COL216 Spring '24
Cache Simulator: Best Cache Configuration

Aditya Prakash (2022CS11595)

Introduction

This report describes experiments conducted using our cache simulator to explore various cache configurations. By adjusting and observing the effects of different cache parameters, we aim to identify an optimal cache setup. The findings from these experiments are analyzed to determine what constitutes the *best cache configuration*.

Different Parameters of a Cache

1. **Write Policies**: The simulator can be configured for various write-policies. Upon a cache hit, one of two policies may be employed: *write-through* (write to both the memory and the cache) or a *write-back* (write only to the cache). Conversely, on a cache miss, we may use *write-allocate* (load the needed block into the cache before writing) or *no-write-allocate* (don't fetch a new block but write directly to the main memory).
2. **Block Sizes and Block Numbers**: Cache configurations vary by the number of *sets* and the number of *blocks* per set. The size of each block is typically configured in powers of two for simplicity. These parameters indirectly decide the *associativity*[1] of the cache.
3. **Eviction Policy**: These are used only for associative caches. An eviction policy decides how blocks are replaced when a set is full. We have considered two popular strategies: FIFO (*First In, First Out*), which evicts the oldest block, and LRU (*Least Recently Used*), which evicts the block that has been used least recently.

Experimentation Strategy

This section outlines the methods I used to evaluate the performance of various cache configurations. All trace files that I used, were derived from real programs. I noted performance metrics such as **miss rates**, **total clock cycle count**, and **total size of the cache (including overhead)**. Detailed results from these experiments are presented in the results section.

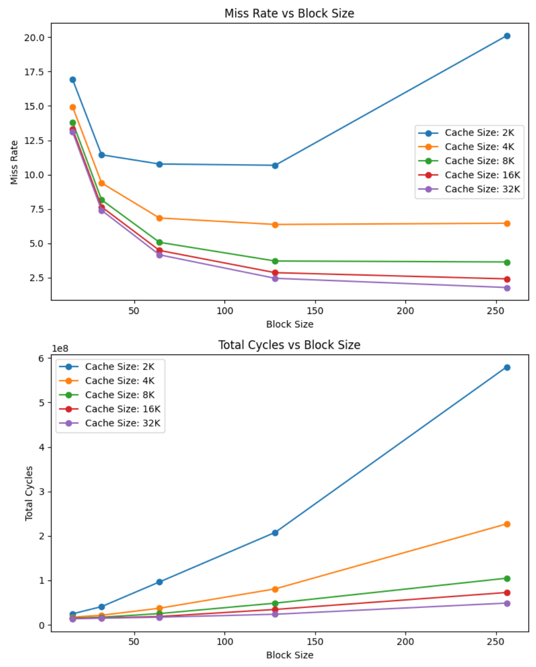
Block Sizes

I simulated the cache across various cache sizes, specifically `2KiB, 4KiB, 8KiB, 16KiB`, and `32KiB`. Each of these configurations again used varying block sizes. This allowed me to find the relationship between block size and cache efficiency. Results were plotted to visualize the trends and derive conclusions (see figure 1).

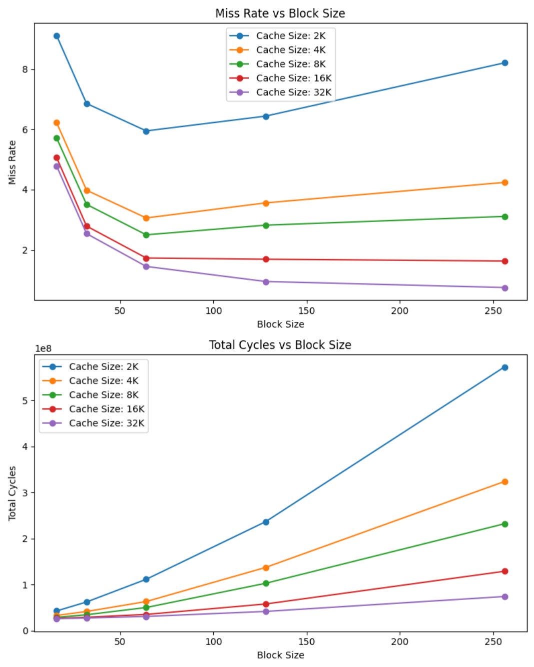
Associativity

Next, I examined how different levels of *associativity* influence cache performance. Associativity values from 4 to 64 were tested with a constant cache size of `2048B` and a block size of `16B`. The simulation results for each associativity level were then plotted (see figure 2).

