Mantik Journal, 9 (1) (2025), ISSN 2685-4236 (Online)

Published by:Institute of Computer Science (IOCS)

Jurnal Mantik

Journal homepage: www.iocscience.org/ejournal/index.php/mantik

Segmented least recently used cache replacement simulator

Marvin Chandra Wijaya[1], Maria Angela Kartawidjaja[2], Kyle Edmund[3]
[1]Program of Engineering Profession, Atma Jaya Catholic University, Jakarta, Indonesia
[2]Departement of Electrical Engineering, Atma Jaya Catholic University, Jakarta, Indonesia
[3]Departement of Informatics Engineering, Maranatha Christian University, Bandung, Indonesia

A R T I C L E I N F O

Article history:

Received Mar 27, 2025
Revised Apr 03, 2025
Accepted Apr 12, 2025

Keywords:

Block Replacement;
Cache Memory;
Hit Ratio;
SLRU.

ABSTRACT

The block replacement process in the memory cache is an essential technique in computing systems to improve the efficiency of data retrieval from high-speed memory. Various caching algorithms have been developed to speed up data retrieval access in the memory cache, including Least Recently Used (LRU), Least Frequently Used (LFU), and First In First Out (FIFO). This study aims to develop a simulator by combining the LRU and LFU methods called Segmented Least Recently Used (SLRU), which is able to process data retrieval from the memory cache more efficiently. Experiments on the simulation program created were carried out on 10 random data groups to determine the effectiveness of each block replacement algorithm. Based on the test results, SLRU had the best performance, with an average hit ratio of 71.4%, followed by LRU (67%), LFU (62%), and FIFO, which showed the lowest hit ratio performance with a hit ratio of 55.8%. The advantage of SLRU lies in dividing cache segmentation into two segments: the probationary segment (LRU) and the protected segment (LFU). Based on the experiment results, it was concluded that SLRU has more efficient results in handling dynamic data access patterns than other algorithms.

Corresponding Author:

Marvin Chandra Wijaya,
Program of Engineering Profession,
Atma Jaya Catholic University,
Jl. Jendral Sudirman 515, South Jakarta,Jakarta, 12930, Indonesia.
Email: marvin.cw@eng.maranatha.edu

1. INTRODUCTION

Cache replacement is the condition in which the data in the cache needs to be changed to remain within its storage limits. The cache itself acts as temporary storage that promptly provides the information when the system needs to access frequently used data compared to real-time when the system would have to access that data in volatile (such as RAM) or permanent storage (Podlipnig & Böszörmenyi, 2003). If cache space is still available, the new data can be placed in the cache without removing any previously stored data. But, when the caches if full, the system should decide which data should be removed from the cache in order to replace with the new data (Priya, Kumar, Begum, & Ramasubramanian, 2019). Data removal is generally based simply on the frequency of recent use or when the data was last requested/accessed. Once the previous data is

removed, the new required data will be then inserted into the cache for fast access upon the next access cycle. This process is done automatically and continuously as long as the system is operational (Wijaya, 2020). Various types of processor architectures, such as CISC architecture and RISC architecture, require cache memory in the data retrieval process (Bachri, Alexander, Osmond, Widawati, & Kartawidjaja, 2024). Cache replacement is done/applied within computer systems, web storage, databases, and hardware components operating in CPUs and SSDs, especially those that need large cache memory (Yennimar, Faturrahman, Nesen, Guci, & Pasaribu, 2023). The data collection process greatly influences the memory architecture used in local and shared memory, especially in parallel computing (Satria, Barakbah, & Sudarsono, 2021). Efficient cache management will allow the system to operate more efficiently and at less total access time when interaction with relevant data is requested.

Least Recently Used (LRU) is a cache replacement strategy that evicts the data that has not been accessed for the longest period (Xiong & Szefer, 2020). This strategy effectively maintains frequently accessed data in the cache, given that access patterns are similar (Souza & Freitas, 2024). The advantage is enhanced performance and cache efficiency because it retains data still of value to the system, but the disadvantage is increased complexity for implementation due to the need to track the order in which a given element is accessed within the cache (Zheng et al., 2022).

First-In, First-Out (FIFO) replaces the data that first enters the cache, regardless of how often it is used (J. Yang, Zhang, Qiu, Yue, & Vinayak, 2023). Simple to apply is its advantage because it does not require keeping track of how frequently data is used. The disadvantage is the potential of replacing data that is still used frequently, resulting in performance that may not be optimized, especially when older data are still in active use.

