A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and AD-LRU for Flash Memory Based Systems

Dissertation

Submitted To

Central Department of Computer Science & Information Technology

Tribhuvan University

Kirtipur, Kathmandu

Nepal

In Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science & Information Technology

Submitted By

Dabbal Singh Mahara

March, 2014

Supervisor

Mr. Arjun Singh Saud

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Student's Declaration

I hereby declare that I am the only author of this work and that no sources other than the listed here have been used in this work.

… … … … … … …

Dabbal Singh Mahara

Date: 25 Feb, 2014

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

Supervisor's Recommendation

I hereby recommend that the dissertation prepared under my supervision by **Mr. Dabbal Singh Mahara** entitled "**A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and AD-LRU for Flash Memory Based Systems**" be accepted as in fulfilling partial requirement for the completion of Masters Degree of Science in Computer Science & Information Technology.

Mr. Arjun Singh Saud

Lecturer,
Central Department of Computer Science and Information Technology,
Institute of Science and Technology,
Kirtipur, Kathmandu, Nepal

Date: 25 Feb, 2014

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science and Information Technology

LETTER OF APPROVAL

We certify that we have read this dissertation work and in our opinion it is appreciable for the scope and quality as a dissertation in the partial fulfillment of the requirements of Masters Degree of Science in Computer Science & Information Technology.

Evaluation Committee

Asst. Prof. Nawaraj Paudel
Head of Department
Central Department of Computer Science
& Information Technology
Tribhuvan University
Kirtipur

Mr. Arjun Singh Saud
Lecturer
Central Department of Computer Science
and Information Technology
(Supervisor)

(External Examiner)

(Internal Examiner)

Date: 6 March, 2014

Acknowledgement

I would like to express my gratitude to all the people who supported and accompanied me during the preparation of this dissertation **"A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and AD-LRU for Flash Memory Based Systems".** This research work has been performed under Central Department of Computer Science and Information Technology (***Tribhuvan University***), Kirtipur. I am very grateful to my department for giving me an enthusiastic support.

First, I would like to express my gratitude to my supervisor **Mr. Arjun Singh Saud,** who gave me an enthusiastic support from the beginning to the end of the preparation of this dissertation. He is the one who listened to all my problems I faced during this thesis and showed me the way to overcome them.

Most importantly I would like to thank to respected Head of Department of Central Department of Computer Science and Information Technology, Asst. Prof. Nawaraj Paudel for his kind support, help and constructive suggestions. I am very much grateful and thankful to all the respected teachers Prof. Dr. Shashidharram Joshi, Prof. Sudarsan Karanjit, Prof. Dr. Subarna Sakya, Mr. Min Bahadur Khati, Mr. Bishnu Gautam, Mr. Jagdish Bhatta, Mr. Dheeraj Kedar Pandey, Mr. Sarbin Sayami, Mrs. Lalita Sthapit, Mr. Yog Raj Joshi and Mr. Bikash Balami of CDCSIT, TU, for providing me such a broad knowledge and inspirations.
Special thanks to my family and members of educational organizations that I have been working, for their endless motivation, constant mental support and love which have been influential in whatever I have achieved so far. All my class fellows are worthy of my gratefulness for their direct or indirect support in completion of my dissertation. Finally, I would like to thank my friends Mr. Dipak Prasad Bhatt and Mr. Bhupendra Singh Saud for their kind co-operation during my work.

I have done my best to complete this research work. Suggestions from the readers are always welcomed, which will improve this work.

Abstract

Flash memory has characteristics of asymmetric I/O latencies for read, write and erase operations and out-of-place update. Thus, buffering policy for flash based systems has to consider these properties to improve the overall performance. Existing buffer replacement algorithms such as LRU, LIRS, ARC etc do not deal with differing I/O latency of flash memory. Therefore, these algorithms have been revised to make them suitable for buffering policy for flash based systems. Among different flash aware buffer replacement algorithms LIRS-WSR and AD-LRU are two new buffer replacement policies that can be suitable for flash based systems. LIRS-WSR enhances LIRS by reordering the writes of not-cold-dirty pages from the buffer cache to flash storage to focus on the reduction of number of write/erase operations as well as preventing serious degradation of buffer hit ratio. AD-LRU also focuses on improving overall performance of flash based systems by reducing number of write /erase operations and by retaining high buffer hit ratio. We evaluate these two different approaches with same objectives of improving buffering policy for flash based systems by using trace driven simulation.

When workload has high reference locality, AD-LRU has significantly superior performance than LIRS-WSR in terms of both hit rate and write count. AD-LRU has higher hit rate up to 22% and minimizes write count up to 40% in comparison to LIRS-WSR. This is because of AD-LRU's good adaptive technique to handle changes in reference patterns.

For uniformly distributed workloads, the difference in hit rates and write count of AD-LRU and LIRS-WSR is comparatively small. AD-LRU outperforms LIRS-WSR by increasing hit rate up to 5% and decreasing write count up to 3% in comparison to LIRS-WSR in its worst case.

Keywords: Flash memory, Buffer Replacement Algorithm, LIRS, LIRS-WSR, AD-LRU, Hit Rate, Write Count

Table of Contents

CHAPTER 1

Background & Problem Formulation

1.1. Background 1-9
1.1.1. Memory Hierarchy 1
1.1.2. Flash Memory 2-4
1.1.3. Virtual Memory 5
1.1.4. Memory Management 5-8
1.1.4.1. Paging 5
1.1.4.2. Paging Algorithms 6
1.1.4.2.1. Fetch Algorithm 6
1.1.4.2.2. Placement Algorithm 6
1.1.4.2.3. Replacement Algorithm 7
1.1.4.3. Replacement Policy in Flash Memory Based Systems 8
1.1.5. Performance Metrics 8-9
1.1.5.1. Page Fault Count 8
1.1.5.2. Hit Rate/Miss Rate 8
1.1.5.3. Write Count 9
1.1.6. Program Behavior 9
1.1.6.1. Locality of Reference 9
1.2. Problem Formulation 9-13
1.2.1. LIRS-WSR 10
1.2.2. AD-LRU 11
1.2.3. Problem Statement 13
1.2.4. Objectives 13
1.3. Motivation 13
1.4. Thesis Organization 14

CHAPTER 2

Literature Review & Methodology

2.1 Literature Review 15-21
2.1.1. Traditional Buffer Replacement Algorithms 15-19
2.1.1.1. OPT or MIN Page Replacement Algorithm 15
2.1.1.2. FIFO Page Replacement Algorithm 15
2.1.1.3. LRU Page Replacement Algorithm 15
2.1.1.4. NRU Page Replacement Algorithm 16
2.1.1.5. LFU Page Replacement Algorithm 16
2.1.1.6. EELRU Page Replacement Algorithm 16
2.1.1.7. LRFU Page Replacement Algorithm 17
2.1.1.8. LRU-K Page Replacement Algorithm 17
2.1.1.9. 2Q Page Replacement Algorithm 17
2.1.1.10. LIRS Page Replacement Algorithm 18
2.1.1.11. ARC Page Replacement 18
2.1.1.12. Clock Based Page Replacement Algorithms 18
2.1.2. Buffer Replacement Algorithms for Flash-Based Systems 19-20
2.1.2.1. CFLRU 19
2.1.2.2. CFDC 19
2.1.2.3. LRU-WSR 20
2.1.2.4. CCFLRU 20
2.1.2.5. LIRS-WSR 21
2.1.2.6. AD-LRU 21
2.2. Research Methodology 21

CHAPTER 3

Program Development

3.1. Development Methodology & Tools 22
3.2. LIRS-WSR 22-30
3.2.1 Data Structure 24
3.2.2 Stack Pruning 24
3 .2.3 Algorithm 24-26
3.2.4 Flowchart 27
3.2.5 Tracing 28-30
3.3. AD-LRU 30-35

3.3.1 Data Structure 31
3.3.2 Algorithm 31-32
3.3.3 Flowchart 33
3.3.4 Tracing 34-35

CHAPTER 4

Test Results & Analysis

4.1 Data Collection 36
4.2 Testing 37-39
4.2.1 Test Result Workload 1 37
4.2.2 Test Result Workload 2 38
4.2.3 Test Result Workload 3 38
4.2.4 Test Result Workload 4 39
4.3 Analysis 39-45
4.3.1 Hit Rate Analysis 39-42
4.3.2 Write Count Analysis 42-45

CHAPTER 5

Conclusion & Future Work

5.1 Conclusion 46
5.2 Limitations and Future Work 47

References 48-49
Bibliography 50
Appendices 51-64

List of Figures

Fig. No.		Caption	Pages
Fig 1.1	-	Computer Memory Hierarchy	1
Fig 1.2	-	The Architecture of NAND Flash Memory System	3
Fig 1.3	-	Two Lists of LIRS Algorithm	10
Fig 1.4	-	Double LRU queues of the AD-LRU algorithm	12
Fig 3.1	-	General LIR HIR Transition Diagram	22

Fig 3.2	-	Specific LIR vs. Resident HIR Transition Diagram	22
Fig 3.3	-	LIR & Non Resident HIR Transition Diagram	23
Fig 3.4	-	Structure of a node in LIRS-WSR Data Structure	24
Fig 3.5	-	Flowchart of LIRS –WSR Algorithm	27
Fig3.6	-	State at Virtual Time 1-9	28-30
Fig.3.7	-	Structure of node of ADLRU Data Structure	31
Fig3.8	-	Flowchart of AD-LRU Algorithm	33
Fig3.9	-	State at Virtual Time 1-9	34-35
Fig4.1	-	Graph of Hit Rate for Workload1	39
Fig4.2	-	Graph of Hit Rate for Workload2	40
Fig4.3	-	Graph of Hit Rate for Workload3	40
Fig 4.4	-	Graph of Hit Rate for Workload4	41
Fig 4.5	-	Graph of Write Count for Workload1	42
Fig 4.6	-	Graph of Write Count for Workload2	43
Fig 4.7	-	Graph of Write Count for Workload3	43
Fig 4.8	-	Graph of Write Count for Workload4	44

List of Tables

Table No.		Caption	Pages
Table 1.1	-	Characteristics of flash memory	2
Table 4.1	-	Simulated trace for Random access	36
Table 4.2	-	Simulated trace for Read-most access	36
Table 4.3	-	Simulated trace for Write-most access	37
Table 4.4	-	Simulated Zipf trace	37
Table 4.5	-	Analysis of Workload 1	37
Table 4.6	-	Analysis of Workload 2	38
Table 4.7	-	Analysis of Workload 3	38
Table 4.8	-	Analysis of Workload 4	39

List of Abbreviations

2Q	-	Two Queue
AD-LRU	-	Adaptive Double Least Recently Used
ARC	-	Adaptive Replacement Cache
CAR	-	Clock with Adaptive Replacement
CFLRU	-	Clean First Least Recently Used
CCFLRU	-	Cold Clean First Least Recently Used
CFDC	-	Clean First Dirty Clustered
CLOCK Pro	-	Clock with Pro
CPU	-	Central Processing Unit
DBMS	-	Database Management System
EELRU	-	Early Eviction Least Recently Used
EEPROM	-	Electrically Erasable Programmable Read Only Memory
FC	-	First Clean
FIFO	-	First In First Out
FTL	-	Flash Translation Layer
HIR	-	High Inter-reference Recency
HIRS	-	High Inter-reference Recency Set
IRR	-	Inter- Reference Recency
KB	-	Kilo Byte
LFU	-	Least Frequently Used
LIR	-	Low Inter-reference Recency
LIRS	-	Low Inter-reference Recency Set
LIRS-WSR	-	Low Inter-reference Recency Set Write Sequence Reordering
LRFU	-	Least Recently Frequently Used
LRU	-	Least Recently Used
LRU-WSR	-	Least Recently Used Write Sequence Reordering
mA	-	mili Ampere
MMU	-	Main Memory Unit
MRU	-	Most Recently Used
NRU	-	Not Recently Used
OLTP	-	Online Transaction Processing
OPT or MIN	-	OPTimum or MINimum

OS	-	Operating System
PC	-	Personal Computer
PDA	-	Personal Digital Assistant
RAM	-	Random Access Memory
ROM	-	Read Only Memory
SRAM	-	Static Random Access Memory
WSR	-	Write Sequence reordering

Chapter 1

BACKGROUND & PROBLEM FORMULATION

1.1 Background

1.1.1 Memory Hierarchy

The evolution of computer from one generation to next generation shows variation not only in processing capabilities, but also in storage capabilities. The varieties of memory devices which vary on response time, cost, reliability, memory capacity etc. are available in today's market. Memory is an important and a very limited resource in a computer. Computer system has to achieve higher performance with in the limited storage capacity. The memory hierarchy system consists of all storage devices employed in a computer system from the slow high-capacity auxiliary memory to a relatively faster main memory, to an even smaller and faster cache memory.

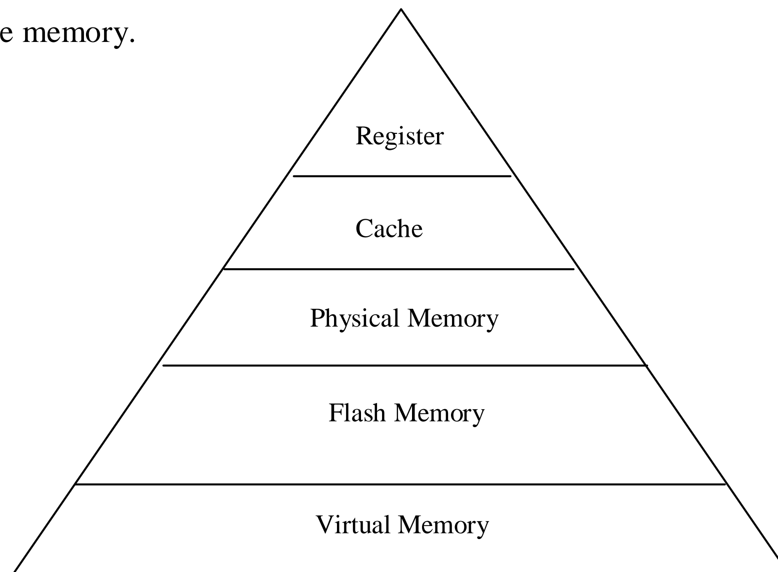


Fig 1.1 Computer Memory Hierarchy

Figure1.1 shows the hierarchy of memories used in a computer system having different speed and memory capacity. The arrangement of memory devices in a computer system is such that faster memory is at top level and slower memory is at the bottom. Overall performance of computer system depends upon management and organization of such memories. All the memory management policies are automatically handled by OS and devices are arranged according as the principles followed by it. Different types of memories available up to now can be categorized into two major groups. They are primary memory and secondary memory which can be taken as real memory. Besides real memory OS uses virtual memory to speed up the overall performance of the computer system.

1.1.2 Flash Memory

Flash memory is an electronic non-volatile computer storage medium that can be electrically erased and reprogrammed. Flash memory was developed by Intel and Toshiba in 1980s. Flash memory has been gaining popularity in mobile embedded systems as non-volatile storage due to its characteristics such as small and lightweight form factor, solid-state reliability, and low power consumption. The emergence of single flash memory chip with several gigabytes capacity makes a strong tendency to replace magnetic disk with flash memory for the secondary storage of mobile computing devices such as tablet PCs, PDAs, and smart phones [12].

Device	Current(mA)		Access time(4kB)		
	Idle	Active	Read	Write	Erase
NOR	0.03	32	20 μs	28ms	1.2 sec
NAND	0.01	10	25μs	250 μs	2 ms

Table 1.1 Characteristics of flash memory [13]

The term "flash" is said to have originated from the observation that it can write a sector of data usually 512 bytes, also called as page, or erase blocks of multiple pages usually 16 or 32 sectors simultaneously in one action, in contrast to the byte-by-byte EEPROM. This form of solid state technology differs from mechanical storage like standard hard drives in which information is stored using magnetism. Depending on the logic gate type used, the flash memory can be divided into two types: NOR and NAND. NOR flash, developed by Intel, is a random-access device, like RAM, that is directly addressable by the processor, and so it is good for executing program code. The most common type of flash memory in use today is NAND. This name is taken from the electronic logical gate NAND operator because flash memory uses floating gate MOSFET transistors that are arranged in a similar way. NAND flash is not directly addressable and is controlled using an indirect disk I/O-like interface through a bus to an internal command and address register. NAND flash requires fewer gates than NOR to store the same number of bits, and so it is smaller and denser and thus is appropriate for large data storage.

Flash memory uses floating gate transistors. These are arranged in a grid. Rather than a typical transistor that has one gate, flash NAND memory has two gates. Having two gates makes it possible to 'store' a voltage between the two gates so that it doesn't drain away, this

is very important and makes any information stored non-volatile. In fact, this 'trapped' voltage which represents information on the chip can stay in a locked state for many years or until we erase the memory. Information stored is erased by draining the voltage away from between the two gates by using the special floating gate feature that is unique to flash memory technology. Advantage of the flash memory comes from the fact that it is an electronic device, unlike the hard disk which is electromechanical and requires disk head and arm movement. This advantage frees the flash memory from the time-consuming seek and rotational delay. Even in high-end applications, flash memory can be arrayed together to offer capacity comparable to that of hard drives at higher speeds [22].

Usually one block consists of 32 sectors each with 512 bytes, and thus its size is usually 16 Kbytes. Such flash memory is called small block NAND flash. Flash memory vendors have started producing large block NAND flash with blocks of 64 sectors and sectors of 2,212bytes (thus, the size of a block is 128Kbytes) in order to allow faster write and erase operations for high-end applications. There are only three basic operations in a NAND flash: read a page, write a page, and erase a block. A read or write command specifies chip#, block#, sector#, where chip# is the flash chip number, block# the block number in the device, and sector# the sector number in the block [22].

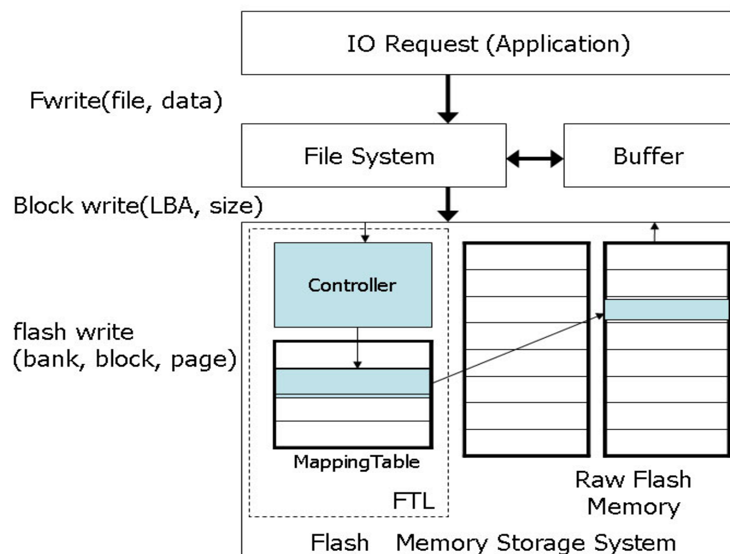


Fig.1.2 The Architecture of NAND Flash Memory System [8]

Figure 1.2 shows the general organization of a NAND flash memory system and the position of the FTL within it. A NAND flash memory system consists of one or more flash memory

chips, a controller that executes the FTL code in ROM, an SRAM (static RAM) that maintains the address mapping information. The host system views the flash memory as a hard disk-like device, and thus issues read or write commands along with logical sector addresses and data. In order to make the flash memory appear to applications as a disk drive, the flash translation layer (FTL) has been developed. The FTL translates the commands into low-level operations, namely read, write and erase, using physical sector addresses. To do the address mapping, the FTL looks up the address mapping information in the SRAM.

