International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

Improving Cache Hits On Replacment Blocks Using Weighted LRU-LFU Combinations

Marvin Chandra Wijaya
Departement of Computer Engineering
Maranatha Christian University
Bandung, Indonesia
marvin.cw@eng.maranatha.edu

Abstract— Block replacement refers to the process of selecting a block of data or a cache line to be evicted or replaced when a new block needs to be brought into a cache or a memory hierarchy. In computer systems, block replacement policies are used in caching mechanisms, such as in CPU caches or disk caches, to determine which blocks are evicted when the cache is full and new data needs to be fetched. The combination of LRU (Least Recently Used) and LFU (Least Frequently Used) in a weighted manner is known as the "LFU2" algorithm. LFU2 is an enhanced caching algorithm that aims to leverage the benefits of both LRU and LFU by considering both recency and frequency of item access. In LFU2, each item in the cache is associated with two counters: the usage counter and the recency counter. The usage counter tracks the frequency of item access, while the recency counter tracks the recency of item access. These counters are used to calculate a combined weight for each item in the cache. Based on the experimental results, the LRU-LFU combination method succeeded in increasing cache hits from 94.8% on LFU and 95.5% on LFU to 96.6%.

Keywords-Block Replacment; LRU; LFU; LFU2

I. INTRODUCTION

Cache memory is a small, high-speed memory component used in computer systems to improve overall system performance. It acts as a buffer between the CPU (central processing unit) and the main memory (RAM), storing frequently accessed data and instructions to reduce the time it takes for the CPU to retrieve information [1]. The primary purpose of cache memory is to bridge the speed gap between the fast CPU and the relatively slower main memory. CPUs can access cache memory much faster than accessing data from RAM, which helps to alleviate the performance bottleneck caused by the speed disparity.

Cache memory operates based on the principle of locality of reference, which states that programs tend to access a relatively small portion of data and instructions repeatedly in a short period of time [2]. The cache exploits this principle by storing copies of frequently accessed data from the main memory. When the CPU needs to read or write data, it first checks the cache [3]. If the data is present in the cache (a cache hit), the CPU retrieves it directly from there, resulting in faster access time. If the data is not in the cache (a cache miss), the CPU has to fetch it from the main memory, and a copy of the data may be stored in the cache for future use [4].

Cache memory is organized into different levels, typically referred to as L1, L2, and L3 caches. L1 cache is the closest and fastest cache to the CPU, followed by L2 and L3 caches, which are larger but slower. The different cache levels are designed to store progressively larger amounts of data but with increasing access latency. The size of cache memory is relatively small compared to main memory, as larger caches would be more expensive to implement. Cache effectiveness depends on factors like cache size, cache replacement policies, and the access patterns of the running programs. By utilizing cache memory, the overall performance of a computer system can be significantly improved by reducing the average time it takes for the CPU to access data.

Block replacement, also known as cache replacement policy, is a mechanism used in cache memory to determine which data should be evicted from the cache when a new data block needs to be loaded [5]. When the cache is full and a cache miss occurs (the data being accessed is not present in the cache), a block replacement policy decides which cache block to replace with the new block [6]. Block replacement is important in cache memory because it directly affects the cache's effectiveness in improving system performance [7].

The primary goal of cache memory is to increase the speed at which the CPU can access frequently used data and instructions. When a cache hit occurs (the requested data is found in the cache), the CPU can retrieve the data much faster compared to accessing it from the main memory. By choosing an effective block replacement policy, the cache can maximize the probability of cache hits, thus increasing overall system performance.

Programs tend to exhibit locality of reference, meaning they access a relatively small portion of data and instructions repeatedly in a short period of time. The block replacement policy should take advantage of this behavior by keeping the most frequently accessed data in the cache. By doing so, the

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

cache can satisfy a larger portion of the CPU's data requests and reduce the need to fetch data from slower main memory. Cache memory is typically limited in size due to cost and implementation constraints. As a result, cache space needs to be utilized efficiently. The block replacement policy determines which data blocks should be evicted when new blocks need to be loaded. An effective policy ensures that the cache stores the most useful and frequently accessed data, avoiding unnecessary evictions of data that may be needed in the near future. Workloads executed by a computer system can vary over time, with different programs and data patterns causing fluctuations in data access patterns.

A good block replacement policy can adapt to these changing workloads by dynamically evicting less useful or less frequently accessed data and replacing it with more relevant data. This adaptability helps maintain a high cache hit (lower cache miss) rate and improves performance under varying conditions [8].

II. LITERATURE REVIEWS

Block replacement is important in cache memory because it directly influences cache hit rate [9], helps exploit locality of reference, optimizes cache space usage, and enables adaptability to dynamic workloads. By making informed decisions about which data to keep in the cache, the system can maximize the benefits of cache memory and enhance overall performance.