Least Frequently Used (LFU) eliminates the least recently used data based on the number of accesses (Alzakari, Dris, & Alahmadi, 2020). Its strength lies in its optimality in terms of retaining the most frequently used data, but its weakness is significant as well, with the potential to retain data that was frequently used in the past but is no longer relevant in the near future.

Random Replacement (RR) causally deletes data based on randomly selected data with no regard to patterns of access (Unterluggauer, Harris, Constable, Liu, & Rozas, 2022). A main advantage to this method is in its simplicity and ability to avoid the additional processing cost of tracking data usage history. However, it is a large downside of this method, as it does not consider the relevance of the data and can lead to the removal of useful data, thereby decreasing the efficiency of the cache storage system.

In block replacement, several important parameters must be considered, including the level of cache associativity, replacement policies such as LRU, FIFO, LFU, or Random, as well as the age factor and frequency of block usage (Kumar & Singh, 2016). In addition, dirty bits affect whether data must be written back to memory before block replacement, depending on the write-through or write-back policy. The cache miss rate is also an important factor because it affects the efficiency of data access, which can be optimized by considering temporal and spatial locality when selecting blocks to be replaced (Asiatici & Ienne, 2019).

Several areas require improvement to enhance the current block replacement philosophy. More precise predictions can be achieved by utilizing machine learning or AI-enhanced replacement policies, predicting data access patterns more effectively than classical methods like LRU or LFU (Q. Yang et al., 2023). Improved algorithms that adapt to application usage patterns enhance efficiency (Krishna, 2025). LRU and LFU need to reduce overhead and complexity because it will burden the memory and speed of the computer due to the required tracking access history. Furthermore, reducing cache miss rates is possible by combining replacement policies with smarter prefetching to ensure

necessary data is available in the cache before it is required. Temporal locality-aware replacement strategies help retain data that is likely to be used soon (Sonia et al., 2021).

Another key improvement is the effective management of dirty bits, which will reduce the number of unintended writebacks to main memory by using dirty bit flushing optimizations (such as flushing a block only if it was modified enough times). In particular, hybrid write-through or write-back algorithms can be configured for workloads (Young, Chishti, & Qureshi, 2019). As computing has progressed, requests for variants of the replacement mechanism will exist to make them more efficient for multi-core and GPU architectures, specifically as they relate to multi-threaded execution with shared caches. Reducing power consumption is equally important, especially for mobile devices and embedded systems, in which offering policies that account for energy for replacement can save energy, maintain cache effectiveness and latency, and refresh the main memory storage to conserve battery life (Sethumurugan, Yin, & Sartori, 2021). In large data centers, these enhancements will lower operational costs (Khan et al., 2021).

Compared to other cache replacement techniques, the SLRU algorithm has several benefits. First, it outperforms rules like FIFO and regular LRU regarding the cache hit ratio (Hasslinger, Ntougias, Hasslinger, & Hohlfeld, 2023). Second, as simulations like Icarus show, it minimizes latency and lowers connection load. Third, it is appropriate for operating systems caching file blocks because of its segmented structure, which enables it to adjust to various usage patterns. Because of these characteristics, the SLRU algorithm is a unique option among contemporary cache policies.

Two major gaps emerge in the research on Segmented Least Recently Used (SLRU): first, it has been evaluated on only a small quotient of diverse, real-world workloads; second, SLRU has never been evaluated and compared against more adaptive algorithms. While SLRU has been evaluated in a contrived setting, it has not been thoroughly explored with active, dynamic, and unpredictable workloads prevalent in a modern context, including, but certainly not limited to, cloud computing or edge devices. Furthermore, SLRU has never been adequately evaluated and compared against newer adaptive policies, such as ARC or CAR that are adaptable in response to workload changes. Closing the aforementioned gaps in research would provide some meaningful insight into SLRU's practical performance in a much more meaningful way, in addition to revealing potential defensible succinct advantages (or disadvantages).

By filling in these research gaps, the results of this study will benefit the world of practice by providing system designers and engineers with more information to make educated decisions during their evaluation of cache replacement policies, whether they simply be adopting or applying them in a real-world setting. By assessing SLRU under varying and realistic workloads, this investigation contributes insight into SLRU's performance in real-world settings, such as cloud infrastructure, web services, or edge computing. Such performance comparisons could improve the efficacy of memory resources or system performance.