The flash memory has characteristics that profoundly affect its performance in managing data. Flash memory usually consists of many blocks and each block contains a fixed set of pages. Read/write operations are performed on page granularity, whereas erase operations use block granularity. The characteristics of flash memory are significantly different from magnetic disks. First, flash memory has no latency associated with the mechanical head movement to locate the proper position to read or write data. Second, flash memory has asymmetric read and write operation characteristic in terms of performance and energy consumption. Table 1.1 compares the access time and the energy consumption in flash memory when 4KB data is read, written, or erased. Third, flash memory does not support in-place update; the write to the same page cannot be done before the page is erased. Thus, as the number of write operations increases so does the number of erase operations. Erase operations are slow and power-wasting that usually decreases system performance. Finally, blocks of flash memory are worn out after the specified number of write/erase operations. Therefore, erase operations should be avoided for better performance and longer flash memory lifetimes. To avoid wearing specific segments out which would affect the usefulness of the whole flash memory, data should be written evenly to all segments. This is called even wearing or wear-leveling. Buffer replacement algorithms used in an OS or a DBMS in general assume that the speed of the read and write operations are about the same, which is true in the case of hard disks. The different characteristics of flash memory make it infeasible for system developed for hard disks as secondary storage to readily be used for the flash memory, and therefore force a reexamination of many key parts of the system architecture. Traditional performance metric such as 'buffer hit ratio' is not sufficient as performance indicator for flash based system. One naïve guide for this scheme may be stated as follows: "Try to reduce the number of writes/erases at the expense of the read operations." A new performance metric such as write count is also needed, in addition to the 'buffer hit ratio' [11].

1.3 Virtual Memory

When the system demands more memory to load a program, it may not find enough space in the memory. One of the techniques used to replace some part of the program to disk to free space for new program to be loaded. A space separated in the hard disk which holds the swapped out part. The identified least used pages are swapped out to these places of Hard-disk. This part of Hard-disk is known as Virtual memory because it behaves like main memory but actually it is secondary memory.

Virtual memory acts as a buffer between main memory and secondary memory. Data is fetched in advance from the secondary memory into the main memory so that data is available in the main memory when required. The benefit is that the large access delays in reading data from secondary storage are avoided. Fotheringham 1961[2], devised a concept of virtual memory which is associated with ability to address a memory space much larger than the available real memory. Virtual memory is a service provided by an OS that allows execution of programs larger than available physical memory. Virtual memory plays vital role to overcome limited primary memory. Handling virtual memory is one of the important issues of today's computer system.

1.1.4 Memory Management

Memory management and organization has been one of the most important factors that influence performance of OS. It has been studied from many years ago. Actually memory management is done by memory manager or memory management unit, which is handled by OS to manage memory hierarchy. The main job of memory management unit is to keep track of processes currently being executed. It keeps track which part of memory is currently in use and which is not. It also allocates memory for a process when required and deallocates memory when work is temporarily finished. It manages memory for a process to load and also manages extra memory that is virtual memory if it is too small to hold for the required process.

1.1.4.1 Paging

Paging is one of the techniques that organize virtual storage. The address referenced by running process is called virtual or logical address whereas the range of address it can reference is called virtual or logical address space. The address available in primary storage is called real or physical address whereas the available range of address is called real or physical address space. Even though a process references only virtual address, the process

must run on available real storage. So for every reference Memory Management Unit (MMU) maps logical address into corresponding available physical address for that page table is maintained. The operating system divides virtual address space into units called pages. Main memory is also divided to fixed size units called page frames. Generally, size of frame is equals to size of page. If a requested page is unavailable in primary storage page fault occurs. Each used page can be either in secondary memory or in a page frame in main memory [2].
A paging algorithm is needed to manage paging. A paging algorithm consists of three algorithms: placement algorithm, fetch algorithm and replacement algorithm. The placement algorithm is used to decide on which free page frame a page is placed. The fetch algorithm decides on which page or pages are to be put in main memory. Finally, the page replacement algorithm decides on which page is swapped out [9].

During page fault MMU notices that the page is unmapped and causes the CPU to trap. Trap is generated by OS to stop CPU until required page is not available. Then OS picks little used page frame as chosen by page replacement policy. If it has dirty bit then the contents are written otherwise if it has clean bit then nothing is written back to secondary storage. Thus the required page is placed into freed frame. Then after successful mapping, trap is restarted and the process is continued [1].

1.1.4.2 Paging Algorithm

1.1.4.2.1 Fetch Algorithm

Fetch algorithm initially identifies the requested page block. Paging algorithm can be categorized into two major groups. They are demand paging and anticipatory paging. Demand paging algorithm waits for a page requested by a running process. But anticipatory or pre-paging algorithm guesses which pages are needed before they are requested. Generally paging mechanism will not have prior knowledge of the page reference stream or the known order of pages requested in. This causes many systems to employ a demand fetch approach, where a page fault notification is the first indication that a page must be moved into the physical memory. Hence demand paging algorithm is much more effective in real systems than pre-paging algorithm. Demand fetching algorithm always fetches a page that has been requested during a page fault [9].

1.1.4.2.2 Placement Algorithm

Placement algorithm decides where to put the fetched page in available free storage. Initially, if placement algorithm allows fully associative then OS can place the requested page any

where using any algorithm like First Fit. After a cache is fulfilled then placement policy is static that means a requested page is placed in place of removed victim page. The page to be replaced is called victim page. A victim page is always replaced by required page which is chosen by replacement policy used in that particular system. In case of partially associative memory mapping, placement algorithm is restricted only for certain memory location.

1.1.4.2.3 Replacement Algorithm

Because the secondary storage, where the remaining pages are stored, has a low speed as compared to the speed of the main memory; the operating system uses different algorithms to replace pages in the primary storage. Replacement algorithm identifies the victim page and replaces it by fetched page because of lack of primary storage. After a primary storage is fulfilled one of the pages must be replaced for execution of the requested page. The replaced page is called victim block. There are many algorithms devised for page replacement. Optimal page replacement algorithm suggests replacing the page that will not be used for longest period of time in future. OS in the least recently used (LRU) algorithm tries to replace the blocks that have low probability of being referenced again and, it tries to retain those blocks which have high probability of being referenced in near future.

Locality of reference is one of the properties of page reference pattern, which is used by many algorithms to predict about the future references. We say a workload (sequence of page references) consists of locality of reference if many memory references are accesses to neighboring page of the page referenced just before it. A good approximation to the optimal algorithm is based on the observation that pages that have been used heavily in the recent past would probably be used again in near future.

Static page replacement algorithm shares frames equally among all processes such as FIFO, LRU, MRU, random, optimal etc. But dynamic page replacement algorithm shares frames according to need rather than equality among all processes such as working set page replacement algorithm.

Also page replacement algorithm can maintain global and local policy. Global policy selects a replacement from the set of all available frames. Local policy selects a replacement from the processes own set of frames. Local page replacement assumes some form of memory partitioning that determines how many pages are to be assigned to a given process.

1.1.4.3 Replacement Policy in Flash Memory Based Systems

Flash memory has characteristics of out-of-place update and asymmetric I/O latencies for read write and erase operations. Write/erase operations are relatively slow compared to read operations. Typically, write operations are about ten times slower than read operations, and erase operations are about ten times slower than write operations [13]. A buffer replacement algorithm for a disk tries to obtain the optimal I/O sequence from the original I/O sequence by reducing the number of accesses for the overall performance. Flash caching is needed for reducing flash I/O latencies. The traditional magnetic-disk-based buffering algorithms focus on hit-ratio improvement alone, but not on write costs caused by the replacement process. So, their straight adoption would result in poor buffering performance and would demote the development of flash-based systems. The replacement policy should minimize the number of write and erase operations on flash memory and at the same time prevent the degradation of the hit ratio.

1.1.5 Performance Metrics

Offline performance of buffer replacement algorithm is measured in terms of page fault count, hit rate/ hit ratio, miss rate/miss ratio and write count. When an accessed block of memory is currently mapped to the physical memory then hit occurs. If it doesn't map them miss occurs. Higher hit rate of the algorithm exhibits higher performance. In case of flash based system, higher hit rate and lesser number of write count is measure for better algorithm.

1.1.5.1 Page Fault Counts

An efficient page replacement algorithm always produces less number of page faults. It can be computed by counting the occurrences of number of page faults between some intervals of references.

1.1.5.2 Hit/Miss Rate

Hit rate can be calculated by using the formula: hr = 1 – mr, where hr is the hit rate and mr is miss rate. Hit rate is the percentage calculation of hit ratio. Hit ratio is calculated by subtracting miss ratio from 1.

Miss rate is calculated by using the formula:

mr = 100 x ((#pf - #distinct) / (#refs – #distinct)) where #pf is number of page faults, #distinct is the number of distinct pages referenced and #refs is the total number of

referenced pages [10]. Miss ratio is calculated by dividing total number of page faults by total number of references.

1.1.5.3 Write Counts

Write count is number of pages propagated to flash memory which can be calculated by counting the number of physical page writes to flash memory and at the end of each test the dirty pages in the buffer are flushed to the flash memory to get exact write counts.

1.1.6 Program Behavior

There are several factors that influence performance of page replacement algorithm. The performance of page replacement algorithm relies on pattern of pages that are referenced. Behavior of program depends upon the access pattern it references memory which is further depends upon working set and locality of reference.

1.1.6.1 Locality of Reference

During the course of execution of program memory references tend to cluster forming certain locality. Locality varies on the basis of time and space. Temporal locality is based on time, it assumes that memory location referenced just now is likely to be reference again in near future. Looping, subroutines, stacks, variable used for counting & totaling etc supports this assumption. Spatial locality is based on space, is assumes that once a memory is referenced there is high chance of nearby memory location to be referenced again. Array traversal, sequential code execution, related variable declaration nearby in source code supports this assumption. Hints of locality are followed in any type memory reference sequence. But some follow strongly and some follow weakly. Memory locations that are referenced repeatedly in a same order can be viewed as cyclic pattern. Loop generates cyclic pattern. Access of memory location at particular place then repeated after some duration, such memory reference pattern can be viewed as correlated pattern. Sequential Scan also generates correlated pattern. When particular memory block has a stationary reference probability and all other blocks are accessed independently with the associated probabilities, such memory reference pattern can be viewed as probabilistic pattern [10].

1.2 Problem Formulation

The traditional magnetic-disk-based buffering algorithms LRU [1], Clock [5] LIRS [19], ARC [15] etc focus on hit-ratio improvement alone, but not on write costs caused by the replacement process. So, their straight adoption would result in poor buffering performance

and would demote the development of flash-based systems. The replacement policy should minimize the number of write and erase operations on flash memory and at the same time prevent the degradation of the hit ratio. Recently, CFLRU [20], LIRS-WSR [8] and AD-LRU [16] were proposed as new buffering algorithms for flash-based systems. These new flash based buffer replacement policies consider not only buffer hit ratios but also replacement costs incurring when a dirty page has to be propagated to flash memory to make room for a requested page currently not in the buffer. These algorithms favor to first evict clean pages from the buffer so that the number of writes incurring for replacements can be reduced.

1.2.1 LIRS-WSR (Low Inter-reference Recency Set – Write Sequence Reordering)

LIRS-WSR algorithm [8] is designed for a buffer cache of the flash memory based storage system by enhancing LIRS algorithm with the application of WSR technique. It tries to minimize the write requests for generating optimal I/O sequence from the given I/O sequence to reduce the write cost and prevent the loss of hit ratio. The objective of LIRS-WSR is to reduce the number of flushes of dirty pages from the buffer into flash memory when page replacement occurs. To achieve this objective, it uses the strategy: delaying eviction of the page which is dirty and has high access frequency as possible.

The LIRS [19] algorithm uses history information of data accesses in the form of two metrics: the Inter-Reference Recency (IRR) and the Recency. The IRR of a data block refers to the number of other distinct blocks accessed between the last two consecutive accesses of the data block in question, while recency refers to the number of other distinct blocks accessed between the last reference to the current time.

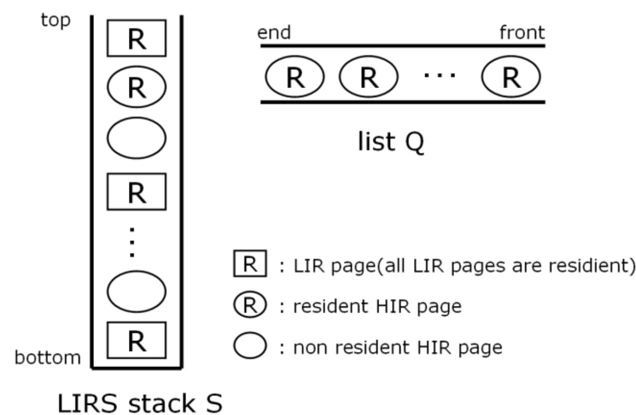


Fig.1.3 Two Lists of the LIRS algorithm [19]

Fig 1.3 shows two data structures LIRS Stack S and HIR queue Q used in LIRS algorithm. It uses two sets of pages based on IRR. Set of pages with low IRR value is taken as hot block and called low inter-reference recency set (LIRS). Set of pages with high IRR value is taken as cold block and called high inter-reference recency set (HIRS). Blocks that can be most probably used in future are taken as hot blocks whereas blocks that may not be used in near future are taken as cold blocks. Hence HIR blocks are always replaced and LIR blocks are never replaced. LIRS always selects HIR page with the largest recency as victim for replacement. LIRS stack contains all the LIR pages and some HIR resident or non-resident pages. HIR queue Q contains all the resident HIR pages only some of which may not be in LIRS stack. Non-resident HIR pages are those HIR pages which have been evicted from HIR queue but they are still in LIRS stack as metadata. WSR policy is a second chance policy in which, a page at the bottom of stack is checked to find whether it is cold or not. if it is found not-cold and dirty, then it's cold flag is set and moved to top of LIRS stack, next page is checked. Otherwise, the page is moved to the head of HIR queue, switching its status to HIR resident page.

LIRS-WSR uses a Stack S and HIR Q for the same purpose as in LIRS. Stack S contains all the pages either LIR, resident HIR or non-resident HIR pages. Q holds all the resident pages at some point of time. The operations on these two data structures are same as that of LIRS. Every page has additional status either cold or not-cold. Initially all pages are cold, this cold flag is cleared if the pages are referenced again when they are in stack S or queue Q. If a page is introduced to the buffer for write request for the first time, it becomes a dirty page and enters the top of the stack S as an LIR page. Every time when Stack bottom is moved to HIR Q, WSR policy is applied. That is, if bottom LIR page is dirty and not cold, then its cold flag is set and moved to the head of Stack, otherwise it is moved to the head of HIR Q. All other operations like pruning, switching between LIR and HIR pages are same as that of LIRS.

1.2.2 AD-LRU (Adaptive Double LRU)

AD-LRU algorithm [16] is buffer replacement algorithm for flash-based systems which focuses to reduce the write costs of the buffer replacement algorithm while keeping a high hit ratio. It tries to integrate the properties: recency, frequency, and cleanness of pages into the buffer replacement policy. AD-LRU has two LRU queues: Cold LRU queue and Hot LRU queue, to capture the concept of recency and frequency of the page references, among which Cold LRU queue stores the pages referenced only once and Hot LRU queue maintains the

pages that are referenced at least twice. The sizes of these two LRU queues are dynamically adjusted according to changes in reference patterns. When a page is first referenced, it is put in the head of cold LRU queue. The pages move from cold LRU queue to head of hot LRU queue when it is referenced again and when a page in hot LRU queue is selected as victim, it is demoted to head of cold LRU queue. During the eviction procedure, least recently used clean page from cold LRU queue is selected as a victim. There is a specific pointer FC (First Clean) to point least recently used clean page in each LRU queue. If clean pages do not exist in the cold LRU queue, second chance policy is applied. For this purpose, each page in double LRU queues is marked by a reference bit which is always set to 1, when the page is referenced. When a new page is referenced and buffer is full, the page pointed by FC pointer of cold LRU queue (Cold_FC) is evicted, if Cold_FC is not null. Otherwise dirty page at the tail of this queue is evicted if its reference bit is not set. Otherwise, the reference bit is cleared to 0 and moved to head of the same queue, process continues to find next dirty page with reference bit 0. The second-chance policy ensures that dirty pages in the cold LRU queue will not be kept in the buffer for an overly long period.

There is a parameter MIN_LC to set lowest limit of size of cold LRU queue. This limits size of cold LRU queue to prevent from frequent replacement of recent pages. This happens for too small size of cold LRU queue because pages are always inserted and removed from cold LRU queue. To prevent such frequent replacement of recent pages, the policy applied is: if the size of cold LRU queue reaches MIN_LC, victim will be selected from hot LRU queue rather than cold LRU queue. In this case victim page is selected in same way as in cold LRU queue but the victim page is moved to head of cold LRU queue.

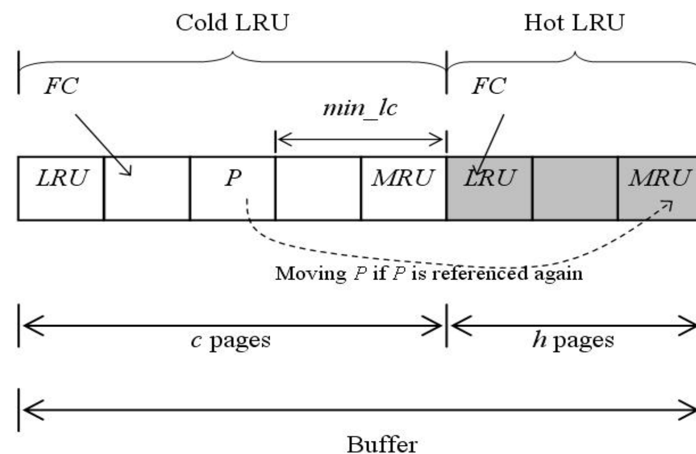


Fig.1.4 Double LRU queues of the AD-LRU algorithm [16]

1.2.3 Problem Statement

The replacement algorithm with flash memory should consider not only the hit rate but also the replacement cost caused by selecting dirty victim pages. **The evaluation of the buffer replacement algorithms for flash-based systems in terms of hit rate and write counts is required to compare their performance.** Comparison among several buffer replacement algorithms have been found in different research papers, but comparison between LIRS-WSR and AD-LRU has not been done yet. **This dissertation work will mainly focus on comparative evaluation of these two algorithms: LIRS-WSR and AD-LRU in terms of hit rate and write counts.**

1.2.4 Objective

The main objective of this dissertation work is:

- To perform comparative study of LIRS-WSR and AD-LRU buffer replacement algorithms for flash based systems in terms of hit rate and write counts for different workloads.

1.3 Motivation

Memory management is not only the burden of today's computing devices. It has been researched for decades. Whatever variety of storage devices found in today's market is the great achievement of computer science. But still computer memory is the limited source which directly hampers the performance of computing system. Performance gain can be achieved by increasing the capacity of primary storage. Expectation of customer is to decrease cost price with sufficient working memory. Hence to fulfill this demand for manufacturing such device fewer materials are used and size of memory is being decreased. But rather than this technical view, it is not possible to gain performance without managing memory logically for its usability. Varieties of techniques had been tried for this achievement. Among such techniques paging is the successful one. Page replacement algorithm is the main part of paging technique because deciding the victim page is a very tough job.

