2-DPR: A Novel, High Performance Cache Replacement Algorithm

Safana Hyder Abbas* & Salam Ayad Hussein**

*Department of computer science-College of Education-University of al-mustansiriya

**Department of computer science-College of Education-University of al-mustansiriya

Received 6 June 2015 ; Accepted 23 August 2015

Abstract

Caching is a fundamental technique commonly employed to hide the latency gap between memory and the CPU by exploiting locality in memory accesses. Different cache replacement algorithms have dramatically different effects on the system performance by deciding which blocks to evict from cache memory in case of a cache miss occurs. The aim of these replacement techniques is trying to get closer to the optimal case by achieving best usage of the total size of the cache, minimizing the miss ratio as much as possible and accomplishing the highest system performance can be reached. In this paper, a simple and elegant new algorithm is proposed, namely, Two-Dimensional Pyramid Replacement, (2-DPR), that combines the advantages of (LRU) and (LFU), and eliminates their disadvantages.

Keywords: Cache Memory, Replacement Algorithms, Cache Miss, Cache Hit.

2-DPR: تقنية استبدال ذاتية الضبط، مقاومة للمسح و عالية الاداء لذاكرة التخزين المؤقت

سفانة حيدر عباس & سلام اياد حسين

الخلاصة

تعد عملية التخزين المؤقت من العمليات الاساسية التي عادة ما تستخدم لتقليص الفجوة بين سرعة الذاكرة الرئيسية والمعالج المركزي عن طريق استغلال عملية إحلال مدخلات الذاكرة الرئيسية في الذاكرة المؤقتة. العديد من خوارزميات الاستبدال في الذاكرة المؤقتة كانت لها آثار مختلفة على مستوى اداء النظام عن طريق تقرير اي كتلة من البيانات سوف يتم حذفها من الذاكرة المؤقتة في حالة كونها ممتلئة، وهناك حاجة لإدخال بيانات جديدة غير موجودة مسبقا فيها. ان الغرض من

خوارزميات الاستبدال هو الاقتراب من الحالة المثالية عن طريق تحقيق افضل استخدام للحجم الكلي للذاكرة المؤقتة وتقليص معدل عدم وجود البيانات الى اقصى حد ممكن والحصول على اعلى اداء للنظام يمكن الوصول اليه. في هذا البحث، سوف نقوم بعرض خوارزمية استبدال جديدة تمتاز بالبساطة التي سوف يطلق عليها خوارزمية استبدال الهرم ثنائي الابعاد (2-DPR) التي تجمع بين مزايا خوارزميات الاستبدال (LRU) و (LFU) و تلغي مساوئهم.

الكلمات المفتاحية: ذاكرة التخزين المؤقت، خوارزميات الاستبدال، عدم وجود بيانات في الذاكرة، وجود بيانات في الذاكرة.

Introduction

Memory management is the art and the procedure of arranging and controlling the utilization of *memory* of a computer system [1]. It is a complex field in computer science, and is continually becoming more efficient due to the increasing number of techniques being developed for this purpose. The key problem during memory management is making the judgment of "when the contained data should be kept and when they can be evicted so that the memory can be reused". Modern designs attempt to eliminate memory management problems using a variety of techniques (instruction pipelines, instruction prefetch, branch prediction, simultaneous multithreading and CPU caches) [2].

As in figure (1), cache memory is a fast but small piece of memory between CPU and M.M that acts as a buffer for frequently used data [3]. One of the problems in cache memory is the so-called "cache miss". A cache miss occurs if the CPU requests an item of data that is not available in the cache. Otherwise, a "cache hit" occurs if the CPU requests an item of data that is available in the cache [4]. If a cache miss occurs, a replacement policy is required if the space required to load the memory blocks is not available in the cache. M.M is going to be searched for that item and transfers it to the cache, and a decision must be made to swap blocks. These steps are made by the replacement algorithms.

