CHAPTER 1

BACKGROUND AND PROBLEM FORMULATION

1 Background

1.1 Flash Memory

Flash memory is a type of electrically erasable programmable read-only memory (EEPROM). Intel and Toshiba first introduce flash memory in 1980s. The behavior of flash memory in terms of read, write, erase are different from other programmable memories like magnetic hard disk and volatile RAM etc. More importantly, the memory cells in a flash device can be erased only limited number of times, between 10,000 and 1,000,000, after which they wear out and become unreliable [1].

It is a type of nonvolatile memory that erases data in units called blocks. A block stored on a flash memory chip must be erased before data can be written, or programmed, to the microchip. Flash memory retains data for an extended period of time whether a flash-equipped device is powered on or off, so it is used to store files and other persistent objects in handheld computers and mobile phones, in digital cameras, in portable music players, in workstations and servers and in numerous other devices. [1, 2].

By adapting clever management, functional life span of flash devices can be dramatically extended. The characteristics of flash memory are significantly different from Magnetic disks; the first difference that can be pointed out is the absence of mechanical components in the flash memory. This leads to no latency in flash memory where the magnetic disks have high latency with its moving parts. When considering the flash memories, there are two types of non-volatile flash memory technologies.

Flash memory usually consists of many blocks and each block contains a fixed set of pages. Read/write operations are performed on page granularity, whereas erase operations use block granularity. Flash memory has characteristics of out-of-place update and asymmetric I/O latencies for read write and erase operations. Write/erase operations are relatively slow compared to read operations.

1.1.1 NAND Vs. NOR Flash

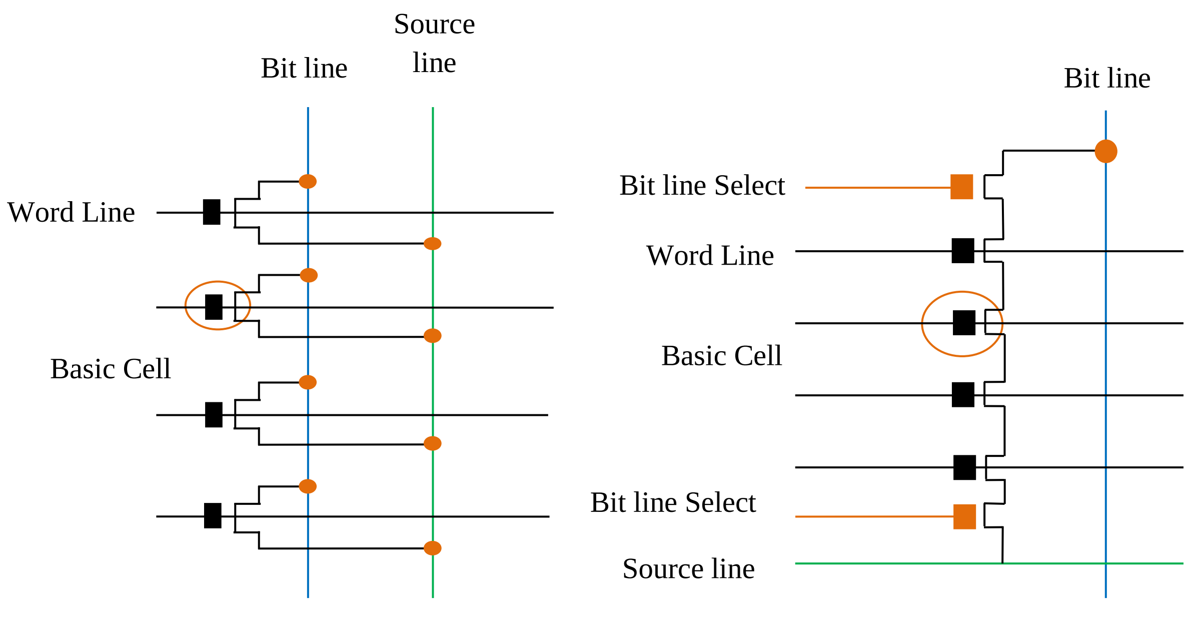
Flash memory could be made of either NAND or NOR architecture. The read latency of NOR is slightly lower than that of NAND and provides full address and data buses, allowing random access to any memory location, but its write and erase latencies are much higher. [3] The NAND architecture offers extremely high cell densities and a high capacity and the I/O interface of NAND flash does not provide a random-access. NOR flash is typically used for code storage and execution, NAND for data storage. Table 1 [4] below shows the different characteristics of two types of flash memory relaying various parameters.

NAND	Access Time (μs/4KB)	Read	284.2
		Write	1833.0
		Erase	499.2
	Energy Consumption (μJ/4KB)	Read	9.4
		Write	59.6
		Erase	16.5
NOR	Access Time (μs/4KB)	Read	53.8
		Write	14054.4
		Erase	15616.0
	Energy Consumption (μJ/4KB)	Read	8.6
		Write	3251.2
		Erase	3609.6

Table 1: The characteristic of flash memory.

NOR-based flash has long erase and write times than NAND. NAND flash memory support page I/O, and its write latency is about 10 times lower than the read latency given in table 1.1 .The internal characteristics of the individual flash memory cells exhibit characteristics similar to those of the corresponding gates. There are three main operations that are used in flash memories; read, write and erase. The read and write operation are performed in units of pages which are usually 512 bytes in size. Erase operations are performed in units of pages, which consist of 32 pages (16KB) each.

Because of the efficient architecture of NAND Flash, its cell size is much smaller than a NOR cell. This, in combination with a simpler production process, enables NAND architecture to offer higher densities with more capacity on a given size. The cost per bit is much lower than NOR. Figure below 1 a) and b) shows the simple hardware architecture of NOR and NAND flash memory respectively [5].



(a) NOR Array **(a) NAND Array**

Figure 1: NOR and NAND flash memory array organization.

(a) In NOR flash memory, cells are connected in parallel. (b) In NAND flash, memory cells are connected in series resulting in increased density [5].

Flash memory uses floating gate transistors. These are arranged in a grid. Rather than a typical transistor that has one gate, flash NAND memory has two gates. Having two gates makes it possible to 'store' a voltage between the two gates so that it doesn't drain away, this is very important and makes any information stored non-volatile. In fact, this 'trapped' voltage which represents information on the chip can stay in a locked state for many years or until we erase the memory. Information stored is erased by draining the voltage away from between the two gates by using the special floating gate feature that is unique to flash memory technology. Advantage of the flash memory comes from the fact that it is an electronic device, unlike the hard disk which is electromechanical and requires disk head and arm movement. This advantage frees the flash memory from the time-consuming seek and rotational delay. Even in high-end applications, flash memory can be arrayed together to offer capacity comparable to that of hard drives at higher speeds.

Usually one page of flash memory consists of 32 sectors each with 512 bytes, and thus its size is usually 16 Kbytes. Such flash memory is called small page NAND flash. Flash memory vendors have started producing large page NAND flash with pages of 64 sectors and sectors of 2,212bytes (thus, the size of a page is 128Kbytes) in order to allow faster write and erase operations for high-end applications. [6, 7] There are only three basic operations in a NAND flash: read a page, write a page, and erase a page.

1.2 Memory Management

Memory management is the functionality of an operating system which handles or manages primary memory. Memory management keeps track of each and every memory location either it is allocated to some process or it is free. It checks how much memory is to be allocated to processes. It decides which process will get memory at what time. It tracks which part of memory is currently in use and which is not. It also allocates memory for a process when required and de-allocates memory when work is temporarily finished [7].

1.3 Paging

Paging is the most popular memory management technique that implements virtual memory. The program generated address is called virtual address and the set of contagious addresses it ranges is called virtual address space. The address in main memory is called physical address. The part of CPU that maps the virtual address into physical address is done by unit is called memory management unit (MMU). Paging techniques split the virtual address space into a units called pages the corresponding units in the main memory is called page frames. The pages and page frames are of same size. When a page is referenced, its virtual address must be mapped to physical address. The page table inside the MMU does this mapping, each entry in the page table holds a flag indicating whether the corresponding page is in main memory or not. If it is in main memory, the page table entry will contain the main memory address at which the page is stored. When a reference is made to a page by the system and if the page table entry for the page indicates that it is not currently in main memory, this situation is called page fault. The operating system picks the little used page frames form main memory to move it to the secondary storage to make the room for new page [7].

A paging algorithm is needed to manage paging which can be accomplished in three steps: fetching, placement and replacement. The fetch procedure decides which page to fetch (extract) from secondary memory to put in main memory, placement procedure determines free page frame to locate the fetched page and finally replacement procedure decides which page to be swapped out when required page have to be brought in. Further, paging algorithm can be demand paging or pre-paging. Demand paging places pages into memory only on their demand. Pre-paging loads the pages before letting processes runs. Demand paging is considered to be better choice because its further uses but pre-paging is not in real use because it requires prediction of page uses which is difficult to predict [8].

1.4 Page Replacement Algorithm

There are lots of optimization has been done on various page replacement algorithm to fulfill the wider gap between processor speed and hard disk speed. As we know that, the data access time form main memory is much faster than from hard disk. Page replacement algorithms has critical role in improving this performance gap. When page fault occur the replacement algorithm must decide which page from the main memory should be removed to make the room for new page in such way that a page that is just flushed to the disk should not be brought in very soon because it adds extra overhead and degrades performance. The best idea is to remove the page that has been unused for longest period.

The main goal of page replacement algorithms is to minimize the no page fault and to increase the no of page hits i.e. most referenced pages are in main memory. Every page fault limits the speed of those accesses because the process that suffers page fault must wait until swap is completed. Page fault rate is one of the criteria to evaluate performance of page replacement algorithm, which is calculated by running it on different memory reference pattern. Reference pattern refers to the list of referenced pages by processor. Page fault rate of algorithm adequately depends on the number of page frames available. Therefore, to determine the number of page fault the number of page frames that are used should also be known. [8] Different types of page replacement algorithms and their working behavior is described more detail in subsequent chapter.

1.5 Replacement Strategy for Flash Memory Based Systems

Flash memory has different characteristics than hard disk as discussed in the previous section. First, it has asymmetric I/O latencies among read, write and erase operations: a write/ erase operations are much slower than read operations. Second, it does not support in place update i.e. writing to same page cannot be done before the page is erased. Third, there is limited no of erasure in each cell of the flash memory. Therefore, the replacement algorithms which have been designed for the disk based system are not sufficient for flash based system. All the traditional algorithms like LRU, LFU, LRFU, 2Q, LIRS etc. are only focused on hit ratio improvement only but they do not have any concern about these different characteristics of flash-based storage. Therefore, it is necessary to revised them to adopt themselves for flash-based system. One of the basic approach they should fallow is to make the delay of eviction of dirty pages form memory or cache in order to reduce the erase operations which improves the overall performance of system. Thus, replacement algorithm for the flash-based system should consider both hit ratio and write count as measure criteria [6].

1.6 Performance Metrics

Performance metrics are the criteria for measuring the performance of any system or algorithm. In the case of page replacement algorithm page fault, hit rate, hit ratio, miss rate and miss ratio are the key terms for measuring the performance. Higher hit rate of the algorithm exhibits higher performance. Furthermore, in context of flash memory, write count is the major criteria of performance measure. In short, higher hit rate and lesser number of write counts is measure for better algorithm. Following performance metrics are used to evaluate the existing and modified algorithm in this dissertation work [6].

i) Page Fault Counts

When the requested page by process is not found in memory it is considered as page fault. Page fault count can be measured by counting total number of page faults occurred between the some intervals of references.

ii) Hit Rate and Hit Ratio

This is the rate of hitting the page in main memory or finding page in memory that is requested by process, out of total referenced pages Hit rate is calculated by using following formula.