The emergence of single flash memory chip with several gigabytes capacity makes a strong tendency to replace magnetic disk with flash memory for the secondary storage of mobile computing devices. Most operating systems are customized for disk based storage systems and their replacement policies only concern the number of cache hits. However, the operating systems that consider flash memory as secondary storage should consider different read and

write cost of flash memory when they replace pages to reclaim free space. There are different buffer replacement algorithms proposed for flash based storage systems. Some of them consider recency factor only, some consider cleanliness, some both of these factors and some consider recency, cleanliness and frequency of page references as well. There are different papers that have compared the different algorithms for flash based buffer replacement. But, the comparison of LIRS-WSR and ADLRU algorithms has not been done yet. So, this dissertation work is trying to compare the performance metrics of these two algorithms for flash based storage context.

1.4 Thesis Organization

Background part of this dissertation work focuses on page replacement algorithm and the related basic terms which are already mentioned above along with an introduction to LIRS-WSR and AD-LRU. Some more chapters are remaining which clarifies the topics LIRS-WSR and AD-LRU fulfilling the objectives of this dissertation work. Chapter 2 consists of literature review which briefly reviews the related topics. Literature review includes summary of several traditional page replacement algorithms like Optimal, LRU, MRU, LRFU, 2Q etc and some flash based page replacement algorithms such as CFLRU, CCF-LRU, CFDC, LRU-WSR etc. This chapter also contains the research methodology part which shows the flow of our research. Chapter 3 consists of program development steps of our simulation. It includes detail design of the program. Also it includes details about the data structures and programming language used to develop simulator. Chapter 4 consists of data collection and analysis part which includes details about memory references that shows trace driven input, output results with several analyzing graphs which are tested for different workloads. Chapter 5 consists of conclusion of this whole dissertation work and the future work which shows guidelines for further research.

Chapter 2

LITERATURE REVIEW & METHODOLOGY

2.1 Literature Review

2.1.1 Traditional Buffer Replacement Algorithms

2.1.1.1 OPT or MIN Page Replacement Algorithm

Various memory management techniques have been used from the beginning for the improvement of performance. Belady [1] in 1966 developed optimal page replacement algorithm called OPT or MIN. His algorithm depends upon principle of optimality which states "To obtain optimal performance the page to replace is the one that will not be used again for the furthest time into the future." His optimal algorithm is not applicable for real implementation because our OS doesn't know which pages will be used before execution. Hence it can be only simulated due to lack of future knowledge. It is used as a benchmark for measuring effectiveness of other page replacement algorithms. OPT Replacement algorithm replaces page that will not be used for the longest period of time by computing maximum forward distance.

2.1.1.2 FIFO Page Replacement Algorithm

Fist-In-First-Out (FIFO) page replacement algorithm [1] replaces oldest page during page fault. Conceptually FIFO is a queue with limited size. Initially queue is filled by inserting page reference from the tail. When the queue is full new reference is inserted from tail and old reference is evicted from the head. FIFO is simple but suffers from Belady's Anomaly, a strange situation in which page fault increased while increasing number of page frame. That is, with increase in physical memory FIFO can decrease page fault performance seemingly at random. Like random page replacement algorithm, FIFO still does not take advantage of locality trends. But it can be modified very easily.

2.1.1.3 LRU Page Replacement Algorithm

This algorithm considers that a page that is recently used will probably be used again very soon, and a page that has not been used for a long time, will probably remain unused. This algorithm is purely based on recency of page references. Recency is evaluated by maintaining LRU stack that is a sorted list on the basis of virtual time, which is the only factor for replacement. When page fault occurs, the page that has been unused for the longest time is evicted. Thus LRU is simple but is not easy to implement without hardware support. It can

adapt faster according as program behavior. LRU like algorithm doesn't suffer from Belady's Anomaly as FIFO. It gives good approximation of optimal algorithm. Although LRU is theoretically realizable, it is not cheap. To fully implement LRU it is necessary to maintain a linked list of all pages in memory, with the most recently used page at head and least recently used page at the tail. The difficulty is that the list must be updated on every memory reference. Finding a page in the list, deleting it and then moving it to the head is a very time consuming operation.

2.1.1.4 NRU Page Replacement Algorithm

Pages are categorized into four classes in not recently used (NRU) algorithm. Class 0 contains pages that are neither referenced nor modified. Class 1 contains pages that are modified but not referenced. Class 2 contains pages that are referenced but not modified and Class 3 contains pages that are modified as well as referenced. During page fault NRU evicts any page from the lowest class [2].

2.1.1.5 LFU Page Replacement Algorithm

Least Frequently Used (LFU) [3] selects a victim page that has not been used often in the past. Instead of using a single recency factor as LRU, LFU defines additional information of frequency of use associated with each page. This frequency is calculated throughout the reference stream by maintaining counting information. Frequency count leads to serious problem after a long duration of reference stream. Because when the locality changes, reaction to such certain change will be extremely slow. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less than the pages associated with the previous program. Since the context has changed, the pages swapped out will most likely be needed again soon which leads to thrashing. One way to remedy this is to use a popular variant of LFU, which uses frequency counts of a page since it was last loaded rather than since the beginning of the page reference stream. Each time a page is loaded, its frequency counter is reset rather than being allowed to increase indefinitely throughout the execution of the program. LFU still tends to respond slowly to change in locality.

2.1.1.6 EELRU Page Replacement Algorithm

Some algorithms use recency as history information like LRU and Most Recently Used (MRU). LRU is suitable for good locality of reference whereas MRU is somewhat suitable

for weak locality of workloads. These two algorithms can be tuned to form adaptive algorithm called Early Eviction LRU (EELRU) [24], which was proposed as an attempt to mix LRU and MRU, based only on the positions on the LRU queue that concentrate most of the memory references. This queue is only a representation of the main memory using the LRU model, ordered by the recency of each page. EELRU detects potential sequential access patterns analyzing the reuse of pages. One important feature of this algorithm is the detection of non-numerically adjacent sequential memory access patterns. Two tunable parameters used are early eviction point and late eviction point. LRU queue concentrate most of the memory references when it reaches late eviction point.

2.1.1.7 LRFU Page Replacement Algorithm

Least Frequently Used (LFU) algorithm uses frequency factor for page replacement. LRU and LFU are tuned to form adaptive algorithm called Least Recently Frequently Used (LRFU) [3] that considers both recency and frequency factors. The performance of the LRFU algorithm largely relies on a parameter called, λ, which determines the relative weight of LRU or LFU and has to be adjusted according to the system configuration, even according to different workloads [6].

2.1.1.8 LRU-K Page Replacement Algorithm

LRU - K [4] evicts the page that is the one whose backward K-distance is the maximum of all pages in buffer. Backward K-distance bt(p,K) can be defined as the distance backward to the Kth most recent reference to page p where reference string known up to time t (r1, r2, …,rt). The value of parameter K can be taken as 1, 2 or 3. If K=1, it works as simple LRU algorithm. Highly increasing value of K the overall performance of algorithm reduces. LRU-K can discriminate better between frequently referenced and infrequently referenced pages. Unlike the approach of manually tuning the assignment of page pools to multiple buffer pools, LRU-K does not depend on any external hints. Unlike LFU and its variants, this algorithm copes well with temporally clustered patterns.

2.1.1.9 2Q Page Replacement Algorithm

2Q [21] algorithm quickly removes sequentially and cyclically referenced block with after a long interval. The algorithm uses special buffer queue A1in of size Kin, ghost buffer queue A1out of size Kout and the main buffer Am. Special buffer contains all missed that is first time referenced block. Ghost buffer contains replaced blocks from special buffer. Frequently

accessed block are available in main buffer. Hence victim blocks are always from special buffer and main buffer.

2.1.1.10 LIRS Page Replacement Algorithm

Another important algorithm is LIRS which is already described in section 1.2. Its objective is to minimize the deficiencies presented by LRU using an additional criterion named IRR (Inter- Reference Recency) that represents the number of different pages accessed between the last two consecutive accesses to the same page. This means that LIRS does not replace the page that has not been referenced for the longest time, but it uses the access recency information to predict which pages have more probability to be accessed in a near future.

2.1.1.11 ARC Page Replacement Algorithm

Adaptive Replacement Cache (ARC) [15] improves the LRU strategy by splitting the cache directory into two lists, T1 and T2, for recently and frequently referenced entries. In turn, each of these is extended with a ghost list (B1 or B2) which is attached to bottom of these two lists. These ghost lists act as score cards by keeping track of the history of recently evicted cache entries, and the algorithm uses ghost hits to adapt to recent change in resource usage. The ghost lists only contain metadata (keys for the entries) and not the resource data itself, i.e. as an entry is evicted into a ghost list its data is discarded. The combined cache directory is organized in four LRU lists:

i. T1, for recent cache entries.
ii. T2, for frequent entries, referenced at least twice.
iii. B1, ghost entries recently evicted from the T1 cache, but are still tracked.
iv. B2, similar ghost entries, but evicted from T2.

T1 and B1 together are referred to as L1, a combined history of recent single references. Similarly, L2 is the combination of T2 and B2.

2.1.1.15 CLOCK Based Page Replacement Algorithm

The clock-based approximations, such as CLOCK [5], CLOCK-PRO [18], and CAR [17], usually cannot achieve the high hit ratio compared to their corresponding original algorithms (LRU, LIRS, ARC respectively). They organize buffer pages into circular list, and use a reference bit or a reference counter to record access information for each buffer page. When a page is hit in the buffer, the clock-based approximations set the reference bit or increment the counter, instead of modifying the circular list themselves. However, the clock-based

approximations can record only limited history access information, i.e. whether a page has been accessed or how many times it has been accessed but not in what order the accesses occur. The lack of richer history information can hurt their hit ratios. Moreover, many sophisticated replacement algorithms do not have clock based approximations since the access information they need cannot be approximated by the clock structure.

2.1.2 Buffer Replacement Algorithms for Flash-Based Systems

2.1.2.1 CFLRU

Clean First LRU (CFLRU) [20] is the first algorithm designed for flash based systems. It modified the LRU policy by introducing a clean first window W, which starts from the LRU position and contains the least recently used w*B pages, where B is the buffer size and w is the ratio of the window size to buffer size. When victim is selected, CFLRU first evicts least recently used clean pages in W. Hence it reduces the number of write operations because clean page is not propagated to flash memory. If no clean page is found, then it behaves like LRU policy. CFLRU has some problems such as clean-first window size is to be tuned to the current workload and cannot suit for differing workloads and it always replaces clean pages first, which causes the cold dirty pages residing in the buffer for long time and, in turn, results in suboptimal hit ratio. The window size, W, can be tuned statically or dynamically. In this sense, CFLRU is known as CFLRU- static or CFLRU- dynamic. In paper [20], CFLRU-static and CFLRU-dynamic has been compared with LRU policy for five different workloads. They found result that CFLRU static and dynamic reduces the replacement cost by 28.4 % and 23.1% for swap system buffer cache and 26.2% and 23.5% for file system buffer cache with compared to LRU.

2.1.2.2 CFDC

Clean First Dirty Clustered (CFDC) [23] manages the buffer in two regions: the working region W for keeping hot pages that are frequently and recently revisited, and the priority region P responsible for optimizing replacement costs by assigning varying priorities to page clusters. A parameter λ, called priority window, determines the size ratio of P relative to the total buffer. Therefore, if the buffer has B pages, then P contains λ pages and the remaining (1-λ)*B pages are managed in W. Various conventional replacement policies can be used to maintain high hit ratios in W and, therefore, prevent hot pages from entering P. CFDC improves the efficiency of buffer manager by flushing pages in clustered fashion based on the observation that flash writes with strong spatial locality can be served by flash disks more

efficiently than random writes. In paper [23] CFDC has been compared with LRU and CFLRU for different four workloads in database engine. The results show CFDC outperforms both competing policies, with a performance gain between 14% and 41% over CFLRU., in turn, is only slightly better than LRU with a maximum performance gain of 6%.

2.1.2.3 LRU- WSR

LRU-WSR [7] is a flash-aware algorithm based on LRU and Second Chance [5], using only a single list as auxiliary data structure. The idea is to evict clean and cold-dirty pages and keep the hot-dirty pages in buffer as long as possible. When a victim page is needed, it starts searching from the LRU end of the list. If a clean page is found, it will be returned immediately (LRU and clean-first strategy). If a dirty page marked as "cold" is found, it will also be returned; otherwise, it will be marked "cold" (Second Chance), moved to the MRU (most-recently used) end of the list, and the search continues. Although LRU-WSR considers the hot/cold property of dirty pages, which is not tackled by CFLRU, it has high dependency on the write locality of workloads. It shows low performance in case of low write locality, which may cause dirty pages to be quickly evicted. In paper [7], LRU-WSR has been compared with LRU, CFLRU algorithms for different workloads collected from PostgreSQL, GCC, Viewperf and Cscope. LRU-WSR has been found 1.4 times faster than LRU. In most of the cases, LRU-WSR has higher hit ratio and lower write count than others.

2.1.2.4 CCF-LRU

The authors of CCF-LRU [25] further refine the idea of LRUWSR by distinguishing between cold-clean and hot clean pages. It maintains two LRU queues, a cold clean queue and a mixed queue to maintain buffer pages. The cold clean queue stores cold clean pages (first referenced pages) while mixed queue stores dirty pages or hot clean pages. It always selects victim from cold clean queue and if cold clean queue is empty then employs same policy as that of LRU-WSR to select dirty page from mixed queue. This algorithm focuses on reference frequency of clean pages and has little consideration on reference frequency of dirty pages. Besides, the CCF-LRU has no mechanism to control length of cold clean queue, which will lead to frequent eviction of recently read pages in the cold clean queue and lower the hit ratio. In paper [25] CCF- LRU has been compared with LRU, CFLRU and LRU-WSR with different four workloads. The results show that CCF-LRU performs better than LRU, CFLRU, and LRU-WSR with respect to hit rate, write count and run time.

2.1.2.5 LIRS-WSR

LIRS-WSR which is already explained in section 1.2.1, is an improvement of LIRS so that it can suit the requirements of flash-based systems. It integrates write sequence reordering (WSR) technique to original LIRS algorithm to reduce the number of page writes to flash memory. In paper [8] LIRS-WSR has been compared with LRU, CFLRU, LIRS and ARC for four different workloads: PostgreSQL, gcc, Viewperf and Cscope. LIRS-WSR has hit ratio very close approximate to LIRS and has higher hit ratio than other algorithms. LIRS-WSR has minimum write count than all other algorithms. In case of run time, LIRS-WSR is 2 times faster than LRU and 1.25 times faster than LIRS algorithm.

AD-LRU

Adaptive double-LRU (AD-LRU) , which is already explained in section 1.2.2, takes into account both reference frequency of clean pages and dirty pages and has a new mechanism to control the length of cold queue to avoid drop in hit ratio. In paper [16] AD-LRU has been compared with LRU, CFLRU, LRU-WSR, CCF- LRU algorithms for different four workloads: random, read-most, write-most and zipf traces. AD- LRU has been found better than other algorithms in terms of hit rate, write count and run time. In specific, AD-LRU reduced write count under zipf trace by 23%, 17% and 21% compared to LRU, CFLRU and LRU-WSR respectively.

2.2 Methodology

Research is a careful study performed to find out new things in a systematic way. In a scientific method of research at first problem is formulated then output information is generated from collected input data and output is analyzed and finally the result is generalized [14]. This dissertation work is truly scientific and flows in the same way. The topics memory management and design has been studied from the early generation of computer. Page replacement algorithm is one of the major strategies to manage memory efficiently. The main exploration of this dissertation focuses on LIRS-WSR and AD-LRU algorithms developed to address flash memory characteristics in memory management. Out of different types of research methodologies, this dissertation is based on the trace driven simulation approach. All the data collected are primary data, which are traces of page references. Output information gathered is analyzed in a quantitative approach. Finally, conclusion is drawn with the help of analyzed data.

Chapter 3
PROGRAM DEVELOPMENT

3.1 Development Methodology and Tools

The simulator is built by using incremental approach. The LRU stack automatically maintains recency factor. Information of recently referenced block is available in top of stack and the oldest in bottom of stack. Every time when the block is accessed it is kept in top of stack. LIRS-WSR and ADLRU algorithms are also implemented by using same stack algorithm. The algorithms have been implemented in C programming language using Dev C++ 4.9.9.2 compiler on Intel(R) Core (TM) i5-4200U CPU @ 1.6 GHz-2.3GHz with 4 GB RAM Windows 7, 32 bit OS.

3.2 LIRS-WSR

LIRS-WSR uses basic concept of LIRS algorithm [19]. Sum of size of HIRS and size of LIRS is equals to size of cache. HIR block that may be resident or non-resident can be promoted to LIR block. At the same time to maintain the LIRS and HIRS size, oldest LIR block must be demoted to HIR-resident block. Then one of the resident HIR block becomes the victim. The promotion demotion policy is shown in the Fig3.1. Figures 3.2 and 3.3 show the specific promotion demotion policy among LIR, resident HIR and non-resident HIR, so as to maintain partition size. Every page has additional status either cold or not-cold.

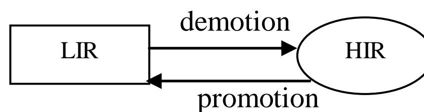


Fig 3.1 General LIR vs. HIR Transition

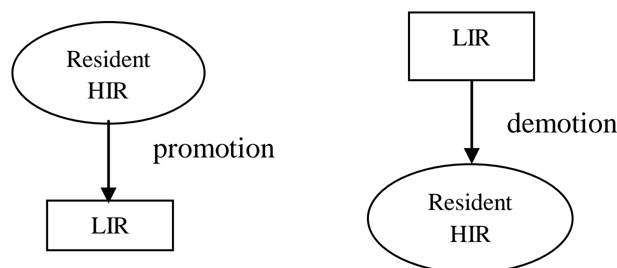


Fig 3.2 Specific LIR vs. Resident HIR Transition Diagram

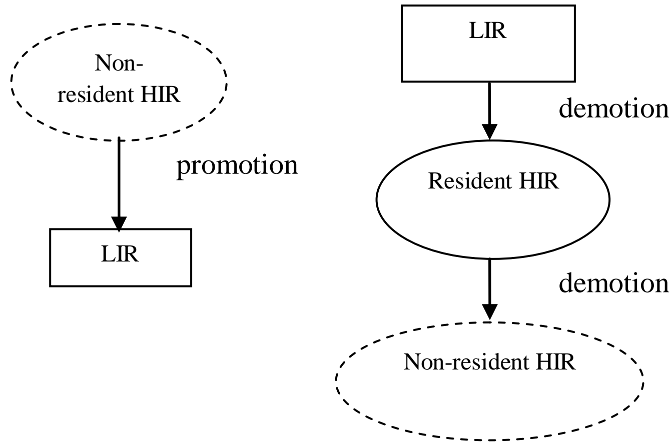


Fig 3.3 LIR vs. Non-Resident HIR Transition

LIRS-WSR can be implemented by using two lists: LIRS Stack S and HIR queue Q. Stack S contains LIR pages as well as HIR pages regardless of their residence status. Some of them are resident and others are not, only their metadata are stored in the Stack. Its main purpose is to maintain recency value. As we move toward bottom recency factor increases. Bottom most one is always LIR block, which is the oldest block having higher recency factor and topmost one is the recent block having recency factor equals to zero. Each stack node contains information about reference block. Here information of every page reference is not available in stack S due to the major event stack pruning. Some information is also available in queue Q and some outdated information is also left in Stack. Queue Q contains collection referenced pages that are resident HIR blocks available in cache. Hence size of HIR cache partition determines the size of Queue Q.

