COM SCI M151B Week 7

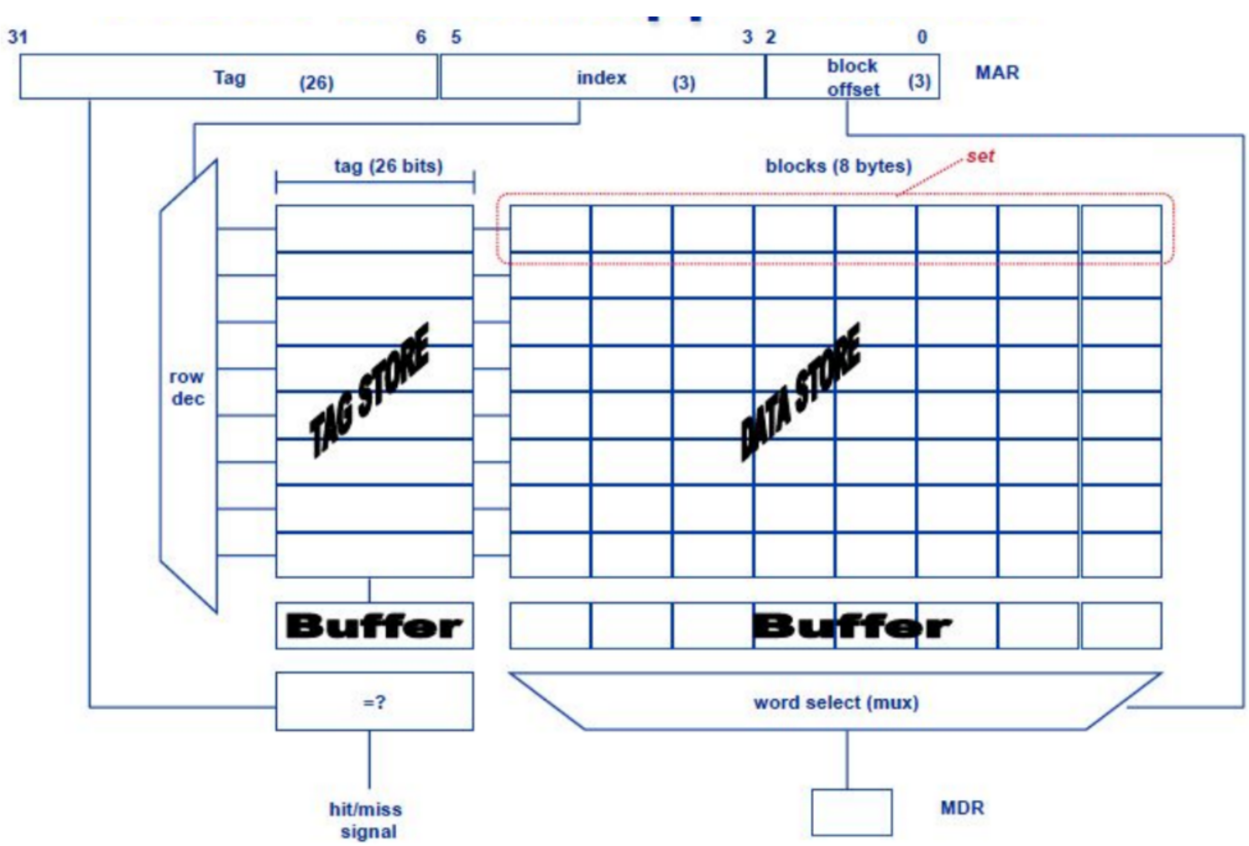
Aidan Jan

November 12, 2024

Cache Block

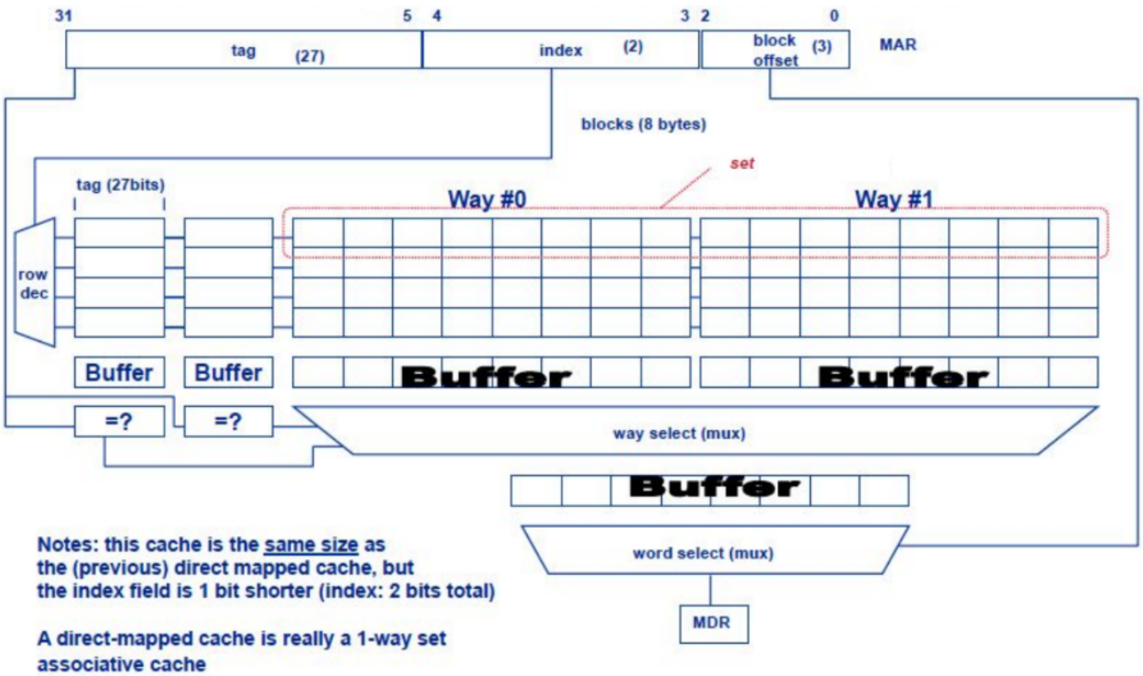
- Instead of storing 1 byte per row, we can store a block with multiple bytes.
  - Every time we need to load something to the cache, we load it at block level. (Spatial Locality)
  - We still send things to the CPU at byte level
    - How to do that? → We need *block offset* to decide which byte within the block should be selected!
  - How big a block should be? It depends! Typically somewhere between 8B-64B.