$HR = 100 - MR$ ……...........................1.1

Where, HR is the hit rate and MR is the miss rate.

Hit ratio is the fraction of total number of hits by total references.

iii) Miss Rate and Miss Ratio

Miss rate (MR) is calculated by using formula:

$MR = 100 \times ((PF - NDR) / (REF - NDR))$ ……...........................1.2

Where, PF is the number of page faults, NDR is the number of distinct pages referenced and REF is the total number of referenced pages.

Miss ratio is the fraction of total number of misses by total references.

iv) Write Counts

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

1.7 Program Behavior

There are several factors, which influence the performance of page replacement algorithm. The performance of page replacement algorithm relies on the pattern of pages that are referenced. Behavior of program depends upon the access pattern it references in memory which is further depends upon working set and locality of reference [7, 8, 9].

1.7.1 Working set

Working set is the smallest collection of frequently accessed pages, which are needed in main memory for execution. If the entire working set is in main memory then the system work without causing page fault until is completed or moves into another stage. Intuitively, it holds only relevant pages. If the working set is unable to fit in main memory, then there will be high number of page faults and it suffers from thrashing.

1.7.2 Locality of Reference

Locality of references is one of the properties of page reference pattern, which is used by many algorithms to predict about the future references. In computer science, locality of reference, also known as the principle of locality, is a phenomenon describing the same value, or related storage location being frequently accessed. There are two basic types of reference locality: temporal and spatial locality.

Temporal locality is based on the time, which refers that if at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again in the near future. Looping, subroutines, stacks, variable used for counting & totaling etc. supports this assumption [10].

Spatial locality is based on the space, and it refers that if a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced in the near future. In this case, it is common to attempt to guess the size and shape of the area around the current reference for which it is worthwhile to prepare faster access. Array traversal, sequential code execution, related variable declaration nearby in source code supports this assumption. Some memory pattern fallow the strong locality and some fallow weekly. Some algorithms are better works in strong locality and some in week locality. The algorithm that exhibits weak locality impacts the cache performance optimization. So, the locality of reference is most important principle in cache optimization [11].

1.7.3 Typical Memory Reference Pattern

The changes in working set generates memory access pattern. The performance of page replacement algorithm depends on the pattern of the pages referenced. Each page replacement algorithm is evaluated by executing it on particular string of reference string. There are several identified page reference patterns which are discussed in the following sections.

i) Cyclic Pattern

When the set of reference pages are repeated in fixed time in same order such types reference pattern is called cyclic pattern. For example if 10,12,14,16 are reference page then their cyclic pattern will likely to be 10,12,14,16,10,12,14,16 and so on.

ii) Probabilistic Pattern

Each page in reference pattern associated stationary reference and probability. These pages are accessed based on their associated reference probability. For example if 1 and 2 are frequently accessed pages then the probabilistic pattern is likely to be 1,2,3,4,5,1,6,2,8,9,10,1.

iii) Temporally Clustered Pattern

A temporally clustered pattern has property that page referenced recently likely to be referenced sooner in the future. For example temporally clustered pattern can be viewed as 1,2,1,3,2,4,3,1,2,5,6.

iv) Mixed Pattern

Mixed pattern is the combination of all identified memory reference pattern. This means it is the mixed form of cyclic, probabilistic and temporal clustered patterns. For example, if 1,2,3,4, 1,2,3,4 is cyclic pattern 1,2,4,5,1,6,2,10,1 is probabilistic pattern and 1,2,1,3,2,4,3,1,2 is temporally clustered pattern then the mixed pattern may whatever be like 1,2,3,4,1,2,3,4,1,2,4,1,2,4,5,1,6,2,10,1 by containing any of these reference strings.

1.8 Problem Formulation

Since, in most operating systems which are customized for disk-based system, the replacement algorithm concerns only the number of memory hits. However, flash memory has asymmetric read and write/erase cost, most importantly the write cost. In the subsequent year, latest research have been done mainly focused on write count minimization which is most useful for the flash memory. Although, there is no any algorithm in literature review which balance the both of these parameter to improve the overall performance of the algorithm and system. Algorithm that are designed for flash-based system should also focus on write count as well to improve the overall performance of system. **So, the main aim of this dissertation work is to modify CFLRU algorithm, which is designed for flash memory based system and then evaluate it using aforementioned performance metrics**. The modification is done mainly in two component, i.e. giving one chance to dirty hot and clean hot both. More detail discussion will be done in following sections.

1.8.1 Basic CFLRU Algorithm

CFLRU (Clean First Least Recently Used) [12], which is modified from the LRU algorithm. CFLRU divides the LRU list into two regions to find a minimal cost point, as shown in Figure 1 below. The working region consists of recently used pages and most of cache hits are generated in this region. The clean-first region consists of pages which are candidates for eviction. CFLRU selects a clean page to evict in the clean-first region first to save flash write cost. If there is no clean page in this region, a dirty page at the end of the LRU list is evicted.

For example, under the LRU replacement algorithm, the last page in the LRU list is always evicted first. Thus, the priority for being a victim page is in the order of P8, P7, P6, and P5, in Figure 1. However, under the CFLRU replacement algorithm, it is in the order of P7, P5, P8, and P6 as shown in figure below:

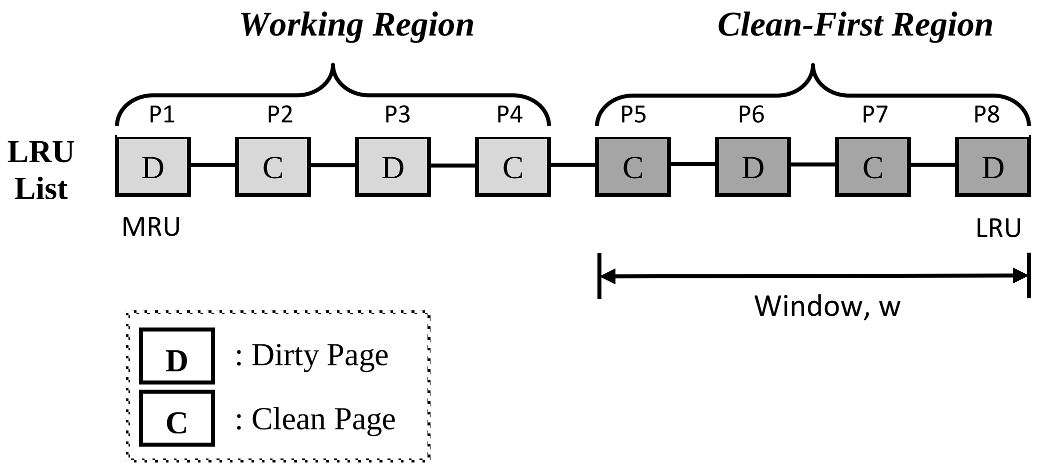


Figure 2: Example of basic CFLRU algorithm.

1.8.2 Proposed DCH-CFLRU Algorithm

DCH-CFLRU (DirtyHot_CleanHot Second Chance CFLRU) is the proposed enhanced version of the basic CFLRU, where second chance is given to 'dirty hot' and 'clean hot' pages. DCH-CFLRU also divides the LRU list into two regions to find a minimal cost point, as shown in Figure 2 below. The working region consists of recently used pages and most of cache hits are generated in this region. The clean-first region consists of pages which are candidates for eviction. This algorithm selects a clean cold and dirty cold page to evict in the clean-first region first to save flash write cost and improve hit ratio.

For example, under the CFLRU replacement algorithm, as discussed in previous section the order of page eviction is in the order of P7, P5, P8, and P6 in figure 2. But under the purposed DCH-CFLRU replacement algorithm it is done in the order of P5, P8, P7, and P6. Here the second chance is given to the pages P6 and P7 because they are flagged as 'Dirty Hot' and 'Clean Hot' respectively as shown in figure below:

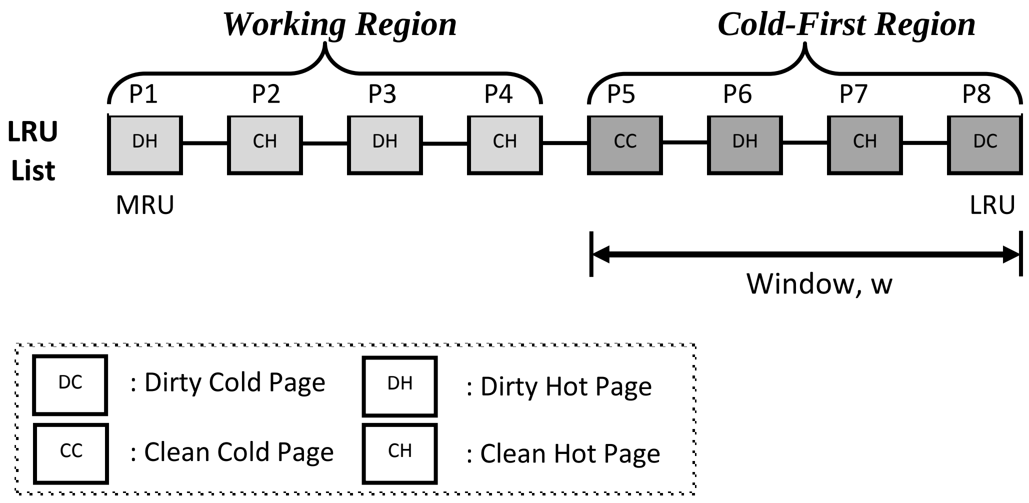


Figure 3: Example of DCH-CFLRU algorithm.

1.9 Objective

The main objective of this dissertation work is – To modify the basic CFLRU algorithm and design improved CFLRU algorithm for flash memory to achieve high hit ratio and minimum write count.

1.10 Motivation

This work is motivated by the observation and study of various replacement policies of the existing flash based operating systems. CFLRU can effectively reduce the number of write and erase operations by delaying the flush of dirty page in the buffer cache in many cases. However, it does not consider the access frequencies of data, keeping cold dirty data and evicting hot clean data will perform more read operations than normal LRU, degrading overall I/O performance. Hence the concept of giving second chance even to clean pages which are hot (i.e. those who are accessed second times) is main focus of this work. In this dissertation, the basic algorithm for flash memory, CFLRU, is modified in such a way that the new algorithm maintain balanced hit ratio and minimize the write count too.

1.11 Organization of the Thesis

Following the brief introduction about the dissertation and background study in this chapter, the rest of the content is divided into following five subsequent chapters.

Chapter 2 consists of literature review and methodology, which briefly reviews the related topics. Literature review includes summary of several traditional page replacement algorithms like Optimal, FIFO, LRU, LRU-K, MRU, ARC, 2Q etc and some algorithms that were designed for flash based system like CFLRU, CFDC, CCF-LRU, LRU-WSR, LIRS-WSR and ADLRU etc. This chapter also contains the research methodology part, which shows the main gist of the research.

Chapter 3 is contains the program development steps of simulation. It includes detail design of the algorithm, program, flowchart and tracing of basic CFLRU and proposed DCH-CFLRU.

Chapter 4 devoted to test result and analysis part, which includes data collection and details about generating traces of memory references that shows trace driven input, output results with several analyzing graphs, which are tested for both weak and strong locality workloads and different performance metrics.

