World Journal of Advanced Research and Reviews
eISSN: 2581-9615 CODEN (USA): WJARAI
Cross Ref DOI: 10.30574/wjarr
Journal homepage: https://wjarr.com/

(RESEARCH ARTICLE)

Check for updates

Enhancing Cache Performance Through FIFO replacement algorithm optimization for high-performance computing

Immaculate Chidimma Agubata *, Mary Ofuru Kama and Dickson Apaleokhai Dako

Department of Software Engineering, Faculty of Natural and Applied Science, Veritas University Abuja, FCT, Abuja, Nigeria.

World Journal of Advanced Research and Reviews, 2025, 27(02), 906-913

Publication history: Received on 27 June 2025; revised on 11 August 2025; accepted on 13 August 2025

Article DOI: https://doi.org/10.30574/wjarr.2025.27.2.2918

Abstract

Cache replacement algorithms are used to optimize the time taken for the central processing Unit (CPU) to process information by storing information needed by the processor at that time and possibly in future so that if the processor needs that information it can be provided immediately. There are a number of techniques (FIFO, LRU, CC, LFU, LRU, GDSF, MRU, Hybrid) that are easily used to organize information in such a way that the information needed by the CPU to remain busy and maintain its speed of processing is readily available. FIFO is known for its ease of implementation and low computational complexity. Therefore, this paper examines the scenarios in which FIFO fails and proposes techniques to enhance cache performance by optimizing FIFO through the integration of machine learning to predict future request and perform intelligent prefetching to preload relevant data into the cache. The machine learning approach increased the cache hit and increased the hit time.

Keywords: First-In-First-Out; High Performance Computing; Cache; Replacement Algorithm; Optimization

1. Introduction

Near the CPU (central processing unit), cache memory is a tiny, fast memory unit that holds frequently accessed information and commands. By serving as a buffer between the processor and main memory also known as Random Access Memory (RAM), it speeds up data retrieval and enhances system performance. The cache memory is often characterized by high speed memory access, small size and ability to store frequently used data. Usually, the cache memory is divided into three levels and each is labelled in ascending order according to the closeness to the central processing unit (CPU) [1].

When the CPU needs data, it first checks the cache memory and if the data is found, it is retrieved quickly (cache hit) otherwise, the CPU fetches the data from the main memory and stores a copy in the cache for future access.

Since cache memory greatly increases system efficiency and data access speeds, it is essential to High-Performance Computing (HPC). Workloads in HPC systems include big datasets, frequent memory accesses, and intricate computations.

A cache replacement algorithm is the process in charge of selecting an item from the cache to be removed and substituted with more popular item. The main aim of the cache replacement algorithms is to maximize the cache hit ratio in order to improve other performance measures. They differ in parameters used to select the item to be evicted from the cache and the way the parameters are applied. Some Cache replacement parameters include:

- Cache miss: This refers to an incident where the data is not found in the cache.

* Corresponding author: Agubata Immaculate Chidimma

- Cache hit: This is a situation or incidence where the data is found.
- Hit time: Refers to the time it takes to access the cache.
- Hit Ratio: Refers to the number of times the data is found in the cache [2].

Some generally known cache replacement algorithms include

1.1. First in, first Out

Cache memory is useful because of the following reasons:

- Lowering Latency: Frequently used data is kept closer to the processor because cache memory may be accessed far more quickly than main memory (RAM), which lowers latency.
- Reducing bottlenecks: Memory access speeds frequently restrict CPU performance. This problem is lessened by cache, which offers quick access to data and instructions that are often used.
- Increasing Throughput: Parallel processing is essential for HPC workloads. Instead of waiting for data, processors can spend more time computing when cache is managed effectively.

Cache memory has an important role to play in high performance computing because cache memory has a finite amount of space, some stored data must be swapped out to make room for new data when it fills up. To maximize performance, cache replacement algorithms decide which data should be removed. These methods are essential to HPC because they guarantee that high-priority or frequently requested data stays in cache, minimizing the amount of expensive memory accesses.

