**Lab 4: Cache Simulation**

A) Simulation Results

I elected to show only the average execution time for the following simulations because it is the best indication of performance. Cache miss rates would matter if the miss penalty did not change as the configuration changed. A lower miss rate paired with a higher miss penalty can result in a slower execution.

Line Size (in Bytes)

Above are the results of running the five traces on similar cache configurations, where only the line size and miss penalty have changed. Since we are looking for the configuration that results in the fastest cache, the lowest execution time is most desirable. Here, the 16-Byte cache line has the best performance of the possible configurations.

Cache Size (in Kilobytes)

In this set, only the cache size and miss penalty have changed, starting with the best configuration from the last test. Again, we find that the original configuration outperforms the other configurations, having the lowest execution time.

Associativity

Finally, we test different associativity configurations (and their incurred miss penalties). Surprisingly, the original cache configuration *again* outperforms the other possible configurations. This leaves us with a cache that is 16KB large, using 16B blocks that are directly mapped.

B) Cache Miss Rate

Above I have supplied a bar graph showing the miss rates of the three cache line configurations we tested. We found that the 16-Byte cache line provided the quickest executions, but we find that it also has the largest miss rate. For this reason, I do not believe that the miss rate provides any indication of performance. I believe the reason the 16-Byte cache line is able to outperform the other configurations is due to the fact that it has a lower miss penalty and the fact that store misses are stored in shorter time.

C) Performance Across Programs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Line size | gcc | gzip | mcf | swim | twolf |
| 16 | 1827252 | 7461815 | 1029309 | 1299114 | 1546284 |
| 32 | 1802278 | 7779490 | 1026585 | 1285125 | 1545118 |
| 64 | 1836672 | 8416460 | 1025003 | 1303328 | 1591677 |

Above is a chart providing the execution time of each program for different line sizes, in nanoseconds. From just this test, we find that the results are not uniform. For gcc, the 32-byte configuration runs quickest. For gzip, the 16-byte configuration runs quickest. For mcf, the 64-byte configuration was quickest. With just the first three programs we find that not one configuration runs consistently quicker than the others. In fact, the 32-byte cache runs quickest most often, but we already know that the 16-byte cache runs quickest overall. The reason behind this is likely the memory accesses in each program. For example, mcf has a 49.51% miss rate with the 16-byte cache, whereas the 64-byte caches returns only a 12.40% miss rate.

D) Speedup

The default design ended up being the quickest configuration out of the seven we tested. Since the final design and the default design are the same, the speedup is 0%.

E) Data Structure/Algorithm

I created a 3D array to hold tags, stalls, and usage information. The first index chose the set, the second chose the block, and the third would choose which information to output. I had a supplemental 2D array, which would hold information for the LRU system. The first index chose the set, and the second chose the block, just as the 3D array had. Here I would save how many memory accesses have passed since it was last called. To keep it efficient, I would only update the set that was being used by the current memory call.

To keep track of write stalls, I created a 1D array with as many index values as there were blocks. This way, even if every block had a store miss, we would not run out of room. To supplement the array, I saved an integer with the number of addresses stored in the array. In use, the array could have index values larger than this number that were non-zero, and we do not want to access these. Therefore instead of using the size of the array in loops, I would use the integer as the maximum bound.

These objects, along with some configuration/access objects make up the virtual cache. Originally, this was all in a separate class, however I moved it into the simulator because I thought I needed some information that was stored in the simulator. Once I had finished the simulator, I realized this was a mistake, but did not believe it necessary to change back.

For stores, I would check the “cache” for that address block, which would update the LRU array if the block were found. If it were found, I would let the program run the next trace line. If it were not found, I would update the “cache”: placing the tag in the least recently used block, updating the write stall value, and adjusting the LRU array.

For loads, I would again check the “cache”, again updating LRU data if the block were found. If it were found, I would check the block’s stall value and compare that to the number of instructions that have passed since the last memory access. If it were smaller, I would do nothing and continue on. However, if the stall were larger, I would add the difference to the number of cycles run. If the block were not found, I would add the tag to the array, update the stall and LRU information.

F) Sample Output

java CacheSimulator -s 4 -a 2 -l 16 -mp 40 -wl

-s : 4

-a : 2

-l : 16

-mp : 40

Set Size: 32 total sets: 128

tag: 21 index: 7 offset: 4

Simulating 'Short\_list.txt' ...

l 0x00000000 3

Load Miss

s 0x00000170 7

Store Miss

l 0x000000D3 2

Load Miss

l 0x00000148 4

Load Miss

s 0x0000043A 6

Store Miss

l 0x000000D2 0

Load Hit

l 0x000000AA 0

Load Miss

s 0x00000428 6

Store Miss

l 0x000000B5 3

Load Miss

l 0x0000043A 2

Load Hit

s 0x0000014A 5

Store Hit

s 0x000000C7 0

Store Miss

s 0x0000001D 0

Store Miss

s 0x00000167 0

Store Miss

done:

Cache Architecture:

-----------------------------------------------------

Cache size (KB) .... 4

Associativity .... 2

Line size (Bytes) .... 16

Number of sets .... 128

Number of blocks .... 256

Write policy .... Write-Allocate

Miss penalty (cycles) .... 40

Cycle time (ns) .... 1

Simulation Results for 'Short\_list.txt':

-----------------------------------------------------

Total Loads & Stores ... 14

Loads ... 7

Stores ... 7

Load Misses ... 5

Store Misses ... 6

Total Miss Rate ... 0.785714

Load Miss Rate ... 0.714286

Store Miss Rate ... 0.857143

Instruction Count ... 52

Cycle Count ... 252

Stall Cycles ... 200

Average Memory Access Time ... 14.285714

CPI ... 4.846154

Execution Time ... 252.000000

You will find the source code in the zip file that contained this document.