COL216 LAB 3 : Cache Simulation Analysis

Poojan Shah 2022CS11594

April 2024

Introduction

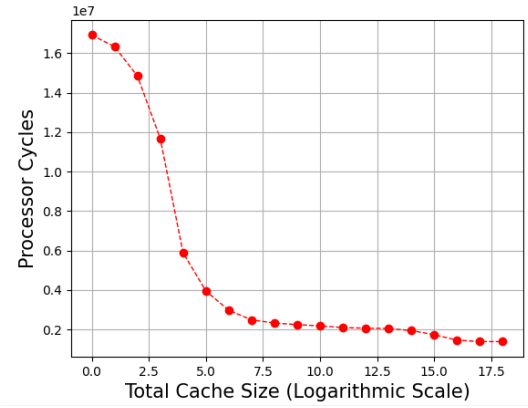
We use the file `gcc.trace` as a benchmark to evaluate cache performance. The file `plot.py` includes the code used to generate the plots and for data analysis. To get the required data for exploratory analysis, we run the cache simulator on each combination of the six possible parameters. The values for the numerical parameters $nsets, nblks, blksz$ were taken from 1 to 4096. So we have a total of '$2*2*2*13*13*13 = 17576$ data points for the analysis. This data is available in the file `data.txt`. When the plots are shown in which x-axis is the Total cache size, we have taken the best across all the non-constant parametrs to give a better perspective (You may see the python code for more detail). To analyse the effect of cache parameters, we will be considering the following aspects:

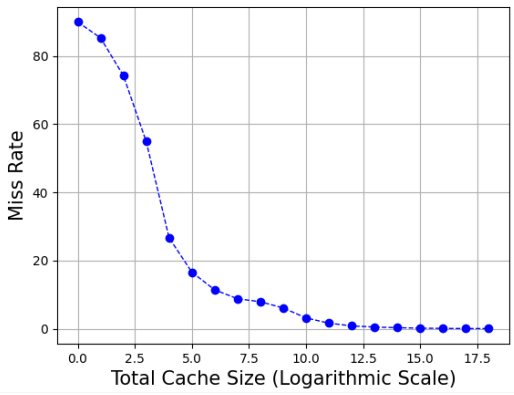
- Effect of Total Cache Size
- Effect of Associativity
- Effect of Block Size
- Effect of Eviction Policy
- Effect of Write Policies

We must note that the metric against which the performance is evaluated is also a design decesion - we will mostly use the total number of processor cycles for execution since it gives a more realistic measure of performance. Another factor to consider is the hit/miss rate of the cache.

Total Cache Size & Overhead

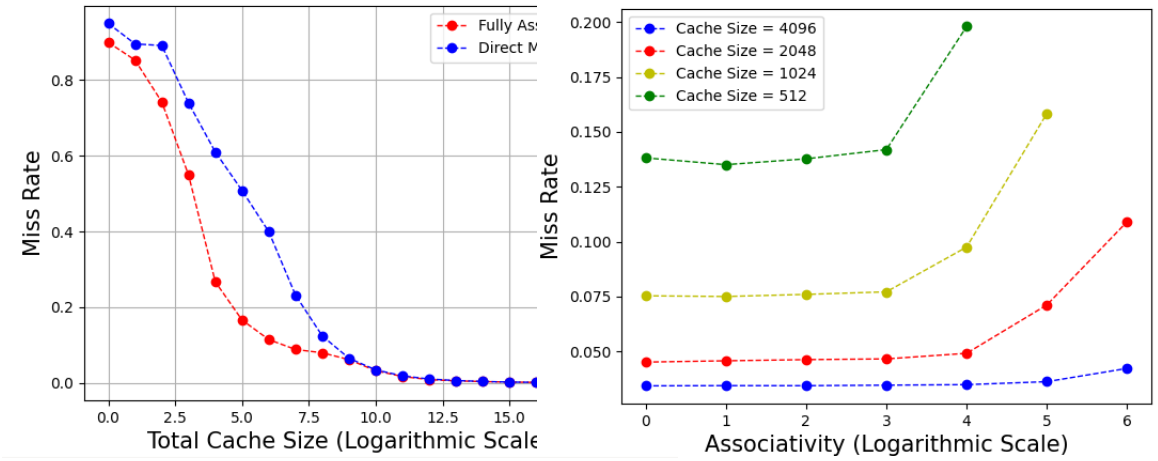
As the total cache size ($nsets * nblks * blksz$) increases, we see a clear increase in cahce performance. But as the cache size increases, the cost of the cache also increases. Hence, we have an obvoius performance-cost tradeoff. We need a cache size which is large enough for a good performance, but not too large to control the cost. The following graph shows the performance (measured as the minimum processor cycles and minimum miss rate) for various cache sizes. Notice the *elbow curve*. After cache size of $2^{10}, 2^{11}$, we do not see significant increase in performance even when doubling the cache size. As the size increases, overhead (memory needed to store tag bits) also increases almost linearly. Overhead reduces the effective cache size available for storing useful data, as it consumes some portion of the cache capacity. This can lead to a decrease in the cache hit rate, and we must take it into account.





