Topics:
- AMAT
- Block Sizing + Alignment
- Decomposition of Addresses
  - Block Offset
  - Block # ("Tag" + "Index")
- Cache Metadata
  - Tags
  - Valid
  - Dirty
  - LRU Counter
- Write Policies
- Type of Caches
  - Fully-Associative
  - Direct Mapped
  - Set Associative

Midterm → Cache Review → Virtual Memory

Cache Review

This lesson is a review of caches. Beginning with the structure of the cache itself, including set associative and direct mapped caches. Then the lesson discusses replacement policies, specifically the LRU policy. The final portion of the lesson covers write policies; write back, write through,write allocate, and no-write allocate.

Locality Principle

Things that will happen soon are likely to be close to things that just happened - Like Branch Prediction

Things that happened recently are likely to happen again.

An example of locality is: it rained 3 times today, so it is likely to rain again today.

Memory References

Temporal Locality: If an address has been accessed recently, it will likely be accessed again.

Spatial Locality: If an address has been accessed, it is likely addresses close to it will be accessed.

caches are exploiting spatial and temporal locality and overcoming the problem of having a slow and large main memory, just like borrowing books from the library is exploiting spatial and temporal locality and overcoming the problem of accessing a huge library slowly.

Locality and Data Accesses video is wrong one - notes when video is correct

Cache Lookups

A cache is a small memory for fast data lookup, to save the time that would be spent going to main memory.

Cache needs to be Fast, so it must be small.

Since it is small, not all the data will fit, so when memory access is required one of two things will happen:

-a Cache hit - the data from the memory location is in the cache

-a Cache miss - the data from the memory location is not in the cache

Cache hits return the data fast, cache misses are slow because they have to go to memory. So caches should be designed to have as few misses as possible.

Cache Performance

The properties of a good cache are:

1. A short AMAT (Average Memory Access Time)

   AMAT = Hit Time + Miss Rate * Miss Penalty = (1 - Miss Rate)(Hit Time) + (Miss Rate)(Miss Time)

   To achieve a low AMAT:

   A low hit time is required - which means the cache must be small and fast.

   A low miss rate is required - which means the cache must be large or smart.

The miss penalty is usually tens or hundreds of processor cycle times.

Miss Time = Hit Time + Miss Penalty

In a well designed cache:

hit time < miss time

miss time > miss penalty
hit rate > miss rate
hit rate is almost 1

Cache Size in Real Processors

Modern processors have several caches

L1 (level 1) ::

size = 16k - 64k bytes
hit rate = 90%
hit time = 1 - 3 cycles

Cache Organization

Two basic criteria for a cache are:

Determining a hit and a miss -

hit = the data at the requested memory location is stored in the cache
miss = the data at the requested memory location is NOT stored in the cache

Determining what to kick out of the cache-

the cache is not large enough to hold all the data required by the program, so it must throw out data to make room for new data.

aka "Cache Line Size"

Block Size = the amount of data stored for each memory address. The block size should at least be as large as the single largest access that can be done in the cache. Usually 32 - 128 bytes work well for a block size. We'd want multiple memory accesses (with different but spatially close addresses) to hit the same cache entry in order to benefit from spatial locality with a medium sized (32 - 128 byte) cache size.

Cache Block Start Address

Where should blocks begin in memory?

Blocks should not overlap - otherwise more than one copy of data will be stored in the cache.

Blocks should start at consistent locations - this reduces cache complexity.

Therefore: blocks should be aligned to match the cache size ex: block size of 64 bytes would go from 0-63, 64-127, etc.

Ex: If blocks are 64 bytes in size, then we'd only fetch 64 byte blocks at block-aligned addresses.

Blocks in Cache and Memory Cache "Lines" are "slots" that Memory "Blocks" occupy

Memory has blocks of data and caches have lines of data. The line size equals the block size.