Initially all pages are cold, this cold flag is cleared if the pages are referenced again when they are in stack S or queue Q. The block in the Queue can be removed from anywhere if it is promoted to LIR. Comparing IRR and recency value is automatically done by the use of Q which increases performance. If a page is introduced to the buffer for write request for the first time, it becomes a dirty page and enters the top of the stack S as an LIR page. Every time when Stack bottom is moved to HIR Q, WSR policy is applied. That is, if bottom LIR page is dirty and not cold, then its cold flag is set and moved to the head of Stack, otherwise it is moved to the head of HIR Q. That is, only clean or dirty cold pages are moved to head of HIR Q from the bottom of S. Stack pruning operation is performed on every move of operation performed on bottom LIR page of S.

3.2.1 Data Structure

The LIRS-WSR algorithm can be implemented by using two LRU lists LIRS stack and HIR queue. Each node in the LIR list and HIR Q are implemented as a doubly linked list. Each node of the list has structure as in Fig. 3.7.

```
struct node
{
    Char pn[9];                     // contains page number
    Char r;                         // access type (r/w)  1 for write and 0 for read
    int isResident;                 // flag to determine Resident/non-resident HIR
    int isHIR_block;                // flag to determine HIR/LIR
    int cold;                       // flag for cold/not cold
    struct node * LIRS_next;        //next node in LIRS stack
    struct node * LIRS_prev;        // previous node in LIRS stack
    struct node* HIR_Rsd_next;      // next node of HIR Q
    struct node *HIR_Rsd_prev;      // previous node of HIR Q
    int    recency;                 // flag to indicate page is in stack or not
} ;
```

Fig. 3.4 Structure of a node in LIRS-WSR Data Structure

3.2.2 Stack Pruning Function

The major function stack pruning is conducted during status change. Bold assumption of the algorithm is that bottom of stack S is always LIR block. While changing status, the page in bottom of stack S is demoted to HIR resident for that it is kept in queue Q. At that time next LIR bottom is chosen which is nearer from bottom of stack S and all other HIR bottom are removed one by one. Information of thus removed HIRs is available in queue Q, if it is resident. Stack pruning is also conducted if the accessed block P is the bottom LIR because recent block is always moved to top of stack S. Stack pruning decreases the size of stack hence the stack doesn't keep track of outdated references.

3.2.3 Algorithm

1. Start.
2. Read new page.
3. If page is in Stack S, Then
   3.1. If page is LIR page, Then
      3.1.1. Page hit LIR page.
      3.1.2. Clear cold flag, i.e. cold=0.
      3.1.3. Move the page to the head of Stack, S.
      3.1.4. If the page is at bottom of S, then

3.1.4.1. Prune the Stack, S.

3.2. Else if page is HIR, then,

3.2.1. If page is resident HIR, then

3.2.1.1. Page hit, HIR resident page.

3.2.1.2. Move the page to the head of Stack, making it LIR.

3.2.1.3. Clear cold flag i.e. cold =0.

3.2.1.4. Remove the page from HIR list Q.

3.2.1.5. While Stack bottom is dirty and not cold, repeat

3.2.1.5.1. Move bottom page of Stack, S to the head of Stack, S.

3.2.1.5.2. Set cold =1 for this page.

3.2.1.5.3. Prune the Stack

3.2.1.6. Move Stack bottom to head of the HIR list, making it HIR.

3.2.1.7. Prune the Stack.

3.2.2. Else page is non-resident HIR

3.2.2.1. Page Miss

3.2.2.2. Remove Tail of HIR Q

3.2.2.3. Move it to the head of Stack, S, making LIR

3.2.2.4. Clear cold flag of the page.

3.2.2.5. While Stack bottom is dirty and not cold, then

3.2.2.5.1. Set the cold flag of bottom page, i.e. Cold =1

3.2.2.5.2. Move the page to the head of Stack, S.

3.2.2.5.3. Prune the Stack.

3.2.2.6. Move the bottom of Stack to the head of HIR queue, Q.

3.2.2.7. Make this page resident HIR.

3.2.2.8. Prune the Stack.

4. Else if Page is in HIR Q then,

4.1. Page Hit in HIR Queue.

4.2. Move to the head of HIR, Q.

4.3. Add to the head of Stack, S.

5. Else

5.1. Page miss occurs

5.2. If free memory is available, then

5.2.1. If free memory is larger than HIR_Limit then,

5.2.1.1. Add page to the head of Stack.

5.2.1.2. Make it LIR page.

5.2.1.3. Decrease free memory by one.

5.2.2. Else

5.2.2.1. If page is write, Then,

5.2.2.1.1. Add the page to the head of Stack, S.

5.2.2.1.2. Make it LIR.

5.2.2.1.3. While Stack bottom is dirty and not cold, do

5.2.2.1.3.1. Move bottom of Stack, S to the head of Stack, S.

5.2.2.1.3.2. Set cold =1 for this page.

5.2.2.1.3.3. Prune the Stack

5.2.2.1.4. Move Stack bottom to head of the HIR list

5.2.2.1.5. Make this page resident HIR.

5.2.2.1.6. Prune the Stack.

5.2.2.1.7. Decrease free memory by one.

5.2.2.2. Else page is read

5.2.2.2.1. Add the page to head of queue Q.

5.2.2.2.2. Add the page to the head of Stack,

5.2.2.2.3. Decrease free memory by one.

5.3. Else Memory is full.

5.3.1. Remove tail of HIR queue, Q.

5.3.2. If page is write, Then,

5.3.2.1. Add the page to the head of Stack, S.

5.3.2.2. Make it LIR.

5.3.2.3. While Stack bottom is dirty and not cold, do

5.3.2.3.1. Move bottom of Stack, S to the head of the Stack.

5.3.2.3.2. Set cold =1 for this page.

5.3.2.3.3. Prune the Stack

5.3.2.4. Move Stack bottom to head of the HIR list

5.3.2.5. Make this page resident HIR.

5.3.2.6. Prune the Stack.

5.3.3. Else page is read

5.3.3.1. Add the new page to head of Stack, S.

5.3.3.2. Add the new page to the head of Q.

6. Stop.

3.2.4 Flowchart

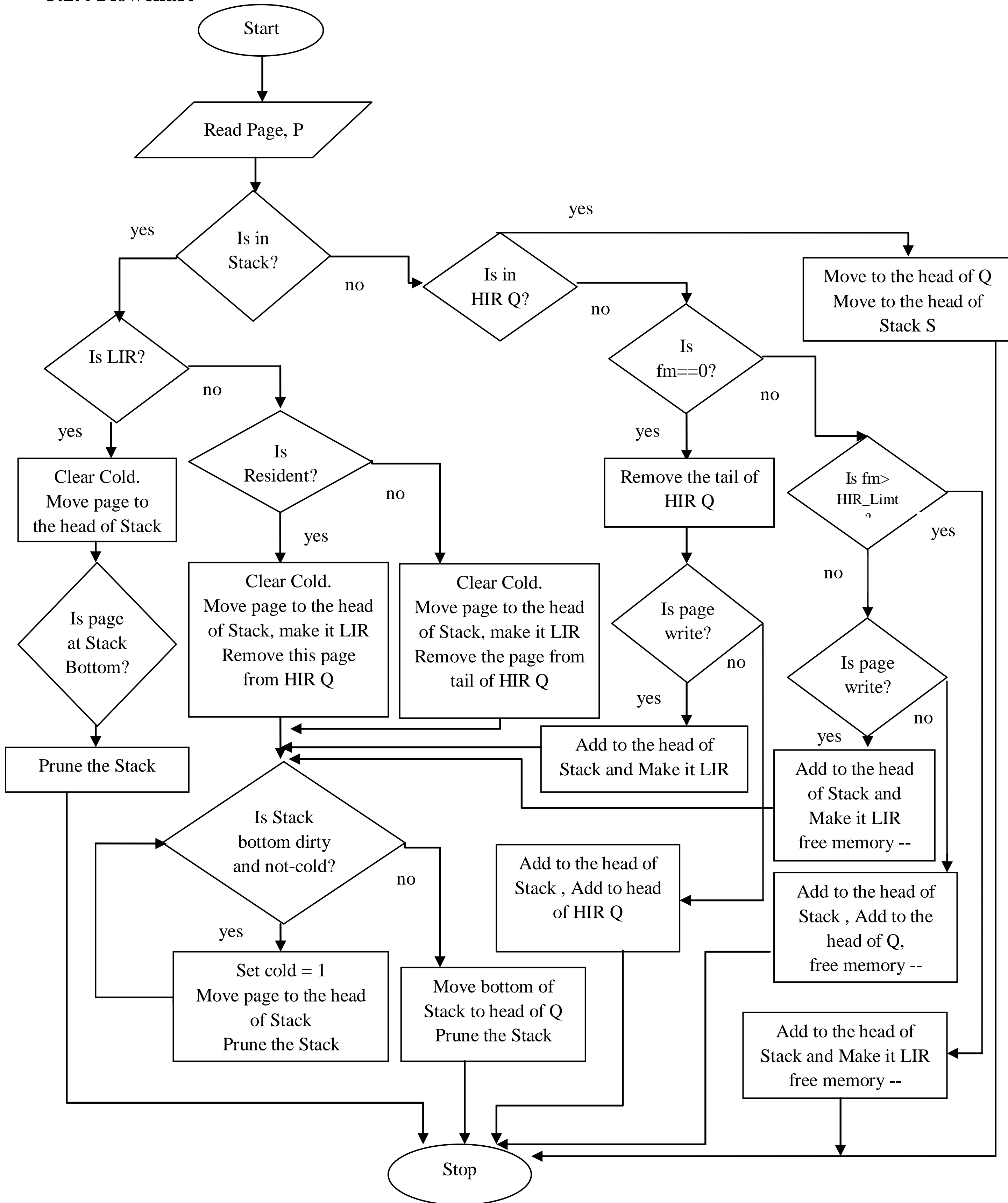


Fig. 3.5 Flowchart of LIRS-WSR Algorithm

3.2.5 Tracing of LIRS-WSR

Size of LIRS: 2

Size of HIRS: 1

Cache Size: 2+1=3

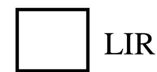
Input References: 1,1 1,2 0,3 0,1 1,4 1,3 0,5 0,2 1,3

1 = write, 0 = read

Number of Distinct References: 5

Total Number of References: 9

Other page status: cold/hot, dirty/clean


LIR

Resident HIR

Non-residents HIR

Upon accessing page 1,1

Page Miss

Page fault

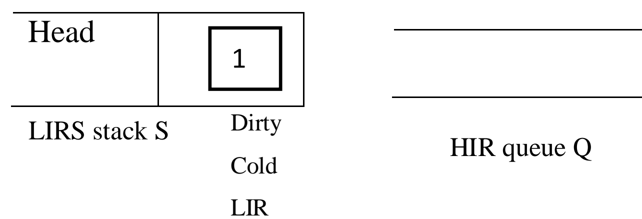


Fig 3.6.1 State at Virtual Time 1

Upon accessing page 1,2

Page Miss

Page fault

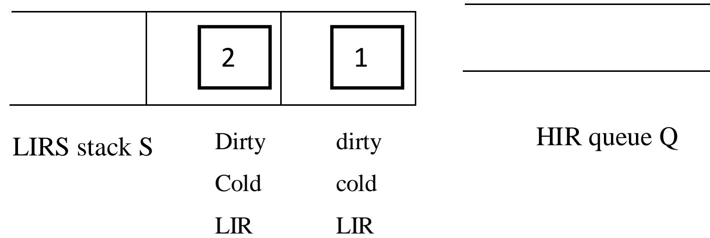


Fig 3.6.2 State at Virtual Time 2

Upon accessing page 0,3

Page Miss

Page fault

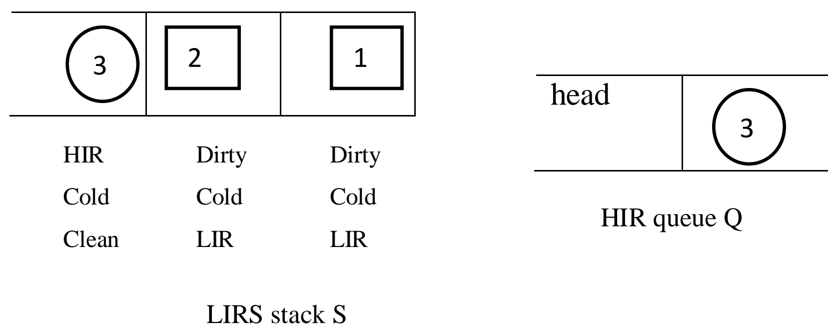


Fig 3.6.3 State at Virtual Time 3

Upon accessing page 0,1

Page Hit

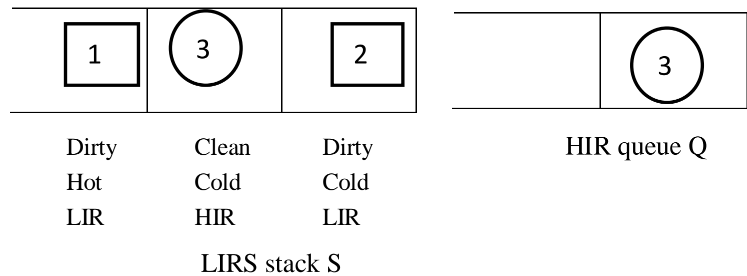


Fig 3.6.4 State at Virtual Time 4

Upon accessing page 1,4

Page Miss, memory full
Write for first time
Added to head of stack, 2 is
Demoted to HIR Q
3 is removed by pruning.
page fault

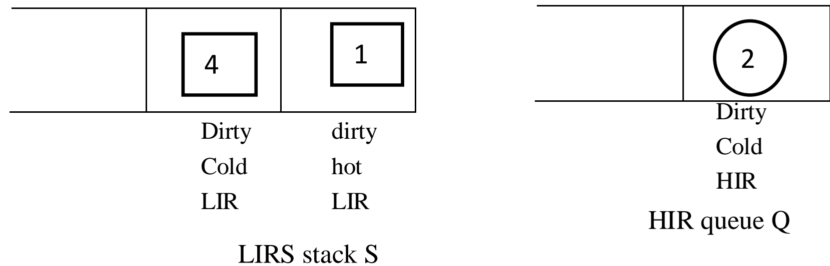


Fig 3.6.5 State at Virtual Time 5

Upon accessing 1,3

Page Miss
Memory full
Inserted at the head of stack,
1 is given second chance and 4
Is demoted to HIR Q
2 is removed from Q.
write count =1
Page fault

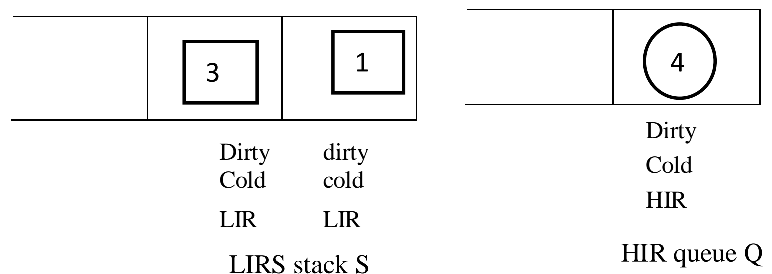


Fig 3.6.6 State at Virtual Time 6

Upon accessing 0,5

Page miss

Page fault
Remove 4 and 5 to head of Stack
And also Q head
Write count =2

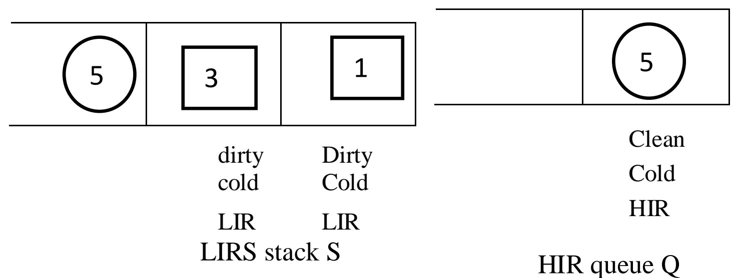


Fig 3.6.7 State at Virtual Time 7

Upon accessing 0,2

Page miss
Page fault

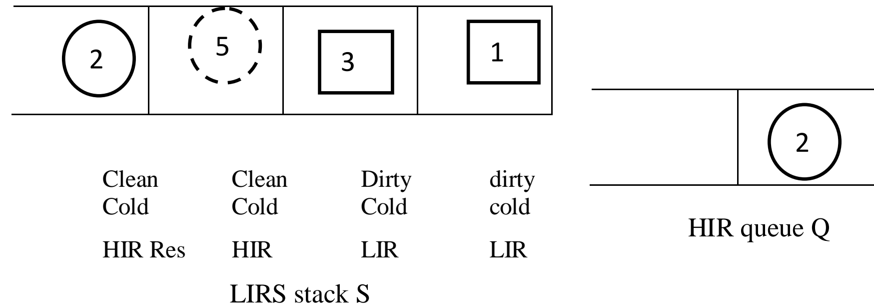


Fig 3.6.8 State at Virtual Time 8

Upon accessing 1,3

Page hit

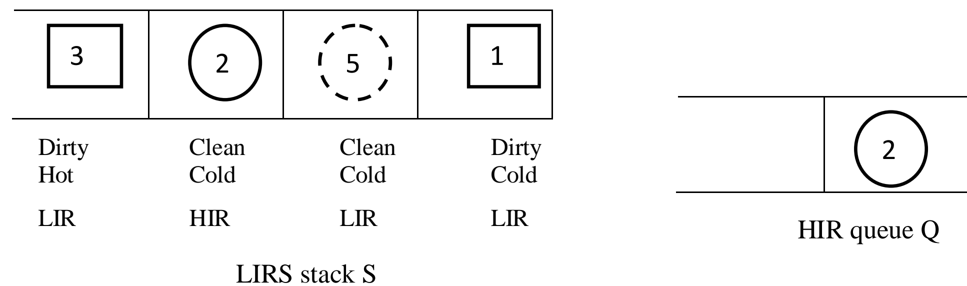


Fig 3.6.9 State at Virtual Time 9

Total page faults = 7
Write counts = 4

3.3 AD-LRU

AD-LRU has two LRU queues to capture the concept of recency and frequency of the page references, among which Cold LRU queue stores the pages referenced only once and Hot LRU queue maintains the pages that are referenced at least twice. The sizes of these two LRU queues are dynamically adjusted according to changes in reference patterns. When a page is first referenced, it is put in the head of cold LRU queue. The pages move from cold LRU queue to head of hot LRU queue when it is referenced again and when a page in hot LRU queue is selected as victim, it is demoted to head of cold LRU queue. During the eviction procedure, least recently used clean page from cold LRU queue is selected as a victim. There is a specific pointer FC (First Clean) to point least recently used clean page in each LRU queue. If clean pages do not exist in the cold LRU queue, second chance policy is applied. For this purpose, each page in double LRU queues is marked by a reference bit which is always set to 1, when the page is referenced. When a page is referenced and buffer is full, the page pointed by FC pointer of cold LRU queue (Cold_FC) is evicted, if Cold_FC is not null.