There are several block replacement policies commonly used in cache systems.

- Random Replacement: This policy selects a random cache block to be replaced. It does not consider any information about the accessed data or the cache contents. While simple to implement, it may not always make optimal decisions in terms of cache hit rate.
- Least Recently Used (LRU) Replacement: LRU replacement policy evicts the cache block that has been least recently used. It maintains a timestamp or a reference bit for each cache block and updates it whenever a block is accessed. When a replacement is needed, the block with the oldest timestamp or the least recently accessed block is chosen for eviction [10], [11].
- First-In, First-Out (FIFO) Replacement: In the FIFO replacement policy, the cache block that was first loaded into the cache is replaced. It uses a queue structure to track the order in which blocks were loaded, and the block at the front of the queue is evicted when necessary [12].
- Least Frequently Used (LFU) Replacement: LFU replacement policy tracks the frequency of cache block accesses. Each block has a counter associated with it, which is incremented each time the block is accessed. The block with the lowest access frequency is replaced when replacement is required. This policy aims to keep frequently accessed blocks in the cache.
- Most Recently Used (MRU) Replacement: MRU replacement policy evicts the most recently used cache block. It assumes that the block that was accessed most recently will likely be accessed again soon. This policy is useful in certain scenarios where temporal locality is high enough.

Jaafar Alghazo in 2004 made a study entitled “SF-LRU Cache Replacment Algorithm” which is a modification of the Least Recently Used and Least Frequently Used methods [13]. The approach is put up against the LRU and LFU algorithms in a thorough comparison. The SF-LRU greatly lowers the number of cache misses when compared to the other two algorithms, according to experimental results. According to the findings of the simulation, our technique can, at most, improve the miss ratio in the data cache by 6.3% and the instruction cache by 9.3% when compared to the LRU algorithm. Figure 1 is the SF_LRU operating system designed by Jaafar.

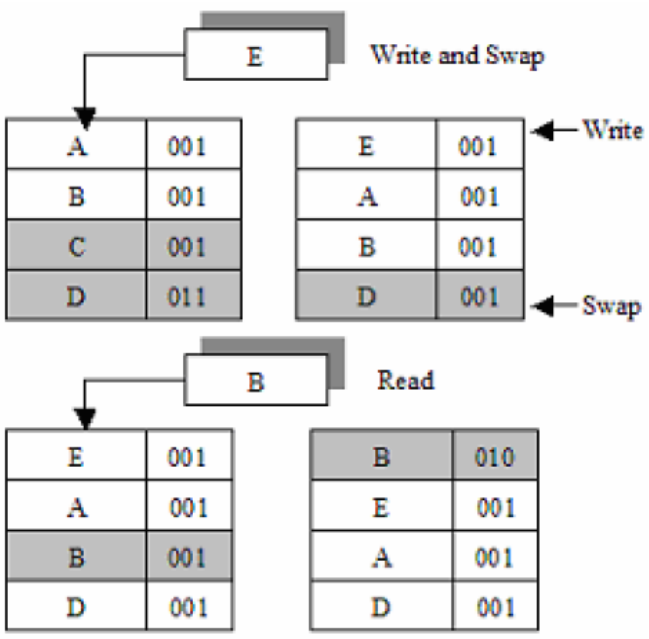

Figure 1. SF-LRU operation by Jaafar Alghazo [13]

In 2004, G.E. Suh studied entitled "Dynamic Partitioning of Shared Cache Memory" which combines information and partitions cache time [14]. Marginal gain counters, a partitioning mechanism, and an OS controller make up the partitioning system. In order to calculate the marginal benefits of running processes, the system uses a set of counters in the beginning. A method in the cache that can truly control the allocation to each process is secondly required by the system. Finally, the operating system establishes the cache allocation and chooses the optimum partition based on data from counters. A CMP simulator based on the SimpleScalar tool set has the partitioning technique implemented in the shared L2 cache. The simulation findings demonstrate that, over a range of cache sizes for specific

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

processes, partitioning can significantly outperform the traditional LRU replacement policy in terms of cache performance. Additionally, for a variety of cache sizes, the partitioning approach can resolve the issue of process interference in caches. However, if the caches are too small for the workloads, partitioning alone won't help with performance. Therefore, it is important to choose concurrent processes wisely, taking into account their memory reference behavior. Figure 2 is the implementation of block replacement (modified LRU) designed by G. E. Suh.