2. Literature review

Algorithmic optimization has been widely applied beyond caching systems to improve computational efficiency in various domains. For instance, Agubata et al. [18] applied Dijkstra's algorithm in the design and optimization of a bus booking system, significantly reducing travel route computation times and improving scheduling efficiency. Although, their application targeted transportation management, the principle of leveraging optimized algorithms to minimize latency and improve service delivery parallels the objectives in cache replacement optimization, particularly in HPC environments. Cache replacement algorithms are essential for managing limited cache storage by determining which data to evict when new data needs to be loaded. Common strategies include Least Recently Used (LRU), Least Frequently Used (LFU), and First-In-First-Out (FIFO). FIFO evicts the oldest data first, operating on a simple queue-based principle. While straightforward and efficient, FIFO doesn't account for data access patterns, which can lead to suboptimal performance compared to more adaptive algorithms like LRU. However, FIFO's simplicity allows for lower overhead and easier implementation, making it suitable for certain applications where access patterns are predictable or the overhead of more complex algorithms is prohibitive. Recent research has explored combining FIFO with other strategies to balance simplicity and performance. For instance, the S3-FIFO algorithm integrates FIFO queues with mechanisms to filter out infrequently accessed items, achieving better scalability and throughput in web caching scenarios.

In 2018, a paper by [2] compares the different cache replacement algorithms used in video services such as FIFO, LRU, LFU(Least frequently used), OPT, Chunk-based caching (CC), quality-based caching (QC) and LRU-2. The research applies a ziph distribution to simulate video popularity and evaluates each algorithm under different cache sizes and request rates. The different replacement algorithms were simulated to determine their performance based on time, ease of implementation and other metrics. However, a study by [3] presents a performance comparison simulation of the seven cache replacement algorithms on various internet traffic extracted from the public IRcache dataset. The results of this study indicate that the Hit Ratio (HR) performance is strongly influenced by cache size, cacheable and unique requests. While a research by [4] shows that caching as a concept can be used to save energy and store up energy for energy consumption devices. Again [5]in his research used the PGM technique to improve cache performance by increasing cache hit. The method increased the overall performance of cache hit by 7% but could only work for cache hit ratio. The table below summarises the review based on the algorithm used, methodology, limitations of FIFO, Findings and the merits it inculcates. Cache replacement algorithms are essential for optimizing performance in computing systems. Classical policies such as FIFO, LRU, and LFU have been extensively studied [1]–[5]. Recent works have also explored advanced FIFO optimizations [6]–[8], machine learning-enhanced cache policies [9]–[15], and HPC-specific cache strategies [16], [17].

Yang et al. [6] and Akbari Bengar [7] demonstrated that FIFO, when augmented with hybrid or priority mechanisms, can achieve performance comparable to or exceeding LRU and LFU. Mahni et al. [8] further optimized file-level cache placement in HPC systems using a multi-criteria approach, boosting hit rates and I/O efficiency.

Machine learning-based approaches have significantly advanced cache management. Sethumurugan et al. [9], Shi et al. [10], Vietri et al. [11], Rodriguez et al. [12], Liu et al. [13], Choi and Park [14], and Zhou et al. [15] applied predictive models, deep learning, and reinforcement learning to anticipate access patterns, thereby enhancing hit ratios and adaptability. Again, for HPC workloads, Jamet et al. [16] and Wu et al. [17] confirmed that tailored replacement strategies can yield substantial performance gains in large-scale, performance-sensitive environments.

Table 1 Summary of review

Author (year of publication)	**Algorithm**	**Methodology**	**Limitations of FIFO**	**Findings**	**Merits of FIFO identified.**
Zulfa etal (2023)	FIFO, LFU, LRU, GDSF, GDS, SIZE, LFUDA	Performed a performance comparison of the different cache replacement algorithms on various internet traffic.	FIFO struggles under high access anomalies.	FIFO had lower hit ratio compared to GDSF and LRU.	FIFO has low computational complexity.
Stallings (2019)	FIFO, Random, LFU		Has higher cache miss	FIFO is predictable but inefficient compared to LRU	Has low processing overhead
Osman & Osman (2018)	FIFO, LRU, LFU, LRU-2, QC, CC, OPT	A simulation-based comparison of cache replacement algorithms for video services	It is mainly a simulation-based research	The research found that the CC algorithm achieves the largest hit ratio and performs well even under small cache sizes. However, the FIFO algorithm has the smallest hit ratio among all algorithms.	Easy to implement
Ali (2016)	FIFO, LRU, MRU, LIFO, Hybrid	Developed an algorithm to increase cache hit through reduction and maintenance of overhead	FIFO does not consider access frequency	FIFO underperforms in dynamic access patterns	FIFO is simple and incurs low overhead
Osman (2016)	Not applicable	Energy efficiency of caching in optical networks	It lacks content popularity awareness	FIFO can reduce network energy consumption but not optimized for large scale	It was not based on any algorithm