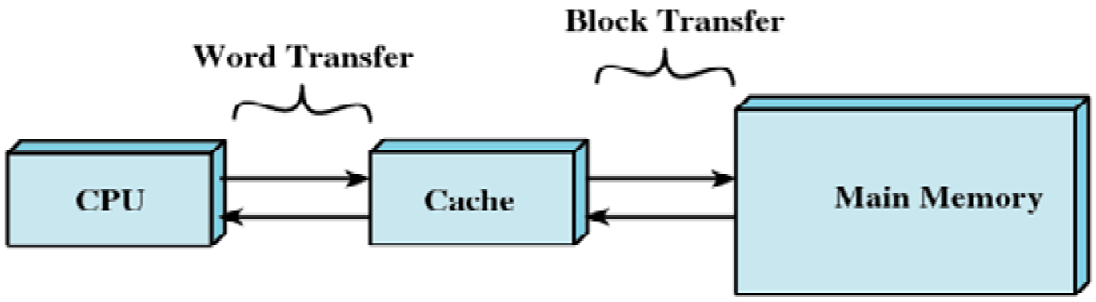

Figure-1 CPU, cache and main memory

Cache memory was invented to reduce the gap between the CPU performance speed and memory access time. The computer CPU runs at very high speed, but even modern memory technology does not provide the ability to convoy, causing a problem called the wait state [5]. A wait state is nothing more than an extra clock cycle to give some device time to complete an operation, it is a delay experienced by a computer processor when accessing external memory or any device that is slow to respond because while the CPU is waiting for data from memory it cannot operate on that data. Cache memory was invented by (M. V. Wilkes) in April 1965. At that time, Wilkes described the cache memory as a second level in High-speed unconventional memory that results in zero-wait state, and called it a "slave memory" [6].

Replacement Algorithms

Replacement Algorithms are needed when a cache miss occurs and there is no place in cache to load the memory blocks. In order to achieve high speed, the algorithm of replacement must be implemented in hardware [7]. A replacement algorithm chooses the existing block in the cache. Some of these algorithms are:

A- optimal: replaces the block that will be needed farthest in the future. This replacement policy is not practical, but is often used for comparison purposes to determine how effective other replacement policies are in the best possible [8].

B- Random: This algorithm would ignore memory references and select a cache block for replacement in a totally random order [9].

C- First-In, First-Out (FIFO): According to this algorithm, if a cache block must be replaced, then the oldest existing block is chosen [9].

D- Least Recently Used (LRU): This algorithm depends on the theory that if a block is referenced in the recent past, it might be referenced again in future. That is if a block has just been referenced it is likely that it will be referenced again [10].

E- Least-Frequently-Used (LFU): Such technique replaces the least-frequently used block when an eviction must take place, by using Software counter associated with each cache block [11]. Moreover, there are many other algorithms that were invented during (1981-2010) such as WSClock, 2Q, LRU-K, ARC, CAR, CART, LIRS, CLOCK-Pro and Dueling CLOCK [12].

The LRU's strength is that it keeps track of the 'recency' information about memory accesses [13]. However, the LRU's weaknesses are [14]:

i) LRU would require expensive hardware.

ii) LRU does not capture the 'frequency' factor.

iii) LRU is not "scan-resistant".

The LFU's strength is that frequently used blocks will stay longer than FIFO [15].

However, The LFU's weaknesses are [16]:

i) Older blocks are less likely to be removed, even if they are no longer frequently used because this algorithm never forgets anything.

ii) LFU does not capture the 'recency' factor.

Both (LRU) and (LFU) replacement algorithms will be the Inspiration of our new proposed technique.

The Proposed Algorithm

Basically, the proposed (2-DPR) technique suggests a Two-dimensional pyramid cache. As shown in (figure-2), it consists of four levels (L1, L2, L3 and L4), which represents the actual cache size. A block insertion is done through the base level (L1). While a block eviction is done through the top level (L4), which should be of one block size.

