**Final Project Report: Cache Simulator**

**1.1 Introduction**

This project falls under the field of Computer Architecture and Memory Systems, focusing on the analysis of the performance of different cache configurations. My motivation for this project began with my Computer Architecture course, where I programmed a basic cache simulator. Because of my prior knowledge of this topic, I decided to further pursue a similar project idea and expand on it. I am also interested in exploring different cache architectures and memory access patterns.

Some important terminology that I would like to explain includes cache memory, cache, cache hit and miss, replacement policies, cache associativity, cache size, read operations, and trace file. Cache is a temporary storage area that holds frequently used data to improve access speed and allow faster retrieval. Cache memory is a small, fast memory located close to the CPU, used to store frequently accessed data. A cache hit occurs when a request for data is found in the cache, meaning it has been stored and can be accessed quickly. A cache miss means the requested data was not found in the cache. Replacement algorithms determine which data to remove from the cache when it becomes full, with the goal of improving overall performance. Cache associativity defines how data is mapped and accessed within the cache; the main types include direct-mapped, set-associative, and fully associative. Cache size refers to the amount of memory allocated for storing cached data and is an important factor in performance optimization. A read operation retrieves data from memory or storage without modifying it.

Cache Memory has been an important part of computer systems for many years, originally created to merge the gap between CPU speed and slower main memory. As modern applications are becoming more complex and memory intensive, enhancing memory access has been more important than ever. This project explores those key foundational concepts in cache architecture and analyze them in a practical simulation.

The purpose of this project is to develop a flexible cache simulator that can model various cache behaviors based on input parameters. From this simulation, I plan to learn about how cache configurations like cache size, associativity, and replacement policy can affect performance metrics like hit and miss rate.

The goals of this project are to implement a fully functional cache simulator in C++. I plan to implement multiple replacement policies and various cache configurations, such as cache size, block size, and associativity. I will analyze results such as hit rate, miss rate, and the different types of miss rates based on memory traces. Additionally, I aim to compare and interpret the impact of memory access patterns using multiple trace files, including both random and sequential patterns.

**1.2 Methodology**

The different cache replacement policies I will be analyzing are FIFO, LRU, Optimal (also known as Belady’s algorithm), the Clock Algorithm (1-Bit and 2-Bit), and LIFO. I will summarize how each of these policies works. FIFO (First-In, First-Out) replaces the oldest block in the cache—the one that was loaded first—regardless of usage frequency. LRU (Least Recently Used) removes the block that has not been accessed for the longest time. Optimal, or Belady’s algorithm, selects the block that will not be used again for the longest period in the future, providing the best possible hit rate in theory. The Clock Algorithm (1-Bit) approximates LRU using a circular buffer and a reference bit that gives each block a second chance before replacement. The 2-Bit Clock Algorithm improves upon this by using two reference bits to more accurately track recent usage. Lastly, LIFO (Last-In, First-Out) replaces the most recently added block, functioning in the opposite manner of FIFO.

Now we will discuss the different associativity’s that we will be comparing, which include direct-mapped, 2-way, 4-way, and fully associative. A direct-mapped cache maps each memory block to exactly one cache line, making it simple and fast but prone to conflict misses. A 2-way set-associative cache allows each block to be placed in one of two locations within a set, reducing conflicts while maintaining reasonable speed. A 4-way set-associative cache increases flexibility by allowing placement in one of four locations per set, further decreasing conflict misses at the cost of more complex hardware. A fully associative cache allows any block to be placed in any line, providing the lowest conflict miss rate but requiring the most complex and time-consuming searching.

We will be using an 8-block cache. The cache size can be calculated using the formula: Number of Blocks × Block Size. Since we are using 8 blocks, the sizes we will test include:

* 8 blocks × 64B = 512B
* 8 blocks × 128B = 1024B
* 8 blocks × 256B = 2048B
* 8 blocks × 512B = 4096B

These are the cache sizes that will be compared: 512B, 1024B, 2048B, and 4096B, and we will discuss how they impact cache performance.

Now, we will introduce the different formulas used to calculate cache statistics.

