Tribhuvan University
Institute of Science and Technology

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

Dissertation
Submitted to

Central Department of Computer Science & Information Technology
Kirtipur, Kathmandu, Nepal

In partial fulfillment of the requirements
for the Masters Degree in Computer Science & Information Technology

By
Mahesh Kumar Yadav
Date: 05 June, 2016

Tribhuvan University
Institute of Science and Technology

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

Dissertation
Submitted to

Central Department of Computer Science & Information Technology
Kirtipur, Kathmandu, Nepal

In partial fulfillment of the requirements
for the Masters Degree in Computer Science & Information Technology

By
Mahesh Kumar Yadav
Date: 05 June, 2016

Supervisor
Prof. Dr. Subarna Shakya, PhD

Co-Supervisor
Arjun Singh Saud

Tribhuvan University
Institute of Science and Technology
Central Department of Computer Science & 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.

... ...

Mahesh Kumar Yadav
contact@maheshyadav.com.np
Date: 05 June, 2016

Supervisor's Recommendation

I hereby recommend that this dissertation prepared under my supervision by **Mr. Mahesh Kumar Yadav** entitled **"A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and CCF-LRU for Flash Memory Based Systems"** in partial fulfillment of the requirements for the degree of M.Sc. in Computer Science and Information Technology be processed for the evaluation.

... ...

Prof. Dr. Subarna Shakya, PhD
Central Department of Computer Science and Information Technology (CDCSIT),
Institute of Engineering (IOE),
Pulchowk, Nepal

Date: 05 June, 2016

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

LETTER OF APPROVAL

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

Evaluation Committee

... ...
Asst. Prof. Nawaraj Paudel
CDCSIT, Tribhuvan University (TU)
Kirtipur, Kathmandu, Nepal
(Act. Head)

... ...
Prof. Dr. Subarna Shakya
CDCSIT, IOE, TU
Pulchowk, Kathmandu, Nepal
(Supervisor)

... ...
Assoc. Prof. Mr. Bal Krishna Bal
(External Examiner)

... ...
Mr. Bishnu Gautam
(Internal Examiner)

Date: 22 June, 2016

ACKNOWLEDGEMENTS

I am very happy to complete this dissertation **"A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and CCF-LRU for Flash Memory Based Systems"**. This research work has been performed under Central Department of Computer Science and Information Technology (CDCSIT), Tribhuvan University (TU), Kirtipur. I am very grateful to my department for giving me an enthusiastic support.

First of all, I would like to express my gratitude to my supervisor Prof. Dr. Subarna Shakya, Institute of Engineering (IOE), Pulchowk, you have been a tremendous mentor for me. This research would not have been possible without his advice and patience. I deeply extend my hearty acknowledgment to my co-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 dissertation and showed me the way to overcome them. Most importantly I would like to thank respected Head of Department of CDCSIT. My sincere thanks also goes to all the respected teachers Prof. Dr. Shashidharram Joshi, Prof. Sudarshan Karanjit, 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: my parents Gena Ray Yadav and Sugandhi Devi and to my brothers. Words cannot express how grateful I am to father and brothers for all of the sacrifices that they've made on my behalf.

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

Last and not least: I beg forgiveness of all those who have been with me over the course of the years and whose names I have failed to mention.

This dissertation is available on `https://maheshyadav.com.np/dissertation/`

ABSTRACT

Flash memory has the characteristics of not-in-place update and asymmetric Input/output (I/O) costs among read, write, and erase operations, in which the cost of write/erase operations is much higher than that of read operation. Among different flash-aware buffer replacement algorithms Low Inter-reference Recency Set Write Sequence Reordering (LIRS-WSR) and Cold Clean First Least Recently Used (CCF-LRU) are two buffer replacement policies that can be suitable for flash based systems. LIRS-WSR enhances Low Inter-reference Recency Set (LIRS) by reordering the writes of not-cold-dirty pages from the buffer cache to flash storage to focus on the reduction of the number of write/erase operations as well as preventing serious degradation of buffer hit ratio. CCF-LRU, which enhances the previous Clean First Least Recently Used (CF-LRU) and Least Recently Used Write Sequence Reordering (LRU-WSR) methods by differentiating clean pages into cold and hot ones, and evicting cold clean pages first and delaying the eviction of hot clean pages.. The objective of this dissertation is mainly focused on evaluating the performance of LIRS-WSR and CCF-LRU buffer replacement algorithms. Finally, the comparative study based on quantitative analysis of those algorithms is performed based on the hit/miss rates and the number write counts.

Using the trace-driven simulation, when workload has high reference locality, CCF-LRU has significantly superior performance than LIRS-WSR in terms of both hit rate and write count. CCF-LRU has higher hit rate up to 22% and minimizes write count up to 40% in comparison to LIRS-WSR. For uniformly distributed workloads, the difference in hit rates and write count of CCF-LRU and LIRS-WSR is comparatively small. CCF-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:

Buffer Replacement Algorithm, CCF-LRU, Flash memory, Hit Rate, LIRS, LIRS-WSR, Least Recently Used (LRU), Write Count

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

2Q Two Queue

μs Microsecond

AD-LRU Adaptive Double Least Recently Used

ARC Adaptive Replacement Cache

CCF Cold Clean First

CAR Clock with Adaptive Replacement

CCF-LRU Cold Clean First Least Recently Used

CDCSIT Central Department of Computer Science and Information Technology

CFDC Clean First Dirty Clustered

CF-LRU Clean First Least Recently Used

CLOCK-Pro Clock with Pro

CPU Central Processing Unit

DRAM Dynamic Random Access Memory

EELRU Early Eviction Least Recently Used

EEPROM Electrically Erasable Programmable Read Only Memory

FIFO First In First Out

GCLOCK Generalized CLOCK

GiB Giga Byte

GHz Giga Hertz

HIR High Inter-reference Recency

HIRS High Inter-reference Recency Set

I/O Input/output

IOE Institute of Engineering

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 Milliampere

MDR Memory Data Register

MRU Most Recently Used

ms Millisecond

NRU Not Recently Used

OLTP Online Transaction Processing

OPT or MIN OPTimum or MINimum

OS Operating System

PC Program Counter

PDA Personal Digital Assistant

RAM Random Access Memory

TU Tribhuvan University

USB Universal Serial Bus

WSR Write Sequence Reordering

Chapter 1

BACKGROUND AND PROBLEM FORMULATION

1.1 BACKGROUND

1.1.1 Memory Hierarchy

The memory hierarchy is the hierarchy of memory and storage devices found in a computer. Often visualized as a triangle, the bottom of the triangle represents larger, cheaper and slower storage devices, while the top of the triangle represents smaller, more expensive and faster storage devices. Figure 1.1 shows the hierarchy of memories used in a computer system with

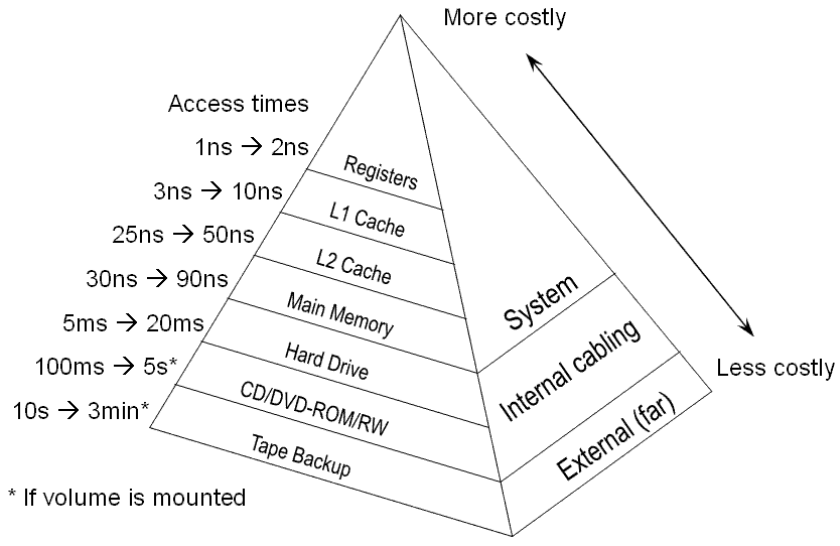


Figure 1.1: Computer Memory Hierarchy

their speed and memory capacity. The arrangement of memory devices in a computer system is such that faster memory is at the top level and slower memory is at the bottom. The overall performance of computer system depends on management and organization of such memories. All the memory management policies are automatically handled by Operating System (OS) and devices are arranged according to as the principles followed by it. Different types of memories available up to now can be categorized into two major groups. They are a 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 Register

Register are used to quickly accept, store and transfer data and instructions that are being used immediately by the Central Processing Unit (CPU), there are various types of Registers those are used for various purpose. Among of the some Mostly used Registers named as AC or Accumulator, Data Register or DR, the AR or Address Register, Program Counter (PC), Memory Data Register (MDR), Index register, Memory Buffer Register. These Registers are used for performing the various Operations. While we are working on the System then these Registers are used by the CPU for Performing the Operations. When We Give Some Input to the System then the Input will be Stored into the Registers and When the System will give us the Results after Processing then the Result will also be from the Registers. So that they are used by the CPU for Processing the Data which is given by the User.

1.1.3 Cache

The cache is a very fast copy of the slower main system memory. Cache is much smaller than main memories because it is included inside the processor chip alongside the registers and processor logic. This is prime real estate in computing terms and there are both economic and physical limits to its maximum size. As manufacturers find more and more ways to cram more and more transistors onto a chip cache sizes grow considerably, but even the largest caches are tens of megabytes, rather than the gigabytes of main memory or terabytes of hard disk otherwise common. The cache is made up of small chunks of mirrored main memory. The size of these chunks is called the line size and is typically something like 32 or 64 bytes. When talking about the cache, it is very common to talk about the line size, or a cache line, which refers

to one chunk of mirrored main memory. The cache can only load and store memory in sizes a multiple of a cache line. Generally, a computer system consists of different levels of cache that are L1 cache and L2 cache. L1 cache is internal cache nearby register and the L2 cache is external cache nearby Random Access Memory (RAM). L1 cache is faster than L2 cache. If L3 cache is available, then it acts as an earlier L2 cache. Hence L2 works as an intermediate cache between L1 cache and L3 cache. Increasing the level of cache doesn't always increase the overall performance. Up to limited cache level, the performance gain can be achieved. If there are more levels of cache, access time will increase due to swapping the blocks back and forth. Hence after crossing certain limitation of cache level overall performance slows down instead of increasing.

1.1.4 Primary Memory

Primary memory is a computer system's volatile storage mechanism. It may be RAM, cache memory and data buses but is primarily associated with RAM. As soon as a computer starts, primary memory stores all running applications, including the base OS, user interface and any user installed and running software utility. A program/application that is opened in primary memory interacts with the system processor to perform all application specific tasks. Primary memory is considered faster than secondary memory.

1.1.5 Secondary Memory

Secondary memory is taken as the backup memory. It consists of a massive volume of data. Comparatively, it is cheaper, slower and less reliable. Secondary memory is an external memory such as hard disk, optical disk, pen drive, flash memory etc.

1.1.6 Flash Memory

Flash memory is non-volatile, shock resistant and power economic. With recent technology breakthroughs in both capacity and reliability, flash-memory storage systems are much more affordable than ever. As a result, flash memory is now among the top choices for storage media in embedded systems [4]. There are two major types of flash memory in the current market: NAND and NOR flash memory. NAND flash memory is mainly designed for data storage and

NOR flash memory is for Electrically Erasable Programmable Read Only Memory (EEPROM) replacement [4]. Flash memory has become a powerful and cost-effective solid-state storage technology widely used in mobile electronics devices and other consumer applications. NAND Flash, which was designed with a very small cell size to enable a low cost-per-bit of stored data, has been used primarily as a high-density data storage medium for consumer devices such as digital still cameras and Universal Serial Bus (USB) solid-state disk drives. NOR Flash has typically been used for code storage and direct execution in portable electronics devices, such as cellular phones and Personal Digital Assistants (PDAs) [5]. However, several hardware limitations exist in a flash memory. Firstly, a data unit of erase operations is a block that is the set of fixed number of contiguous pages even if a data unit of read/write operations is a page. Secondly, it is impossible to re-write the page in-place in a flash memory. So, in order to update data of the page, a system should perform only one of the following:

1. Writing these data to a newly allocated page and invalidating the original page; or
2. Writing these data to the original page only after erasing the block containing that page.

Thirdly, the lifetime of a flash memory is shorter than the lifetime of a hard disk and a Dynamic Random Access Memory (DRAM). In other words, only a limited number of erase operations can be performed safely to each memory cell, typically, between 100,000 and 1,000,000 cycles. Finally, there exist differences among I/O latencies according to the kinds of I/O operations, i.e., read, write and erase. The write operation is about 10 times slower than the read operation and the erase operation is about 20 times slower than the write operation [6, 7, 8].

Flash caching is needed for reducing flash I/O latencies. The traditional magnetic-disk-based buffering algorithms LRU [9], LIRS [1], Adaptive Replacement Cache (ARC) [10] 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 writes and erases operations on flash memory and at the same time prevent the degradation of the hit ratio. Recently, CF-LRU [11], LIRS-WSR [6] and Adaptive Double Least Recently Used (AD-LRU) [12] 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 try to obtain the optimal

I/O sequence from the given I/O sequence by discriminatively selecting the accesses according to the type of I/O operations. These algorithms favor to first evict clean pages from the buffer so that the number of writes incurring for replacements can be reduced.

Table 1.1: Characteristics of flash memory [3]

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

Compare to the magnetic disk, flash memory has special properties. Firstly, flash memory has no latency associated with the mechanical head movement to locate the proper position to read or write data. Secondly, 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 4 kB data is read, written, or erased [4]. Thirdly, 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. If the erase operations are involved, the cost imbalance would be even worse. Finally, blocks of flash memory are worn out after the specified number of write/erase operations.

1.1.7 Performance Metrics

The off-line performance of buffer replacement algorithm is measured in terms of page fault count, hit rate and hit ratio, miss rate and 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. The Higher hit rate of the algorithm exhibits higher performance. In the case of flash based system, a minimum number of write count is a measure for optimal cost algorithm. flash based system, a minimum number of write count is a measure for optimal cost algorithm.

1.1.7.1 Page Fault Counts

Page Fault is an interrupt generated when the processor references a page that is neither in cache nor in main memory. An efficient page replacement algorithm always produces less number of

page faults. It can be computed by counting the occurrences of a number of page faults between some intervals of references.

1.1.7.2 Hit/Miss Rate

When the processor needs to read or write a location in main memory, it first checks whether that memory location is in the cache. This is accomplished by comparing the address of the memory location to all tags in the cache that might contain that address. If the processor finds that the memory location is in the cache, we say that a **cache hit** has occurred; otherwise, we speak of a **cache miss**.

Miss rate can be calculated by using the formula:

miss rate = 1-hit rate.

Hit ratio is calculated by subtracting miss ratio from 1.

Miss ratio (mr) is calculated by using the formula:

mr = 100 * ((#pf - #distinct) / (#refs - #distinct))

where #pf is a number of page faults, #distinct is the number of distinct pages referenced and #refs is the total number of referenced pages [13].

Hit ratio is also calculated by dividing a total number of hit counts by a total number of reference counts.

Hit ratio = Total number of Hit Counts/Total number of Reference Counts

To represent it as a percentage: Hit% = Hit ratio * 100

1.1.7.3 Write Counts

Write count is a number of pages propagated to flash memory which can be calculated by counting the number of physical pages 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.

1.1.8 Program Behavior

There are several factors that influence the performance of page replacement algorithm. The performance of page replacement algorithm relies on the pattern of pages that are referenced.

The behavior of program depends on the access pattern it references memory which further depends on working set and locality of reference.

1.1.8.1 Locality of Reference

The locality of reference, also known as the principle of locality. 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, the variable used for counting and totaling etc. supports this assumption. Spatial locality is based on space, is assumes that once a memory is referenced there is a 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 the locality are followed in any type memory reference sequence.

1.1.8.2 Memory Reference Patterns

Altogether three types of standard synthetic traces i.e. random traces, read-most traces and write-most traces are used in this dissertation.

1.1.8.2.1 Random Traces The page references having random read and write nature of pages are called random traces.

1.1.8.2.2 Write-most Traces The page references having most of the pages with write mode nature are called write-most traces.

1.1.8.2.3 Read-most Traces The page references having most of the pages with read mode nature are called read-most traces.

1.1.8.2.4 Zipf Traces 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.

1.2 Problem Formulation

1.2.1 Problem Definition

Since the use of flash memory requires buffer replacement policies considering not only buffer hit ratios or miss ratios but also replacement costs incurring when a dirty page has to be propagated to flash memory, not in the buffer. There are many buffer replacement algorithms developed for flash-based systems. The traditional magnetic-disk-based buffering algorithms LRU [9], LIRS [1], ARC [10] etc. focus on hit-ratio improvement alone, but not on write costs caused by the replacement process. However, Flash memory has characteristics of out-of-place update and asymmetric I/O latencies for read-write and erase operations in the aspects of time and energy. So, the replacement algorithm with flash memory should consider not only the hit count but also the replacement cost caused by selecting dirty victim pages. There are many buffer replacement algorithms developed for flash-based systems. The evaluation of these buffer replacement algorithms for flash-based systems in terms of hit rate and write counts is required to rate their performance. This dissertation work will be mainly focused on the comparative evaluation of two algorithms: LIRS-WSR and CCF-LRU.