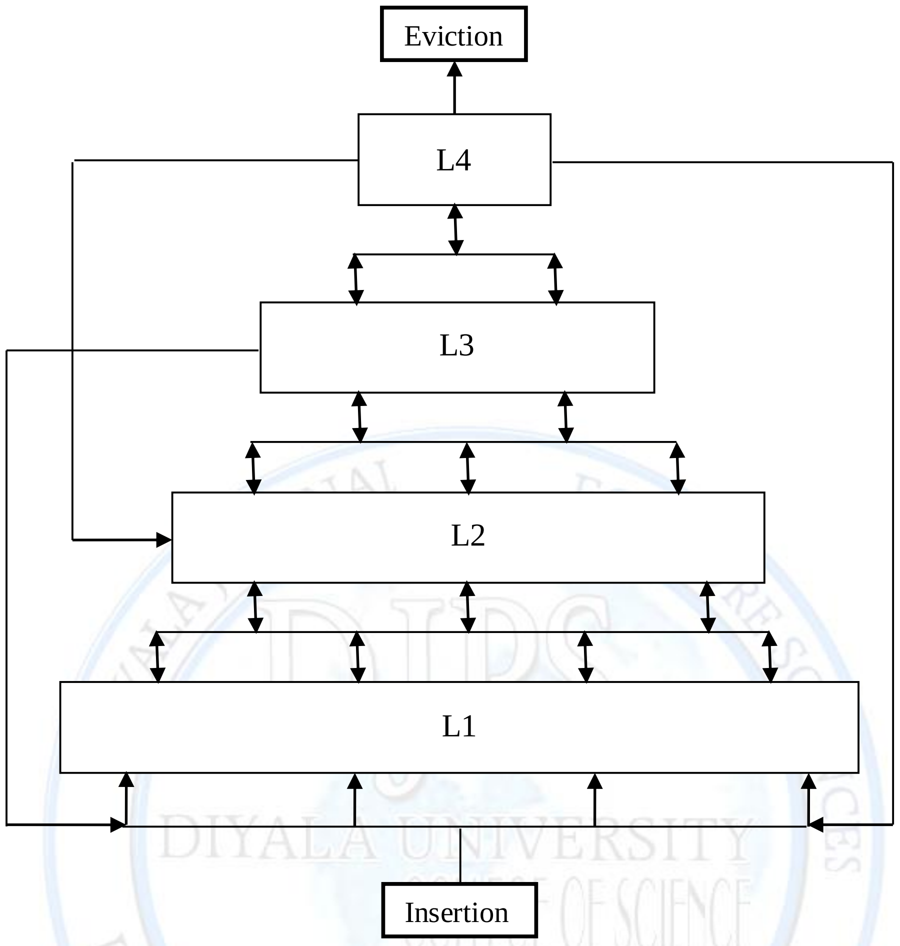

Figure-2 the proposed algorithm

Each block in this system is associated with two counters:

1. A reference counter (R): captures the 'recency' for all the blocks that reside in cache and works as follows:
- R=0 for any block is referenced or resides in cache and re-referenced.
- R=R+1 for any block that resides in cache and not referenced.
2. Frequency counter (F): captures the 'frequency' for all the blocks that reside in cache and works as follows:
- F=F+1 for any block is referenced or resides in cache and re-referenced.
- F=F for any block that resides in cache and not referenced.

The counter of the frequency will be controlled by a threshold that its value will be determined by system designers. These (R) and (F) can be implemented by software (by the operating system) or can be implemented totally by hardware. Initially, these four levels (L1, L2, L3 and L4) will be empty, in order to start filling them, the following steps will be performed:

1 Cache-Miss Steps

1. If a cache miss occurs, and there was an empty space in (L1). The new referenced block will be fetched from M.M and inserted in (L1) and the counters of referenced and existing blocks will be organized as:

- Referenced block: R=0 / F=F+1 (which holds an initial value of zero).
- Existing blocks: R=R+1 / F=F.

2. If a cache miss occurs, and there was no empty space in (L1) but (L2) has an empty space, the counters of the existing blocks will be organized as: R=R+1/ F=F.

   Then the block with [MAX_R] in (L1) will be shifted from (L1) to (L2). The new referenced block will be fetched from M.M and inserted in (L1) and the counters of the referenced block will be organized as:

   R=0 / F=F+1 (which holds an initial value of zero).

3. If a cache miss occurs, and there was no empty space in both (L1) and (L2) but (L3) has an empty space, the counters of the existing blocks will be organized as: R=R+1/ F=F.

   Then the block with [MAX_R] in (L2) will be shifted from (L2) to (L3), and the block with [MAX_R] in (L1) will be shifted from (L1) to (L2). The new referenced block will be fetched from M.M and inserted in (L1) and the counters of the referenced block will be organized as:

   R=0 / F=F+1 (which holds an initial value of zero).

4. If a cache miss occurs and there was no empty space in (L1), (L2) and (L3) but (L4) have an empty space, the counters of the existing blocks will be organized as: R=R+1 / F=F.