- Therefore, **Cache Address = {tag, index, block offset}**

A 64B Direct Mapped Cache



- Address is used to select index, of tag, which selects "rows", and block offset selects the word "columns".

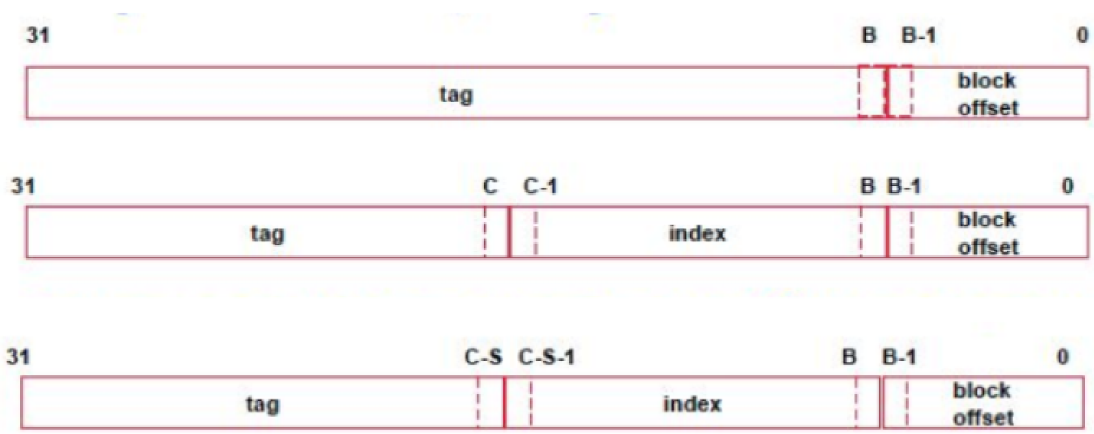
A 64B 2-way Set Associative Cache



- Compared to the direct mapped cache, the tag is two bits shorter to make way for an index.
- Here, the tag is used to select which "Way" is used, and the two bits of the index selects the row. The block offset then gets the column.

