Mr.C.C.Kavar, Mr. S.S.Parmar / International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622 www.ijera.com
Vol. 3, Issue 1, January -February 2013, pp.2070-2076

Performance Analysis of LRU Page Replacement Algorithm with Reference to different Data Structure

Mr.C.C.Kavar[1], Mr. S.S.Parmar[2]
[1]PG Student, [2]Asst. Professor
[1,2]C.U.Shah College Of Engg.& Technology, Wadhwan,Gujarat,India.

Abstract—
The process takes less time for handling instructions then would be speedy and efficient. The speed of the process is not only depends on architectural features and operational frequency, but also depends on the algorithm and data structure, which is used for that process. There are many page replacement algorithms such as Least Recently Used ((LRU), First-In-First-Out (FIFO), etc. are available in memory management. Performance of any page replacement algorithm depends on data structure which is used to implement a page table. Now a day, hash table is widely used to implement a page table because of its efficiency in dictionary operations. In this paper we use self-adjustable doubly circular link list, skip list and splay tree as a data structure to implement page table for LRU algorithm. This paper shows that how the combination of LRU with self-adjustable doubly circular link list, skip list and splay tree towards improvement of hit ratio.

***Keywords*— Memory Management, Cache Performance, Replacement Policy, Data structure.**

I. INTRODUCTION

Page replacement is an important concept in memory management system. When new process is created, require pages for this process must reside in main memory. Process search require pages from main memory. If pages are not available, then page fault will occur [1,2,4]. After that process pass through the following steps [3]:

Step 1: Find the faulted page from the Disk.

Step 2: Find a free space for faulted page in main memory:

(a) If space is available, use it.
(b) If there is no free space, use a page replacement algorithm to evict a page.
(c) Write the evict page to the disk.

Step 3: Read the required page into the newly created free space

Step 4: Restart the process

There are many page replacement algorithms such as Optimal, Least Recently Used ((LRU), Not Recently Used (NRU), First-In-First-Out (FIFO), Not Frequently Used (NFU), Second Chance, Clock, Aging, Working Set, WSClock, etc. are available in memory management [3,4]. Scheme of every page replacement algorithm is operated by the following three operations:

1) Search: To search required pages from main memory.
2) Delete: To delete the evict page from main memory.
3) Insert: To insert the page into main memory.

Therefore hit ratio and time complexity of any of the page replacement algorithm is depends on the data structure such as Queue, Link List, Hash Table, Skip List, Splay Tree etc. A linked list is a data structure consisting of a group of nodes which together represent a sequence. Complexity of Link List is depends on which type of Link List we are using. In hash table, search and delete complexity is O(1+ N/k) and insertion complexity is O(1) [14]. A hash table is widely used to implement a page table because of its efficiency in dictionary operations. In Splay Tree and Skip List, search, delete and insertion complexity is O(log N) [3,4,6,13,14].

The remainder of this paper is organized as follow: In the next section, we review some of the previous LRU policy with data structure used for that. In section III, we describe the LRU with Self Adjustable Link List, Skip List and Splay Tree. Proposed algorithms are discussed in section IV. In section V, the simulation and its result analysis are discussed. In section VI we discussed the comparison and conclusion of our algorithms.

II. BACKGROUND

LRU page replacement algorithm is based on the observation that most heavily used pages in last few instruction will probably used in the next few instruction. When a page fault occurs in the LRU, throw out the page that has been not used for longest time. Implementation of LRU is done either in software or in hardware [1,2,3,4]. In software implementation of LRU, it is necessary to maintain a linked list of all pages in main memory, with most recently used page at the front end and least recently used page at the rear end. In hardware implementation of LRU, require equipping the hardware with a64-bit counter that is automatically incremented after each instruction. When a page fault occurs, operating system examines all the

counters in the page table to find the lowest counter. Another hardware implementation, n frames, LRU hardware can maintain n x n bits, initially all set to zero. When page k is referenced, hardware first sets all the bits of row k to 1, and then sets all the bits of column k to 0. Now when page fault occur, the row whose binary value is lowest is the least recently used [3,4,7,8].

