A Novel Adaptive Cache Replacement Policy using Weighting and Ranking Parameter

Bhalgama Sanjay R.
Computer Engineering Department
C. U. Shah College of Engineering and Technology
Wadhwan City, Gujarat, India
bhalgamas@gmail.com

Parmar Shaktisinh S.
Computer Engineering Department
C. U. Shah College of Engineering and Technology
Wadhwan City, Gujarat, India
Shaktisinh.parmar@yahoo.co.in

Abstract— **Efficient cache replacement policy can improve the performance of processor. Efficient cache replacement has higher hit ratio and low overhead on system. We can improve the performance of system by using efficient cache replacement policy. The basic cache replacement policies, like Least Recently Used (LRU) and Least Frequently Used (LFU), only depends on either recency factor or frequency factor. To increase hit ratio, we can use hybrid policy based on features of two or more replacement policy. We propose new replacement policy which uses special buffer between cache memory and main memory and it uses weighting and ranking parameter. This replacement policy calculates weight of each block using three parameters: last access time, number of access and number of replacement done. And we will use special buffer between cache memory and main memory to store the page which are thrown away from cache For each miss in cache it checks into buffer and if block is not in buffer than it has to replace block from main memory to cache.. So we can improve the hit ratio by using new parameter to calculate weight and rank of block.**

Keywords- Buffer; Hit Ratio; LFU; LRU; WRP;

I. INTRODUCTION

Memory is essential component in computer system for storage, retrieve and access the programs and data. In computer system memory is physical devices used to store programs or data on a temporary or permanent basis for use in a computer or other digital electronic device [9]. The memory unit that communicates directly with the CPU is called the main memory. Device that provide backup storage are called auxiliary memory. This stores the data and program which are loaded into main memory and then to CPU for execution. A special very high speed memory called a *cache* is sometime used to increase the speed of processing by making current program and data available to the CPU at rapid rate. Cache memory is to make CPU processing faster. Cache is small, faster memory between CPU and main memory. It store the block that are repeatedly accessed. Cache memory is used to make CPU processing faster. Cache is small, faster memory between CPU and main memory. It store the block that are repeatedly accessed [9]. If the active portion of the program and data are placed in a fast small memory, the average memory access time can be reduced, thus reducing the total execution time of the program. And this small cache is placed between processor and main memory.

When CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast cache memory. If word addressed by the CPU is not found in the cache, the main memory is accessed to read the word. A block of words containing the one just accessed is then transferred from main memory to cache memory.

Cache memory is very fast memory but it has small size. When CPU needs to access memory, the cache is examined. If the word is found in the cache, it is read from the fast cache memory and this is called cache hit. If word addressed by the CPU is not found in the cache, the main memory is accessed to read the word and it is called cache miss [2]. A block of words just accessed is then transferred from main memory to cache memory. If cache memory has free space than it make entry otherwise when cache is full than we have to evict the block from cache and make space for new entry. The scheme for selecting the block for eviction is called cache replacement policy [2].

Performance of CPU is heavily depends on cache replacement policy [1]. Efficient cache replacement policy is used to improve hit ratio and it leads to higher performance. There are so many cache replacement policies which has their own strategy.

In next section we will discuss about different cache replacement policy. In section III we will discuss about our proposed replacement policy and then in section IV we will discuss about implementation and result analysis. Then in last section we will conclude our work.

II. CACHE REPLACEMETN POLICY

A. *First In First Out (FIFO) Policy*

This is very basic low overhead cache replacement algorithm. This is very easy to implement. First In First Out (FIFO) algorithm works like simple Queue. Cache stores the each data like queue. Block at head of list is oldest one and block at the tail of list is current block [4]. So when we want to replace the block from cache memory using FIFO policy it replaces the block head of the list which is oldest one.

