University of Central Florida

Department of Computer Science

CDA 5106: Spring 2024

Machine Problem 1: Cache Design, Memory Hierarchy Design

Enhancement of Cache Performance using Adaptive Replacement Policy

by

GROUP 12

Honor Pledge: "I have neither given nor received unauthorized aid on this test or assignment."

Student's electronic signature: Manasa Sai Karanam

Student's electronic signature: Anirudh Sai Eriki

Student's electronic signature: Raja Rajeswari Porumalla

Student's electronic signature: Manya Achanta

Student's electronic signature: Likitha Santhapalli

Student's electronic signature: Rama Meghana Pokala

ENHANCEMENT OF CACHE PERFORMANCE USING ADAPTIVE REPLACEMENT POLICY

*

Manasa Sai Karanam
University of Central Florida
5608350

Likitha Santhapalli
University Of Central Florida
5601460

Raja Rajeshwari Porumalla
University of Central Florida
5609059

Manya Achanta
University of Central Florida
5608986

Anirudh Sai Eriki
University of Central Florida
5604960

Rama Meghana Pokala
University of Central Florida
5602080

***Abstract*—As the field of computer architecture rapidly evolves, optimizing and updating technology becomes crucial for maintaining a competitive edge. This paper introduces a new cache replacement algorithm, Logical Cache Segmentation (LCS), aimed at enhancing the performance of cache memory by adapting dynamically to varying data access patterns. Initially, we have developed a specialized cache design and memory hierarchy simulator, utilizing a subset of the SPEC-2000 benchmark suite. The simulator allows for adjustable cache parameters such as size, associativity, and block size and includes a variety of replacement policies such as LRU, FIFO, and Optimal, along with write-back and write-allocate policies. The LCS algorithm is implemented using counter values in range of 0-15 and Regions of Likeliness (ROL) to manage data eviction more adaptively compared to traditional methods. To evaluate the performance of the Logical Cache Segmentation (LCS) algorithm, we compared it with traditional replacement policies such as Least Recently Used (LRU) and First In First Out (FIFO). Key performance metrics used in the comparison include Average Access Time (AAT), L1 Cache Miss Rate, and L1 Hit Rate. The LCS algorithm's dynamic adjustment of eviction likelihood results in significant improvements in cache management, leading to enhanced overall system performance.**

***Index Terms*—cache memory; cache replacement algorithms; logical cache segmentation; memory hierarchy simulator; computer architecture.**

I. Introduction

Caches are critical in modern computing for their role in enhancing processor performance by reducing memory access times. Cache Replacement Policies are essential in optimizing cache performance by ensuring frequently accessed data is stored efficiently.

The success of a cache depends on its replacement policy, which determines how it manages data to minimize the need to access slower main memory. While widely used policies like Least Recently Used (LRU) work well under stable data conditions, they struggle with fluctuating data, often requiring hardware support. The First in First Out (FIFO)) is less effective due to Belady's Anomaly and although Optimal Policy is best among other policies it is difficult to implement as it requires, the farthest element to replace the data.

In this paper, to address these above limitations, we propose a new replacement algorithm called Logical Cache Segmentation (LCS). In this algorithm, data is replaced based on counter values ranging from 0 to 15 and Region of Likeliness(ROL). Regions of Likeliness (ROL) is divided into four regions which include Most Likely to be replaced(MLR) which consists of counter values 0 and 1, Likely to be replaced with counter values ranging from 2 to 5, Less Likely to be replaced(LLR) which includes of counter values ranging from 6 to 11 and Least Likely to be replaced (LELR) with counter values ranging from 12 to 15.

The lower counter values in LCS indicate a higher chance of replacement, ensuring data with lower counters remain in the cache longer. On the other hand, higher counter values indicate a lower chance of being replaced. We have evaluated the performance of LCS with LRU and FIFO using performance metrics such as Hit rate, Miss rate, and Average Access Time.