Instead of hardware, one possible solution is NFU algorithm. It requires a software counter associated with each page, initially zero. When a page fault occurs, the page with the lowest counter is chosen for replacement. But the main problem with NFU is never forgets anything [3,8,9]. A simple modification in NFU algorithm helps to simulate LRU quite well. The modification consists of two parts. First, counters are each shifted right 1 bit before the R bit is added. Second, the R bit is added to the leftmost. This modified algorithm is known as a Aging algorithm. When a page fault occurs, lowest counter page is removed [3,5].

LRU-k algorithm is to record of the times of the last K references. LRU-k algorithm gives the benefits about page access frequency; LRU-k is different from LFU. The difference is that LRU-K has a predefined concept of "aging", considering only last K references to a page [9,10]. Whereas the LFU has no means to recent or past recent. 2Q provide little bit significant than LRU without increasing the overhead [11,12].

III. LRU WITH VARIOUS DATA STRUCTURE

A. *LRU with Self Adjustable Doubly Circular Link List*

A self adjustable doubly circular link list is a link list that arranges its elements based on some heuristic to improve average access time. The goal of a self adjustable doubly circular link list is to improve efficiency of linear search. There are many techniques available for rearranging the nodes such as Move to Front Method (MTF), count method, transposes method etc. Here we are used MTF method for rearrange the nodes [14]. In MTF method, moves the element which is accessed to the head of the link list shown in Fig. 1.

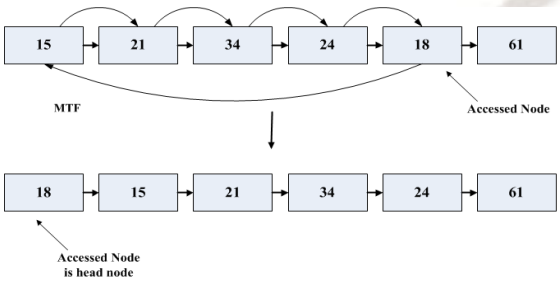

Fig. 1. MTF method

How LRU implemented using self adjustable circular link list shown in Fig. 2.

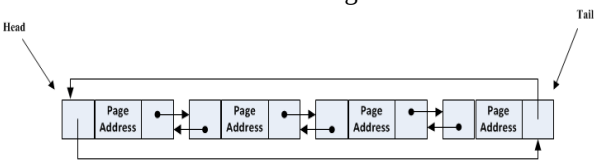

Fig. 2. Self adjustable doubly circular link list

When any page is accessed, it directly moves to head of the link list. Therefore least recently pages moves at head or near about it. And we insert the element at the head position and delete the element at the tail position of the self adjustable doubly circular link list [15].

B. *LRU with Skip List*

We propose a data structure for efficiently manage pages based on Skip List. A Skip List is a probabilistic data structure in which nodes are arranged in parallel link list. In the Skip List, search, insertion and deletion complexity is O (log N) [6,13]. William Pugh compared the performance of implementation of Skip List and other data structure Table I shows the comparison of the implementation of the Skip List, AVL Trees and 2-3 Trees.

Table I
TIMINGS OF IMPLEMENTATION OF DIFFERENT ALGORITHM [6]

Implementation	Search	Insertion	Deletion
Skip List	0.051ms	0.065ms	0.059ms
AVL Trees	0.046ms	0.100ms	0.085ms
2-3 Trees	0.150ms	0.160ms	0.180ms

A node in the Skip List consist of a page address, page frequency and two pointers, pointing to previous and next page or node. In our method, we store page address shown in Fig. 3. And the node for level in the Skip List consist of a predefine frequency and three pointers, pointing to its upper layer, lower level and first node of their layer shown in Fig.4.

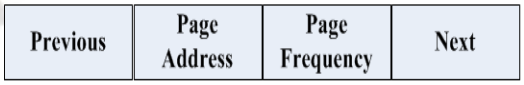

Fig. 3. The node structure for skip list node

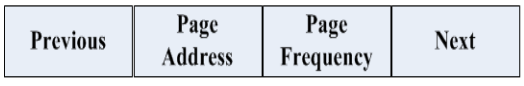

Fig. 4.The node structure of skip list level

In our modified skip list we insert the page at the head position in the bottom layer. Every time when page is accessed, its page frequency is increase

by one and it moves to head position of their layer. And if page frequency is exceeding than layer's predefined frequency, page is moves to the head position of the upper layer.