B. *Least Recently Used (LRU) Policy*

This is basic replacement policy which focuses on recency of block. When here is need to replace the block from cache memory it will select the block which is not referenced from longer time [2]. It means that block is not used for longer time so it will assume that it will not use in near future also. This is most used replacement policy. The main advantage of this policy is its easy implementation and ease of use. The limitation of this policy is that it only focuses on recency of block so block which is most frequently used are not considered as future use. This algorithm sorts cached documents by the latest access time. When a cache hit occurs then access time of the document is updated and it is moved to the data head of the list [1]. The least recently used document which is located at the tail of the list is the next to be replaced.

C. *Least Frequently Used (LFU) Policy*

LFU replacement policy focuses on frequency of block. When there is need to replacement from cache than it will select the block which has lowest frequency [4]. Frequency means number of time block is referenced. So LFU assume that block which has lowest frequency is not referenced repeatedly so it will not access in near future. So it will replace the block with low frequency [2]. The LFU replacement policy has few drawbacks: it only focus on frequency not on recency.

D. *Weighting Replacement policy (WRP)*

This is replacement policy based on weighting and ranking of block. It will assign weight to each block according to W_i value. For ranking referenced pages, we consider three factors. Let L_i be the counter which shows the recency of block *i* in the buffer and F_i be the counter which shows the number of times that block *i* in buffer has been referenced [5]. Consider the time difference $\Delta T_i = T_{ci} - T_{pi}$ where T_{ci} is the time of last access and T_{pi} is the time of penultimate access. Then the weighting value of block *i* can be computed as

$$wi = \frac{Li}{Fi \times \Delta T}$$

L_i will work like LRU counter, when new object k is placed in the buffer then all of the above mentioned factors in weighting function must be set, and it will be followed by setting L_k to 0, F_k to 1 and ΔT_k to 1. *Fk* has been set to 1 because it means that object *k* has been used once and we assume initial value of ΔT_k equals to 1 because the time between each reference to a block would be at least one in its minimum case [5]. In every access to buffer, if referenced block *j* is in the buffer then a hit is occurred and our policy will work in this way:

1) L_i will be changed to $L_i + 1$ for every $i \neq j$.
2) For $i = j$ first we put $\Delta T_i = L_i$, $F_i = F_i + 1$, and then $L_i = 0$.

But if referenced block *j* is not in the buffer a miss occurs and the algorithm will choose the block in buffer which the value of its weighting function is greater than the others. Searching for object with greatest weighting value will be started in the buffer from top to down. In this way, if values of some object are equal to each other, the object which is placed upper in the buffer will be chosen to evict from buffer [3]. It means that our policy follows FIFO low in its nature. Let assume that a miss has been occurred and block *k* has the greatest weighting value and it must be evicted from buffer, first we change *Li* to *Li + 1* for every $i \neq k$ and then replace new referenced block with block *k* and at last we must set all weighting factors of block *k* to their initial values. It will replace the block with maximum W_i value. Weighting Replacement policy has higher hit ratio than existing LRU and LFU [5].

III. PROPOSED POLICY

We have modified the available weighting replacement policy (WRP). We have designed a new equation that will used to select block from memory for replacement. This new equation is used to give weight and rank to each block of cache. We are using extra buffer to give each block another chance [6]. When block is not into cache then it will check into buffer and if it is not into buffer than it access block from main memory.

A. *Parameter used in new policy*

We have designed new equation as bellow. That uses three parameter like last access time, number of time block is replaced and number of time block accessed. We have used extra buffer so we have assumed that block which is replaced many time from buffer to cache and cache to buffer are most important block and that is used in near future also. So we have taken number of replacement in consideration for calculating the weighting value. When time to last access is smaller and number of access and number of replacement is higher than it will give smaller Ri value. This shows that block with smaller Ri is most recent and most frequent. Due to this reason we are replacing the block with higher Ri value when replacement is needed.

$$Ri = \frac{Tlast}{Na \times Nr} \qquad \text{Equation 1}$$

Where T_{last} = Last time of access, N_a = No. of time block accessed and N_r = No. of time block replaced.