[1] An *associativity* of m indicates that each line in the cache has m blocks

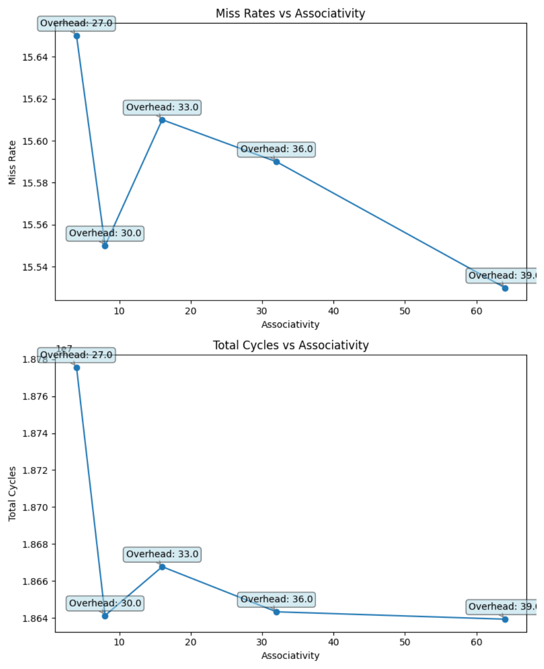


(a) Plot 1

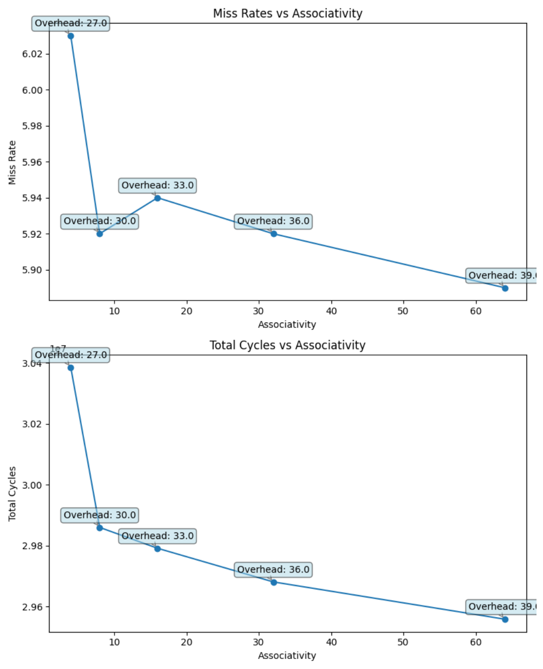


(b) Plot 2

Figure 1: Miss Rates and Clock Cycles against Block Sizes

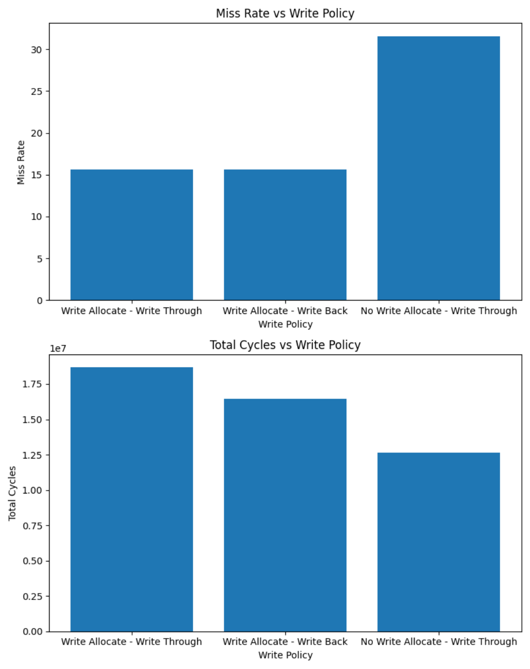


(a) Plot 1

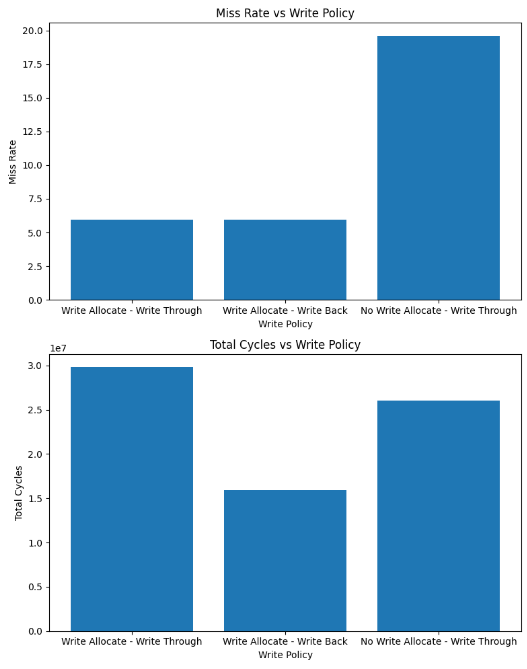


(b) Plot 2

Figure 2: Miss Rates and Clock Cycles against Associativity



(a) Plot 1



(b) Plot 2

Figure 3: Miss Rates and Clock Cycles against Write Policies

Write Policies

I tried all combinations of the different hit and miss policies for a *store* to formulate different write policy scenarios for the cache. Although four combinations are possible, I **excluded the pairing of write-back with no-write-allocate** as it is not typically used in practice. The remaining three write policies were simulated using a cache configuration of eight cache lines, each containing 16 blocks of `16 bytes`. The different results were plotted on a bar graph (see figure 3).