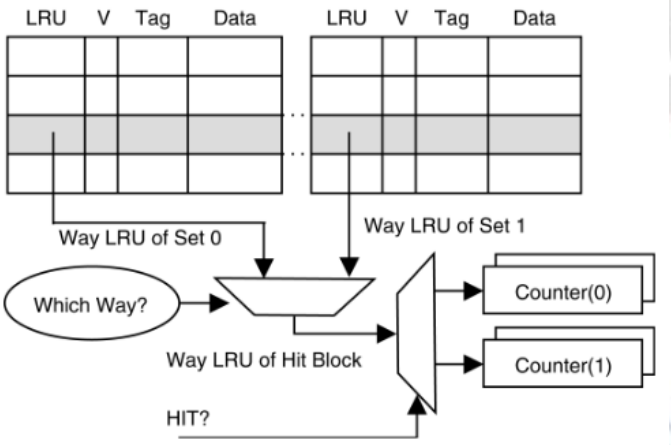

Figure 2. Implementation of block replacement by G.E. Suh [14]

In 2019, Davood Akbari Bengar studied entitled "A page replacement algorithm based on a fuzzy approach to improve cache memory performance," which aims to improve cache memory performance. This research proposes a page replacement algorithm that is simple to implement. The algorithm, which uses three parameters to cluster cache pages, is called the fuzzy page replacement algorithm, as shown in Figure 3. Whenever a miss occurs, it selects a page of the cluster with the lowest priority which has the smallest Euclidean distance with its center and then exits the cache. The most significant advantage of the algorithm is using the FCM (fuzzy c-means) algorithm to cluster pages, resulting in better replacement and hence higher memory performance.

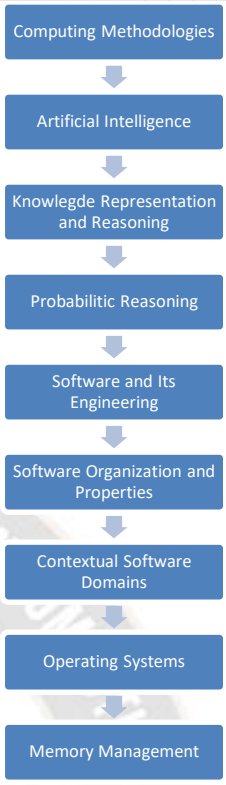

Figure 3. Page Replacement implementation by Davood Akbari Bengar [15]

In 2011, Abu Asaduzzaman studied entitled "An efficient memory block selection strategy to improve the performance of cache memory subsystem" which aims to improve the performance of the cache memory subsystem. This study suggests a simple but effective memory block selection technique to improve cache locking and cache replacement execution, as well as the overall performance of the cache memory subsystem. The suggested technique identifies the blocks that, if unlocked, result in the most cache misses and stores the block address and miss information (BAMI) at the cache level. Memory blocks with larger cache misses should be locked using the cache locking technique, while blocks with lesser cache misses should be chosen as victims by the cache replacement strategy. To assess the suggested block selection strategy, we simulate single-core and multi-core systems with two-level cache memory subsystems. According to experimental findings, using the memory block selection scheme can increase the hit ratio by up to 11% while reducing overall power usage by up to 20% [16].

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

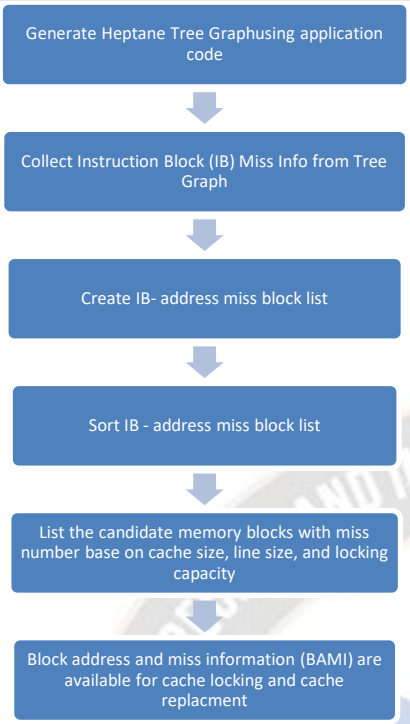

Figure 4. Workflow diagram of proposed block selection strategy by Abu Asaduzzaman [16]

In 2016, Somayeh Sardashti studied "Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache" to increase cache memory's adequate capacity. The Yet Another Compressed Cache (YACC), a brand-new compressed cache design that aims to increase useful cache capacity with a straightforward design, is what the authors propose. While packing variable-size compressed blocks to reduce internal fragmentation, YACC uses super-blocks to lower tag overheads. Decoupled Compressed Cache (DCC) and Skewed Compressed Cache (SCC), two cutting-edge compressed caches, are utilized by YACC to achieve its goals. The structure of the YACC cache is comparable to that of traditional caches, with a virtually unaltered tag array and unaltered data array. In contrast to DCC and SCC, YACC does not require the substantial additional metadata that DCC needs to track blocks or the complexity and costs of skewed associativity that SCC does. In addition, YACC supports contemporary replacement techniques like RRIP, which gives it an edge over earlier research. According to experiment, YACC enhances performance by an average of 8% and as much as 26% when compared to a traditional uncompressed 8MB LLC and decreases overall energy by an average of 6% and as much as 20%. With only 1.6% more area than an 8MB conventional cache, an 8MB YACC offers roughly the same performance and energy improvements as a 16MB conventional cache in a substantially smaller silicon footprint. Similar to DCC and SCC in performance, YACC is significantly easier to deploy [17].

