CACHE SIMULATOR

CS 5513 Computer Architecture - Group Project

Utkarsh Singh (udg538)
Janani Ramesh (vgw957)
Jacinto Molina (xtd781)

Background of the thesis provided:

The aim of this project is to put together a CPU cache simulator and to understand the efficiencies discussed in the class. Caches are the fastest memory banks but are very small in capacity. Caches are the first place where any kind of information needed for a currently executing instruction are fetched from. If the required information is not found, we say it is a cache miss and then we move upwards, towards the slower form of memory banks to fetch the required data. Now, since the capacity is limited in a cache, we have to carefully examine it to decide if we bring in a new piece of data, where does it fit. That involves kicking some information out of the cache using a replacement algorithm and then writing the new information in place of the information kicked out. For the sake of this project we work with the least recently used policy applied in current times.

Implementation:

Now that we have established that caches are very small in capacity to hold data let us describe where the cache sits. The cache is a bridge between the RAM and the CPU of a machine. Our aim here is to reduce the number of RAM hits to get data when a process runs. We try to depict the LRU (Least Recently Used) policy in this project.

What is the LRU policy?

In the LRU policy, the algorithm searches for the oldest piece of data in the cache and evicts that from the cache. This victim is chosen based upon the time comparisons, done to see which piece of data was used the longest time ago –written, updated or read. If the chosen victim is needed again, it is written back to the cache using the same algorithm which then chooses another victim.

The eviction process mandates that the memory engine to lock the LRU chain during the process.

Write Back Policy:

As the name suggests, the write back policy dictates the data written in the cache. Write back is a storage method in which data is written into the cache every time a change occurs but is written into the corresponding location in main memory only at specified intervals or under certain conditions. When a data location is updated in write back mode, the data in cache is called fresh, and the corresponding data in main memory, which no longer matches the data in cache, is called stale. If a request for stale data in main memory arrives from another application program, the cache controller updates the data in main memory before the application accesses it. Write back optimizes the system speed because it takes less time to write data into cache alone, as compared with writing the same data into both cache and main memory. However, this speed comes with the risk of data loss in case of a crash or other adverse event. Write back is the preferred method of data storage in applications where occasional data loss events can be tolerated. In more critical applications such as banking and medical device control, an alternative method called write through practically eliminates the risk of data loss because every update gets written into both the main memory and the cache. In write through mode, the main memory data always stays fresh.

Pros: This provides high throughout and low latency for “write” oriented applications.

Cons: The data availability risk is always there because the cache could fail and have data loss before the data is persisted. This would result in the data being lost.

Implementation steps:

A pearl script (cachesim.sh) accepts a filename along with the cache size, cache line size and the number of ways from the user on the terminal. We have used the pearl script to make sure that the code is run using python3 environment. The pearl script calls the python script with the parameters passed to the pearl script.

Process Implementation:

1. Checks the required parameters passed. If the parameters are invalid, shoots out an error message.
2. Creates a cache with the size, cache line size and number of ways provided.
3. Scans the input file one line at a time. If the input is valid then add it to the list else the input is ignored. Then we go on to calculate the memory tag from the address from the valid read.
4. Let us start with the read process. The program scans the input file one line at a time. If the input is valid then add it to the list else the input is ignored. Then we go on to calculate the memory tag from the address from the valid read. Now for a read operation. Then calculate the rows associated with the slot number If it is a hit then, update the access timestamp for that entry accordingly else it is a cache read miss and needs proper updates. When this happens, we look for empty slots. If we find an empty slot, we store the record in that slot else we employ the LRU policy to choose a victim and replace that victim with the needed piece of data.
5. Now let us look at the write part. We employ the write back policy to calculate which records are paired with a slot number. Then we check if this is a cache hit. In this case we update the access timestamp for that record and mark it as recently accessed. If it is not the case, then this would be declared a cache write miss and this would need an update. We again look for empty slots. If we find an empty slot, we place the record in that slot. If a slot is not found, then we choose a victim using the LRU policy. We then go on repeating the same step until the end of file is reached. Then we print the result of cache – miss rate in proper format.

Results:

The provided files were downloaded from SharePoint. After the extraction the files were tested against the simulator by passing them as an argument to the bash script created.

As part of the experiment different cache sizes were used and a subset of those choices are depicted below and the provided files are on the next page:

The cache miss rates can be seen in the table below:

File name	Cache size	Result
1KB_64B	1024	50.00%
4MB_4B	4194304	2.08%
32MB_4B	4194304	6.25%
naive_dgemm.trace.txt	262144	50.25%
ls.trace.txt	32768	2.17%
bw_mem.traces.txt	4194304	1.56%
gcc.trace.txt	32768	1.89%
naive_dgemm_full.trace.txt	262144	49.37%
openblas_dgemm.trace.txt	262144	8.30%
openblas_dgemm_full.trace.txt	262144	7.50%

Provided files:

1KB_64B	bw_mem.traces.txt
4MB_4B	gcc.trace.txt
32MB_4B	naive_dgemm_full.trace.txt
naive_dgemm.trace.txt	openblas_dgemm.trace.txt
ls.trace.txt	openblas_dgemm_full.trace.txt

You can see different run times each dataset provided took to load in the cache along with miss rates.

```
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh 1KB_64B
Cache miss rate: 50.00%

real    0m0.037s
user    0m0.020s
sys     0m0.012s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh 4MB_4B
Cache miss rate: 2.08%

real    0m14.428s
user    0m14.338s
sys     0m0.055s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh 32MB_4B
Cache miss rate: 6.25%

real    1m56.895s
user    1m56.390s
sys     0m0.326s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh naive_dgemm.trace.txt
Cache miss rate: 50.25%

real    0m7.436s
user    0m7.311s
sys     0m0.051s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh ls.trace.txt
Cache miss rate: 2.17%

real    0m1.398s
user    0m1.339s
sys     0m0.027s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh bw_mem.traces.txt
Cache miss rate: 1.56%

real    0m4.496s
user    0m4.399s
sys     0m0.052s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh gcc.trace.txt
Cache miss rate: 1.89%

real    0m1.972s
user    0m1.911s
sys     0m0.028s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh naive_dgemm_full.trace.txt
Cache miss rate: 49.37%

real    4m4.418s
user    4m3.367s
sys     0m0.630s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh openblas_dgemm.trace.txt
Cache miss rate: 8.30%

real    0m4.569s
user    0m4.480s
sys     0m0.030s
[Jananis-MacBook-Pro:cache simulator jananiramesh$ time sh run_sim.sh openblas_dgemm_full.trace.txt
Cache miss rate: 7.50%

real    1m55.283s
user    1m54.691s
sys     0m0.337s
```

Fig 1: Run time with different files and cache sizes.

Individual Contribution:

All of us had effectively contributed to the project. We had meetings on a regular interval to discuss, divide and check on the progress of the tasks that we assigned to ourselves. Even though we had divided the tasks amongst ourselves, we met on and off, discussing and helping each other during the project and collaborated on the report. Our divided work can be found below:

I: Policy implementation - Utkarsh Singh

II: Downloading, understanding and interpreting the provided files – Jacinto Molina

III: Testing and code optimization- Janani Ramesh

This project has been a great learning experience for all of us.