CBS

- C = log(bytes per cache)
- B = log(bytes per block)
- S = log(blocks per set)



How Big A Block Should Be?

- Bringing more data is nice because you have spatial locality
- However, it is not always the best idea because it increases overhead

- You are essentially making a trade off between miss rate and miss penalty.

Reducing Miss Rate

- Miss rate can be reduced by making blocks bigger, but that comes at the trade-off of miss penalty.
- What if we increase associativity? (e.g., add more ways)
  - More ways leads to higher hit time. (As log(cache size) increases, miss rate drops, but it drops following an exponential decay function. "diminishing returns")
  - An 8-way set associative cache is as good as fully-associative. After that, the limit is capacity miss.
- We can also increase cache size. But this leads to slower hit time. Making cache larger also has diminishing returns!
- Prefetching: Idea: if we can guess the access pattern we can bring data before it is needed!

Prefetching - Four Questions!

- **What** addresses to prefetch (i.e., address prediction algorithm)
- **When** to initiate a prefetch request (early, late, on time)
- **Where** to place the prefetched data (different layers of caches, separate buffer)
- **How** does the prefetcher operate and who operates it (software, hardware, hybrid)

Prefetchers look at the history of addresses accessed to predict the next address access. Similar to how a branch predictor looks at the history of branches, the prefetcher looks at the history of addresses.

- This reduces compulsory misses and therefore miss rate
- However, this leads to cache pollution
  - Need to monitor prefetching accuracy to change its *aggressiveness*
  - Other than this, no other negative impacts! No correctness issues!

Software vs. Hardware Prefetch

- Software prefetching
  - ISA provides prefetch instructions
  - Programmer or compiler inserts prefetch instructions (effort)
  - Usually works well only for "regular access patterns"
- Hardware prefetching
  - Hardware monitors processor accesses
  - Memorizes or finds patterns/strides
  - Generates prefetch addresses automatically

Example: Hardware Prefetcher

Next line prefetcher:

- Always prefetch next N cache lines after a demand access
- Pros:
  - Simple to implement
  - No need for sophisticated pattern detection
  - Works well for sequential/streaming access patterns (instrctions?)
- Cons:
  - Can waste bandwidth with irregular patterns
  - Low prefetch accuracy if access stride = 2 or when the program is traversing memory from higher to lower addresses.
- Better options? Stride prefetcher, stream buffers, etc.

Victim Cache

- Idea: for heavily conflicting addresses, a few "extra" temporary sets could remove conflicts!
  - Use a very small buffer (called victim cache) to save the recently discarded blocks. Search through them as well.
  - Reduce conflict misses
    - Research shows a 4-entry victim cache can remote up to 90% of conflicts.
  - Extra overhead
  - More complex design.

Compiler and Software

- Reorder accesses/arrays to increase locality.
- Combine loops with similar behavior
- Use "tiling" to access arrays region by region instead of whole
  - If column-major, `x[i+1, j]` follows `x[i, j]` in memory.
  - Meanwhile, `x[i, j + 1]` is far away from `x[i, j]`.
  - Poor code:

    ```
    for i = 1:
        for j = 1:
            sum = sum + x[i, j]
    ```
  - Better code:

    ```
    for j = 1:
        for i = 1:
            sum = sum + x[i, j]
    ```
- Use compiler profiling to improve prefetching

Reducing Miss Rate

- Replacement policy
  - LRU vs. PLRU vs. Random
  - Storage vs. Accuracy tradeoff!

Reducing Miss Penalty

- Write buffer: use a load store queue
- Pros:
  - No wait for stores needed.
  - Lower miss penalty for loads.
- Cons:
  - More overhead.

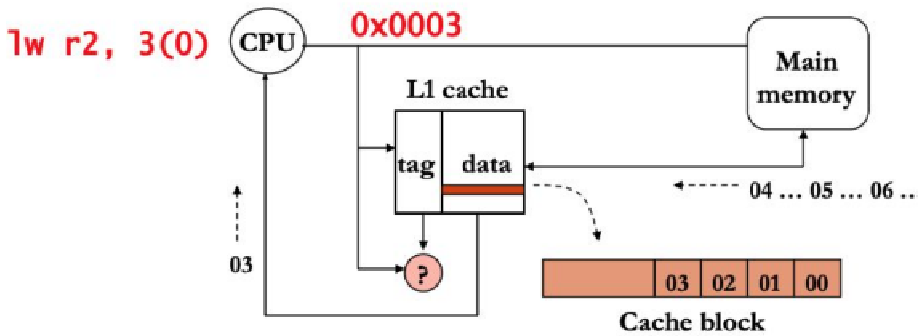
What happens on a store?