Finally, conclusion of the work and recommendations are given in chapter 5.

CHAPTER 2

LITERATURE REVIEW AND METHEDOLOGY

2 Literature Review

Literature review describes the previous works and findings related to the field of study. First chapter briefly describe the background of page replacement algorithms. This chapter is dedicated to the description of some replacement algorithms, which are relevant to this work.

2.1 Traditional Buffer Replacement Algorithms

2.1.1 The Optimal Page Replacement Algorithm.

The best possible page replacement algorithm is easy to describe but impossible to implement. It goes like this. At the moment that a page fault occurs, some set of pages is in memory. One of these will be referenced on the very next instruction (the page containing that instruction). Other pages may not be referenced until 10, 100 or perhaps 1000 instruction later. Each page can be labeled with the number of instructions that will be executed before that page is first referenced. The optimal page replacement algorithm simply says that “the page with the highest label should be removed”. The problem with this algorithm is that it is unrealizable. At the time of page fault, OS has no way of knowing when each page will be referenced next [7].

2.1.2 FIFO Page Replacement Algorithm

Fist-In-First-Out (FIFO) page replacement algorithm [7] replaces oldest page during page fault. Conceptually FIFO is a queue with limited size. The operating system maintains a list of all pages currently in memory, with the page at the head of list the oldest one and the page at the tail the most recent arrival. On a page fault, the page at the head is removed and the new page added to the tail of the list. FIFO does not take advantage of locality trends. But, it can be modified very easily.

2.1.3 LRU Page Replacement Algorithm

The LRU policy [8] replaces the page that has been unused for longest time. This algorithm considers that a page that is recently used will probably be used again very soon, and a page that has not been used for a longest time, will probably remain unused. 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.

2.1.4 LRU-K Algorithm

LRU-K algorithm [13] dynamically keeps the records of k^{th} backward distance of each block where k^{th} backward distance BK(x, t) is defined as the number of accessed blocks from last k^{th} reference to the most recent reference. A block having maximum backward distance is evicted first. The parameter value k is positive integer like 1, 2 or 3. When k=1, it works like a simple LRU algorithm. Thus one may say that LRU-K algorithm as the generalization of LRU algorithm.

2.1.5 NRU Page Replacement Algorithm

Pages are categorized into four classes in not recently used (NRU) algorithm [7]. Class 0 contains pages that are neither referenced nor modified. Class 1 contains pages that are modified but not referenced. Class 2 contains pages that are referenced but not modified and Class 3 contains pages that are modified as well as referenced. During page fault NRU evicts any page from the lowest class.

2.1.6 LFU Page Replacement Algorithm

The LFU policy [14] keeps the track of the number of references to each pages, and the page selected for replacement is the page that has the least number of references. The policy is based on the presumption that the page that has been more frequently referenced in the past is more likely to be referenced in near future. This is accomplished by assigning a counter to every page that is loaded into the cache. When cache reaches its limit a page is needed to be removed to make room for new page, for this system will search the page with lowest counter and remove it from the cache.

2.1.7 LRFU Page Replacement Algorithm

Least Frequently Used (LFU) algorithm uses frequency factor for page replacement and LRU uses recency factor only. LRU and LFU are tuned to form adaptive algorithm called Least Recently Frequently Used policy (LRFU) [14] that considers both recency and frequency factors. The performance of the LRFU algorithm largely relies on a parameter called λ, which determines the relative weight of LRU or LFU and has to be adjusted according to the system configuration, even according to different workloads.

2.1.8 2Q Page Replacement Algorithm

The main intuition of 2Q algorithm [15] is the detection of real hot pages and removal of cold pages from the main memory. Those pages which are considered important for replacement are called hot pages and those pages which are considered less important for replacement are called cold pages. It consists of two lists of pages where the first one is queue managed by FIFO which contains data accessed once and another is hot list managed by LRU stack. The first queue is further partitioned into two sets Fin and Fout. The first referenced block is placed in Fin list while Fout contains only information of a missed block. The re-accessed block on Finlist is moved to the hot list which is managed by LRU. In this way algorithm distinguish frequently and infrequently accessed blocks. However, the problem is the management of two queues and migration of block from one queue to another which is complicated for hardware implementation and cycle consuming.

2.1.9 ARC Algorithm

ARC(Adaptive Replacement Cache) algorithm [16] keeps track of both frequently and recently used pages along with history data regarding eviction. ARC uses two types of LRU lists L1 and L2 to manage the pages. L1 holds pages accessed only once and L2 keeps the pages that were re-accessed at least once. These two lists are again partitioned in two sets top and bottom where top contains MRU part and Bottom contains LRU part so as $|T1+T2|=c$. where c is the cache size. Suppose $|T1| = p$ then $|T2|=c-p$. The parameter p is dynamic and it may be incremented and decremented based on the respective size of two sets B1 and B2 same parameter controls the replacement point in L1 and L2.

2.1.10 CLOCK Based Page Replacement Algorithm

CLOCK, CAR, CART and CLOCK-Pro are all clock based algorithms. The pages in the cache are organized as circular ring and each block associates the reference bit to record the access information [7]. CLOCK based algorithms hold the information regarding how frequently block has been accessed and but these algorithms have limitation that unable to detect an access pattern.

2.1.11 LIRS Algorithm

The LIRS [17] chooses a victim block by considering the inter-reference recency (IRR) of each block. The scheme divides cache blocks into two block sets: low IRR (LIR) block set and high IRR (HIR) block set. By replacing a block in HIR block set, which has comparatively low reference probability, the LIRS has good locality. It is accomplished by assuming that if the IRR of a block is large, the next IRR of the block is likely to be large again. However, the assumption is not always correct because of the constraint of timing scope.

2.2 Buffer Replacement Algorithms for Flash-Based Systems

2.2.1 CFLRU (Clean -First LRU)

CFLRU [10] is the first replacement algorithm designed for the flash based system. It modified the LRU algorithm. CFLRU divides the LRU list into two regions the working region and the clean first region is called window w to find a minimal cost point. The working region consists of recently used pages and most cache hits are generated in this region. The clean-first region consists of pages, which are candidates for eviction. CFLRU selects a clean page to evict from the clean-first region first to save flash write cost. If there is no clean page in this region, a dirty page at the end of LRU list is evicted.

2.2.2 Clean First Dirty Clustered (CFDC)

Clean First Dirty Clustered (CFDC) [18] manages the buffer in two regions: the working region W for keeping hot pages that are frequently and recently revisited, and the priority region P responsible for optimizing replacement costs by assigning varying priorities to page clusters. 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.

2.2.3 LRU-WSR replacement algorithm

LRU-WSR policy consider the cold/hot property of dirty pages, which is not tackled by the CFLRU algorithm. LRU-WSR always tries to remain hot dirty pages in the buffer and first replaces the clean pages or cold-dirty pages. The main difference between CFLRU and LRU – WSR is that the later considers the eviction of dirty cold pages. Hence, dirty cold pages will not reside in the buffer as long in the case of CFLRU which may cause the degradation hit ratio. In paper [4], LRU-WSR has been compared with LRU, CFLRU algorithms for different workloads collected from PostgreSQL, GCC, Viewperf and Cscope. LRU-WSR has been found 1.4 times faster than LRU. In most of the cases, LRU-WSR has higher hit ratio than others.

2.2.4 CCF-LRU

The authors of CCF-LRU [17] further refine the idea of LRU-WSR. It maintains two LRU queues, a cold clean queue and a mixed queue to maintain buffer pages. The cold clean queue stores cold clean pages (first referenced pages) while mixed queue stores dirty pages or hot clean pages. It always selects victim from cold clean queue and if cold clean queue is empty then employs same policy as that of LRU-WSR to select dirty page from mixed queue.

2.2.5 LIRS-WSR algorithm

The LIRS-WSR algorithm [2] is an improvement of LIRS [15] so that it can suit the requirements of flash-based systems. It is the integration of LIRS and WSR policy into single algorithm to reduce the writes of dirty pages from buffer to flash memory. It tries to improve the I/O performance by using recency and cleanness property of pages. LIRS-WSR algorithm improves the overall performance significantly by up 2 times faster than LRU algorithm by effectively reducing the number of physical writes and erase operations.

2.2.6 AD-LRU (Adaptive Double LRU)

AD-LRU algorithm [6] is buffer replacement algorithm for flash-based systems which focuses to reduce the write costs of the buffer replacement algorithm while keeping a high hit ratio. It tries to integrate the properties: recency, frequency, and cleanness of pages into the buffer replacement policy. AD-LRU has two LRU queues: Cold LRU queue and Hot LRU queue, to capture the concept of recency and frequency of the page references, among which Cold LRU queue stores the pages referenced only once and Hot LRU queue maintains the pages that are referenced at least twice. The sizes of these two LRU queues are dynamically adjusted according to changes in reference patterns. When a page is first referenced, it is put in the head of cold LRU queue. The pages move from cold LRU queue to head of hot LRU queue when it is referenced again and when a page in hot LRU queue is selected as victim, it is demoted to head of cold LRU queue.

2.3 Research Methodology

Research methodology is the very fundamental aspect, which underpin our work and methods we use in order to collect data and thus organize overall work in a systematic way [20]. In a scientific method of research, at first problem is formulated then output information is generated from collected input data and output is analyzed and finally the result is generalized. This dissertation work is truly scientific and flows in the same way. The topics flash memory and design has been studied from the early generation of computer. Page replacement algorithm is one of the major strategies to manage memory efficiently. The main contribution of this dissertation is to develop new flash memory friendly replacement algorithm DCH-CFLRU by modifying the basic CFLRU algorithm. Out of different types of research methodologies, this dissertation is based on the trace driven simulation approach. Page references used in this dissertation are secondary data, whereas data generated by the algorithm are primary data. Output information gathered is analyzed in a quantitative approach. Finally, conclusion is drawn with the help of analyzed data.

CHAPTER 3

PROGRAM DEVELOPMENT

3.1 Simulation Environment and Tools

The implementation is done in simulator, which is programmed in C# programming language in .Net Framework. The simulation and testing is done in the computer system having *Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz* speed processor with *4.00 GB RAM* and *64-Bit Windows 7 OS*. In this simulation the four different types of references used namely random trace, read most, write most and zipf access type. Where first three have 100k references and last one have 500k references.

3.2 Basic CFLRU Algorithm

As discussed in introduction portion earlier, CFLRU is the modified version of LRU algorithm addressing the flash memory properties. CFLRU divides the LRU list into two regions to find a minimal cost point. The size of the clean first region is called a window size. Large windows will increase the cache miss rate while small windows will increase the number of evicted dirty page. Hence the window size is kept half of cache size to balance both of these issue.

The working region consists of recently used pages and most of cache hits are generated in this region. The clean-first region consists of pages which are candidates for eviction. CFLRU selects a clean page to evict in the clean-first region first to save flash write cost. If there is no clean page in this region, a dirty page at the end of the LRU list is evicted.

