Cache Design

CS/COE 1541 (Fall 2020)

Wonsun Ahn

University of Pittsburgh

Evaluating Cache Design

Impact of Memory on Performance

- CPU Cycles = CPU Execution Cycles + Memory Stall Cycles
  - CPU Execution Cycles = cycles where CPU is doing useful work
  - Memory Stall Cycles = cycles where CPU is waiting on cache miss
    - Same memory stall cycles we measured using the PMU

- Processor design features that impact CPU execution cycles:
  - Pipelining, branch prediction, wide execution, out-of-order, …

- Processor design features that impact Memory stall cycles:
  - Caches
  - Write buffer
  - Prefetcher (we haven't learned this yet)

Impact of Memory on Performance

- CPU Cycles = CPU Execution Cycles + Memory Stall Cycles

- This gives rise to two categories of programs
  - **CPU Bound**: Programs where CPU execution is majority of cycles
  - **Memory Bound**: Programs where Mem stalls is majority of cycles

- To improve performance of **CPU Bound** programs
  - Improve HW by getting **beefier superscalar CPUs**
  - Improve SW by **optimizing the computation** in the program

- To improve performance of **Memory Bound** programs
  - Improve HW installing more CPU **caches** or faster DRAM
  - Improve SW by **optimizing memory access pattern** of program
  - We already saw how array.c is much faster than linked-list.c

How about overclocking using DVFS?

- CPU Time = CPU Cycles * Cycle Time
  = (CPU Execution Cycles + Memory Stall Cycles) * Cycle Time

- What if we halved the Cycle Time using DVFS?
  - Memory Stall Cycles could increase by close to 2X!
  - Why? You may speed up CPU, but DRAM speed remains the same
    - The bus (wire) that connects CPU to DRAM is not getting any faster
    - The DRAM chip itself is not getting clocked any faster
  - So if DRAM access speed was 100 ns,
    - If CPU is clocked at 1 GHz, it takes 100 cycles to access memory
    - If CPU is clocked at 2 GHz, it takes 200 cycles to access memory

- So if a program is **Memory Bound, overclocking is mostly useless**
  - Reduction in Cycle Time canceled out by increase in Memory Stall Cycles

Oracle Cache

- CPU Cycles = CPU Execution Cycles + Memory Stall Cycles

- **Oracle cache**: a cache that never misses
  - Impossible, since even with infinite capacity, there are still cold misses
  - But useful to set **bounds** on performance
  - With oracle cache, CPU Cycles is simply CPU Execution Cycles

- Real caches may approach performance of oracle caches but can't exceed

- What metric can we use to compare and evaluate real cache designs?
  - AMAT (Average Memory Access Time)

AMAT (Average Memory Access Time)

- **AMAT** (Average Memory Access Time) is defined as follows:
  - $\text{AMAT} = \text{hit time} + (\text{miss rate} \times \text{miss penalty})$
  - **Hit time:** time to get the data from cache when we hit
  - **Miss rate:** what percentage of cache accesses we miss
  - **Miss penalty:** time to get the data from lower memory when we miss
  - Hit time is incurred regardless of hit or miss (miss rate)
    - Since even after a miss, the cache must be accessed again and "hit"
  - Miss penalty is incurred only on a miss

- Hit time, miss rate, miss penalty are the 3 components of a cache design
  - When evaluating a cache design we need to consider all 3
  - Cache designs trade-off one for the other
    - E.g. a large cache trade-offs longer hit time for smaller miss rate
    - Whether trade-off is beneficial depends on the AMAT

AMAT for Multi-level Caches

- For a single-level cache (L1 cache):
  - $\text{AMAT(L1)} = \text{L1 hit time} + (\text{L1 miss rate} \times \text{DRAM access time})$

- For a multi-level cache (L1, L2 caches):
  - $\text{AMAT(L2)} = \text{L1 hit time} + (\text{L1 miss rate} \times \text{L1 miss penalty})$
  - $\text{L1 miss penalty} = \text{L2 hit time} + (\text{L2 miss rate} \times \text{DRAM access time})$
  - $\text{AMAT(L2)} = \text{L1 hit time} + \text{L1 miss rate} \times \text{L2 hit time} + \text{L1 miss rate} \times \text{L2 miss rate} \times \text{DRAM access time}$