And delete operation is done at the tail position of bottom layer and Search operation starts from upper layer to bottom layer. In order to explain our method, suppose we insert 12 and search 32 and 45 from the given Skip List shown in Fig. 5.

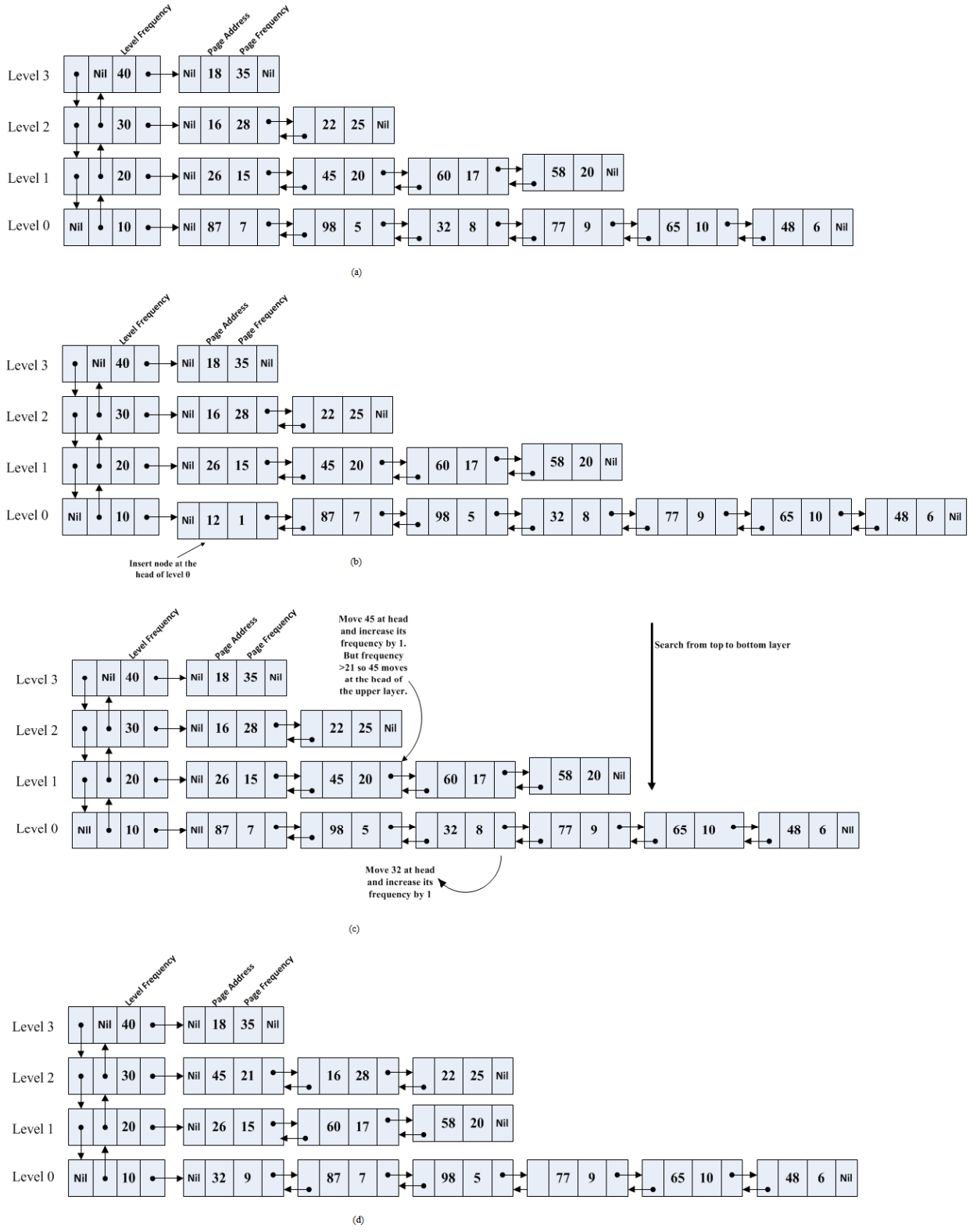

Fig 5. (a) The skip list representation (b) Insert 12 in the link list (c) Search 32 and 45 in the link list (d) After searching 32 and 45