3.2.1 Algorithm:

```
1. BEGIN
2. Set the cache size and window size (w) such that:
        2.1. W contains the LRU pages and
        2.2. Remaining part contains the MRU pages.
3. Fetch the page with its mode Read/Write
4. If cache is empty and fetched page is not found in cache then page fault occurs
        4.1. Insert fetched page at MRU position and adjust queue.
        4.2. Page fault++
5. Else //Cache is full
        5.1. If the fetched page is found in the cache then page hit occurs
                5.1.1. Insert fetched page at MRU position and adjust queue
        5.2. Else //Fetched page not found in cache and cache is full
                5.2.1. Page fault++
                5.2.2. If LRU clean page is found in w
                        5.2.2.1 Victim page = LRU clean page
                        5.2.2.2 Return the reference to the victim page
                        5.2.2.3 Insert fetched page at MRU position and adjust queue
                5.2.3. Else //If LRU clean page is not found
                        5.2.3.1. Victim page = LRU dirty page from w
                        5.2.3.2. Increment the write count
                        5.2.3.3. Return the reference to the victim page
                        5.2.3.4. Insert fetched page at MRU position and adjust queue.
6. END
```

3.2.2 Data Structure

The CFLRU algorithm can be implemented by using doubly linked list. Each node contain page access type whether it is dirty or clean. A doubly linked list is collection of sequential records called nodes. Where each node contains different properties of pages such as clean or dirty and each node references the next and previous one. The advantages of using doubly linked list are that if items are inserted and deleted from the list, the doubly linked list is very fast. Another beauty is that it can be traversed in both (forward and backward) more easily. The structure of doubly linked list is illustrated in figure 3.1.

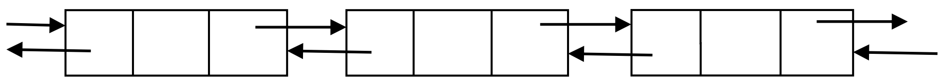


Figure 4: Structure of the Linked List

While implementing CFLRU algorithm, two operations have to be performed- insertion of new referenced node at front of list and deletion from the tail of list when replacement occurs. If the existed page in linked list is re-referenced then this used node can be delinked from its middle and move it to the head of the list. Structure of node is given below.

```
class CflruNode
    {
        public CflruNode prev { get; set; }
        public object Data { get; set; }
        public CflruNode next { get; set; }
        public Pagestatus status;

        public CflruNode()
        {
            Data = null;
            prev = null;
            next = null;
        }

        public CflruNode(object data, Pagestatus Pstatus)
        {
            status = Pstatus;
            Data = data;
            prev = null;
            next = null;
        }

      public enum PageStatus
             {
                    Dirty,
                    Clean
             };
```

3.2.3 Flowchart of Basic CFLRU algorithm

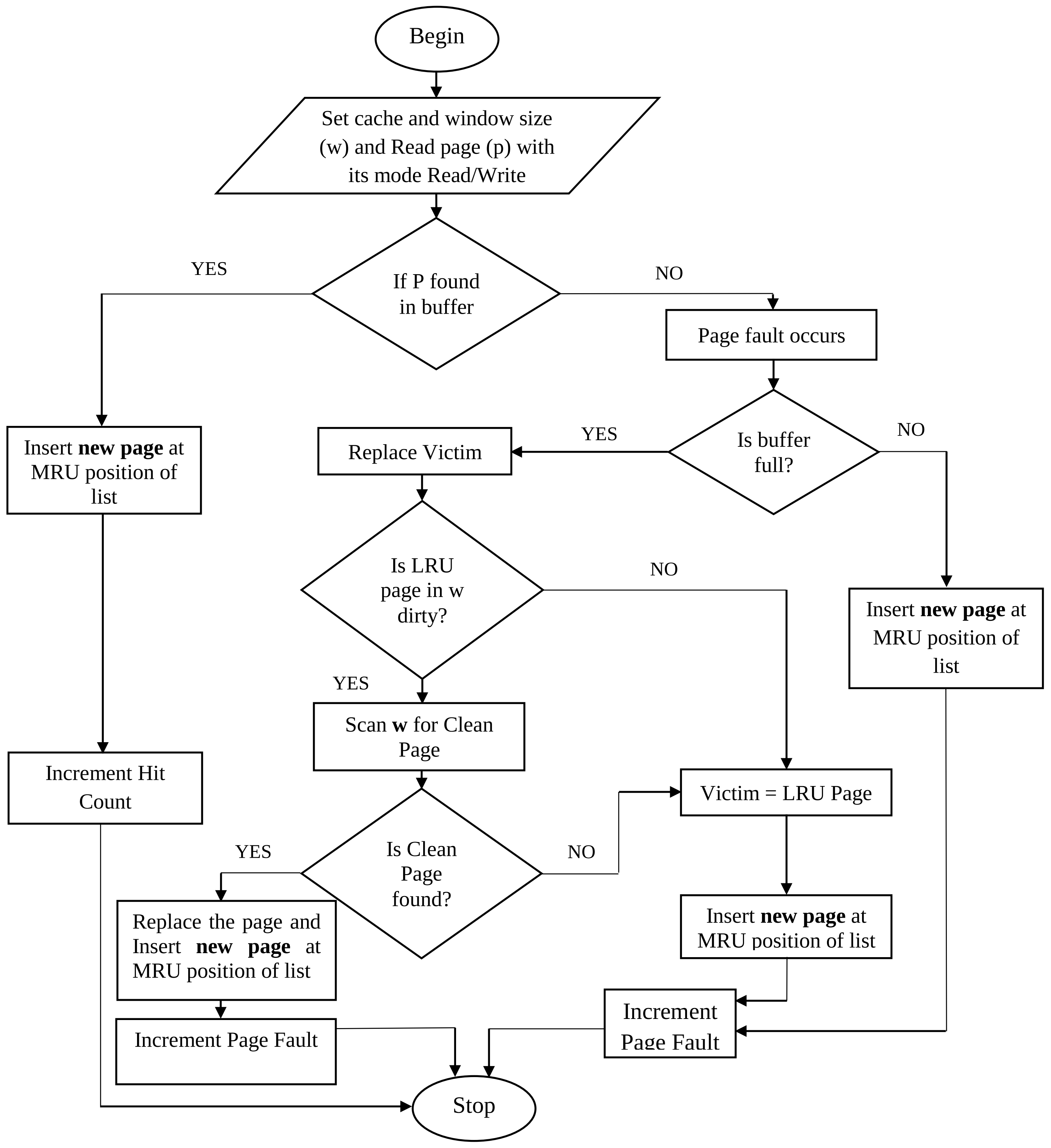

Begin
Set cache and window size (w) and Read page (p) with its mode Read/Write
If P found in buffer
YES
NO
Page fault occurs
Is buffer full?
YES
NO
Insert new page at MRU position of list
Replace Victim
Is LRU page in w dirty?
NO
YES
Insert new page at MRU position of list
Scan w for Clean Page
Increment Hit Count
Victim = LRU Page
Is Clean Page found?
YES
NO
Insert new page at MRU position of list
Replace the page and Insert new page at MRU position of list
Increment Page Fault
Increment Page Fault
Stop

Figure 5: Flowchart of Basic CFLRU

3.2.4 Tracing of Basic CFLRU Algorithm

Input References: 3,0 1,1 4,0 2,0 5,1 2,0 1,1 9,0 8,1 6,0

Cache Size: 4

No. of Distinct References: 8

Total No. of Reference: 10

Window Size: 2 (50% of total cache)

Page Status: 0 = Clean, 1 = Dirty

Notation:

MRU - Most Recently Used

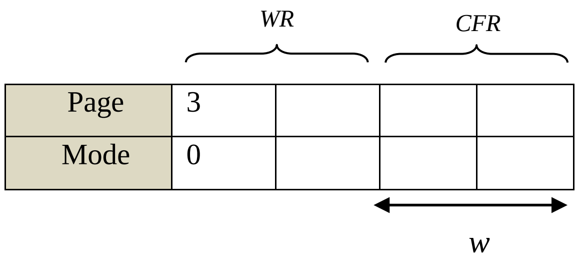
LRU - Least Recently Used

W - Window Size

WR - Working Region

CFR - Clean First Region

Step 1. Page Access: 3, 0

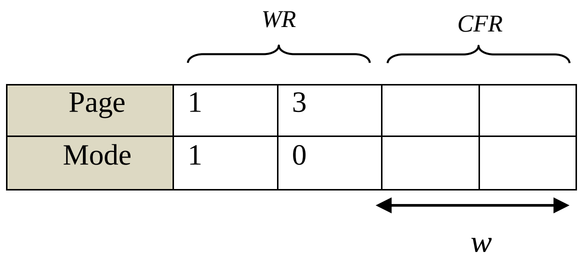


Page	3			
Mode	0			

Table 2: Basic CFLRU Tracing - State at virtual time 1

Page Fault = 1
Hit Count = 0
Write Count = 0

Step 2. Page Access: 1, 1

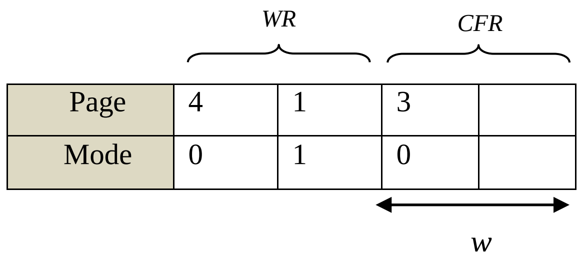


Page	1	3		
Mode	1	0		

Table 3: Basic CFLRU Tracing - State at virtual time 2

Page Fault = 2
Hit Count = 0
Write Count = 0

Step 3. Page Access: 4, 0

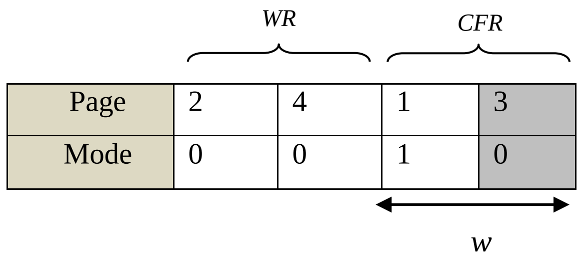


Page	4	1	3	
Mode	0	1	0	

Table 4: Basic CFLRU Tracing - State at virtual time 3

Page Fault = 3
Hit Count = 0
Write Count = 0

Step 4. Page Access: 2, 0



Page	2	4	1	3
Mode	0	0	1	0

Table 5: Basic CFLRU Tracing - State at virtual time 4

Page Fault = 4
Hit Count = 0
Write Count = 0

Step 5. Page Access: 5, 1

WR *CFR*

Page	5	2	4	1
Mode	1	0	0	1

w

Table 6: Basic CFLRU Tracing - State at virtual time 5

Page Fault = 5
Hit Count = 0
Write Count = 0

Since, the cache is full and the new page reference is fetched which is not found in the cache, so cache miss occurs. Here the page replacement occur. According to CFLRU page replacement policy LRU page from the window is replaced as victim.

Step 6. Page Access: 2, 0

WR *CFR*

Page	2	5	4	1
Mode	0	1	0	1

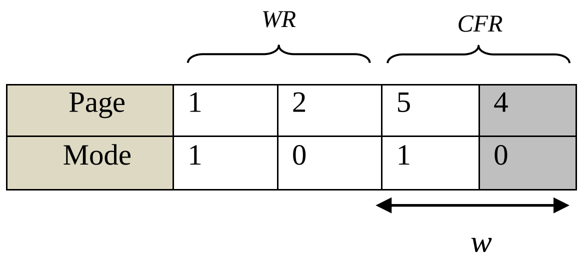
w

Table 7: Basic CFLRU Tracing - State at virtual time 6

Page Fault = 5
Hit Count = 1
Write Count = 0

In this step page 2,0 is found in cache. Hence page hit occur and repositioned at MRU of the queue.