Otherwise dirty page at the tail of this queue is evicted if its reference bit is 0. But if reference bit is 1, then the reference bit is cleared to 0 and moved head of this queue, process continues to find next dirty page with reference bit 0. There is a parameter MIN_LC to set lowest limit of size of cold LRU queue. If the size of cold LRU queue reaches MIN_LC, victim will be selected from hot LRU queue rather than cold LRU queue. In this case victim page is selected in same way as in cold LRU queue but the victim page is moved to head of cold LRU queue.

3.3.1 Data Structure

The data structures used are based on stack algorithm. There are two LRU queues one for cold pages and another for hot pages. These both queues have two external pointers HEAD and TAIL to indicate most recently used (MRU) and LRU pages in the queue. These queues are implemented as a doubly linked list each node has structure as shown in Fig 3.7.

```
struct node
{
    Char pn[9];              // contains page number and access type flag
    Char r;                  // access type (r/w), 1 for write and 0 for read
    int cold;                // to flag cold page 1 for cold and 0 for hot
    int reference;           // to assist in second chance policy in eviction of page
    struct node * next;      // pointer to point next node
    struct node * prev;      // pointer to point previous node
  };
```

Fig.3.7 Structure of node of ADLRU Data Structure

3.3.2 Algorithm

1. Start
2. Read new page, P.
3. If P is in Hot queue, Then,
   3.1. Hit in Hot queue.
   3.2. Set Ref(P) = 1.
   3.3. Move the page P to the head of Hot queue.
   3.4. Adjust Hot First Clean (Hot_FC) pointer to point LRU clean page in Hot queue.
4. Else if P is in Cold queue, Then,
   4.1. Hit in Cold queue.
   4.2. Set Ref(P) =1.

4.3. Move the page P to the head of Hot queue from Cold queue.

4.4. Adjust Cold First Clean (Cold_FC) pointer in Cold queue.

4.5. Adjust Hot_FC in Hot queue.

4.6. Decrease the size of cold queue, i.e. Cold_Size--.

5. Else

5.1. Miss occurs.

5.2. If free_memory >0, Then

5.2.1. Free memory is available, Add new page to head of Cold queue.

5.2.2. Increment size of cold queue, i.e., Cold_Size ++ .

5.2.3. Decrement free memory.

5.3. Else

5.3.1. Memory is full, page replacement is needed.

5.3.2. While Cold_Size <= Cold_Min_limit, do,

5.3.2.1 If Hot_FC != NULL, Then

- Move the page pointed by Hot_FC to the head of Cold queue
- Adjust Hot_FC

5.3.2.2 Else

-While Ref(Hot queue tail) is 1, do

- ▪ Ref(Hot queue tail) =0
- ▪ Move the Hot queue tail to the head of Hot queue

- Move the Hot queue tail to the head of Cold queue

5.3.2.3 Cold_Size ++

5.3.3. If Cold_FC != NULL, Then,

5.3.3..1. Remove page pointed by Cold_FC from Cold queue

5.3.3..2. Adjust Cold_FC pointer to point next LRU clean page

5.3.4. Else

5.3.4..1. While Ref(Cold queue tail) is 1, do

- Ref(Cold queue tail) =0
- Move the Cold queue tail to the head of Cold queue.

5.3.4..2. Remove page from tail of Cold queue.

5.3.5. Add the new page, P to the head of Cold queue.

5.3.6. Adjust Cold_FC.

6. Stop.

3.3.3 Flowchart

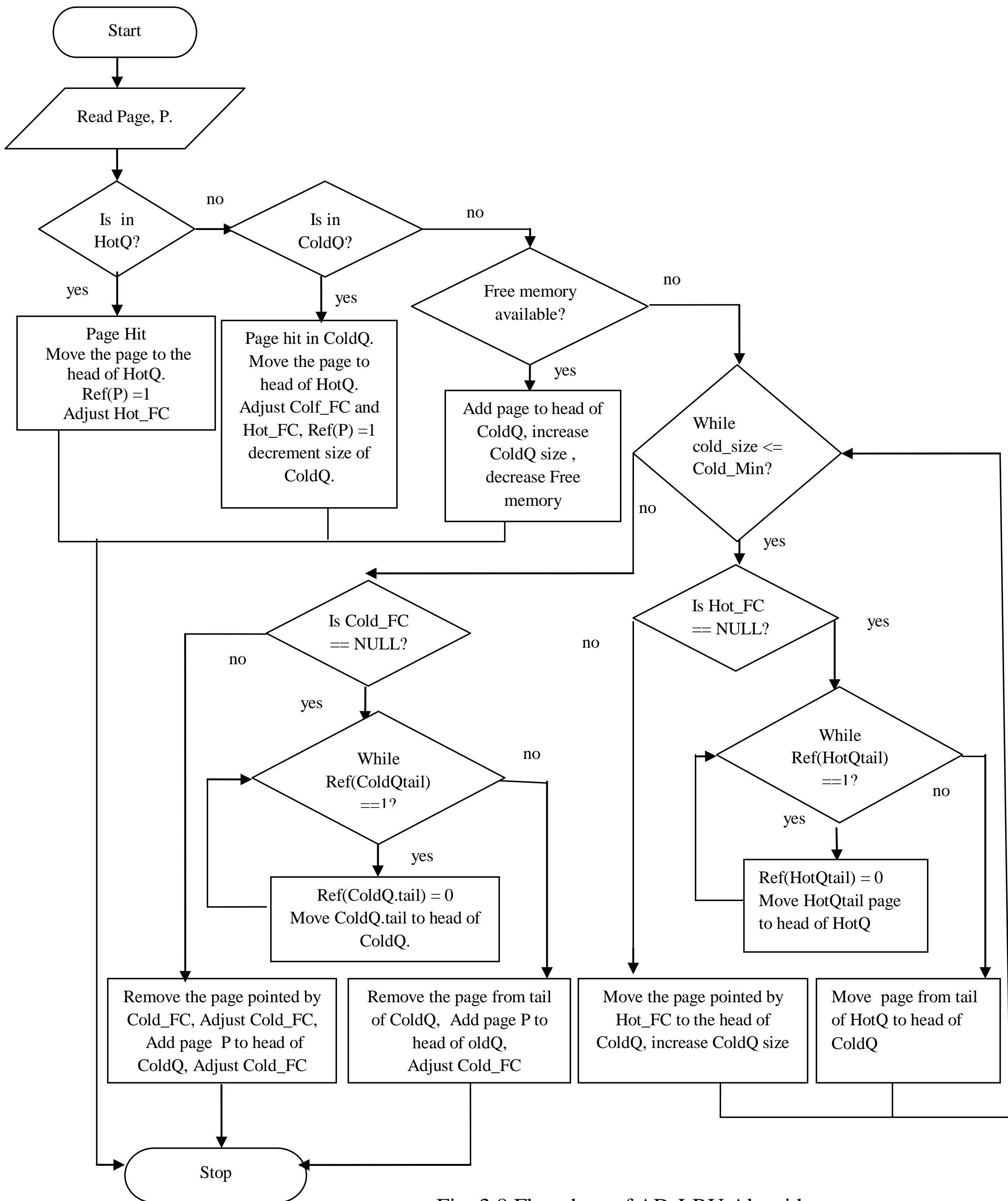


Fig. 3.8 Flowchart of AD-LRU Algorithm

3.3.4 Tracing of AD-LRU

Cache Size: =3

Minimum size of Cold Queue (MIN_LC) = 1

Input References: 1,1 1,2 0,3   0,1 1,4   1,3 0,5 0,2 1,3

1 = write, 0 = read

Number of Distinct References: 5

Total Number of References: 9

Page status: cold/hot, dirty/clean

- Upon accessing: 1,1

  Page fault

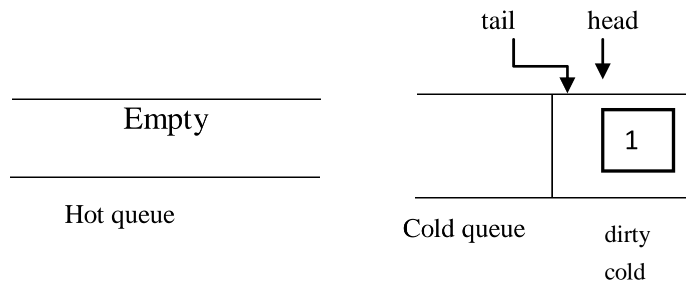


Fig 3.9.1 State at Virtual Time 1

- Upon accessing: 1,2

  Page fault

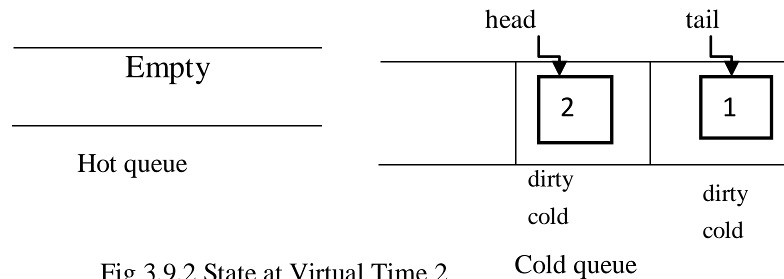


Fig 3.9.2 State at Virtual Time 2

- Upon accessing 0,3

  Page fault

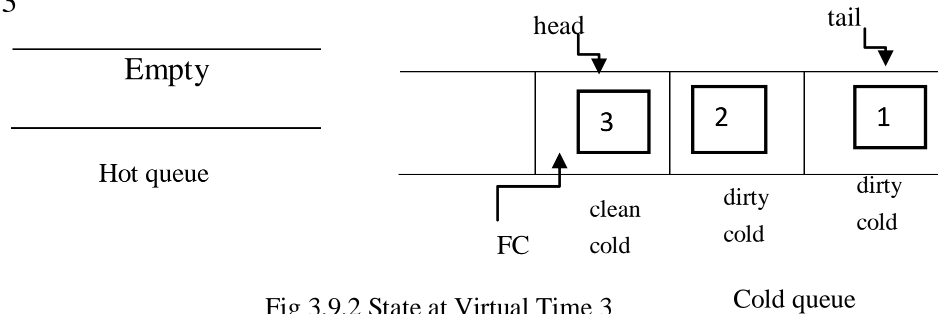


Fig 3.9.2 State at Virtual Time 3

- Upon accessing 0,1

  Page hit

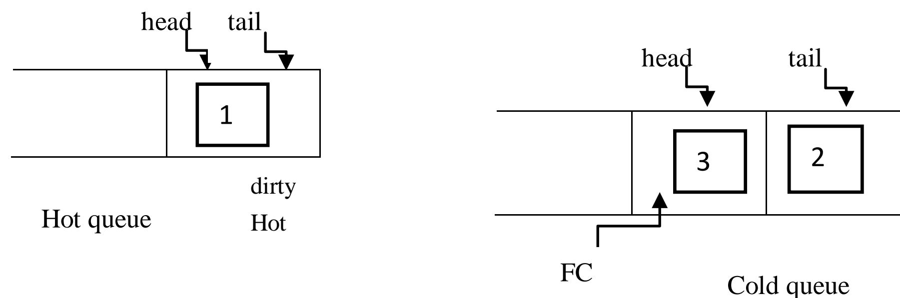


Fig 3.9.4 State at Virtual Time 4

- Upon accessing 1,4

  Page fault

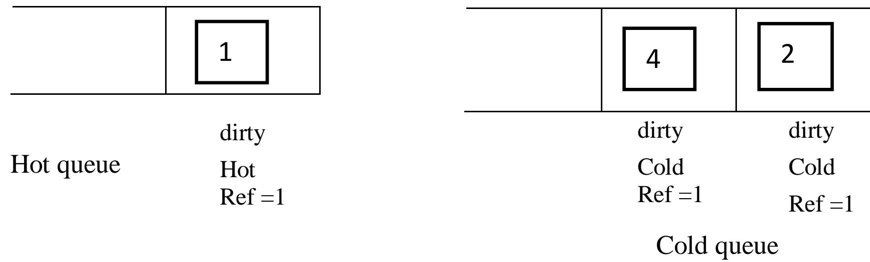


Fig 3.9.5 State at Virtual Time 5

- Upon accessing 1,3

  Page miss

  Page fault

  Write count =1

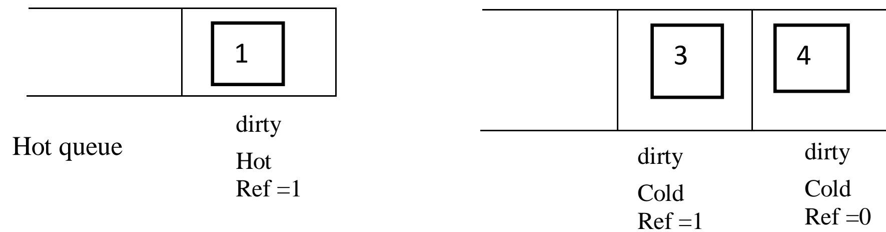


Fig 3.9.6 State at Virtual Time 6

Cold queue

- Upon accessing 0,5

  Page fault

  Write count = 2

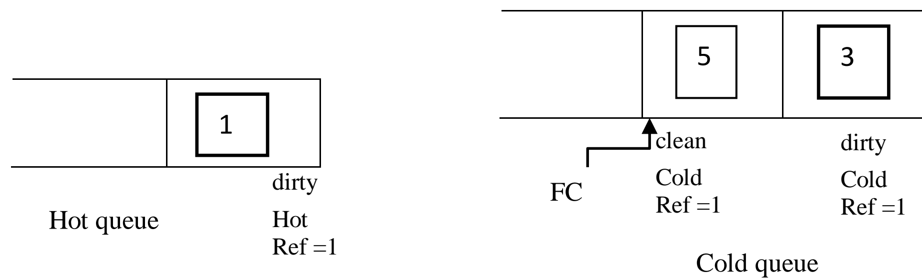


Fig 3.9.7 State at Virtual Time 7

- Upon accessing 0,2

  Page miss

  Page fault

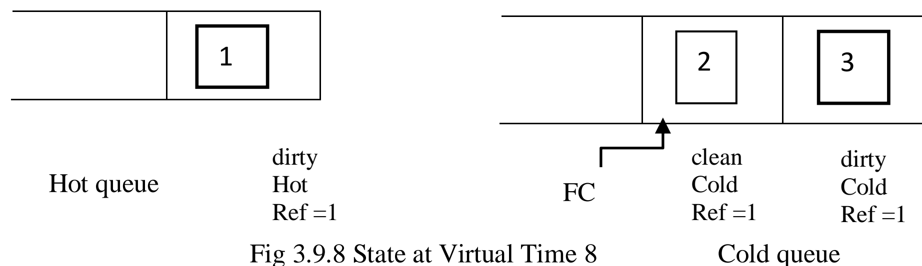


Fig 3.9.8 State at Virtual Time 8

Cold queue

- Upon accessing 1,3

  Page hit

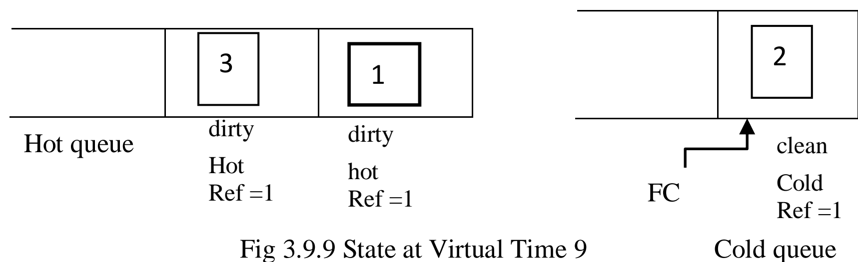


Fig 3.9.9 State at Virtual Time 9

Cold queue

Total page faults = 7

Write count = 4

Chapter 4

TEST RESULTS & ANALYSIS

4.1 Data Collection

Data are the sources of information. Hence data should be collected very carefully. In this dissertation work four types of synthetic traces [16] have been used in the simulation experiment, i.e., random trace, read- most trace (e.g., of decision support systems), write-most trace (e.g., of OLTP systems), and Zipf trace as Workload1, Workload 2, Workload 3 and Workload 4 respectively. These data are real memory traces. Workload represents different locality of memory reference pattern that are generated during execution of process in real OS. There are total 100,000 page references in each of the first three traces, which are restricted to a set of pages whose numbers range from 0 to 49,999. The total number of page references in the Zipf trace is set to 500000 in order to obtain a good approximation, while the page numbers still fall in [0, 49999]. Zipf trace has a referential locality "20/80" meaning that eighty percent of the references deal with the most active twenty percent of the pages. Sample of Workload 1, Workload 2, Workload 3 and Workload 4 are in appendix A, appendix B, appendix C and appendix D respectively. Table 4.1 to Table 4.4 shows the details concerning these workloads.

Attributes	Value
Total I/O references	100,000
Total Distinct references	43247
Read/Write ratio	50% /50%
Reference Patterns	Uniform

Table 4.1 Simulated trace for Random access

Attributes	Value
Total I/O references	100,000
Total Distinct references	43212
Read/Write ratio	90% /10%
Reference Patterns	Uniform

Table 4.2 Simulated trace for Read-most access

Attributes	Value
Total I/O references	100,000
Total Distinct references	43182
Read/Write ratio	10% /90%
Reference Patterns	Uniform

Table 4.3 Simulated trace for Write-most access

Attributes	Value
Total I/O references	500,000
Total Distinct references	47023
Read/Write ratio	50% /50%
Reference Locality	20%/80%

Table 4.4 Simulated Zipf trace

4.2 Testing

Each workload is tested in LIRS-WSR and AD-LRU simulator by varying the cache size from 512 to 18432. In case of LIRS-WSR algorithms HIR, LIR partition is maintained as 1% and 99% of cache size. For AD-LRU parameter MIN_LC is set 0.5 of the cache size for all Workloads.