1.2.2 LIRS-WSR

LIRS-WSR [6] algorithm is designed for a buffer cache of the flash memory based storage system. The objective of LIRS-WSR is reducing 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. It enhances an existing LIRS buffer replacement algorithm with add-on buffer replacement strategy, namely Write Sequence Reordering (WSR). WSR reorders writing not-cold dirty pages from the buffer cache to the disk to reduce the number of write operations while preventing excessive degradation of the hit ratio. The LIRS [1] 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.

LIRS algorithm uses two sets of pages based on IRR. Set of pages with low IRR value is taken

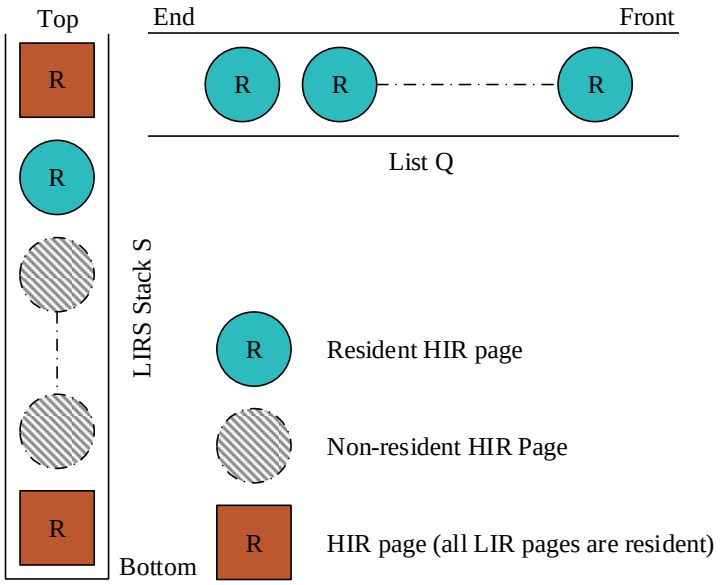


Figure 1.2: Two Lists of the LIRS algorithm [1]

as a hot block and called LIRS. Set of pages with high IRR value is taken as a 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, High Inter-reference Recency (HIR) blocks are always replaced and Low Inter-reference Recency (LIR) blocks are never replaced. LIRS always selects HIR page with the largest recency as a victim for replacement. WSR policy is developed to adapt LIRS with flash memory [6]. Basic scheme of WSR is following:

1. Use cold-detection algorithm to judge whether the page is cold or not; and
2. Delays flushing dirty pages which are not regarded as cold.

LIRS-WSR is implemented using 2 lists: LIR stack S which stores all LIR pages as well as HIR pages regardless of the residence status some of them are resident and others are not (actually, only their meta-data are stored in the list) and HIR list Q that stores HIR resident pages. The operations on these two data structures are same 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 a 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.

1.2.3 CCF-LRU

An efficient new buffer replacement algorithm for flash memory based storage systems called CCF-LRU [2]. The goal of CCF-LRU is to improve the overall I/O performance by focusing on reducing the write count incurring in the replacement process. In order to accomplish this goal, it tries to first evict clean pages with low access frequencies. If there are no such clean pages, it will evict the dirty pages with low access frequencies instead of the clean pages with high access frequencies. Using the cold-detection mechanism of LRU-WSR, pages in the buffer list can be classified into the following four groups, namely cold clean page, hot clean page, cold dirty page and hot dirty page. A cold flag attached with each page is used to distinguish cold pages from hot pages. The CCF-LRU algorithm maintains two LRU lists, which are called

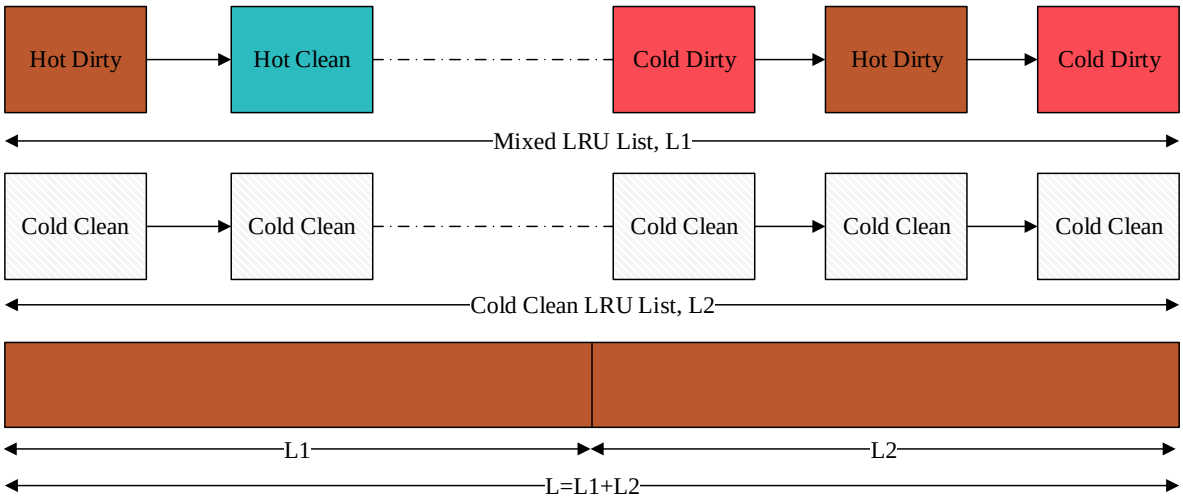


Figure 1.3: Mixed LRU list and cold clean LRU list in CCF-LRU [2]

mixed LRU list and cold clean LRU list. The mixed LRU list contains L1 pages and is used to maintain hot clean pages and dirty pages regardless of the status of its cold flag and the cold clean LRU list with the size of L2 is only for cold clean pages. If the buffer contains a total of L pages, the sizes of the two LRU lists are both from 0 to L. Moreover, the sum of L1 and L2 is L. The first referenced pages are regarded as cold by default, each of which is inserted into the cold clean LRU list with a cold flag. When the page in the cold clean LRU list is referenced again or becomes dirty, it will be moved from the cold clean LRU list to the Most Recently Used (MRU) position in the mixed LRU list. When the page in the mixed LRU list is

referenced, it will be moved to the MRU position of the mixed LRU list. The CCF-LRU selects a victim page by the following rules in order:

1. If the cold clean LRU list is not empty, the LRU page in the cold clean LRU list is selected as the victim; and

2. If the cold clean LRU list is empty, the LRU page in the mixed LRU list is chosen as the victim candidate. If the candidate is a cold dirty page, it is selected as the victim. If the candidate is a hot dirty page, it is labeled as cold and moved to the MRU position of the mixed LRU list. If the candidate is a hot clean page, it is set to cold and moved from the mixed LRU list to the MRU position of the cold clean LRU list and we continue to check the LRU position in the mixed LRU list. If there is no victim found after traversing the mixed LRU list, it needs to call the CCF-LRU algorithm one more time to select a victim.

1.2.4 Objectives

The main objectives of this dissertation work are:

- To perform a comparative study of LIRS-WSR and CCF-LRU buffer replacement algorithms for flash based systems in terms of hit rate and write count; and

- To evaluate the performance of LIRS-WSR and CCF-LRU.

1.2.5 Motivation

Since the use of flash memory requires buffer replacement policies considering not only buffer hit ratios or miss ratios but also replacement costs incurring when a dirty page has to be propagated to flash memory, not in the buffer. As a consequence, a replacement policy should minimize the number of write/erase operations on flash memory and at the same time increase the hit ratio. 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. The 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 the 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.

1.3 Dissertation Organization

Background part of this dissertation work focuses on memory hierarchy and the related basic terms and terminologies of the performance of buffer algorithm which are already mentioned. Some chapters are remaining which clarifies the topics LIRS-WSR and CCF-LRU fulfilling the objectives of this dissertation work.

Chapter 2 consists of literature review which briefly reviews the related topics. The literature review includes the summary of several traditional page replacement. This chapter also contains the research methodology part which shows the flow of 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 the 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 the conclusion of this whole dissertation work and the future work which shows guidelines for further research.

Chapter 2

LITERATURE REVIEW AND 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. Bélády [9] in 1966 developed optimal page replacement algorithm called OPTimum or MINimum (OPT or MIN). His algorithm depends on the 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 the effectiveness of other page replacement algorithms. OPT Replacement algorithm replaces the page that will not be used for the longest period of time by computing maximum forward distance. From the past experiences and research papers, the research on the page replacement algorithms are categorized into LRU based replacement algorithms and CLOCK based replacement algorithms [14].

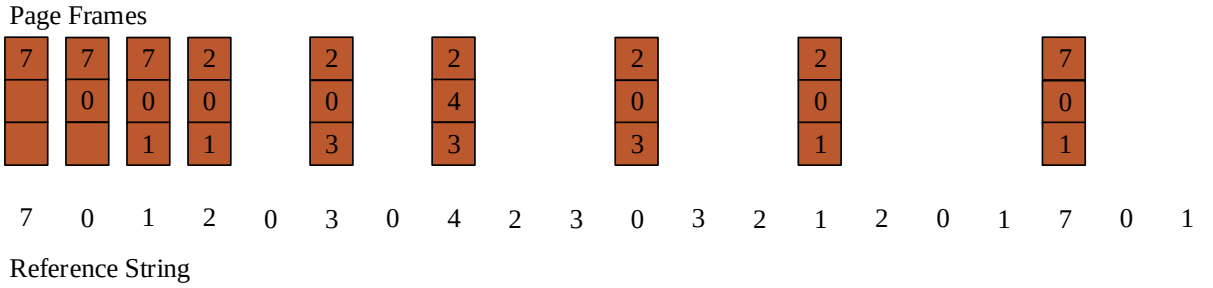


Figure 2.1: Optimal page replacement example

2.1.1.2 LRU Based Page Replacement

2.1.1.2.1 FIFO Page Replacement Algorithm The simplest page-replacement algorithm is a First In First Out (FIFO) [9] algorithm. In this algorithm, operating system keeps track of all pages in the memory in a queue, oldest page is in the front of the queue. When a page needs to be replaced the page in the front of the queue is selected for removal. Conceptually FIFO is a queue with limited size. Initially, the 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 Bélády's Anomaly, a strange situation in which page

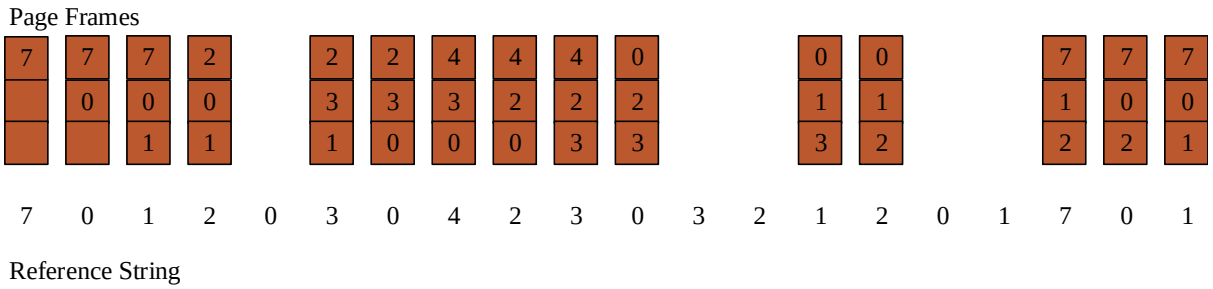


Figure 2.2: FIFO page replacement example

fault increased while increasing number of the page frame. That is, with an 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.

2.1.1.2.2 LRU Page Replacement Algorithm A good approximation to the optimal algorithm is based on the observation that pages that have been heavily used in the last few instructions will probably be heavily used again in the next few. Conversely, pages that have not been used for ages will probably remain unused for a long time. This idea suggests a realizable algorithm: when a page fault occurs, throw out the page that has been unused for the

longest time. This strategy is called LRU paging [9]. 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. Thus LRU is simple but is not easy to implement without hardware support. It can adapt faster according to as program behavior. LRU like algorithm doesn't suffer from Bélády's Anomaly as FIFO. It gives a 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.

2.1.1.2.3 NRU Page Replacement Algorithm Not Recently Used (NRU), sometimes known as the LRU, page replacement algorithm is an algorithm that favors keeping pages in memory that have been recently used. This algorithm works on the following principle: when a page is referenced, a referenced bit is set for that page, marking it as referenced. Similarly, when a page is modified, a modified bit is set. The setting of the bits is usually done by the hardware, although it is possible to do so on the software level as well. Pages are categorized into four classes in 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 [15].

2.1.1.2.4 LFU Page Replacement Algorithm Least Frequently Used (LFU) [16] 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 the 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 the locality.

2.1.1.2.5 EELRU Page Replacement Algorithm Some algorithms use recency as history information like LRU and MRU. LRU is suitable for the good locality of reference whereas MRU is somewhat suitable for the weak locality of workloads. These two algorithms can be tuned to form adaptive algorithm called Early Eviction Least Recently Used (EELRU) [17], 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 concentrates most of the memory references when it reaches late eviction point.

2.1.1.2.6 LRFU Page Replacement Algorithm The LRU and LFU replacement policies are two extreme replacement policies. The LRU policy gives weight to only one reference for each block, that is, the most recent reference to the block while giving no weight to older ones representing one extreme, and the LFU gives equal weight to all references representing the other extreme. These extremes imply the existence of a spectrum between them. In [18], proposed such a spectrum which is called the Least Recently Frequently Used (LRFU) policy. The LRFU policy associates a value with each block. This value is called the CRF (Combined Recency and Frequency) value and quantifies the likelihood that the block will be referenced in the near future. 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 [19].

2.1.1.2.7 LRU-K Page Replacement Algorithm The LRU policy takes into account the recency information while evicting pages, without considering the frequency. To consider the frequency information, LRU-K [20] was proposed which evicts pages with the largest backward K-distance. Backward K-distance $b_t(p, K)$ can be defined as the distance backward to the K^{th} most recent reference to page p where reference string is known up to time t (r_1, r_2, ..., r_t). The value of parameter K can be taken as 1, 2 or 3. If K=1, it works as simple LRU algorithm. Highly increasing the 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.

2.1.1.2.8 2Q Page Replacement Algorithm Two Queue (2Q) [21] is a good buffering algorithm (giving a 5-10% improvement in hit rate over LRU for a wide variety of applications and buffer sizes and never hurting), having constant time overhead, and requiring little or no tuning. It works well for the same intuitive reason that LRU/B works well: it bases buffer priority on sustained popularity rather than on a single access. 2Q 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. The special buffer contains all missed that is first time referenced block. Ghost buffer contains replaced blocks from the special buffer. Frequently accessed blocks are available in the main buffer. Hence victim blocks are always from the special buffer and main buffer.

2.1.1.2.9 LIRS Page Replacement Algorithm Another important algorithm is LIRS which is already described in section 1.2.2. Its objective is to minimize the deficiencies presented by LRU using an additional criterion named IRR 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.

2.1.1.2.10 ARC Page Replacement Algorithm The ARC [10] is an adaptive page replacement algorithm developed at the IBM Almaden Research Center. The algorithm keeps a track of both frequently used and recently used pages, along with some history data regarding eviction for both. It improves the LRU strategy by splitting the cache directory into two lists, T_1 and T_2, for recently and frequently referenced entries. In turn, each of these is extended with a ghost list (B_1 or B_2) which is attached to the 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:

1. T_1, for recent cache entries;
2. T_2, for frequent entries, referenced at least twice;
3. B_1, ghost entries recently evicted from the T_1 cache but are still tracked; and
4. B_2, similar ghost entries, but evicted from T_2.

T_1 and B_1 together are referred to as L_1, a combined history of recent single references. Similarly, L_2 is the combination of T_2 and B_2.

2.1.1.3 CLOCK Based Page Replacement Algorithm

2.1.1.3.1 CLOCK Page Replacement Algorithm Frank Corbató (who later went on to win the ACM Turing Award) introduced CLOCK [22] as a one-bit approximation to LRU, and its performance characteristics are very similar to those of LRU. So all the performance disadvantages about LRU are also applied to CLOCK. In CLOCK, the memory spaces holding the pages can be regarded as a circular buffer. Here each page is associated with a bit, called reference bit, which is set by hardware whenever the page is accessed. When it is necessary to replace a page to service a page fault, the page pointed to by the hand is checked. If its reference bit is unset, the page is replaced. Otherwise, the algorithm resets its reference bit and keeps moving the hand to the next page [23].
When a page fault occurs, the page being pointed to by the hand is inspected.