In 2k byte cache the following block sizes are NOT good:

1 byte -- does not exploit spatial locality and word accesses will need to access multiple lines

48 byte -- this is not a power of 2, which makes the cache more complicated

1k byte - this is too big, only 2 lines will fit in the cache

Choose a block size that:

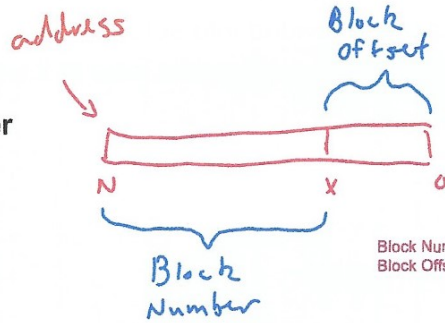
is a Power of 2

exploits spatial locality

is small enough to have a significant number fit in the cache.

Block Offset and Block Number

The address is divided into
- block offset
- block number



Given the block size, you can figure out how many bits you need to be able to index each byte in a given block c said block size. Use that number of bits of an address as the Block Offset. The remaining bits in the address are the Block Number.

Block Number = Which Memory Block does the data belong to?
Block Offset = Within a Memory Block, where in that block is the data that we want?

Cache Tags

Tags determine which block is to be accessed. The tag is the most significant bits of the address.

The tag always:
- contains at least one bit from the block number

Note that the Tags MIGHT NOT have the same # of bits as our Block #'s. For ex, we might have more Block #'s than the # of Cache Lines/Tags in the system, so we wouldn't need to use all of the Block # bits, just a sub-portion of them. So # of Tag Bits <= # of Block # Bits.

The "Lines" are made up of multiple fields. The "Cache Tags" are a separate field that we use to compare a given "Block Number" with to tell us if this Line has the data for that Block Number. If a Cache Tag entry matches the Block Number, then its data can be found in this Cache Line Otherwise, it's a cache miss.

The Cache Tags are a separate structure used along with the Cache Lines. If the Tag matches the Block Number, then the corresponding Cache Line has the BLOCK that we're looking for. Once we find the Cache Line where the Block Number matches the Tag, then we use the Block Offset to pick out the data we want within that Block.

Valid Bit

The valid bit is stored in the cache, one bit for each cache line. The valid bit stores information about the validity of the data. If valid = 0, the data is not valid and is considered not a hit. If valid = 1, the data is valid and is considered a hit.

The Valid Bit is another field alongside the Cache's Lines "Data" and "Cache Tags" that te us if the Line is valid. Initialized to 0 on boot, set to 1 when we fetch data from memory an populate the Cache Line's Data field.

Types of Caches

Fully Associative - any block can be placed in any line in the cache — Requires that we search the ENTIRE cache. No smart indexing.

Direct Mapped - a block can only go to a specific line in the cache — multiple blocks can map to the same line, but that's the only line they'll EVER map to. Smart indexing to quickly find the line.

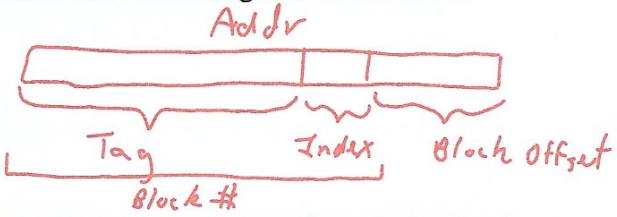
Set-Associative Cache - A block can be placed in N number of lines in the cache

^ middle ground option

A Block can only ever map to a specific SET, but within a SET there's N number of lines that that Block can possibly go to. So smart index to the SET, and then can do a linear search within the N lines in that SET.
Fully Associative -> Set-Associative where N = Size of the entire cache (# of Lines in the Cache) - so the Cache has 1 large set.
Direct Mapped -> Set-Associative where N = 1 (Set size of 1) - so every Cache Line is its own Set.

Direct Mapped Cache

More than one memory location will map to the same cache line. The least significant bits of the block number are used to determine the cache line, these bits are called the block index. These index bits do *not* need to be stored with the cache line - a cache line whose index is K can only store blocks whose index is K. But we do need to store the remaining bits of the block number - the tag bits.