II. Background

In the literature, researchers have explored various cache replacement techniques. Kharbutli and Solihin pioneered a counter-based method effective for longer data retention but struggled with variable patterns [1]. Reineke and Grund studied the impact of the initial hardware state on replacement algorithms in [2], while Odule and Osingua proposed runtime techniques for adapting to data variations [3].

Additionally, the Protected Round Robin by Tseng et al. reduced miss rates compared to the basic Round Robin [4]. Kedzierski et al. introduced adaptive cache partitioning with the Pseudo Least Recently Used Algorithm, improving quality of service [5]. Ghasemzadeh et al. proposed Modified Pseudo LRU, reducing cache misses [6].

Moving forward, our focus is on Counter Based Replacement with ROLs, aiming to further enhance cache performance
.

III. Design

As was previously said, because the current replacement techniques—LRU and FIFO—are not dynamically optimized to different shifting workload patterns, they perform sub parly for multiple applications. Therefore, we're putting forth a method that calculates the likelihood that the victim's data will be replaced by using a logical 0-15 range counter and four exclusive counter value zones, or ROLs (Region of Likeliness). From here on, our algorithm will be referred to as LCS, and the four counter zones will be called ROLs. In essence, the data inside pre-set counter values make up the ROLs. They calculate the probability that data will be removed from the cache and replaced with fresh data. A given set of data's likelihood of being removed from the cache decreases as its ROL increases, and vice versa. The amount of hits and misses in an operation determines how dynamically the ROLs increase and drop. The four ROLs and the related counter values are displayed in Figure 1. The ROLs are organized in increasing order of retention in the cache, as suggested by their names, and have matching counter values from 0 to 15. The primary stages of our replacement policy are Cache Initialization, Replacement Logic, Cache Entry Mechanism, Cache Priority Adjustment. The next sections will explain these stages. Figure 2 shows how our strategy operates as a flowchart.

Abbreviation	Region Of Likeliness	Counter Value
LeLR	Least Likely to be Replaced	12-15
LLR	Less Likely to be Replaced	6-11
LR	Likely to be Replaced	2-5
MR	Most Likely to be Replaced	0-1

Fig. 1. ROL ABBREVIATION AND COUNTER VALUES

IV. Implementation

A. Cache Initialization

During a cold start, the cache memory is devoid of any stored data, initiating with an empty state. All data designated for cache storage begins with a counter value of "5" within the ROL-LR region until the cache reaches its capacity. This process ensures a systematic allocation of resources within the cache, optimizing storage efficiency.

B. Replacement Logic

When a cache miss occurs, the data piece is retrieved from secondary memory and placed in the cache memory. To accommodate the new element in the cache, either the cache must be empty or a suitable victim must be selected and removed if the cache is full. Our suggested approach utilizes ROLs and 0-15 counter values for this purpose.

An element with a higher counter value is assigned to a higher ROL, reducing the likelihood of it being selected as the victim since it's more frequently accessed. Replacement occurs in two scenarios:

1) When all elements in the cache have the same counter value, the oldest unused element with that counter value becomes the victim.
2) For subsequent replacements, the victim is the element with the lowest counter value or in the lowest ROL. This applies only to items sharing a lower counter value.

C. Cache Entry Mechanism

Upon successful replacement, a candidate from secondary memory fills the position of the final victim in the cache. With the cache now empty after the eviction of the last victim, the newly inserted data element receives a counter value of '5' and is assigned to the ROL - LR region. This process mirrors the initial cache phase when the cache is devoid of any data.

D. Cache Priority Adjustment

In the event of a hit operation, the existing data element in the cache receives the highest counter value within its respective ROL, and the ROL is incremented to the next higher level. For example, if an element with a counter value of '5' in the ROL - LR region registers a hit, its ROL is elevated to the LLR area, and the counter value is set to "11," representing the highest counter value within the LLR ROL region.