By using this function for replacement we are focusing on last access time, no of time block is replaced and also on no time block is accessed. So it will give benefits of both LRU and LFU and also focuses on number of replacement.

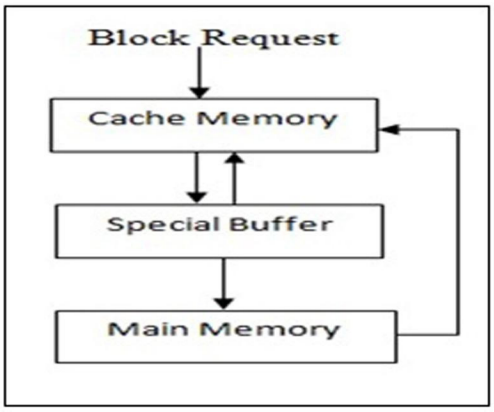

Figure 1. Buffer memory[8]

This figure shows that how our policy works. First of all we will check into cache memory and if block in not into cache memory we have to look up into buffer if block is available into buffer than we will replace block from buffer to cache otherwise we will fetch block from main memory. By using this special buffer between cache and Main memory we are reducing power consumption because power consumption of fetching block from main memory is higher than power of accessing block from cache [8]. Time to access block from cache is 10 times smaller than main memory [7]. We assume the size of the blocks is equal and the replacement algorithm is used to manage finite number of blocks.

A hit occur when there is a reference to a block in the cache. A partial hit occur when there is a reference to a block in the special buffer. When we have a reference to a block not in cache and buffer, a miss will occur. When a miss occur and no free frame available in cache, we must replace a new block to evicted block.

B. Procedure

We will use the new equation of R_i for replacement from cache memory. This equation will be used to calculate the weight to each block in cache memory. And we have to replace the block from cache than we will select the block which has highest R_i value. This equation will focus on number of replacement, number of access to each block and last time of block access.

Following figure shows the pseudo code for our algorithm. This shows that when there is a miss in cache memory than we have to look in buffer. If we block is not available in buffer than we have to replace the block from main memory to cache. And if block is in buffer than we move block from buffer to cache so it will reduce the time to search the block from main memory.

```
Step 1: Take trace file as input
Step 2: Fetch address from trace file
Step 3: Check if address is available in cache or not.
Step 4: IF address found in cache than Hit occurs.
        Go to Step 2. And continue for all address in trace file.
Step 5: IF address not found in cache than Partial Miss occurs.
        {
            // check in buffer.
            IF (miss in Buffer)
            {
              // Move block from main memory to cache
              IF (Free space in cache)
                  Move block
              ELSE
                 Execute Replacement policy using Equation 1.
            }
            ELSE
            {
                // Move block from Buffer to cache
                IF (Free block in cache)
                      Move block
                ELSE
                  Execute Replacement policy using Equation 1.
            }
        }
Step 6: Continue this process for all address in trace file.
Step 7: Calculate Hit Ratio.
Step 8: END.
```

Figure 2. Proposed Algorithm

When new block is inserted into cache memory than counter T_{last} is set to 0, number of access N_a =1 and number of replacement N_r =1. For every hit into block N_a is incremented by 1 and N_a is set to 0 for that block and for remaining block it will incremented by 1. And for miss in the block T_{last} for each block is incremented by 1 and when block is replaced N_r is incremented by 1. For hit in the cache, Hit counter is increm1ented and for miss in the cache, Miss Counter is incremented. Hit ratio is calculated by total hit divide by total number of block accessed. We will compare the hit ratio of different cache replacement policy to measure performance of our new replacement policy.

IV. IMPLEMETATION AND RESULT ANALYSIS

In this chapter we will discuss implementation and simulation results. It also contains the platform and data set. We calculate the performance based on the hit ratio parameter. We will simulate our new policy using following dataset, tools and the bellow procedure.