* Hit Rate (%) = (100 × totalHits) / totalAccesses
* Miss Rate (%) = (100 × totalMisses) / totalAccesses
* Block Size = totalSize / totalBlocks
* Total Accesses = totalHits + totalMisses

Furthermore, we will explain the different types of cache misses. The three types of misses are compulsory (cold), capacity, and conflict. Compulsory misses occur when a block is accessed for the first time and is not yet in the cache. A capacity miss happens when the cache cannot hold all the required data, causing older blocks to be evicted. A conflict miss occurs when multiple blocks compete for the same cache set in set-associative or direct-mapped caches. I will be analyzing these different types of misses and recording how many of each occur.

I programmed my cache simulator using C++ and developed it in Visual Studio Code. In VS Code, I used the Ubuntu WSL environment to run the code. I will be testing the simulator with two trace files—one with random access patterns and one with sequential access patterns, each containing 500 memory access operations. I chose to use trace files with a higher number of memory accesses to obtain more accurate and meaningful results.

I will discuss what standard library headers were used in our code. I used stdio.h, which includes basic I/O operations. I used the string header file for working with strings. I used stdlib.h, which provides basic library functions for memory allocation and process control. I created multiple files to organize my code and separate them based on functionality. I created a header file called cache.h that defines the important data structures and variables needed in my project. I divided the functionalities into multiple files, such as one for cache operations, one for the main function—which includes defining the cache configurations and printing out simulation results—and one for implementing the replacement policies. I also added a makefile to run the program An additional feature that I implemented in the command line is an extra parameter called [-t] that lets us see the tag for every memory access and the image of the set. It enables a debug mode that helps us better visualize the cache behavior by showing exactly when memory access results in a hit, miss, or eviction. The test run input is:  
./cache-sim-os <trace-file> <associativity> <cache-size> <replacement-algorithm-#> [-t]

The system architecture of the cache simulator is designed to show the interaction between hardware-level cache configurations and software-focused memory access behavior. The results exemplify how real CPU caches handle memory. By combining all the components mentioned above, we can analyze how the simulator works with these architectural decisions and how it impacts system performance, allowing us to understand memory hierarchy behavior.

**1.3 Results**

First, I will discuss the results for the randomized trace file with 500 memory accesses (read/write). Using the randomized trace file, I will be comparing the hit/miss rates for different cache associativity’s with each replacement policy using a constant 1024 cache size. When comparing the direct-mapped cache, the hit rate and miss rate were consistent across all replacement policies, with a hit rate of 71% and a miss rate of 28%. This is because direct-mapped associativity does not have a second choice for where to place a block from main memory into the cache, which means there is no choice of what to replace. When using multi-block sets, more than one address maps to the same set, so you have a choice of which address chunk to remove when you need to replace it with a new chunk that needs space. On the other hand, with direct-mapped caches, which are 1-block sets, there is no other choice other than the one specific location in the cache.

For the 2-way associative cache, I noticed a slight improvement in the hit rate for each of the replacement policies. The hit rate for FIFO was 72%, which is slightly lower than LRU because FIFO can evict an old block without considering how frequently it has been used, which makes it less effective than LRU. To elaborate, LRU showed a higher hit rate of 74% because blocks are accessed again shortly after they are used, keeping recently accessed blocks longer, which leads to a more efficient cache by removing less frequently used data. In contrast, the Optimal Replacement policy, also known as Belady’s, had the highest hit rate of 78% because it has the ability to predict future memory access traces and evict blocks that won’t be used for the longest time in the future. This means it is likely to always make the best eviction decision, lowering the chances of unnecessary evictions that could lead to misses. Optimal is said to be the most accurate because it can see into the future and make better replacement decisions, even though it is mostly used as a theoretical benchmark.

Furthermore, the Clock Replacement Algorithm 1-bit produced a hit rate of 72%, and the 2-bit clock was 73%. The 2-bit clock was higher than the 1-bit because it keeps blocks slightly longer and gives them another chance before being replaced. The 1-bit is an approximation but suffers from a drawback because it tends to evict blocks too quickly, even if they are useful. Finally, the last policy is LIFO, which produced a lower hit rate of 71% because, similar to FIFO, it does not consider usage—it only removes the most recently added block from a set when it is full. This can be a disadvantage because recently used blocks are more likely to be used again, which LIFO immediately discards, and it does not keep track of access patterns, leading to a higher number of misses.