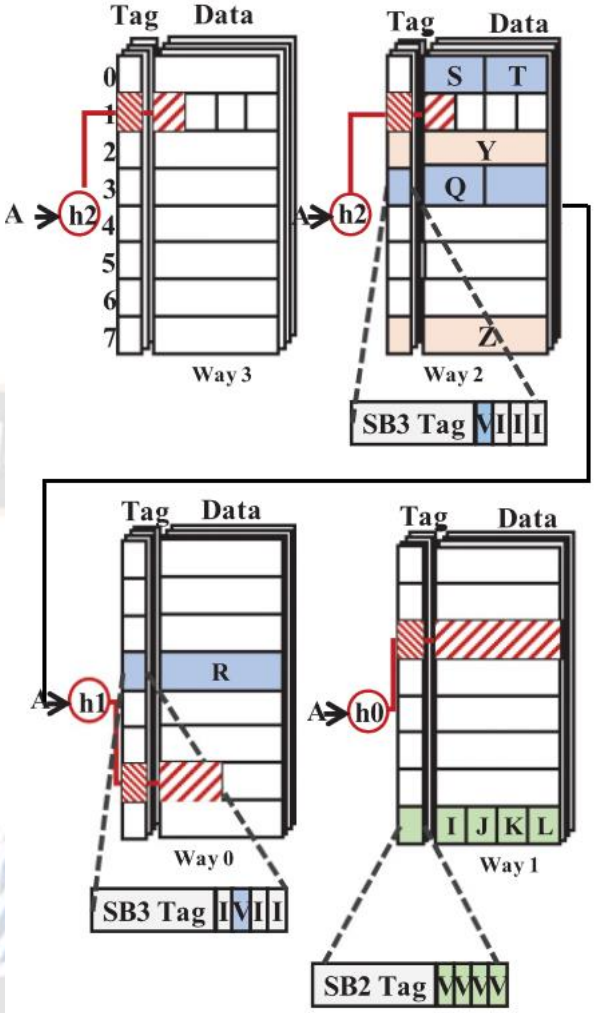

Figure 5. Skewed Compressed Cache by Somayeh Sardashti [17]

Srikanthaiah Hiremath and Mahmoud A Manzoul studied " An Improved Fuzzy Replacment Algorithm For Cche Memories "to improve cache memory replacement performance. This study suggested a modified fuzzy replacement technique (MFRA), which significantly outperforms the FRA in terms of cache memory performance. The FRA has undergone the following significant modifications: the number of inputs has decreased from three to two; the number of fuzzy linguistic variables and the dimensions of the universes of discourse have increased; and a larger knowledge base that employs 34 new rules is used. Under actual workload situations, the performance of the MFRA is comparable to that of conventional replacement algorithms like least recently used (LRU) and first in first out (FIFO), as seen in the results [18].

In 2006, Sedigheh Khajoueinejad studied "A Fuzzy Cache Replacement Policy and Its Experimental Performance Assessment" to improve access performance between computing units and memory. One of the best ways to increase the speed of 10 accesses between computational units and memories is to use caching. Which objects are removed from the cache to create way for new objects is determined by the cache

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
***Article Received:** 26 August 2023 **Revised:** 08 October 2023 **Accepted:** 20 October 2023*

replacement policy. In this study, we provide a novel fuzzy technique termed fuzzy page replacement (FPR). Through simulation, this algorithm's performance is evaluated in comparison to well-known cache replacement algorithms. The simulation findings demonstrate that the fuzzy technique outperforms the other algorithms and may be considered in subsequent studies [19].

In 2019, Bhukya Krishna Priya studied "Cache lifetime enhancement technique using hybrid cache-replacement-policy" to reduce the error rate when cache replacement occurs. By lessening the influence of the replacement algorithm, the proposed Hybrid-Cache-Replacement (HCR) strategy enhances write endurance while lowering mistake rates. The incoming data is properly inserted into the already-existing block, which has a low error rate and fewest writes. It has been put into practice by comparing the incoming bits to the bits that are already present in a cache set. According to the simulation results, as compared to the current approaches, the STT-RAM caches' lifespan increases by 135%, 165%, and 27%, with a 2%, 2%, and 1% performance overhead [20].