We used following traces to simulate our algorithm. Each trace is a hexadecimal address of a running program, taken from the Standard Performance Evaluation Corporation (SPEC) benchmarks. We have simulated our policy using gcc, swim, bzip and sixpack trace file.

Efficiency of algorithm also depends on data structure used to implement algorithm [1]. We have implemented our algorithm in C++ language using Linked list data structure to create virtual cache memory blocks and a queue is used to create buffer.

A. *Buffer Size*

We have simulated our algorithm on different cache size and buffer size and taken the hit ratio and time of miss in calculation. We have taken these results on the gcc, swim, sixpack and bzip trace files. Following tables shows the comparison of hit ratio and miss time on different trace file and buffer size.

TABLE I. COMPARISON OF HIT RATIO ON DIFFERENT BUFFER SIZE AND 1000 BLOCK CACHE SIZE

Buffer Size	**Cache Size =1000 (#blocks)**			
	Gcc	***Swim***	***Sixpack***	***Bzip***
100	92.8962	92.2468	88.0694	97.9876
300	92.8859	92.2353	88.0696	98.0000
500	92.8867	92.2339	88.0710	98.0216
700	92.8869	92.234	88.0723	98.0347
1000	92.8874	92.2184	88.0770	98.0521

This table shows the hit ratio comparison on 1000 block cache size and different buffer size. We have taken here 1000, 700, 500, 300 and 100 block buffer. We have compared results with gcc, swim, sixpack and bzip trace file.

TABLE II. COMPARISON OF HIT RATIO ON DIFFERENT BUFFER SIZE AND 700 BLOCK CACHE SIZE

Buffer Buffer Size	**Cache Size =700 (#blocks)**			
	Gcc	***Swim***	***Sixpack***	***Bzip***
70	92.7831	92.1828	87.3580	97.7685
100	92.7833	92.2791	87.3584	97.7722
300	92.7759	92.2821	87.3610	97.7949
500	92.7777	92.2824	87.3614	97.8082
700	92.7783	92.2824	87.3625	97.8160

This table shows the hit ratio comparison on 700 block cache size and different buffer size. We have taken here 700, 500, 300, 100 and 70 block buffer.

TABLE III. COMPARISON OF HIT RATIO ON DIFFERENT BUFFER SIZE AND 500 BLOCK CACHE SIZE

Buffer Size	**Cache Size =500 (#blocks)**			
	Gcc	***Swim***	***Sixpack***	***Bzip***
50	92.6359	92.0111	86.5472	97.5423
100	92.6435	92.0166	86.5485	97.5444
200	92.6437	92.0170	86.5499	97.5492
300	92.6468	92.0294	86.5520	97.5556
400	92.6487	92.0390	86.5539	97.5645
500	92.6503	92.0430	86.5569	97.5709

This table shows the hit ratio comparison on 500 block cache size and different buffer size. We have taken here 500, 400, 300, 200, 100 and 50 block buffer.

TABLE IV. COMPARISON OF HIT RATIO ON DIFFERENT BUFFER SIZE AND 300 BLOCK CACHE SIZE

Buffer Size	**Cache Size =300 (#blocks)**			
	Gcc	***Swim***	***Sixpack***	***Bzip***
30	92.5029	91.7864	84.8511	97.1789
50	92.5048	91.7880	84.8517	97.1808
100	92.5103	91.7953	84.8527	97.1886
200	92.5115	91.8019	84.8574	97.2004
300	92.5184	91.8025	84.8591	97.2199

This table shows the hit ratio comparison on 300 block cache size and different buffer size. We have taken here 300, 200, 100, 50 and 30 block buffer.

TABLE V. COMPARISON OF HIT RATIO ON DIFFERENT BUFFER SIZE AND 100 BLOCK CACHE SIZE

Buffer Size	**Cache Size =100 (#blocks)**			
	Gcc	***Swim***	***Sixpack***	***Bzip***
10	91.1755	90.4460	73.3117	96.5071
30	91.1757	90.4522	73.3125	96.5074
50	91.1757	90.4602	73.3131	96.5088
70	91.1767	90.4681	73.3134	96.5113
100	91.1771	90.4710	73.3140	96.5139

This table shows the hit ratio comparison on 100 block cache size and different buffer size. We have taken here 100, 70, 50, 30 and 10 block buffer.

