Youssef Hawas 900150248

Omar El-Seadawy 900150105

CSCE 2303-02

Computer Organisation and Assembly Language

Spring 2017

Project II

Cache Simulator

17/05/2017

**Implementation**

The code written implements a generic cache using a 2-D array of dimensions number of blocks and ways. The code uses #defines in order to define some characteristics of the cache, like the blocksize, cachesize, and ways. For a direct mapped cache the ways are defined as one. For a fully associative cache, the ways are defined so as to have the number of blocks to be one. For any other values, it is a set associative cache. The main, calls the function CacheSim, sending it an address generated by one of the Memory Generation functions, and it returns HIT or MISS.

The CacheSim function depends on the type of cache you're simulating. First it calculates the size of the tag, index and offset. If it is a fully associative cache (n ways) , the function loops over the entire cache array, if it finds the tag of the address then it returns a HIT and updates the entry's fields, like the time, frequency and so on, else it tries to find an empty place in the cache by checking the validity bit of every element in the cache array, if it does, return MISS and replace the data and initialize its fields. If it fails to find an empty place it switches over the replacement define in order to determine what policy you're simulating with. The policy obtains the index of the cache entry you're going to replace and returns a MISS.

If it is running as a direct mapped cache (1 way). The program accesses the cache array at the row that corresponds to the index portion of the address. If the tag matches, then the function returns the HIT and updates the entry's fields. If not, the program checks if the cache has been validated at that index. If it hasn't, the data is inserted into that index and the value of its fields are initialized. If it has been validated, the data at that index is replaced by the current data, and its fields are initialized.

The Set Associative cache works in a similar way to the Direct Mapped Cache. The program checks all the ways at the index. If the tag matches, the function returns a HIT and updates the cache entry's fields. If not, it checks if any of the ways at the index are empty. If they are, it inserts the data and initializes its fields and returns a MISS. If not, it removes one of the entries randomly and replaces it with the current data and initializes its fields and returns a MISS.

The main receives either HIT or MISS and updates the hit and miss counts accordingly. It then ends by showing the output which is the miss rate.

**Data**

**Direct Mapped (Cache Size 64KB) (Varying Block Size)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **4** | **8** | **16** | **32** | **64** | **128** |
| **MemGen1** | 0.75 | 0.875 | 0.9375 | 0.96875 | 0.984375 | 0.992187 |
| **MemGen2** | 0.49134 | 0.495452 | 0.49726 | 0.498911 | 0.499732 | 0.499663 |
| **MemGen3** | 0.00965 | 0.000954 | 0.000997 | 0.001059 | 0.00101 | 0.000994 |
| **MemGen4** | 0.999744 | 0.999872 | 0.999936 | 0.999968 | 0.999984 | 0.999992 |
| **MemGen5** | 0.983616 | 0.991808 | 0.995904 | 0.997952 | 0.998976 | 0.999488 |
| **MemGen6** | 0 | 0 | 0 | 0 | 0 | 0 |

**Direct Mapped (Block Size 16) (Varying Cache Size)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **4** | **8** | **16** | **32** |
| **MemGen1** | 0.9375 | 0.9375 | 0.9375 | 0.9375 | 0.9375 | 0.9375 |
| **MemGen2** | 0.007825 | 0.015411 | 0.031077 | 0.062161 | 0.124373 | 0.249003 |
| **MemGen3** | 1.9E-05 | 4.6E-05 | 6.9E-05 | 0.000148 | 0.000277 | 0.000529 |
| **MemGen4** | 0.999936 | 0.999936 | 0.999936 | 0.999936 | 0.999936 | 0.999936 |
| **MemGen5** | 0.9375 | 0.9375 | 0.9375 | 0.9375 | 0.9375 | 0.9375 |
| **MemGen6** | 0 | 0 | 0 | 0 | 0 | 0 |