In 2022, Swapnita Srivastava studied entitled "Proof of Optimality based on Greedy Algorithm for Offline Cache Replacement Algorithm" which aims to optimize cache replacement. An easier proof based on the greedy algorithm is presented in this article. In order to show that the greedy solution is delivering optimality, the study shows that any ideal solution can be repeatedly turned into the solution offered by the greedy algorithm without increasing the miss of the optimal solution. The Greedy Weight-based Cache Replacement Algorithm (GWCRA), which is based on the Greedy algorithm and also adds the weighted access-based parameters of recency and frequency, is another replacement algorithm that is presented in this work. When compared to LRU and SRRIP, which have average speedups of 55.58% and 55.65%, respectively, the GWCRA obtains a speedup of 57.29% [21].

III. METHOD

The method proposed in this study is a combination of weighted LRU and LFU. The LRU algorithm works on the principle that items that have been accessed most recently are likely to be accessed again in the near future, while items that have not been accessed for a long time are less likely to be accessed again soon. In the context of caching, the LRU algorithm keeps track of the order in which items are accessed. When the cache reaches its maximum capacity and a new item needs to be added, the algorithm removes the least recently used item from the cache to make space for the new item. On subsequent accesses, if an item is already present in the cache, it is marked as the most recently used item, moving it to the front of the cache. The LRU algorithm helps optimize cache performance by maximizing the utilization of cache space for frequently accessed items and minimizing cache misses for less frequently accessed items. LRU maximizes the utilization of cache space by evicting the least recently used items. This ensures that the most frequently accessed items are kept in the cache, reducing cache misses and improving overall system performance. LRU takes advantage of the principle of temporal locality, which states that recently accessed items are likely to be accessed again in the near future. By keeping the most recently used items in the cache, LRU improves the hit rate and reduces the time taken to retrieve data from the cache. The LRU algorithm is relatively simple to implement compared to other caching algorithms. It requires tracking the order of item access and updating it accordingly. The straightforward nature of LRU makes it a popular choice for cache management. LRU assumes that future access patterns will be similar to past access patterns. However, in some cases, access patterns may change over time, and LRU may not be able to adapt quickly. For example, if there is a sudden shift in the popularity of certain items, LRU may continue to keep less popular items in the cache while evicting more popular items. LRU does not perform well in the presence of skewed or uneven access patterns. If a few items are accessed very frequently while the rest of the items are rarely accessed, LRU may repeatedly evict and re-fetch the less frequently accessed items, leading to a high cache miss rate and decreased performance. While LRU is relatively simple to implement for small cache sizes, it becomes more complex and computationally expensive as the cache size increases. Maintaining the order of accessed items and updating it in a large cache can require additional memory and overhead, impacting the overall efficiency of the algorithm.

In contrast to the LRU algorithm, which focuses on the recency of item access, LFU prioritizes items that have been accessed the least number of times. In the LFU algorithm, each item in the cache is associated with a usage counter that keeps track of the number of times the item has been accessed. When the cache reaches its maximum capacity and a new item needs to be added, the algorithm selects the item with the lowest usage counter for eviction. LFU identifies items that are accessed infrequently and removes them from the cache. This approach aims to optimize cache performance by keeping frequently accessed items in the cache and reducing cache pollution caused by rarely used items. LFU requires tracking the usage counters for each item in the cache. When an item is accessed, its counter is incremented. This process ensures that the algorithm accurately reflects the frequency of item usage. LFU is capable of adapting to changes in access patterns. If an item that was previously accessed infrequently suddenly becomes popular, its usage counter will increase, making it less likely to be evicted. This adaptiveness can be beneficial in scenarios where access patterns vary over time. LFU heavily relies on accurate tracking of item usage. If an item is accessed frequently initially and then becomes less popular, it may remain in the cache longer than desired. Similarly, an item that starts with a low usage count but

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

suddenly becomes popular may face eviction before its popularity is recognized. Maintaining usage counters for each item in the cache requires additional memory and computational resources. As the cache size or the number of items increases, the overhead associated with tracking and updating the counters can become significant. LFU performs well when access patterns are consistent or slowly changing. However, if there are sudden bursts of activity or intermittent access to certain items, LFU may not effectively adapt to these patterns, potentially leading to suboptimal cache performance.

The combination of weighted LRU and LFU can optimize the benefits of each block replacement method and reduce the disadvantages of each block replacement method. This study proposes a method of combining weighted LRU and LFU as shown in Figure 6. Main memory is the main memory storage area that will replace a block of cache memory according to the combined weight of LRU and LFU.