2. RESEARCH METHOD

The Segmented Least Recently Used (SLRU) method is a segmented cache replacement of cache memory policy that aims to improve cache data management. It improves performance by prioritizing frequently used data and reducing less-used elements. This method boosts the cache hit rate, resulting in faster access to vital information. Parting the cache into segments effectively distinguishes frequently accessed material from rare items. There are some strategies to increase efficiency and better use existing memory resources.

SLRU method improves cache performance by separating it into two parts: probationary and protected. Data used frequently moves to the protected area, preventing unnecessary data loss. The size of the shielded portion can be adjusted to meet the system requirements. The SLRU algorithm monitors data usage via reference bits, allowing vital data to be stored for longer periods. The SLRU algorithm is suitable for various applications, including operating systems and databases.

The probationary and protected segments are the two primary segments into which the SLRU algorithm divides the cache. The probationary phase is always where new entries begin. The protected section prioritizes frequently used data, where an entry goes if it is reaccessed. Due to its structure, the cache will adjust to both short-term and long-term access patterns. When the cache is full, the program removes the least recently utilized material from the probationary section. This method increases overall efficiency and reduces needless cache eviction.

The SLRU algorithm uses reference bits to monitor the frequency of data access. An entry's reference bit changes each time you visit it, indicating its importance. Less-used entries stay in the probationary section, while entries with more reference activity are moved to the protected segment. Due to this dynamic management, the cache prioritizes data according to real-time usage patterns. The SLRU algorithm compromises between allowing for new entries and keeping frequently used data by utilizing reference bits.

By splitting the cache into two segments, the page replacement process known as SLRU (Segmented Least Recently Used) improves on the conventional LRU (Least Recently Used) technique (as shown in Figure 1): (a) Pages that are often accessed are stored in the protected segment. (b) Newly uploaded pages or just removed from the protected segment are stored in the probationary segment.

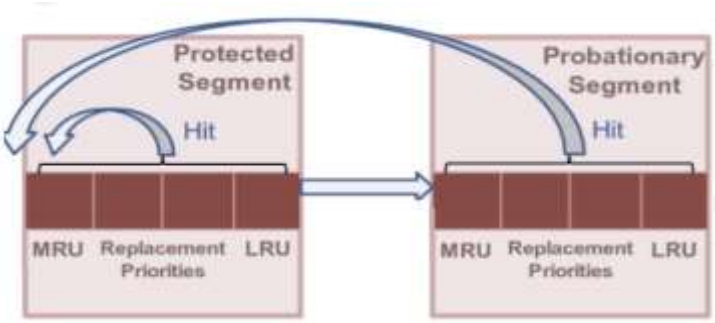

Figure 1. Protected Segment and Probationary Segment (Yamaki, 2019).

Figure 2 shows the four steps of the SLRU algorithm procedure. The CPU will give instructions on how to retrieve data from memory. The data to be retrieved will be checked to see whether it is in the memory cache. If it is not in the memory cache, then the page will be retrieved and inserted into the probationary segment. Pages that are frequently accessed will be inserted into the protected segment. Because the protected segment has limited space, pages that are already in the protected segment but are least used will be returned to the probationary segment.

Figure 2. SLRU Algorithm

Flowchart for Segmented Least Recently Used Simulator algorithm as follows:

1. Start
2. Access a Key
    a. Check if the key is in the Protected Segment (LFU)
        i. If Yes, increase its frequency
    b. If No, check if it's in the Probationary Segment (LRU)
        i. If Yes, Promote it to the Protected Segment
        ii. If No, add it to the Probationary Segment
3. Handle Overflows.
4. If the probationary segment is full, evict the least recently used item (LRU).
5. If the Protected Segment is full, evict the Least Frequently Used (LFU) item.
6. Update Cache & Log the Action
7. Display & Visualize Cache

Validity in this research was addressed through specific strategies that ensured that the SLRU algorithm was executed faithfully according to established theoretical frameworks and previous work. This included careful modeling of various algorithm structures (such as dividing the algorithm into probationary and protected lists to determine the use of reference bits and others) according to specification. The simulator was designed to provide behaviors that replicated real-world cache operations, including the access pattern associated with memory usage, indicative of a real system. Reliability was supported by treating simulations with consistent input parameters, workload types, and simulation system configurations. In particular, multiple simulations were conducted under the same conditions in order to ensure that the results (such as cache hit rate and eviction) were reliable and thus replicable.