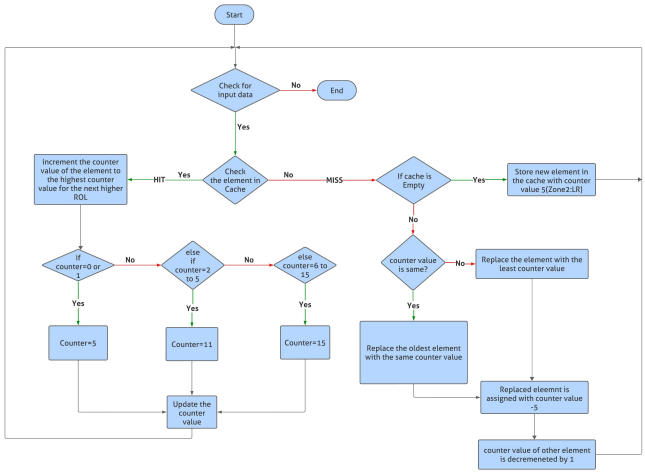

Fig. 2. Flowchart method.

E. Counter Value Constraints

To ensure the integrity of the logical 0-15 counter values, constraints are imposed to confine them within the designated range. During each modification iteration, a boundary condition is enforced to prevent the counter value from decreasing below '0' or exceeding '15'.

F. Performance Analysis of LCS Algorithm

In order to gain deeper insights into the operational dynamics of the LCS algorithm, a specific data sequence is employed: "....57925631757345673."The data sequence is utilized to demonstrate the performance of the LCS algorithm in terms of hits. Assuming a cache capacity of 4 blocks at any given time, input data is represented in the left column, while hits are denoted by 'H' and misses by 'M' in the right column.

Initially, the cache is empty. As elements are inputted into the cache, each is assigned an initial counter value of '5' along with its corresponding Region of Likeliness (ROL). Figure 3 illustrates the execution of the sequence using the LCS algorithm.

Observing the sequence, it becomes evident that elements '5' and '7' are frequently accessed. According to the algorithm's logic, these elements are retained in the cache due to their recurring references. As demonstrated in Figure 3, these elements persist in the cache for an extended duration across multiple iterations.

INPUT	CACHE CONTENT				HIT /MISS
5	5 (5)	-	-	-	M
7	5(5)	7(5)	-	-	M
9	5(5)	7(5)	9(5)	-	M
2	5(5)	7(5)	9(5)	2(5)	M
5	5(11)	7(5)	9(5)	2(5)	H
6	5(10)	6(5)	9(4)	2(4)	M
3	5(9)	6(4)	3(5)	2(3)	M
1	5(8)	6(3)	3(4)	1(5)	M
7	5(7)	7(5)	3(3)	1(4)	M
5	5(15)	7(5)	3(3)	1(4)	H
7	5(15)	7(11)	3(3)	1(4)	H
3	5(15)	7(11)	3(11)	1(4)	H
4	5(14)	7(10)	3(10)	4(5)	M
6	5(13)	7(9)	3(9)	6(5)	M
5	5(15)	7(9)	3(9)	6(5)	H
7	5(15)	7(15)	3(9)	6(5)	H
3	5(15)	7(15)	3(15)	6(5)	H

Fig. 3. LCS Implementation Example.

The LCS algorithm adapts dynamically, adjusting the cache contents based on the frequency of access. Upon complete execution of the sequence, the LCS algorithm yields a total of 7 hits.

V. Evaluation

A. Experimental Set-Up

In our experimental setup, we maintained several parameters constant while varying the cache replacement algorithms (LCS, LRU, and FIFO) and the input trace files. The stable parameters included a block size of 16 bytes, an L1 cache size of 1024 bytes, and an L1 cache associativity of 4. By keeping these parameters consistent across all simulations, we aimed to create a controlled environment for evaluating the performance of different cache algorithms under comparable conditions. This approach allowed us to isolate the impact of the cache replacement strategy and the nature of the workload on cache performance metrics such as hit rate, miss rate, and average access time. By systematically varying the input trace files representing diverse application workloads, ranging from Vortex and Compress to go , gcc and Perl, we could assess the adaptability and effectiveness of each cache algorithm across a spectrum of real-world scenarios. This experimental setup provided valuable insights into the relative strengths and weaknesses of the LCS algorithm compared to traditional approaches like LRU and FIFO, offering valuable guidance for cache design and optimization in computer systems.