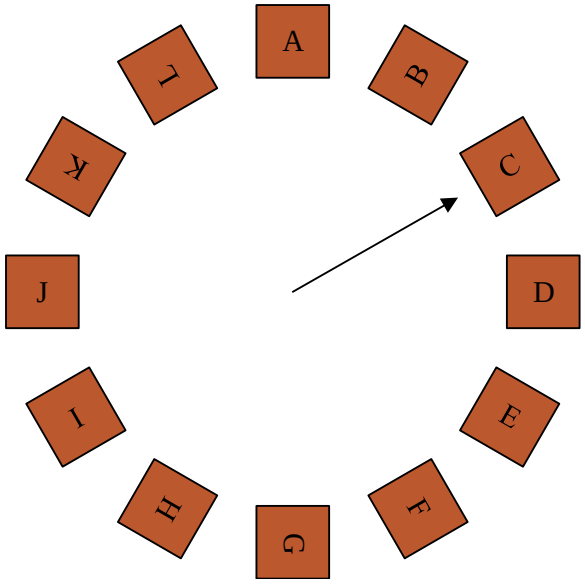


Figure 2.3: CLOCK page replacement algorithm

- R = 0, the page is evicted, the new page is inserted into the clock in its place, and the hand is advanced one position; and
- R = 1, it is cleared and the hand is advanced to the next page. This process is repeated until a page is found with R = 0.

2.1.1.3.2 Clock-Pro In Paper [23], proposed an improved CLOCK replacement policy, called Clock with Pro (CLOCK-Pro). It takes the same principle as that of LIRS-it uses reuse distance (called IRR in LIRS) rather than recency in its replacement decision. When a page is accessed, the reuse distance is the period of time in terms of the number of other distinct pages accessed since its last access. A page is categorized as a cold page if it has a large reuse distance or as a hot page if it has a small reuse distance. Although there is a reuse distance between any two consecutive references to a page, only the most current distance is relevant in the replacement decision.

2.1.1.3.3 CAR Page Replacement Algorithm In paper [24], proposed a simple and elegant new algorithm, namely, Clock with Adaptive Replacement (CAR), that has several advantages over CLOCK:

1. It is scan-resistant;

2. It is self-tuning and it adaptively and dynamically captures the recency and frequency features of a workload;

3. It uses essentially the same primitives as CLOCK, and, hence, is low-complexity and amenable to a high-concurrency implementation; and

4. It outperforms CLOCK across a wide-range of cache sizes and workloads.

The algorithm CAR is inspired by the Adaptive Replacement Cache (ARC) algorithm, and inherits virtually all advantages of ARC including its high performance, but does not serialize cache hits behind a single global lock.

This algorithm uses two clocks T_1 & T_2 and two lists B_1 & B_2. T_1 and T_2 contain cold pages and hot pages i.e. contain pages in the cache, while B_1 & B_2 maintain history information about the recently evicted pages from B_1 & B_2 respectively.

2.1.1.3.4 GCLOCK Page Replacement Algorithm Paper [25] proposed a new algorithm, namely, Generalized CLOCK (GCLOCK). The GCLOCK buffer replacement policy uses a circular buffer and a weight associated with each page brought in buffer to decide on which page to replace. Whenever a page is referenced, the associated count field is set to i. When a page fault occurs, a pointer that circles around this circular list of page frames is observed. If the count field pointed to zero, then the page is removed and the new page is placed in that frame. Otherwise, the count is decremented by 1, the pointer is advanced to the next count field, and the process is repeated. When a new page is placed in the page frame, the count field is set to i if the page is to be referenced (demand fetch) and it is set to j if the page has been pre-paged and is not immediately referenced. This algorithm abbreviated by writing CLOCKP (j, i). The P indicates that this is a pre-paging algorithm (the pre-paging strategy has not been specified).

2.1.2 Buffer Replacement Algorithms for Flash-Based Systems

2.1.2.1 CF-LRU

CF-LRU [11] is the first algorithm designed for flash-based systems. CF-LRU tries to reduce the number of costly write operations and potential erase operations until the degradation of

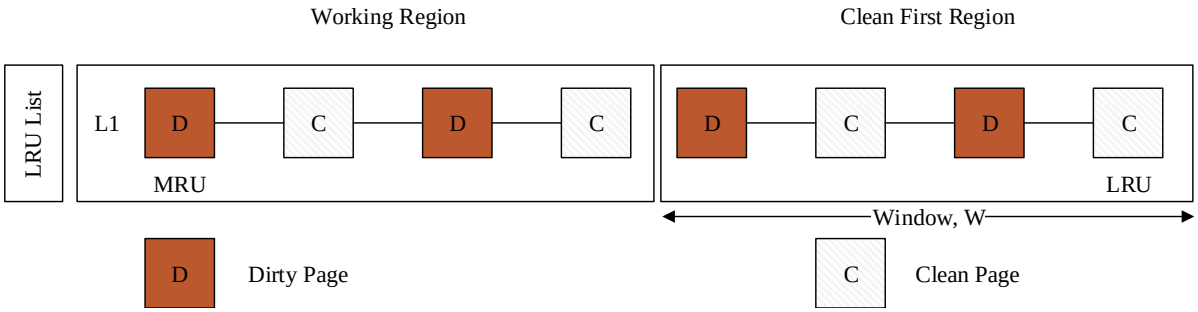


Figure 2.4: CF-LRU page replacement algorithm

cache hit rate does not harm the performance. 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 as shown in Figure 2.4, where B is the buffer size and w is the ratio of the window size to buffer size. When the victim is selected, CF-LRU first evicts least recently used clean pages in W. Hence it reduces the number of write operations because the clean page is not propagated to flash memory. If no clean page is found, then it behaves like LRU policy. CF-LRU 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 a long time and, in turn, results in suboptimal hit ratio. The window size, W, can be tuned statically or dynamically. In this sense, CF-LRU is known as CF-LRU-static or CF-LRU-dynamic. In paper [11], CF-LRU-static and CF-LRU-dynamic have been compared with LRU policy for five different workloads. They found a result that CF-LRU 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) [26] improves the efficiency of the buffer manager by flushing pages in a clustered fashion, based on the observation that flash writes with strong spatial locality can be served by flash disks more efficiently than random writes. It manages the buffer in two regions: the working region W for keeping hot pages that are frequently revisited and the priority region P responsible for optimizing replacement costs by assigning varying priorities to pages. 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

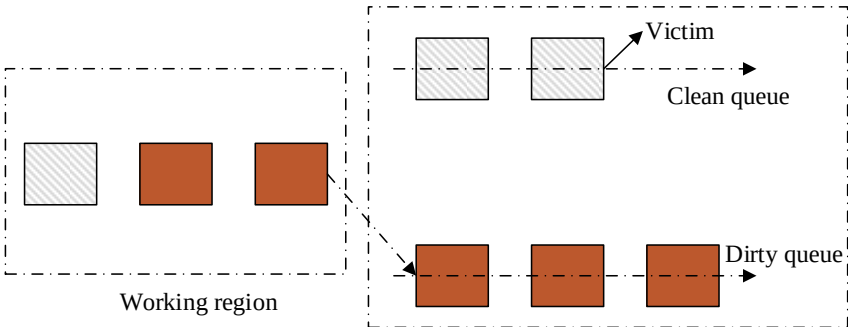


Figure 2.5: CFDC page replacement algorithm

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 [26], CFDC has been compared with LRU and CF-LRU for different four workloads in the database engine. The results show CFDC outperforms both competing policies, with a performance gain between 14% and 41% over CF-LRU, which, in turn, is only slightly better than LRU with a maximum performance gain of 6%.

2.1.2.3 LRU-WSR

LRU-WSR [27] is a flash-aware algorithm based on LRU and Second Chance [22], using only a single list as an auxiliary data structure. The idea is to evict clean and cold-dirty pages and keep the hot-dirty pages in the 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 end of the list and the search continues. Although LRU-WSR considers the hot/cold property of dirty pages, which is not tackled by CF-LRU, it has a 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 [27], LRU-WSR has been compared with LRU, CF-LRU 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 LIRS-WSR

LIRS-WSR which is already explained in section 1.2.2, is an improvement of LIRS so that it can suit the requirements of flash-based systems. It integrates WSR technique to original LIRS algorithm to reduce the number of page writes to flash memory. In paper [6], LIRS-WSR has been compared with LRU, CF-LRU, 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 the case of runtime, LIRS-WSR is 2 times faster than LRU and 1.25 times faster than LIRS algorithm.

2.1.2.5 CCF-LRU

CCF-LRU [2] which is already explained in section 1.2.3. The authors of CCF-LRU [2] further refine the idea of LRU-WSR 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 the cold clean queue and if the cold clean queue is empty then employs same policy as that of LRU-WSR to select a dirty page from the mixed queue. This algorithm focuses on a reference frequency of clean pages and has little consideration on a reference frequency of dirty pages. Besides, the CCF-LRU has no mechanism to control the length of the cold clean queue, which will lead to the frequent eviction of recently read pages in the cold clean queue and lower the hit ratio. In paper [2], CCF-LRU has been compared with LRU, CF-LRU and LRU-WSR with different four workloads. The results show that CCF-LRU performs better than LRU, CF-LRU and LRU-WSR with respect to hit rate, write count and run time.

2.1.2.6 AD-LRU

AD-LRU algorithm [12] is buffer replacement algorithm for flash-based systems which focuses on reducing 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 a victim, it is demoted to head of cold LRU queue. During the eviction procedure, least recently used a clean page from cold LRU queue is selected as a victim.

2.2 Methodology

The main purpose of research is to discover answers to the questions through the applications of scientific procedures. Research is a careful study performed to find out new things in a systematic way. In a scientific method of research at the first problem is formulated then output information is generated from collected input data and output is analyzed and finally, the result is generalized [28]. This dissertation work is truly scientific and flows in the same way. Page replacement algorithm is one of the major strategies to manage memory efficiently. The main exploration of this dissertation focuses on LIRS-WSR and CCF-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, the 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 an incremental approach. The LRU stack automatically maintains recency factor. Information of recently referenced block is available on top of the stack and the oldest in the bottom of the stack. Every time when the block is accessed it is kept on top of the stack. LIRS-WSR and CCF-LRU algorithms are also implemented by using same stack algorithm. The algorithms have been implemented in C++ programming language using Microsoft® Visual Studio[1] on Intel®[2] Core™ i5-4210U CPU @ 1.7 GHz with 4 GiB RAM Microsoft® Windows[3] 10 1607, 64 bit OS.

3.2 LIRS-WSR

The LIRS [1] algorithm can be implemented using 2 lists: LIR stack S which stores all LIR pages as well as HIR pages regardless of the residence status some of them are resident and others are not (actually, only their meta-data are stored in the list) and HIR list Q that stores HIR resident pages. The sum of the size of HIRS and size of LIRS equals to the size of the 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 blocks becomes the victim. The promotion/demotion policy is

[1] https://www.visualstudio.com
[2] http://www.intel.com
[3] https://www.microsoft.com/en-us/windows

shown in the Figures 3.1, 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. 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.

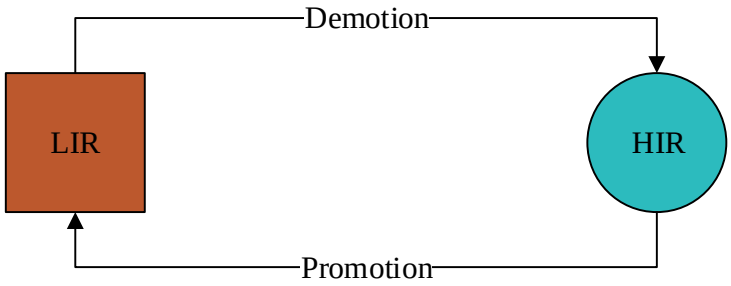


Figure 3.1: General LIR vs. HIR Transition Diagram

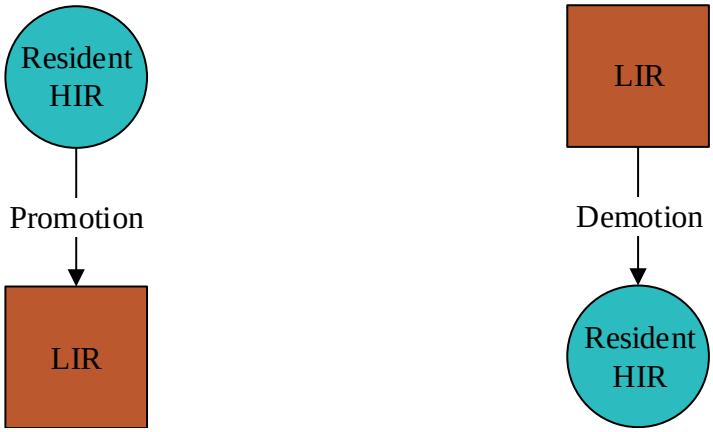


Figure 3.2: Specific LIR vs. Resident HIR Transition Diagram

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 the cache. Hence the 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 a write request for

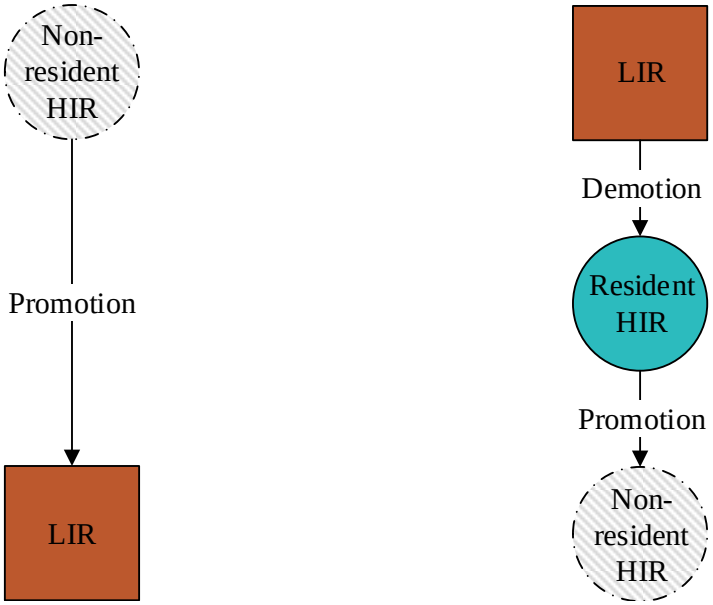


Figure 3.3: LIR vs. Non-Resident HIR Transition Diagram

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 it's 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 Stack Pruning Function

The major function stack pruning is conducted during the status change. The Bold assumption of the algorithm is that bottom of stack S is always LIR block. While changing status, the page in the 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 the 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 a resident. Stack pruning is also conducted if the accessed block P is the bottom LIR because the recent block is always moved to the top of stack S. Stack pruning decreases the size of stack hence the stack doesn't keep track of outdated references.

3.2.2 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 a structure as in Figure 3.4.

```
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
};
```

Figure 3.4: LIRS-WSR Data Structure

3.2.3 Algorithm

Algorithm 3.1 LIRS-WSR

INPUT: Pages

OUTPUT: Miss rate, hit rate etc.

```
Start
Read new page
if page is in Stack S then
    if If page is LIR page then
        Page hit LIR page
        Clear cold flag, i.e. cold=0
        Move the page to the head of Stack, S
        if If the page is at bottom of S then
            Prune the Stack, S
        end if
    else if Page is HIR then
        if Page is resident HIR then
            Page hit, HIR resident page
            Move the page to the head of Stack, making it LIR
            Clear cold flag i.e. cold=0
            Remove the page from HIR list Q
            while Stack bottom is dirty and not cold do
                Move bottom page of Stack, S to the head of Stack, S
                Set cold=1 for this page
                Prune the Stack
            end while
            Move stack bottom to head of the HIR list, making it HIR
            Prune the Stack
        end if
    else if Page is non-resident HIR then
        Page miss
        Remove Tail of HIR Q
        Move it to the head of Stack S, making it LIR
        Clear cold flag of the page
        while Stack bottom is dirty and not cold do
            Set the cold flag of bottom page, i.e. Cold=1
            Move the page to the head of Stack S
            Prune the Stack
        end while
        Move the bottom of Stack to head of the HIR queue Q
        Prune the Stack
    end if
else if Page is in the HIR Q then
    Page hit in HIR Queue
    Move to the head of HIR, Q
    Add to the head Stack S
```

Algorithm 3.1 LIRS-WSR (cont.)

```
else
    Page miss occurs
    if Free memory is available then
        if Free memory is larger than HIR_Limit then
            Add page to the head of Stack S
            Make it LIR page.
            Decrease free memory by one
        else
            if Page is write then
                Add the page to the head of Stack S
                Make it LIR
                while Stack bottom is dirty and not cold do
                    Move the bottom of Stack S to the head of Stack S
                    Set cold=1 for this page
                    Prune the Stack
                end while
                Move Stack bottom to head of the HIR list
                Make this page resident HIR
                Prune the Stack
                Decrease free memory by one
            else if Page is read then
                Add the page to head of queue Q
                Add the page to the head of Stack
                Decrease free memory by one
            end if
        end if
    else if Memory is full then
        Remove tail of HIR queue Q
        if Page is write then
            Add the page to the head of Stack S
            Make it LIR
            while Stack bottom is dirty and not cold do
                Move the bottom of Stack S to the head of Stack S
                Set cold=1 for this page
                Prune the Stack
            end while
            Move Stack bottom to head of the HIR list
            Make this page resident HIR
            Prune the Stack
        else if Page is read then
            Add the page to head of Stack S
            Add the page to the head of queue Q
        end if
    end if
end if
Stop
```