4.2.1 Test Result of Workload 1 (Trace with Random Access)

No. of References = 100000 No. of Distinct Reference = 43247
No. of Write References = 49974

Buffer Size	AD-LRU				LIRS-WSR			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98903	98.1	1.9	49085	98955	98.2	1.8	48950
1024	97661	95.9	4.1	48141	97926	96.3	3.7	47959
2048	95446	92.0	8.0	46396	95847	92.7	7.3	46008
4096	90937	84.0	16.0	42913	91892	85.7	14.3	42538
6144	86792	76.7	23.3	39875	87951	78.8	21.2	39344
8192	82462	69.1	30.9	36713	84103	72.0	28.0	36482
9216	80418	65.5	34.5	35274	82253	68.7	31.3	35180
10240	78411	62.0	38.0	33871	80421	65.5	34.5	33950
12288	74679	55.4	44.6	31282	76800	59.1	40.9	31694
14336	71090	49.1	50.9	28945	73358	53.1	46.9	29706
16384	67615	42.9	57.1	26794	70076	47.3	52.7	27996
18432	64354	37.2	62.8	24814	66834	41.6	58.4	26446

Table 4.5 Test Result of Workload 1

4.2.2 Test Result of Workload 2 (Trace with Read-Most Access)

No. of References = 100000
No. of Distinct Reference = 43212
No. of Write References = 9919

Buffer Size	AD-LRU				LIRS-WSR			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98873	98.0	2.0	9295	98948	98.1	1.9	9078
1024	97698	95.9	4.1	8714	97954	96.4	3.6	8505
2048	95554	92.2	7.8	7656	95943	92.9	7.1	7680
4096	91129	84.4	15.6	5849	91877	85.7	14.3	6547
6144	86854	76.9	23.1	4513	87870	78.6	21.4	5832
8192	82575	69.3	30.7	4258	84144	72.1	27.9	5339
9216	80556	65.8	34.2	4258	82330	68.9	31.1	5135
10240	78449	62.1	37.9	4258	80491	65.6	34.4	4973
12288	74543	55.2	44.8	4258	76953	59.4	40.6	4706
14336	70717	48.4	51.6	4258	73402	53.2	46.8	4527
16384	67324	42.5	57.5	4258	70137	47.4	52.6	4403
18432	64004	36.6	63.4	4258	66828	41.6	58.4	4334

Table 4.6 Test Result of Workload 2

4.2.3 Test Result of Workload 3 (Trace with Write-Most Access)

No. of References = 100000
No. of Distinct Reference = 43182
No. of Write References = 89145

Buffer Size	AD-LRU				LIRS-WSR			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98825	97.9	2.1	87981	98933	98.1	1.9	88084
1024	97664	95.9	4.1	86833	97859	96.2	3.8	87018
2048	95407	91.9	8.1	84613	95831	92.7	7.3	84999
4096	91173	84.5	15.5	80454	91903	85.7	14.3	81111
6144	87102	77.3	22.7	76494	88066	79.0	21.0	77380
8192	83130	70.3	29.7	72614	84211	72.2	27.8	73621
9216	81222	67	33.0	70765	82390	69.0	31.0	71869
10240	79361	63.7	36.3	68953	80578	65.8	34.2	70126
12288	75550	57	43.0	65258	77008	59.5	40.5	66720
14336	71887	50.5	49.5	61736	73484	53.3	46.7	63398
16384	68319	44.2	55.8	58324	70197	47.5	52.5	60300
18432	64838	38.1	61.9	55007	66856	41.7	58.3	57298

Table 4.7 Test Result of Workload 3

4.2.2 Test Result of Workload 4 (Zipf Trace)

No. of References = 500000

No. of Distinct References = 47023

No. of Write References = 244790

Buffer Size	AD-LRU				LIRS-WSR			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	262251	47.5	52.5	125666	361423	69.4	30.6	163145
1024	237659	42.1	57.9	112052	327682	62.0	38.0	144910
2048	210314	36.0	64.0	96416	288224	53.2	46.8	123920
4096	178852	29.1	70.9	77813	242417	43.1	56.9	100009
6144	158119	24.5	75.5	65191	212645	36.6	63.4	84817
8192	142079	21.0	79.0	55381	189960	31.6	68.4	73661
9216	135200	19.5	80.5	51088	180187	29.4	70.6	68958
10240	128848	18.1	81.9	47226	171186	27.4	72.6	64718
12288	117825	15.6	84.4	40509	155797	24.0	76.0	57657
14336	107985	13.5	86.5	34465	142185	21.0	79.0	51627
16384	99273	11.5	88.5	29392	130416	18.4	81.6	46630
18432	91286	9.8	90.2	25349	119561	16.0	84.0	42206

Table 4.8 Test Result of Workload 4

4.3 Analysis

All the results obtained from simulation is analyzed by drawing different graphs. Hit rate and write count are used as criteria for analyzing their goodness.

4.3.1 Hit Rate Analysis

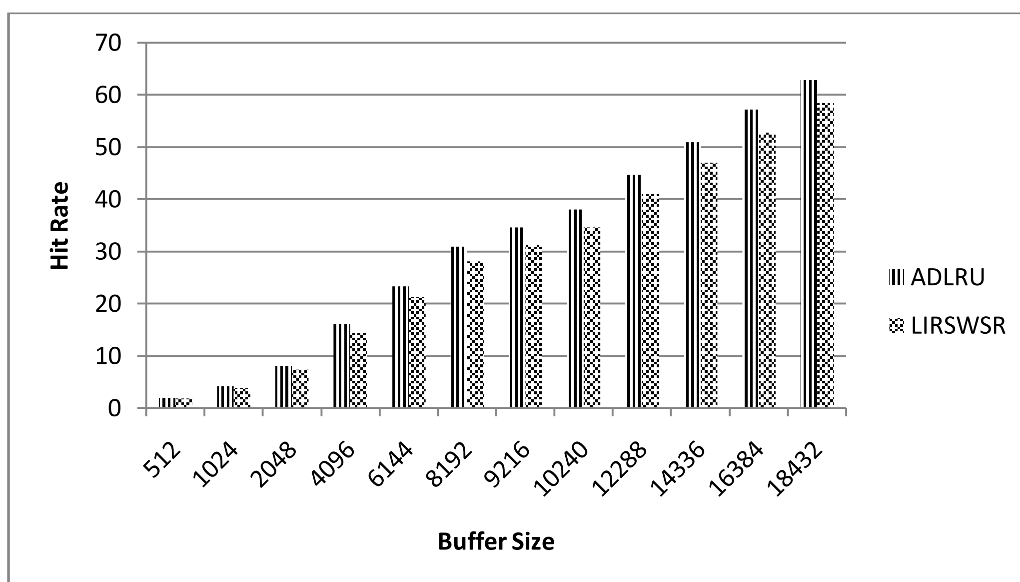


Fig 4.1 Graph of Hit Rate for Workload 1

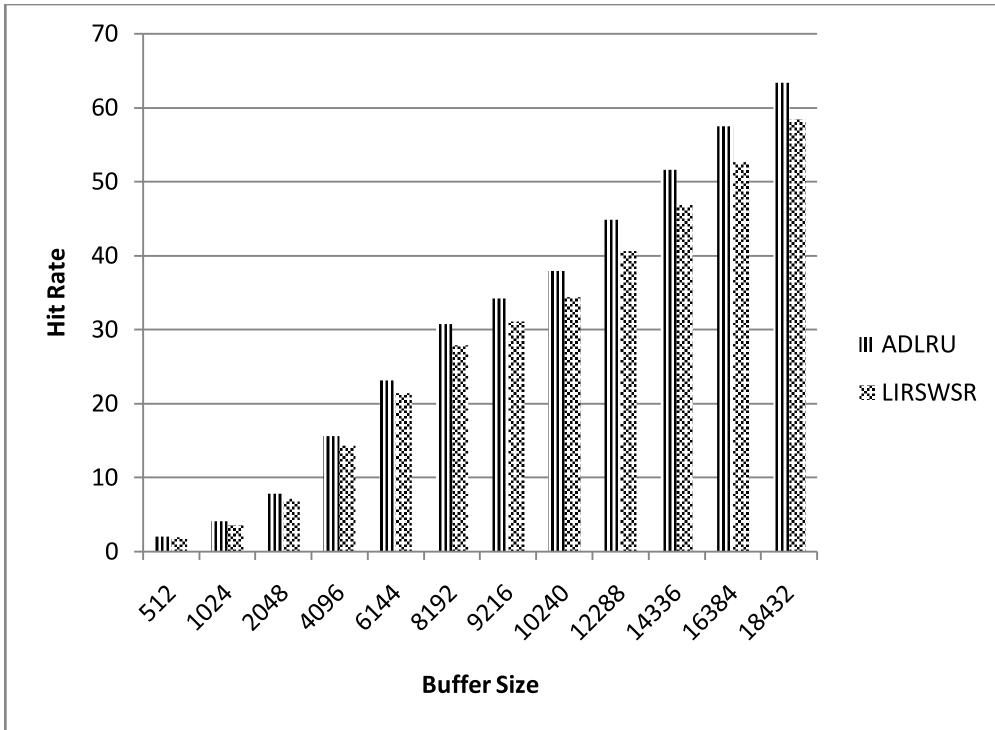
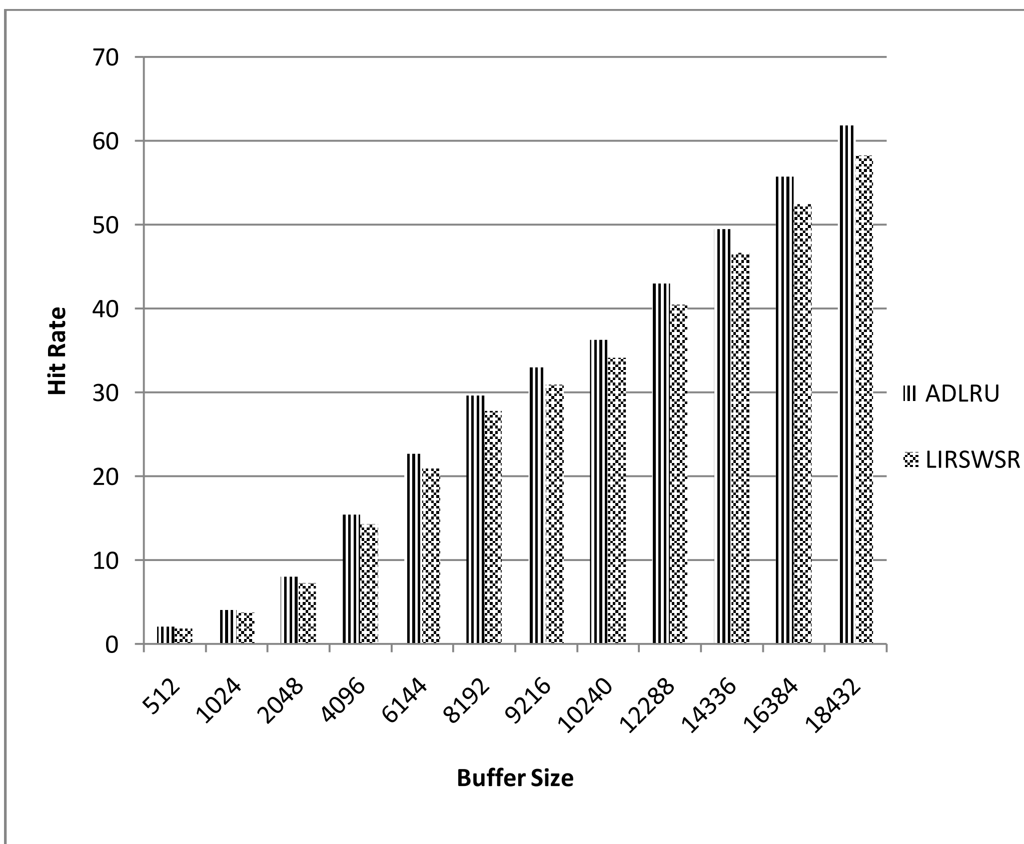


Fig 4.2 Graph of Hit Rate for Workload 2



4.3 Graph of Hit Rate for Workload 3

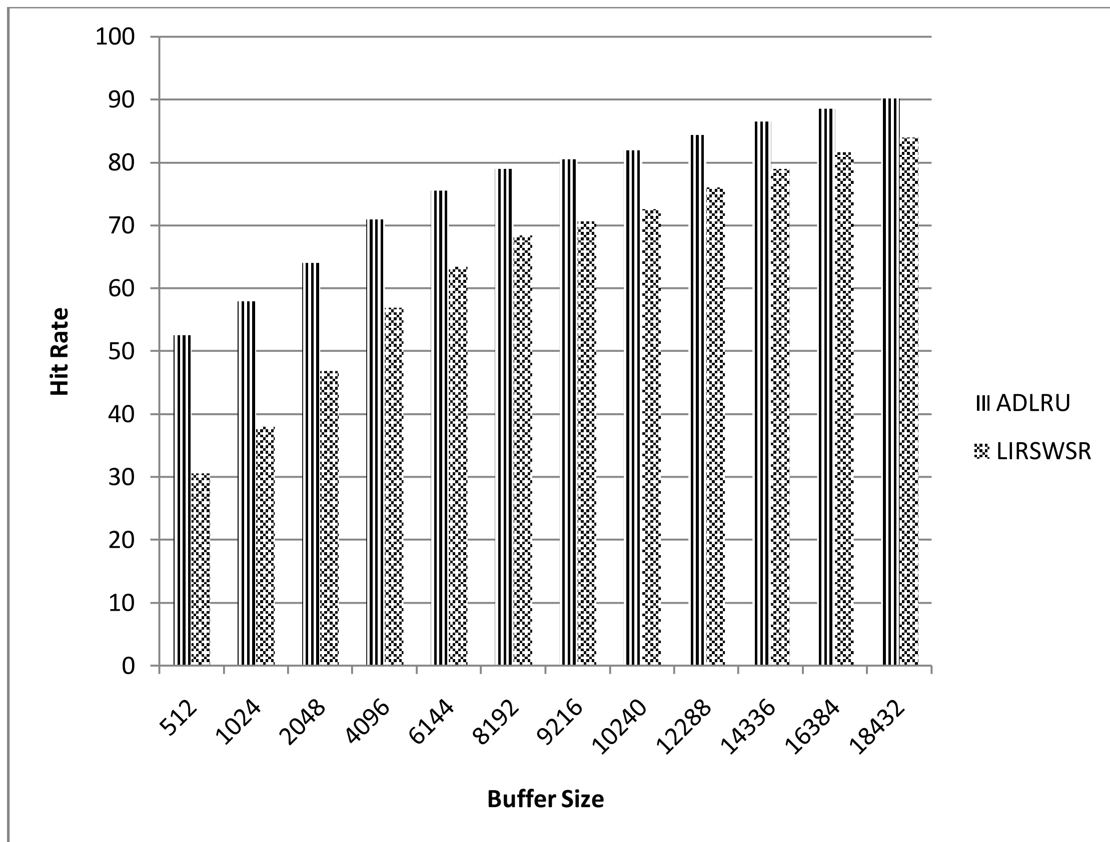


Fig 4.4 Graph of Hit Rate for Workload 4

The graphs of Figure 4.1 to Figure 4.4 show that the AD-LRU algorithm is better than the LIRS-WSR algorithm since AD-LRU has higher hit rate for different cache sizes. In the Figure 4.1 to Figure 4.3 for workloads random, read-most and write-most access type traces the hit rates of algorithms are not much different because of the uniform distribution of page references. In these workloads there is not clear distinction between hot and cold pages as reference locality is not high. Despite the nature of page references in these workloads, AD-LRU has better hit rate as it adapts the changes in page references pattern dynamically. LIRS-WSR always treats pages with write request as LIR pages or hot pages. So these write reference pages are kept in cache for longer time than read request pages. The read request pages are evicted quickly although these may be referenced soon. This reduces the hit rate.

In addition, the hit rates of both algorithms increase in similar way when buffer size increases. The hit rate increases with large buffer size because buffer can hold more pages that increases page hit.

The graph of Figure 4.4 shows significantly difference in hit rate. This is due to high reference locality of page references in zipf trace where 80% of pages references deal with active 20 % of pages. AD-LRU adapts more effectively to distinguish hot and cold pages in this workload. So, hit rate is much higher than that of LIRS-WSR. As the buffer size increases, the difference in hit rate of these two algorithms is decreasing. This is because of increased hit of both as the capacity of buffer increases.

As a minimum value AD-LRU has up to 3.6% higher hit rate than LIRS-WSR algorithm for workload 3 (write-most trace). As a maximum value AD-LRU has up to 22% higher hit rate than LIRS-WSR for smaller buffer size for workload 4 (Zipf trace) and for larger buffer size AD-LRU has more than 6% higher hit rate than LIRS-WSR with this workload. AD-LRU has higher hit rate than LIRS-WSR in case of high reference locality workloads.

