Multi-Core Computer Architecture: Storage and Interconnects Week 2 and 3

Aronya Baksy

August 2021

1 Cache Memory

- During instruction fetch and memory access stage, memory speed is a bottleneck
- Hierarchical memory creates the illusion that there is a large amount of very fast memory.
- Smallest and fastest memories are close to the processor. Largest and slowest memories are farthest and they store the entire addressable memory

1.1 Principle of Locality

- **Temporal Locality**: If program accesses some memory location X at time T, then there is a high probability that$X$ will be accessed again at time $T + \Delta T$
- **Spatial Locality**: If program accesses some memory location X at time T, then there is a high probability that location $X + \Delta X$ will be accessed at time $T + \Delta T$
- Put the data at location X, as well as the data near X into the faster cache memory to reduce avg. memory access time

1.2 Terminologies

- **Cache Block**: aka cache line, minimum unit of info. that is either present or not present in the cache
- **Cache Hit**: Data requested by the processor is present in the cache
- **Cache Miss**: Data requested by the processor is not present in the cache and has to be brought in from main memory
- **Hit Time**: Time needed to access cache block and return it to the processor
- **Hit Rate**: Fraction of memory accesses that result in cache hits
- **Miss Penalty**: Time needed to replace a block in the cache with the corresponding block from the next level of memory

2 Cache Organization

- Cache is an array of S (where $S = 2^s$) sets. Each set has E cache lines.
- Each line has:
  1. Valid bit
  2. T Tag bits (where $T = 2^t$)
  3. B bytes of data (where $B = 2^b$)
- Total cache size is $S \times E \times B$
- The address generated by the CPU is divided into t tag bits, s index bits and b bits for block offset.

2.1 Design Choices

- Block placement in cache
- How to know if block is in the cache or not? (block identification)
- Which block to be replace on a cache miss? (block replacement)
- How to handle write miss (i.e. CPU wants to write to block but block is not in cache, aka the write strategy)

2.2 Block Placement

- Assume that main memory has 32 blocks and cache memory has 8 blocks
- Assume that block number 12 from main memory has to be brought into cache

2.2.1 Direct-Mapped Cache

- $12 \% 8 = 4$
- Hence, the block 12 of main memory will be stored in block 4 of the cache
- Hence, if cache size is M blocks, then the N^{th} block of main memory is mapped to $(N \% M)^{th}$ block of cache
- This can also be called a *1-way set associative cache*. Hence there is only one line per set
- Steps involved in direct-mapped cache access:
  1. Indexing: Use the index bits to determine which set
  2. Block Matching: Find the appropriate set block that has a matching tag and valid bit set (not 0).
  3. Word Extraction: Using the block offset, select the correct word from the block and transfer to CPU

2.2.2 N-way Set Associative Cache

- In an N-way set associative cache, the cache is split into sets of size N blocks each.
- Assuming a 2-way set associative cache, each set has 2 blocks, hence there are 4 sets. Hence block number 12 is placed in set 0 ($12 \% 4 = 0$)
- Within set 0, the block can be placed in any of the free cache lines
- Steps involved in set-associative cache access:
  1. Indexing: Use the index bits to determine which set
  2. Block Matching: Within the set, check all the tags to check which line has a matching tag (each access requires N tag comparision in N-way associative cache)
  3. Word Extraction: Using the block offset, select the correct word from the block and transfer to CPU

2.2.3 Fully Associative Cache

- The block of main memory can be placed in any of the free blocks in the cache memory
- This is basically a set-associative cache with 1 set
- Steps involved in fully-associative cache access:
  1. Block Matching: Check all the tags to check which line has a matching tag (each access requires N tag comparision in N-way associative cache)
  2. Word Extraction: Using the block offset, select the correct word from the block and transfer to CPU

2.3 Block Replacement Techniques

2.3.1 Random Replacement

- Uses a psuedorandom number generator to generate the index of the victim block in a set.
- Adds an $O(1)$ overhead to each block replacement action
- Does not take temporal or spatial locality into account

2.3.2 FIFO Replacement

- The block that has been in the cache for the longest amount of time is replaced
- This requires storing all the block references in a queue. Enqueuing and dequeuing operations take O(1) overhead on each operation.
- Also does not take principles of locality into account

2.3.3 Optimal Replacement

- Remove the block with the longest reuse distance in the future
- It cannot be built practically, but it can be modelled given some data.

2.3.4 LRU (Least Recently Used)

- The block that was least recently used in the past (i.e. longest reuse distance in the **past**) will be evicted
- It is the exact inverse of optimal algorithm (if a reference string is run through the optimal algo, and the reverse of that string is run through the LRU algo, then both will yield same answers)
- Maintaining information for LRU is simple for 2-way set associative (each line needs one bit) but becomes expensive for larger associative cache
- The sorted list of access times for each block is re-sorted on each cache access, and this adds an overview of $log_2(N)$ for an N-way set associative cache.