3.2.4 Flowchart

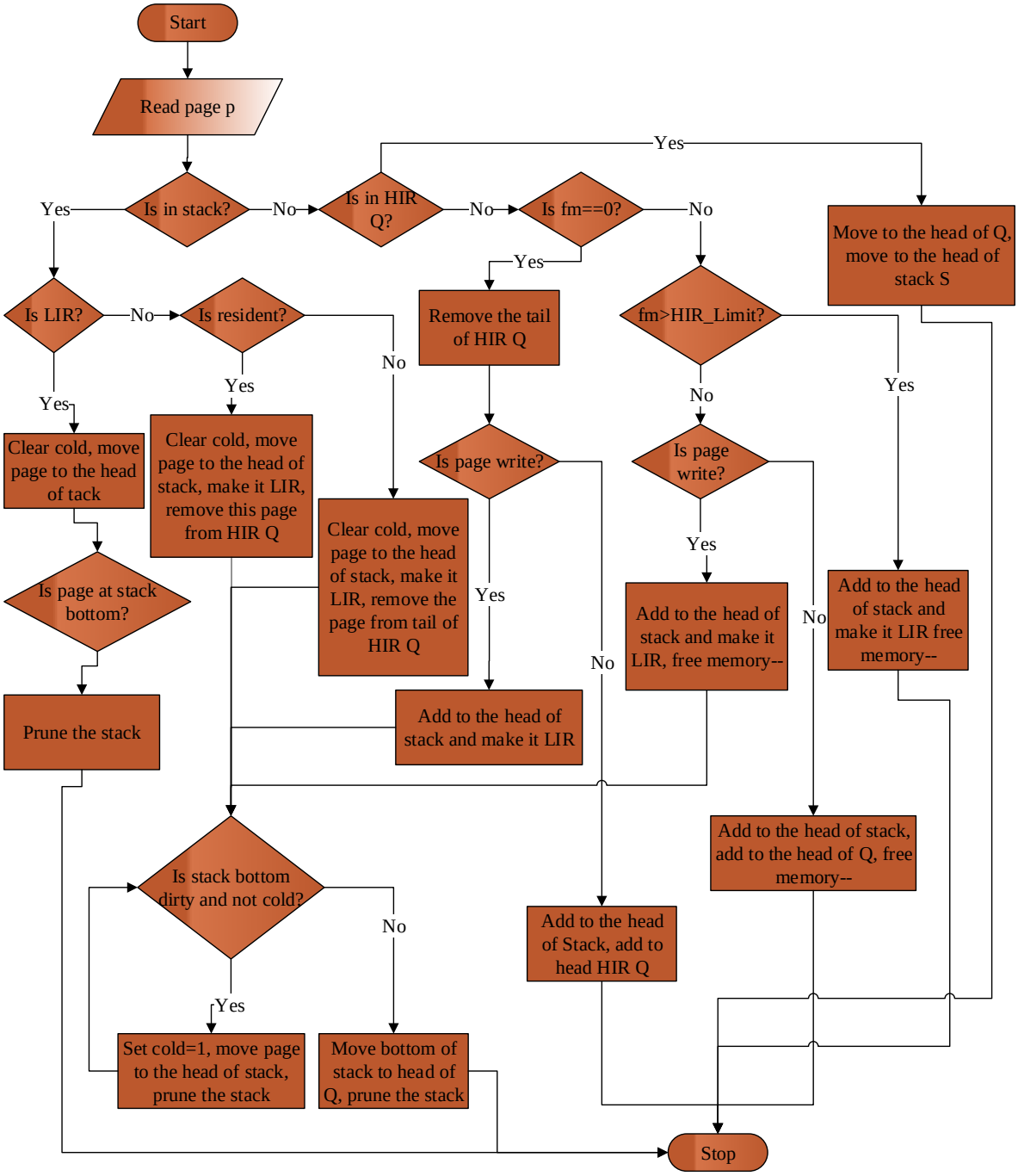


Figure 3.5: Flowchart of LIRS-WSR Algorithm

3.2.5 Tracing of LIRS-WSR

Size of LIRS: 2 Size of HIRS: 1 Cache Size: 2+1=3

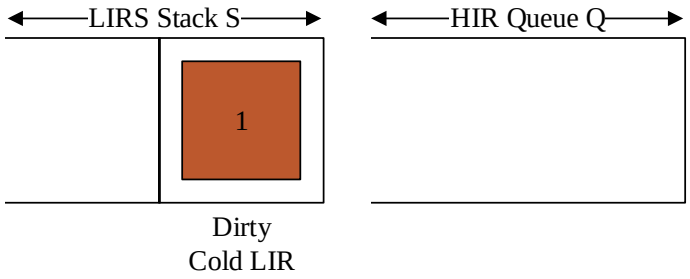
Input References: 1,1 1,2 0,3 0,1 1,4 1,3 0,5 0,2 1,3 where 1 = write, 0 = read

Total Number of References: 9, Number of Distinct References: 5

Other page status: cold/hot, dirty/clean

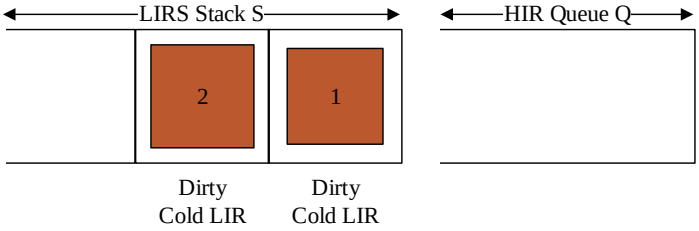
Figure 3.6: Symbols of LIRS-WSR tracing

3.2.5.1 Tracing



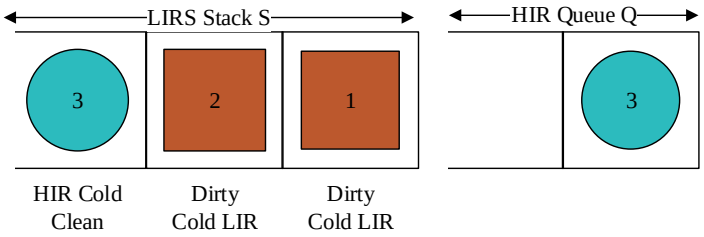
(a) State at Virtual Time 1

Accessing Page 1, 1 page miss, page fault occurred



(b) State at Virtual Time 2

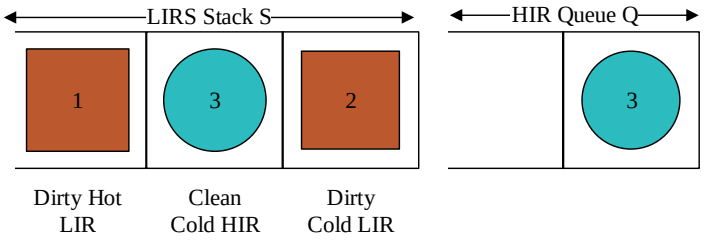
Accessing Page 1, 2 page miss, page fault occurred



(c) State at Virtual Time 3

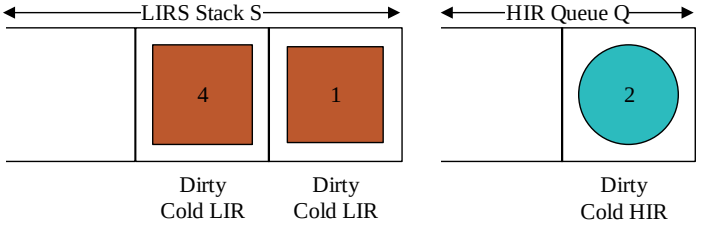
Accessing Page 0, 3 page miss, page fault occurred

Figure 3.7: Tracing of LIRS-WSR



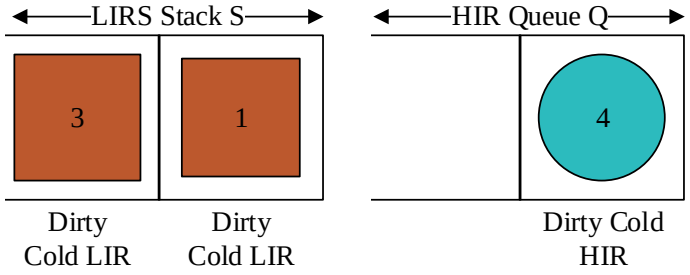
(d) State at Virtual Time 4

Accessing Page 0, 1 page hit occurred



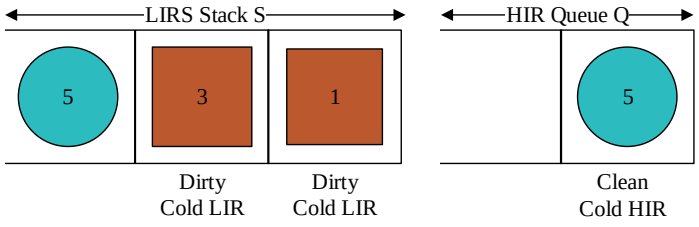
(e) State at Virtual Time 5

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



(f) State at Virtual Time 6

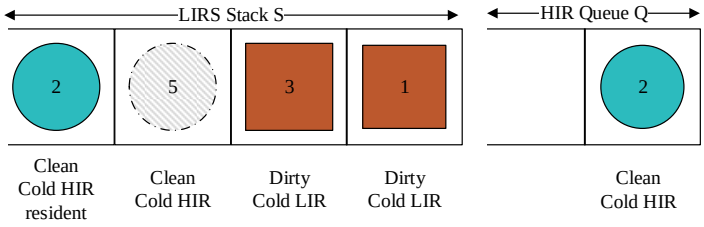
Accessing Page 1, 3 Page miss, memory full, inserted at the head of stack, 1 is given second chance and 4 is demoted to HIR Q, 2 is removed from Q, write count=1, page fault



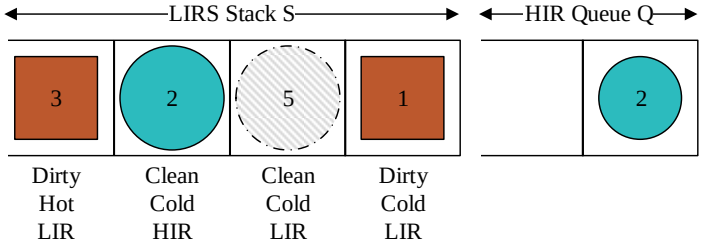
(g) State at Virtual Time 7

Accessing Page 0, 5 page miss, page fault, removed 4 and 5 to head of stack and also Q head, write count=2

Figure 3.7: Tracing of LIRS-WSR (cont.)



(h) State at Virtual Time 8
Accessing Page 0, 2 page miss, page fault occurred



(i) State at Virtual Time 9
Accessing Page 1, 3 page hit

Figure 3.7: Tracing of LIRS-WSR (cont.)
Total page faults = 7 Write counts = 4

3.3 CCF-LRU

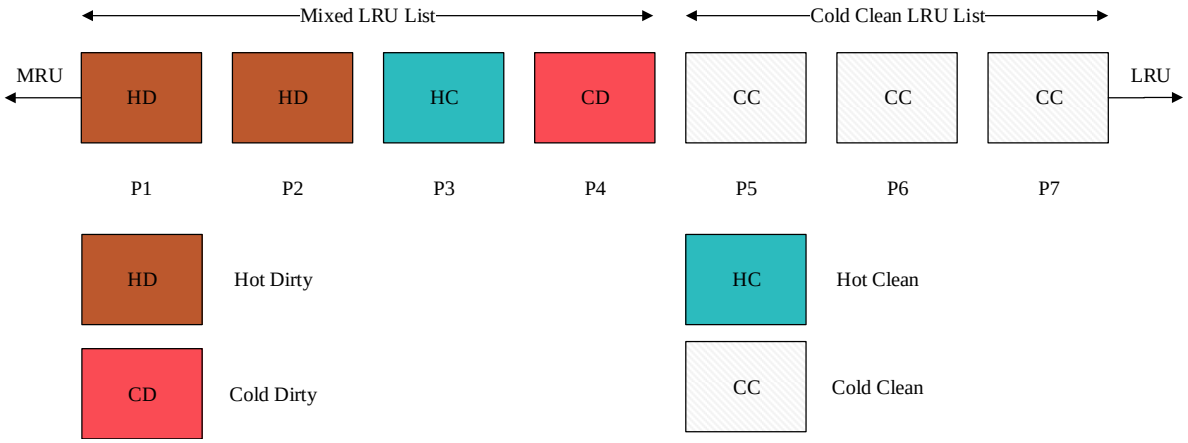


Figure 3.8: An example of CCF-LRU

CCF-LRU [2] enhances the CF-LRU and LRU-WSR algorithms by further differentiating clean pages into cold and hot ones. It uses the cold-detection mechanism of LRU-WSR to classify the buffered pages into four classes, including cold clean pages, hot clean pages, cold dirty pages, and hot dirty pages. As illustrated in Figure 3.8 , the cold clean pages are stored in a cold clean LRU list while other types of pages are held in the mixed LRU list. When a replacement occurs, CCF-LRU always evicts cold clean pages first. If there are no cold clean pages, cold dirty pages are preferentially chosen as the victim and the eviction of hot dirty pages are delayed as long

as possible. The relation of four classified buffer pages can be defined as follows:

$$C_{CC} < C_{CD} < C_{HC} < C_{HD} \tag{3.1}$$

Here, the C_{CC} is the cost of evicting cold clean page, the C_{CD} is the cost of evicting cold dirty page, the C_{HC} is the cost of evicting hot clean page and the C_{HD} is the cost of evicting hot dirty page.

From the expression 3.1, we must reduce the number of evicting hot dirty page and hot clean page as far as possible. If we want to improve the overall I/O performance for flash memory based storage system. However, the LRU-WSR keeps cold clean pages in the buffer until they are evicted, which not only lowers the hit ratio but also makes cold dirty page and hot clean page evicted more quickly. Besides, it always evicts clean pages without considering their access frequencies, which may lower the hit ratio. The main idea of the Cold Clean First (CCF) strategy is described as follows:

1. It uses the cold-detection algorithm to judge whether the page is cold or hot;
2. It evicts cold clean page preferentially as possible, especially for clean page accessed only once recently; and
3. If there is no cold clean page mentioned above, evicting cold dirty page instead of hot clean page.

3.3.1 Flowchart

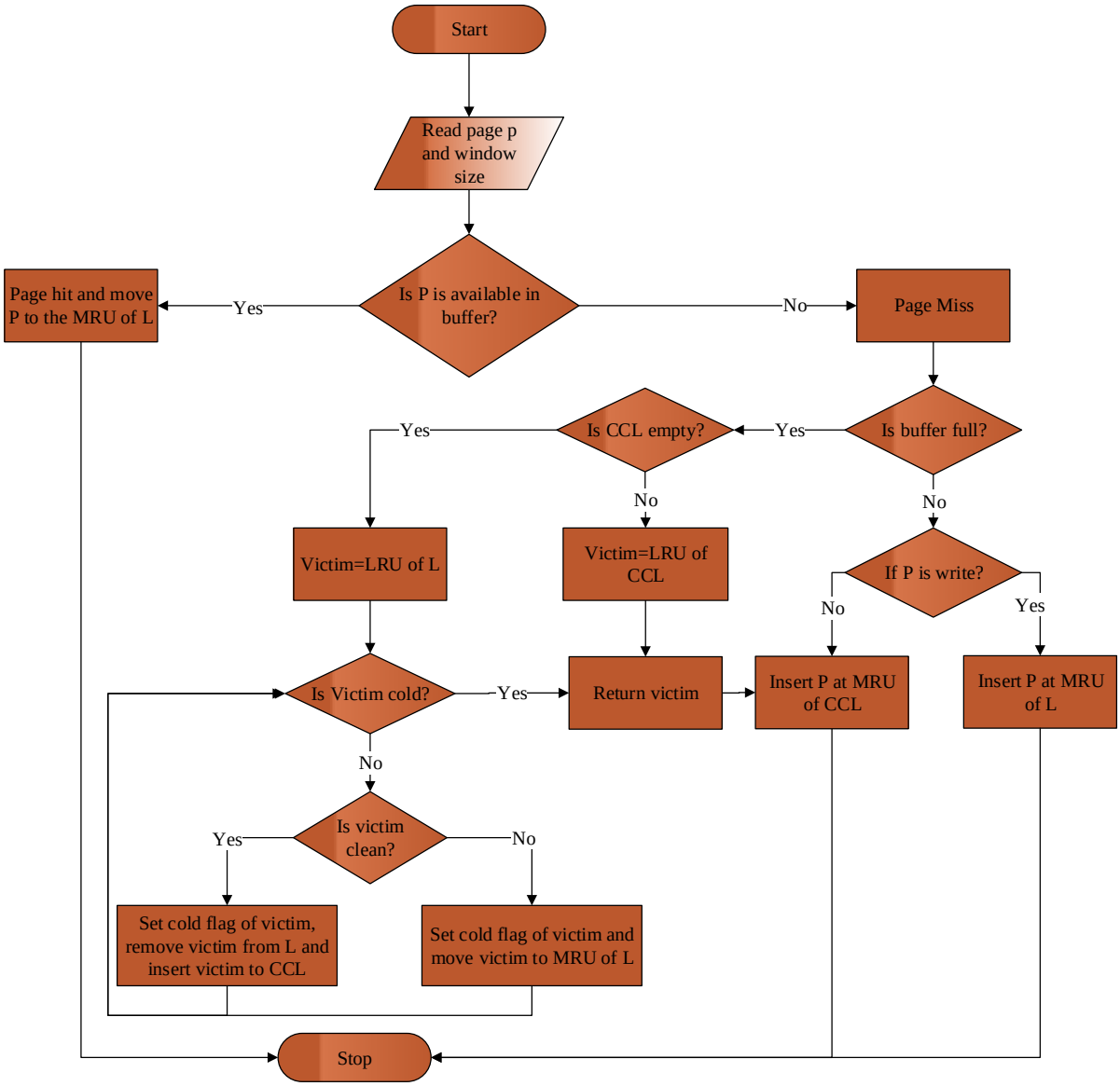


Figure 3.9: Flowchart of CCF-LRU Algorithm

Chapter 4

TEST RESULTS AND 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 [12] 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 Online Transaction Processing (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 patterns that are generated during execution of the process in real OS. There is total 100,000 pages reference 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. The Sample of Workload 1, Workload 2, Workload 3 and Workload 4 are in appendix A.1, appendix A.2, appendix A.3 and appendix A.4 respectively. Tables 4.1, 4.2, 4.3 and 4.4 show the details concerning these workloads.

