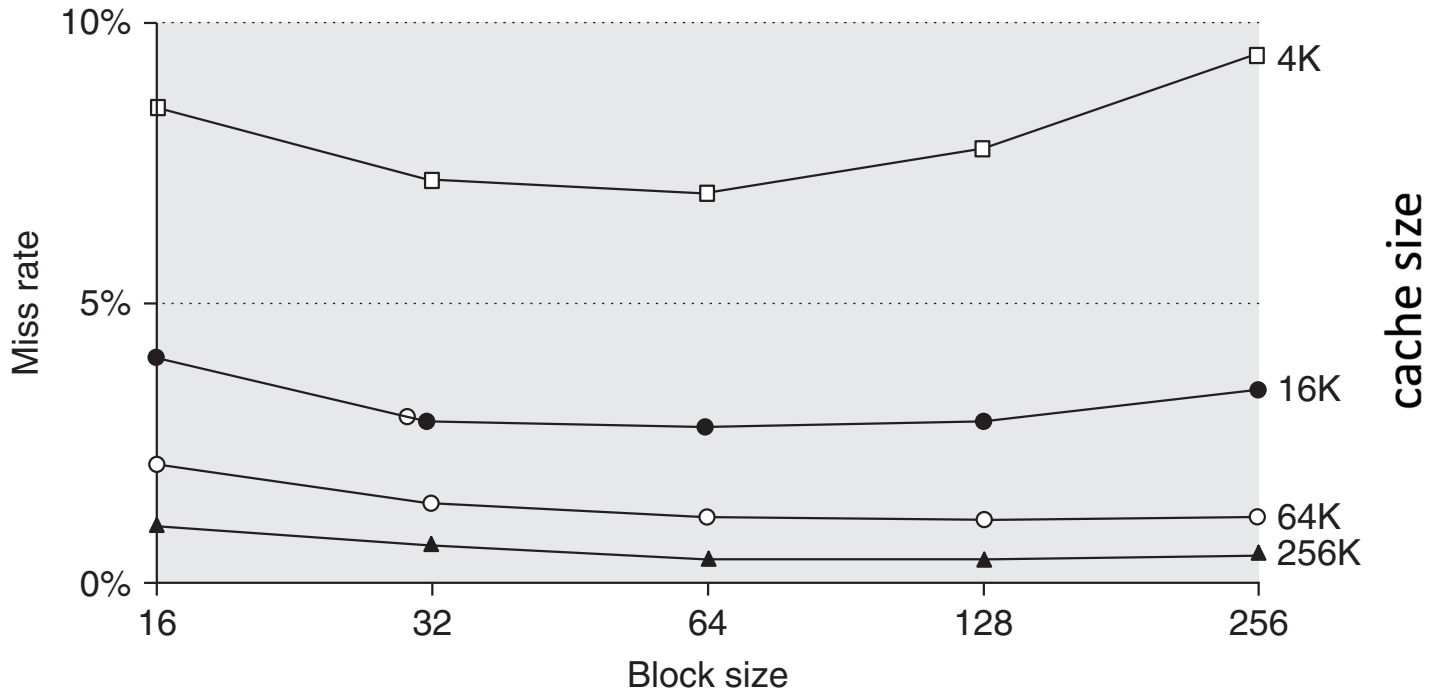
- Hit time
- Miss rate
- Miss penalty

Six Basic Cache Optimizations

- Larger block size
  - → Reduces compulsory misses
  - → Increases capacity and conflict misses, increases miss penalty
- Larger total cache capacity to reduce miss rate
  - → Increases hit time, increases power consumption
- Higher associativity
  - → Reduces conflict misses
  - → Increases hit time, increases power consumption
- Multi-level cache to reduce miss penalty
  - → Reduces overall memory access time
- Giving priority to read misses over writes
  - → Reduces miss penalty
- Avoiding address translation in cache indexing
  - → Reduces hit time

Optimization 1 – Larger Block Size

- Reduce compulsory misses due to spatial locality
- May increase conflict/capacity misses
- Increase miss penalty



Optimization 1 – Larger Block Size

- Reduce compulsory misses due to spatial locality
- May increase conflict/capacity misses
- Increase miss penalty

		Cache size			
Block size	Miss penalty	4K	16K	64K	256K
16	82	8.027	4.231	2.673	1.894
32	84	**7.082**	3.411	2.134	1.588
64	88	7.160	**3.323**	**1.933**	**1.449**
128	96	8.469	3.659	1.979	1.470
256	112	11.651	4.685	2.288	1.549

AMAT

Optimization 1 – Block Size Selection

- Determined by lower level memory
- High latency and high bandwidth – larger block size
- Low latency and low bandwidth – small block size

Optimization 2 – Larger Cache

- Reduce capacity misses
- May increases hit time
- Increases power consumption

Optimization 3 – Higher Associativity

- Reduces conflict misses
- Increases hit time
- Increases power consumption

	Associativity			
Cache size (KB)	1-way	2-way	4-way	8-way
4	3.44	3.25	3.22	**3.28**
8	2.69	2.58	2.55	**2.62**
16	2.23	**2.40**	**2.46**	**2.53**
32	2.06	**2.30**	**2.37**	**2.45**
64	1.92	**2.14**	**2.18**	**2.25**
128	1.52	**1.84**	**1.92**	**2.00**
256	1.32	**1.66**	**1.74**	**1.82**
512	1.20	**1.55**	**1.59**	**1.66**

Optimization 4 – Multilevel Cache

- L1 cache – small to keep hit time fast
- L2 cache – capture as many L1 misses & lower miss penalty
- Local miss rates – L1/L2 miss rates
- Global miss rates – miss rate(L1) * miss rate(L2)
- AMAT = hit(L1) + miss rate(L1) * miss penalty(L1)

  → miss penalty(L1) = hit(L2) + miss rate(L2) * miss penalty(L2)
- Mem Stall cycles/inst = miss/inst(L1) * hit(L2) +

  misses/inst(L2) * miss penalty(L2)

Optimization 4 – Multilevel Cache

- L1 hit time affects CPU speed
  → Small and fast L1
- Speed of L2 affects L1 miss penalty
  → Large L2 with higher associativity

Optimization 5 – Giving Priority to Read Misses over Writes

- Reduces miss penalty. See example below.

```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

- Write-through cache with write buffers suffers from RAW conflicts with main memory reads on cache misses:
  - → Write buffer holds updated data needed for the read.
  - → Alt #1 – wait for the write buffer to empty, increasing read miss penalty (in old MIPS 1000 by 50%).
  - → Alt #2 – Check write buffer contents before a read; if no conflicts, let the memory read go first.

Optimization 5 – Giving Priority to Read Misses over Writes

- Reduces miss penalty. See example below.

```
SW R3, 512(R0)     ;M[512] ← R3     (cache index 0)
LW R1, 1024(R0)    ;R1 ← M[1024]    (cache index 0)
LW R2, 512(R0)     ;R2 ← M[512]     (cache index 0)
```

- In a write-back cache, suppose a read miss causes a dirty block to be replaced
  - → Alt #1 – write the dirty block to memory first, then read memory.
  - → Alt #2 – copy the dirty block to a write buffer, read the new block, then write the dirty to memory.