Next, I will compare the results for the 4-way associative cache. I noticed the same overall trends discussed above for the 2-way associative cache. The hit rate for FIFO and LRU was 75%, 81% for Optimal, 76% for Clock (1-bit), 77% for Clock (2-bit), and 69% for LIFO.

Lastly, for the fully-associative cache, the results were similar: the hit rate for FIFO was 76%, LRU was 79%, Optimal was 82%, Clock (1-bit) was 77%, Clock (2-bit) was 79%, and LIFO was 59%. Although LIFO had a drastically lower hit rate than the other policies, overall, I noticed that the hit ratio was relatively high for all the cache replacement policies, and Optimal proved to be the most efficient cache replacement algorithm.

One common trend that was noticed is the increased hit rate as the associativity increases. This is because in a direct-mapped cache, every block can only be assigned to one specific set, which can lead to a higher number of evictions due to conflicts when multiple blocks map to the same set. On the other hand, with 2-way or 4-way associative caches, each block can be assigned to more than one spot, making it less likely to be replaced. With fewer conflicts, we can expect fewer blocks to be evicted, thus leading to a higher hit rate. This is further supported by the increased hit rate observed in the fully-associative cache.

Lastly, I will compare the common trend among the three different types of misses for each replacement policy. The total number of compulsory (cold) misses is 8, the number of conflict misses ranges from about 3 to 16, and the majority of the misses are capacity misses. The reason behind this is that the total number of memory blocks being accessed is greater than 8, which means that even if blocks are reused, they can’t stay in the cache long enough because they are removed due to lack of space. A block size of 8 is too small for the workload's set, leading to a higher number of capacity misses, which is one downside.

**Random Trace File: trace-MINIFE.500.txt:**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *8 block cache, Direct-Mapped, 1024 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 71% | 71% | 71% | 71% | 71% | 71% |
| Miss Rate: | 28% | 28% | 28% | 28% | 28% | 28% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 144 | 144 | 144 | 144 | 144 | 144 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 16 | 16 | 16 | 16 | 16 | 16 |
| Capacity: | 128 | 128 | 128 | 128 | 128 | 128 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *8 block cache, 2-way, 1024 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 72% | 74% | 78% | 72% | 73% | 71% |
| Miss Rate: | 27% | 25% | 22% | 27% | 26% | 28% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 138 | 129 | 110 | 138 | 114 | 142 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 3 | 3 | 3 | 3 | 3 | 3 |
| Capacity: | 127 | 118 | 99 | 127 | 123 | 131 |
|  |  |  |  |  |  |  |
| *8 block cache, 4-way, 1024 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 75% | 77% | 81% | 76% | 77% | 69% |
| Miss Rate: | 24% | 22% | 18% | 23% | 22% | 31% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 122 | 111 | 92 | 118 | 113 | 155 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 3 | 3 | 3 | 4 | 4 | 5 |
| Capacity: | 111 | 100 | 81 | 106 | 101 | 142 |
|  |  |  |  |  |  |  |
| *8 block cache, Fully-Associative, 1024 Cache Size:* | | | | |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 76% | 79% | 82% | 77% | 79% | 59% |
| Miss Rate: | 24% | 20% | 18% | 22% | 20% | 40% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 120 | 104 | 90 | 111 | 104 | 204 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 0 | 0 | 0 | 0 | 0 | 0 |
| Capacity: | 112 | 96 | 82 | 103 | 96 | 196 |

Now, I will discuss the results of the sequential trace file. I noticed an interesting trend: the results for the sequential trace file were consistent across all replacement policies as well as the different associativity’s. The hit rate and miss rate were about 50%, and there were no conflict misses; most of the misses were capacity misses. This is because sequential trace files are highly predictable and follow an ordered pattern. The predictable access and deterministic nature of the trace file can greatly affect the behavior of the cache. Furthermore, accessing memory in a sequence demonstrates principles like spatial and temporal locality. Spatial locality means that if a specific memory address is accessed, it is more likely that neighboring memory locations will be accessed soon after—this occurs when data is accessed linearly. Temporal locality is the idea that if a memory location is accessed, it is more likely to be accessed again in the near future. The nature of sequential trace files causes predictable memory access patterns, which explains why the results are similar across different cache policies. In addition, with the sequential trace file, I noticed that the majority of the misses were capacity misses—similar to the trend mentioned above.