According to this results we have analyzed that increasing the buffer size on same cache size may increase the hit ratio. But these show the small amount of change up to 0.01 to 0.1. And this might take more time to run on higher buffer size. Smaller buffer size near about 10% of cache size also gives increment in hit ratio and it takes limited time but when we increase buffer size it increase the hit ratio with small change. So we have decided to take buffer size 10% of cache size.

B. Simulation Result

Simulation results are the average of the result obtains after running same parameter 10 times for each. Here we are compared the result of existing weighting replacement policy (WRP), Least Recently Used policy (LRU), and Clock Algorithm with our augmented algorithms.

TABLE VI. HIT RATIO COMPARISON USING GCC TRACE FILE ON DIFFERENT CACHE SIZE

Size Of Cache	Hit Ratio Comparison Of Different Policy.			
	LRU	*CLOCK*	*WRP*	*Modified WRP*
100	87.3000	84.4500	84.4300	91.1755
200	90.5200	88.6790	88.6600	92.2648
300	91.5900	90.0740	90.0700	92.5029
400	91.9700	90.7310	90.7300	92.5812
500	92.1030	91.0570	91.0500	92.6359
600	92.1660	91.4140	91.4100	92.7342
700	92.2620	91.5840	91.5800	92.7831
800	92.3930	91.7650	91.7600	92.8351
900	92.4690	91.8930	91.8900	92.8680
1000	92.5200	92.0060	92.0000	92.8962

The table shows the hit ratio of different algorithm on cache size of 100 to 1000.

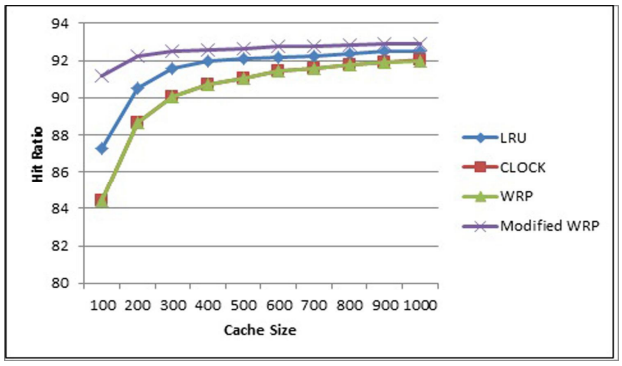

Figure 3. Comparison of hit ratio using gcc trace file

This graph shows the comparison of different cache replacement algorithm according to hit ratio. This comparison is based on gcc trace file. This will clearly show that our modified WPR policy has higher hit ratio than existing one. It has 1 % greater hit ratio than LRU, 2.16 % greater hit ratio than CLOCK, 2.16 % greater hit ratio than WRP. So this will clearly shows that our proposed equation will give higher performance than existing one according to hit ratio comparison.

TABLE VII. HIT RATIO COMPARISON USING SWIM TRACE FILE ON DIFFERENT CACHE SIZE

Size Of Cache	Hit Ratio Comparison Of Different Policy.			
	LRU	*CLOCK*	*WRP*	*Modified WRP*
100	84.0000	81.7020	81.6500	90.4460
200	87.6700	85.4400	85.4100	91.4770
300	88.8500	87.4116	87.4100	91.7864
400	89.7100	88.3981	88.3900	91.8988
500	90.5800	88.9880	88.9800	92.0370
600	91.1600	89.2900	89.2900	92.1145
700	91.6700	90.1440	90.1400	92.1828
800	91.9000	90.4470	90.4400	92.1887
900	92.0000	90.5600	90.5600	92.2211
1000	92.1100	90.6120	90.6100	92.2468

The table shows the hit ratio of different algorithm on cache size of 100 to 1000.

