CMPE 110: Computer Architecture

Week 7
Cache

Jishen Zhao (http://users.soe.ucsc.edu/~jzhao/)

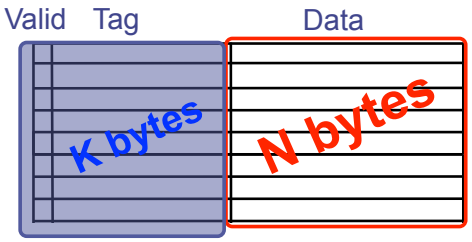
[Adapted in part from Jose Renau, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

Reminder

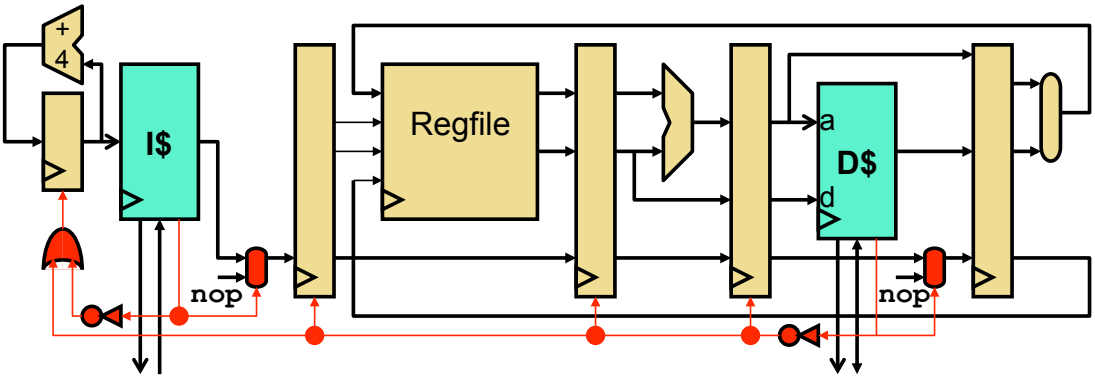
- Homework 2 Question 4 will be a bonus question
- Homework 3 is posted yesterday
  - Due on Nov. 16 (Wed) midnight

Review: How to calculate tag overhead

- Tag and valid bit overhead = (K / N) * 100%



Review: Cache Misses and Pipeline Stalls



- I$ and D$ misses stall pipeline just like data hazards
  - Stall logic driven by miss signal
    - Cache "logically" re-evaluates hit/miss every cycle
    - Block is filled → miss signal de-asserts → pipeline restarts

Today

- Handling cache hit and miss
- Cache performance, and AMAT
- Types of caches

Handling Cache Hits

- Read hits (I$ and D$)
  - this is what we want – no challenges
- Write hits (D$ only) -- Important!
  - Method 1: Allow cache and memory to be inconsistent → "write-back" cache
    - write the data only into the cache block (write-back the cache contents to the next level in the memory hierarchy when that cache block is "evicted")
    - (need a dirty bit for each data cache block to tell if it needs to be written back to memory when it is evicted)
  - Method 2: Require the cache and memory to be consistent
    - always write the data into both the cache block and the next level in the memory hierarchy (write-through) so don't need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – slow

Handling a Cache Miss

- What if requested data isn't in the cache?
  - How does it get in there?
- **Cache controller** (a finite state machine)
  - Remembers miss address
  - Accesses next level of memory
  - Waits for response
  - Writes data/tag into proper locations
- What if we need to kick a cache block out?

Classifying Misses: 3C Model

- Divide cache misses into three categories
  - **Compulsory (cold)**: never seen this address before
    - **Would miss even in infinite cache**
  - **Capacity**: miss caused because cache is too small
    - **Would miss even in fully associative cache**
    - Identify? Consecutive accesses to block separated by access to at least N other distinct blocks (N is number of frames in cache)
  - **Conflict**: miss caused because cache associativity is too low
    - Identify? **All other misses**

Cache and CPI

Example: CPI Calculation with Cache Misses

- Base CPI = 1
- Instruction mix: 30% loads/stores
- L1 I$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
- L1 D$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

} Considering cache misses, CPI = ?

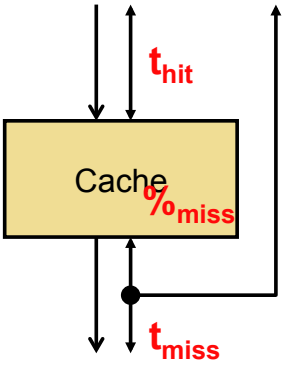
Assume 100 instructions in total → how many cycles to run?