**Sequential Trace File: trace-sequential.64bytes.500.txt:**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *8 block cache, Direct-Mapped, 1024 Cache Size:* | | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | | **Clock2** | **LIFO** |
| Hit Rate: | 50% | 50% | 50% | 50% | | 50% | 50% |
| Miss Rate: | 50% | 50% | 50% | 50% | | 50% | 50% |
| Total Accesses: | 500 | 500 | 500 | 500 | | 500 | 500 |
| Total Misses: | 250 | 250 | 250 | 250 | | 250 | 250 |
| Compulsory: | 8 | 8 | 8 | 8 | | 8 | 8 |
| Conflict: | 0 | 0 | 0 | 0 | | 0 | 0 |
| Capacity: | 242 | 242 | 242 | 242 | | 242 | 242 |

Note: I did not include the rest of the sequential results because they were the same

Now I will compare the different cache sizes with the randomized trace file. The different cache sizes we will be comparing are 512, 1024, 2048, and 4096, with an 8-block cache size. I noticed a similar trend when comparing the replacement policies as explained above. One key observation I noticed is that the hit rate consistently increases as the cache size increases. For example, the hit rate for Optimal with a cache size of 512 is 74%. When we increase the cache size to 1024, the hit rate increases to 78%. Similarly, it increases to 83% and 85% for larger sizes. The reason the hit rate consistently increases as we increase the cache size is because having a larger cache means we can store more data, which results in more available memory locations. It also makes it more likely that frequently accessed data will be present in the cache when it’s requested, thus explaining the outputs. In addition, Optimal has confirmed that it is the best-performing replacement policy. After comparing the sequential trace file, I noticed the same pattern explained above with the different replacement policies. Although the results stayed consistent for the various algorithms, there was a change in the hit and miss rates when testing the different cache sizes. I noticed that the hit rate increased as the cache size increased, similar to the trend discussed above, with the hit rate rising from 50% to 87%. We can confirm that this trend was observed in both the random and sequential trace files.

**Random Trace File: trace-MINIFE.500.txt**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| *8 block cache, 2-way, 512 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 70% | 71% | 74% | 70% | 71% | 66% |
| Miss Rate: | 30% | 28% | 25% | 30% | 29% | 33% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 150 | 141 | 128 | 150 | 147 | 166 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 4 | 4 | 4 | 5 | 5 | 6 |
| Capacity: | 138 | 129 | 116 | 137 | 134 | 152 |
|  |  |  |  |  |  |  |
| *8 block cache, 2-way, 1024 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 72% | 74% | 78% | 72% | 73% | 71% |
| Miss Rate: | 27% | 25% | 22% | 27% | 26% | 28% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 138 | 129 | 110 | 138 | 114 | 142 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 3 | 3 | 3 | 3 | 3 | 3 |
| Capacity: | 127 | 118 | 99 | 127 | 123 | 131 |
|  |  |  |  |  |  |  |
| *8 block cache, 2-way, 2048 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 79% | 81% | 83% | 79% | 88% | 77% |
| Miss Rate: | 20% | 19% | 16% | 20% | 19% | 22% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 102 | 95 | 82 | 101 | 98 | 113 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 2 | 1 | 1 | 1 | 1 | 1 |
| Capacity: | 92 | 86 | 73 | 92 | 89 | 104 |
|  |  |  |  |  |  |  |
| *8 block cache, 2-way, 4096 Cache Size:* | | | |  |  |  |
|  | **FIFO** | **LRU** | **Optimal** | **Clock** | **Clock2** | **LIFO** |
| Hit Rate: | 83% | 84% | 85% | 83% | 84% | 79% |
| Miss Rate: | 16% | 15% | 14% | 16% | 16% | 20% |
| Total Accesses: | 500 | 500 | 500 | 500 | 500 | 500 |
| Total Misses: | 82 | 77 | 71 | 82 | 80 | 101 |
| Compulsory: | 8 | 8 | 8 | 8 | 8 | 8 |
| Conflict: | 0 | 0 | 0 | 0 | 0 | 0 |
| Capacity: | 74 | 69 | 63 | 74 | 72 | 93 |