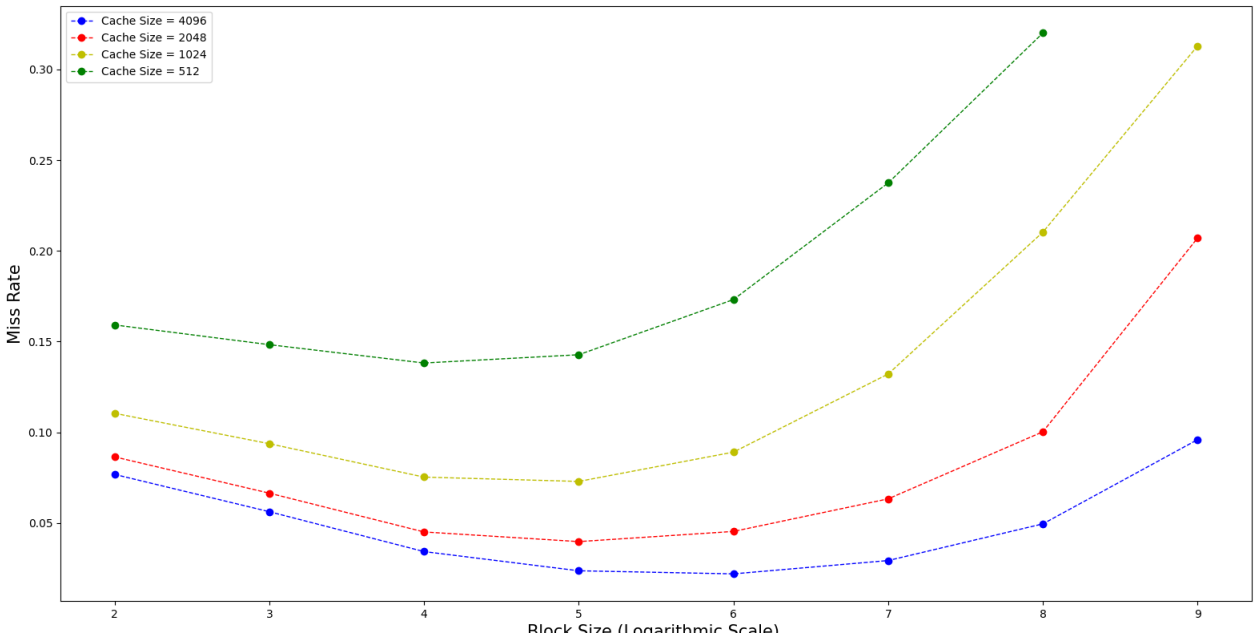
Associativity

Fully associative caches provide flexibility in block placement, reducing cache misses but increasing complexity and hardware overhead (which we do not consider in this assignment). Direct-mapped caches are are more prone to cache conflicts. We find that for hit rate, fully associative caches perform better than direct mapped ones. The following graph shows the variation of miss rate with the number of sets in the cache , with the block size being constant at 4 words. We see that the miss rate increases with more directness. There is a slight minima for the number of sets $nsets = 2^2$.



Block Size

Larger block sizes enhance cache hit rates by capturing more spatial and temporal locality, allowing more useful data to be fetched into the cache at once. However, they can lead to cache pollution by loading unnecessary data, potentially evicting valuable information prematurely. Finding the optimal block size involves balancing improved cache utilization and hit rates against the risks of cache pollution and increased miss penalties. Moreover, increasing block size has a large effect on the total number of cycles, since allocating blocks takes time proportional to the block size. The following graph shows the variation of miss rate for various block sizes and cache sizes. We see that there is an optimal block size of nearly 2^6 bytes = 8 words.



Eviction & Write Policies

We found that among write policies, the `write-back` policy significantly outperforms the `write-through` policy since it avoids the unnecessary memory access during each write-hit. Similar results were found for `write-allocate`. Although it gives up some performance during a write miss, but due to the moving of missed data from main memory to cache, it increases future hit rates. The performance of the `lru` and `fifo` policies were almost the same, but `lru` gains a slight advantage due to using the principle of temporal locality.

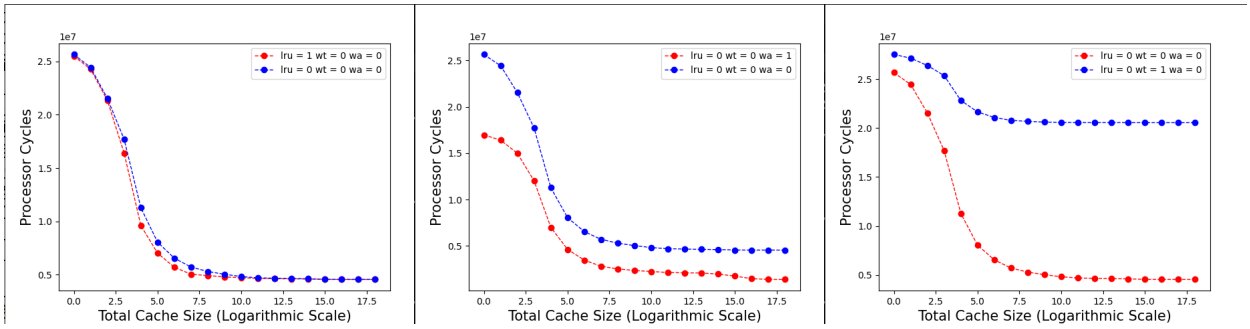


Figure 1: Images

Conclusion and Insights

From the above analysis, we decided that the optimal cache parameters are

- `2-way-set-associative`
- `32 blocks in each set`
- `8 word blocks`
- `write-allocate`
- `write-back`
- `lru eviction policy`

For these cache parameters, we obtained a hit rate of 96.8 %. These parameters are a general suggestion, and according to application, we may have different parametrs. We have neglected the hardware overhead for implementing fully-associative caches, and also neglected the effect of cache size on the clock pulse width, since they were not part of the assignent, but including them can lead to a more accurate parameters.