   Then the block with [MAX_R] in (L3) will be shifted from (L3) to (L4), block with [MAX_R] in (L2) will be shifted from (L2) to (L3) and the block with [MAX_R] in (L1) will be shifted from (L1) to (L2). The new referenced block will be fetched from M.M and inserted in (L1) and the counters of the referenced block will be organized as:

   R=0 / F=F+1 (which holds an initial value of zero).

5. Finally, if cache miss occurs and there was no empty space in (L1), (L2), (L3) and (L4), which means that cache is full and one block should be evicted in order to release a space to the new referenced block, then the block in (L4) will be evicted (because it will be the best candidate for eviction since it has been both least recently and frequently in use). The counters of the existing blocks will be organized as: R=R+1 / F=F.

Then the block with [MAX_R] in (L3) will be shifted from (L3) to (L4), block with [MAX_R] in (L2) will be shifted from (L2) to (L3) and the block with [MAX_R] in (L1) will be shifted from (L1) to (L2). The new referenced block will be fetched from M.M and inserted in (L1) and the counters of the referenced block will be organized as:

R=0 / F=F+1 (which holds an initial value of zero).

2 Cache-Hit Steps

Cache hit means that when CPU requests data item that is resides in the cache.

1. When a cache hit occurs, and the re-referenced block resides in (L1), then the only action that will be performed is to update the counters of the re-referenced and existing blocks as follows:

- Re-referenced block: R=0 / F=F+1.
- Existing blocks: R=R+1 / F=F.

2. When a cache hit occurs, and the re-referenced block resides in (L2), then the counters of the re-referenced and existing blocks will be organized as:

- Re-referenced block: R=0 / F=F+1.
- Existing blocks: R=R+1 / F=F.

Then the block with [MAX_R] in (L1) will be swapped with the re-referenced block.

3. If the cache hit occurs, and the re-referenced block resides in (L3), then the counters of the re-referenced and existing blocks will be organized as:

- Re-referenced block: R=0 / F=F+1.
- Existing blocks: R=R+1 / F=F.

Then one of three choices will be performed according to the following conditions:

A. If the frequency counter ($F_{counter}$) of the re-referenced block is ($1 \leq F \leq 3$), then the block with [MAX_R] in (L2) will be swapped with the re-referenced block.

B. If the ($F_{counter}$) of the re-referenced block is ($4 \le F \le 6$), the block with [MAX_R] in (L1) will be shifted from (L1) to (L2) and the block with [MAX_R] in (L2) will be shifted from (L2) to (L3). Then the re-referenced block will be placed in (L1).

C. If the ($F_{counter}$) of the re-referenced block is ($F>6$), then the block with [MAX_R] in (L2) will be swapped with the re-referenced block. And the re-referenced block counters will be organized as: [R=0/F=1].

4. If the cache hit occurs, and the re-referenced block resides in (L4), then the counters of the re-referenced and existing blocks will be organized as:

- Re-referenced block: R=0 / F=F+1.
- Existing blocks: R=R+1 / F=F.

Then one of four choices will be performed according to the following conditions:

A. If the ($F_{counter}$) of the re-referenced block is ($1 \le F \le 2$), then the block with [MAX_R] in (L3) will be swapped with the re-referenced block.

B. If the ($F_{counter}$) of the re-referenced block is ($3 \le F \le 4$), the block with [MAX_R] in (L2) will be shifted from (L2) to (L3) and the block with [MAX_R] in (L3) will be shifted from (L3) to (L4). Then the re-referenced block will be placed in (L2).

C. If the ($F_{counter}$) of the re-referenced block is ($5 \le F \le 6$), the block with [MAX_R] in (L1) will be shifted from (L1) to (L2) and the block with [MAX_R] in (L2) will be shifted from (L2) to (L3) and the block with [MAX_R] in (L3) will be shifted from (L3) to (L4). Then the re-referenced block will be placed in (L1).

D. If the ($F_{counter}$) of the re-referenced block is ($F>6$), then the block with [MAX_R] in (L3) will be swapped with the re-referenced block. And the re-referenced block counters will be organized as: [R=0/F=1].