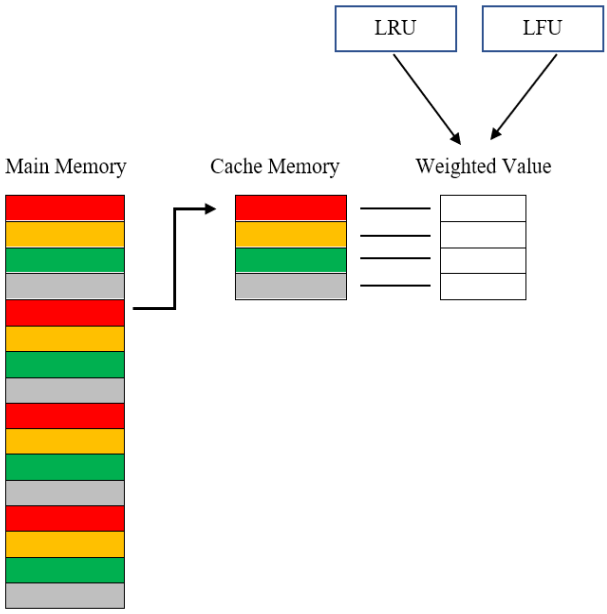

Figure 6. Combination of Weighted LRU and LFU

To retrieve data or instructions from a program, the locality of reference idea is used. The CPU only ever uses a small amount of the address space at once. Performance of the cache memory is calculated using the page fault rate, miss penalty, and average access time. The percentage of memory accesses that are Page Fault Rate are those that are not cached, while The hit rate or hit ratio of the cache refers to the proportion of accesses that return a cache hit. Miss Penalty is the sum of the Cycles (time) to replace a block in the higher level cache and the Cycles (time) to deliver the block to the processor. It is the total number of cycles the CPU is stopped for a memory access. The equation (1), (2), and (3) are used to calculate average access time and CPU execution time.

$$AAT = (HT \times HR) + (MP \times MR) \quad (1)$$

$$CET = (CCC + MSC) \times CCT \quad (2)$$

$$MSC = NM \times MP \quad (3)$$
$$= IC \times (M / I) \times MP$$
$$= IC \times ((MA / I) \times MR \times MP$$

Where :
AAT =Average Access Time
HT = Hit Time
HR = Hit Rate
MP = Miss Penalty
MR = Miss Rate
CET = CPU Execution Time
CCC = CPU Clock Cycle
MSC = Memory Stall Cycles
CCT = Clock Cycle Time
NM = Number of Misses
IC = Instruction Cyle
M = Missed
I = Instruction
MA = Memory Access

Memory read and write cycle counts may differ in a similar manner. Reading penalties may differ from writing penalties as (4).

$$MSCC = MRSC + MWSC \quad (4)$$

Where:

MSCC = Memory Stall Clock Cycle
MRSC = Memory Read Stall Cycles
MWSC = Memory Write Stall Cycles

IV. RESULTS

Experiments were carried out to test the cache hit rate of the proposed method using simulation [22]. This simulation program will simulate various states of a set of instructions and data. Testing experiments by executing vector calculations with vector -vector in a pseudocode as follows.

```
main
{
   read(n)
   for(i=1,i<=n,i++)
   {
      read(x[n])
      read(y[n])
```

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

```
    }
   for(i=1,i<=n,i++)
   {
       Z[n]  = x[n] + y[n]
    }
   for(i=1,i<=n,i++)
   {
       write(z[n])
    }
}
```

Data collection was carried out with different amounts of n. Data collection was simulated in three different methods (LRU, LFU and combination LRU - LFU weighted). Table 1 is the result of data collection when using the LRU method. From Figure 7 it can be seen that the cache hits will decrease if the amount of data (n) increases.

TABLE I. LRU EXPERIMENT

No.	n	Cache hit (%)
1	*1000*	*98*
2	2000	98
3	3000	97
4	4000	97
5	5000	96
6	6000	95
7	7000	94
8	8000	93
9	9000	91
10	10000	89

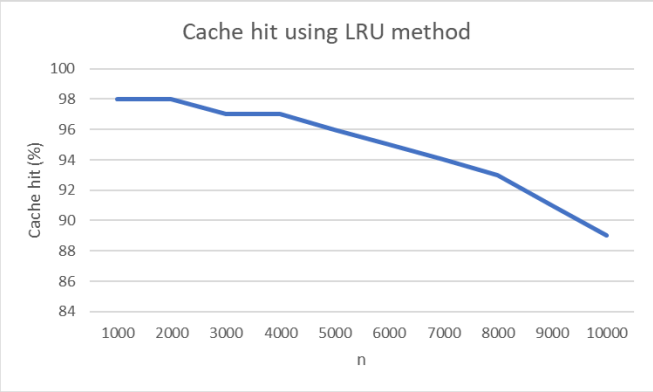

Figure 7. Cache hit using LRU method

Table 2 is the result of data collection when using the LFU method. From Figure 8 it can be seen that the cache hits will decrease if the amount of data (n) increases.