3. RESULTS AND DISCUSSIONS

The experiment will be conducted by simulating a group of data for cache memory as follows:

a. Cache Access Simulation,

The code executes several access requests based on the listrequests= ["A", "B", "C", "A", "D", "A", "E", "B", "F", "B", "C", "A"]. Each time an item is accessed, the code checks for its existence in the cache and determines whether it needs to be promoted or replaced.

b. Logging Cache Activity

Every cache change is logged in the slru_log.txt file. The log will contain information such as: (a) Item being accessed, (b) Promotion from Probationary to Protected Segment, (c) Item removal if the cache is full

c. Displaying Cache Contents After Each Access

The cache is displayed after each new access. The Probationary Segment contains newly added items, while the Protected Segment contains frequently accessed items.

d. Visualizing Access Frequency in Protected Segment

The code will display a bar chart using Matplotlib, showing how many times each item in the Protected Segment has been accessed. If there are no items in the Protected Segment, the message will appear: " There is no data in the Protected Segment to visualize."

The cache is used as an initial space to temporarily store information needed in CPU operations (Panda, Patil, & Raveendran, 2016). When the CPU requests information stored in memory and the information is already in the cache, it is also called a cache hit (or Hit Ratio), as in equation (1) (Ma, Hao, Shen, Tian, & Al-Rodhaan, 2018). If a cache hit occurs, the CPU can immediately receive information from data from the cache memory. However, if the information is not in the cache, it is called a cache miss (or Miss Ratio), as in equation (2). When a cache miss occurs, the CPU is given data from the main memory. Average Memory Access Time (AMAT) is a computer memory system performance measurement for memory processes. Average Memory Access Time (AMAT) measures the average time required to access data from memory, involving factors such as cache hit rate and main memory access time as in equation (3) (Pedro-Zapater, Rodríguez, Segarra, Gran Tejero, & Viñals-Yúfera, 2020).

$$Hit\ Ratio = \frac{Hits}{Hits + Miss} \times 100\% \quad (1)$$

$$Miss\ Ratio = \frac{Misses}{Hits + Misses} \times 100\% \quad (2)$$

$$Average\ Memory\ Access\ Time\ (AMAT) = Hit\ Times + Miss\ Ratio \times Miss\ Penalty \quad (3)$$

Figure 3 and Figure 4 are the results of cache replacement simulation using the SLRU algorithm designed in this study. The next experiment uses 10 different data groups to compare the cache hit ratio and the miss ratio of the SLRU cache replacement method designed in this study compared to the LFU, LRU, and FIFO methods.

