GRANT WILSON

WLSGRA012-Architechtures II

Cache Simulation Report

This report contains my finding when building a Direct Map Cache simulation

CSC2002S

Index

Technical Overview

Simulation classes

The Cache Simulator uses a total of 4 Classes.

- Utils – Converts the hexadecimal value given by the CPU into a decimal number.
- CPU – Reads the data from an instructions file and uses it to look up values in the caches. It also calculate the cost of fetching these instructions.
- Cache – Data Structure that stores memory addresses in a 2D Integer array.
- GUI – User interface to interact with the program instead of using parameters when calling main.

How does the simulation work?

The Simulation starts off with the GUI launching. The GUI enables the user to enter the test which they desire. The data will be encoded to create the CPU and Cache objects. The CPU will take in 3 parameters and then the simulation will begin. The 3 parameter in the CPU are, a Cache object and a File to read the addresses to be searched. The third parameter is an optional L-2 Cache. The Cache objects will take two parameters. These parameters are, the number of blocks in the cache and the size of a block in cache.

Once the required objects are created, the CPU will create an array of each address value in the file. It then checks if each value is in the L-1 Cache. If the address is not in the L-1 Cache then the CPU will look in the L-2 Cache. This leads to 3 outcomes in a search.

1. The item is in the L-1 Cache – This will increment the L-1Hit variable and the CPU will look for the next item in the data list.
2. The item is in the L-2 Cache and not in the L-1 Cache – This will increment the L-2Hit Variable and the CPU will look for the next item in the data list.
3. The item is in neither Cache – This will increment the miss Variable and the CPU will look for the next item in the data list.

Once all items in the data list has been searched for, the CPU will generate the CPI. The GUI will then generate a report on the test and allow the user to run another test using the same or different cache configuration.

I chose not to pass in Parameters to the actual program as the GUI is a better solution for inputting data.

Results and Analysis

The following spreadsheet represents the comparisons of the cache performances against each other.

Cache Simulation: Simulation results are done using the HeapSort instruction

Setup 1	Setup 2	CPI ratio	Percentage Speed Up
1	2	1.35	26% slower
1	3	2.35	57% slower
1	4	2.35	57% slower
2	1	0.74	35% faster
2	3	1.74	43% slower
2	4	1.74	43% slower
3	1	0.43	135% faster
3	2	0.57	75% faster
3	4	1.00	75% faster
4	1	0.43	135% faster
4	2	0.57	75% faster
4	3	1.00	75% faster

Setup		
ID	Setup	CPI
1	Single: 16 blocks and 16 bytes / block	127
2	Single: 16 blocks and 32 bytes / block	94
3	Double: L1: 16 blocks and 16 bytes / block; L2: 64 blocks, 64 bytes / block	54
4	Double: L1: 8 blocks and 32 bytes / block; L2: 64 blocks, 64 bytes / block	54

The effective CPI for a Setup that does not have a cache is 1000.

From the above results the CPI Ratio shows how much slower one cache setup performs to another. For instance, cache setup 1 performs 1.35 times slower than setup 2. This calculates to a Percentage Speed up of 26% slower than setup 2. Conversely setup 2 performs 0.74 times slower than setup 1. This calculates to setup 2 being 35% faster than setup 1. We can also see that setup 3 and 4 perform very strongly to setup 1 and 2. This is due to the double cache system that is in place.

Evaluation

Which configuration generally performs best?

In the Simulation, setup 3 and 4 performed the best. This is due to the large L2 cache that prevented a complete miss (i.e. a memory fetch). How ever, I found it interesting that setup 3 and setup 4 performed the same even though setup 3 had a larger L1cache but smaller block sizes. I then realized that this was due to the cache having the same capacity to store the same amount of words shown below.

Setup 3: 16Blocks, 16Byte/Block =16(16) = 256Bytes*

Setup 4: 8Blocks, 32Bytes = 8(32) = 256Bytes*

Therefore, the CPU can access the same amount of items in both cases. This will result in the two caches performing the same.