Upside and Downside of Direct Mapped Caches

Upsides:

Hit Time is very good

Only one line is checked for a hit or miss, so it is fast.
It is also cheaper because only one line is checked.
It is energy efficient.

The "Index" bits tell us which cache line to use. So we split up the Address by
1) Carving out the # of bits we need to index into the block (Block Offset - determined by the block size).
2) Carving out the # of bits we need for the Index (which is determined by the number of SETs we have - For Direct Mapped, # of Sets == # of Cache Lines).
3) The rest are for the Tag.

Downsides:

The block must go in only one line. This can lead to conflicts in the cache, which increases the miss rate.

Conflicts could increase miss rate because a memory access pattern happens to continuously keep kicking things out of the cache. This is (sorta) similar to the "worst case patterns" that a History Based Branch Predictor has.

There's only a Conflict when two addresses have the same Index but Different Tags!

If they have the SAME INDEX AND TAG (but different Block Offsets), then they have the same Block Number, which means that they're the SAME BLOCK (a cache hit - not a conflict). So those two addresses are referencing different pieces of data that exist within the same Memory Block.

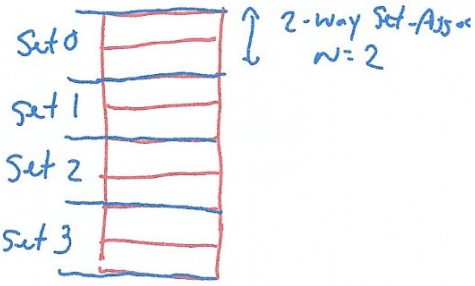
Number of Index Bits = Log2(# of SETs in the Cache)
or 2^(x) = # of Sets

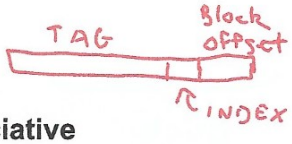
Similar to Direct-Mapped Caches, a block can only be mapped to a specific SET, but within a set there are multiple LINES where that specific block can occupy a line. So the term "N-Way Associative Cache" means that each Set has N lines within it. So we can calculate the number of Sets = (# of total Cache Lines) / (# of Lines per set aka N)

Set Associative Caches

N-way set associative = the cache is divided into sets of N lines each.

2 Way set associative has sets of 2 lines, N= 2.





Offset, Index, Tag for Set-Associative

The address is divided into:

Offset: which portion of the block is to be accessed, the number of bits used for this are determined by the block size.

Index: which set is to be accessed, determined by the number of sets in the cache

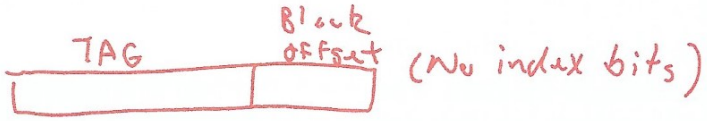
Tag: the unique portion of the address, used to identify the correct address to be accessed.

Everything in a Set has the same "Index" bits, so there's no need to keep track of that in our Cache Structure.

Fully Associative Cache

Any block can map to any line.

The address is divided into:



Offset: bits used to determine which part of the line is being accessed.

Tag: Used to identify the line to be accessed.

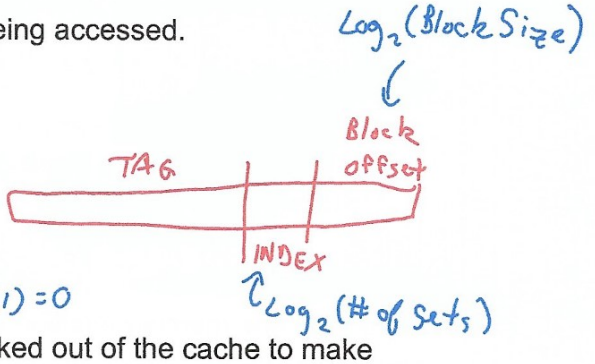
Direct Mapped and Fully Associative

Each Line is its own "Set"

Direct Mapped = 1-way Set associative

Fully Associative = N-way Set Associative

Here, there is 1 set, so $\log_2(1) = 0$ so 0 index bits



Cache Replacement

When the set is full and there is a miss, a block needs to be kicked out of the cache to make room for the new one.