(2-DPR) algorithm:

The Detailed algorithm steps can be explained as the following:

Initialization: set R=0, F=0 and cache levels L1, L2, L3 and L4 to empty.

```
Update (X): /*update referenced block/
Rx=0.
Fx=Fx+1.
```

```
Update (y): /*update existing blocks/
R=R+1.
F=F.
Input: the requested block (X).
Begin
If (X is in L1) then /*cache hit in L1/
Update (X).
Update (y).
Else /*cache miss in L1 and L1 is not full/
If (X is not in L1) and (L1< >c) then
Fetch (X).
Insert (X) in L1.
Update (X).
Update (y).
Else /*cache miss in L1 and L1 is full/
If (X is in L2) then /*cache hit in L2/
Move from (L1 to L2) block with $MAX_R$.
Move (X) to L1.
 Update (X).
Update (y).
Else /*cache miss in L2/
If (X is not in L2) and (L2< >c) then /*L2 is not full/
Move from (L1 to L2) block with $MAX_R$.
Fetch (X).
Insert (X) in L1.
Update (X).
Update (y).
Else /*cache miss in L2 and L2 is full/
If (X is in L3) then /*cache hit in L3/
Begin
```

```
If $(1 \le Fx \le 3)$ then
Move from (L2 to L3) block with $MAX_R$.
Move (X) to L2.
Update (X).
Update (y).
Else if $(4 \le Fx \le 6)$ then
Move from (L1 to L2) and (L2 to L3) blocks with $MAX_R$.
Update (X).
Update (y).
Else if $(Fx > 6)$ then
Move from (L2 to L3) block with $MAX_R$.
Move (X) to L2.
Rx=0.
Fx=1.
Update (y).
End if.
Else /*cache miss in L3/
If (X is not in L3) and (L3 < > c) then /*L3 is not full/
Move from (L1 to L2) and (L2 to L3) blocks with $MAX_R$.
Fetch (X).
Insert (X) in L1.
Update (X).
Update (y).
Else /*cache miss in L3 and L3 is full/
If (X is in L4) then /*cache hit in L4/
Begin
If $(1 \le Fx \le 2)$ then
Move from (L3 to L4) block with $MAX_R$.
Move (X) to L3.
Update (X).
Update (y).
Else if ($3 \le Fx \le 4$) then
Move from (L2 to L3) and (L3 to L4) blocks with $MAX_R$.
Move (X) to L2.
Update (X).
Update (y).
Else if ($5 \le Fx \le 6$) then
Move from (L1 to L2) and (L2 to L3) and (L3 to L4) blocks with $MAX_R$.
Move (X) to L1.
Update (X).
Update (y).
Else if ($Fx > 6$) then
Move from (L3 to L4) block with $MAX_R$.
Move (X) to L3.
Rx=0.
Fx=1.
Update (y).
Else /*cache miss in L4/
If (X is not in L4) then
Delete block in L4 out of cache.
Move from (L1 to L2) and (L2 to L3) and (L3 to L4) blocks with $MAX_R$.
Fetch (X).
Insert (X) in L1.
Update (X).
Update (y).
End if.
```

Performance Analysis

The efficiency criteria used to judge the performance of the proposed system will be:

A. The simplicity of the design, which make it implementable in the real world (as hardware).

B. Ease of the design work steps.

C. Processing speed (an algorithm must be scan-resistant in order to achieve high processing speed).

D. Hit ratio.

To evaluate our replacement algorithm experimentally, we simulated our policy and compared it with other policies like (LRU) and (LFU). We obtained the following results:

A. **2-DPR can be implemented in the real world as a hardware cache (IC)** that can be attached to the CPU (on-chip) or detached from it (off-chip). This Feature makes it achieve higher processing speed than some of the modern algorithms such as LRU (which needs an expensive hardware), or ARC, CAR, CART, CLOCK-Pro and Dueling CLOCK (which can be implemented only by the operating system) as a software.

B. Although this technique is highly dynamic, but the **2-DPR working steps are easy** to be performed, remembered and programmed.

C. When a cache miss occurs, and there is no empty space to insert the new data block, the eviction is done without scanning all cache blocks. The cache block in (L4) will be deleted (because it is the best candidate for eviction at any clock tick), since it will be balanced between least recently and frequently used. So, this means that **2-DPR is scan-resistant**, and can achieve a high processing speed comparing with (LRU) and (LFU).