Given that L1 cache is much more expensive than L2 cache, what are the tradeoffs between them?

L1Cache is much faster than L2Cache and obviously both are faster than Main Memory. The main reason for this is that L1 and L2 cache use Static RAM and Main Memory uses Dynamic RAM. But this does not answer why L1 Cache is more expensive than L2 Cache. The answer is that L1 and L2 cache are optimized for speed over capacity.

This means we are paying for speed. If we were to convert the purchase per unit of RAM, L1 and L2 RAM would be orders of magnitudes more expensive. But why is L1 Cache more expensive than L2 Cache? L1 Cache is designed primarily for speed. L2 Cache on the other hand is designed primarily for catching L1 Cache misses.

L1 latency time is calculated by the time it takes for a cache hit. We want this value to be as low as possible. L2 latency time is calculated by the L1 miss time + the L2 cache hit time. We are quite dependent on L1 cache performance. L2 cache attempts to catch as many L1 cache misses as possible. To do this L2 cache sacrifices speed for space. This is why L2 caches are larger is space. Another factor is that L1 cache is manufactured so that a search happens in parallel for optimal speed. We want to maximize the hit latency as this operation is most common.

Thus the trade-offs are as follow:

- Latency time vs a high hit rate.
- L2 is able to facilitate more cache hits, thus its hit time is much longer than L1 hit time.
- L1 is able to retrieve requests much faster than L2 Cache if a hit were to occur. However, it would not be able to have a hit as often as L2 Cache due to its lacking space.

For a fixed size L1 cache, do you recommend increasing block size or the number of blocks? Does this change for different configurations?

This option depends on what you want the processor to do. In a general sense, I would increase the block size provided that the latency times remain unchanged. This would increase my hit rate resulting in a better CPI.

On the other hand, these small blocks can only store a limited size of some value X(X=4) words (16Bytes). If we were to search for an address containing a value Y with containing 32Bytes, then we would have to search for a value split over 2 blocks. This will result in time wasted in combining the 2 16Byte values. There are strong trade-offs between the scenarios mentioned above. One has to think of the processes that will be calculated.

There is a strange case where 2 Caches have the same capacity but their block numbers and size vary.

Setup 3: 16Blocks, 16Byte/Block =16(16) = 256Bytes*

Setup 4: 8Blocks, 32Bytes = 8(32) = 256Bytes*

What would be the optimal solution if there were no restrictions on speed and space trade-offs?

The ultimate super computer would have infinite blocks of L1 cache. This would cause the computer to never miss any request other than the initial miss. The computer would have data always read at high speeds.

We would not need to even have other caches to facilitate cache misses as the cache would never miss. However, computers are limited by physical laws and are unable to implement this. So designers manufacture computers to fake this ideal goal.

Why do we not have many layers of cache? I.e. L1, L2, L3, L4...Ln?

We know that L1 Cache is the fastest. We also know that L1 Cache is the smallest. L2 cache offers trade-offs sacrificing speed for capacity. As L3 cache comes into the picture, we can see that L3 cache sacrifices even more speed for capacity. As the layers of cache increase, we are constantly trying to increase cache size to prevent the event of a cache miss. However, in doing so, we are also slowing down the time it takes for a search to be done.