Methods for choosing which block to kick out of the cache:

Random - pick one at random to replace

FIFO - kick out oldest block

LRU - least recently used, the block that has not been accessed the longest is the one to replace. - good policy but not easy to do

NMRU is an approximation of LRU

NMRU- not most recently used. Only track the most recently used block, then randomly kick out one of the other blocks.

Implementing LRU

LRU exploits locality.

The cache now has the data, tag, valid bit, and LRU bits.

The LRU bits are valued from 0 to N. The LRU block = 0, and the MRU block = N.

For an N-Way Set-Associative Cache (which means that in each Set of the Cache, there are "N" lines), we need "N" LRU Counters (one for each Cache Line in the Set), where each counter is Log_2(N) Bits

Algo for updating an LRU counter. For whatever entry you're updating, remember the original counter value (its value before you change it to be the Most Recently Used value). For all the other LRU counters that have values greater than this value, decrement them by one. For all other LRU counters that have values lower than this value, leave them alone. Finally, update your original counter so that it now holds the greatest value (indicating it is the most recently used)

When replacing a block, the least recently used block (LRU = 0) is replaced and the LRU is changed to equal the highest value (N). All other LRU bits are decremented by one. This will designate a new LRU block.
Implementing LRU's methods are hardware intensive.

We don't need an LRU for Direct-Mapped Caches, since each line is its own Set, so all updates will cause a replacement.

Write Policy

Write-Allocate is the more popular "Write-Miss" policy.

Write Miss Policies

These policies are for Writes requests for Blocks that AREN'T in the Cache (MISS), so we need to write to Main Memory. In that scenario, after we write to Memory, do we want to bring the Memory Block in to the cache?

Write allocate - the block is written to the cache
No-write allocate - the block is not written to the cache.

Write Hit Policies

or Cache HITs on Writes, do e also want to push the rites/updates to Memory?

Write-through - the memory is updated when the cache is updated. This is an unpopular method.
Write-back - only write to the memory when the block is replaced in the cache.

Usually write-allocate is paired with write-back.

So that if you have a Write-Miss, the block is brought to the cache (because of Write-Allocate) so that any future Writes to that same block will result in a hit and then you can keep the updates in the cache. This prevents a lot of writes to main memory and keeps it all in the cache.

Write-Back Cache

If a block is modified, it needs to be written to memory at replacement, if not modified, then the block does not need to be written.
A dirty bit is used to track if when the block is written.

Dirty = 0 : the block was not modified
Dirty = 1: the block was modified and needs to be written to memory when replaced.

The Dirty bit is always Set = 1 on Write instructions.

So the Dirty bit adds another field that we need to add to our Cache Line metadata (along with the "Valid" fields, "Tag" fields, and "LRU Counter" fields).

Ex: 4kB, 4-Way SA, 64 Byte Line, Write-Back, Write-Alloc Cache. CPU uses 64-Bit Addresses.
1) How many bits for Block Offset? 2^x = 64 Byte -> x = 6
2) How many bits for Index?
- First, how many Sets do we have? Cache is 4096 bytes with 64 byte lines, so the cache has 4096 / 64 = 64 Lines. The Cache is 4-Way SA, so each Set has 4 lines, which means there is 64 / 4 = 16 Sets.
- How many bits do we need to index into the 16 sets? 2^y = 16 -> y = 4
3) How many bits for Tag?
- Addresses are 64 bits, we use 6 bits for the Block Offset and 4 bits for the Index, so we've used 10 bits so far. 64 - 10 = 54 bits left, which are all used for the Tag.
4) What actual bits do we need to use for a Cache Line?
- 1 Valid Bit
- 1 Dirty Bit
- 54 Tag Bits (to compare with the addresses)
- 2 LRU Bits (since each set is 4-Way, we need to indicate, within those 4 Lines of a single Set, each of their respective LRU #s).
- The Actual Memory Block data (64 bytes, or 64 * 8 = 512 bits)