Eviction Policies

Finally, the effect of eviction policies was assessed, namely *FIFO* and *LRU* (Least Recently Used). For both of them, I used the same cache specifications used in the previous section, but set the write policy to write-back with write-allocate. Once again, the results were plotted on a bar graph (see figure 4).

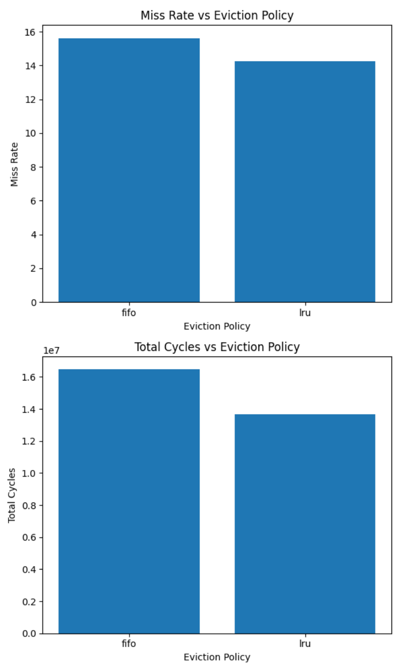
Results

The various experiments yielded a lot of interesting observations. The results of all the experiments are summarized below:

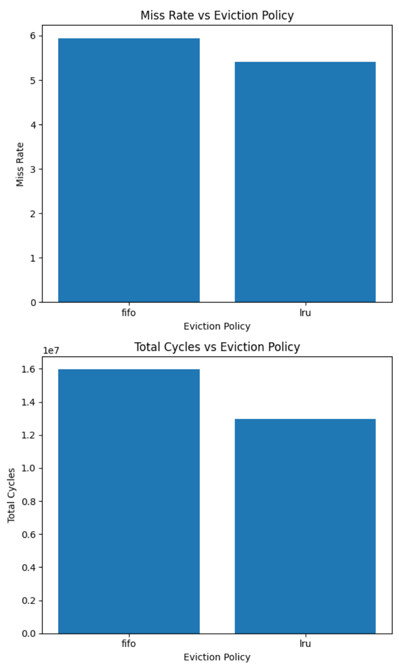
1. **Block Sizes**: It was observed that increasing the size of the cache blocks generally leads to **reduced miss rates**. However, there is an interesting trade-off as the **miss penalty increases substantially** due to the higher cost of loading larger blocks during a cache miss. Additionally, when the block size makes up a significant portion of the total cache size, the miss rates tend to increase due to the reduced number of blocks available, creating competition for cache space.

2. **Associativity**: Higher levels of associativity resulted in both **lower miss rates and reduced total cycle counts**, indicating improved cache performance. However, increased associativity comes at the cost of **higher overhead**, requiring additional resources. Additionally, caches with higher associativity levels are generally more difficult to implement in hardware.

3. **Write Policies**: Among the tested policies, **write-back coupled with write-allocate** seemed the most effective, optimizing both miss rates and speed. This makes sense as write-allocate helps



(a) Plot 1



(b) Plot 2

Figure 4: Miss Rates and Clock Cycles against Eviction Policies

reduce the clock cycles while write-allocate ensures the cache remains up-to-date. Besides, write-through and no-write-allocate were more efficient when direct memory writing was preferable, as both strategies inherently write directly to memory, bypassing the cache.

4. **Eviction Policies**: The performance of the eviction policies was mostly the same, although **LRU showed a slight advantage over FIFO**. This outcome is expected, as LRU leverages temporal locality better than FIFO, leading to slightly improved cache performance under typical memory usage patterns.

Conclusion (Best Cache Configuration)

Considering the results of all the experiments, I believe the best cache is: **a 16 way-set-associative LRU cache with a block size of `16B`**. The **number of sets and blocks can be found using the cache size** which can be chosen as needed.[2] The cache must use **write-back along with write-allocate**. It should be noted that there is *no best cache* and a lot of it comes down to use requirements and design choices. However, under the constraints of this assignment the above seemed the best choice.

[2]For instance if the size is `2048B`, we have 8 cache lines containing 16 blocks each