If we keep adding caches to an arbitrary size of let's say L20; then we have sacrificed a lot of speed for capacity. We will then be tending towards the last Cache (being the cache that never accommodates a miss) being equivalent to the hard drive. Also it is worth noting that-

Ln Cache latency time = L1 latency + L2 latency + … +Ln-1 latency

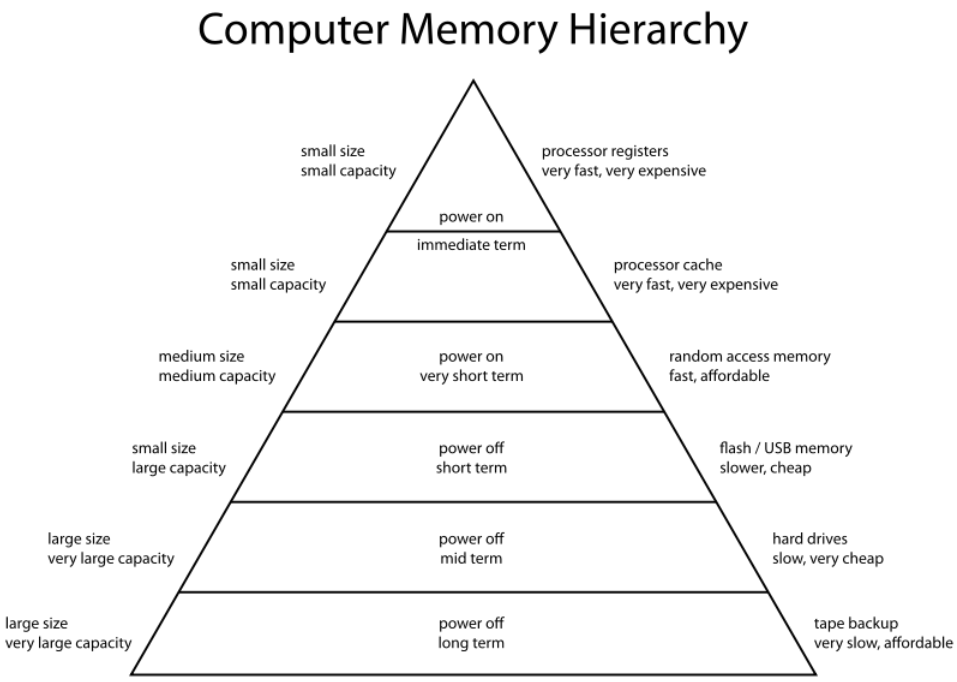
Thus it is worth noting that Ln cache latency >> L1 cache latency. Thus the cache latencies are increasing greatly.

The Strongest point is that Cache memory is static memory. Main memory is Dynamic memory and Hard disk is magnetic memory. We must remember that static memory is very expensive per converted value and magnetic disk memory is very cheap per converted value. Thus we maintain the current cache systems that balance speed, capacity and cost.

L-caches and main memory are also volatile. They lose their content when power is lost. We want to store our data permanently!

Fun fact: The highest level of cache is L4. Intel owns a 128MB cache which they call broad well.

Below shows the speeds of various speeds of memory layers.



Future Work

How would I extend my simulation to facilitate an n-way Associative cache?

To extend my program I would make a new cache class called AssociativeCache. This class would extend the Cache class that I currently have. I would have to override the *hasValue()* and *setValue()* methods to cater for an n-way associative cache object address search. The methods would contain calculations to find out which set to search for a particular value. I would keep using a 2D Integer array for the addresses that need to be searched. I would use Integer object and not the primitive because I cannot check if a cache block is empty using ints.

E.g. Lets say we want to find the value "0". All uninitialized blocks will return zero as the cache is a 2D in array. I use Integer object so that I can check if a cell is null. If the cell is null then I can increment a miss and so on.

I would also extend my CPU to support L3 and possibly L4 caches.

I would love to implement a Main memory. If there was main memory, I could return data to the CPU. I would create block objects and populate the caches with these objects. The block would contain a tag, index, offset and valid bit. This would simulate a real cache more accurately.

If I wanted to go completely out of the scope of the project, I would also make my search methods completely parallel like a real L1 cache.

I would give my GUI a face lift to accommodate custom cache sizes that the user wants to specify.

E.g. L1: 16 Blocks, 64Bytes L2: 512Blocks, 256Bytes etc.

I would also create a functionality that would compare 2 CPI's of cache setups and save them to a file for easier evaluation and analysis.