Table 4.1: Trace for Random Access

Attributes	Value
Total I/O references	100,000
Total Distinct references	43247
Total Distinct references	50%/50%

Table 4.2: Trace for Read-most Access

Attributes	Value
Total I/O references	100,000
Total Distinct references	43212
Total Distinct references	90%/10%

Table 4.3: Trace for Write-most Access

Attributes	Value
Total I/O references	100,000
Total Distinct references	43182
Total Distinct references	10%/90%

Table 4.4: Trace for Zipf

Attributes	Value
Total I/O references	100,000
Total Distinct references	47023
Total Distinct references	20%/80%

4.2 Testing

Each workload is tested in LIRS-WSR and CCF-LRU simulator by varying the cache size from 512 to 18432. In the case of LIRS-WSR algorithms HIR, LIR partition is maintained as 1% and 99% of cache size. For CCF-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

Table 4.5: Test Result of Workload 1

	LIRS-WSR				CCF-LRU			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98952	98.2	1.8	49433	98903	98.1	1.9	49085
1024	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
2048	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
4096	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
6144	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
8192	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
9216	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
10240	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
12288	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
14336	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
16384	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
18432	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms

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

Table 4.6: Test Result of Workload 2

	LIRS-WSR				CCF-LRU			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98952	98.2	1.8	49433	98903	98.1	1.9	49085
1024	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
2048	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
4096	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
6144	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
8192	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
9216	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
10240	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
12288	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
14336	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
16384	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
18432	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms

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

Table 4.7: Test Result of Workload 3

	LIRS-WSR				CCF-LRU			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98952	98.2	1.8	49433	98903	98.1	1.9	49085
1024	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
2048	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
4096	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
6144	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
8192	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
9216	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
10240	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
12288	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
14336	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
16384	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
18432	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms

4.2.4 Test Result of Workload 4 (Zipf Trace)

No. of References = 500000, No. of Distinct References = 47023

No. of Write References = 244790

Table 4.8: Test Result of Workload 4

	LIRS-WSR				CCF-LRU			
	Page Fault	Miss Rate	Hit Rate	Write Count	Page Fault	Miss Rate	Hit Rate	Write Count
512	98952	98.2	1.8	49433	98903	98.1	1.9	49085
1024	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
2048	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
4096	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
6144	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
8192	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
9216	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
10240	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
12288	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
14336	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms
16384	0.03	32	20 μs	28 ms	1.2 sec	25 μs	250 μs	2 ms
18432	0.01	10	25 μs	250 μs	2 ms	25 μs	250 μs	2 ms

4.2.5 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.

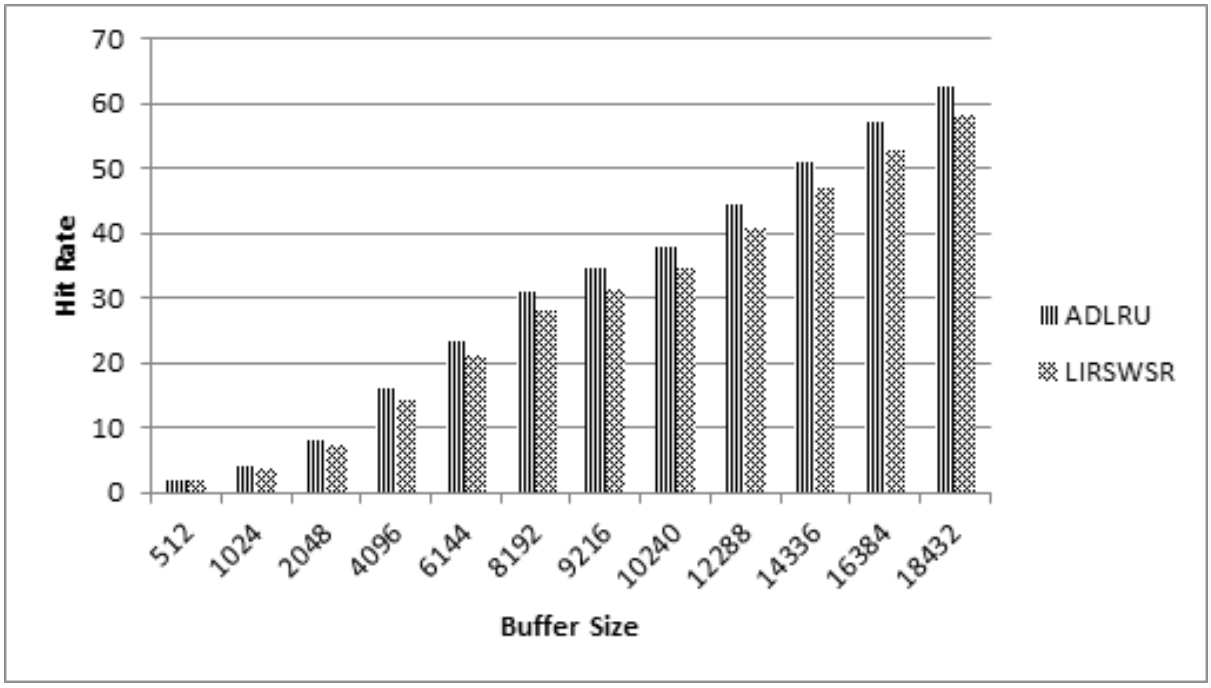


Figure 4.1: Graph of Hit Rate for Workload 1

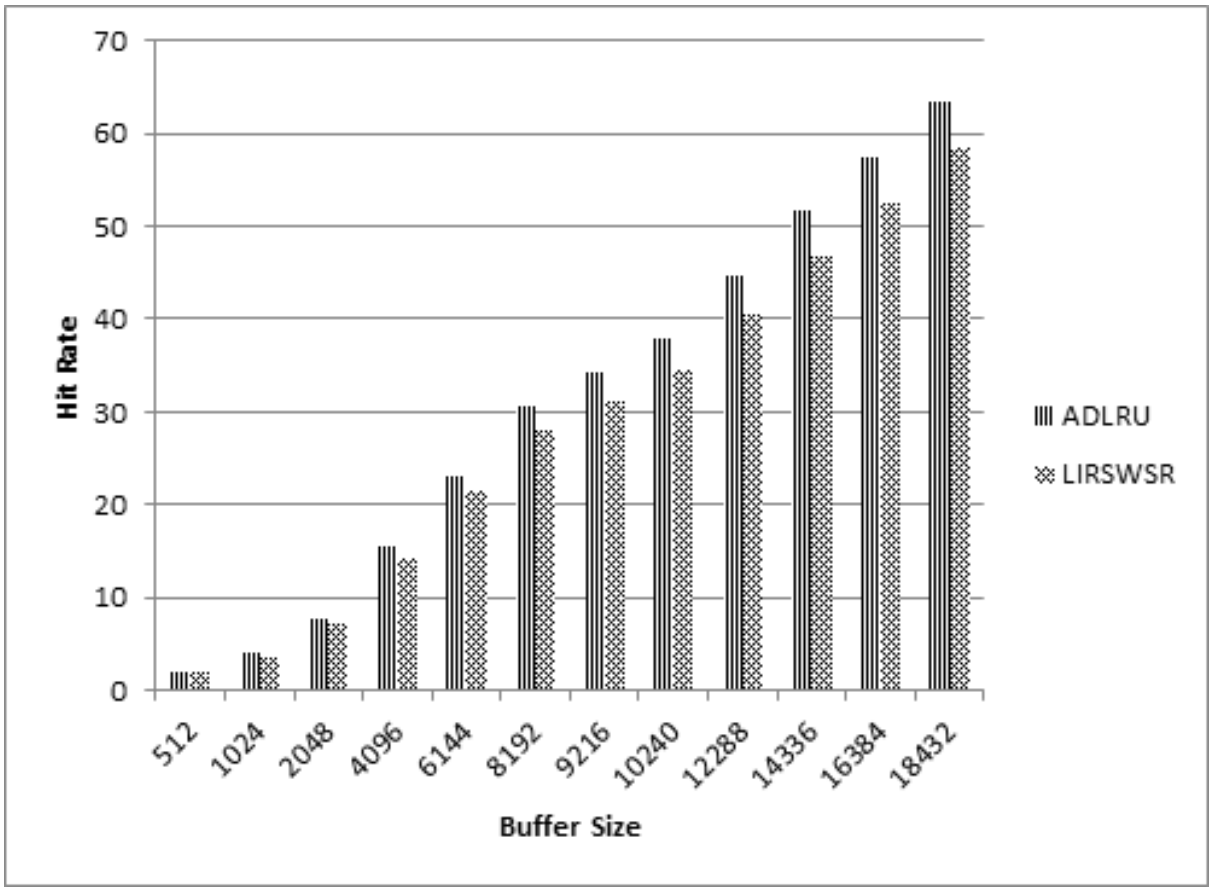


Figure 4.2: Graph of Hit Rate for Workload 2

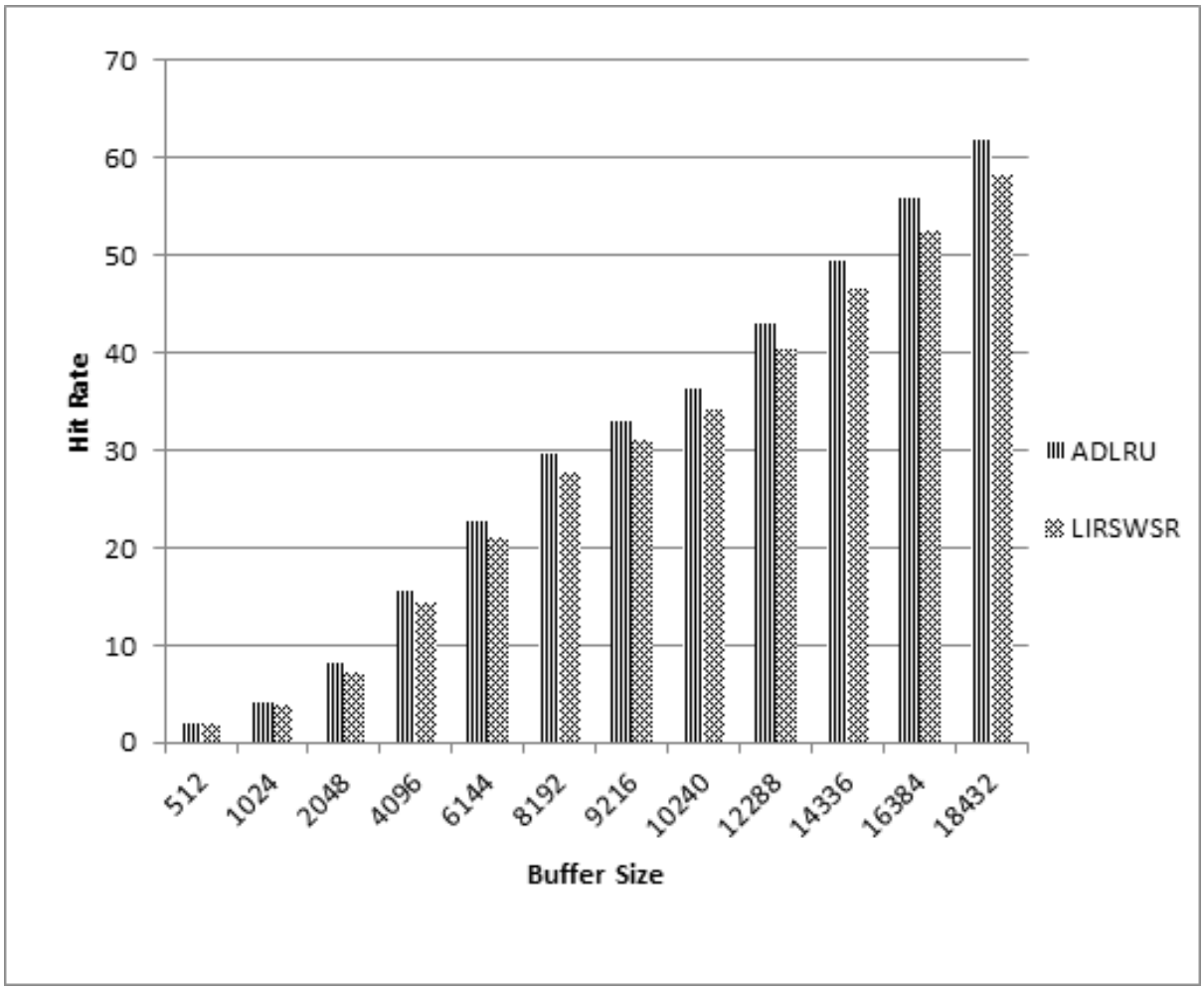


Figure 4.3: Graph of Hit Rate for Workload 3

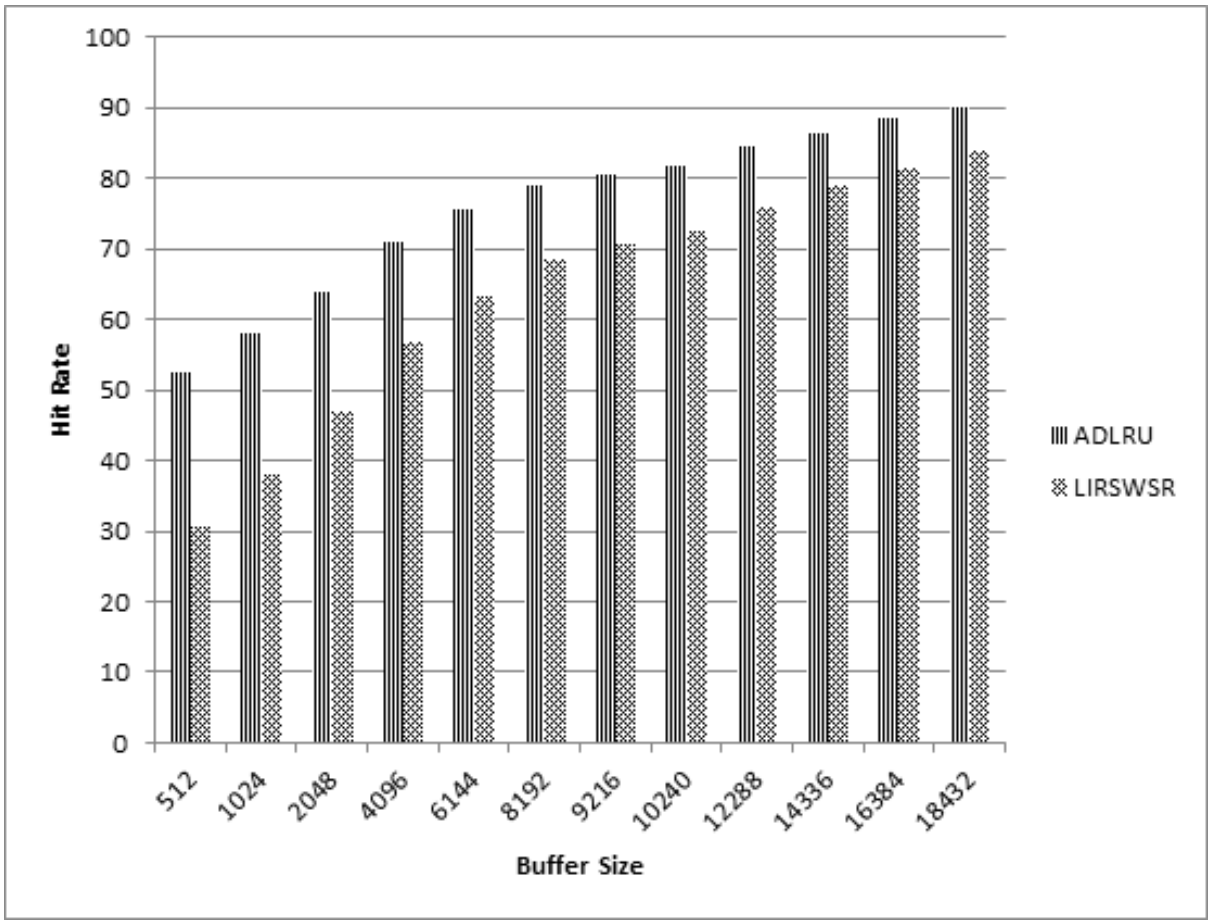


Figure 4.4: Graph of Hit Rate for Workload 4

4.2.6 Hit Rate Analysis

The graph of Figures 4.1, 4.2, 4.3 and 4.4 show that the CCF-LRU algorithm is better than the LIRS-WSR algorithm since CCF-LRU has higher hit rate for different cache sizes. In Figure 4.1 to Figure 4.4 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 a clear distinction between hot and cold pages as reference locality is not high. Despite the nature of page references in these workloads, CCF-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 a 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 a similar way when buffer size increases. The hit rate increases with large buffer size because buffer can hold more pages that increase page hit. The graph of Figure 4.4 shows a significant 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. CCF-LRU adapts more effectively to distinguish hot and cold pages in this workload. So, the hit rate is much higher than that of LIRS-WSR. As the buffer size increases, the difference in the 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, CCF-LRU has up to 3.6% higher hit rate than LIRS-WSR algorithm for workload 3 (write-most trace). As a maximum value, CCF-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 CCF-LRU has more than 6% higher hit rate than LIRS-WSR with this workload. CCF-LRU has higher hit rate than LIRS-WSR in the case of high reference locality workloads.

