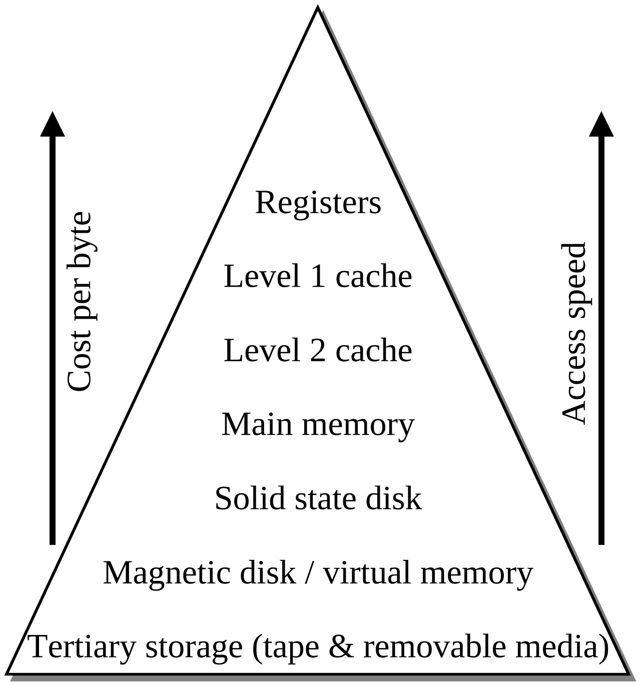
Memory hierarchy: the storage pyramid

- Principle of locality: programs don’t access code and data uniformly
- Faster hardware has less capacity and costs more per byte
- Result: memory hierarchy
  - Keep frequently used code & data in fast memory
  - Keep everything else in slower memory
- Two views of hierarchy
  - “Infinite supply of memory, some parts slower than others”
  - “Lots of objects”



Characterizing the memory hierarchy

- Questions about any 2 levels of the memory hierarchy:
  - Where can a block be placed in the upper level?
    - ⇒Block placement
  - How is a block found if it is in the upper level?
    - ⇒Block identification
  - Which block should be replaced on a miss?
    - ⇒Block replacement
  - What happens on a write?
    - ⇒Write strategy
- Focus on the interface between
  - CPU’s memory cache and main memory
  - Dynamic RAM and disk (virtual memory)

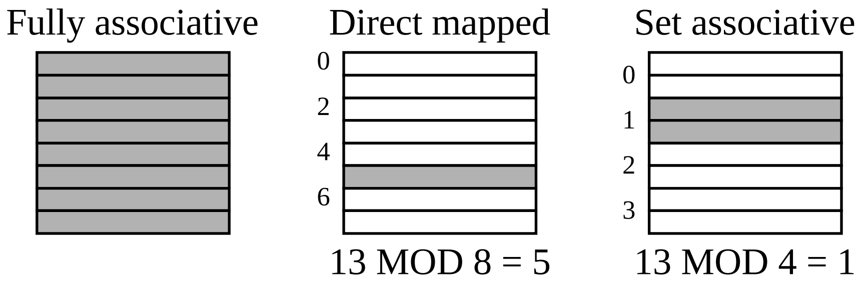
Memory system performance

- Evaluate the effectiveness of the memory hierarchy with:
  Memory stall cycles = IC * memory refs per instruction * miss rate * miss penalty
- Use a related formula to evaluate the performance of various memory system configurations
- Several factors in this equation:
  - IC * memory refs per instruction
    - Frequency with which the CPU uses memory
    - A memory system that needs to satisfy just 1-2 refs per cycle is easier to build than one that satisfies 4-5 refs per cycle
  - Miss rate: fraction of references not satisfied in the upper level
  - Miss penalty: length of time it takes to get a value from the lower level
  - Low miss rate doesn't help if miss penalty is too high (and vice versa)

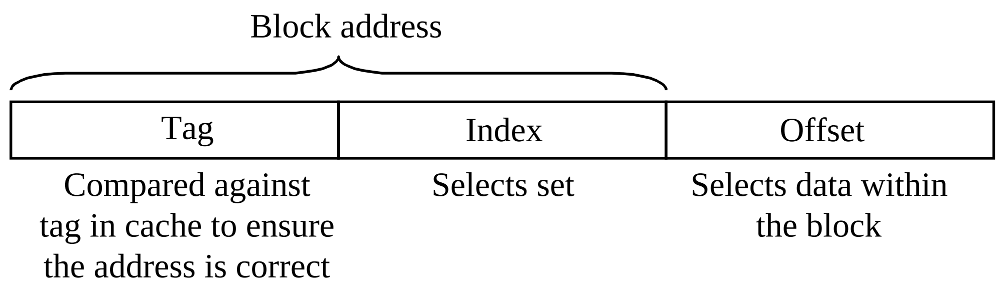
What is a cache?