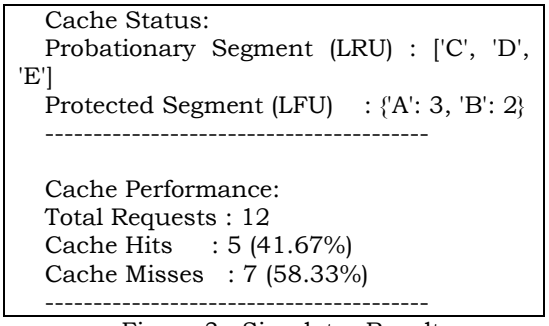

Figure 3. Simulator Results

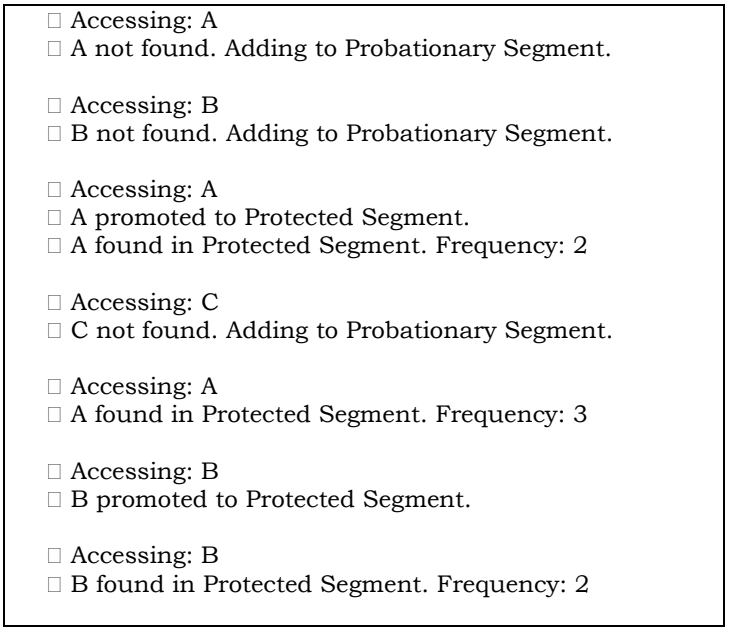

Figure 4. Log file

Based on the simulation results on 10 groups of random data, the comparison results of the hit ratio and miss ratio show that SLRU (Segmented Least Recently Used) has the best performance compared to the LRU (Least Recently Used), LFU (Least Frequently Used), and FIFO (First In First Out) methods. The SLRU simulation results have an average success ratio of around 71.4% and a failure ratio of 28.6%, higher than other algorithms, as shown in Table 1 and Figure 5.

The LRU method, which only maintains the most recently accessed elements, shows a hit ratio of around 67%, which is slightly lower than that of SLRU. LFU is a method that stores elements based on access frequency. Based on the test results, it shows lower results with an average hit ratio of around 62%. The FIFO method has the worst performance compared to other methods, with a hit ratio of only around 55.8%.

Table 1. Cache Hit Ratio and Miss Ratio Comparison

Data Group	SLRU (Hit%)	SLRU (Miss%)	LRU (Hit%)	LRU (Miss%)	LFU (Hit%)	LFU (Miss%)	FIFO (Hit%)	FIFO (Miss%)
1	72.00%	28.00%	68.00%	32.00%	63.00%	37.00%	58.00%	42.00%
2	70.00%	30.00%	66.00%	34.00%	60.00%	40.00%	55.00%	45.00%
3	74.00%	26.00%	69.00%	31.00%	64.00%	36.00%	57.00%	43.00%
4	71.00%	29.00%	67.00%	33.00%	61.00%	39.00%	54.00%	46.00%

Data Group	SLRU (Hit%)	SLRU (Miss%)	LRU (Hit%)	LRU (Miss%)	LFU (Hit%)	LFU (Miss%)	FIFO (Hit%)	FIFO (Miss%)
5	73.00%	27.00%	68.00%	32.00%	65.00%	35.00%	59.00%	41.00%
6	69.00%	31.00%	65.00%	35.00%	60.00%	40.00%	53.00%	47.00%
7	75.00%	25.00%	70.00%	30.00%	66.00%	34.00%	60.00%	40.00%
8	68.00%	32.00%	64.00%	36.00%	59.00%	41.00%	52.00%	48.00%
9	72.00%	28.00%	67.00%	33.00%	62.00%	38.00%	56.00%	44.00%
10	70.00%	30.00%	66.00%	34.00%	60.00%	40.00%	54.00%	46.00%
Average	71.40%	28.60%	67.00%	33.00%	62.00%	38.00%	55.80%	44.20%