Step 7. Page Access: 1, 1



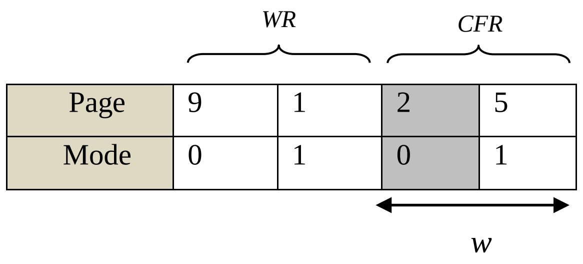
Page	1	2	5	4
Mode	1	0	1	0

Table 8: Basic CFLRU Tracing - State at virtual time 7

Page Fault = 5
Hit Count = 2
Write Count = 0

Here, again page hit occur, so the page is repositioned at MRU.

Step 8. Page Access: 9, 0



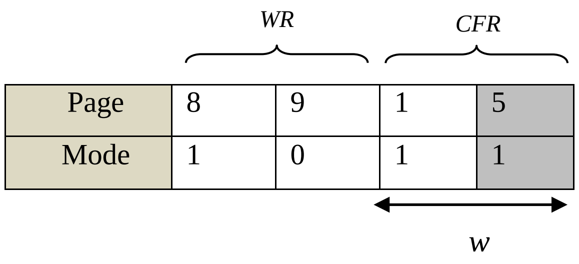
Page	9	1	2	5
Mode	0	1	0	1

Table 9: Basic CFLRU Tracing - State at virtual time 8

Page Fault = 6
Hit Count = 2
Write Count = 0

According to CFLRU replacement policy, LRU page 4,0 is victim and new page is inserted at MRU.

Step 9. Page Access: 8, 1



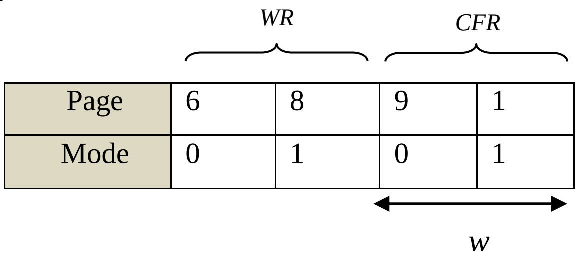
Page	8	9	1	5
Mode	1	0	1	1

Table 10: Basic CFLRU Tracing - State at virtual time 9

Page Fault = 7
Hit Count = 2
Write Count = 0

In this step, new page is fetched and not found in cache, hence page miss occur. And the LRU page is 'dirty'. So the algorithm look for clean page in window. That's why 2,0 is victim.

Step 10. Page Access: 3, 0



Page	6	8	9	1
Mode	0	1	0	1

Table 11: Basic CFLRU Tracing - State at virtual time 10

Page Fault = 8
Hit Count = 2
Write Count = 1

In this step, new page is fetched and not found in cache, hence page miss occur. And the LRU page is 'dirty'. So the algorithm look for clean page in window. But there is no clean page in window. So LRU dirty page become victim.

3.3 DCH- CFLRU Algorithm

As discussed in earlier sections of Chapter 1, DCH-CFLRU (DirtyHot_CleanHot Second Chance CFLRU) is the proposed improved algorithm based on basic CFLRU algorithm, where one chance is given to hot pages, whether they are dirty or cold, and replaced other pages even if it is dirty. This modification is done because CFLRU does not consider the access frequencies of data, keeping cold dirty data and evicting hot clean data will perform more read operations than normal LRU, degrading overall I/O performance. It divides the LRU list into two regions to find a minimal cost point. The working region consists of recently used pages and most of cache hits are generated in this region. The clean-first region consists of pages which are candidates for eviction. This algorithm selects a clean cold and dirty cold page to evict in the clean-first region first to save flash write cost and improve hit ratio.

3.3.1 Algorithm

```
1. BEGIN
2. Set the cache size and window size (w) such that:
    2.1. W contains the LRU pages and
    2.2. Remaining part contains the MRU pages.
3. Fetch the page with its mode Read/Write
4. If cache is empty and fetched page is not found in cache then page fault occurs
    4.1. Insert fetched page at the MRU position and adjust queue.
    4.2. Page fault++
    4.3. Set cold flag i.e. cold = 1
5. Else //Cache is full
    5.1. If the fetched page is found in the cache then page hit occurs.
        5.1.1. Insert fetched page at the MRU position and adjust queue.
        5.1.2. Clear cold flag i.e. cold =0
    5.2 Else //Fetched page not found in cache and cache is full
```

```
        5.2.1. Page fault++
        5.2.2. If LRU page is clean
            5.2.2.1. If page is clean cold
                5.2.2.1.1. Victim = LRU clean cold
                5.2.2.1.2. Insert fetched page at MRU position and adjust
                queue.
                5.2.2.1.3. Set cold flag i.e. cold = 1
            5.2.2.2. Else if page is clean hot
                5.2.2.2.1. While page is clean hot do
                    5.2.2.2.1.1. Move page to MRU of Clean First
                    region
                    5.2.2.2.1.2. Set cold flag i.e. cold = 1
                5.2.2.2.2. Victim = Clean cold OR dirty cold
        5.2.3. Else if LRU page is dirty
            5.2.3.1. If page is dirty cold
                5.2.3.1.1. Victim = dirty cold
                5.2.3.1.2. Insert fetched page at MRU position and adjust
                queue
                5.2.3.1.3. Set cold flag i.e. cold =1
                5.2.3.1.4. Write count++
            5.2.3.2. Else //if page is dirty hot
                5.2.3.2.1. While page is dirty hot do
                    5.2.3.2.1.1. Move page to MRU of Clean First
                    region
                    5.2.3.2.1.2. Set cold flag i.e. Cold = 1
            5.2.3.3. Victim = Dirty cold
6. END
```

3.2.2 Data Structure

The DCH-CFLRU algorithm also implemented using doubly linked list as CFLRU algorithm. Major difference is that while implementing DCH-CFLRU each node contain page access type whether it is dirty or clean and in addition status such as cold or not cold (hot). The node structure is given as below:

```
public class DCHCflruNode
    {

        public DCHCflruNode Prev { get; set; }
        public string Data { get; set; }
        public CflruNode Next { get; set; }
        public PageStatus Status;
        public PageFlage Flag;
        public DCHCflruNode(string data, PageStatus pstatus)
        {
            Status = pstatus;
            Data = data;
            Flag = PageFlage.Cold;
            Prev = null;
            Next = null;
        }
    }

    public enum PageStatus
    {
        Dirty,
        Clean
    };

    public enum PageFlage
    {
        Hot,
        Cold
    };
}
```

3.4.2 Flowchart of DCH-CFLRU Algorithm

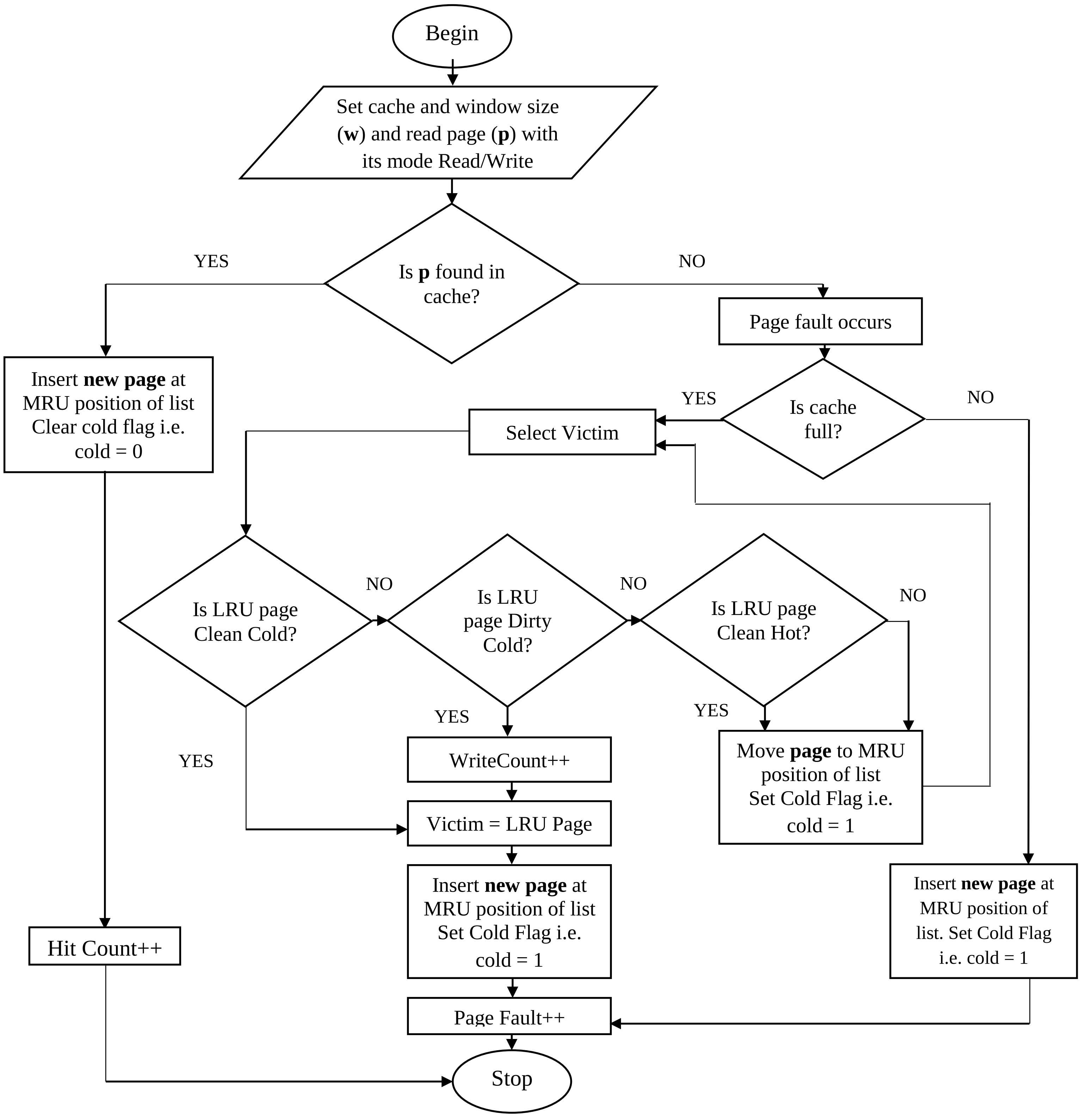

Begin
Set cache and window size (w) and read page (p) with its mode Read/Write
Is p found in cache?
YES
NO
Page fault occurs
Insert new page at MRU position of list Clear cold flag i.e. cold = 0
Is cache full?
YES
NO
Select Victim
Is LRU page Clean Cold?
NO
Is LRU page Dirty Cold?
NO
Is LRU page Clean Hot?
NO
YES
YES
YES
WriteCount++
Move page to MRU position of list Set Cold Flag i.e. cold = 1
Victim = LRU Page
Insert new page at MRU position of list Set Cold Flag i.e. cold = 1
Insert new page at MRU position of list. Set Cold Flag i.e. cold = 1
Hit Count++
Page Fault++
Stop

Figure 6: Flowchart of DCH-CFLRU Algorithm

3.4.3 Tracing of DCH-CFLRU Algorithm

Input References: 3,0 1,1 4,0 2,0 5,1 2,0 1,1 9,0 8,1 6,0

Cache Size: 4

No. of Distinct References: 8

Total No. of Reference: 10

Window Size: 2 (50% of total cache)