- What does "cache" refer to?
  - No modifiers => usually means the fast memory closest to the CPU
  - "Cache" has been used for everything from files to WWW pages
- Block placement: three options
  - Fully associative (block can go anywhere)
  - Direct mapped (block can go in one place)
  - Set associative (block mapped to set, but can go anywhere in set)
  - Direct mapped = 1-way set associative

Where does block 13 go in cache?



Block identification


Block address
Tag
Index
Offset
Compared against tag in cache to ensure the address is correct
Selects set
Selects data within the block

- Block offset: the first few bits of the address give the offset of the byte within a block
- Block address (index): used to pick a set from the cache
- Tag
  - Only the tag is stored in the cache (the rest of the address is implied)
  - All tags within a set are searched in parallel
- Valid bit: indicates that the block in this location contains valid data
  - Otherwise, a random sequence of bits could be mistaken for a valid entry that matched the tag

Block replacement

- Which block is replaced?
  - For direct mapped, each block can only go in one location!
  - Question relevant for fully associative and set associative caches
- Block to replace chosen by
  - Random: choose a block from the set at random
  - LRU: least-recently used
    - Replace the block that has been unused for the longest time
    - This requires extra bits in the cache to keep track of accesses

  $\Rightarrow$ LRU isn’t much better than random replacement for small sets

$\Rightarrow$ Use LRU only for 2-way set associative
  - Other caches use random
  - Even 2-way set associative may use random to save bits

Write strategy

- What happens on a write?
- Memory access distribution
  - All instruction access are reads
  - Most data accesses are reads (DLX, 9% stores and 26% loads)
- Make the common case fast => optimize caches for reads
  - The common case is also the easy case to handle since tag checking and reading can occur in parallel
  - Extra bytes read can be safely ignored
- Amdahl's law reminds us that we can't *ignore* writes!
  - Problem: Tag checking and writing ***can't*** occur in parallel
  - ⇒ Writing is usually slower than reading
  - Extra bytes can't be safely written

Write policy

- Determines when the write is communicated to the lower level
- Write-through: block is written to both the cache and main memory at the same time
  - Read misses don't result in writes
  - Memory hierarchy is consistent
  - Simple to implement
- Write back (also known as copy back): block modified in cache only at time of write
  - Main memory modified when the block must be replaced in the cache
  - Requires the use of a dirty bit to keep track of block modification status
  - Writes occur at speed of cache
  - Multiple writes occur to the same block can be "collapsed"

Write misses

- Two options when a write is made to a block not in cache
  - Write allocate: the block is loaded into the cache on a miss before anything else occurs
  - Write around (no write allocate): the block is only written to main memory; it isn't stored in the cache
- Generally,
  - Write-back caches use write-allocate
    - ⇒ Hopefully, subsequent writes to that block will be captured by the cache
  - Write-through caches use write-around
    - ⇒ Subsequent writes to that block will still go to memory even if the block is fetched into cache

Write buffers & write merging

- Many CPUs use a write buffer to avoid stalling on writes
  - Write buffer => a small cache that can hold a few values waiting to go to main memory
  - This buffer helps when writes are clustered
  - It doesn't entirely eliminate stalls since it is possible for the buffer to fill if the burst is larger than the buffer
- Write merging
  - Blocks are often larger than a machine word
  - Write buffers can merge memory writes to save write buffer slots and memory traffic
  - ⇒ Writes to the same location can be collapsed
  - ⇒ Writes to sequential locations can be merged into a single buffer slot

Split vs. unified caches

- Should there be a single or two caches in the system?
- Unified cache: all memory requests go through a single cache
  - + Requires less hardware
  - – Has lower bandwidth
  - – More opportunity for collisions
- Split I & D caches: instructions & data are stored in separate caches
  - – Uses additional hardware
    - Some simplifications (I-cache is read-only)
  - + Higher bandwidth (2 is greater than 1)
  - + No collisions between data & instructions

Cache performance

- Average memory access time (AMAT) is a useful measure to evaluate the performance of a memory-hierarchy configuration
  AMAT = hit time + miss rate * miss penalty
- AMAT shows how much penalty the memory system imposes on each access (on average)
  - ⇒ It can easily be converted into clock cycles for a particular CPU
- Leaving the penalty in nanoseconds allows two systems with different clock cycles times to be compared using a given memory system

