**Project 1 Report:**

**Cache Simulator**

By: Joshua Nichols

Computer Architecture

EEL 4768-003

11/08/2019

**Objective**

* To see the impact of changing cache size, associativity, write policy, and replacement policy on the performance of the cache simulator that we made.

**Process**

* Run each experiment using the setup mentioned for each.
* Due to not being able to run this in Eustis, the following method was used to test the program in command prompt:
  + gcc -o SIM SIM.c
  + SIM.exe [ S ] [ A ] [ R ] [ W ] "C:\Users\Josh\Desktop\[ T ]"
  + Where:
    - S is the size of the cache in bytes.
    - A is the associativity of the cache.
    - R is the replacement policy (0 is Least Recently Used (LRU) and 1 is First in First Out (FIFO))
    - W is the write policy.
    - T is the trace file name.

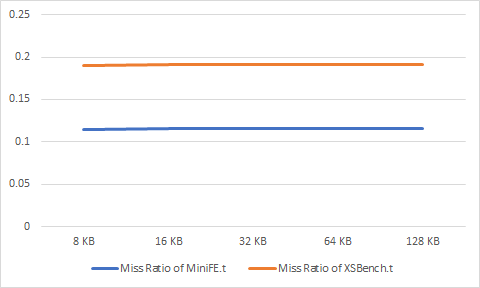
**Cache Size Analysis**

The following settings were maintained for the cache as instructed:

* 4 way associative cache
* Least Recently Used (LRU) Replacement Policy
* Write Back Policy

|  |  |  |
| --- | --- | --- |
| **Cache Size** | **Miss Ratio of MiniFE.t** | **Miss Ratio of XSBench.t** |
| 8 KB | 0.115507 | 0.19144 |
| 16 KB | 0.115511 | 0.191441 |
| 32 KB | 0.115521 | 0.191443 |
| 64 KB | 0.115538 | 0.191447 |
| 128 KB | 0.115572 | 0.191454 |

Table 1: *Miss Ratio vs Cache Size*



Graph 1: *Graph of Miss Ratio vs Cache Size for both trace files.*

Larger caches can hold more entries and therefore should have fewer misses. My code unfortunately does not well reflect this. Additionally, MiniFE missed a lot less than XSBench did.

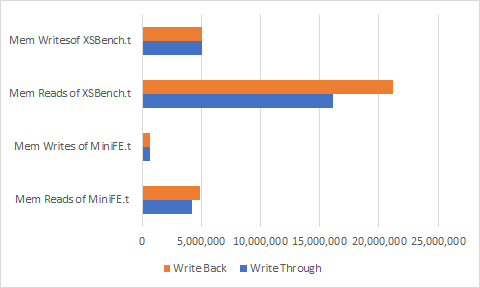
**Write Policy Analysis**

The following settings were maintained for the cache as instructed:

* 4 way associative cache
* Cache size of 32KB
* LRU Replacement Policy

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Write Policy** | **Mem Reads of MiniFE.t** | **Mem Writes of MiniFE.t** | **Mem Reads of XSBench.t** | **Mem Writesof XSBench.t** |
| Write Through | 4,237,570 | 636,483 | 16,161,466 | 5,013,495 |
| Write Back | 4,874,053 | 636,593 | 21,174,961 | 5,013,621 |

Table 2: *Write Policy vs Memory Reads and Memory Writes*



Graph 2: *Memory Reads and Writes for the trace files using different Write Policies*

How it’s supposed to work:

For write through, read miss never leads to needing to write in main memory, avoiding a big penalty, but every write needs main memory access instead. In write back, multiple writes within a block require only one write to main memory. However, reads that result in replacement may write dirty blocks to memory.

What my program does:

The trace files are impossible to go through manually, so it is likely that the Write Back may have needed to deal with more dirty blocks than it was supposed to.

MiniFE reads went up quite a bit along with XSBench reads. MiniFE and XSBench writes however stayed mostly the same.