1. Data exists in the cache?
   - Should we update memory AND cache on every write?
     - Write through strategy
   - Update the memory only when the line is evicted.
     - Write back strategy
   - Tradeoff: Less writes vs. Storage overhead vs. Memory status.
2. Data does not exist.
   - Should we bring it to the cache? -write allocate
   - We probably don't need it anymore, so don't bring it. -write no allocate

- *Write back* often compined with *write-allocate.*
- *Write-through* often combined with *write-no allocate.*
- How to pick?
  - It depends!
    * Could be different for each level!
    * Can be optimized using simulation and architectural search!

Reducing Miss Penalty via Early Restart

- Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.

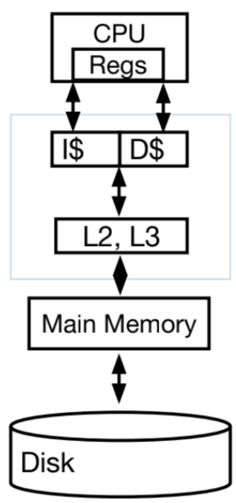


Early Restart *with critical word first*:

- Instead of waiting for all bytes (in a block) to arrive, forward data to CPU as soon as the requested byte(s) arrives.
- To further optimize this, first read the requested byte!

Multi-level Cache

- We can also reduce miss penalty by adding more levels of cache:
  - Adding L2, L3, etc.
- Higher level cache means bigger and slower. However, it is still both smaller and faster than the main memory.



Cache Performance

- Average time to access the cache:

$$AAT = HitTime + MissRate \times MissPenalty$$

- $HitTime$: Time it takes to access the (L1) cache.
- $MissRate$: The average frequency of misses (in L1).
- $MissPenalty$: The time required to access the main memory.

- What if there are multiple levels?
  - The miss penalty is the average time required to grab the data from either the other caches or main memory.

Inclusive vs. Exclusive Cache

- Inclusive: L_i is a subset of L_{i+1}
  - (pro) Easier to find data
  - (con) Wasted capacity
- Exclusive: Data is **only** in one of the levels.
  - (con) Difficult to find data
  - (pro) Efficient capacity

Modern Designs

- Split vs Unified "Caches"
  - L1 I/D caches commonly split and asymmetrical
    * Double bandwidth and no-cross pollution on disjoint I and D footprints
    * i-cache is smaller, simpler with more spatial locality. Usually a prefetcher and/or trace cache is connected to i-cache.
  - L2 and L3 are unified for simplicity
- "Havard" design referred to a microarchitecture with **separate** instruction and data memory.
- "Princeton" design referred to von Neumann's **unified** instruction and data memory. This is the most common design.

Sub-Blocking

- Higher block size improves miss rate but also increases miss penalty!
- Idea: keep a large block size, but divide it into smaller "subblocks". Bring only a subset of subblocks on a miss.
  - (pro) lower miss rate
  - (pro) lower miss penalty
  - (con) need separate storage for valid bits for each subblock
  - (con) more complex circuitry.

Reducing Hit Time via reducing associativity and size

- DM $<$ FA (direct mapping $<$ fully associative)
  - Use SA to balance between the two
- Use smaller cache in lower levels (L1, L2, ...)

Reducing Hit Time via Parallel lookup

- Access tag and data in parallel.
- Access each way in parallel.

Reducing Hit Time via Speculative load

- Instead of waiting for a store (potentially conflicting), issue the load speculatively.
  - Once store is resolved, check whether there was a conflict or not. Recover if there was.

Cache Summary

- Miss Rate
  - Increase block size
  - Increase associativity
  - Increase cache size
  - Prefetching
  - Victim cache
  - Compiler
  - Replacement Policy
- Miss Penalty
  - Write buffer
  - Early restart with critical block first
  - Adding more levels
  - Sub-blocking
- Hit Time
  - Set associative cache
  - Add more levels
  - Parallel lookup
  - Speculative loads