C. ***LRU with Splay Tree***

A splay tree is a self adjustable binary search tree and it required O(log N) amortized time per operation. Here "amortized time" means the time per operation averaged over a worst case sequence of operations. For sufficiency long access sequence, splay trees are efficient. The efficiency of splay tree comes from applying a restructuring heuristic called splaying, whenever tree is accessed. Allen and Munro[16] and Bitner [17] proposed two restructuring heuristics shown in Fig. 9 and Fig. 10.

Single rotation. After accessing an element *i* in node *x*, rotate the edge joining *x* to its parent.

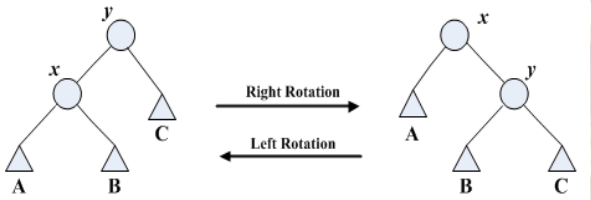

Fig. 6. Rotation of the edge joining nodes *x* and *y* (single rotation) [18]

Move to root. After accessing an element *i* in node *x*, rotate the edge joining *x* to its parent, and repeat this step until *x* is the root.

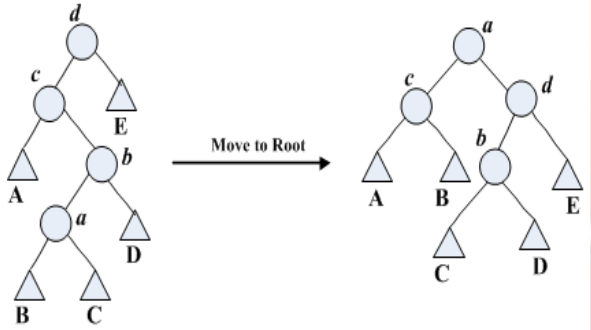

Fig. 7. The node accessed is *a* (move to root) [18]

Neither of this method heuristics is efficient. Therefore Daniel Dominic Sleator and Robert endre tarjan [18] apply following heuristic based on the structure of access path. To splay a tree at a node *x*, we repeat the following splaying step until *x* is the root of the tree shown in Fig. 11 [16,17,18].

Case 1 (zig). If *p(x)*, the parent of *x*, is the tree root, rotate the edge joining *x* with *p(x)*.
Case 2 (zig-zig). If *p(x)* is not the root and *x* and *p(x)* are both left or both right children, rotate the edge joining *p(x)* with its grandparent *g(x)* and then rotate the edge joining *x* with *p(x)*.
Case 3 (zig-zag). If *p(x)* is not the root and *x* is a left child and *p(x)* a right child, or vice versa, rotate the edge joining *x* with *p(x)* and then rotate the edge joining *x* with new *p(x)*.