Hence, from the literature review summary the following findings can be deduced about FIFO:

- FIFO is an easy algorithm to implement and takes less system resources and logic to execute.
- FIFO requires simple modifications and upgrade to outperform other algorithms since it is easy to implement.

Although cache is a very efficient algorithm and performs well in most scenario, it performs poorly in scenarios where data access is unpredictable and workloads are dynamic. Some reasons why cache may not work well includes:

Frequent Reuse of Older Data (Cache Pollution Issue): In database cache querying, older data is been used more than the just recently queued data, hence FIFO does not apply in this type of scenario.

Looping or Cyclic Access Patterns: FIFO performs poorly in scenarios that involves a cyclic or looping structure. This is because FIFO eliminated the older cache items.

The methodology is formulated in such a way to ensure that the advantages/benefits of FIFO is duly harnessed and limitations it possesses are managed with the improved or hybrid approaches that ensures the best results in cache performance.

Multi-level FIFO (ML-FIFO): This methodology proposes to create a multi-level or tiered FIFO queues that will handle different types of data efficiently. This includes:

- Short-term FIFO cache: This will handle short recent request
- Long-term FIFO cache: This will handle frequently accessed data.

Hence, these methodologies ensure that when the cache is full, only the least important data is evicted from the cache.

Dynamic Threshold FIFO (DT-FIFO): This solution works on the premise that if access frequency increases, FIFO behaves like LRU. However, workload is random, FIFO maintains its basic form of first in, first out.

Hybrid Model FIFO (HM-FIFO): This methodology suggests that FIFO be combined with other replacement algorithms to get optimal result. The following combinations are considered to give optimal result.

FIFO with LFU: This method will combine the simplicity of FIFO and the high hit ratio of LFU; therefore, this will work in a way that enables frequently accessed items to stay longer in the cache while rarely accessed items are replaced first.

FIFO with LRU: This will combine the ability to prioritize LRU ability to retain recently accessed data reducing cache misses. It works by setting a threshold where FIFO is used initially but is accessed frequently within a window. It is moved to an LRU-managed sub-cache.

Adaptive FIFO with aging mechanism: This involves the implementation strategy that delays the eviction of frequently accessed items. The methodology works by assigning weights to the different cache items in the queue. The weights will be assigned in ascending order from the most frequently accessed. As the weight increases, the least frequently accessed items increase in the value of their assigned weights, hence the item in the queue with the highest weight is always the item to be evicted next on the FIFO queue. Enabling FIFO check usage patterns and instead of evicting blindly, it delays the eviction of frequently accessed items.

2.1. Intelligent Prefetching & Machine Learning Integration

Predictive Prefetching: This basically involves the use of historical access patterns and predict the future of how data will possibly be accessed. This includes the prediction of pre-load data based on past records. The method integrates machine learning models such as markov chains and neural networks to anticipate the cache entries that will be needed. This is particularly useful in video streaming services and cloud computing workloads.

Reinforcement Learning-Based Cache Replacement: This improvement method uses Artificial Intelligence- driven (AI-driven) decision making from past performance to dynamically adjust cache policies. The basic application areas are in data centers and cloud storage. This methodology adapts in real-time to changing workloads.

3. Methodology

The proposed Machine Learning-Enhanced FIFO (ML-FIFO) framework builds upon the simplicity of the First-In-First-Out (FIFO) cache replacement policy while addressing its key limitation: lack of adaptability to varying access patterns in high-performance computing (HPC) environments. By integrating predictive analytics, the system estimates the likelihood of future access for each cached item and reorders eviction priorities accordingly. The methodology follows the structured workflow outline below.