Page Status: 0 = Clean AND/OR Hot, 1 = Dirty AND/OR Cold

Notations:

MRU - Most Recently Used

LRU - Least Recently Used

W - Window Size

WR - Working Region

CFR - Cold First Region

Step 1. Page Access: 3, 0

	WR		*CFR*	
Page	3			
Mode	0			
Cold Flag	1			

$\xleftrightarrow{\quad w \quad}$

Table 12: DCH-CFLRU Tracing - State at virtual time 1

Page Fault = 1
Hit Count = 0
Write Count = 0

Step 2. Page Access: 1, 1

WR *CFR*

Page	1	3		
Mode	1	0		
Cold Flag	1	1		

w

Table 13: DCH-CFLRU Tracing - State at virtual time 2

Page Fault = 2
Hit Count = 0
Write Count = 0

Step 3. Page Access: 4, 0

WR *CFR*

Page	4	1	3	
Mode	0	1	0	
Cold Flag	1	1	1	

w

Table 14: DCH-CFLRU Tracing - State at virtual time 3

Page Fault = 3
Hit Count = 0
Write Count = 0

Step 4. Page Access: 2, 0

WR *CFR*

Page	2	4	1	3
Mode	0	0	1	0
Cold Flag	1	1	1	1

w

Table 15: DCH-CFLRU Tracing - State at virtual time 4

Page Fault = 4
Hit Count = 0
Write Count = 0

Step 5. Page Access: 5, 1

WR *CFR*

Page	5	2	4	1
Mode	1	0	0	1
Cold Flag	1	1	1	1

w

Table 16: DCH-CFLRU Tracing - State at virtual time 5

Page Fault = 5
Hit Count = 0
Write Count = 0

Here in this step, the cache is full and the new page reference is fetched which is not found in the cache, so cache miss occurs. Here the page replacement occur. According to DCH-CFLRU page replacement policy cold LRU page from the window is replaced as victim.

Step 6. Page Access: 2, 0

WR *CFR*

Page	2	5	4	1
Mode	0	1	0	1
Cold Flag	0	1	1	1

w

Table 17: DCH-CFLRU Tracing - State at virtual time 6

Page Fault = 5
Hit Count = 1
Write Count = 0

In this step page 2, 0 is found in cache. Hence page hit occur and repositioned at MRU of the queue and its cold flag has been cleared such that it will be considered as hot page in near future while replacement occurs.

Step 7. Page Access: 1, 1

WR *CFR*

Page	1	2	5	4
Mode	1	0	1	0
Cold Flag	0	0	1	1

w

Table 18: DCH-CFLRU Tracing - State at virtual time 7

Page Fault = 5
Hit Count = 2
Write Count = 0

Again, page hit occur and it is repositioned at MRU of the queue and its cold flag has been cleared such that it will be considered as hot page so that second chance will be given while replacement occurs afterward.

Step 8. Page Access: 9, 0

WR *CFR*

Page	9	1	2	5
Mode	0	1	0	1
Cold Flag	1	0	0	1

w

Table 19: DCH-CFLRU Tracing - State at virtual time 8

Page Fault = 6
Hit Count = 2
Write Count = 0

Here also, the cache is full and the new page reference is fetched which is not found in the cache, so cache miss occurs. Hence, page replacement should be done. According to DCH-CFLRU page replacement policy cold LRU page (whether it is clean or dirty) from the window is replaced as victim

Step 9. Page Access: 8, 1

$$\underbrace{}_{WR} \quad \underbrace{}_{CFR}$$

Page	8	9	1	2
Mode	1	0	1	0
Cold Flag	1	1	0	0

$\longleftrightarrow$ *w*

Table 20: DCH-CFLRU Tracing - State at virtual time 9

Page Fault = 7
Hit Count = 2
Write Count = 1

Here too, LRU page is victim according to DCH-CFLRU page replacement algorithm.

Step 10. Page Access: 6, 0

$$\underbrace{}_{WR} \quad \underbrace{}_{CFR}$$

Page	3	8	9	2
Mode	0	1	0	0
Cold Flag	1	1	1	1

$\longleftrightarrow$ *w*

Table 21: DCH-CFLRU Tracing - State at virtual time 10

Page Fault = 8
Hit Count = 2
Write Count = 1

Here, in this step 10, when new page occurs, cache is full and algorithm find out the reference page '2' as LRU page, but its cold flag is NOT set (i.e. the page is hot) so second chance is given and inserted into the MRU position of cold first region and it's status is also changed to cold and queue is adjusted. Then, algorithm checks for the page to evict at LRU and it finds reference page '1' as dirty and hot again, that's why second chance is given to this page also and queue is adjusted. Now, algorithm find '2' as a victim, because this time it is cold page.

CHAPTER 4

TEST RESULTS & ANALYSIS

4.1 Data Collection

Data is the basic element of any sort of research work which gives the meaningful result after processing and analyzing. Using the simulation tool discussed above, data are collected and tested carefully to obtain accurate and desired result.

Here, in this dissertation work four types of trace data are used, i.e., Random trace, Read most trace, write-most trace, and Zipf trace as Workload 1, Workload 2, Workload 3 and Workload 4 respectively. These four different traces with different nature are used as workloads for analyzing existing basic CFLRU and DCH-CFLRU. These data are real memory traces. Workload represents different locality of memory reference pattern that are generated during execution of process in real Operating System. There are total 100,000 page references in each of the first three traces, which are restricted to a set of pages whose numbers range from 0 to 49,999. The total number of page references in the Zipf trace is set to 500000 in order to obtain a good approximation, while the page numbers still fall in [0, 49999]. Zipf trace has a referential locality "20/80" meaning that eighty percent of the references deal with the most active twenty percent of the pages. Sample of Workload 1, Workload 2, Workload 3 and Workload 4 are in appendix A, appendix B, appendix C and appendix D respectively. Table 22 to Table 25 shows the details concerning these workloads [6].

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

Table 22: Detailed properties of Random Access Reference

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

Table 23: Detailed properties of Read Most Reference

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

Table 24: Detailed properties of Write Most Access Reference

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

Table 25: Detailed properties of Zipf Access Reference

4.2 Testing

For the analysis purpose, both the algorithm are tested with four distinct workload. Using the developed CFLRU and DCH-CFLRU simulator test has been carried out by varying cache size from 512 to 18192. Table below shows the test result.

i) Test Result of Workload 1 (Random Access)

Total No. of references = 100,000 || Total No. of Distinct Pages = 35,843

Cache Size	Basic CFLRU		DCH-CFLRU	
	Hit Rate	*Write Count*	*Hit Rate*	*Write Count*
512	1.57	49161	1.53	48971
1024	3.18	48428	3.19	47882
2048	6.39	46868	6.41	45850
4096	12.69	43726	12.70	41940
8192	24.79	37428	24.62	34808
10192	30.49	34345	30.38	31395
12192	35.92	31349	35.94	28272
14192	41.04	28361	41.20	25281
16192	46.12	25448	46.29	22600
18192	51.03	22421	51.14	19918

Table 26: Test result of Workload 1 (Random Access) for both algorithm.

ii) Test Result of Workload 2 (Read Most Access)

Total No. of references = 100,000 || Total No. of Distinct Pages = 35,834

Cache Size	Basic CFLRU		DCH-CFLRU	
	Hit Rate	*Write Count*	*Hit Rate*	*Write Count*
512	1.62	9589	1.7	9728
1024	3.28	9260	3.26	9561
2048	6.38	8569	6.39	9253
4096	12.55	7194	12.71	8557
8192	24.71	4329	24.73	7252
10192	30.46	2902	30.35	6621
12192	35.91	1488	35.77	6036
14192	41.31	1000	41.18	5396
16192	46.44	800	46.26	4796
18192	51.32	689	51.12	4175

Table 27: Test result of Workload 2 (Read Most Access) for both algorithm.

iii) Test Result of Workload 3 (Write Most Access)

Total No. of references = 100,000 || Total No. of Distinct Pages = 35,824

Cache Size	Basic CFLRU		DCH-CFLRU	
	Hit Rate	*Write Count*	*Hit Rate*	*Write Count*
512	1.67	87744	1.67	86899
1024	3.31	86393	3.31	84737
2048	6.51	83712	6.48	80634
4096	12.66	78408	12.64	72916
8192	24.58	67971	24.51	59216
10192	30.15	62977	30.12	53273
12192	35.50	58114	35.47	47831
14192	40.85	53242	40.72	42647
16192	46.03	48427	46.08	37600
18192	50.99	43721	54.65	33104

Table 28: Test result of Workload 3 (Write Most Access) for both algorithm.

iv) Test Result of Workload 4 (Zipf Trace Access)

Total No. of references = 500,000 || Total No. of Distinct Pages = 40,612

Cache Size	Basic CFLRU		DCH-CFLRU	
	Hit Rate	*Write Count*	*Hit Rate*	*Write Count*
512	30.39	180406	29.98	168061
1024	38.02	163207	37.58	149449
2048	46.76	143278	46.33	128292
4096	56.77	119804	56.28	103904
8192	68.21	91125	67.76	75536
10192	72.12	80881	71.72	65795
12192	75.49	71774	75.07	57592
14192	78.33	63727	77.97	50402
16192	80.84	56419	80.51	44064
18192	83.09	49688	82.78	38350

Table 29: Test result of Workload 4 (Zipf Trace Access) for both algorithm.

4.3 Analysis

The result obtained by simulating these algorithms is analyzed by using different graphs. Hit ratio and write count is taken as measure criteria for measuring their performance. The algorithm that has higher hit ratio and lower write count is considered to be good in the case of flash-based system. The result is different for different workloads according to the reference pattern.

4.3.1 Hit Ratio Analysis

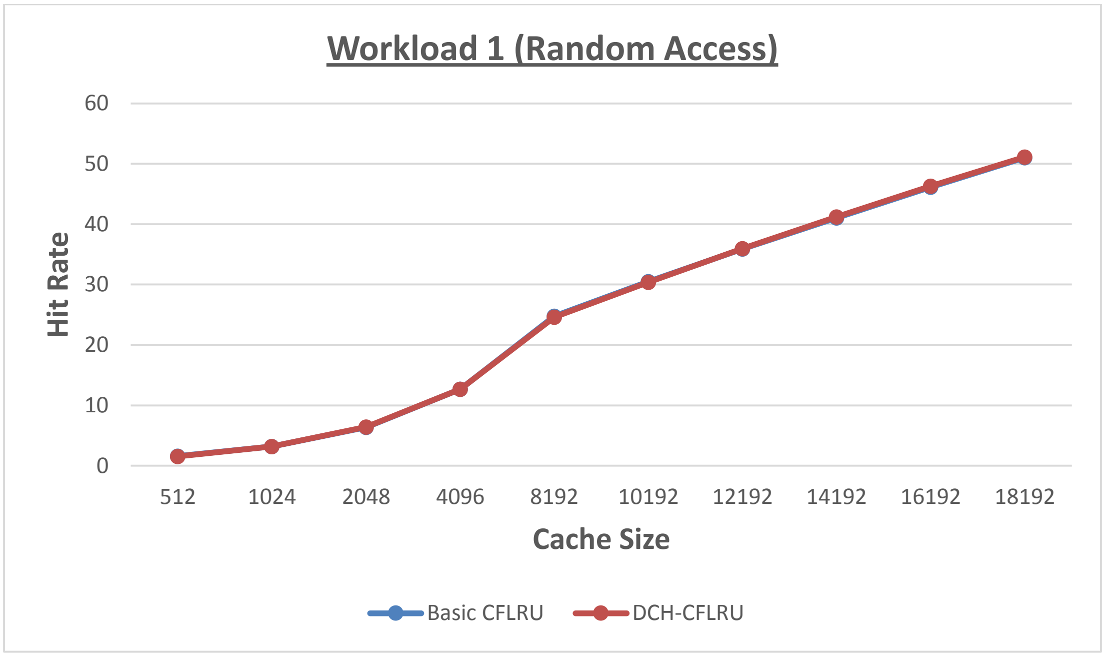


Figure 7: Hit ratio analysis for workload 1 (Random Access) for both algorithm.

