Week9 Cache Performance

Wednesday, July 27, 2022 10:59 PM

1. Review and compare **Cache Placement Policies**

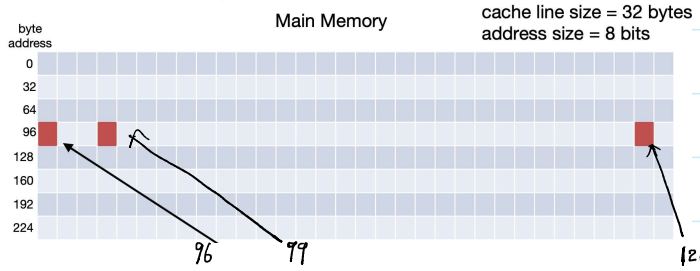
Given a memory space, which has:

- **Cache line size** = **32Bytes**
- **Address size = 8Bit** => **256 Bytes** in total

Attempting to Access bytes with addresses:

- **96_{10}** = 8'b0110 0000
- **99_{10}** = 8'b0110 0011
- **126_{10}** = 8'b0111 1110

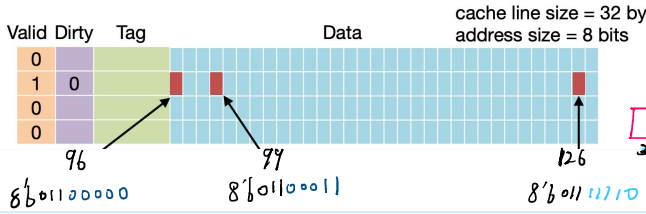
0. In main memory



— When accessing addr = 96; it's a miss. Given a 32 byte cache line, Byte 96 ~ 127 are placed in the cache

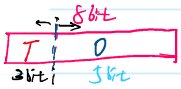
— Then access to 99, 126 will be hits

1. Fully associative cache (N way set associative cache (block# = way#))



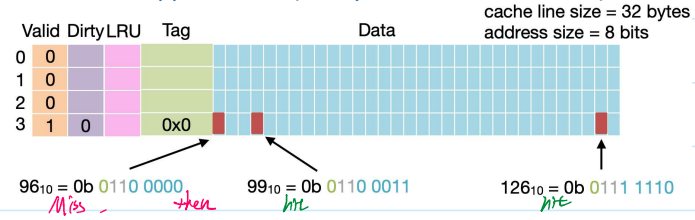
, cache size = 128B, 4 way. cache line size = 32B

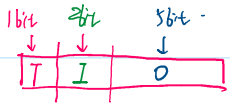
↳ then only one set ⇒ basically a fully associative cache



2. Direct Mapped Cache (1 way set associative cache)

→ 1 way. we get 1 set, set# = cache line # = 4, basically direct map



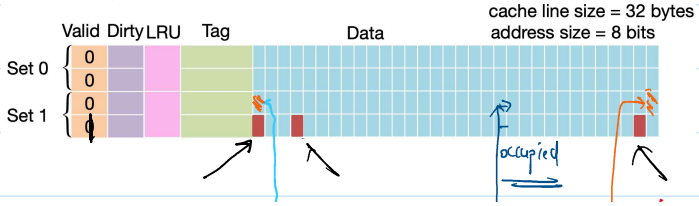


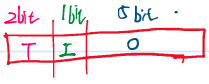
Assuming then we are trying to access 224_{10} to 255_{10} (Address are 8'b 111 00000 to 8'b 111 11111)

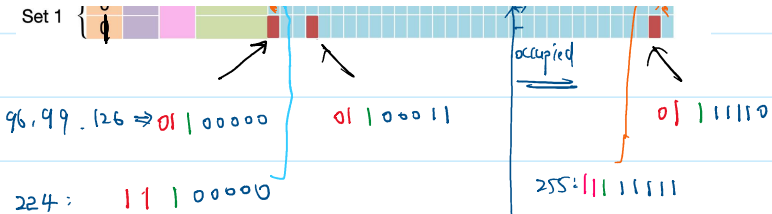
Then we will get Conflict Misses. as Index conflicts with address 96 ~ 127

3. 2-way set-associative cache

→ 2 way. → 64B for each way → 2 entry for each way → 1 bit for Index, 5 bit for offset