4.2.7 Write Count Analysis

The graph in Figures 4.5, 4.6, 4.7 and 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 the buffer are also added to the count to get the exact write count. From the above graphs, it is clear that CCF-LRU has smaller write count for

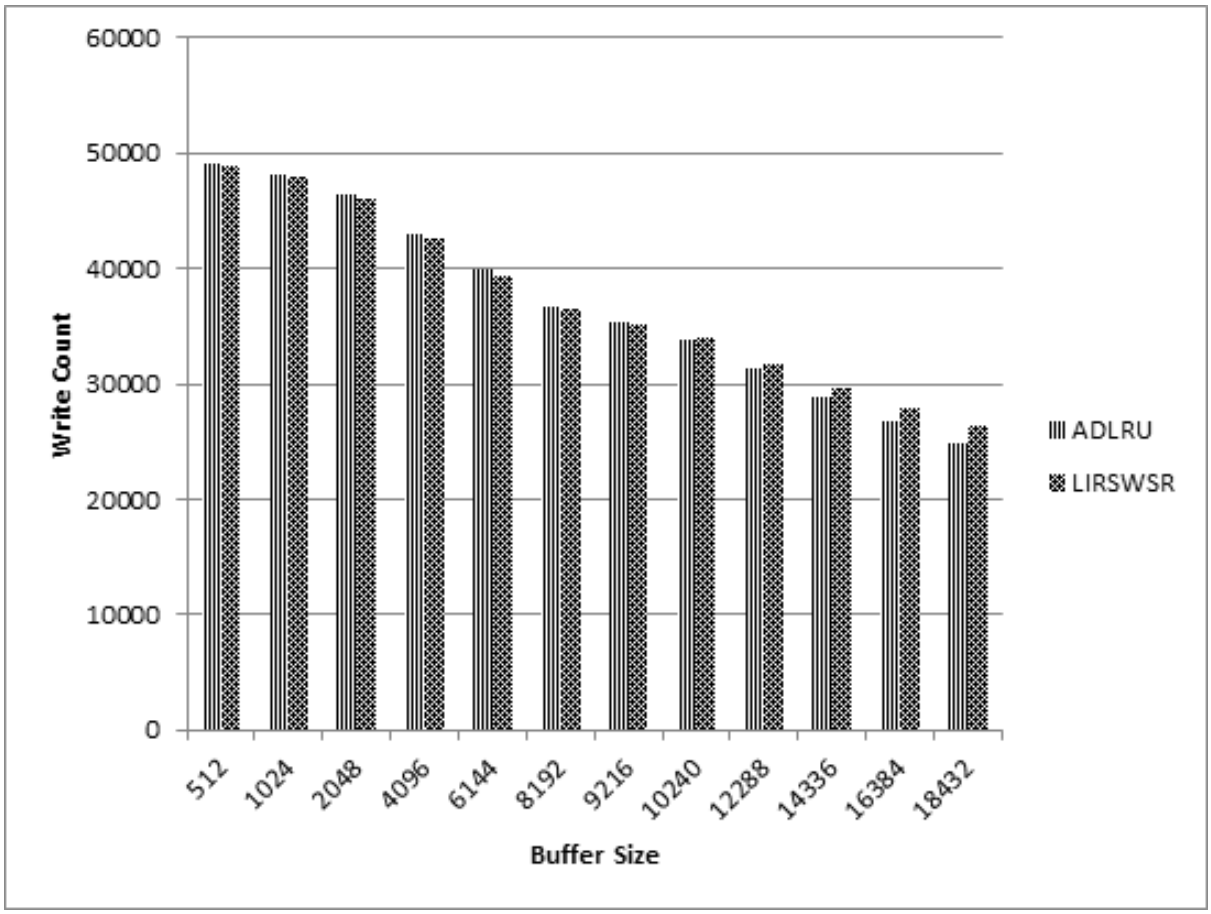


Figure 4.5: Graph of Write Count for Workload 1

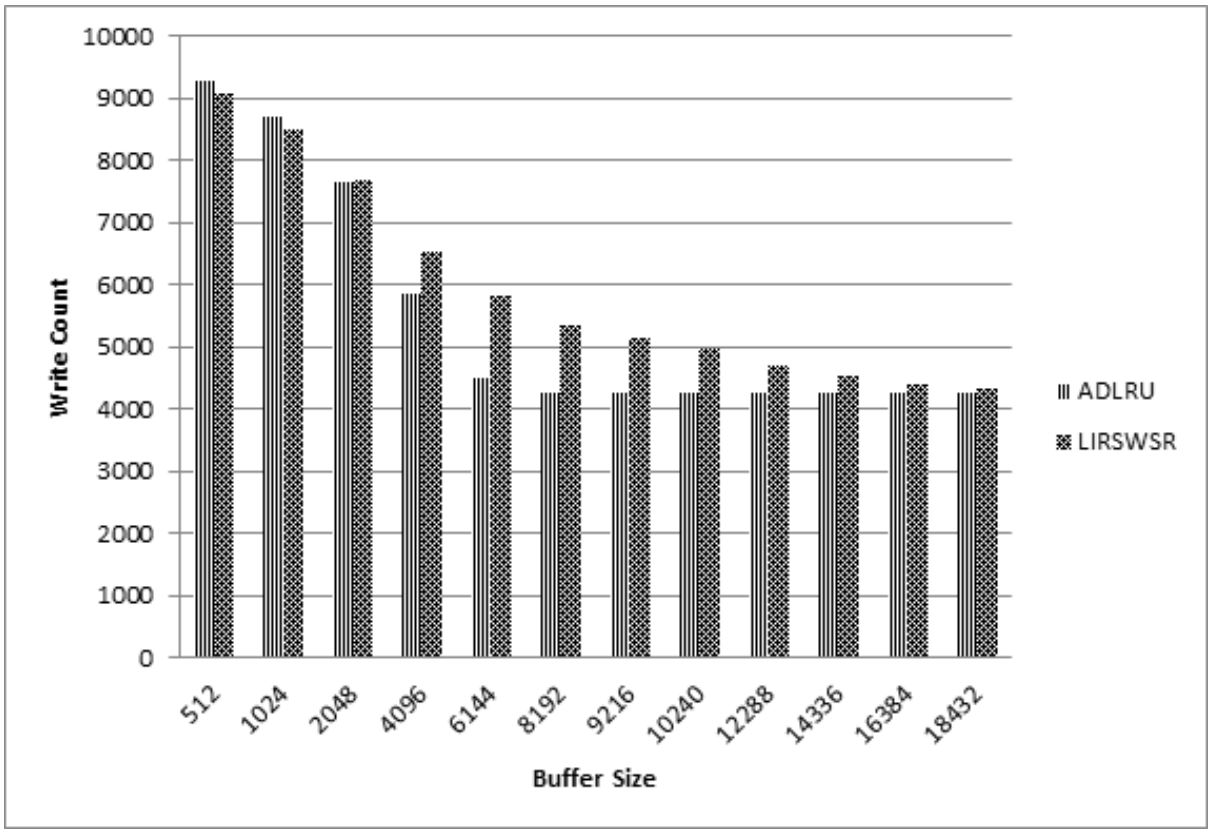


Figure 4.6: Graph of Write Count for Workload 2

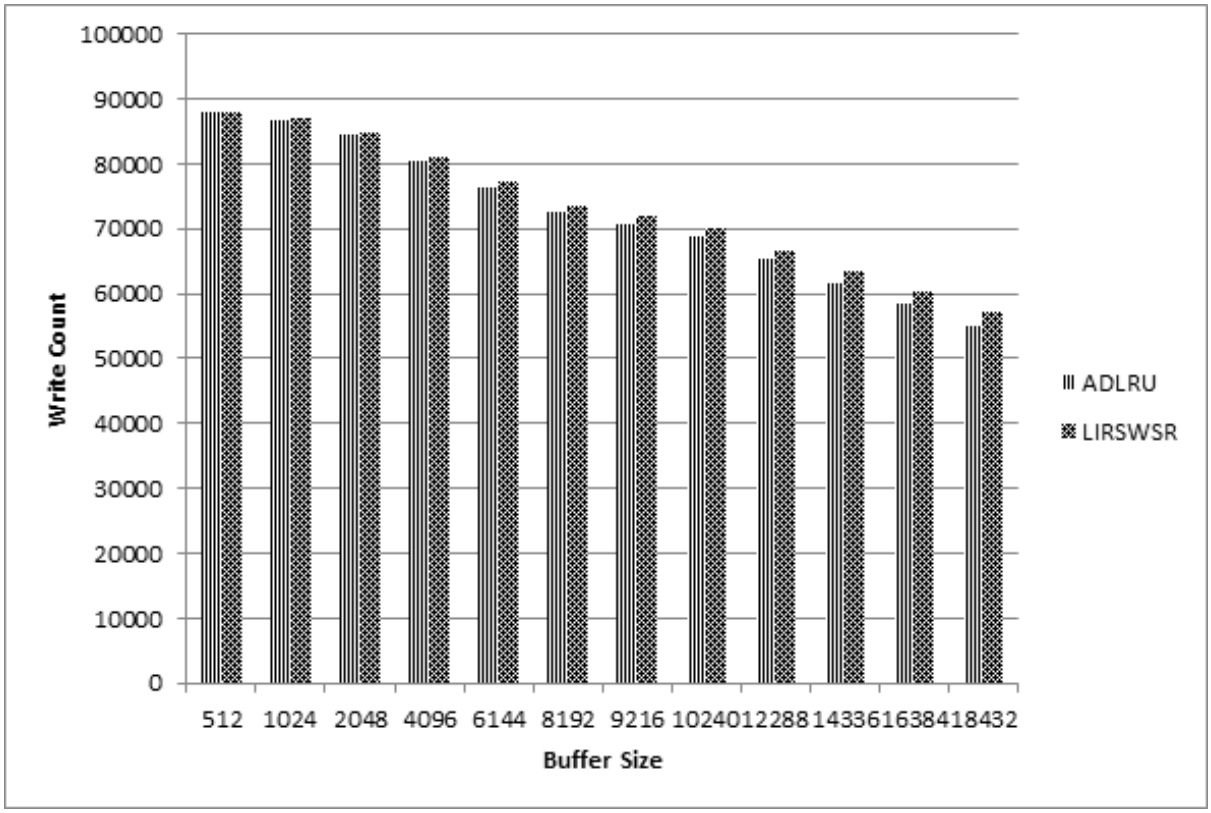


Figure 4.7: Graph of Write Count for Workload 3

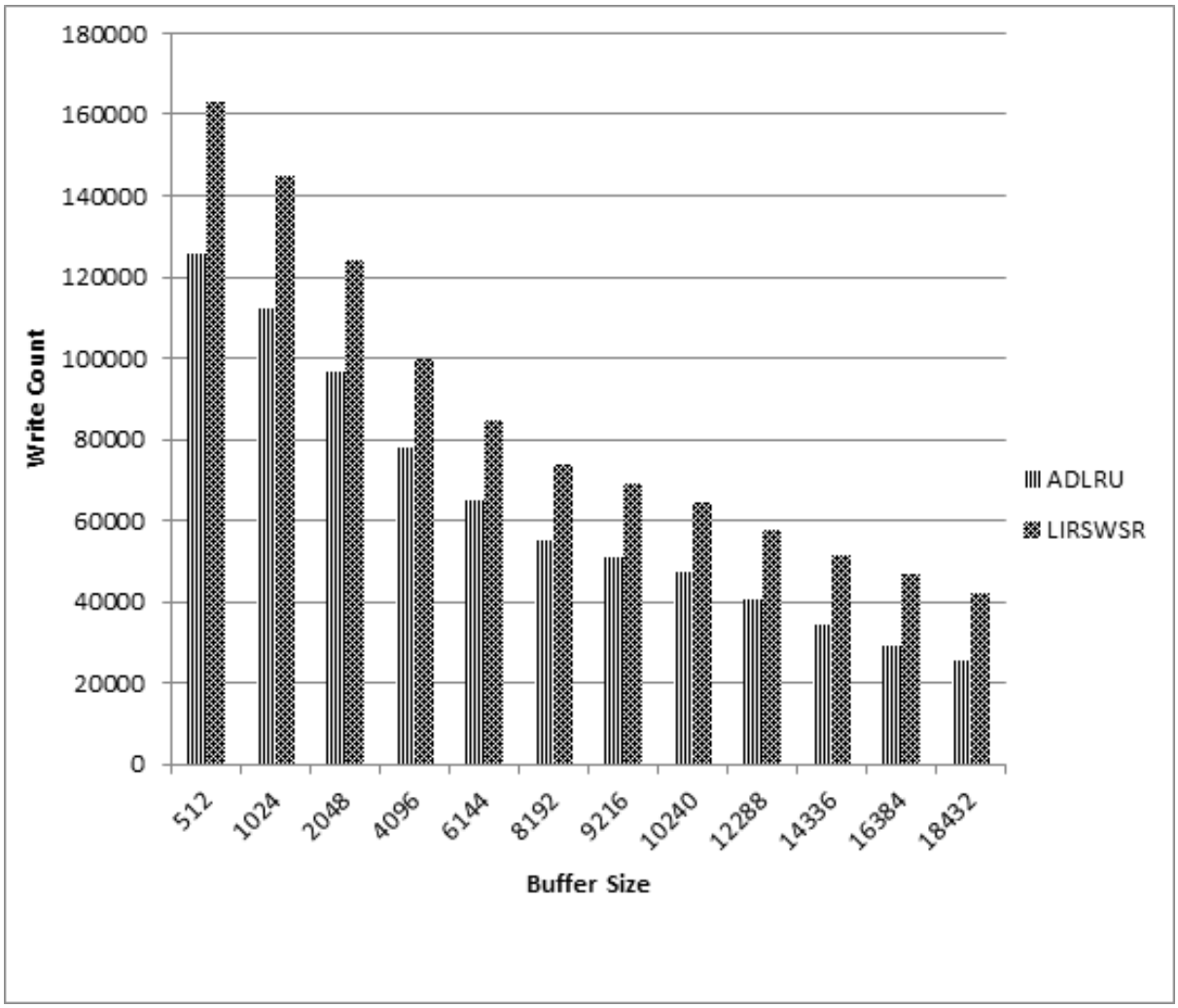


Figure 4.8: Graph of Write Count for Workload 4

all the workloads used in the simulation. Workload 1 has 50% pages as write pages which are uniformly distributed. For smaller buffer size, CCF-LRU has slightly higher write count than LIRS-WSR. This is because there is not a clear idea to adapt hot and cold pages and as the buffer has a smaller capacity, dirty pages are evicted faster in CCF-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, CCF-LRU outperforms LIRS-WSR in write count because of increased page hit in the 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. CCF-LRU cannot differentiate them as hot or cold so evicts faster-increasing write count. But for higher buffer size, CCF-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, CCF-LRU works better for all buffer sizes. There is a large number of write pages in trace LIRS-WSR cannot accommodate all pages in LIRS-Stack and evicts dirty pages as well continuously. CCF-LRU adapts reference pattern so it has better output. Workload 4 zipf trace has 50%/50% read and write references but has high reference locality of 80% page references are references to 20% of pages. CCF-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 CCF-LRU with a comparison to LIRS-WSR algorithm. This is the minimum value by which CCF-LRU outperforms the LIRS-WSR in write count. For zipf trace (workload 4) CCF-LRU decreased write count up to 40% with a comparison to LIRS-WSR algorithm. This is the largest gap between the value of write count of these two algorithms. Thus, CCF-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 the 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 imbalance I/O property of flash memory. LIRS-WSR uses delayed eviction strategy when the dirty page is to be replaced. It uses recency and little bit frequency information in replacement policy. CCF-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 CCF-LRU always outperforms LIRS-WSR for hit rate and write count. Especially, when workload has high reference locality, CCF-LRU has significantly superior performance than LIRS-WSR in terms of both hit rate and write count. This is because of CCF-LRU's good adaptive technique to handle changes in reference patterns.