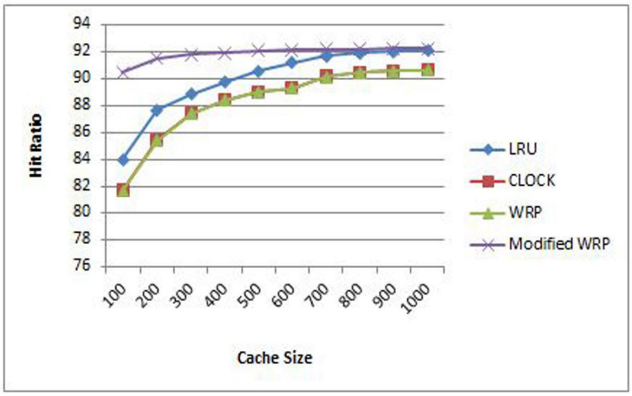

Figure 4. Comparison of hit ratio using swim trace file

This graph shows the comparison of different cache replacement algorithm according to hit ratio. This comparison is based on swim trace file. This will clearly show that our modified WPR policy has higher hit ratio than existing one. It has 1.894 % greater hit ratio than LRU, 3.569 % greater hit ratio than CLOCK, 3.571 % greater hit ratio than WRP. So this will clearly shows that our proposed equation will give higher performance than existing one according to hit ratio comparison.

TABLE VIII. HIT RATIO COMPARISON USING SIXPACK TRACE FILE ON DIFFERENT CACHE SIZE

Size Of Cache	Hit Ratio Comparison Of Different Policy.			
	LRU	*CLOCK*	*WRP*	*Modified WRP*
100	63.4718	59.5060	59.4546	73.3117
200	67.6957	64.9383	64.9266	82.0327
300	75.1328	71.0743	71.0648	84.8511
400	76.3007	73.7495	73.7245	85.9515
500	77.7612	75.2248	75.2113	86.5472
600	78.9158	76.2193	76.2160	86.9265
700	79.9169	77.5127	77.5097	87.3580
800	80.7698	78.2126	78.2004	87.6510
900	81.3861	78.9103	78.9069	87.9391
1000	81.9360	79.4196	79.4175	88.0694

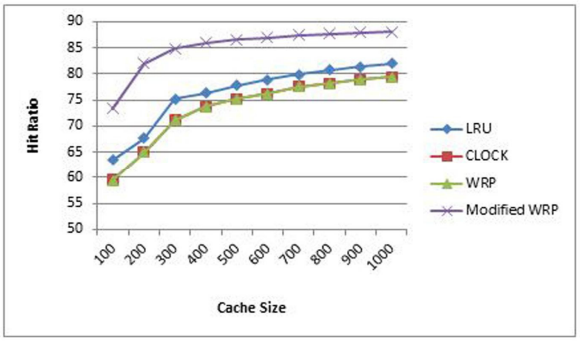

Figure 5. Comparison of hit ratio using sixpack trace file

This graph shows the comparison of different cache replacement algorithm according to hit ratio. This comparison is based on sixpack trace file. This will clearly show that our modified WPR policy has higher hit ratio than existing one. It has 9.735 % greater hit ratio than LRU, 12.593 % greater hit ratio than CLOCK, 12.6 % greater hit ratio than WRP. So this will clearly shows that our proposed equation will give higher performance than existing one according to hit ratio comparison.