3.1. Data Collection

To model and evaluate the proposed ML-FIFO approach, historical access logs are obtained from content Delivery Network (CDN) workloads and HPC application traces. Data traces from public repositories like IRCache and simulated HPC job schedulers will be sourced. Each log entry includes the timestamps of access, object identifier, access frequency, and cache hit/miss status. The dataset encompasses different traffic scenarios such as sequential streaming, random access, and busty request patterns to reflect realistic HPC operational conditions. Also, logs are segmented into fixed-length observation windows to capture temporal locality while maintaining manageable computational requirements.

This dataset forms the baseline for both model training and performance benchmarking.

3.2. Feature Engineering

Raw access logs are processed to extract predictive features that capture both short-term and long-term access behavior. These features include:

- Recency: Time elapsed since the last access for each cached item.
- Frequency: Number of accessed within the current observation window.
- Inter-arrival Time: Average duration between successive accesses.
- Access Trend: Difference between short-term and long-term frequency, indicating increasing or decreasing popularity.
- Temporal Patterns: Time-of-day and workload-phase indicators to account for cyclical usage patterns in HPC.

All Features are normalized and encoded for compatibility with machine learning models. Missing values are imputed using rolling averages to avoid bias.

3.3. Model Training

Two complementary models are trained to leverage the extracted features:

- Long Short-Term Memory (LSTM) Network: Captures sequential dependencies and long-range temporal correlations in access patterns. Trained using a sliding window sequence approach, the LSTM predicts the probability of a future access within a defined horizon (e.g., next n accesses).
- Random Forest Classifier: Handles non-linear relationships between engineered features and performs binary classification into 'retain' or 'evict'. Feature importance rankings from the RF model are also used to validate the significance of engineered features.

3.4. Training process

Data is split into 80% training and 20% testing sets. Hyperparameters (e.g., LSTM hidden units, RF tree depth) are tuned using cross-validation and the models are evaluated using precision, recall, and F1-score to ensure balanced predictive performance.

3.5. Integration with FIFO

The integration chosen preserves FIFO's O(1) time complexity for most operations while selectively altering eviction order to retain valuable items. The standard FIFO queue is enhanced with a predictive decision layer:

- When the cache is full, the item at the front of the FIFO queue is examined.
- The ML prediction module estimates its future access probability.
- If the predicted probability exceeds a defined threshold (e.g., 0.7), the item is moved to the rear of the queue, delaying its eviction.
- If the probability is below the threshold, the item is evicted following standard FIFO logic.

3.6. Evaluation

ML-FIFO is benchmarked against standard FIFO, Least Recently Used (LRU), and Least Frequently Used (LFU) policies. Evaluation includes statistical significance testing to confirm performance improvements. Performance is evaluated in a controlled simulation environment replicating HPC workload. The following metrics are measured:

- Hit Ratio: Percentage of requests served directly from the cache.
- Miss Ratio: Complement of the hit ratio, indicting cache inefficiency.
- Execution Time: Time taken for cache operations, including any additional overhead from ML inference.
- Throughput: Number of requests processed per second.

3.7. Proposed Methodology

The research proposes a machine-learning enhanced FIFO for predictive prefetching. The method uses machine learning models (long short-term memory (LSTM), random forest) to predict which cached item will be accessed next and FIFO is adjust accordingly with the items to be accessed next, organized at the back of the queue. Hence the

organization allows the FIFO algorithm evict items that are not likely to be accessed at the front of the queue where can easily be evicted by the algorithm.

How it works

- Step 1: The LSTM model is trained on access pattern using data from the previous or past access pattern generated.
- Step 2: Before an item is evicted, the LSTM model predicts the future access probability of the item
- Step 3: Given the result of the prediction in step 2 above, an item with high future access probability gets retained i.e. gets a retention boost instead of an eviction

This will intelligently prevent cache misses by retaining important items longer in the cache. This will be implemented using data from content delivery network (CDN) where viral content must stay in the cache longer. The proposed framework balances FIFO's scalability and simplicity with the predictive intelligence of machine learning, enabling adaptability in highly dynamic HPC environments without imposing excessive computational overhead.