For uniformly distributed workloads, the difference in hit rates of CCF-LRU and LIRS-WSR is comparatively small. CCF-LRU leads LIRS-WSR in hit rate by a value up to 5%. For high reference locality workloads CCF-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 the buffer size is highly smaller in comparison to the 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, CCF-LRU always outperforms LIRS-WSR in write count as CCF-LRU adapts hot and cold pages better than LIRS-WSR. It seems that in the case of uniformly distributed write-most access type workload, write count is decreased up to 3% by CCF-LRU, but for high reference locality workload CCF-LRU minimizes write count up to 40% with a comparison to LIRS-WSR algorithm. Thus, CCF-LRU minimizes write counts significantly when reference locality is high.

5.2 Limitations and Future Work

In this work, the 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. The 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 a variety of real memory trace with different reference locality.

Thank you!

References

[1] S. Jiang and X. Zhang, "Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 30, pp. 31–42, June 2002.

[2] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "Ccf-lru: A new buffer replacement algorithm for flash memory," *IEEE Trans. on Consum. Electron.*, vol. 55, pp. 1351–1359, Aug. 2009.

[3] M.-L. Chiang, P. C. H. Lee, and R.-C. Chang, "Managing flash memory in personal communication devices," in *Consumer Electronics, 1997. ISCE '97., Proceedings of 1997 IEEE International Symposium on*, pp. 177–182, Dec 1997.

[4] L.-P. Chang and T.-W. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *Trans. Storage*, vol. 1, pp. 381–418, Nov. 2005.

[5] N. Toshiba, "vs. nor flash memory technology overview," tech. rep., Technical Report, 2006.

[6] H. Jung, K. Yoon, H. Shim, S. Park, S. Kang, and J. Cha, "Lirs-wsr: Integration of lirs and writes sequence reordering for flash memory," in *Proceedings of the 2007 International Conference on Computational Science and Its Applications - Volume Part I*, ICCSA'07, (Berlin, Heidelberg), pp. 224–237, Springer-Verlag, 2007.

[7] H.-j. Kim and S.-g. Lee, "A new flash memory management for flash storage system," in *23rd International Computer Software and Applications Conference*, COMPSAC '99, (Washington, DC, USA), pp. 284–289, IEEE Computer Society, 1999.

[8] E. Gal and S. Toledo, "Mapping structures for flash memories: Techniques and open problems," in *Proceedings of the IEEE International Conference on Software - Science, Tech-*

nology & Engineering, SWSTE '05, (Washington, DC, USA), pp. 83–92, IEEE Computer Society, 2005.

[9] A. Silberschatz, P. B. Galvin, G. Gagne, *et al.*, "Memory management strategies," *Operating System Concept, 8th ed. Wiley Student Edition*, pp. 315–417, 2010.

[10] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, (Berkeley, CA, USA), pp. 115–130, USENIX Association, 2003.

[11] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cflru: A replacement algorithm for flash memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, (New York, NY, USA), pp. 234–241, ACM, 2006.

[12] P. Jin, Y. Ou, T. Härder, and Z. Li, "Ad-lru: An efficient buffer replacement algorithm for flash-based databases," *Data Knowl. Eng.*, vol. 72, pp. 83–102, Feb. 2012.

[13] H. Paajanen, "Page replacement in operating system memory management," Master's thesis, Department of Mathematical Information Technology, University of Jyväskylä, Jyväskylä, Finland, October 2007.

[14] B. B. Rawal, "Quantitive Evaluation of Buffer Replacement Algorithms for Flash Based Systems," Master's thesis, Central Department of Computer Science and Information Technology, Tribhuvan University, Kirtipur, Kathmandu, Nepal, 2014.

[15] A. S. Tanenbaum, *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2007.

[16] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, pp. 134–143, May 1999.

[17] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru: Simple and effective adaptive page replacement," *SIGMETRICS Perform. Eval. Rev.*, vol. 27, pp. 122–133, May 1999.

[18] D. Lee, J. Choi, J. hun Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu (least recently/frequently used) replacement policy: A spectrum of block replacement policies," 1996.

[19] G. P. 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, Central Department of Computer Science and Information Technology, Tribhuvan University, Kirtipur, Kathmandu, Nepal, 2007.

[20] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *SIGMOD Rec.*, vol. 22, pp. 297–306, June 1993.

[21] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, (San Francisco, CA, USA), pp. 439–450, Morgan Kaufmann Publishers Inc., 1994.

[22] F. Corbató, "Festschrift: In honor of pm morse, chapter a paging experiment with the multics system, pages 217–228," 1969.

[23] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, (Berkeley, CA, USA), pp. 323–336, USENIX Association, April 2005.

[24] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, (Berkeley, CA, USA), pp. 187–200, USENIX Association, 2004.

[25] A. J. Smith, "Sequentiality and prefetching in database systems," *ACM Trans. Database Syst.*, vol. 3, pp. 223–247, Sept. 1978.

[26] Y. Ou, T. Härder, and P. Jin, "Cfdc: A flash-aware replacement policy for database buffer management," in *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, (New York, NY, USA), pp. 15–20, ACM, 2009.

[27] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, “Lru-wsr: Integration of lru and writes sequence reordering for flash memory,” *IEEE Trans. on Consum. Electron.*, vol. 54, pp. 1215–1223, Aug. 2008.

[28] M. Singh, “Understanding research methodology,” 1991.

Bibliography

- B. Saud, “Sensitivity Analysis of Cache Partition in Clock-Pro Page Replacement and its Comparison with Adaptive Clock-Pro,” Master’s thesis, Central Department of Computer Science and Information Technology, Tribhuvan University Kirtipur, Kathmandu, Nepal, April 2014.

- B. Subedi, “An Evaluation of Page Replacement Algorithm Based on Low Inter-reference Recency Set on Weak Locality Workloads,” Master’s thesis, Central Department of Computer Science and Information Technology, Tribhuvan University Kirtipur, Kathmandu, Nepal, March 2012.

- D. S. Mahara, “A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and AD-LRU for Flash Memory Based Systems,” Master’s thesis, Central Department of Computer Science and Information Technology, Tribhuvan University Kirtipur, Kathmandu, Nepal, March 2014.

Appendix A

Sample Input Traces

A.1 Random Input Trace

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 1,20480 1,22293 0,20389 0,23900 0,18555 0,46427 0,8516 1,49886 0,10679 0,9400 1,24467 0,3709 0,411 0,25540 0,22153 1,3954 0,23179 0,9759 1,33020 0,2711 1,42697 1,34063 1,22716 1,23599 0,25436 1,22036 . . .

A.2 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 0,2856 0,41237 0,19933 0,10902 1,6574 0,14599 0,39656 0,15879
0,9645 0,32760 0,11311 0,14258 0,38921 0,47086 0,24615 0,36799 1,23373 0,30556 0,6997
0,5647 0,22385 0,14890 0,5537 0,11311 0,18829 0,9608 0,44776 0,35106 0,21597 0,18245
0,25921 0,19819 0,41022 0,2924 0,33953 0,9818 0,21029 0,1955 0,26130 0,48683 0,16144
0,42243 0,1071 0,48155 0,17289 0,24699 1,19033 0,18424 0,39192 0,45975 0,949 0,25811
0,35775 0,28294 0,7946 0,2748 0,36907 0,5078 0,27022 1,14669 0,37419 0,12382 0,8955
0,43073 0,4139 0,37292 0,31386 0,15131 0,44501 0,40518 0,6139 0,49892 0,22521 0,9057
1,43638 0,45879 0,30391 0,14690 0,25367 0,10125 0,24894 0,41810 0,49555 0,38776 0,16140
0,49637 0,36102 0,13534 0,4838 1,33623 0,19639 0,33611 1,38969 0,34042 0,32887 0,13925
0,12100 0,10997 0,8528 0,11794 0,23601 0,15213 0,29736 0,47737 0,15336 0,16109 0,10809
0,36945 0,49102 0,40775 1,2132 0,12292 1,28002 0,29787 0,12657 0,27496 0,11586 0,35950
0,19189 0,7309 0,16707 0,45708 0,43469 0,32897 0,21864 0,18648 0,36112 0,29233 0,18148
0,37425 0,21023 0,8947 0,30022 0,43937 0,18352 0,32213 0,26617 0,6472 0,9465 ...

A.3 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 1,5498 0,24528 1,43092 1,11633 1,27217 1,10035 1,5380 1,2269
1,41075 1,7928 1,8105 1,3437 1,22547 1,45582 0,8817 1,38670 1,20172 1,30414 1,47214
1,19627 1,26446 1,40787 1,39687 1,3454 1,37369 1,30931 1,33101 1,18169 1,22790 1,11904
0,47052 1,3672 1,42585 1,9384 1,5275 1,13720 1,19348 1,49136 1,20843 1,19068 1,25883
1,16481 1,27189 0,29307 1,16008 1,45273 0,9839 1,38955 1,48500 1,48560 1,47897 1,37830
1,39217 1,9133 1,18904 0,10499 1,48972 0,42043 1,45152 1,1636 1,12524 1,39143 1,37057
1,9006 1,47238 1,45840 1,5534 1,45368 0,28865 1,6060 1,41228 1,31789 1,48175 1,22391
1,23196 1,34069 0,27033 1,11358 1,21846 1,38558 1,36046 1,4791 1,26938 1,20824 1,4823
1,48716 1,44135 1,28505 1,49252 1,44939 1,36081 1,29232 0,30656 0,47723 1,48222 1,35146
1,878 1,18288 1,8098 1,31077 1,8318 0,21097 0,7152 0,13565 1,46677 1,1957 1,31401 1,39787
1,27588 1,17227 1,31164 1,47753 1,12432 1,2839 1,47863 1,26882 1,6630 1,21134 1,19651
1,27453 1,14355 1,10102 1,29343 0,7942 1,1493 1,28572 0,38982 1,9057 1,15971 1,890 1,41953
1,49738 1,23491 1,31693 1,33812 1,32832 1,9872 0,9447 1,3797 0,32651 1,40169 ...

A.4 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 0,610 0,790 1,24567 0,2514 0,9 1,55 0,39162 1,1007 1,117 1,1203 1,6 1,34659 1,13 0,21797 1,6 0,123 1,2306 0,431 0,763 0,205 0,25 0,5509 0,663 1,920 0,1340 0,36399 1,179 0,17631 0,2 0,96 1,624 0,549 0,1383 1,3269 0,114 1,26 1,11056 0,37000 1,8808 1,39 1,10 0,1006 1,3 0,12883 1,592 1,4360 0,13 0,2387 1,4 0,183 0,1789 0,6300 1,207 1,33724 1,3747 0,23521 0,67 0,2285 0,8867 1,3101 0,33604 1,4 0,99 0,21 1,4084 1,105 1,19222 1,24966 0,18158 0,45453 0,329 1,13 1,625 0,33 0,45 0,2427 1,14065 0,37 1,213 1,14 1,42281 0,7631 0,275 0,204 1,9059 0,39832 0,9638 0,26 0,10692 1,22052 0,29487 1,1265 1,8160 1,10 1,232 0,30695 1,696 1,296 0,7309 0,3 0,45299 0,52 0,1 0,5550 0,16061 0,2100 0,7392 0,6111 0,28824 1,810 1,15492 1,534 0,21235 1,858 0,2 0,4657 1,1 1,4492 0,35641 0,12254 0,6213 1,9804 0,49110 1,31345 1,3 0,137 0,11385 1,2 1,2387 1,9581 0,1 0,9 1,19562 0,2436 1,14530 1,25 0,35087 1,650 0,5731 0,1481 0,437 1,97 0,8280 1,10875 1,422 0,791 1,9719 0,1 0,23043 0,22546 0,292 1,2 1,5320 0,5 1,10 1,83 0,3253 0,1128 1,6808 0,39912 1,5172 0,175 0,26391 0,679 0,3 1,27741 1,24082 1,3 0,692 0,8297 0,80 0,843 0,26940 0,2092 1,37 0,2655 1,38 1,2622 0,3115 1,23585 1,25069 1,1705 0,11 1,427 1,230 0,14530 0,788 0,27979 1,44389 1,2713 1,93 0,1 0,292 1,42296 0,1307 0,484 0,115 0,8 0,389 0,7576 1,36924 0,180 0,3128 0,27230 0,2 0,28 0,475 1,83 0,36531 1,8931 0,9726 1,2378 1,7889 1,1504 1,481 1,31 0,256 1,10779 1,6 0,64 0,17928 1,1031 0,3837 1,5605 0,145 0,3127 1,42369 1,548 1,12 0,5 . . .

Appendix B

Sample Source Codes

This appendix contains the C++ code for the implementation of algorithm using Microsoft® Visual Studio[1] on Intel®[2] Core™ i5-4210U CPU @ 1.7 GHz with 4 GiB RAM Microsoft® Windows[3] 10 1607, 64 bit OS.

B.1 LIRS-WSR

Listing B.1: LIRS-WSR pseudo code

```
1  // program to implement LIRS-WSR algorithm for cache
    ↪ replacement policy for flash memory
2  // programmmed by: Maheshh Kumar Yadav
3  // M.Sc. Roll.no: 18 (2010-12) batch Tribhuvan University,
    ↪ Kathmandu, Nepal
4
5  /* Input File:
6  *              trace file
7  *              parameter file
8  * Output File:  hit rate  it describes the hit rates for
    ↪ each
9  cache size specified in parameter file, pagefaults, write
    ↪ counts
10
11 /* Input File Format:
12 * (1) trace file: Block Number of each reference, which
```

