**April 2, 2020**

Processor

Registers

Cache

Main Memory

Secondary Memory (Disk)

Larger

Slower

Cheaper

**General notes about the hierarchy structure:**

* The goals of the hierarchy:
  + To make the memory system look as big as the largest level and operate as fast as the fastest (top) level from the perspective of the processor.
  + Reduce the average access time to memory.
* Every byte represents 1 memory location (i.e. every byte is an addressable location in the memory address space).
* Each level in the hierarchy contains a number of memory locations (bytes) that is a power of 2.
* At each level, memory locations are “grouped” into blocks (a.k.a. lines) and each level consists of a subset of the blocks that are available at the next lower level.
* The lowest layer contains every addressable memory location (i.e. the entire address space).
* The block/line size must be a power of 2.
* 1 K = 1024 bytes ≠ 1000 bytes
* EVERYTHING IS A POWER OF 2
* Cache (and registers) will be constructed using a volatile memory technology
  + Volatile means it loses its contents when power is removed
* Main memory and levels below it will be non-volatile technologies
  + Non-volatile means the memory maintains its data when no power is supplied
* There is a 10x – 100x access time penalty for each successively lower level.
* Built to exploit 2 principles of locality:
  + Temporal locality: This is the observation that if a memory location is used, it is likely to be needed again soon. (exploited by moving items needed to higher levels of the hierarchy so the next time they are needed it is faster to get them)
  + Spatial locality: This is the observation that if a memory location is used, it is very likely that its neighbors will be needed soon. (exploited by using blocks at each level so when one location is moved up so are its neighbors)

**Hierarchy operation:**

The processor needs to access a memory location for instructions or data. The cache is searched first to see if it contains the needed location.

If so, CACHE HIT. The data is read from the cache and used by the processor.

If not, CACHE MISS. The system then moves to the next lower level in the hierarchy to search for the needed location there. Once found, the whole block containing the specific memory location needed is moved to the next higher level until reaching the cache where it is accessed by the processor.

**Average access time calculation:**

For memory having 1 cache level and a main memory level:

avg\_memory\_access\_time = (cache\_hit\_rate \* cache\_access\_time)

+

[(1 - cache\_hit\_rate) \* main\_memory\_access\_time]

**Average memory access time examples:**

**EX 1:**

Cache size = 2K = 2048 bytes (each byte is 1 location in memory)

Cache memory access time = 2 clock cycles

MM access time = 20 clock cycles

Hite rate = 85%

Avg\_memory\_access\_time = (0.85 \* 2) + (0.15 \* 20) = 4.7 cycles

**EX 2:**

Cache size = 4K = 4096 bytes (each byte is 1 location in memory)

Cache memory access time = 3 clock cycles

MM access time = 20 clock cycles

Hite rate = 93.5%

Avg\_memory\_access\_time = (0.935 \* 3) + (0.065\* 20) = 4.105 cycles

**EX 3:**

Cache size = 8K = 8192 bytes (each byte is 1 location in memory)

Cache memory access time = 4 clock cycles

MM access time = 20 clock cycles

Hite rate = 94.435%

Avg\_memory\_access\_time = (0.94435 \* 4) + (0.05565\* 20) = 4.8904 cycles

These examples show that it is important to note that as cache size increases so does its access time. Eventually, the increased access time will outweigh the improvements in hit rate.

**What does cache memory look like?**



Valid bit is a single 1-bit flag indicating whether the data is good or not

Block ID is a variable-bit value used to uniquely ID each block

Block of data is the actual memory data (bytes of data) residing in each block inside the cache

**Cache Mapping:**

Definition: This is the method used to assign a cache block to a main memory block

Think of the cache blocks as parking spots. The UA parking lots (all of cache memory) can’t hold all the cars for every student (the main memory blocks). So, main memory blocks (student cars) are assigned specific locations (cache blocks) where they can park.

3 ways to make these assignments:

1) Direct-mapped cache

Each MM block is assigned exactly one CM block.

The math: (MM byte address) ÷ (# of bytes per block) [ignoring the remainder] = MM block #

(MM Block #) % (total number of blocks in cache) = CM block # where MM block must go

But, the system doesn’t do the math like this. Instead, it uses the bits in the MM address to figure out the mapping. This is described in the excel spreadsheet.

Offset field 🡪 bits used to pick the specific byte from the block

Index field 🡪 bits that show the exact CM block to which the MM block is mapped (this tells the memory controller where to find the MM block in the CM)

Tag field 🡪 the unique “binary code” that identifies a MM block from all the other MM blocks that could be in that CM block

Advantages:

Easy to implement

Fast to search for MM block in the CM

Simple hardware

Disadvantages:

Results in poor hit rate if two commonly used MM blocks are mapped to the same CM block.

April 7, 2020

Direct-mapped is only one possible cache mapping policy. We already said its primary disadvantage is that if two commonly used MM blocks are mapped to the same cache block, performance will suffer.

The example in the spreadsheet shows how the hit rate is impacted because MM blocks 0 and 8 both map to CM block 0. They replace each other resulting in a poor hit rate and poor memory system performance.

So, we need more flexibility in where a MM block can be placed in the cache.

Other mapping policies:

2) Set associative mapping

In this scheme, the cache is broken into sets and each cache set contains a power of two number of CM blocks.

One CM block per CM set 🡪 SAME AS DIRECT MAPPING

Two CM blocks per CM set is 2-way set associative.