D. Note that the **2-DPR frequency threshold had removed the (LFU) problem which 'keeps the older blocks in the cache even if they are no longer in use'**. Because they captures a higher frequency than other blacks.

E. The two parameters of 2-DPR, (R and F), at every cache referencing, fluctuates over its entire range of the existing cache blocks to a specific position and control their movement by balancing between 'recency' and 'frequency'. So, this proves that **2-DPR is a Self-Tuning** and **a high dynamic** technique.

F. After testing (LRU), (LFU) and (2-DPR) in cache size of (10) for a (data set-1) and (data set-2) and cache size of (11) for the (data set-3) and (data set-4): [14]

Data set-1:
{32,40,25,30,40,13,20,25,10,12,70,50,10,50,60,10,15,12,9,8,2,5,3,4,2,8,4,2,5,7,14,16,12,9,8,14,5,4,3,14,23,27,19,23,18,4,23,17,6,18}

Data set-2:
{100,110,120,130,1,2,3,4,1,2,4,140,150,3,2,1,4,6,5,170,180,190,200,5,6,1,7,3,4,2,5,6,7,4,210,220,1,3,5,7,2,4,6,1,8,6,7,2,3,4,2,3,6,7,8,230,240,250,260,270,280,290,300,7,8,1,3,5,6,2,4,1,8,6,310,320,330,340,350,1,8,7,2,4,3,8,6,5,7,1,360,370,380,390,400,410,6,5,4,3,2,1,6}

Data set-3:
{1,15,2,3,15,17,16,4,17,16,7,6,5,15,17,8,5,21,16,15,17,2,3,16,21,17,15,16,4,7,6,7,3,2,17,21,23,24,8,15,12,17,19,21,16,4,1,2,3,8,19,8,7,6,1,2,3,4,5,6,7,19,18,17,16,2,13,1,2,3,4,5,21,22,18,16,14,12,2,4,6,1,2,3,4,5,18,19,20,21,22,23,24,10,11,12,13,14,15,9,6,7,6,5,4,3,2,1}

Data set-4:
{32,40,25,30,40,13,20,25,10,12,70,50,10,50,60,10,15,12,9,8,2,5,3,4,2,8,4,2,5,7,14,16,12,9,8,14,5,4,3,14,23,27,19,23,18,4,23,17,6,18}

It can be observed from the table (1) and figure (3), after calculating the hit ratio with the equation (1), that (**2-DPR) has a higher hit ratio comparing to (LRU) and (LFU).**

$$\textbf{Hit ratio, \%} = \frac{\text{the Number of cache hits}}{\text{Total number of memory references}} * 100 \quad (1)$$

Table-1 Hit Ratio (%)

Data set	Data set-1	Data set-2	Data set-3	Data set-4
LRU (Hit %)	45.00	44.66	39.81	45.00
LFU (Hit %)	33.33	25.24	40.74	34.00
2-DPR (Hit %)	46.15	45.63	42.59	46.00

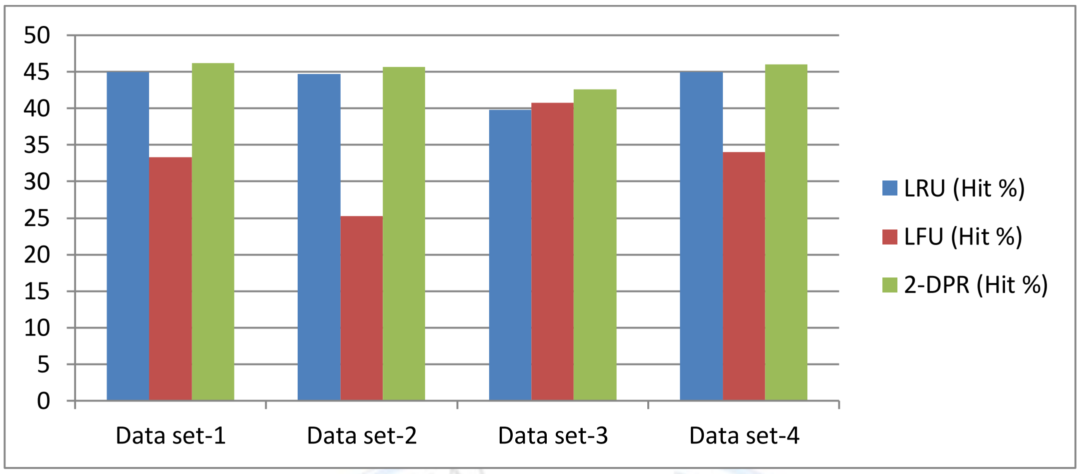

Figure -3 Hit Ratio chart

Conclusion

In this paper, by combining ideas and the best features from the LRU and LFU with the proposed data structure (Two-dimensional pyramid), the algorithm (2-DPR) removes disadvantages I, II and III of LRU. And I, II of LFU, which was mentioned in (2), so, now a new cache replacement algorithm introduced which is:

1- High-performance (by obtaining a higher hit ratio than LRU and LFU).

2- High-speed (because it is Scan-resistant).

3- Easy working steps.

4- Easy to implement in the real world (as hardware), which also makes it high-speed.

5- Self-tuned

References

1. Swain, D., Dash, B. N., O. Shamkuwar, D., and Swain, D., “Analysis and Predictability of Page Replacement Techniques towards Optimized Performance,” International Conference on Recent Trends in Information Technology and Computer Science (IRCTITCS), **2011**, pp. 12-16.
2. Kumari, M., Kumar, A., and Neha, “Simulation Of LRU Page Replacement Algorithm For Improving Performance Of System,” Int. J. Computer Technology & Applications, **2013**, Vol 4 (4), pp. 683-685.

3. Paajanen, H., “Page Replacement in Operating System Memory Management,” M.Sc, University of Jyväskylä, **2007**.

4. Murdocca, M. J., and Heuring, V. P., Principles of Computer Architecture: Class Test Edition, USA: New Jersy, **1999**, pp. 266-279.

5. Shamsheer Daula, S.M., Sreenivasa Murthy, K.E., and Amjad Khan, G., “A Throughput Analysis on Page Replacement Algorithms in Cache Memory Management,” International Journal of Engineering Research and Applications (IJERA), **2012**, Vol. 2, pp. 126-130.

6. Abd-El-Barr, M., and El-Rewini, H., Fundamentals of Computer Organization and Architecture: First Edition, New Jersey: A John Wiley & Sons, Inc Publication, **1999**, pp. 126-146.

7. Shiva, S. G., Computer Design and Architecture: Third Edition, New York: Marcel Dekker, Inc, **2000**, pp. 431-446.

8. Denning, P. J., and Ullman, J. D., “Principles of Optimal Page Replacement,” Journal of the Association for Computing Machinery, **1971**, Vol. 18, pp. 80-93.

9. Belady, L. A., “A study of replacement algorithms for a virtual-storage computer,” IBM Systems Journal, **1966**, Vol. 5. NO. 2, pp. 78-101.

10. Tanenbaum, A. S., Modern operating system: third Edition, New Jersey: Pearson Prentice Hall, **2009**, pp. 199-213.

11. Shah, K., Mitra, A., and Matani, D., “An O (1) algorithm for implementing the LFU cache eviction scheme,” *dhruvbird.com/lfu.pdf*, **2010**, pp. 1-8.

12. Bala, I., Raina, V., Sharma, O., and Alam, A., “Analytical Study of CAR, CLOCK and LRU Page Replacement Algorithms in Operating System,” International Journal of Advanced Research in Computer Science and Software Engineering, **2014**, Vol. 4, pp. 747-751.

13. Al-Zoubi, H., Milenkovic, A., and Milenkovic, M., “Performance Evaluation of Cache Replacement Policies for the SPEC CPU2000 Benchmark Suite,” Proc. 42nd ACM Southeast Conference, **2004**, pp. 267-272.

14. Godavarthy, A., Lakshminarasimhachar, S., and Gopinathan, S., “Simulation and Analysis of Cache Replacement Algorithms,” Proc. International *Conference* on Computer Design, **2010**, pp. 1-41.

15. Chavan, A. S., Nayak, K. R., Vora, K. D., Purohit, M. D., and Chawan, P. M., “A Comparison of Page Replacement Algorithms,” IACSIT International Journal of Engineering and Technology, **2011**, Vol. 3, NO.2, pp. 171-174.

16. Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C., Operating Systems: Three Easy Pieces: version 0.80, Wisconsin: Arpaci-Dusseau Books, Inc, **2014**, pp. 227-244.