[1] https://www.visualstudio.com
[2] http://www.intel.com
[3] https://www.microsoft.com/en-us/windows

```
13  is the unique number for each accessed block together with
     ↪ integer 0 or 1 to represent access type
14  * (2) parameter file:    one or more cache sizes we want to
     ↪ test
15  */
16
17  #include "stdafx.h"
18  #include "string.h"
19  #include "stdlib.h"
20  #include "stdio.h"
21
22  /* the size percentage of HIR blocks, default value is 1% of
     ↪ cache size */
23  #define HIR_RATE 1
24  #define LOWEST_HG_NUM 2
25
26  #define TRUE 1
27  #define FALSE 0
28
29  struct node
30  {
31    char pn[9];
32    char r;
33    int cold;
34    int isResident;
35    int isHIR_block;
36    struct node * LIRS_next;
37    struct node * LIRS_prev;
38    struct node *HIR_rsd_next;
39    struct node *HIR_rsd_prev;
40    int    recency;
41  };
42
43  unsigned long total_pg_refs;
44  long distinct_refs; /* counter excluding duplicate refs */
45  unsigned long num_pg_flt;
46  unsigned long write_request;
47  unsigned long write_count;
48  unsigned long free_mem_size, mem_size;
49
50  struct node * LIRS_stack_head;
51  struct node * LIRS_stack_tail;
52  struct node * HIR_list_head;
53  struct node * HIR_list_tail;
54
55  unsigned long HIR_block_portion_limit;
56  FILE *trace_fp, *out_1_fp, *para_fp;
57
```

```
58  char history[200000][9];
59
60  void add_LIRS_stack_head(struct node *new_ref_ptr);
61  void move_to_LIRS_stack_head(struct node *new_ref_ptr);
62  int add_to_HIR_List(struct node * new_HIR_ptr);
63  int remove_from_HIR_List(struct node *HIR_block_ptr);
64  void move_to_head_HIR_List(struct node * new_HIR_ptr);
65  int remove_HIR_tail();
66  FILE *openReadFile(char file_name[]);
67  void LIRSWSR();
68  void stack_prune();
69  //start of main
70  int main()
71  {
72    int i, j;
73    char trc_file_name[100];
74    char para_file_name[100];
75    char out_file_name[100];
76
77    printf("Please input file name of parameter:\t");
78    gets_s(para_file_name);
79    strcat_s(para_file_name, ".par");
80    para_fp = openReadFile(para_file_name);
81
82    printf("\nPlease input file name of trace:\t");
83    gets_s(trc_file_name);
84    strcat_s(trc_file_name, ".trace");
85    trace_fp = openReadFile(trc_file_name);
86
87    printf("\nPlease input file name of output:\t");
88    gets_s(out_file_name);
89    strcat_s(out_file_name, ".txt");
90    errno_t err;
91    out_1_fp = fopen(out_file_name, "w");
92
93    fprintf(out_1_fp, "\nOUTPUT OF LIRS-WSR for flash
      ↪ ALGORITHM\n-----------------------------------\n\n\n\n");
94
95    fprintf(out_1_fp, "Memory size | Total #References | MISS
      ↪ Rate | HIT Rate | #Page faults | #Distict references |
      ↪ HIR | Limit | #Write Count | #Write Requests ");
96
97
98    /* Actually read the size of whole reference space and
      ↪ trace length */
99    fscanf(para_fp, "%lu", &mem_size);
100
101   while (!feof(para_fp))
```

```
102     {
103       /* the memory ratio for hirs is 1% */
104       HIR_block_portion_limit = (unsigned long)(HIR_RATE /
      ↪  100.0*mem_size);
105       if (HIR_block_portion_limit < LOWEST_HG_NUM)
106         HIR_block_portion_limit = LOWEST_HG_NUM;
107
108       printf(" \nLhirs (cache size for HIR blocks) = %lu\n",
      ↪  HIR_block_portion_limit);
109       printf("\nmem_size = %lu\n", mem_size);
110
111       distinct_refs = 0;
112       num_pg_flt = 0;
113       write_request = 0;
114       write_count = 0;
115       total_pg_refs = 0;
116       free_mem_size = mem_size;
117
118       LIRS_stack_head = NULL;
119       LIRS_stack_tail = NULL;
120       HIR_list_head = NULL;
121       HIR_list_tail = NULL;
122
123       for (i = 0; i<200000; i++)
124       {
125         for (j = 0; j<9; j++)
126           history[i][j] = '\0';
127
128       }
129       // calling LIRS replacement algorithm
130       LIRSWSR();
131
132       printf("\n\nTotal references = %lu\nTotal distinct
      ↪  references = %lu\nTotal page faults = %lu \nTotal write
      ↪  count = %lu\nTotal Write requests = %lu", total_pg_refs,
      ↪  distinct_refs, num_pg_flt, write_count, write_request);
133       printf("\nTotal blocks refs = %lu \nNumber of misses
      ↪  (page faults) = %lu \nMiss rate = %2.1f \n",
      ↪  total_pg_refs, num_pg_flt, (float)(num_pg_flt -
      ↪  distinct_refs) / (total_pg_refs - distinct_refs) * 100);
134
135       fprintf(out_1_fp,
      ↪  "\n-----------------------------------------------------------------
```

```
136       fprintf(out_1_fp, "%08lu     | %08lu\t         | %3.1f\t
  ↪ | %3.1f       | %d\t       | %lu\t\t     | %03u | %03.2f |
  ↪ %lu\t | %lu", mem_size, total_pg_refs,
  ↪ ((float)num_pg_flt - distinct_refs) / (total_pg_refs -
  ↪ distinct_refs) * 100, 100 - ((float)num_pg_flt -
  ↪ distinct_refs) / (total_pg_refs - distinct_refs) * 100,
  ↪ num_pg_flt, distinct_refs, HIR_block_portion_limit,
  ↪ (float)write_count / write_request * 100, write_count,
  ↪ write_request);
137
138       fscanf(para_fp, "%lu", &mem_size);
139     } //while close
140
141     fclose(out_1_fp);
142     fclose(trace_fp);
143     fclose(para_fp);
144     system("pause");
145     return 0;
146   } // end of the main
147
148   FILE *openReadFile(char file_name[])
149   {
150     FILE *fp;
151     fp = fopen(file_name, "r");
152     if (fp == NULL) {
153       printf("Invalid file %s.\n", file_name);
154       return NULL;
155     }
156     return fp;
157   }
158
159   //LIRS WSR algorithm
160   void LIRSWSR()
161   {
162     char ref_block[9], s[9];
163     char r;
164     int i, j;
165     int flag;
166
167     struct node *temp, *newnode;
168     struct node *hold;
169     struct node *temp1;
170
171     fseek(trace_fp, 0, SEEK_SET);
172     fscanf(trace_fp, "%s", s);
173     while (!feof(trace_fp))
174     {
175       for (i = 0; i<9; i++)
```

```
176         ref_block[i] = '\0';
177       if (s[0] == '1')
178         r = 'W';
179       else
180         r = 'R';
181       for (i = 2, j = 0; s[i] != '\0'; i++, j++)
182         ref_block[j] = s[i];
183       ref_block[j] = '\0';
184       flag = 0;
185       total_pg_refs++;
186       for (i = 1; i <= distinct_refs; i++)
187       {
188         if (strcmp(history[i], ref_block) == 0)
189         {
190           flag = 1;
191           break;
192         }
193       }
194       if (flag == 0)
195       {
196         distinct_refs++;
197         printf("\t %lu", distinct_refs);
198         strcpy(history[distinct_refs], ref_block);
199       }
200       if (r == 'W')
201       {
202         write_request++;
203       }
204       // isin cache
205       // Search for hit in LIRS Stack
206       temp = LIRS_stack_head;
207       while (temp != NULL)
208       {
209         flag = 0;
210         if (strcmp(ref_block, temp->pn) == 0)
211         {
212           if (temp->isHIR_block == 0)
213           {
214             if (r == 'W')
215               temp->r = 'W';
216             move_to_LIRS_stack_head(temp);
217             temp->cold = 0;
218             if (temp->LIRS_next == NULL)
219               stack_prune();
220             // printf("\n hit in LIR....%s",ref_block);
221           }
222           else if (temp->isHIR_block == 1)
223           {
```

```
224             if (temp->isResident == 1)
225             {
226               // printf("\n hit in
    ↪ HIR....%s\t%d",temp->pn,temp->isHIR_block );
227               if (r == 'W')
228                 temp->r = 'W';
229               temp->cold = 0;
230               move_to_LIRS_stack_head(temp);
231               remove_from_HIR_List(temp);
232
233               while (LIRS_stack_tail->r == 'W' &&
    ↪ LIRS_stack_tail->cold == 0)
234               {
235                 move_to_LIRS_stack_head(LIRS_stack_tail);
236                 LIRS_stack_head->cold = 1;
237                 stack_prune();
238               }
239               //clean or cold and dirty page
240               add_to_HIR_List(LIRS_stack_tail);
241               LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
242               LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
243               LIRS_stack_tail->LIRS_next = NULL;
244               stack_prune();
245             }
246             else
247             {
248               num_pg_flt++;
249               if (r == 'W')
250                 temp->r = 'W';
251               else
252                 temp->r = 'R';
253               remove_HIR_tail();
254
255               move_to_LIRS_stack_head(temp);
256               temp->cold = 0;
257
258               while (LIRS_stack_tail->r == 'W' &&
    ↪ LIRS_stack_tail->cold == 0)
259               {
260                 move_to_LIRS_stack_head(LIRS_stack_tail);
261                 LIRS_stack_head->cold = 1;
262                 stack_prune();
263               }
264               add_to_HIR_List(LIRS_stack_tail); // move stack
    ↪ bottom to HIR list head
265               LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
266               LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
267               LIRS_stack_tail->LIRS_next = NULL;
```

```
268             stack_prune();
269           }
270         }
271
272         flag = 1;
273         break;
274       }
275       temp = temp->LIRS_next;
276     }
277     if (flag == 1)
278     {
279       fscanf(trace_fp, "%s", s);
280       continue;
281     }
282     else // Not in stack
283     {
284       // Search for hit in HIR Q
285       temp = HIR_list_head;
286       while (temp != NULL) // hit in cache
287       {
288         flag = 0;
289         if (strcmp(ref_block, temp->pn) == 0)
290         {
291           if (r == 'W')
292             temp->r = 'W';
293           move_to_head_HIR_List(temp);
294           add_LIRS_stack_head(temp);
295           flag = 1;
296           break;
297         }
298
299         temp = temp->HIR_rsd_next;
300       }
301     }
302     if (flag == 1)
303     {
304       fscanf(trace_fp, "%s", s);
305       continue;
306     }
307     else
308     {
309       // printf("\ngenerates block miss */");
310       num_pg_flt++;
311       if (free_mem_size == 0)
312       {
313         remove_HIR_tail();
314
```

```
315         newnode = (struct node *) malloc(sizeof(struct
    ↪ node));
316         newnode->isHIR_block = 1;
317         newnode->isResident = 1;
318         strcpy(newnode->pn, ref_block);
319         newnode->recency = 1;
320         newnode->LIRS_next = NULL;
321         newnode->LIRS_prev = NULL;
322         newnode->HIR_rsd_next = NULL;
323         newnode->HIR_rsd_prev = NULL;
324         if (r == 'W')
325           newnode->r = 'W';
326         else
327           newnode->r = 'R';
328         if (newnode->r == 'W')
329         {
330           add_LIRS_stack_head(newnode);
331           newnode->isHIR_block = 0;
332
333           while (LIRS_stack_tail->r == 'W' &&
    ↪ LIRS_stack_tail->cold == 0)
334           {
335             move_to_LIRS_stack_head(LIRS_stack_tail);
336             LIRS_stack_head->cold = 1;
337             stack_prune();
338           }
339           add_to_HIR_List(LIRS_stack_tail); // move stack
    ↪ bottom to HIR list head
340           LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
341           LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
342           LIRS_stack_tail->LIRS_next = NULL;
343           stack_prune();
344         }
345         else
346         {
347           add_to_HIR_List(newnode);
348           add_LIRS_stack_head(newnode);
349         }
350
351         fscanf(trace_fp, "%s", s);
352         continue;
353       }
354       else if (free_mem_size> HIR_block_portion_limit) //to
    ↪ place page in LIR block
355       {
356         newnode = (struct node *) malloc(sizeof(struct
    ↪ node));
357         newnode->isHIR_block = 0;
```

```
358           newnode->isResident = 0;
359           strcpy(newnode->pn, ref_block);
360           newnode->recency = 1;
361           newnode->LIRS_next = NULL;
362           newnode->LIRS_prev = NULL;
363           newnode->HIR_rsd_next = NULL;
364           newnode->HIR_rsd_prev = NULL;
365           if (r == 'W')
366             newnode->r = 'W';
367           else
368             newnode->r = 'R';
369
370           add_LIRS_stack_head(newnode);
371           free_mem_size--;
372           fscanf(trace_fp, "%s", s);
373           continue;
374         }
375         else    //to place page in HIR block
376         {
377           newnode = (struct node *) malloc(sizeof(struct
    ↪ node));
378           newnode->isHIR_block = 1;
379           newnode->recency = 1;
380           newnode->isResident = 1;
381           strcpy(newnode->pn, ref_block);
382           newnode->LIRS_next = NULL;
383           newnode->LIRS_prev = NULL;
384           newnode->HIR_rsd_next = NULL;
385           newnode->HIR_rsd_prev = NULL;
386           if (r == 'W')
387             newnode->r = 'W';
388           else
389             newnode->r = 'R';
390           if (newnode->r == 'W')
391           {
392             add_LIRS_stack_head(newnode);
393             newnode->isHIR_block = 0;
394
395             while (LIRS_stack_tail->r == 'W' &&
    ↪ LIRS_stack_tail->cold == 0)
396             {
397               move_to_LIRS_stack_head(LIRS_stack_tail);
398               LIRS_stack_head->cold = 1;
399               stack_prune();
400             }
401             add_to_HIR_List(LIRS_stack_tail); // move stack
    ↪ bottom to HIR list head
402             LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
```

```
403             LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
404             LIRS_stack_tail->LIRS_next = NULL;
405             stack_prune();
406             free_mem_size--;
407           }
408           else
409           {
410             add_LIRS_stack_head(newnode);
411             add_to_HIR_List(newnode);
412             free_mem_size--;
413           }
414           fscanf(trace_fp, "%s", s);
415           continue;
416         }
417       } //else close
418     } //while closed
419
420     hold = LIRS_stack_head;
421     while (hold != NULL)
422     {
423       if (hold->r == 'W')
424       {
425         write_count++;
426       }
427       hold = hold->LIRS_next;
428     }
429     hold = HIR_list_head;
430     while (hold != NULL)
431     {
432       if (hold->recency == 0 && hold->r == 'W')
433       {
434         write_count++;
435       }
436       hold = hold->HIR_rsd_next;
437     }
438   }
439   //moves a node to top of stack from stack itself
440   void move_to_LIRS_stack_head(struct node *new_ref_ptr)
441   {
442     if (new_ref_ptr->isHIR_block == 1)
443       new_ref_ptr->isHIR_block = 0;
444     if (new_ref_ptr == LIRS_stack_head)
445     {
446       return;
447     }
448     else
449     {
450
```

```
451       new_ref_ptr->LIRS_prev->LIRS_next =
      ↪   new_ref_ptr->LIRS_next;
452       if (new_ref_ptr->LIRS_next == NULL)
453         LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
454       else
455         new_ref_ptr->LIRS_next->LIRS_prev =
      ↪   new_ref_ptr->LIRS_prev;
456
457       new_ref_ptr->LIRS_next = LIRS_stack_head;
458       LIRS_stack_head->LIRS_prev = new_ref_ptr;
459       new_ref_ptr->LIRS_prev = NULL;
460       LIRS_stack_head = new_ref_ptr;
461       return;
462     }
463   }
464   // stack prunning
465   void stack_prune()
466   {
467     if (LIRS_stack_tail->isHIR_block == 0)
468       return;
469     else
470     {
471       LIRS_stack_tail->recency = 0;
472       LIRS_stack_tail = LIRS_stack_tail->LIRS_prev;
473       LIRS_stack_tail->LIRS_next->LIRS_prev = NULL;
474       LIRS_stack_tail->LIRS_next = NULL;
475
476       stack_prune();
477     }
478   }
479   /* Add a node to head of LIRS stack */
480   void add_LIRS_stack_head(struct node *new_ref_ptr)
481   {
482     struct node *temp;
483     if (new_ref_ptr->recency == 0)
484       new_ref_ptr->recency = 1;
485
486     if (LIRS_stack_head == NULL) {
487       LIRS_stack_head = LIRS_stack_tail = new_ref_ptr;
488     }
489     else
490     {
491       LIRS_stack_head->LIRS_prev = new_ref_ptr;
492       new_ref_ptr->LIRS_next = LIRS_stack_head;
493       LIRS_stack_head = new_ref_ptr;
494     }
495     return;
496   }
```

```
497 // removes a node from HIR list
498 int remove_from_HIR_List(struct node *HIR_block_ptr)
499 {
500   HIR_block_ptr->isResident = 0;
501   if (HIR_list_head == HIR_list_tail)
502   {
503     HIR_list_head = HIR_list_tail = NULL;
504     return TRUE;
505   }
506   else if (HIR_block_ptr->HIR_rsd_prev == NULL) // first
      ↪ node
507   {
508     HIR_list_head = HIR_block_ptr->HIR_rsd_next;
509     HIR_list_head->HIR_rsd_prev = NULL;
510     HIR_block_ptr->HIR_rsd_next = NULL;
511   }
512   else if (HIR_block_ptr->HIR_rsd_next == NULL) //last node
513   {
514     HIR_list_tail = HIR_block_ptr->HIR_rsd_prev;
515     HIR_list_tail->HIR_rsd_next = NULL;
516     HIR_block_ptr->HIR_rsd_prev = NULL;
517   }
518   else // node in between
519   {
520     HIR_block_ptr->HIR_rsd_prev->HIR_rsd_next =
      ↪ HIR_block_ptr->HIR_rsd_next;
521     HIR_block_ptr->HIR_rsd_next->HIR_rsd_prev =
      ↪ HIR_block_ptr->HIR_rsd_prev;
522     HIR_block_ptr->HIR_rsd_next = NULL;
523     HIR_block_ptr->HIR_rsd_prev = NULL;
524   }
525   return TRUE;
526 }
527 //Adding to the head of HIR list
528 int add_to_HIR_List(struct node * new_HIR_ptr)
529 {
530   if (new_HIR_ptr->isHIR_block == 0)
531   {
532     new_HIR_ptr->isHIR_block = 1;     // This stack bottom is
      ↪ demoted to HI
533     new_HIR_ptr->recency = 0;      // removes from stack
534
535   }
536   new_HIR_ptr->isResident = 1;
537   if (HIR_list_head == NULL)
538     HIR_list_tail = HIR_list_head = new_HIR_ptr;
539   else
540   {
```

```
541       HIR_list_head->HIR_rsd_prev = new_HIR_ptr;
542       new_HIR_ptr->HIR_rsd_next = HIR_list_head;
543       new_HIR_ptr->HIR_rsd_prev = NULL;
544       HIR_list_head = new_HIR_ptr;
545     }
546     return TRUE;
547   }
548   // moves a node from HIR list to its head
549   void move_to_head_HIR_List(struct node * new_HIR_ptr)
550   {
551     if (new_HIR_ptr == HIR_list_head)
552       return;
553     else
554     {
555       new_HIR_ptr->HIR_rsd_prev->HIR_rsd_next =
      ↪  new_HIR_ptr->HIR_rsd_next;
556
557       if (new_HIR_ptr->HIR_rsd_next == NULL)
558         HIR_list_tail = HIR_list_tail->HIR_rsd_prev;
559       else
560         new_HIR_ptr->HIR_rsd_next->HIR_rsd_prev =
      ↪  new_HIR_ptr->HIR_rsd_prev;
561
562       new_HIR_ptr->HIR_rsd_next = HIR_list_head;
563       new_HIR_ptr->HIR_rsd_prev = NULL;
564       HIR_list_head->HIR_rsd_prev = new_HIR_ptr;
565       HIR_list_head = new_HIR_ptr;
566     }
567   }
568   // remove HIR Tail node
569   int remove_HIR_tail()
570   {
571     // printf("\n remove tail");
572     HIR_list_tail->isResident = 0;
573     if (HIR_list_tail->r == 'W')
574     {
575       HIR_list_tail->r = 'R';
576       write_count++;
577     }
578     HIR_list_tail = HIR_list_tail->HIR_rsd_prev;
579
580     if (HIR_list_tail == NULL) // single node case
581       HIR_list_head = NULL;
582     else
583     {
584       HIR_list_tail->HIR_rsd_next->HIR_rsd_prev = NULL;
585       HIR_list_tail->HIR_rsd_next = NULL;
586
```

```
587     }
588     return TRUE;
589   }
```