B. Comparative Analysis of Cache Replacement Algorithms: LCS, LRU, and FIFO

1) Miss Rate: Miss Rate in the context of cache performance analysis refers to the ratio of cache misses to the total number of memory accesses. It provides insight into how efficiently a cache is utilized, with a lower miss rate indicating better cache performance. A high miss rate suggests that a significant portion of memory accesses result in cache misses, leading to increased latency and reduced overall performance

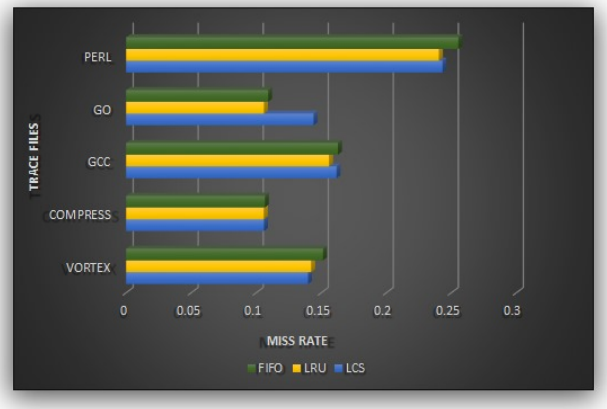

Fig. 4. Comparison against Miss Rate

Figure 4 depicts the results of 15 simulations conducted on a cache simulator, focusing on the comparison of the LCS algorithm with LRU and FIFO in terms of Miss Rate. The simulations highlight that LCS performed particularly well for Vortex and Compress workloads. In these scenarios, LCS exhibited superior performance compared to both LRU and FIFO. Moreover, for the Pearl workload, LCS performed comparably to LRU. In the case of the gcc trace files, LCS showed slightly better performance than FIFO. These findings underscore the effectiveness of the LCS algorithm in managing cache performance across various workloads.

2) Average Access Time: Average Access Time (AAT) is a metric used to evaluate cache performance, representing the average time taken to access data in the cache. It takes into account both cache hits and cache misses, providing a comprehensive measure of the efficiency of the cache system. A lower AAT indicates faster access times and better cache performance overall.

Figure 5 illustrates the outcomes of 15 simulations executed on a cache simulator, focusing on comparing the performance of the LCS algorithm with that of LRU and FIFO in terms of AAT. The simulations reveal that LCS demonstrated particularly robust performance for workloads such as Vortex and

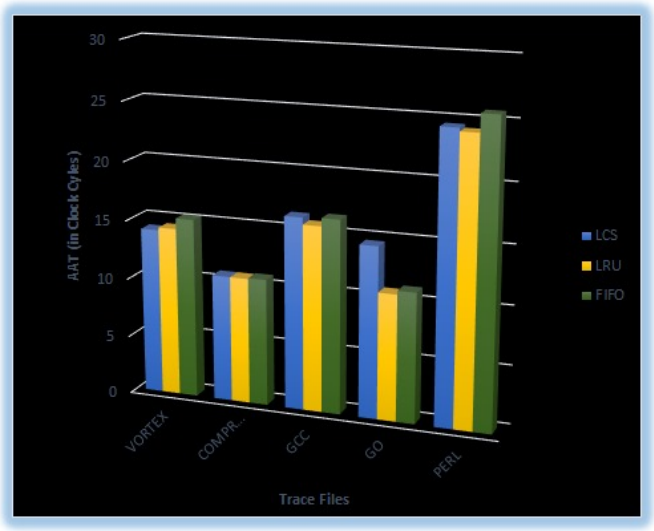

Fig. 5. Comparison against AAT

Compress. In these scenarios, LCS outperformed both LRU and FIFO. Additionally, for the Pearl workload, LCS performed comparably to LRU. When considering the gcc trace files, LCS exhibited slightly superior performance compared to FIFO. These observations underscore the efficacy of the LCS algorithm in optimizing cache performance across diverse workloads.