4.3.2 Write Count Analysis

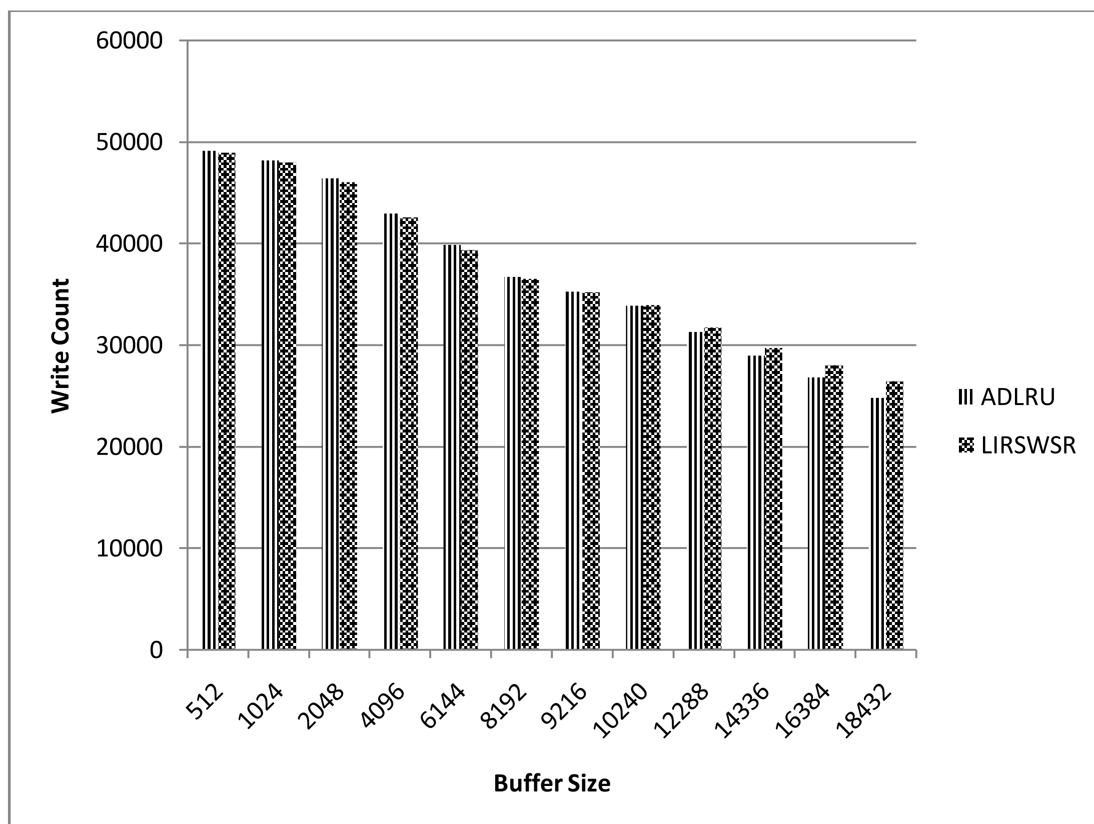


Fig 4.5 Graph of Write Count for Workload 1

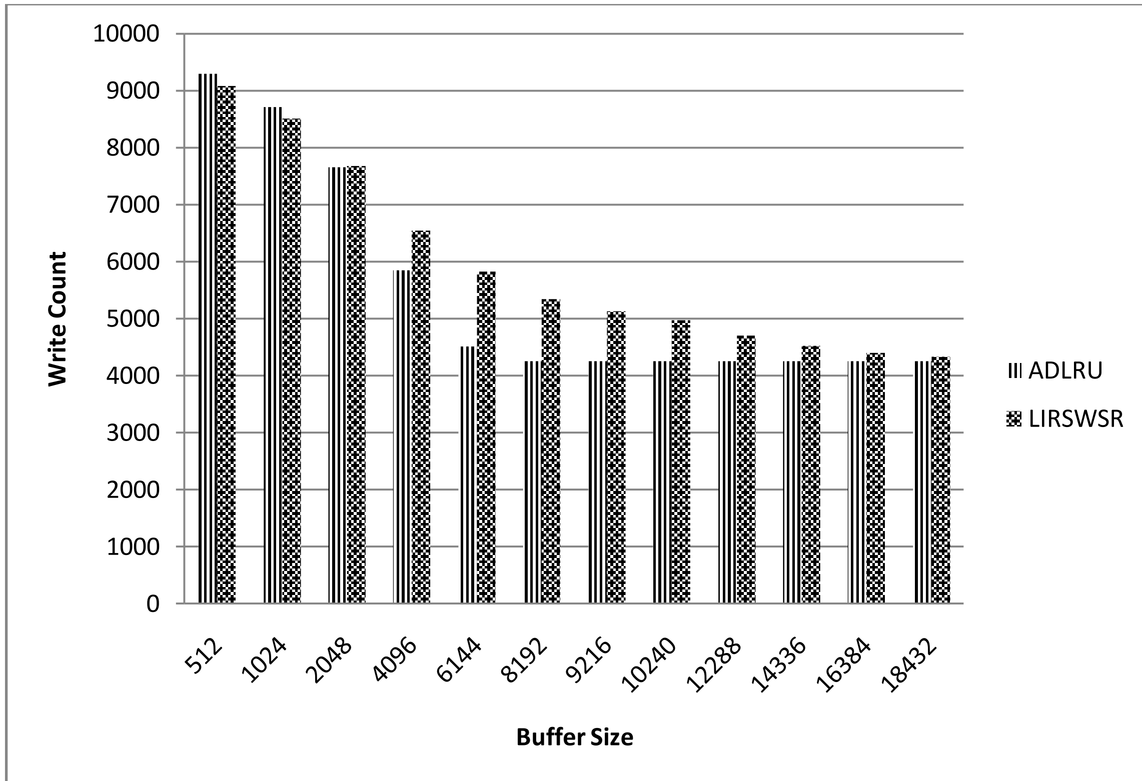


Fig 4.6 Graph of Write Count for Workload 2

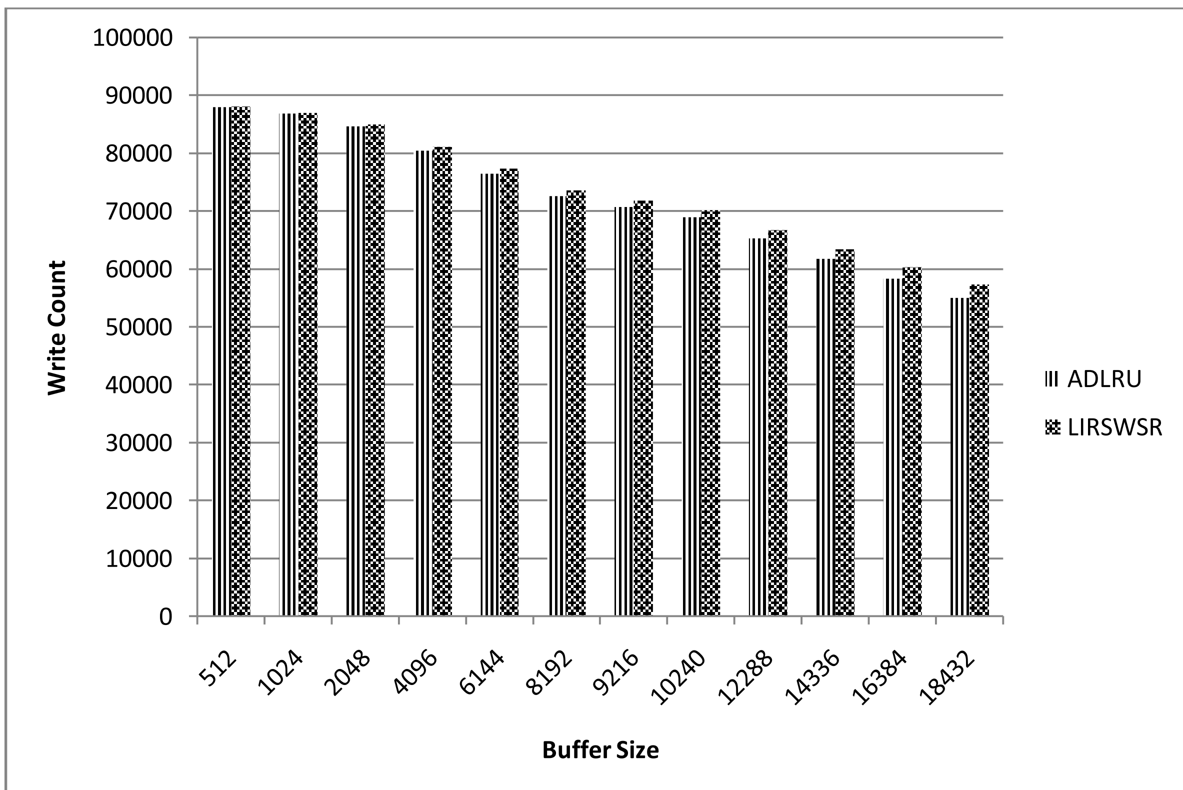


Fig 4.7 Graph of Write Count for Workload 3

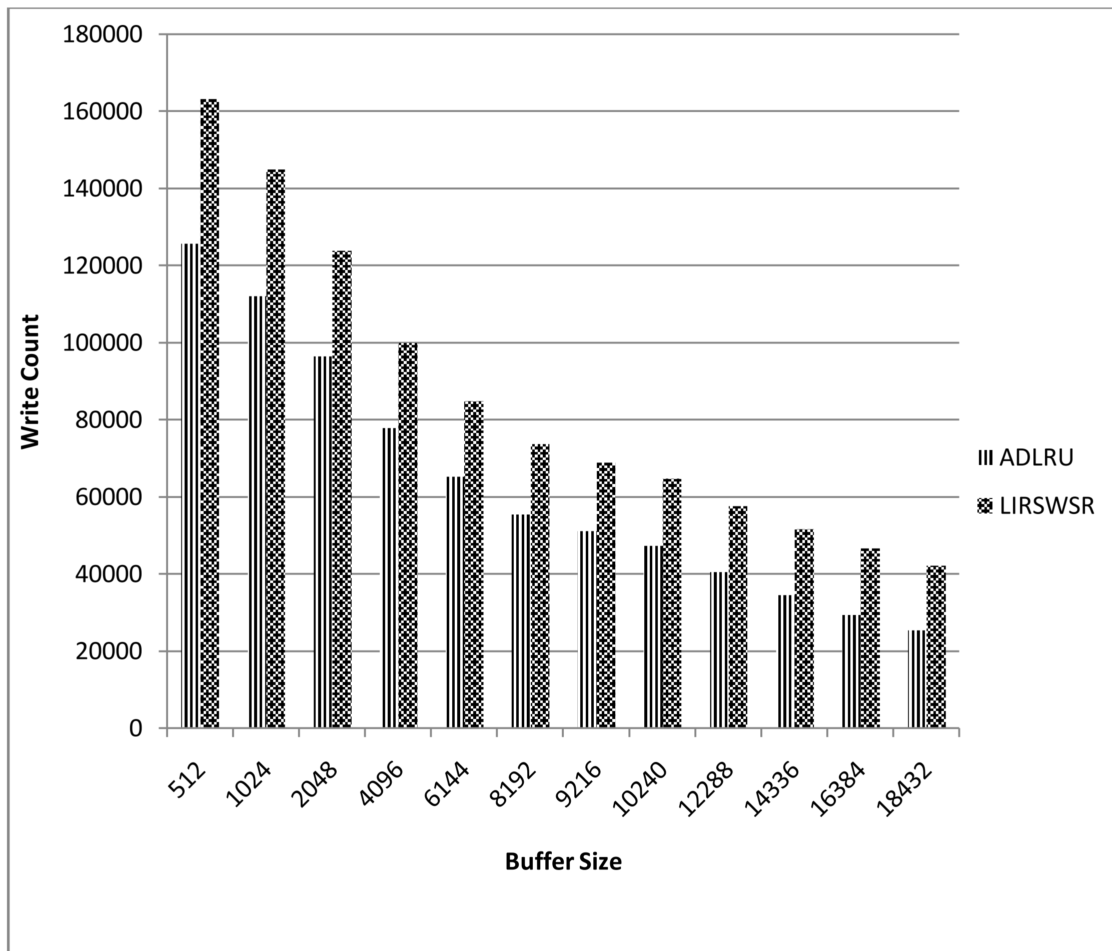


Fig 4.8 Graph of Write Count for Workload 4

The graphs in the Figure 4.5 to 4.8 show the number of pages propagated to flash memory. The number of pages flushed to flash memory is write count. The number is obtained by counting the eviction of page references with write request during page replacement event. At the end of simulation all the dirty pages in buffer are also added to the count to get the exact write count. From the above graphs it is clear that AD-LRU has smaller write count for all the work loads used in the simulation.

Workload 1 has 50% pages as write pages which are uniformly distributed. For smaller buffer size, AD-LRU has slightly higher write count than LIRS-WSR. This is because there is not clear idea to adapt hot and cold pages and as buffer has smaller capacity, dirty pages are evicted faster in AD-LRU than LIRS-WSR. In LIRS-WSR dirty pages are kept in LIRS Stack as LIR pages and due to WSR policy they are kept for longer time. As the buffer size

increases, AD-LRU outperforms LIRS-WSR in write count because of increased page hit in larger buffer.

Workload 2 has read-most access pattern in page references with 10% writes and 90% reads. For small buffer size of 512 and 1024, LIRS-WSR has lower write count because of its high priority to write pages to keep them in LIRS stack only and delay eviction using WSR policy. AD-LRU cannot differentiate them as hot or cold so evicts faster increasing write count. But for higher buffer size, AD-LRU works better. This is due to increased hit rate because of large buffer capacity. It can delay write page eviction more than WSR technique.

Workload 3 has 90% writes and only 10% reads in page references. In this case, AD-LRU works better for all buffer sizes. There is large number of write pages in trace LIRS-WSR cannot accommodate all pages in LIRS-Stack and evicts dirty pages as well continuously. AD-LRU adapts reference pattern so it has better output.

Workload 4 zipf trace has 50% / 50% read write references but has high reference locality of 80% page references are references to 20% of pages. AD-LRU adapts changes in reference pattern and locality. So it has much less write count than LIRS-WSR for all buffer sizes.

For write-most trace (Workload 3) write count is decreased up to 3% by AD-LRU with comparison to LIRS-WSR algorithm. This is the minimum value by which AD-LRU outperforms the LIRS-WSR in write count.

For zipf trace (workload 4) AD-LRU decreased write count up to 40% with comparison to LIRS-WSR algorithm. This is largest gap between value of write count of these two algorithms. Thus, AD-LRU minimizes write counts significantly when reference locality is high.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

Flash memory has become an alternative to the magnetic disks, which brings new challenges to traditional disk based system. To efficiently support the characteristics of flash storage devices, traditional buffering approaches need to be revised to take into account the imbalanced I/O property of flash memory. LIRS-WSR uses delayed eviction strategy when dirty page is to be replaced. It uses recency and little bit frequency information in replacement policy. AD-LRU captures frequency and recency of page references by using two LRU queues to classify all the buffer pages into a hot set and a cold set. It uses an adaptive mechanism to make the size of two LRU queue suitable for different reference patterns.

From the simulation of these two algorithms for varying buffer size it is found that the AD-LRU always outperforms LIRS-WSR for hit rate and write count. Specially, when workload has high reference locality, AD-LRU has significantly superior performance than LIRS-WSR in terms of both hit rate and write count. This is because of AD-LRU's good adaptive technique to handle changes in reference patterns.

For uniformly distributed workloads, the difference in hit rates of AD-LRU and LIRS-WSR is comparatively small. AD-LRU leads LIRS-WSR in hit rate by a value up to 5%. For high reference locality workloads AD-LRU has significantly higher hit rate up to 22% in comparison to LIRS-WSR.

The LIRS-WSR may perform better in write count for uniformly distributed locality workloads when buffer size is highly smaller in comparison to size of workload as it treats all write pages as hot pages and delays eviction. For larger buffer size and high write reference locality workloads, AD-LRU always outperforms LIRS-WSR in write count as AD-LRU adapts hot and cold pages better than LIRS-WSR. It seems that in case of uniformly distributed write-most access type workload, write count is decreased up to 3% by AD-LRU, but for high reference locality workload AD-LRU minimizes write count up to 40% with comparison to LIRS-WSR algorithm. Thus, AD-LRU minimizes write counts significantly when reference locality is high.

Limitations and Future Work

In this work, size of HIR block is chosen 1% of total buffer size in LIRS algorithm and minimum size of cold LRU queue MIN_LC is selected 50% of buffer size. These values have been used by authors of algorithms. Dynamic approach can be used to self tune these parameters. Further research can be done to find the optimal value of these parameters for different workloads. In addition to this, in this work, only four different memory traces have been used for simulation purpose. Three of which are of uniform reference patterns and last one is with 80/20 reference locality. These are the limitations of this work. This work can be further extended by using variety of real memory trace with different reference locality.

References

.[1] A. Silberschatz, P. B Galvin, G. Gagne, “Virtual Memory,” Operating System Concept, 8th ed., Wiley Student Edition, ch. 9, 2010, pp. 369-390.

[2] A.S. Tanenbaum, Modern Operating Systems (Prentice Hall Second Edition), 2007, pp. 201-232.

[3] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho and C. Kim, “On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies”, Proceeding of 1999 ACMSIMETRICS Conference, 1999.

[4] E.J. O’Neil, P.E. O’Neil, G. Weikum, The LRU-K Page Replacement Algorithm for Database Disk Buffering. In: Proc.of SIGMOD ’93, 1993, pp. 297-306.

[5] F. J. Corbato, A Paging Experiment with the Multics System, in: In Honor of Philip M. Morse, MIT Press, Cambridge, Mass, 1969, p. 217.

[6] G. Prakash Joshi, Calculation Of Control Parameter λ That Results Into Optimal Performance In Terms Of Page Fault Rate In The Algorithm Least Recently Frequently Used(LRFU) For Page Replacement, Master's Thesis, Tribhuvan University, CDCSIT, 2007.

[7] H. Jung, H. Shim, S. Park, S. Kang, J. Cha, LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory, IEEE Trans. On Consumer Electronics 54 (3) , 2008, pp.1215-1223.

[8] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, J. Cha, LIRS-WSR: Integration of LIRS and Writes Sequence Reordering for Flash Memory, in: ICCSA'07, Vol. 4705 of LNCS, 2007, pp. 224-237.

[9] H. M. Deitel, Operating Systems, Chap.9 Virtual Storage Management (Pearson A Education, Second Edition).

[10] H. Paajanen, Page Replacement In Operating System Memory Management, Master’s Thesis in Information Technology, University of Jyväskylä, Department of Mathematical Information Technology(October 23, 2007).

[11] J. Kim, J. M. Kim, S. H. Noh, S. L. Min, Y. Cho, A Space-Efficient Flash Translation Layer for Compact Flash Systems, IEEE Transactions on Consumer Electronics, Vol. 48, No. 2, May 2002.

[12] L. P. Chang, T. W. Kuo, Efficient Management for Large-Scale Flash-Memory Storage Systems with Resource Conservation, ACM TOS 1 (4), 2005, pp.381-418.

[13] M. L. Chiang, C.H. Paul, R.C. Chang, Manage Flash Memory in Personal Communicate Devices. In: Proc. of IEEE Intl. Symposium on Consumer Electronics, IEEE Computer Society Press, Los Alamitos, 1997.

[14] M.L. Singh, Understanding Research Methodology, Chap.1 Scientific Method and Research, pp 4.

[15] N. Megiddo, D. S. Modha, ARC: A Self-Tuning, Low Overhead Replacement Cache, in: FAST'03, 2003.

[16] P. Jin, Y. Ou, T. Harder, Z. Li, AD-LRU: An Efficient Buffer Replacement Algorithm for Flash-Based Databases. Data Knowl. Eng. : 72, 2012, pp. 83-102.

[17] S. Bansal, D. Modha, CAR: Clock with Adaptive Replacement, In Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04), San Francisco, 2004, pp 187-200.

[18] S. Jiang, X. Zhang, F. Chen, CLOCK-Pro: An Effective Improvement of the Replacement, ATEC '05 Proceedings of the annual conference on USENIX Annual Technical Conference, 2005, pp. 35.

[19] S. Jiang, X. Zhang, LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance, ACM SIGMETRICS Performance Evaluation Review archive Vol. 30, Issue 1, 2002, pp. 31-42.

[20] S. Park, D. Jung, J. Kang, J. Kim, J. Lee, CFLRU: A Replacement Algorithm for Flash Memory, in: CASES'06, 2006, pp. 234-241.

[21] T. Johnson, D. Shasha, 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm, Proceedings of the 20th International Conference on VLDB, 1994, pp 439-450.

[22] W. Kim and S. W. Lee: "On Flash-Based DBMSs: Issues for Architectural Re-Examination", in Journal of Object Technology, vol. 6, no. 8, September-October 2007, pp. 39-49, http://www.jot.fm/issues/issue_2007_09/column4.

[23] Y. Ou, T. Harder, P. Jin, CFDC: a Flash-Aware Replacement Policy for Database Buffer Management, in: Proc. of the 5th International Workshop on Data Management on New Hardware, ACM, 2009, pp. 15-20.

[24] Y. Smaragdakis, S. Kaplan, P. Wilson, EELRU: Simple and Effective Adaptive Page Replacement, In Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'99), Atlanta, 1999, pp. 122-133.

[25] Z. Li, P. Jin, X. Su, K. Cui, L. Yue, CCF-LRU: A New Buffer Replacement Algorithm for Flash Memory, Trans. on Cons. Electr. 55, 2009, pp. 1351-1359.

