Two-Level Cache Simulation with Replacement Policies

Sai Sriman Kudupudi[1]
Computer Science Department
University of South Dakota, Vermillion, South Dakota
Saisriman.kudupudi@coyotes.usd.edu

Sai Neeraj Samineni[2]
Computer Science Department
University of South Dakota, Vermillion, South Dakota
Saineeraj.samineni@coyotes.usd.edu

Venkata Surya Deepak Lakshimpalli[3]
Computer Science Department
University of South Dakota, Vermillion, South Dakota
VenkataSuryaDeepak.L@coyotes.usd.edu

Lalitha Priya Bijja[4]
Computer Science Department
University of South Dakota, Vermillion, South Dakota
Lalithapriya.bijja@coyotes.usd.edu

Abstract – Cache memory is an essential component of contemporary computer systems that helps to increase overall system performance by facilitating quick access to data that is often accessed. The goal of this research is to simulate cache memory activity and assess how various replacement rules affect cache performance. A two-level cache system is examined in relation to two main replacement policies: Least Recently Used (LRU) and First-In-First-Out (FIFO).

In order to simulate memory access patterns, the paper first implements a cache memory model that includes L1 and L2 caches. Cache hits, misses, and replacements are monitored using a range of workloads, such as sequential and random accesses. The purpose of the simulation is to provide light on how effective the caching mechanism is in various situations.

Keywords – Cache, Cache Replacement Policy, Two-level Cache Architecture, Least Recently Used Replacement policy, First In First Out Replacement policy

I. Introduction

A key component of any memory hierarchy's architecture is the cache replacement policy. A cache system's hit rate and access latency are both impacted by the replacement policy's effectiveness. The replacement policy becomes more important the higher the cache's associativity. Consequently, a great deal of work has been suggested, both in business and education, to improve the performance of the policy for replacement.

The memory system comprises a tiered structure of caches rather than a singular cache. If each cache within the hierarchy independently decides which block to replace, it can significantly degrade the performance of the entire memory hierarchy, particularly when inclusion needs to be maintained. For example, if the level 2 cache opts to evict a Least Recently Used (LRU) block, it may inadvertently eliminate one or more blocks at level 1 that are not LRU, leading to a decline in level 1's hit rate and overall system performance.

Optimizing memory access is essential for improving system performance in the field of computer architecture and system design. Cache memory is an essential component in reducing the delay related to retrieving data from slower memory tiers. It is a compact, high-speed storage device placed between the processor and main memory. This project simulates cache memory behavior and examines the impact of various replacement rules on system performance, providing an in-depth look at the complex world of cache memory.

Due to its focus on retaining recently accessed items, LRU replacement proves beneficial in capturing temporal locality. Unlike FIFO, implementing LRU can pose greater complexity because it requires updating the access history for each cache line. Various strategies, such as counters, linked lists, and specialized algorithms, can be employed to effectively track and update access history.

A. Two level Cache Architecture

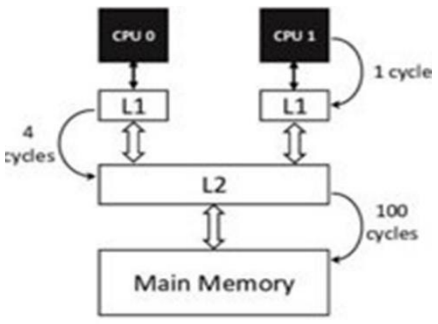


Fig : 2 – level Cache Architecture

A computer system employs a two-level cache architecture by integrating two tiers of cache memory, known as L1 and L2 caches. This dual-cache arrangement is designed to decrease the latency in accessing regularly utilized data. Acting as intermediaries between the processor and main memory, each cache level contributes to enhancing the overall performance of the system [1].

Cache Level 1 (L1): The L1 cache, which is either on the same chip as the CPU cores or very close to them, is the closest cache to the processor. Because of its close proximity, the CPU can receive data from it fast and with minimal delay.

Size and Speed: L1 cache has a lower overall size but a faster access rate. It enables the processor to access data and instructions rapidly during operations by storing a subset of the most frequently used information.

Direct Communication with CPU Cores: The L1 cache communicates directly with the CPU cores to deliver quick access to vital data for processing.

L2 Cache (Level 2): L2 cache serves as an intermediary between L1 cache and main memory. It is situated in this manner. Compared to L1, it acts as a bigger but marginally slower cache. Its function is to gather extra information and commands that might not fit in the L1 cache.

Shared or Split Architecture: The L2 cache can be divided into separate portions for each CPU core in a split architecture, or it can be shared by several CPU cores. Multi-core processors frequently have shared L2 caches.