Performance for split I & D caches

- Instruction and data accesses may have different penalties
  - They may have to be computed separately
  - This requires knowledge of the fraction of references that are instructions and the fraction that are data
  - For example, the text says that (usually) 75% of memory references are instructions, and 25% are data references
- The write penalty can also be computed separately from the read penalty
  - Miss rates may be different for each situation
  - Miss penalties may be different for each situation (i.e., writeback vs. write through)

$$CPU\ \ time = IC \times \left(CPI_{execution} + \frac{Memory\ \ accesses}{instruction} \times Miss\ \ rate \times Miss\ \ \ penalty \right) \times Clock\ \ cycle\ \ time$$

Cache performance example

- Problem: compare the performance of a 64KB unified cache with a split cache with 32KB data and 16KB instruction
  - Miss penalty for either cache is 100 ns, and the CPU clock runs at 500 MHz
  - Don’t forget that the unified cache requires an extra cycle for load and store hits because of the structural conflict
  - Calculate the effect on CPI rather than the average memory access time
- Assume miss rates are as follows (Fig. 5.7 in text):
  - 64K Unified cache: 1.35%
  - 16K instruction cache: 0.64%
  - 32K data cache: 4.82%
- Assume a data access occurs in 1/3 of all instructions

Cache performance example

- Compute the CPI penalty separately for instructions and data
- First, figure out the miss penalty in terms of clock cycles: 100 ns/2 ns = 50 cycles
- Unified cache
  – Instruction access penalty is (0 + 1.35% * 50) = 0.675 cycles
  – Data access penalty is (1 + 1.35% * 50) = 1.675 cycles
  – Overall penalty is 0.675 + (1/3) * 1.675 = 1.23 cycles per instruction
- Split cache
  – Instruction access penalty is (0 + 0.64% * 50) = 0.32 cycles
  – Data access penalty is (0 + 4.82% * 50) = 2.41 cycles
  – Overall penalty is 0.32 + (1/3) * 2.41 = 1.12
- Split cache performs better => no stall on data accesses

Effects of cache on CPU performance

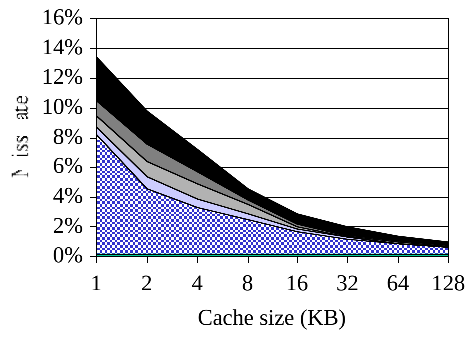
- Low CPI machines suffer more relative to some fixed CPI memory penalty
  – A machine with a CPI of 5 suffers little from a 1 CPI penalty.
  – A processor with a CPI of 0.5 has its execution time tripled!
- Cache miss penalties are measured in cycles, not nanoseconds
  ⇒ A faster machine will stall more cycles on the same memory system
- Amdahl’s Law raises its ugly head again
  – Fast machines with low CPI are affected significantly from memory access penalties
  – Fast machines spend most of their time accessing memory!

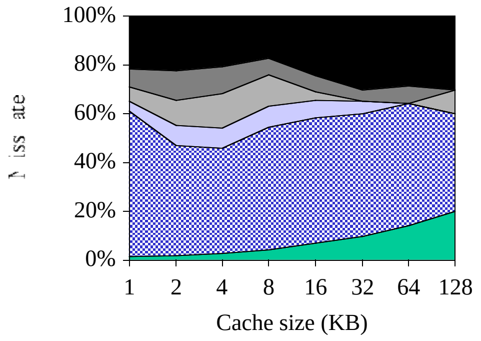
Improving cache performance

- The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important
- There are many distinct methods system architect use to reduce average memory access time
- These methods can be classified by whether they
  - Reduce the miss rate
  - Reduce the miss penalty
  - Reduce the time to hit in a cache
- Other methods may also increase capacity for a given cost...

Components of cache miss rate

- Three "C"s of cache misses
- Compulsory misses
  - First access to a block ***can't*** be in the cache
  - Occur regardless of cache size
- Capacity misses
  - Occur because cache isn't large enough to hold all blocks
  - Compulsory miss rate - the miss rate of a fully associative cache
- Conflict (collision) misses
  - The block can't be kept because the set is full
  - Difference between fully- and set- associative cache





Reducing cache miss rate

- To reduce cache miss rate, we must eliminate some of the misses due to the three C's
  - ***Capacity*** misses can't be reduced much except by making the cache larger
  - ***Conflict*** misses and ***compulsory*** misses can be reduced in several ways
- Larger cache blocks
  - Decrease the compulsory miss rate by taking advantage of spatial locality
  - May increase the miss penalty by requiring more data to be fetched per miss
  - Likely to increase conflict misses since fewer blocks can be stored in the cache
  - May even increase capacity misses in small caches