3) Hit Rate: Hit Rate is a key metric used to assess the effectiveness of a cache algorithm, representing the ratio of cache hits to the total number of memory accesses. A higher hit rate indicates better cache utilization and improved performance, as it signifies that a larger portion of memory accesses are satisfied by the cache.

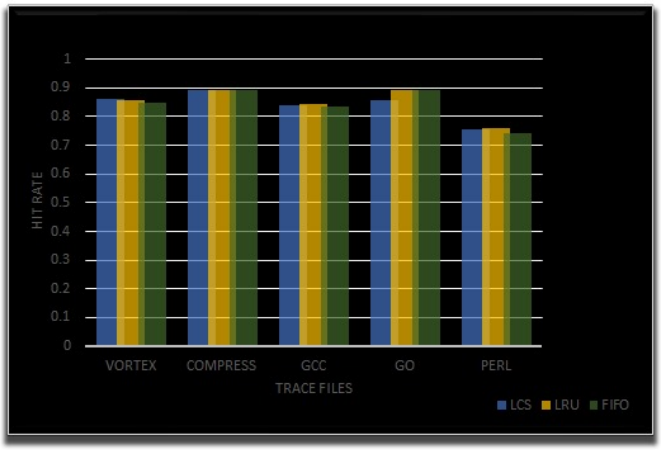

Fig. 6. Comparison against Hit Rate

Figure 6 displays the results of 15 simulations conducted on a cache simulator, focusing on comparing the Hit Rate performance of the LCS algorithm with that of LRU and FIFO. Across various workloads such as Vortex, Compress, gcc, and Pearl, LCS consistently demonstrated strong performance. It exhibited high hit rates, indicating efficient cache utilization in these scenarios. However, in the case of the go workload, LCS slightly underperformed compared to other workloads. Nevertheless, these findings highlight the overall effectiveness of the LCS algorithm in achieving favorable hit rates across a range of workloads.

C. Limitations of LCS

1) Complexity: : Implementing the LCS algorithm may require additional computational overhead compared to simpler cache replacement policies like FIFO (First-In, First-Out) or LRU (Least Recently Used). The need to maintain and update likelihood counters for each cache block can introduce complexity, potentially impacting overall system performance.

2) Parameter Sensitivity: : The effectiveness of the LCS algorithm can be influenced by the choice of parameters, such as the number of counter value zones (ROLs) and the initial counter value. Fine-tuning these parameters to achieve optimal performance across diverse workloads may require empirical testing and experimentation.

3) Storage Overhead: : Maintaining likelihood counters for each cache block incurs additional storage overhead, which can be a concern in memory-constrained systems. The increased memory footprint required by the LCS algorithm may reduce the effective cache size available for storing data, potentially impacting overall cache performance.

VI. RESULTS

In this section, we present the results of our cache simulation experiments as mentioned in the MP1Instructions.pdf, focusing on four key graphs derived from a series of simulations. These graphs offer valuable insights into the performance characteristics of different cache configurations, replacement policies, and inclusion properties. The experiments were conducted using varying cache sizes, associativities , block sizes, replacement policies, and inclusion properties to comprehensively analyze cache behavior across different scenarios.