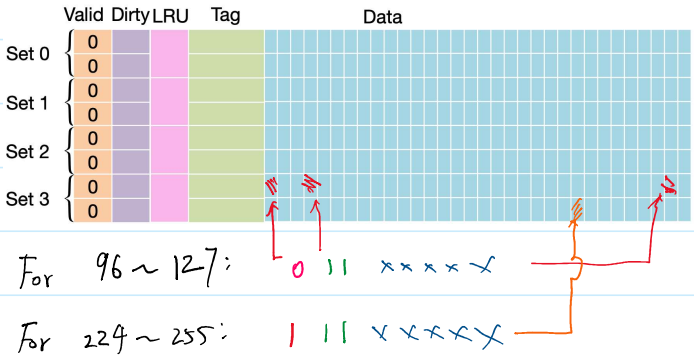


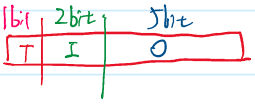
→ Assume we want to access 32~64: 00 | 00000 to 00 | 11111 ⇒ will cause Conflict Miss

3.1. 2-way set-associative cache with more "sets" (or entries per way)

⇒ this is a 256 B cache:

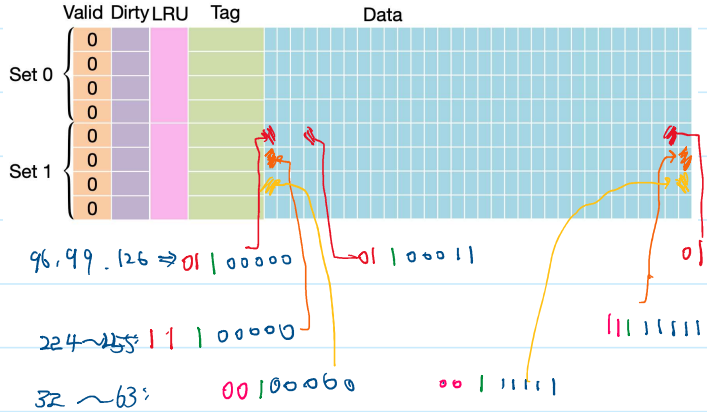
$\frac{256}{2 \times 32} = 4$ entry / way. then need 2bits for indexing

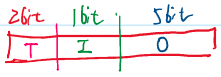




3.2. 4-way set-associative cache

⇒ 256 B cache: $\frac{256}{4 \times 32} = 2$ entry / way. 1 bit for indx





4. Cache placement Summary and Notes

A. Cache structure

Given cache size, cache line size:

For **n-way set associative cache**, we can figure out the cache layout as follows:

- The entry per way, or namely the number of **"sets"**
  - A set includes all (**#n**) ways of that set index
  - And for each way, we have the same **# of sets** for each way
- **CacheSize** = **# sets** * **# way**s * cache **line size**

Therefore, direct-mapped or fully associative caches are two special cases:

- **1-way set-associative**(#sets = Cache_size / cache_line_size) cache <=> **direct mapped** cache
- **N-way (#sets = 1) set-associative** cache <=> fu**lly associative c**ache

For LRUs, LRU bits tells us which way is the **least recently used** in the **set**

B. Cache address partition

- **Cache line size determines # of offsets (for byte or word)**
- **Index is determined by the number of sets**
- **Tag bits are all address bits got left**

C. Pros and Cons

- **Fully associative:**

- **Pros: No conflicts**
- **Cons: A lot of hardware resources for tag matches**

- **Direct Mapped:**
  - **Pros: Only need to check one entry in the cache, fast and cheap**
  - **Cons: Too many conflicts**

- **Set Associative:**
  - **Pros: Less hardware to fully, and better hit rates comparing to direct mapped**
  - **Cons: Still have conflicts**

5. Cache review:

Write Policies - Allocation Policies Pairs

a. Write- Policies

The cache contains a subset of data that is stored in memory. When we perform a write operation (like SW), we need to ensure that anyone trying to access the data gets the most up-to-date copy. There are two different policies to handle Writes:

Write-back:

- On a write, we **only write to the cache**, and **not** write to the **memory**
- To indicate that the data **in the cache** is **more up-to-date** that the **data in memory**. We set the **dirty bit of that cache line to 1**. When **this line gets evicted from the cache**, the **dirty bit indicates** that the **line needs to be written to memory**
- The data in the cache will be **more up-to-date** than the data in memory for a short period of time. This is not a problem because **if we want to access this data**, we will **search for it in the cache before** looking for the memory
- **Writing to cache is much faster** than **writing to memory**, which makes the write latency of write-back caches smaller than write-through caches
- The **write-back policy** also **reduces the number of writes to memory**. Once we bring a piece of data into the cache, we **may write to it several times before evicting** it from the cache. With the **write-back policy**, we **only have to write** to the memory **when the line is evicted**, instead of every time it is updated.