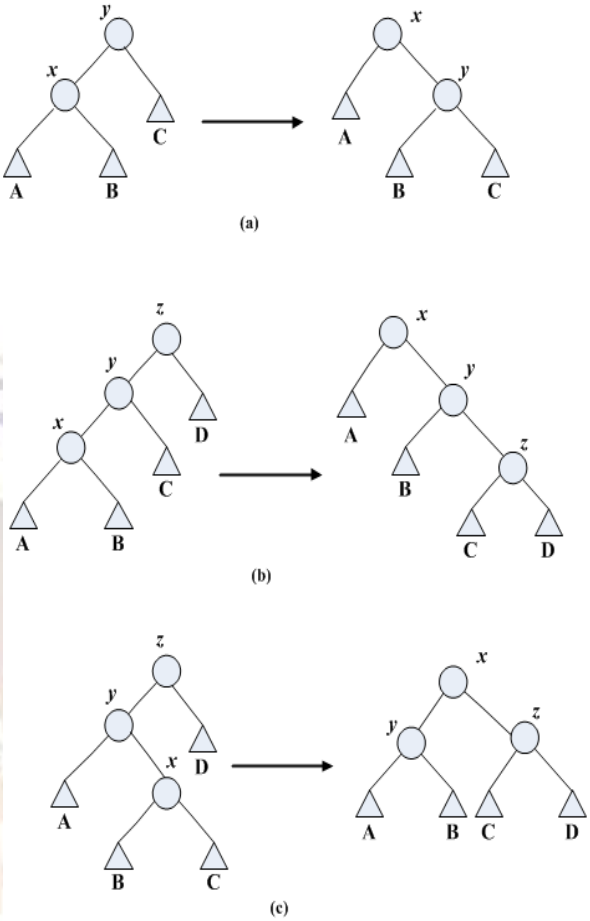

Fig. 8(a) Zig: terminating single rotation. (b) Zig-zig: two single rotation (c) Zig-zag: double rotation [18]

In splay trees, all the least recently pages are moved the root and near about root. Therefore we evict the page from the leaf the leaf nodes.

IV. PROPOSED ALGORITHM

A. ***LRU using Self Adjustable doubly Circular Link List***

```
PROCEDURE LRU_LinkList(page p)
    IF p available THEN
            CALL moveToFront(p)
    ELSE
            IF freeFrame THEN
                    CALL insertToFront(p)
            ELSE
                    CALL deleteAtLast()
                    CALL insertToFront(p)
            END IF
    END IF
END PROCEDURE
```

B. ***LRU using Skip List***

```
PROCEDURE LRU_SkipList(page p)
```

```
    IF p available THEN
                INCREMENT p_frequency
        IF  p_frequency  >  skipLevel_frequency
THEN
                CALL  moveUpperLayer(p)
        ELSE
                CALL  moveToFront(p)
        END IF
    ELSE
        IF freeFrame THEN
            CALL insertToFrontOfBottomLayer(p)
        ELSE
            CALL deleteAtLastOfBottomLayer()
            CALL insertToFrontOfBottomLayer(p)
        END IF
    END IF
END PROCEDURE
```

C. **LRU using Splay Tree**

```
PROCEDURE LRU_SplayTree(page p)
    IF p available THEN
        WHILE !(p is root) DO
                CALL splay(p)
        ENDWHILE
    ELSE
        IF freeFrame THEN
                CALL insertPage(p)
                WHILE !(p is root) DO
                        CALL splay(p)
                ENDWHILE
        ELSE
                CALL deleteAnyLeaf()
                CALL insertPage(p)
                WHILE !(p is root) DO
                        CALL splay(p)
                ENDWHILE
        END IF
    END IF
END PROCEDURE
```

V. SIMULATIONS AND RESULTS

To evaluate our page replacement algorithm experimentally, we simulated our policy and compared it with each other. The simulator program was design to run some trace file implement self adjustable circular link list, skip list and splay tree with different cache size. The obtained hit ratio depends on the data structure, cache size and the locality of reference for cache request.

A. Input traces

We used two traces to simulate our algorithm. Each trace is a hexadecimal address of a running program, taken from the SPEC benchmarks. According to traces that have been used, we considered 3 cache sizes and ran the simulator program to test the performance of our proposed algorithm.

B. Simulation Result

We executed out simulation program for case gcc trace, with all 3 different cache sizes and compared with each other. When cache size is 128 then skip list perform 1% and 2% better than self adjustable doubly circular link list and splay tree respectively shown in fig. 9. In case of 256 cache sizes skip list 2% better than self adjustable link list and 3.5% better than splay tree shown in fig. 10. And if cache size is 512 then skip list is 4% better than self adjustable link list and 4% better than splay tree. In some cases skip list perform 10% better than splay tree shown in fig. 11.