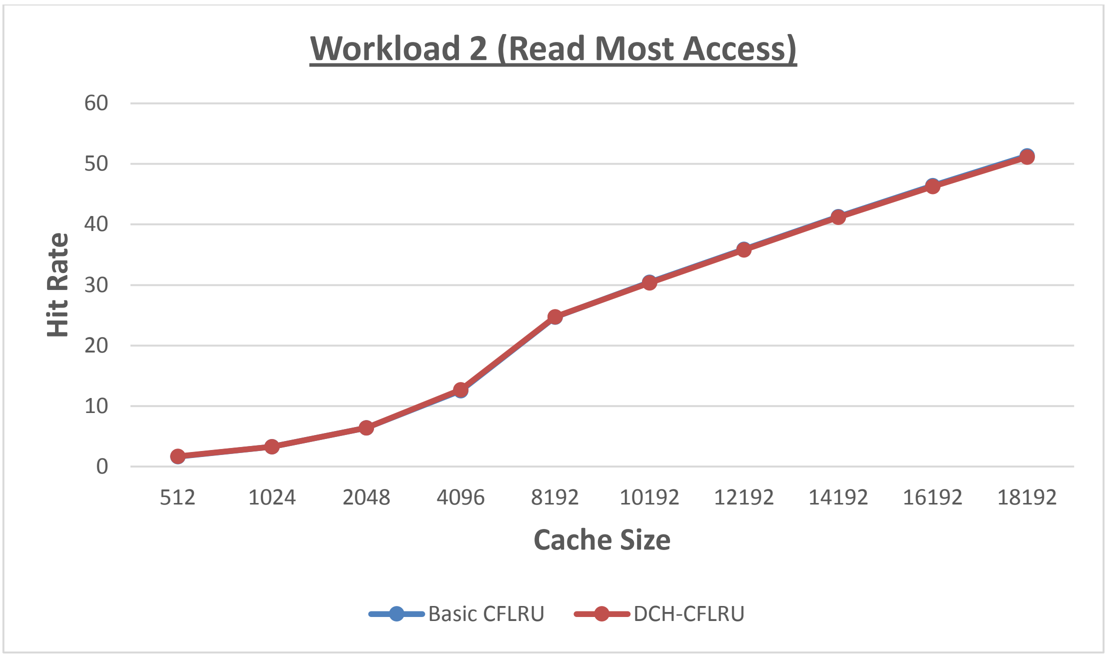


Figure 8: Hit ratio analysis for workload 2 (Read Most Access) for both algorithm.

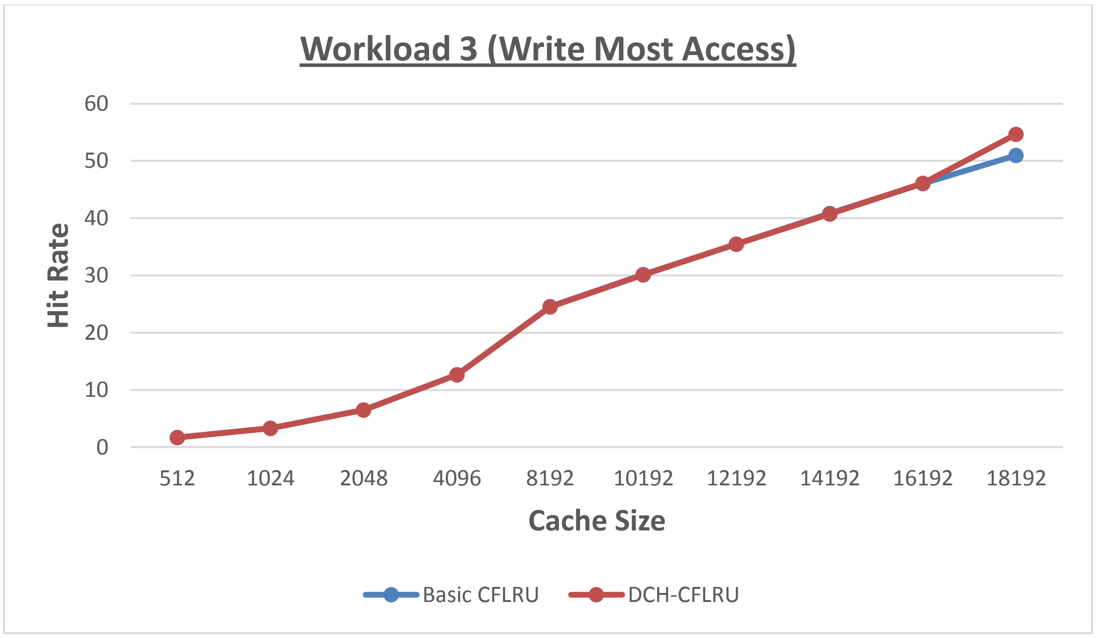


Figure 9: Hit ratio analysis for workload 3 (Write Most Access) for both algorithm.

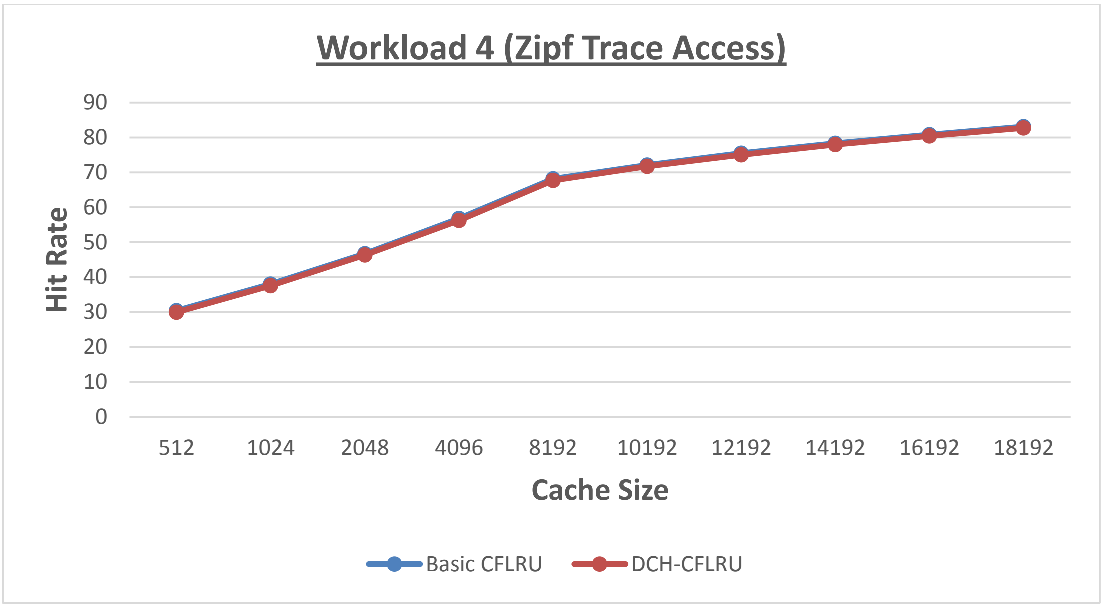


Figure 10: Hit ratio analysis for workload 4 (Zipf Trace Access) for both algorithm.

The line graph from Figure 7 to Figure 10 shows that the basic CFLRU algorithm and DCH-CFLRU algorithm have almost identical hit ratio. Hit ratio is the case when the page requested by process is found on memory. Here, both the algorithm are tested with four different types of page reference traces, namely random, read-most, write-most and zipf type with varied cache size. In these workloads there is not clear distinction between hot and cold pages as reference locality is not high. Despite the nature of page reference in write most workloads, DCH-CFLRU has slightly better hit ratio when cache size increases. Reason of occurring almost identical hit ratio is that in both algorithm try to put dirty page in memory for a while, but DCH-CFLRU algorithm gives second chance to all the pages other than cold pages.

The graph of Figure 10 shows significantly difference in hit rate in zipf workload in comparison to other three page reference type. This is due to high reference locality of page references in zipf trace where 80% of pages references deal with active 20 % of pages. But both the algorithm gives same result in this page reference pattern too. In this workload both the algorithm has higher hit ratio even in small cache size but the case is different in other workloads.

As a maximum value DCH-CFLRU has 4% higher hit rate than basic CFLRU algorithm for write most workload when cache size is larger than 16000 approximately. But in other workloads there is no significant difference in hit ratio.

4.3.2 Write count Analysis (Line Graph)

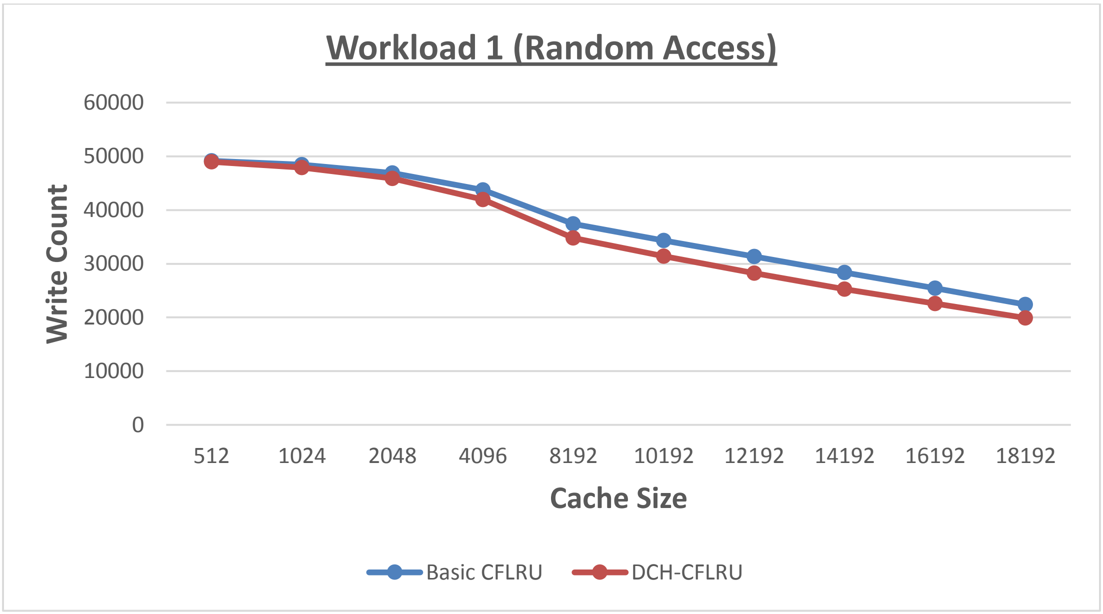


Figure 11: Write count analysis for workload 1 (Random Access) for both algorithm.