**Fully Associative(Block Size 16)(Random Replacement)(Varying Cache Size)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **4** | **8** | **16** |
| **MemGen1** | 0.96875 | 0.96875 | 0.96875 | 0.96875 | 0.96875 |
| **MemGen2** | 0.007741 | 0.015665 | 0.031276 | 0.062535 | 0.124813 |
| **MemGen3** | 2.1E-05 | 3.8E-05 | 8.4E-05 | 0.000144 | 0.000257 |
| **MemGen4** | 0.999968 | 0.999968 | 0.999968 | 0.999968 | 0.999968 |
| **MemGen5** | 0.969211 | 0.969692 | 0.97065 | 0.972568 | 0.976403 |
| **MemGen6** | 9.2E-05 | 0.000188 | 0.00038 | 0.000762 | 0.00153 |

**Fully Associative (Cache Size 4KB)(Block Size 32)(Varying Replacement policies)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Random** | **LRU** | **LFU** | **FIFO** |
| **MemGen1** | 0.96875 | 0.96875 | 0.96875 | 0.96875 |
| **MemGen2** | 0.0312 | 0.031252 | 0.031246 | 0.031252 |
| **MemGen3** | 8.3E-05 | 8.7E-05 | 8.6E-05 | 8.7E-05 |
| **MemGen4** | 0.999968 | 0.999968 | 0.999968 | 0.999968 |
| **MemGen5** | 0.970652 | 0.970655 | 0.970655 | 0.970655 |
| **MemGen6** | 0.00038 | 0.000381 | 0.000381 | 0.000381 |

**Set Associative(Cache Size 64KB)(Block Size 32)(Varying Ways)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **2** | **4** | **8** | **16** |
| **MemGen1** | 0.96875 | 0.96875 | 0.96875 | 0.96875 |
| **MemGen2** | 0.499366 | 0.498308 | 0.499093 | 0.499412 |
| **MemGen3** | 0.000998 | 0.00096 | 0.000991 | 0.000996 |
| **MemGen4** | 0.999968 | 0.999968 | 0.999968 | 0.999968 |
| **MemGen5** | 0.997952 | 0.997952 | 0.997952 | 0.997952 |
| **MemGen6** | 0.000256 | 0.000576 | 0.00064 | 0.00072 |

**Analysis**

**1. Sequential Memory Generation**

**MemGen1:**

* **Direct Mapped**
  + MemGen1 generates addresses sequentially from 0x0 till the number of iterations we are performing the simulation on. In the case of a Direct Mapped Cache, the hits will always depend on the block size due to the sequential nature of the function. As the block size increases, the misses will decrease, which explains the positive correlation between both the block size and hit rate. In experiment 2, the block size is constant, and therefore so is the hit rate.
* **Fully Associative**
  + Due to the fully associative cache's nature, there is no index for the addresses generated. This means that the misses occur only when the tag changes, which happens only after every block is exceeded, which is fixed as 32 for the first experiment for the fully associative cache. So after every 32 generations from the first one, there occurs a miss, and this is constant throughout the experiment. The same applies for the 2nd experiment where we change the replacement policy, the sequential and increasing nature of the MemGen causes it to not be affected by the change in replacement policy.
* **Set Associative**
  + As mentioned before, MemGen1 has a positive correlation with block size, and there's only one Direct Mapped experiment that changes the block size. For the rest of the experiments, the block size is always constant, and therefore so is the hit rate.

**MemGen4:**

* **Direct Mapped/Fully Associative/Set Associative**
  + MemGen4 also generates addresses sequentially in steps of 1 up to 3ff. This causes it to have a very high hit rate in all cases, due to the fact that the only misses that occur are compulsory ones, when blocks are being accessed for the first time. After the compulsory misses of the first loop from 1 to 3ff, the rest of the accesses are all hits. A larger block size also helps avoid some of these compulsory misses therefore, the hit ratio rises gradually as the block size increases.
  + Due to the fact that MemGen4 only causes very few compulsory misses and no other misses, it stays almost constant despite changing the cache type. It is only affected very gradually by the change of block size.