Write-through:

1. On a write, data is written to both cache and main memory
2. Writing to cache is fast, but writing to memory is slow. This makes write latency in write-through cache is slower than write-back
3. Though the hardware implementation of write-through cache is simpler

b. Allocation Policies

When get a write miss, we also need to decide whether to put the missed block into the cache

Write-allocate

- On a write miss, pull the block you missed into the cache

NO Write-allocate

1. On a write miss, do not pull the block you missed into the cache, only the memory is updated
2. On a read miss, the data is still loaded into the cache

c. Common Combinations

Note: For read miss, both combinations will load the block into the cache

Write-through / No-write allocate

On write hits: write to **both** the **cache** and **main memory**
On write misses: the main memory is updated and the **block is not brough**t into the cache

- If **read to the same block** occurs after the **write-miss**, there would be an unnecessary miss
  - If we allocate the write at write miss, we will have the block in cache and get a hit
- This policy is useful for when **writing to a piece of data** that we **don't plan to access** again

Write-back / Write allocate

On write hits: only the **cache's copy is updated**, and therefore the dirty bit is **set to 1**
On write misses, **the corresponding block is brought into the cache**, updated, the dirty bit is set to 1

If the same block get accessed, all subsequent writes would be hits. Dirty bit would remain at 1 until the block is evicted, at which point it would be moved back to main memory

Replacement Policies

When we decide to evict a cache block to make space, we may choose one of the replacement policies

- **LRU (Least Recently Used)** - select the block that has been used the **furthest back in time** of all the blocks
- Random - **Randomly** select one of the blocks in the cache to evict
- MRU (Most Recently Used) - Select the block that has been used the **most recently** of all the blocks

3Cs for misses

- **Compulsory Miss**

- Caused by the first access to a block that has never been in the cache

- **Capacity Miss**
  - Caused when the cache cannot contain all the blocks needed during the execution of a program
  - Occurs when blocks were in the cache, replaced, and later retrieved
- **Conflict Miss**
  - Occurs in set-associative or direct mapped caches when multiple blocks compete for the same set
  - Never appear in a fully-associative cache

2. Improving Cache Performance through Programming techniques

The program under test is shown below:

```
1  int sum_array(int *my_arr, int size, int stride) {
2      int sum = 0;
3      for (int i = 0; i < size; i += stride) {
4          sum += my_arr[i];
5      }
6      return sum;
7  }
```

A. Array strides

Considering the following Cache structure

Parameter	Value
Size of int	4 Byte
Array size	32 elements
Cache placement?	Fully associative
Cache Line size	16 Bytes
# of Cache Line	16

- **Stride = 1**

- **Stride = 2**

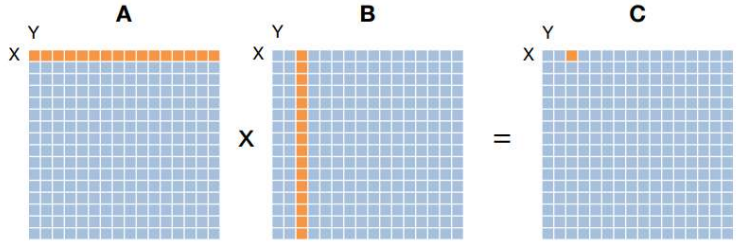
- **Stride = 4**

By observing the cache access pattern, we find:

if **stride size(in byte) >= block size**. You **cannot** take advantage of bringing the entire cache line

B. Matrix Multiply

0. Review of MxM



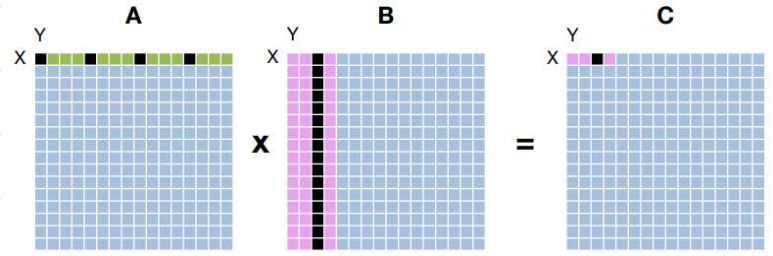
Note: in this course, arrays are stored in **row-major order**