Greater Capacity: The L2 cache can hold a larger collection of frequently requested data and instructions since it is larger than the L1 cache.

The notion of geographical and temporal locality, which states that programs typically retrieve data and instructions that are close together in both time and location, is what the two-level cache architecture aims to capitalize on. While the L2 cache adds capacity and helps minimize the need to visit the slower main memory for each data request, the L1 cache manages the most important and often accessed data. By balancing speed, capacity, and cost-effectiveness, this hierarchical configuration seeks to maximize system performance as a whole [4].

II. Implementation

LRU & FIFO Replacement policies

The implementation of the project's code involves the use of Python within the Jupyter Notebook development environment. This combination of a versatile and interactive programming language with a dynamic platform enables both code execution and documentation. The Jupyter Notebook's cell-based structure facilitates the seamless integration of code, visuals, and explanatory text, thereby enhancing the overall clarity and understanding of the project. Python's extensive library and strong community support contribute significantly to the project's stability and efficiency. The utilization of Jupyter Notebook promotes experimentation and collaborative efforts, simplifying the process for team members to contribute to and comprehend the codebase.

The Least Recently Used (LRU) cache replacement policy is integrated into the current cache structure in the implemented code. In order to track the order of access, the LRUCacheLine class adds an access_order field to the basic CacheLine class. An OrderedDict is used by the LRUCache class to preserve the order of cache line access. Cache hits in the read method change the access order; cache misses result in the inclusion of new entries; if the cache is full, the least-recently-used entry is removed. On the other side, the FIFOCacheLine class, which derives from

CacheLine, is used to implement the First-In-First-Out (FIFO) cache replacement policy [4].

In order to enforce the FIFO policy, the FIFOCache class uses a deque. Cache hits are determined by determining if any line in the cache has a matching tag. When a cache miss occurs, information is retrieved from main memory and a new cache line is added to the end of the deque. If the cache is full, the oldest entry is automatically removed. The integration of these rules into a two-level cache system facilitates the assessment and comparison of their individual hit, miss, and replacement rates across a range of workload conditions [5].

The following classes have been implemented in this project :-

A. Least Recently Used policy

Class Structure: To track the order of access for LRU replacement, the LRUCacheLine class adds an extra attribute called access_order to the base CacheLine class.

LRUCache Class: In a two-level cache hierarchy, the LRUCache class is the representation of the L1 cache.
It keeps an ordered dictionary of cache lines using an OrderedDict from the collections module. The cache line that hasn't been accessed in a while can be easily found because the order is established by the access history.

The read method first determines whether the requested data is cached (cache hit). In the event of a hit, the accessible item is moved to the end of the ordered dictionary, indicating the most recently utilized, updating the access order. In the event of a miss, the code updates the cache by simulating the fetching of the data from main memory.[2]

First – In First Out Policy(FIFO)

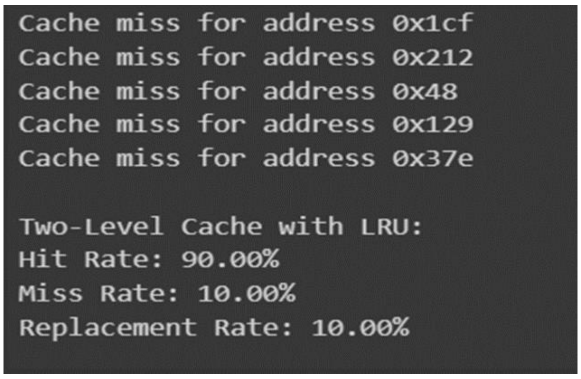
Class Structure: The root class for both FIFO and LRU cache implementations is Cache Line.

Associative Cache Class: In a two-level cache system, the Associative Cache class is the representation of the L2 cache.
It makes use of a simple associative caching paradigm, in which the tag is used to determine whether each cache line matches. A cache hit occurs when a match is detected; a cache miss occurs when none.

In the event of a cache miss, the code updates an empty cache line, employs a particular replacement procedure, and mimics retrieving the data from main memory.