**MemGen5:**

* **Direct Mapped**
  + MemGen5 also generates sequentially from 1 to 0xffff. Due to its sequential nature and spatial locality of sequential memory addressing, it too has a high hit rate, because after the first miss, a lot of hits occur due to the addresses after having the same index and tag but are within the same block(only differ in offset). This is shown by the fact that when we increase the block size, the hit rate gradually increases.
* **Fully Associative**
  + MemGen5's hit rate in a fully associative cache is a little less than in a direct mapped cache, due to the fact that there is no index, so the tag is more susceptible to change in the fully associative cache, and therefore, there are more misses in the fully associative cache.
* **Set Associative**

Like it's behavior in the Direct Mapped cache, it maintains a high hit rate in the Set Associative cache as well.

**MemGen6:**

* **Direct Mapped**
  + MemGen6 also generates addresses sequentially but in steps of 0x100. Therefore, in the case of a Direct Mapped Cache, every entry will miss due to the fact that every address entered will have a difference that is greater than the block size, and therefore will have a different index and miss.
* **Fully Associative**
  + MemGen6's hit rate increases as the Cache Size increases in a fully associative cache. This is due to the fact that the Cache Size is directly proportional to the number of ways in a fully associative cache with a constant block size. An increase in the number of ways causes fewer misses.
* **Set Associative**
  + MemGen6 has a higher hit rate as the number of ways increases, as it helps avoid some of the misses caused by the sequential generation. This is similar to MemGen6's behavior in a Fully Associative cache.

**2. Random Memory Generation**

**MemGen2:**

* **Direct Mapped**
  + MemGen2 generates random addresses that of a 'memory' of size 128KB, therefore it has a very large pool of potential addresses to generate. This means that the larger the cache is, the more chance there is to have a higher hit rate, as the addresses generated will be within the range of the cache. We can see this when the cache size is fixed at 64KB, the hit rate in that case is around 0.5. The hit rate is much less when we fix the block size and use caches with less size, as seen in the 2nd direct mapped experiment.
* **Fully Associative/Set Associative**
  + Due to the functions random nature, the only thing we can definitively correlate it with is the cache size. All the experiments with cache size show an increase in the hit rate along with the increase in cache size.

**MemGen3:**

* **Direct Mapped**
  + MemGen3 generates random addresses that are supposed to be within the range of the predefined DRAM of size 64 MB. Like previously stated in MemGen2, this allows for a massive pool of possible memory addresses, as the DRAM is much larger than the cache. Like stated in MemGen2, the hit rate also shows a positive correlation with the cache size.
* **Fully Associative/Set Associative**
  + Much like MemGen2, we can only conclude the positive correlation of MemGen3's hit rate with the cache size. However, as previously stated, MemGen3 has a much smaller hit ratio due to the range of addresses it generates.

**3. Replacement Policies**

In a fully associative cache we can use many different removal policies, LRU, LFU, FIFO, and Random removal. During the experiment in the fully associative cache, we change the replacement policy while maintaining the cache size and block size at a constant. Due to the LRU and the FIFO being similar in implementation, their hit rates are basically identical. However, the Random removal and LFU are slightly different. Both of them have a hit rate that is slightly less than that of LRU and FIFO. However, this is not representative of the random removal policy due to the fact that it is, in the end, random. For one simulation, it can show a higher hit rate, while it can show a lower one for another simulation. Another thing to keep note of, is that the function used to implement the random removal policy is not truly random, but a weak pseudorandom function.

**4. Comparisons**

**Direct Mapped**

* Best Hit Ratio : MemGen4
* MemGen4 has the best rate for the direct mapped because of its sequential generation and that it avoids any misses that aren't compulsory.
* Hit Rate is directly correlated to block size.

**Fully Associative**

* Best Hit Ratio: MemGen4
* As mentioned in the Direct Mapped comparisons, MemGen4 avoids misses that aren't compulsory.
* Hit Rate is directly correlated to Cache size, inversely correlated to block size. This is because of each's effect on the number of ways of the cache.

**Set Associative**

* Best Hit Ratio: MemGen4
* It is the highest as it only suffers from compulsory misses.
* Hit Rate is directly correlated to the number of ways.

**Note:** While MemGen4 has the best hit ratio, it depends very heavily on the fact that all generated addresses are very close in value. It is a good representation of spatial locality. However, it is not a very generic case of memory accessing.