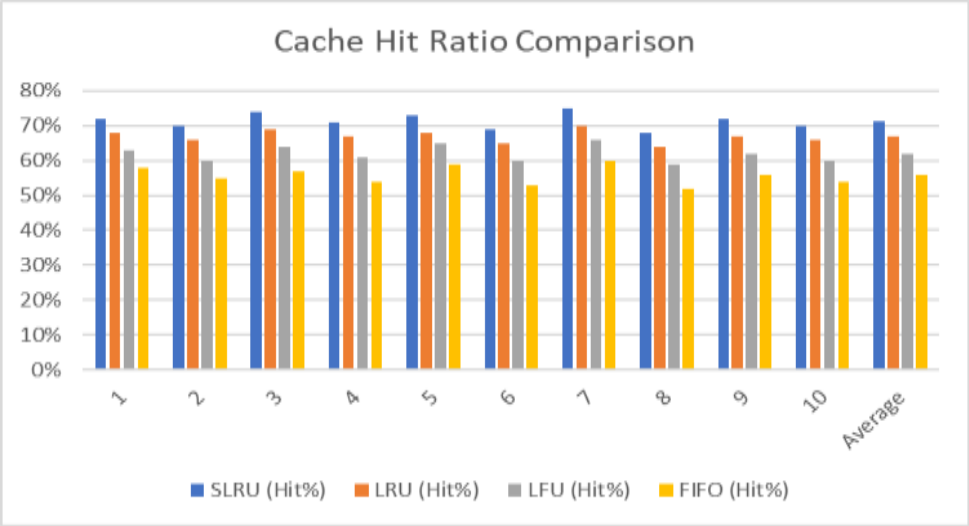

Figure 5. Cache Hit Ratio Comparison

4. CONCLUSION

Overall, SLRU has advantages compared to other methods because it is able to separate the cache into two segments, namely probationary (LRU) and protected (LFU). Due to the division of these two segments, frequently accessed data will be promoted from the probationary segment to the protected segment. Pages in the protection segment will remain in the cache longer. These two segments produce a more optimal combination of the LRU and LFU methods. This explains why SLRU performed the best in this test.

Based on the test results, it can be concluded that the SLRU method is more effective in handling various data access patterns compared to LRU, LFU, and FIFO. SLRU can be a more efficient choice for implementing block replacement in various needs such as database systems, memory management, or proxy servers, and this is because it will significantly reduce the number of cache misses and increase the speed of the data retrieval process to memory. Based on the experimental results, the average hit ratio using the LRU method was 71.40%, which is better than the LRU method (67%), LFU (62%) and FIFO method (55.8%).

A significant drawback is the fixed sizes of segments, which may not be optimal for all workloads. An additional consideration is that the structure of the SLRU algorithm is relatively complex compared with traditional replacement techniques, such as FIFO and LRU, requiring overhead that may not be feasible in low-resource systems. Also, SLRU functions relatively statically and does not have the potential to adapt to changes in workload access patterns. Accordingly, it is likely to function well under organized

conditions, but the performance of SLRU may vary in real-time or dynamically changing workload conditions.

Further improvements to cache replacement decisions could also be achieved through predictive or machine-learning techniques to model access patterns. In addition, testing SLRU on realistic datasets and exploring SLRU at the hardware level in practice will help provide further insight into the advantages and disadvantages of using SLRU. Furthermore, future investigations may address efficiency-related investigations of these techniques, particularly energy efficiency for use in mobile or embedded systems where consumption is important.

REFERENCES:

Alzakari, N., Dris, A. Bin, & Alahmadi, S. (2020). Randomized Least Frequently Used Cache Replacement Strategy for Named Data Networking. *2020 3rd International Conference on Computer Applications & Information Security (ICCAIS)*, 1–6. https://doi.org/10.1109/ICCAIS48893.2020.9096733

Asiatici, M., & Ienne, P. (2019). Stop Crying Over Your Cache Miss Rate: Handling Efficiently Thousands of Outstanding Misses in FPGAs. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 310–319. https://doi.org/10.1145/3289602.3293901

Bachri, K. O., Alexander, J., Osmond, E., Widawati, E., & Kartawidjaja, M. A. (2024). SPArc-subset general-purpose microprocessor design and implementation in field programmable gate array. *Mantik*, *8*(3), 1447–1455. https://doi.org/10.35335/mantik.v8i3.5681

Hasslinger, G., Ntougias, K., Hasslinger, F., & Hohlfeld, O. (2023). Scope and Accuracy of Analytic and Approximate Results for FIFO, Clock-Based and LRU Caching Performance. *Future Internet*, *15*(3). https://doi.org/10.3390/fi15030091

Khan, T. A., Zhang, D., Sriraman, A., Devietti, J., Pokam, G., Litz, H., & Kasikci, B. (2021). Ripple: Profile-Guided Instruction Cache Replacement for Data Center Applications. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 734–747. https://doi.org/10.1109/ISCA52012.2021.00063

Krishna, K. (2025). Advancements in cache management: a review of machine learning innovations for enhanced performance and security. *Frontiers in Artificial Intelligence*, *8*, 1441250. https://doi.org/10.3389/frai.2025.1441250

Kumar, S., & Singh, P. K. (2016). An overview of modern cache memory and performance analysis of replacement policies. *2016 IEEE International Conference on Engineering and Technology (ICETECH)*, 210–214. https://doi.org/10.1109/ICETECH.2016.7569243

Ma, T., Hao, Y., Shen, W., Tian, Y., & Al-Rodhaan, M. (2018). An Improved Web Cache Replacement Algorithm Based on Weighting and Cost. *IEEE Access*, *6*, 27010–27017. https://doi.org/10.1109/ACCESS.2018.2829142