A. Observations

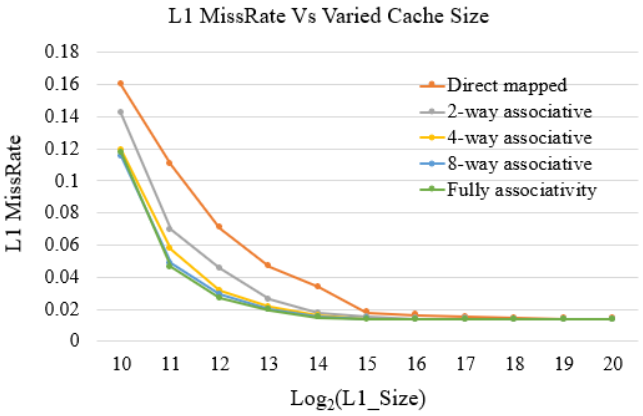

Fig. 7. GRAPH 1

Cache Size	Direct mapped	2-way associative	4-way associative	8-way associative	Fully associativity
10	0.14636	0.12852	0.10557	0.10137	0.10375
11	0.09637	0.05585	0.04354	0.0346	0.03246
12	0.05695	0.03154	0.01754	0.0152	0.01312
13	0.03302	0.01232	0.00773	0.00596	0.0055
14	0.02005	0.0036	0.002	0.00142	0.00109
15	0.00403	0.00124	0.00028	0.00005	0
16	0.00237	0.00022	0.00001	0	0
17	0.00129	0.00005	0	0	0
18	0.00081	0	0	0	0
19	0.00006	0	0	0	0
20	0.00006	0	0	0	0

Fig. 8. Table 5

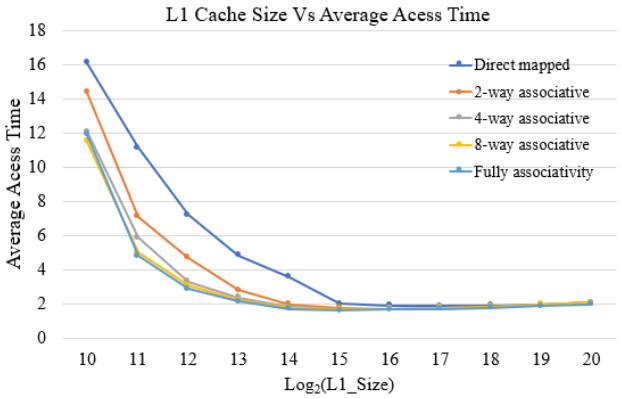

Fig. 9. GRAPH 2

1) From Fig 7, which represents the L1 miss rate to different cache sizes it is noticeable that as the cache size increases, the miss rates decrease for all the cache associativity. We can also observe that for all the associativity, at a certain point, the miss rate converges indicating that as the cache size increases, the advantage of higher associativity becomes minimal.
2) The compulsory miss rate obtained from the graph1 is 0.01376.
3) From Fig 7 the table presents the conflict miss rate for each associativity and cache size. This outcome was obtained by calculating the differences in miss rates for cache associativities of 1-way, 2-way, 4-way, and 8-way compared to a fully associative cache.

From Fig 9 which represents Graph 2 depicts the relationship between L1 Cache Size (on the x-axis) and the Average Access Time (AAT) (in milliseconds, on the y-axis). Four different lines represent different AATA (Average Access Time Acceleration) values: AAT(Associaivity=1), AAT(Associaivity=2), AAT (Associaivity=4), and AAT(Associaivity=8). As the L1 Cache Size increases, the Average Access Time generally decreases for all AAT values.Initially, there is a rapid decline in access time, which then flattens out as the cache size continues to increase.

From Fig 10 which represents Graph 3, we observe a clear inverse correlation between L1 cache size and Average Access Time (AAT).Three lines represent different replacement policies: “LRU” (Least Recently Used), “FIFO” (First In, First Out), and “OPTIMAL.” As the L1 Size increases from (in powers of 2) 10 to 18, the AAT for all three policies decreases.The “OPTIMAL” policy consistently has the lowest AAT, followed by “LRU,” and then “FIFO.”