- Which is better? AMAT(L1) or AMAT(L2)?
  - $\text{AMAT(L2)} - \text{AMAT(L1)} = \text{L1 miss rate} \times \text{L2 hit time} + (\text{L1 miss rate} \times \text{L2 miss rate} - \text{L1 miss rate}) \times \text{DRAM access time}$
    $= \text{L1 miss rate} \times (\text{L2 hit time} + (\text{L2 miss rate} - 1) \times \text{DRAM access time})$

AMAT for Multi-level Caches

- Which is better? AMAT(L1) or AMAT(L2)?
  - $\text{AMAT(L2)} - \text{AMAT(L1)}$
    $= \text{L1 miss rate} \times (\text{L2 hit time} + (\text{L2 miss rate} - 1) \times \text{DRAM access time})$
- For AMAT(L1) > AMAT(L2) (that is, for it to be worth it to put in an L2):
  - $\text{L1 miss rate} \times (\text{L2 hit time} + (\text{L2 miss rate} - 1) \times \text{DRAM access time}) < 0$
  - $\text{L2 hit time} + (\text{L2 miss rate} - 1) \times \text{DRAM access time} < 0$
  - $\text{L2 hit time} < (1 - \text{L2 miss rate}) \times \text{DRAM access time}$
  - If L2 hit time = 10 cycles and DRAM access time = 100 cycles,
    $10 < (1 - \text{L2 miss rate}) \times 100$
    $\text{L2 miss rate} < 0.9$
  - **Unless L2 cache miss rate is greater than 90%** (which is horrible), **worth it to install an L2 cache** (if we can keep hit time at 10 cycles)!
- But that conclusion is application dependent
  - If your program has poor locality and **L2 miss rate is above 90%**, the **additional L2 cache will hurt performance**!

Cache Design Parameter 1: Cache Block Size

Impact of Cache Block Size

- AMAT = hit time + (miss rate × miss penalty)

- **Cache block** (a.k.a. **cache line**)
  - Unit of transfer for cache data (typically 32, 64, or 128 bytes)
  - If program accesses any byte in cache block, entire block is brought in
  - Each level of a multi-level cache can have a different cache block size

- Impact of larger cache block size on **miss rate**
  - Maybe **smaller miss rate** due to **better** leveraging of **spatial locality**
  - Maybe **bigger miss rate** due to **worse** leveraging of **temporal locality** (Bringing in more data at a time may push out other useful data)

- Impact of larger cache block size on **miss penalty**
  - With a limited bus width, may take multiple transfers for a large block
  - E.g. DDR 4 DRAM bus width is 8 bytes, so 8 transfers for 64-byte block
  - Could lead to **increase in miss penalty**

So what cache block size should I choose?

- Again, that depends on the application
  - How much spatial and temporal locality the application has

- Simulate multiple cache block sizes and choose one with lowest AMAT
  - Simulate on the benchmarks that you care about
  - You may have to simulate different combinations for multi-level caches

Cache Design Parameter 2: Cache Associativity

Mapping blocks from memory to caches

- Cache size is much smaller compared to the entire memory space
  - Must map all the blocks in memory to limited CPU cache
- Doesn't this sound familiar? Remember branch prediction?
  - Had similar problem of mapping PCs to a limited BHT
  - What did we do then?
    - We hashed PC to an entry in the BHT
    - On a hash conflict, we replaced old entry with more recent one

- We will use a similar idea with caches
  - **Hash cache block addresses** in memory to entries in cache
  - On a conflict:
    - Replace old cache block with more recent one
    - Or, chain multiple blocks for same hash value

Cache Associativity

- Depending on hash function and chaining, a cache is either:
  - Direct-mapped (no chaining allowed)
  - Set-associative (some chaining allowed)
  - Fully-associative (limitless chaining allowed)

- Impact of more associativity on **miss rate**
  - **Smaller miss rate** due to less misses due to hash conflicts
  - Misses due to hash conflicts are called **conflict misses**
    - A third category of misses besides, cold and capacity misses

- Impact of more associativity on **hit time**
  - **Longer hit time** due to need to search through long chain