TABLE II. LFU EXPERIMENT

No.	n	Cache hit (%)
1	*1000*	*98*
2	2000	98
3	3000	97
4	4000	97
5	5000	96
6	6000	96
7	7000	95
8	8000	94
9	9000	93
10	10000	91

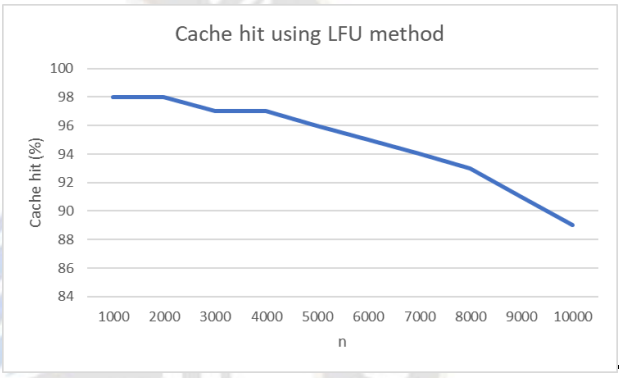

Figure 8. Cache hit using LFU method

Table 3 is the result of data collection when using the LFU method. From Figure 9 it can be seen that the cache hits will decrease if the amount of data (n) increases.

TABLE III. COMBINATION LRU - LFU WEIGHTED EXPERIMENT

No.	n	Cache hit (%)
1	*1000*	*99*
2	2000	99
3	3000	98
4	4000	98
5	5000	97
6	6000	97
7	7000	96
8	8000	95
9	9000	94
10	10000	93

International Journal on Recent and Innovation Trends in Computing and Communication
ISSN: 2321-8169 Volume: 11 Issue: 10
Article Received: *26 August 2023* ***Revised:*** *08 October 2023* ***Accepted:*** *20 October 2023*

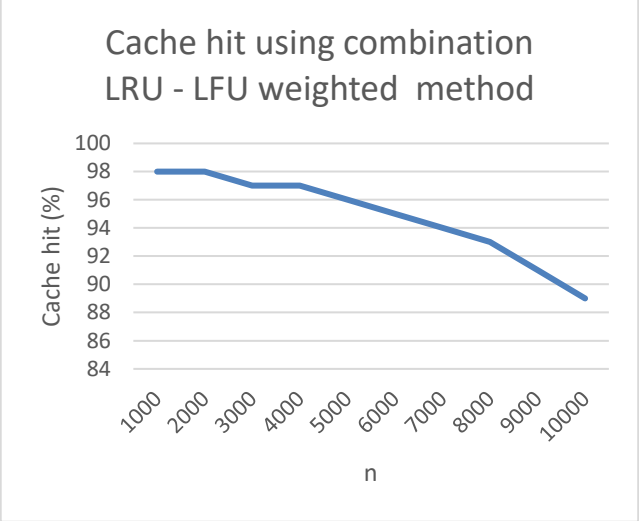

Figure 9. Cache hit using combination LRU - LFU weighted method

Based on the experimental results, the average cache hit using the LRU method is 94.8%, the LFU method is 95.5% and the weighted LRU-LFU combination method is 96.6%.

V. CONCLUSIONS

When the cache reaches its maximum capacity and an eviction is necessary, LFU2 selects the item with the lowest combined weight for eviction. The combined weight is typically calculated as a weighted sum of the usage and recency counters, with different weights assigned to each counter based on their relative importance. By combining the recency and frequency aspects of LRU and LFU, LFU2 aims to strike a balance between favoring recently accessed items and favoring frequently accessed items. This approach allows LFU2 to adapt to changing access patterns, giving more weight to recently accessed items that may become popular while still considering the overall frequency of item access. The specific weightings assigned to the recency and usage counters can vary based on the specific implementation and tuning requirements. Different weighting schemes can be used to give more emphasis to recency, frequency, or a combination of both, depending on the characteristics of the workload and the desired caching behavior. Based on the experimental results, the LRU-LFU combination method succeeded in increasing cache hits from 94.8% on LFU and 95.5% on LFU to 96.6%.

ACKNOWLEDGMENT

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression, “One of us (R.B.G.) thanks . . .” Instead, try “R.B.G. thanks”. Put applicable sponsor acknowledgments here; DO NOT place them on the first page of your paper or as a footnote.

REFERENCES

[1] M. Kowarschik and C. Weiß, “An Overview of Cache Optimization Techniques and Cache-Aware Numerical Algorithms BT - Algorithms for Memory Hierarchies: Advanced Lectures,” in Algorithm for Memory Hierarchies, U. Meyer, P. Sanders, and J. Sibeyn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 213–232.