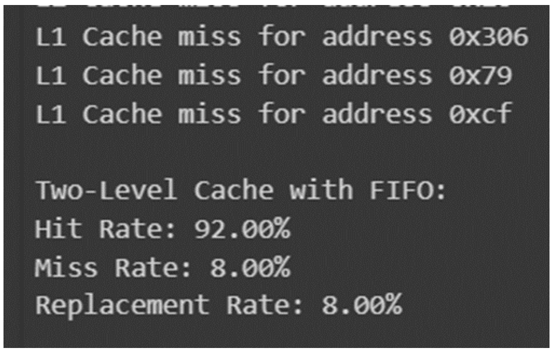
FIFO Cache Evaluation: The hit, miss, and replacement rates for the two-level cache system with LRU and FIFO are computed by the TwoLevelCacheEvaluator class. With a modified replacement policy, the cache system's efficiency can be understood by the FIFO measurements. [3]

III. RESULTS

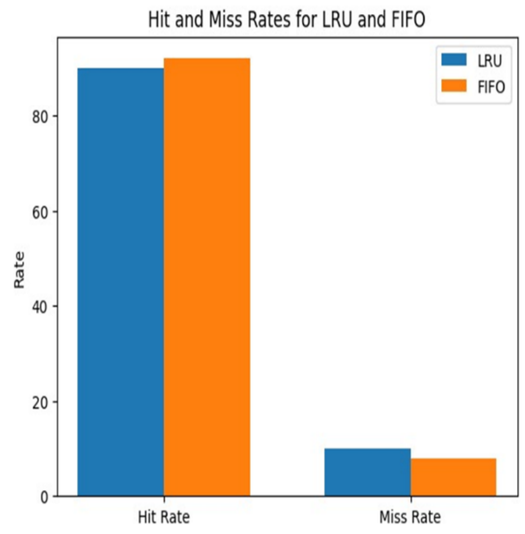


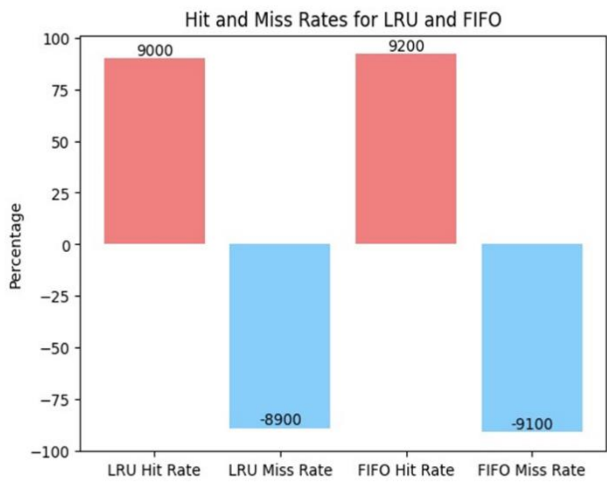
```
Cache miss for address 0x1cf
Cache miss for address 0x212
Cache miss for address 0x48
Cache miss for address 0x129
Cache miss for address 0x37e

Two-Level Cache with LRU:
Hit Rate: 90.00%
Miss Rate: 10.00%
Replacement Rate: 10.00%
```

The results show that the cache missed several addresses. This means that the requested address. This means that the requested data was not found in the cache and had to be fetched from main memory.



```
L1 Cache miss for address 0x306
L1 Cache miss for address 0x79
L1 Cache miss for address 0xcf

Two-Level Cache with FIFO:
Hit Rate: 92.00%
Miss Rate: 8.00%
Replacement Rate: 8.00%
```





IV. CONCLUSION

Performance Comparison: The simulations revealed different performance characteristics in terms of hit, miss, and replacement rates for LRU and FIFO:

- The results showed that LRU had a 90.00% hit rate, a 10.00% miss rate, and a 10.00% replacement rate.
- FIFO showed an 8.00% replacement rate, a reduced miss rate of 8.00%, and a slightly higher hit rate of 92.00%.

Benefits of LRU:

In situations where past access patterns significantly influenced present and future accesses, LRU proved successful.

Its replacement process demonstrated its flexibility in responding to changing workloads by choosing the least often used criterion.[2]

Benefits of FIFO:

On the other hand, FIFO demonstrated strong performance, especially for hit rates.

Because of its replacement policy's simplicity, it works effectively in situations where access patterns are predictable. [3]

Trade-offs and Points to Think About:

The particulars of the application and workload should be taken into consideration while choosing between LRU and FIFO.
While FIFO offers simplicity and possibly better efficiency in some situations, LRU offers flexibility to shifting patterns.

References

[1]https://www.sciencedirect.com/topics/computer-science/cache-controller

[2] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, "Feedback Control of Computing Systems." John Wiley & Sons, 2004

[3] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approach." Morgan Kaufmann Publishers, 2011.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in Journal of Parallel and Distributed Computing, 1996.

[5] P. K. Dubey, S. C. Seth, and D. K. Banerji, "Computer System Organization: B5700/B6700 Series." New Jersey: Prentice-Hall, 1972.