4. Discussions

The study introduced a series of enhancements to the traditional FIFO cache replacement algorithm, each aimed at addressing specific limitations while retaining its inherent simplicity and low computational cost. The proposed modifications which is hybrid FIFO, Adaptive FIFO, and ML-Based FIFO demonstrate how incremental improvements can significantly impact cache performance across diverse application domains. The table below shows the enhancement that were made to the FIFO replacement algorithm.

Table 1 Enhancement that were made to the FIFO replacement algorithm

Proposed Modification	Key change	Benefit	Application area
Hybrid FIFO	Keeps Tracks of the frequency of use of an item before eviction	Reduction in the number of unnecessary removals	Database caching
Adaptive FIFO	Dynamically adjust the retention of items by FIFO, so as to avoid to static organization of FIFO of first come, first serve.	This leads to adaptation to workload changes.	Used in cloud computing
ML-Based FIFO	Prediction of future access of items in the Cache	Generally, enhances cache efficiency	Used in CDNs and AI-driven caching

4.1. Hybrid FIFO

Incorporates frequency tracking alongside the FIFO queue structure. It maintains an access count for each cached item, the algorithm can prevent premature eviction of frequently accessed data. This hybrid approach reduces unnecessary removals and improves data locality, making it particularly beneficial for database caching, where repetitive queries to the same data segments are common. Compared to standard FIFO, this method introduces minimal additional metadata but yields noticeable gains in hit ratio.

4.2. Adaptive FIFO

Addresses the static nature of conventional FIFO by dynamically adjusting item retention times based on workload characteristics. Instead of eviction strictly on a first-in-first-out basis, the algorithm adapts its eviction decisions according to changing access patterns. This adaptability is critical in cloud computing environments, where workloads can shift unpredictably due to multi-tenant resource allocation and elastic scaling. The observed improvement in hit ratio over baseline FIFO confirms that responsiveness to workload variability can significantly enhance overall cache efficiency.

4.3. ML-Based FIFO

Represents the most advanced enhancement, employing predictive modeling to forecast future access probabilities. Harnessing historical access patterns, the machine learning module reorders the eviction queue, prioritizing the

retention of items with high likelihood of reuse. This approach is especially well-suited to content delivery networks (CDNs) and AI-driven caching systems, where predictive intelligence can mitigate cache misses for trending or high-demand content. While this method introduces moderate computational overhead due to model inference, the potential for substantial gains in hit ratio justifies the trade-off in many high-performance applications

Table 2 Benchmark Results from existing algorithms based on parameters such as hit rate, miss rate and execution time

Algorithm	Hit Rate	Miss Rate	Execution Time (s)
Standard FIFO	40.1%	59.9%	0.0023
Adaptive FIFO	55.6% (Better)	44.4%	0.0032
Hybrid FIFO-LFU	64.3% (Much Better)	35.7%	0.0041
LRU (Baseline)	71.2% (Best Performance)	28.8%	0.0055
LFU (Baseline)	69.4%	30.6%	0.0060

4.4. Performance Analysis

The benchmark results (Table 3) highlight the comparative performance of the enhanced FIFO approaches against established algorithms such as LRU and LFU. Standard FIFO recorded a hit rate of 40.1%, significantly lower than LRU's 71.2%, confirming the limitations of a purely sequential eviction policy. The Adaptive FIFO improved the hit rate to 55.6%, indicating that responsiveness to workload dynamics can further increased the hit rate to 64.3%, approaching LFU's 69.4% while maintaining lower execution time. The table below shows the expected performance improvements.

Table 3 Expected Performance Improvements

Algorithm	Hit Rate	Miss Rate	Execution Time
FIFO (Baseline)	40-50%	50-60%	Fast
Adaptive FIFO	55-65%	35-45%	Moderate
Hybrid FIFO-LFU	60-70%	30-40%	Moderate
LRU (Baseline)	70-80%	20-30%	High
ML-Enhanced FIFO (Proposed)	75-85%	15-25%	Moderate (With ML Overhead)

The proposed ML-Enhanced FIFO is projected to achieve hit rates between 75% and 85%, outperforming both LRU and LFU in predictive scenarios. Although its execution time is moderate due to the machine learning overhead, it remains competitive for HPC and CDN workloads where prediction accuracy translates directly into performance and cost savings.