[2] M. C. Wijaya, “Distributed proxy cache replacement algorithm to improve web server performance,” Jurnal Teknologi dan Sistem Komputer, vol. 8, no. 1, pp. 1–5, 2020.

[3] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi, “Using Dead Blocks as a Virtual Victim Cache,” in Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, 2010, pp. 489–500.

[4] M. W. Ahmed and M. A. Shah, “Cache Memory: An Analysis on Optimization Techniques,” International Journal of Computer and Information Technology, vol. 4, no. 2, pp. 414–418, 2015.

[5] H. B. Jang, A. Kashif, M.-S. Park, and S. W. Chung, “Reliable Cache Memory Design for Sensor Networks,” in 2008 Third International Conference on Convergence and Hybrid Information Technology, 2008, vol. 1, pp. 651–656.

[6] A.-C. Lai, C. Fide, and B. Falsafi, “Dead-Block Prediction & Dead-Block Correlating Prefetchers,” in Proceedings of the 28th Annual International Symposium on Computer Architecture, 2001, pp. 144–154.

[7] P. Panda, G. Patil, and B. Raveendran, “A survey on replacement strategies in cache memory for embedded systems,” in 2016 IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER), 2016, pp. 12–17.

[8] S. Kumar and P. K. Singh, “An overview of modern cache memory and performance analysis of replacement policies,” in 2016 IEEE International Conference on Engineering and Technology (ICETECH), 2016, pp. 210–214.

[9] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, “A Case for MLP-Aware Cache Replacement,” in Proceedings of the 33rd Annual International Symposium on Computer Architecture, 2006, pp. 167–178.

[10] Y. Nagasako and S. Yamaguchi, “A Server Cache Size Aware Cache Replacement Algorithm for Block Level Network Storage,” in 2011 Tenth International Symposium on Autonomous Decentralized Systems, 2011, pp. 573–576.

[11] E. Cheshmikhani, H. Farbeh, S. G. Miremadi, and H. Asadi, “TA-LRW: A Replacement Policy for Error Rate Reduction in STT-MRAM Caches,” IEEE Transactions on Computers, vol. 68, no. 3, pp. 455–470, 2019.

[12] R. Hassan, A. Harris, N. Topham, and A. Efthymiou, “Synthetic Trace-Driven Simulation of Cache Memory,” in 21st International Conference on Advanced Information Networking and Applications Workshops (AINAW’07), 2007, vol. 1, pp. 764–771.

[13] J. Alghazo, A. Akaaboune, and N. Botros, “SF-LRU cache replacement algorithm,” in Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004., 2004, pp. 19–24.

[14] G. E. Suh, L. Rudolph, and S. Devadas, “Dynamic

Partitioning of Shared Cache Memory," The Journal of Supercomputing, vol. 28, no. 1, pp. 7–26, 2004.

[15] D. Akbari Bengar, A. Ebrahimnejad, H. Motameni, and M. Golsorkhtabaramiri, "A page replacement algorithm based on a fuzzy approach to improve cache memory performance," Soft Computing, vol. 24, no. 2, pp. 955–963, 2020.

[16] A. Asaduzzaman, "An efficient memory block selection strategy to improve the performance of cache memory subsystem," in 14th International Conference on Computer and Information Technology (ICCIT 2011), 2011, pp. 12–17.

[17] S. Sardashti, A. Seznec, and D. A. Wood, "Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache," ACM Trans. Archit. Code Optim., vol. 13, no. 3, pp. 1–25, Sep. 2016.

[18] S. HIREMATH and M. A. MANZOUL, "AN IMPROVED FUZZY REPLACEMENT ALGORITHM FOR CACHE MEMORIES," Cybernetics and Systems, vol. 24, no. 4, pp. 325–339, Jan. 1993.

[19] S. Khajoueinejad, M. Sabeghi, and A. Sadeghzadeh, "A Fuzzy Cache Replacement Policy and Its Experimental Performance Assessment," in 2006 Innovations in Information Technology, 2006, pp. 1–5.

[20] B. K. Priya, S. Kumar, B. S. Begum, and N. Ramasubramanian, "Cache lifetime enhancement technique using hybrid cache-replacement-policy," Microelectronics Reliability, vol. 97, pp. 1–15, 2019.

[21] S. Srivastava and P. K. Singh, "Proof of Optimality based on Greedy Algorithm for Offline Cache Replacement Algorithm," International Journal of Next-Generation Computing, vol. 13, no. 3, 2022.

[22] M. Grigoriadou, M. Toula, and E. Kanidis, "Design and evaluation of a cache memory simulation program," in Proceedings 3rd IEEE International Conference on Advanced Technologies, 2003, pp. 170–174.