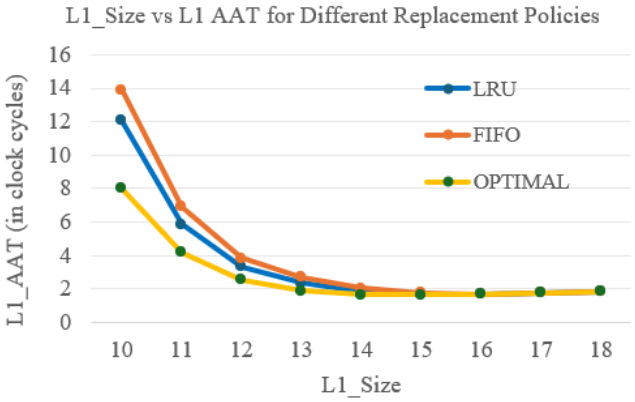

Fig. 10. GRAPH 3

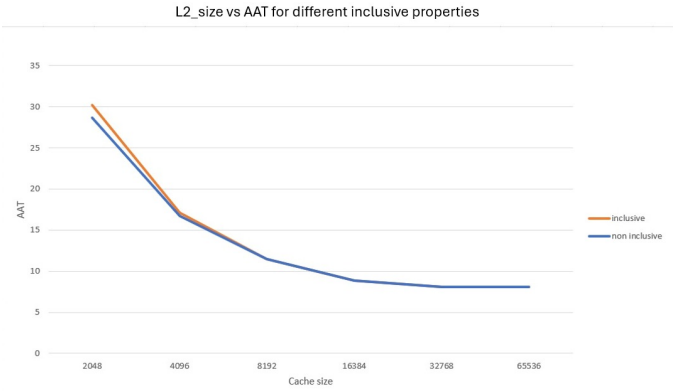

Fig. 11. GRAPH 4

From Fig 11 which represents Graph 4,Two lines are plotted, representing two different inclusive properties: ”inclusive” and ”non-inclusive”.The graph illustrates the relationship between cache size (on the x-axis) and the Average Access Time (AAT) (on the y-axis). Initially the Average Access Time (AAT) is lower for the Non-Inclusive configuration (represented by the blue line in the graph). However,they converge as the cache size grows.

Conclusion

In order to dynamically manage cache memory, we propose a method that logically partitions cache contents, demonstrating superior performance compared to traditional replacement algorithms. Our algorithm, characterized by conceptual partitioning into Regions of Likeliness (ROLs) and utilization of a 0-15 counter, exhibits efficient data storage and retrieval mechanisms.

REFERENCES

[1] M. Kharbutli and Y. Solihin, "Counter-Based Cache Replacement and Bypassing Algorithms," IEEE Transactions on Computers, vol. 57, no. 4, pp. 433–447, 2008.

[2] J. Reineke and D. Grund, "Sensitivity of cache replacement policies,"Trans. Embedded Comput. Syst, vol. 9, pp. 39–39, 2010

[3] T. J. Odule and I. A. Osinuga, "Dynamically self- adjusting cache replacement algorithm," Int. J. Future Generat. Commun. Netw, vol. 6,pp. 25–25, 2013.

[4] W.-C. Tseng, C. J. Xue, Q. Zhuge, J. Hu, and E. H. Sha, "PRR: A lowoverhead cache replacement algorithm for embedded processors," 17th Asia and South Pacific Design Automation Conference, pp. 35–40, 2012

[5] K. Kedzierski, M. Moreto, F. J. Cazorla, and M. Valero, "Adapting cache partitioning algorithms to pseudo-LRU replacement policies," 2010 IEEE International Symposium on Parallel Distributed Pro- cessing (IPDPS), pp. 1–12, 2010.

[6] H. Ghasemzadeh, S. Mazrouee, and M. R. Kakoee, "Modified pseudo LRU replacement algorithm," in *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pp. 6–376, 2006.

[7] D. Padua, "PARSEC Benchmarks," in *Encyclopedia of Parallel Computing, P. D., Ed.* Springer, 2011.

[8] E. Keshav, K. Manas Reddi, A. Pritam and S. Muthukumar, "Cache Performance Optimization Using Logical Cache Segmentation," 2021 Asian Conference on Innovation in Technology (ASIANCON), PUNE, India, 2021, pp. 1-6, doi: 10.1109/ASIANCON51346.2021.9544106. keywords: Technological innovation;Heuristic algorithms;Cache memory;Computer architecture;Tools;Market research;Distance measurement;Cache Replacement;LRU;gem5;PARSEC;4 Bit Counter,

[9] K. McMaster, S. Sambasivam, and N. Anderson, "How Anomalous Is Belady's Anomaly?," *Issues in Informing Science and Information Technology*, vol. 6, pp. 825–836, 2009.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.