TABLE IX. HIT RATIO COMPARISON USING BZIP TRACE FILE ON DIFFERENT CACHE SIZE

Size Of Cache	HIT RATIO COMPARISON OF DIFFERENT POLICY.			
	LRU	*CLOCK*	*WRP*	*Modified WRP*
100	94.95	93.5786	93.5352	96.5071
200	95.3921	95.3623	95.3611	97.0639
300	95.8530	95.7641	95.7637	97.1789
400	96.4450	96.1693	96.1694	97.2801
500	96.6480	96.6518	96.6516	97.5423
600	96.8598	96.8966	96.8961	97.6604
700	96.9503	97.0868	97.0869	97.7685
800	96.9637	97.2135	97.2134	97.8653
900	97.0639	97.3715	97.3716	97.9419
1000	97.1588	97.4643	97.4642	97.9876

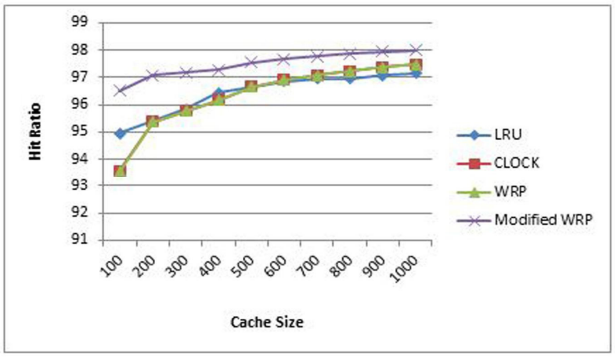

Figure 6. Comparison of hit ratio using bzip trace file

This graph shows the comparison of different cache replacement algorithm according to hit ratio. This comparison is based on gcc trace file. This will clearly show that our modified WPR policy has higher hit ratio than existing one. It has 1.042 % greater hit ratio than LRU, 1.114 % greater hit ratio than CLOCK, 1.119 % greater hit ratio than WRP. So this will clearly shows that our proposed equation will give higher performance than existing one according to hit ratio comparison.

V. CONCLUSION

Performance of system depends on cache replacement policy. Efficient cache replacement policy has higher hit rate and low overhead on system. Proposed policy is used calculate weight factor of each cache block. This replacement policy replaces the block which has higher weight. We have simulated this new policy on different trace file like gcc, swim, sixpack and bzip. And we have compared the hit ratio of different algorithm on different cache size. This clearly shows that our proposed policy improves the hit ratio compared existing cache replacement policy.

VI. REFERENCES

[1] Mr. C. C. Kavar, Mr. S. S. Parmar, "Performance Analysis of Page Replacement Algorithm with reference to different Data Structure", International Journal of Engineering Research and Application (IJERA), vol-3, issue-1, January-2013.

[2] S.M.Shamsheer Daula, Dr. K.E. Sreenivasa Murthy and G amjad Khan, "A Throughput Analysis on Page Replacement Algorithms in Cache Memory Management," IJERA, vol. 2, pp. 126-130, March-April 2012.

[3] Debabala Swain, Bijay Paikaray and Debabrata Swain, "AWRP: Adaptive Weight Ranking Policy for Improving Cache Performance," Journal of Computing, Volume 3, February 2011.

[4] Dr. K a parthasarathy." Performance evaluation of page removal policies" in journal of theoretical and applied information technology, 15th April 2011, vol. 26 no. 1

[5] Kaveh Samiee, "WRP: Weighting Replacement Policy to Improve Cache Performance," International Journal of Hybrid Information Technology, vol.2, 2009.

[6] Zhansheng L., Dawei L., and Huijuan B., "CRFP: A novel adaptive replacement policy combined the LRU and LFU policies," in proceedings of IEEE 8th international conference on computer and information technology workshops, Sydney, pp. 72-79, 2008

[7] Theodore Johnson and Dennis Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," Proceedings of the 20th VLDB Conference Santiago, Chile, 1994.

[8] Mr. S. R. Bhalgama, Mr. C. C. Kavar, Mr. S. S. Parmar, "LWRP: Low Power Consumption Weighting Replacement Policy using Buffer Memory" in International journal of Computer Trends and Technology (IJCTT) volume-7, number-3 pp. 147-150, Jan-2014.

[9] M. Morris Mano. "Memory Organization" in Computer System Architecture, 3rd Edition, Pearson Prentice Hall, 1992, pp.447-489.