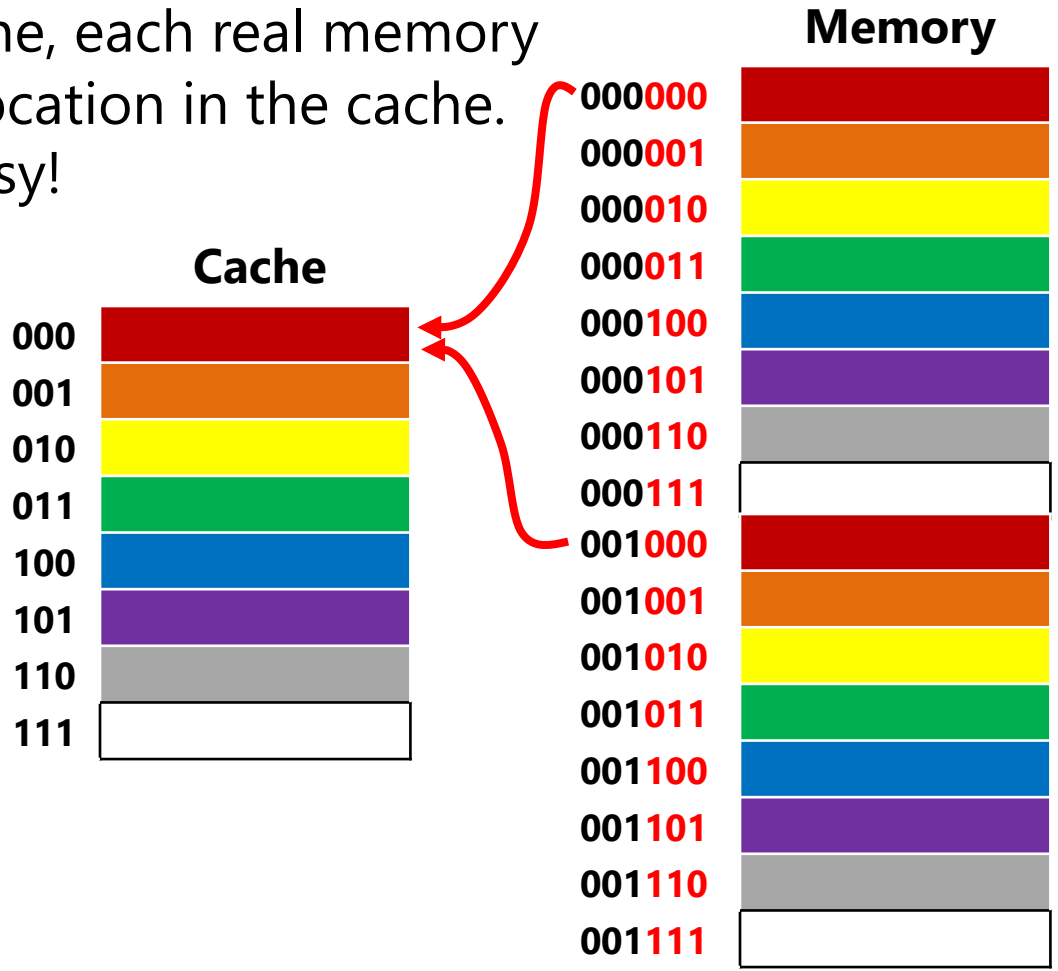
Direct-mapped Caches

Hash Function

- In a direct-mapped cache, each real memory location maps to **one** location in the cache.
- Implementing this is easy!

For this 8-entry cache, to find the cache block for a memory address, take the lowest 3 address bits.

But if our program accesses **001000**, then **000000**, how do we tell them apart? Tags!



Tags

- Each cache block has a **tag** which indicates the memory address that it corresponds to.

This seems redundant...

	Tag	Data
000	001000	DEADBEEF
001		
010		
011	110011	CAFEFACE
100	101100	B0DECA7
101		
110		
111		

For address **110011, 011** is the block, and **110** is the tag.

	Tag	Data
000	001	DEADBEEF
001		
010		
011	110	CAFEFACE
100	101	B0DECA7
101		
110		
111		

How do we tell what entries are empty/full?

Just add another bit (a **valid** bit)!

Watching it in action

- When the program first starts, we **set all the valid bits to 0**. The cache is empty.
- Now let's try a sequence of reads... do these **hit** or **miss?** How do the cache contents change?

000000 **miss**
100101 **miss**
100100 **miss**

} Why did these misses happen?

100101 **hit**
010000 **miss** ← Why did this miss happen?

	V	Tag	Data
000	1	010	something
001	0		
010	0		
011	0		
100	1	100	something
101	1	100	something
110	0		
111	0		

Steps in Hardware

1. Split the address into two parts: block and tag
   - *a bus splitter*
2. Use the block index to find the right cache entry.
   - *a mux*
3. If the valid bit is 1, and the entry's tag matches,
   - *an equality comparator and an AND gate*
   - **It's a hit!** Read the cached data.
4. Otherwise…
   - **It's a miss.**
   - If load miss, find something else to issue in instruction queue
   - If store miss, put in Write Buffer and retire following instructions