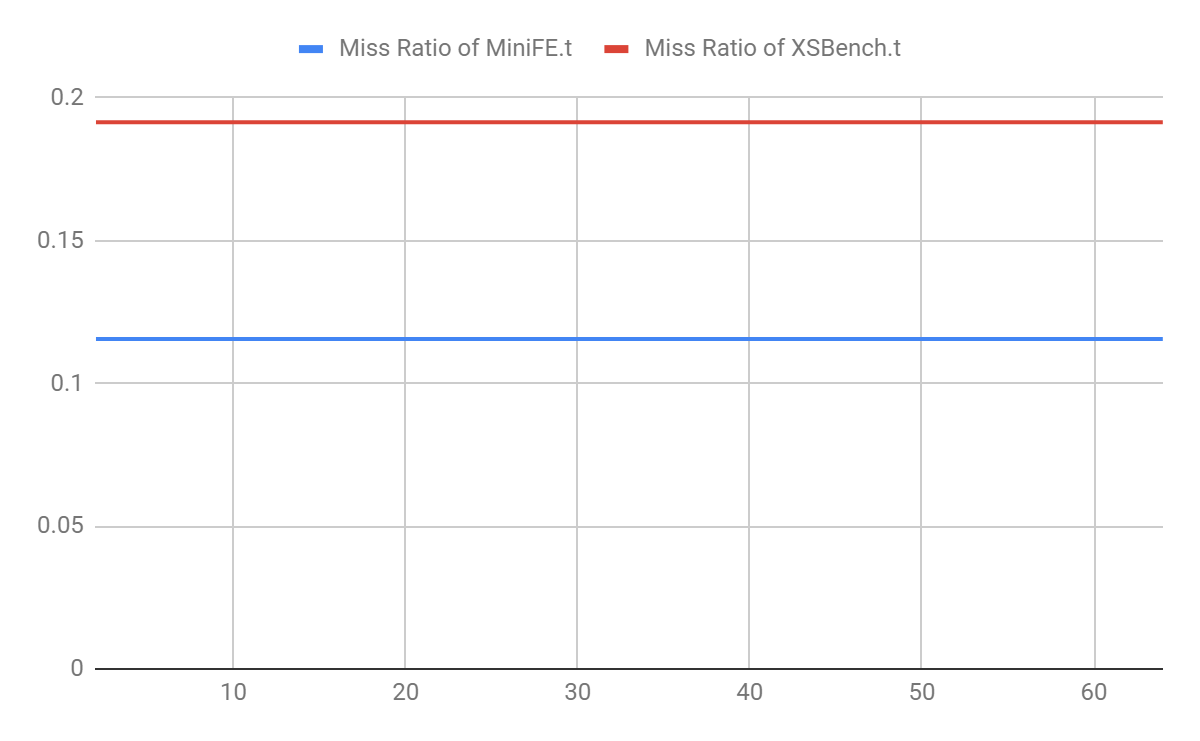
**Associativity Analysis**

The following settings were maintained for the cache as instructed:

* Cache size of 32KB
* LRU Replacement Policy
* Write Back Policy.

|  |  |  |
| --- | --- | --- |
| **Associativity** | **Miss Ratio of MiniFE.t** | **Miss Ratio of XSBench.t** |
| 1 (Direct Mapping) | 0.115573 | 0.191455 |
| 2 | 0.115538 | 0.191447 |
| 4 | 0.115521 | 0.191443 |
| 8 | 0.115511 | 0.191441 |
| 16 | 0.115507 | 0.191440 |
| 32 | 0.115505 | 0.191440 |
| 64 | 0.115504 | 0.191439 |

Table 3: *Associativity vs Miss Ratio for each trace file.*



Graph 3: *The law of diminishing returns observed in the miss ratios vs cache setup. Associativity on the y-axis. (The software I have been using for graphs has not been too helpful, so unfortunately the change is barely visible.)*

The reason for this trend is because higher associativity leads to fewer cache conflicts and lower miss rates. However, large associativity can significantly increase the lookup time, and in real life, some compromise must be made in this aspect of cache design.

Whilst my program may not properly calculate the miss ratios for each trace file, it does successfully reflect the law of diminishing returns that a real cache does. Significant gains are made improving from one-way associativity to two and four. Going past four-way begins to not make that much of a difference.

MiniFE and XSBench miss ratios decreased as a result of increasing associativity.

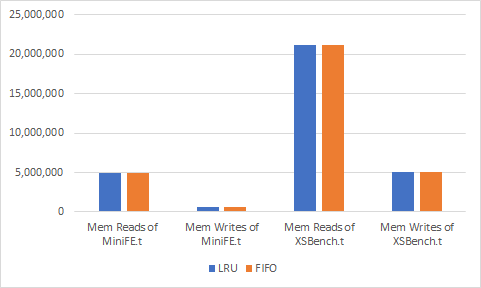
**Replacement Policy Analysis**

The following settings were maintained for the cache as instructed:

* 4 way associative cache
* Cache size of 32KB
* Write Back Policy.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Replacement Policy** | **Mem Reads of MiniFE.t** | **Mem Writes of MiniFE.t** | **Mem Reads of XSBench.t** | **Mem Writes of XSBench.t** |
| LRU | 4,874,053 | 636,593 | 21,174,961 | 5,013,621 |
| FIFO | 4,874,053 | 636,593 | 21,174,961 | 5,013,621 |

Table 4: *The effect of replacement policy on reads and writes to the memory per trace file.*



Graph 4: *Effect LRU vs FIFO replacement policies on Reads and Writes.*

What should happen:

In a policy like LRU, the maintenance of highly used entries reduces the miss rate, and therefore increases the number of successful reads and writes to the cache. In the case of FIFO, we could accidentally remove a very popular address, and pay a high penalty in order to get it back. Removing this popular address would significantly increase the miss rate of the cache, leading possibly to performance issues within the application using that cache.

Unfortunately, my program did not reflect this difference. LRU should be better, but the program does not simulate this.

Neither MiniFE reads nor XSBench reads change in response to a change of replacement policy. The same goes for writes.

**Impact of replacement policy changes with associativity:**

Based off of the results for LRU replacement policy (as shared in both experiments), the changing the associativity has a major impact on the miss ratio of the cache. However, in a real cache (or better cache simulator) changing the replacement policy to be LRU should also reduce the miss ratio. In a real cache, changes to associativity and replacement policy will have amplifying effects. For a fixed size cache, as the number of sets increase, the number of blocks in each will decrease. This decreased number of blocks will allow the LRU policy to work better, as it is easier to find the least recently used block.

**Conclusion**

From this project and the experiments performed above, we are supposed to see how data is saved and shifted within the first level of the cache hierarchy. Additionally, we should have seen the impact (in decreasing order) that associativity, replacement policies, write policies, and cache size have on the performance of a cache. Increasing the associativity will help to a point, as part of the law of diminishing returns, and the policies that seem to help the most to reduce the miss rate are the least recently used replacement policy and the write back write policy.