I was able to achieve all the goals I had planned for this project. I tested different cache replacement policies and cache configurations and achieved the expected benchmark results based on the simulator outputs. Lastly, I successfully compared and evaluated the data by analyzing the hit and miss rates under various conditions.

Some technical obstacles I faced included being unable to implement the Clock-Pro algorithm I originally planned to use due to its complexity and time constraints. Instead, I decided to implement a 2-bit Clock algorithm as a replacement. In addition, my main focus was on the read operations of the program. I had hoped to implement write operations as well, but due to time limitations and technical challenges, I decided not to include them. Moreover, I had some difficulty implementing the Clock algorithms, as I had no prior knowledge of the topic. I also struggled to understand the different types of cache misses and had to do additional research to fully grasp them.

After completing this project, I gained a deeper understanding of how cache memory works and how different cache policies can significantly affect performance. In addition, I learned how to manage my time, overcome obstacles, and ask for help and guidance when needed. Because of time constraints and the heavy workload from other classes, such as Senior Design, I had to adjust and limit the scope of this project—but I was still able to achieve my primary goals. I learned about new algorithms such as Clock and Optimal and how to implement them. I also improved my debugging and technical skills and enhanced my knowledge in the field of computer architecture.

**1.4 Conclusion**

Through the process of completing this project, I was able to gain a deeper understanding of cache systems and programming a custom-built simulator. Based on the findings from my results, I observed that the higher the associativity, the higher the hit rate. I noticed that LRU performed better than FIFO, and Optimal performed the best out of all the policies. I explained the reasoning behind these observations in the results section. I was able to compare the Clock replacement algorithms and noticed that the 2-bit version performed better than the 1-bit, which had results similar to FIFO. LIFO, on the other hand, had a lower hit rate than the rest of the algorithms. I also noted that direct-mapped caches produced similar results across all replacement policies, and I explained why this occurs in the results section. For the sequential trace file, I found that the results were similar across replacement policies and associativities. However, I noticed that the results changed when the cache size was varied, while remaining consistent across algorithms, as previously explained. After comparing the different types of misses, the trend I observed was that the majority were capacity misses, with only a few cold and conflict misses. Despite these differences, I was able to achieve a high hit rate with various configurations, which demonstrates the effectiveness of the cache strategies and implementations based on the memory access patterns.

I was able to understand how memory access patterns can significantly affect cache performance. Implementing the different cache replacement policies helped me learn more about them and how theoretical concepts can be applied in a realistic cache simulator. This project gave me the opportunity to build on the concepts I learned in my undergraduate computer architecture and operating systems classes. I was able to apply my prior knowledge from implementing a simpler version of a cache simulator in my computer architecture class and use this experience to grow and tackle a more complex and challenging project.

One area that I was not able to explore was thrashing, which occurs when performance severely decreases because the system spends more time evicting blocks than executing important operations. Even though it would have been interesting to investigate whether different configurations could cause thrashing, I was not able to focus on that due to the complexity of the scenario and time constraints.

In the future, I would like to focus on extending the functionality of my project by adding write operations and incorporating more replacement policies such as Clock-Pro and MRU/LRU. I would also like to test the code with more diverse trace files and perform a more in-depth comparison of the results. Additionally, I hope to expand the scope of my project by adding a multi-level cache and enhancing the user interface. I am also interested in conducting further research in the field of computer architecture and potentially extending this simulator to handle page replacement in virtual memory systems.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note: I took assistance from ChatGPT when listing the terminology and for guidance purposes.

Please reference the makefile for running the code.

Example Compilation Command: ./cache-sim-os trace-MINIFE.500.txt 2 4096 5