- Without stalls, 100 instructions will take 100 cycles (base CPI = 1)
- All 100 instructions will need to access I$
  - 100 * 2% = 2 instructions miss in L1 I$ → each stall for 10 cycles
- 30 instructions are `lw` or `sw` that need to access D$
  - 30 * 10% = 3 instructions miss in L1 D$ → each stall for 10 cycles
- Total cycles stalled:
  - I$ stalls: 2 * 10 = 20 cycles
  - D$ stalls: 3 * 10 = 30 cycles

} Stall for 50 cycles in total

- Total cycles to execute the 100 instructions: 100 + 50 = 150 cycles
- CPI = 150 / 100 = 1.5

Cache Performance Equation



- For a cache
  - **Access**: read or write to cache
  - **Hit**: desired data found in cache
  - **Miss**: desired data not found in cache
    - Must get from another component
    - No notion of "miss" in register file
  - **$\%_{miss}$** (i.e., miss-rate): #misses / #accesses
  - **t_{hit}**: time to read data from (write data to) cache
  - **t_{miss}**: time to read data into cache
- Performance metric: average access time

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

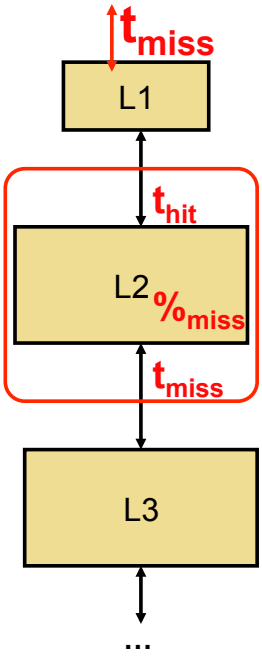
Example: Use the equation to calculate CPI

$$t_{avg} = t_{hit} + (\%_{miss} * t_{miss})$$

- Simple pipeline with base CPI of 1
- Instruction mix: 30% loads/stores
- I\$: $\%_{miss}$ = 2%, t_{miss} = 10 cycles
- D\$: $\%_{miss}$ = 10%, t_{miss} = 10 cycles

- What is CPI?
  - $CPI = CPI_{base} + (\%_{cache\_access} \%_{miss} * CPI_{miss})$
  - CPI = 1 + (100%*2%*10 cycles/insn) + (30%*10%*10 cycles/insn)
    = 1 + 0.2 + 0.3
    = 1.5

How about multiple levels of caches



- For a cache
  - $\%_{miss}$ (i.e., miss-rate): #misses / #accesses
  - t_{hit}: time to read data from (write data to) cache
  - t_{miss}: time to read data into cache

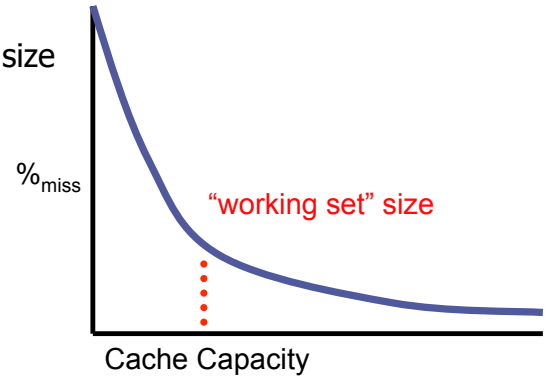
  *note: t can be *ns* or *cycles*
- Average time to access a level of cache

  $t_{avg} = t_{hit} + \%_{miss} * t_{miss}$
- t_{miss} = t_{avg} of next level →
- $t_{miss}(L1) = t_{hit}(L2) + \%_{miss}(L2)*t_{miss}(L2)$
-
- $= t_{hit}(L2) + \%_{miss}(L2)* (t_{hit}(L3) + \%_{miss}(L3)*t_{miss}(L3)$
- $= \ldots$

AMAT = t_{avg} of memory access

Capacity and Performance

- Simplest way to reduce $\%_{miss}$: increase capacity
  - \+ Miss rate decreases monotonically
    - **"Working set"**: insns/data program is actively using
    - Diminishing returns
  - – However t_{hit} increases
    - Latency grows with cache size
  - t_{avg} ?