2.3.5 Pseudo-LRU

- Using a binary tree structure, which indicates which child is older and newer.
- On each cache access, the tree is updated. The LRU victim is found by following the pointers to the oldest child (from the root to the leaf)
- The tree structure indicates the pseudo-LRU order. Each level points to the LRU half of the subtree
- At each cache access, the nodes along that path are flipped. If block replacement is neededem then follow the LRU path.

2.3.6 NRU (Not Recently Used)

- The state is kept in 1 bit for each cache block in the set.
- The bit is reset to 0 when the block is installed or referenced
- The bit is set to 1 when it is not referenced but another block in the same set is referenced
- Those blocks with state=1 are preferred for eviction. If all blocks have same state then replace at random.

2.3.7 RRIP (Re-Reference Interval Prediction)

- An extension of RRIP to multiple bits (instead of single bit)
- In a multiple-bit status, the middle bit is set to 1. In case of a hit on that block, the 1 is promoted to the next position. Otherwise, the 1 is pushed back by 1 position
- Near-immediate, intermediate or distant re-references can be predicted

2.3.8 LFU (Least Frequently Used)

- Each cache block has a counter that is incremented on each reference to that block
- When a block is to be replaced, the one with the lowest counter value is evicted
- It requires a^2 comparisions per access (where a is the associativity of the cache) and a constant storage overhead

2.4 Write Strategy

- Caches may be of 2 types: **write-through** or **write-back**.
- In case of a write miss, either:
  - **Write Allocate**: load the block into cache upon a write miss
  - **No-Write Allocate**: Block is modified in memory but not in cache

2.4.1 Write-Through Cache

- In such a cache, a write is performed both on cache as well as the same data in main memory
- A read miss does not need to write back the evicted line contents to the memory

2.4.2 Write-Back Cache

- All writes are performed only inside the cache
- The modified cache block is written back to the main memory only when the cache block is evicted
- Each block contains a dirty bit to identify whether a block has been written to after being loaded into the cache or not

3 Types of Cache Misses

3.1 Compulsory Miss

- Very first access to a block causes a miss (because that block has not been accessed before hence it is not in cache)
- Happens even in cache of ∞ size

3.2 Capacity Miss

- Miss caused by cache not being able to contain all the needed blocks
- Happens when the memory footprint of a program is larger than the cache size
- Occur in fully-associative caches

3.3 Conflict Misses

- These occur in set-associative caches when too many blocks map to the same set
- These occur even in direct-mapped caches

4 Cache Optimizations

- Average memory access time is given as

$$AMAT = \text{Hit Time} + (\text{Miss Rate} \times \text{Miss Penalty})$$

- **Hit Time**: Time to find the block in the cache and return it to processor (indexing, tag comparision, transfer time)
- **Miss Rate**: Fraction of cache access that result in a miss
- **Miss Penalty**: Number of additional cycles required upon encountering a miss to fetch a block from the next level of memory hierarchy

4.1 Reducing Miss Rate

4.1.1 Larger Block Size

- **Advantages**:
  - Utilize spatial locality
  - Reduce compulsory misses
- **Disadvantages**
  - Increased miss penalty
  - More time to fetch block from cache
  - More number of blocks will be mapped to the same location
  - May bring useless data and evict useful data (pollution)

4.1.2 Larger Cache Size

- **Advantages**:
  - Reduce capacity misses
  - Can accomodate larger memory footprint
- **Disadvantages**:
  - Longer hit time (because of more indexing and tag comparisions)
  - Higher cost, area and power

4.1.3 Higher Associativity

- **Advantages**:
  - Reduce conflict miss
  - Reduce miss rate and eviction rate
- **Disadvantages**:
  - Increase in the hit time
  - More complex design than direct mapped
  - More time to search in the set (tag comparison time)

4.2 Reduce Miss Penalty

4.2.1 Multilevel Caches

- First level cache (L1) is small and fast enough to match the clock speed of the processor
- Second level cache (L2) is large enough to capture many accesses that would otherwise have to be serviced by main memory
- **Local Miss Rate**: Number of misses in a cache level divided by number of memory access to this level
- **Global Miss Rate**: Number of misses in a cache level divided by number of memory access generated by CPU
- **Inclusive Cache**: Each level of cache is a subset of the higher cache level (e.g. L1 is a subset of L2)
- **Exclusive Cache**: Each of the levels are exclusive of each other, not subsets of each other (e.g.: L1 and L2 are exclusive)

4.2.2 Prioritize Read Misses

- If a read miss has to evict a dirty memory block, the normal procedure is to write the dirty block to memory, and then bring the missed block into the cache and read it.
- **Optimized**: Copy dirty block to a buffer, read from memory and then write the block (reduce CPU waiting time on read miss)
- In write-through cache, write buffer may hold updated values of block that encountered read miss
- Wait till write-buffer is empty, or search in write buffer before searching in memory

4.3 Reduce Hit Time

4.3.1 Avoid Address Translation for Indexing

- Address translation: from Virtual address generated by CPU to Physical address needed by memory.
- This is done by MMU using page tables and TLBs and is to be done before every cache access.
- Optimization: Use Virtual Address for indexing and physical address only for tag comparision
- **VIPT cache**: Virtually Indexed, Physically Tagged