Four CM blocks per CM set is 4-way set associative.

Eight CM blocks per CM set is 8-way set associative (fully-associative for this example)

In each case, the mapping from MM is to a specific CM set, and then the system can decide to put a MM block anywhere within that CM set. This is what UA uses for a parking scheme. Students are assigned to park in a specific lot (mapping to a specific CM set), but they can park anywhere within the lot (CM set).

To get the CM set to which a MM block is assigned:

(MM block #) % (# of CM sets) = CM set assigned

OR

Use the index field of the binary MM address (The index field is the pointer to the CM set assigned.)

Look at spreadsheet to compare direct-mapped, 2-way, 4-way, and 8-way set associative structures. 8-way SA is the limit for this example because there are only 8 CM blocks.

Notice how for each architecture, the binary address of the MM location is broken down into fields of different sizes. The Offset field doesn’t change because the block/line size stays the same. However, as the number of CM sets changes, the Index and Tag fields have to change accordingly.

Advantages:

This is the compromise between direct-mapped and fully-associative; can improve hit rate and performance

Disadvantages:

Still has to search multiple CM blocks to determine hit/miss. So, longer CM access times.

Large cache size (more TAG bits)

More complex hardware

3) Fully associative mapping

In this scheme, the cache is broken into only 1 set containing every CM block. So, a MM block can go into any CM block. Fully associative is N-way set associative when N = the total number of CM blocks.

To carry the parking analogy forward, this is equivalent to UA allowing students to park anywhere on campus. Imagine how long it would take for a 3rd party to find your car!

This is the most flexible mapping policy but it comes with the longest search times to determine a cache hit/miss because every CM block must be searched.

Advantages:

The most flexible mapping policy

Could possibly increase hit rate

Disadvantages:

Longest CM search times

Larger cache size (more TAG bits)

More complex hardware

Now, work the example on the right side of each worksheet. This represents a program executing on the CPU and generating MM locations from which it needs to READ. The leftmost column shows the MM locations to access. The next 3 columns represent the binary MM address broken into its three fields. Then we have the MM block # containing the MM address needed, the CM set, and the CM blocks to which this MM block is mapped. Finally on the right side is whether the address reference created a CM hit or miss during execution.

Below this table are some stats that are things your Lab #7 code will also have to compute including:

Best possible hit rate

Actual hit rate (calculated after you complete the table)

Total cache size

For the more flexible mapping policies, a new problem must be addressed: when there is a choice of where to put a MM block in a CM set, where does it go? If the CM set is already full of valid MM blocks, some MM block already in the CM has to be “replaced”. These are called CM replacement policies.

**Cache Replacement Policies:**

Definition: The technique used to decide which cache block will be overwritten when a miss occurs.

Direct mapped caches don’t really have a replacement policy. It is sort of built into the direct mapping. The new MM block coming into the cache has but one spot to go so the block that is already there gets overwritten.

For set-associative (and fully associative), some method must be used to decide which block in a particular CM set will be overwritten. There are three commonly used methods:

1) random – any block in the set is chosen for replacement randomly. This technique does not necessarily help to keep two commonly referenced blocks in the cache simultaneously.

2) Least-recently used (LRU) – replaces the MM block in the CM set that has not been referenced for the longest time. This technique can create a good hit/miss ratio, but it does require more hardware to keep track of past references.

3) First-in-first-out (FIFO) – the first MM block moved into the CM set is the first MM block to be replaced. This technique does not necessarily help to keep two commonly referenced blocks in the cache simultaneously. But, it requires less hardware than LRU.

April 9, 2020

The final complexity deals with writing to something in the cache. Remember, the cache contains COPIES of locations in MM. If the cache copy is modified by a write operation, how is MM updated to reflect this new data?

**Cache Write Techniques:**

Definition: This deals with when info that is written into a CM block is also written into main memory.

Why is this important?

Once a write into a cache is performed, the data at the cache location is no longer the same as the data AT THE SAME ADDRESS in main memory (i.e. there are 2 copies of the MM location and they contain different data). This is called memory coherency, INCOHERENCY at this point since data at the same addresses but in different parts of the hierarchy hold different values.

There are two primary techniques for this:

1) Write-through – Every time the cache is written, the data is written into main memory as well. This is easy to implement, but is not the most efficient since multiple writes into the same cache location require multiple updates to main memory.

2) Write-back – A cache block is written to main memory only when it is being replaced in the cache and only if it was written to while in the cache. This is usually determined by adding an extra bit to the cache block, called the dirty bit, that indicates if a write to this block occurred.

The Dirty Bit represents another bit of overhead info that must be maintained in the cache for each CM block. So, it makes the CM larger and must be included in the total CM size calculations. The Dirty Bit is initialized to zero upon power up, indicating the data in the CM has not been changed (doesn’t really matter at this point since it would initialize to “invalid” as well). When a MM block is moved into the cache, its Dirty Bit remains low until there is a WRITE operation to a location in the block. At that point, the Dirty Bit gets set to tell the system that the data in this MM block no longer matches the data in the same MM block in MM. The system doesn’t know WHICH EXACT location in the block got changed, but that something did. As long as this MM block stays in the CM, it’s OK that it doesn’t match its copy in MM. However, as soon as the block is replaced in the cache, the WHOLE BLOCK is copied back to MM to make sure all the MM locations are up-to-date.