Bibliography

- B. Saud, Sensitivity Analysis of Cache Partition in Clock-Pro Page Replacement and its Comparison with Adaptive Clock-Pro, Master's Thesis, Tribhuvan University, Central Department of Computer Science and Information Technology.
- B. Subedi, An Evaluation of Page Replacement Algorithm Based on Low Inter-reference Recency Set on Weak Locality Workloads, Master's Thesis, Tribhuvan University, Central Department of Computer Science and Information Technology.
- Toshiba America Electronic Components, Inc., NAND Flash Applications Design Guide, 2004.
- http://www.cse.ohio-state.edu/~zhang/influential-papers.html
- http://www.docstoc.com/docs/21106969/Role-of-OS-in-virtual-memory-management

Appendix 1 Sample Random Trace input

1,8575	0,17754	0,33289	0,3838	0,19942	1,25113
1,35145	1,1939	0,40780	0,12831	0,31724	1,37162
1,861	1,35912	0,39216	1,10863	0,15454	0,32425
0,42141	1,34769	0,29923	1,3050	0,4043	0,39113
1,11686	1,25837	0,4941	0,7882	0,39262	1,32631
0,36490	1,11934	1,8851	1,16962	0,37665	1,23980
0,41727	0,15074	1,19029	1,1750	0,49554	1,18797
1,6747	0,31276	1,786	1,42798	0,30971	0,42594
1,49503	0,23075	1,8717	1,13521	1,988	0,22467
1,12586	1,45284	1,39329	0,45058	0,14795	0,21120
0,7786	1,43211	0,47655	0,42213	0,919	0,603
0,4844	0,44923	1,29324	0,26292	1,31526	1,38097
1,39819	0,30117	1,14208	1,27844	1,8361	1,16455
1,5699	0,10670	1,1066	0,9039	1,6477	1,41170
0,23504	1,32354	0,14280	1,36795	1,8732	1,46002
1,4880	1,5637	1,21680	1,3496	1,3220	0,13282
1,42670	0,11669	0,2716	1,49749	0,12437	0,42550
0,27038	0,26790	1,44095	1,25674	0,4498	1,32206
0,33123	0,9846	0,46190	0,20089	0,14060	1,28875
1,16434	1,12575	1,47687	1,41433	0,16610	0,3411
1,23633	1,17429	0,49681	1,25625	1,34155	0,33804
0,21089	0,16647	1,3104	0,3843	1,7142	0,30193
0,12695	1,28453	0,9115	0,25532	0,47722	1,47868
1,49752	0,6476	1,41825	1,7631	0,14127	0,29127
1,12805	0,48855	1,33911	0,41079	1,25483	1,39430
0,1037	0,3297	0,16599	1,36036	0,15578	0,10091
1,25578	0,23037	0,24073	1,16386	0,15490	1,1048
0,19682	0,8798	0,26493	0,48889	1,7791	1,35987
1,16638	1,45825	1,38057	0,30566	0,48228	0,38949
1,47502	0,26137	1,22920	1,32430	1,7944	1,35589
1,40867	1,47773	0,46838	1,44616	1,39286	1,39175
0,42242	1,20480	1,22293	0,20389	0,23900	0,18555

Appendix 2 Sample Read most Input Trace

0,47138	0,8885	0,46509	0,30725	1,15160	0,2460
0,9807	0,46791	1,5087	0,11237	0,22932	0,37902
0,6713	0,34922	0,4119	0,42689	0,25737	0,39402
0,9355	0,10606	0,641	0,27320	0,38193	0,21972
0,42518	0,10783	0,28314	1,1900	0,13867	0,39219
0,46605	1,38017	0,46494	0,23527	0,38630	1,21176
0,293	0,12907	0,39277	0,40610	0,7266	1,41366
0,30769	1,8749	1,10029	0,1320	0,46614	0,41918
0,26128	0,41673	0,19547	0,48693	0,37972	0,38947
0,15954	0,3438	0,18472	0,16481	0,6566	0,9291
0,43502	1,33032	0,3183	0,19948	0,6053	1,38512
0,46694	0,33131	0,29974	0,19584	0,49468	1,24278
0,17376	0,46130	0,4161	0,3133	0,45468	0,35567
0,36470	0,24196	1,34021	0,39449	0,18771	0,19982
0,26021	0,17350	0,44669	0,11232	0,2877	0,14913
0,26197	0,37578	0,44932	0,27057	0,8577	0,21545
1,19614	0,26010	0,31719	0,21978	0,9246	0,32690
0,35125	0,29523	0,34981	1,3135	1,2971	0,1054
0,15836	0,29720	0,39483	0,42668	0,23341	1,7058
1,37083	0,5836	0,39234	0,30664	0,47423	0,48384
0,49832	0,47732	0,6181	0,28049	0,20673	0,14815
1,16584	0,35416	0,15178	1,22743	0,37824	0,20809
0,43815	0,7992	1,22767	1,981	0,6349	0,22302
0,1909	0,37810	0,24271	0,27349	0,21940	0,11289
0,3186	0,14000	0,38546	1,20359	0,34039	0,3939
0,3492	0,44098	0,2151	0,17422	0,30562	1,24662
0,23074	0,26344	1,31895	0,6416	0,48410	0,15522
0,14390	0,34163	0,13073	0,19750	0,985	0,48011
1,18012	0,11608	0,14481	0,34997	0,22648	0,26672
0,15980	0,49335	1,34079	0,11814	0,31534	0,20259
0,11874	0,45185	1,20792	0,39186	0,18681	0,24097
0,8582	0,26107	1,11335	0,33248	0,31662	0,47539

Appendix 3 Sample Write most Input Trace

1,12527	1,1216	1,698	1,35286	1,39722	1,25887
1,45028	1,47558	1,44966	1,10018	1,41052	0,8011
1,42731	1,9714	1,39263	1,40196	1,6269	1,39623
1,33031	1,1853	1,29107	1,5242	1,1010	1,28122
1,35606	1,27792	1,19845	1,24155	1,20899	1,37819
1,27592	1,1272	1,2536	1,35733	1,33645	1,37360
1,13287	1,35073	1,24973	1,31865	1,7424	1,5993
1,8751	1,2237	1,39556	1,8440	1,35811	1,25015
1,42880	1,12603	1,8230	1,45262	1,10924	0,40802
1,24112	1,38237	1,31304	1,5412	1,43801	1,29898
1,10638	1,47683	1,4487	1,44810	1,8571	1,9911
1,33896	0,35169	1,35950	1,9344	1,2859	1,32483
1,2158	1,46525	1,32777	1,20380	0,25035	1,5188
1,6797	0,24879	1,8889	1,19975	1,8644	1,8494
1,17945	1,5175	1,29078	1,36322	1,46605	0,3722
1,23254	1,35573	1,44707	1,16353	1,23944	1,24724
0,40235	1,9453	1,33001	1,23185	1,19468	1,4818
1,18662	1,14189	0,1378	1,16011	1,18092	1,36090
1,37183	1,4364	1,33538	1,41008	1,19253	1,34763
1,21453	1,5052	1,38178	1,39783	1,33887	1,46310
1,2396	1,41563	1,18490	1,18554	1,46076	1,3812
1,46712	1,22442	0,15937	1,38230	1,45473	1,6945
1,24479	1,9632	1,21724	1,12421	1,20451	1,35388
0,980	1,4486	1,47436	1,44968	1,42560	1,34505
1,42484	1,8868	1,13237	1,45460	1,40381	1,46871
1,18937	1,1389	1,22092	1,20688	1,30869	0,45818
1,47306	1,3497	1,1803	1,6096	1,24012	1,43783
1,7630	1,24744	1,47367	1,42187	1,43951	1,21302
1,26076	1,12092	0,38106	1,21666	1,45645	1,12638
1,5712	0,14779	1,33647	1,29306	1,20191	1,33315
1,26443	1,11996	1,28139	1,18374	1,24340	1,26206
1,6606	1,1590	1,16723	1,48509	1,29078	1,36414

Appendix 4 Sample zipf Input Trace

1,8550	0,3609	1,654	1,17913	0,145	0,2550
1,5970	0,2461	1,33806	0,17	0,1	0,17
0,1,370	0,159	0,10290	0,54	0,4	1,40078
0,481	0,14300	0,1	0,16	0,18	1,1167
0,7	1,27473	0,47	0,127	0,286	0,35
1,1	0,63	0,15	0,17	0,574	1,1815
0,173	0,6	0,9172	0,5565	1,69	1,7723
0,39491	0,2020	1,1	0,16	1,17217	1,3717
1,3294	1,31	1,40143	0,49198	0,15221	0,191
0,49491	1,2842	1,2797	0,25825	0,7165	1,40
0,3	1,47	0,6	0,8631	0,7375	0,9649
0,3530	0,21	1,508	1,8	0,16	1,20349
1,4506	1,279	1,111	0,1472	0,2768	0,36002
0,168	0,631	0,50	0,44415	1,800	0,1847
0,1353	0,115	0,28497	1,2611	0,697	1,1728
0,1	1,32	0,57	1,358	0,522	1,4
0,612	0,2599	1,2	1,5	0,47719	1,8
1,889	0,345	0,1136	0,242	1,10958	0,1178
0,17	1,3	1,20063	0,1992	0,4	1,7485
1,1406	0,168	1,87	1,602	0,1638	0,265
0,15042	1,42	0,16832	0,12	0,49373	0,2
1,2880	0,28	1,761	0,2	1,412	0,30
0,40	0,1244	0,146	1,9	0,30606	0,3
1,2	1,327	0,27188	0,29109	0,4	0,22156
0,145	1,11837	1,12173	1,41	0,49	0,11
1,1256	0,13969	1,18099	1,202	0,9684	0,31846
0,1303	1,5233	1,109	0,35427	0,29287	0,2080
1,74	0,303	1,34	1,4342	1,172	1,46482
0,4	1,30884	0,10994	0,1	1,36777	1,22311
0,4502	1,104	0,2	0,2258	0,26534	0,10
1,1693	1,15000	1,890	0,14941	1,5200	1,89

Appendix 5 Source Code for LIRS-WSR Algorithm

```
//LIRS WSR algorithm
void LIRSWSR()
{
  char ref_block[9],s[9];
  char r;
  int i,j;
  int flag;
  struct node *temp,*newnode;
  struct node *hold;

  fseek(trace_fp,0, SEEK_SET);

  fscanf(trace_fp,"%s",s);
   while (!feof(trace_fp))
  {

     for(i=0;i<9;i++)
          ref_block[i] = '\0';
      if(s[0]=='1')
           r='W';
      else
          r='R';
     for(i=2,j=0;s[i]!='\0';i++,j++)
          ref_block[j] = s[i];
     ref_block[j] = '\0';

     flag =0;
     total_pg_refs++;

     for(i=1;i<=distinct_refs;i++)
      {
        if(strcmp(history[i],ref_block)==0)
          {
                flag =1;
               break;
          }
       }
     if(flag == 0)
      {
           distinct_refs++;
           strcpy(history[distinct_refs],ref_block);
      }

  if(r=='W')
     write_request++;

// isin cache
// Search for hit in LIRS Stack
    temp= LIRS_stack_head;
```

```
while(temp!=NULL)
{
  flag =0;
 if(strcmp(ref_block,temp->pn)==0)
  {
   if(temp->isHIR_block==0)
   {
     if(r=='W')
          temp->r='W';
      move_to_LIRS_stack_head(temp);
      temp->cold =0;
      if(temp->LIRS_next == NULL)
          stack_prune();

    }
   else if(temp->isHIR_block==1)
    {
      if(temp->isResident ==1)
      {
         if(r=='W')
                temp->r='W';
         temp->cold =0;
         move_to_LIRS_stack_head(temp);
         remove_from_HIR_List(temp);
        while(LIRS_stack_tail->r=='W' && LIRS_stack_tail->cold==0)
           {
               move_to_LIRS_stack_head(LIRS_stack_tail);
               LIRS_stack_head->cold=1;
               stack_prune();
           }
         add_to_HIR_List(LIRS_stack_tail);
         LIRS_stack_tail=LIRS_stack_tail->LIRS_prev;
         LIRS_stack_tail-> LIRS_next->LIRS_prev = NULL;
         LIRS_stack_tail->LIRS_next=NULL;
         stack_prune();
       }
       else
       {
          num_pg_flt++;
          if(r=='W')
            temp->r='W';
          else
            temp->r='R';
          remove_HIR_tail();

          move_to_LIRS_stack_head(temp);
          temp->cold=0;
          while(LIRS_stack_tail->r=='W' && LIRS_stack_tail->cold==0)
          {
              move_to_LIRS_stack_head(LIRS_stack_tail);
```

```
                LIRS_stack_head->cold=1;
                stack_prune();
              }
            add_to_HIR_List(LIRS_stack_tail); // move stack bottom to HIR list head
            LIRS_stack_tail=LIRS_stack_tail->LIRS_prev;
            LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
            LIRS_stack_tail->LIRS_next=NULL;
            stack_prune();
           }
         }

       flag =1;
       break;
     }
    temp=temp->LIRS_next;
   }
   if(flag == 1)
    {
          fscanf(trace_fp,"%s",s);
          continue;
     }
   else // Not in stack
   {

     // Search for hit in HIR Q
      temp = HIR_list_head;
     while(temp!= NULL) // hit in cache
     {
       flag=0;
       if(strcmp(ref_block,temp->pn)==0)
       {
          if(r=='W')
            temp->r='W';
          move_to_head_HIR_List(temp);
          add_LIRS_stack_head(temp);
          flag=1;
          break;
       }

       temp= temp->HIR_rsd_next;
     }
  }
if(flag == 1)
{
   fscanf(trace_fp,"%s",s);
   continue;
    }
    else
    {
      // printf("\ngenerates block miss */");
```

```
 num_pg_flt++;
 if (free_mem_size == 0)
 {
        remove_HIR_tail();

         newnode = (struct node *) malloc(sizeof(struct node));
         newnode->isHIR_block = 1;
         newnode->isResident  = 1;
         strcpy(newnode->pn,ref_block);
         newnode->recency = 1;
         newnode->LIRS_next = NULL;
         newnode->LIRS_prev = NULL;
         newnode->HIR_rsd_next = NULL;
         newnode->HIR_rsd_prev  = NULL;
         if(r=='W')
               newnode->r='W';
         else
               newnode->r='R';

       if(newnode->r=='W')
        {
           add_LIRS_stack_head(newnode);
           newnode->isHIR_block = 0;
          while(LIRS_stack_tail->r=='W' && LIRS_stack_tail->cold==0)
            {
              move_to_LIRS_stack_head(LIRS_stack_tail);
              LIRS_stack_head->cold=1;
              stack_prune();
            }
          add_to_HIR_List(LIRS_stack_tail);
          LIRS_stack_tail=LIRS_stack_tail->LIRS_prev;
          LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
          LIRS_stack_tail->LIRS_next=NULL;
          stack_prune();
        }
        else
        {
          add_to_HIR_List(newnode);
          add_LIRS_stack_head(newnode);
        }

        fscanf(trace_fp,"%s",s);
         continue;
     }

else if( free_mem_size> HIR_block_portion_limit) //to place page in LIR block
  {
     newnode = (struct node *) malloc(sizeof(struct node));
     newnode->isHIR_block = 0;
     newnode->isResident  = 0;
```

```
      strcpy(newnode->pn,ref_block);
      newnode->recency = 1;
      newnode->LIRS_next = NULL;
      newnode->LIRS_prev = NULL;
      newnode->HIR_rsd_next = NULL;
      newnode->HIR_rsd_prev  = NULL;
      if(r=='W')
            newnode->r='W';
      else
            newnode->r='R';
      add_LIRS_stack_head(newnode);
      free_mem_size--;
      fscanf(trace_fp,"%s",s);
      continue;

 }
else    //to place page in HIR block
 {
      newnode = (struct node *) malloc(sizeof(struct node));
      newnode->isHIR_block = 1;
      newnode->recency = 1;
      newnode->isResident = 1;
      strcpy(newnode->pn,ref_block);
      newnode->LIRS_next = NULL;
      newnode->LIRS_prev = NULL;
      newnode->HIR_rsd_next = NULL;
      newnode->HIR_rsd_prev  = NULL;
      if(r=='W')
            newnode->r='W';
      else
            newnode->r='R';

     if(newnode->r=='W')
      {
           add_LIRS_stack_head(newnode);
           newnode->isHIR_block = 0;
           while(LIRS_stack_tail->r=='W' && LIRS_stack_tail->cold==0)
               {
                  move_to_LIRS_stack_head(LIRS_stack_tail);
                  LIRS_stack_head->cold=1;
                  stack_prune();
                }
            add_to_HIR_List(LIRS_stack_tail);
            LIRS_stack_tail=LIRS_stack_tail->LIRS_prev;
            LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
            LIRS_stack_tail->LIRS_next=NULL;
            stack_prune();
            free_mem_size--;
          }
       else
```

```
                    {
                        add_LIRS_stack_head(newnode);
                        add_to_HIR_List(newnode);
                        free_mem_size--;
                    }

                    fscanf(trace_fp,"%s",s);
                   continue;
                }
              }
         } //while closed

         hold = LIRS_stack_head;
         while(hold!=NULL)
         {
            if(hold->r=='W')
                        write_count++;
             hold=hold->LIRS_next;
           }
        hold = HIR_list_head;
        while(hold!=NULL)
         {
           if(hold->recency== 0 && hold->r=='W')
                 write_count++;
            hold=hold->HIR_rsd_next;
          }
}// End of LIRSWSR function
```

Appendix 6 Source Code for AD-LRU algorithm

```
//ADLRU algorithm
void ADLRU()
{

  char ref_block[9],s[15];
  char r;
  int i,j;
  int flag;
  struct node * temp, * newnode;
  struct node *hold;
  fseek(trace_fp,0, SEEK_SET);

  fscanf(trace_fp,"%s",s);

  while (!feof(trace_fp))
  {
     for(i=0;i<15;i++)
      ref_block[i] = '\0';

    if(s[0]=='1')
      r='W';
    else
      r='R';
    for(i=2,j=0;s[i]!='\0';i++,j++)
      ref_block[j] = s[i];
   ref_block[j] = '\0';
   total_pg_refs++;
   flag =0;
   for(i=1;i<=distinct_refs;i++)
     {
       if(strcmp(history[i],ref_block)==0)
         {
           flag =1;
           break;
         }
      }
    if(flag == 0)
     {
          distinct_refs++;
          strcpy(history[distinct_refs],ref_block);
     }
    if(r=='W')
      write_request++;

   // is in hot LRU or not
    temp = HOT_lru_head;
    while(temp!=NULL)
    {
```

```
     flag =0;
    if(strcmp(temp->pn,ref_block)==0)
    {
       if(r=='W')
           temp->r='W';
       if(temp->reference ==0)
          temp->reference=1;

       move_to_head_of_hotq(temp);
       adjust_Hot_FC();
       flag = 2;
       break;
    }

      temp= temp->next;
 } // while closed

 if(flag==2)
  {
         fscanf(trace_fp,"%s",s);
          continue;
  }
  else
  {
    temp = COLD_lru_head;
     while(temp!=NULL)
     {
       flag =0;
       if(strcmp(temp->pn,ref_block)==0)
       {
          if(r=='W')
             temp->r='W';

          if(temp->reference ==0)
             temp->reference=1;

         remove_from_coldq(temp);
         add_to_head_of_hotq(temp);
         adjust_Cold_FC();
         cold_counter--;
           flag = 3;
             break;
       }

      temp= temp->next;
    } // while closed
  }  // else closed
if(flag == 3)
 {
         fscanf(trace_fp,"%s",s);
```

```
                continue;
          }
          else
          {
                // printf("\ngenerates block miss */");
               num_pg_flt++;

               if (free_mem_size >0)  // there is space in buffer
               {

                 newnode = (struct node *)malloc(sizeof(struct node));
                 strcpy(newnode->pn,ref_block);
                 newnode->cold=1;
                 newnode->reference =1;
                 newnode->prev = NULL;
                 newnode->next = NULL;
                  if(r=='W')
                     newnode->r='W';
                 else
                     newnode->r='R';
                 add_to_head_of_coldq(newnode);
                 cold_counter++;
                 free_mem_size--;

               }
              else    //memory full
              {
                 newnode = (struct node *)malloc(sizeof(struct node));
                 strcpy(newnode->pn,ref_block);
                 newnode->cold=1;
                 newnode->reference =1;
                 newnode->prev = NULL;
                 newnode->next = NULL;
                 if(r=='W')
                     newnode->r='W';
                else
                     newnode->r='R';
                 if(cold_counter>COLD_MIN_limit)
                       select_victim_from_coldq();
                  else
                      select_victim_from_hotq();
                 add_to_head_of_coldq(newnode);
                 cold_counter++;
                }
              fscanf(trace_fp,"%s",s);
         }  // else closed
} // while closed
         while(HOT_lru_head!=NULL)
        {
```

```
        if(HOT_lru_head->r=='W')
          {
            write_count++;

          }
         hold = HOT_lru_head;
         HOT_lru_head= HOT_lru_head->next;
         free(hold);
       }

       while(COLD_lru_head!=NULL)
       {
        if(COLD_lru_head->r=='W')
          {
            write_count++;
          }
          hold= COLD_lru_head;
          COLD_lru_head= COLD_lru_head->next;
          free(hold);
       }
} // end of ADLRU function
```