1. Matrix Multiply and Cache

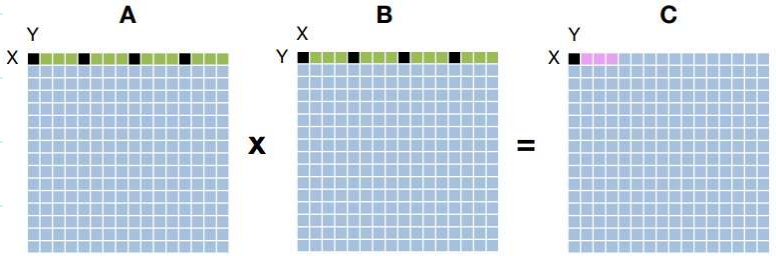
- Considering multiplying two **4x4 matrices.** (16Byte x 16Byte)
- And **Cache is configured** as:

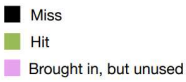
Cache placement?	Fully associative
Cache Line size	16 Bytes
# of Cache Line	16

a. Brute force without any modification



b. Transposing matrix B and make better use of the cache

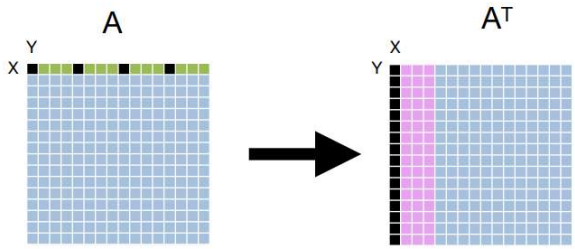




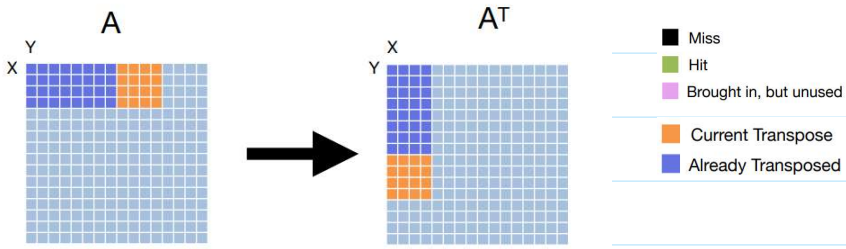
C. Matrix Transpose and Blocking

We learned that transposing B will improve cache utilization, how to transpose a matrix efficiently?

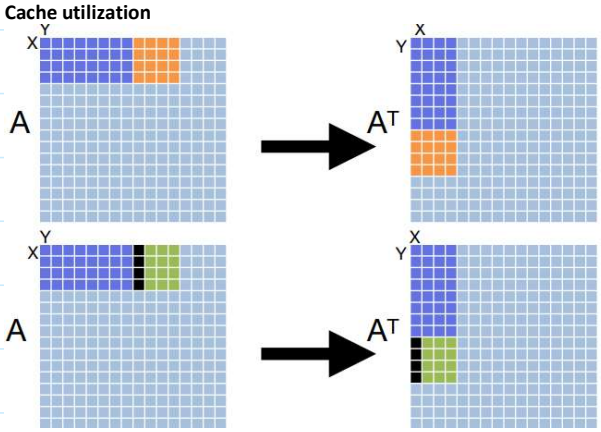
a. Brute force without any modification



b. Transpose with blocking



Cache utilization



3. Analyzing Cache Performance

3.0. Multi-level Cache

A. Common cache levels:

- L1 cache
  - Embedded in the processor chip
  - Fast, but the capacity is limited
- L2 cache
  - Embedded on the processor chip, or on its own separate chip
  - Reduces L1 miss penalty
- L3 cache
  - On a separate chip
  - Reduces L1 and L2 miss penalty
- Higher level caches are uncommon

B. Higher cache characteristics

- For L2 cache, or higher level caches
  - L2 is bigger than L1
  - L2 is **accessed only** if the **requested data is not found** in L1
  - L2 takes **longer** to access because it is larger and farther away from the processor
    - All data in L1 can be found in L2 as well *(depends on policy)
    - If the line in L1 is dirty when evicted, you update the copy in L2 *(depends on policy)

C. Real cache info in i7-6500u

Core i7-6500U (Skylake)