Panda, P., Patil, G., & Raveendran, B. (2016). A survey on replacement strategies in cache memory for embedded systems. *2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, 12–17. https://doi.org/10.1109/DISCOVER.2016.7806218

Pedro-Zapater, A., Rodríguez, C., Segarra, J., Gran Tejero, R., & Viñals-Yúfera, V. (2020). Ideal and Predictable Hit Ratio for Matrix Transposition in Data Caches. *Mathematics*, *8*(2). https://doi.org/10.3390/math8020184

Podlipnig, S., & Böszörmenyi, L. (2003). A survey of Web cache replacement strategies. *ACM Comput. Surv.*, *35*(4), 374–398. https://doi.org/10.1145/954339.954341

Priya, B. K., Kumar, S., Begum, B. S., & Ramasubramanian, N. (2019). Cache lifetime enhancement technique using hybrid cache-replacement-policy. *Microelectronics Reliability*, *97*, 1–15. https://doi.org/https://doi.org/10.1016/j.microrel.2019.03.011

Satria, B. D., Barakbah, A. R., & Sudarsono, A. (2021). Implementation Parallel Computation for Automatic Clustering. *Mantik*, *5*(2), 994–1005. https://doi.org/10.35335/jurnalmantik.Vol5.2021.1439.pp994-1005

Sethumurugan, S., Yin, J., & Sartori, J. (2021). Designing a Cost-Effective Cache Replacement Policy using Machine Learning. *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 291–303. https://doi.org/10.1109/HPCA51647.2021.00033

Sonia, Alsharef, A., Jain, P., Arora, M., Zahra, S. R., & Gupta, G. (2021). Cache Memory: An

Analysis on Performance Issues. *2021 8th International Conference on Computing for Sustainable Global Development (INDIACom)*, 184–188.

Souza, M. A., & Freitas, H. C. (2024). Reinforcement Learning-Based Cache Replacement Policies for Multicore Processors. *IEEE Access*, *12*, 79177–79188. https://doi.org/10.1109/ACCESS.2024.3409228

Unterluggauer, T., Harris, A., Constable, S., Liu, F., & Rozas, C. (2022). Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks. *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, 13–24. https://doi.org/10.1109/SEED55351.2022.00009

Wijaya, M. C. (2020). Algoritme penggantian cache proxy terdistribusi untuk meningkatkan kinerja server web. *Jurnal Teknologi Dan Sistem Komputer*, *8*(1), 1–5. https://doi.org/10.14710/jtsiskom.8.1.2020.1-5

Xiong, W., & Szefer, J. (2020). Leaking Information Through Cache LRU States. *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 139–152. https://doi.org/10.1109/HPCA47549.2020.00021

Yamaki, H. (2019). Flow Characteristic-Aware Cache Replacement Policy for Packet Processing Cache BT - Advances in Information and Communication Networks. *Advances in Intelligent Systems and Computing*, *886*, 258–273. https://doi.org/10.1007/978-3-030-03402-3_18

Yang, J., Zhang, Y., Qiu, Z., Yue, Y., & Vinayak, R. (2023). FIFO queues are all you need for cache eviction. *Proceedings of the 29th Symposium on Operating Systems Principles*, 130–149. https://doi.org/10.1145/3600006.3613147

Yang, Q., Jin, R., Fan, N., Inupakutika, D., Davis, B., & Zhao, M. (2023). *AdaCache: A Disaggregated Cache System with Adaptive Block Size for Cloud Block Storage*. Retrieved from https://arxiv.org/abs/2306.17254

Yennimar, Y., Faturrahman, M. R., Nesen, S., Guci, M. A., & Pasaribu, S. R. (2023). Implementation of artificial neural network and support vector machine algorithm on student graduation prediction model on time. *Mantik*, *7*(2), 925–934. https://doi.org/10.35335/mantik.v7i2.3992

Young, V., Chishti, Z. A., & Qureshi, M. K. (2019). TicToc: Enabling Bandwidth-Efficient DRAM Caching for Both Hits and Misses in Hybrid Memory Systems. *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 341–349. https://doi.org/10.1109/ICCD46524.2019.00055

Zheng, Q., Yang, T., Kan, Y., Tan, X., Yang, J., & Jiang, X. (2022). On the Analysis of Cache Invalidation With LRU Replacement. *IEEE Transactions on Parallel and Distributed Systems*, *33*(3), 654–666. https://doi.org/10.1109/TPDS.2021.3098459