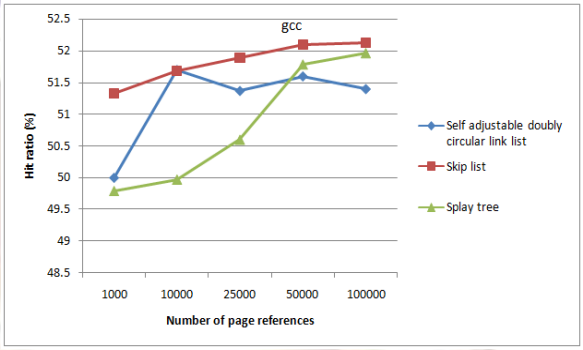

Fig. 9. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=128 for gcc trace.

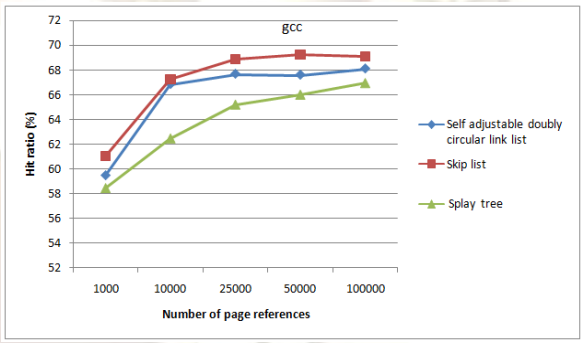

Fig. 10. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=256 for gcc trace.

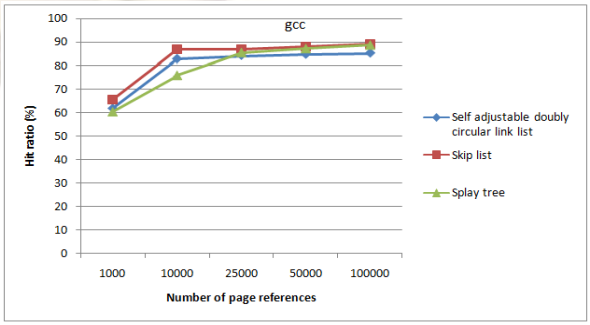

Fig. 11. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=512 for gcc trace.

Also we executed out simulation program for case swim trace, with all 3 different cache sizes and compared with each other. When cache size is 128 then skip list perform 3% and 4% better than self adjustable doubly circular link list and splay tree respectively shown in fig. 12. Some cases hit ratio is same for three different cache sizes. In case of 256 cache sizes skip list 2% better than self adjustable link list and 5% better than splay tree shown in fig. 13. And if cache size is 512 then skip list is 2.5% better than self adjustable link list and 5% better than splay tree. In some cases skip list perform 10% better than splay tree shown in fig. 14.

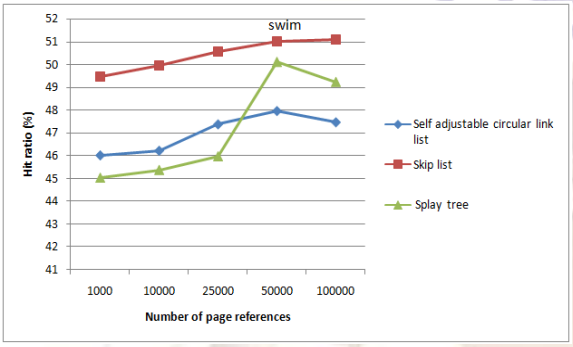

Fig. 12. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=128 for swim trace

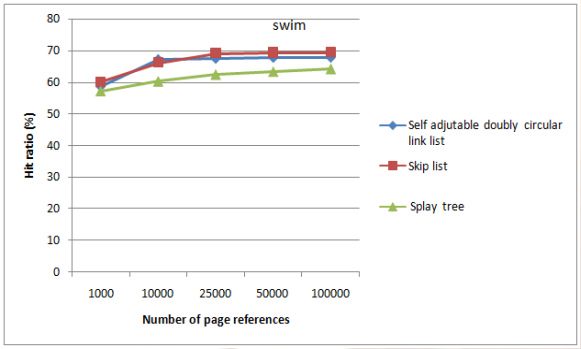

Fig. 13. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=256 for swim trace.