- L1 data cache
  - Size: 32 kB
  - Associativity: 8
  - Number of sets: 64
  - Way size: 4 kB
  - Latency: 4 cycles Link
  - Replacement policy: Tree-PLRU (with linear insertion order if empty) Link 1 Link 2
- L2 cache
  - Size: 256 kB
  - Associativity: 4
  - Number of sets: 1024
  - Way size: 64 kB
  - Latency: 12 cycles Link
  - Replacement policy: QLRU_H00_M1_R2_U1 Link
    - Similar to the Cannon Lake L2 policy, but:
      - If the cache is empty (after executing the WBINVD instruction), blocks are inserted from right to left
      - The initial ages of blocks inserted into an empty cache can depend on the previous state
      - See also Vila et al.
- L3 cache
  - Size: 4 MB
  - Associativity: 16
  - Number of CBoxes: 2
  - Number of slices: 4
  - Number of sets (per slice): 1024
  - Way size (per slice): 64 kB
  - Latency: 34 cycles Link
  - Replacement policy: Adaptive Link

3.1. Terminology for AMAT analysis

A. For single level cache

- **Hit Rate**
  - $\frac{\#\ of\ Hits}{\#\ of\ Accesses}$
- **Miss Rate**
  - 1 - hit rate
- **Hit Time**
  - The time that it takes for you to access an item on a cache hit
- **Miss Penalty**
  - On a miss, the time it takes to access the block after discovering that is not in the cache

- **Average Memory Access Time (AMAT)**
  - AMAT = hit time + miss rate * miss penalty

B. Extend concepts to multi-level caches

- **Hit Rates** (global or local)
  - Local Hit Rate: $\frac{\#\ of\ Hits\ @\ this\ level}{\#\ of\ Accesses\ to\ this\ level}$
  - Global Hit Rate: $\frac{\#\ of\ Hits\ @\ this\ level}{Total\ \#\ of\ Accesses}$

- **AMAT** for **higher-level cache** (illustrate with 2-level cache)
  - The equation Still holds true: AMAT = hit time + miss rate * miss penalty (local miss rate)
  - AMAT = L1 hit time + L1 miss rate * L1 miss penalty
    - Where, L1 miss penalty = L2 AMAT = L2 hit time + L2 miss rate * L2 miss penalty
  - AMAT = L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)

C. AMAT examples

a. AMAT example (Single level)

- Given: Hit **rate = 0.9**, hit **time = 4 cycle**s, **miss penalty = 20 cycles**
  - AMAT = 4 + 0.1 * 20 = 6 cycles
- Given: Hit **rate = 0.75**, hit time = **5 cycles**, miss **penalty** = **24 cy**cles
  - AMAT = 5 + 0.25 * 24 = 11 cycles

B. AMAT example (2-Level cache)

- Given L1 **hit rate = 0.75**, **L1 hit time = 4 cycles**;
- L2 **hit rate** = **0.9**, L2 **hit time** = **6 cycles**; L2 **miss penalty** = **20 cycles**
  - AMAT = 4 + 0.25 * (6 + 0.1 * 20) = 6 cycles

- Given L1 **hit rate = 0.6**, **L1 hit time = 5 cycles**;
- L2 **hit rate** = **0.95**, L2 **hit time** = **8 cycles**; L2 **miss penalty** = **40 cycles**
  - AMAT = 5 + 0.4 * (8 + 0.05 * 40) = 9 cycles

D. Impacts that affect AMAT

- Higher associativity (More ways)
  - Increase hit time (due to larger hardware delays)
  - Lower miss rate because of fewer conflicts
  - Miss penalty is unchanged, as replacement policy runs in parallel with fetching missing lines from memory
- More entries (More sets)
  - Increased hit time
  - Lower miss rate due to increased capacity and fewer conflicts
  - Miss penalty is unchanged
  - At some point, the increase in hit time for a larger cache may overcome the improvement in hit rate. Leading to a decrease in performance
- Larger cache line size
  - Mostly unchanged, but may be slightly reduced as cache is smaller
  - Miss rate decrease initially, due to spatial locality. Then increases due to increased conflict misses (Cache size is constant, but #entries is reduced)
  - Miss penalty rises with larger block size

	Hit Time	Miss Rate	Miss Penalty
Higher Associativity	++	--	unchanged
More entries	++	--	unchanged
Larger block size	+	- Then +	+