Block Size

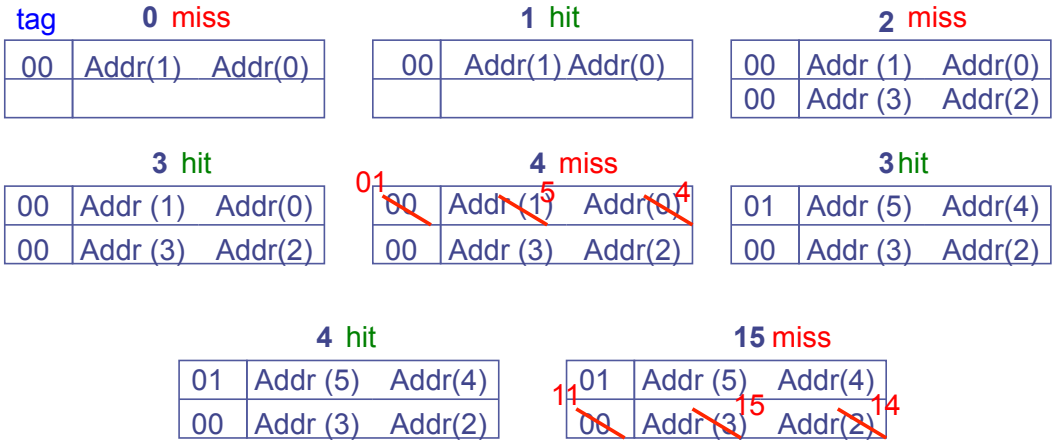
- For fixed capacity, reduce $\%_{miss}$ by changing organization
- One option: increase **block size** (an example on next slide)
  - Exploit **spatial locality**
  - Notice index/offset bits change
  - Tag remain the same

Taking Advantage of Spatial Locality

- Let cache block hold more than one word

Start with an empty cache block address: 0 1 2 3 4 3 4 15

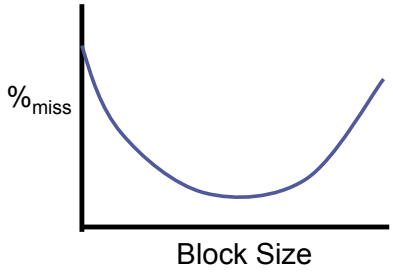
(Block address: 0000 0001 0010 0011 0100 0011 0100) 1111



- 8 requests, 4 misses

Effect of Block Size on Miss Rate

- Two effects on miss rate
  - + **Spatial prefetching (good)**
    - For adjacent locations
    - Turns miss/miss into miss/hit pairs
  - – **Interference (bad)**
    - For non-adjacent locations that map to adjacent frames
    - Turns hits into misses by disallowing simultaneous residence
    - Consider entire cache as one big block



- Both effects always present
  - Spatial "prefetching" dominates initially
    - Depends on size of the cache
  - Reasonable block sizes are 32B–128B
- But also increases traffic
  - More data moved, not all used

Types of cache

Directly vs. Associative Mapped Caching

- Direct mapped caching allows any given main memory block to be mapped into exactly one unique cache location.

- Set-associative mapped cache allows any given main memory block to be mapped into two or more cache locations.

- Fully-associative mapped caching allows any given main memory block to be mapped into any cache location.

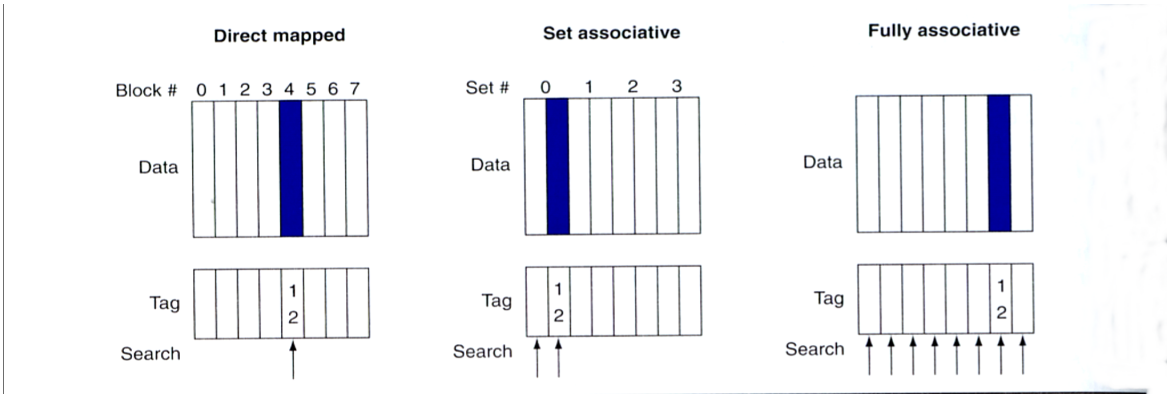


FIGURE 7.13 The location of a memory block whose address is 12 in a cache with eight blocks varies for direct-mapped,