5. Conclusion

These findings suggest that FIFO, often regarded as a simplistic baseline, can be transformed into a high-performance cache replacement policy through targeted enhancements. The results also emphasis the importance of lightning algorithmic improvements with specific application domains. Hybrid FIFO for databases, Adaptive FIFO for cloud platforms, and ML-Enhanced FIFO for intelligent caching in distributed systems. The trade-off between computational overhead and predictive accuracy should guide implementation choices, especially in latency-sensitive environments.

Compliance with ethical standards

Disclosure of conflict of interest

No conflict of interest to be disclosed.

References

[1] W. Stallings, Computer Organization and Architecture: Designing for Performance, Pearson, 2020.

[2] N. I. O. Areej M. Osman, "A Comparison of Cache Replacement Algorithms for Video Services," International Journal of Computer Science and Information Technology, vol. 10, no. 2, pp. 95-111, 2018.

[3] A. E. P. A. F. W. A. Mulki Indana Zulfa, "Performance comparison of cache replacement algorithms on various internet traffic," Jurnal Infotel, vol. 15, no. 1, pp. 1-7, 2023.

[4] N. I. Osman, "Will video caching remain energy efficient in future core optical networks?," digital communications and networks, vol. 3, pp. 39-46, 2017.

[5] A. Sarwan, Cache replacement algorithm, Peshawar, 2016.

[6] J. Yang, Z. Qiu, Y. Zhang, Y. Yue, and K. V. Rashmi, "FIFO can be better than LRU: The power of lazy promotion and quick demotion," in Proc. HotOS '23, 2023.

[7] D. Akbari Bengar, "Priority Cache Object Replacement by Using LRU, LFU and FIFO algorithms to Improve Cache Memory Hit Ratio," Trans. Soft Comput., vol. 1, no. 1, 2025.

[8] H. Mahni, S. Rubini, S. Gougeaud, P. Deniel, and J. Boukhobza, "Multicriteria File-Level Placement Policy for HPC Storage," in Proc. SAC '25, 2025.

[9] S. Sethumurugan, J. Yin, and J. Sartori, "Designing a cost-effective cache replacement policy using machine learning," in Proc. HPCA, 2021, pp. 291–303.

[10] Z. Shi, X. Huang, A. Jain, and C. Lin, "Applying deep learning to the cache replacement problem," in Proc. MICRO '52, 2019, pp. 413–425.

[11] G. Vietri, L. V. Rodriguez, S. Lyons, et al., "Driving cache replacement with ML-based LeCaR," in Proc. HotStorage, 2018.

[12] L. V. Rodriguez, F. Yusuf, S. Lyons, et al., "Learning cache replacement with CACHEUS," in Proc. FAST '21, 2021, pp. 341–354.

[13] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, "An imitation learning approach for cache replacement," in Proc. ICML, 2020, pp. 6237–6247.

[14] H. Choi and S. Park, "Learning future reference patterns for efficient cache replacement decisions," IEEE Access, vol. 10, pp. 25922–25934, 2022.

[15] Y. Zhou, F. Wang, Z. Shi, and D. Feng, "An end-to-end automatic cache replacement policy using deep reinforcement learning," in Proc. ICAPS '22, 2022, pp. 537–545.

[16] A. V. Jamet, L. Alvarez, D. A. Jiménez, and M. Casas, "Characterizing the impact of last-level cache replacement policies on big-data workloads," in Proc. IISWC, 2020, pp. 134–147.

[17] H. Wu, Y. Luo, and C. Li, "Optimization of heat-based cache replacement in edge computing system," J. Supercomput., vol. 77, no. 3, pp. 2268–2301, 2021.

[18] I. C. Agubata, M. Kharrat, O. C. Onyedeke, M. Ezema, and C. N. Okwueze, "Design and Optimization of Bus Booking System using Dijkstra's Algorithm," *International Journal of Science and Business*, vol. 4, no. 12, pp. 21–37, Nov. 2020, doi: 10.5281/zenodo.4232573.