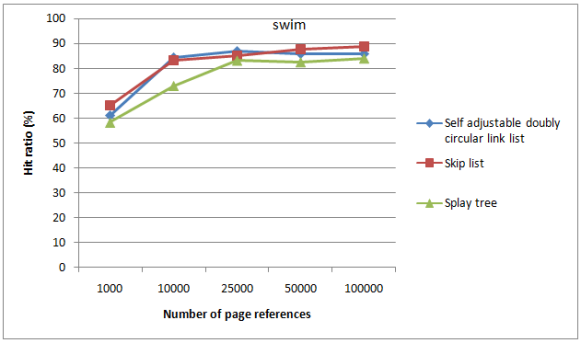

Fig. 14. Performance of LRU using self adjustable link list, skip list and splay tree with cache size=512 for swim trace.

VI. CONCLUSION

In this paper we have introduce a LRU page replacement algorithm with different data structures. We simulated LRU page replacement algorithm with self adjustable doubly circular link list, skip list and splay tree using two famous trace file gcc and swim.

Also in this paper we compare the result of these three data structure with LRU and we show that skip list is a best suitable data structure for LRU page replacement algorithm compare to self adjustable doubly circular link list and splay tree.

REFERENCES

[1] Debabrata Swain, Bancha Nidhi Dash, Debendra O Shamkuwar, Debabala Swain," Analysis and Predictability of Page Replacement Techniques towards Optimized Performance", *IRCTITCS*,2011, pp. 12-16.

[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, March-April 2012, pp. 126-130.

[3] Abraham Silberschatz, Peter B Galvin and Greg Gagne, "Virtual Memory," *Operating System Concept*, 8th ed., Wiley Student Edition, ch. *9*, pp. 315-370.

[4] Andrew S. Tanenbaum, "Memory Management," *Modern Operating System*, edition, year, Pearson Prentice Hall, 2008, ch. *3*, pp. 175-248.

[5] Kaveh Samiee and GholamAli Rezai Rad, "WRP: Weighting Replacement Policy to Improve Cache Performance," *International Symposium on Computer Science and its Application, IEEE*, 2008.

[6] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, Vol. 33, 1990.

[7] Jaafar Alghazo, Adil Akkaboune and Nazeih Botros, "SF-LRU Cache Replacement Algorithm," *MTDT, IEEE*, 2004.

[8] Debabala Swain, Bijay Paikaray and Debabrata Swain, "AWRP: Adaptive Weighting Replacement Policy to Improve Cache Performance," *IJournal of Computing, volume 3, Issue 2, February* 2011.

[9] Elizabeth J. O'Neil, Patrick E. O'Neil and Gerhard Weikum, "An Optimally Proof of the LRU-K Page Replacement Algorithm," *Journal of the ACM, vol. 46, No. 1, January 1999, pp. 92-112.*

[10] Donghee Lee, Jongmoo choi, Jong-Hun Kim, Sem H. Noh, Sang Lyul Min, Yookun Cho, Chong Sang Kim, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transaction on computer, vol. 50, no. 12, December 2001.*

[11] Sorav Bansal and Dharmendra S. Modha, "CAR: Clock with Adaptive Replacement," *USENIX File and Storage Technologies (FAST), March 31, San Francisco, CA.*

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

[13] Dan Wang and Jiangchuan Liu, "A Dynamic Skip List-Based Overlay for On-Demand Media Streaming with VCR Interactions," *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, VOL. 19, NO. 4, APRIL 2008, pp. 503-514.

[14] *Wikipedia. (2011, Jan 24). [Online]. Available:* http://en.wikipedia.org/wiki/List_of_data_structures

[15] J.H. Hester and D.S. Hirschberg, Self-Organizing Linear Search, *University of California*, Irvine.

[16] Allen, B., and Munro, "Self-organizing search trees," *J. ACM 25*, Oct. 1978, pp. 526-535.

[17] *BITNER, J.R., "Heuristics that dynamically organize data structures," SIAM J. Comput. 8,* 1979, pp. 82-1 10.

[18] Daniel dominic sleator and Robert endre tarjan, "Self-Adjusting Binary Search Trees," Journal of the Association for Computing *Machinery*, Vol. 32, No. 3, July 1985, pp. 652-686.