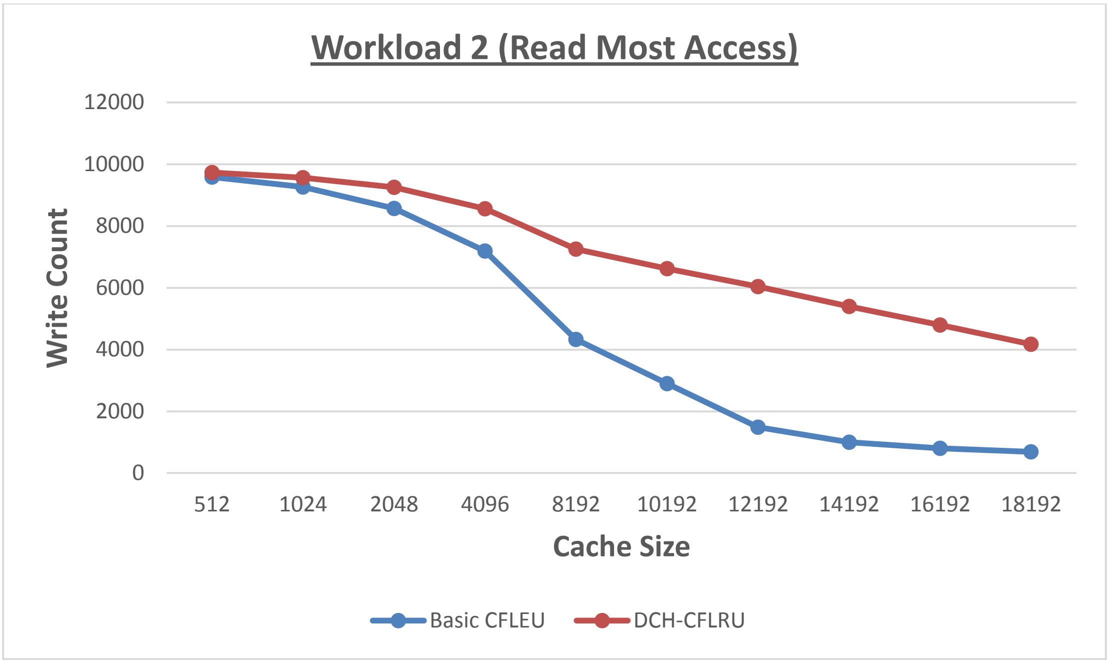


Figure 12: Write count analysis for workload 2 (Read Most Access) for both algorithm.

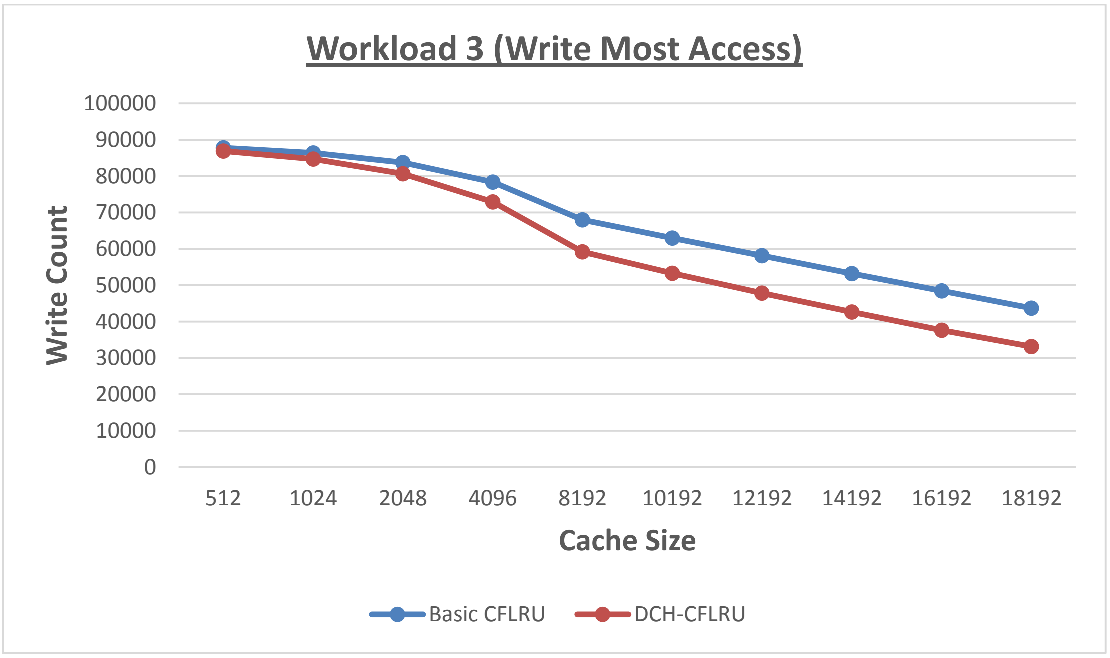


Figure 13: Write count analysis for workload 3 (Write Most Access) for both algorithm.

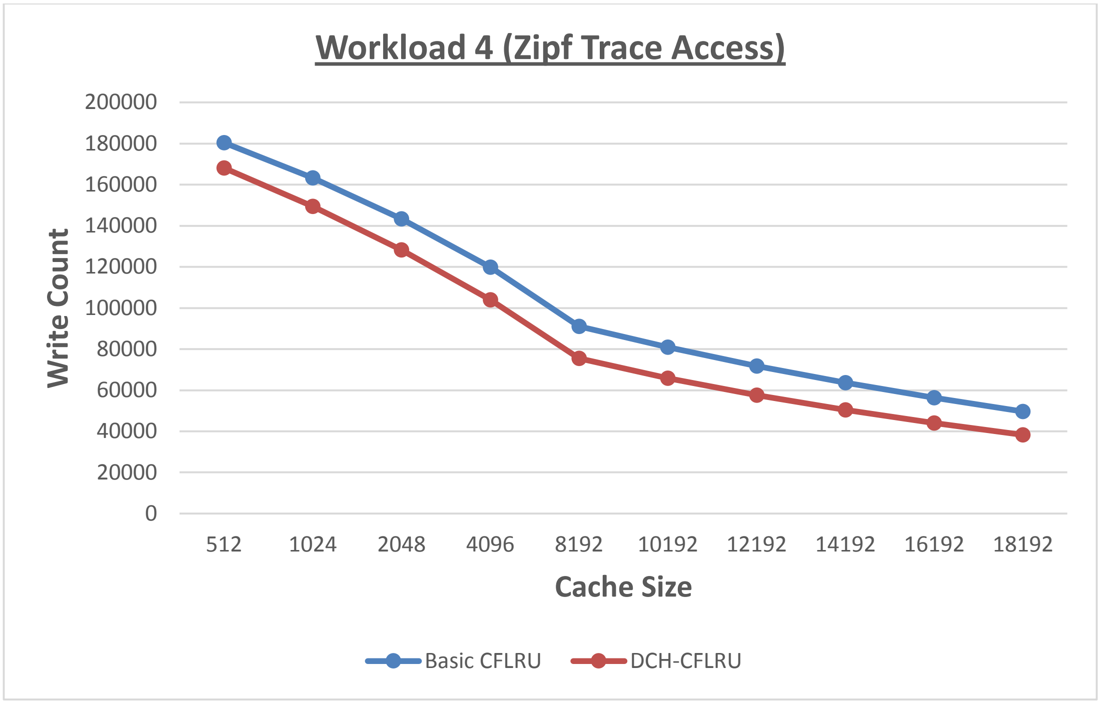


Figure 14: Write count analysis for workload 4 (Zipf Trace Access) for both algorithm.

The graphs in figure 11 to 14 shows the write count of basic CFLRU and proposed DCH-CFLRU in four different workloads. Write count is the number of dirty pages flushed to flash memory. The number is obtained by counting the eviction of page references with write request during page replacement event. From the above line graphs it is clear that DCH-CFLRU has less write count for all the workload except read most workload.

Random access reference pattern is uniformally distributed having 50-50% read/write pages. When the cache size is smaller, there is small differences between the algorithms. But as cache size increases DCH-CFLRU outperform CFLRU significantly. This is because, when buffer size increases, DCH-CFLRU get large room in working region to moves those write pages to MRU giving second chance and stay in buffer for long duration. And thus have more chances of hit in subsquent fetch.

Workload 2 has read most access pattern containing 10% writes and 90% reads page references. For small cache size both the algorithm have almost equal number of write counts. But when cache size increase CFLRU outperform the DCH-CFLRU, tending to zero in case of CFLRU and steady decrement in DCH-CFLRU. Reason behind this is that, in workload 2, there are very less number of write pages and when the cache size increase largly, there is very high probablity of cantaining all write pages in cache.

In type three workload, write most, DCH-CFLRU gives better result CFLRU when the cache size increased. This type of workload contains 90% write pages and 10% read pages. This is due to there are large number of write pages and DCH-CFLRU has policy of keeping those pages in working region giving one chance.

In context of zipf trace, there is a constant difference in term of write count for both algorithms. This is because the trace has 50%/50% read/write references but has high reference locality of 80% page references are references to 20% of pages. DCH-CFLRU adapts changes in reference pattern and locality. So it has less write count than CFLRU for all buffer size.

Excluding read most traces, in all the three workload DCH-CFLRU has less number of write counts upto 14% (14900 write count) less than basic CFLRU.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Conclusion

In this dissertation work, an efficient buffer algorithm for flash memory, named DCH-CFLRU is proposed, originally based on basic CFLRU algorithm. It gives the second chance to hot pages whether they are dirty or clean and evict cold pages when replacement occurs. Trace-driven experiment in a simulation environment using four types of synthesized traces. The experiments have shown that the proposed algorithm DCH-CFLRU outperforms CFLRU in terms of write count in all the workload except read most workload. Whereas, in read most case CFLRU gives better result. Regarding the hit ratio, both algorithm gives the almost same result and DCH-CFLRU, at least preserve the CFLRU's performance.

Newly proposed algorithm, DCH-CFLRU proves that by adopting the strategy of giving second chance to both clean hot and dirty hot pages can reduce the number of write count in various workload pattern. This nature of replacement algorithm efficiently support the characteristic of flash storage devices.

Recommendation

In this work, the window size is kept fixed during the experiment, to balance hit ratio and write count. But it is yet to discover that, what value of window gives better result in which types of workload. Recommendation is that, finding the optimal value of window size which could refine and improves the performance of the algorithm. Algorithm could be more dynamic in nature by using various technique like adding more queue to get higher hit ratio and lesser write count on several workloads too.

And, it is also recommended that, the analysis could be done regarding the performance of DCH-CFLRU with respect to other flash memory based page replacement algorithms using other benchmarks and additional real input traces for further performance evaluation.

References

[1]. Bez, Roberto. Camerlenghi, E`millo. Modelli, Albrto. And Visconti, Angelo, "Introduction to Flash memory. Proceedings of the IEEE", VOL. 91, NO. 4, April 2003.

[2]. Jung, H. Shim, H. Park, S. Kang, S. Cha, J., "LRU-WSR: Integration of LRU and Writes Sequence Reordering for Flash Memory", IEEE Trans. On Consumer Electronics, 2008.

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

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

[5]. Dirik, C., "Performance Analysis of NAND Flash Memory Solid-State Disks" Doctor of Philosophy, 2009.

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

[7]. Tanenbaum, A.S., "Modern Operating Systems", 2nd Edition. Prentice Hall, 2007, pp 199-212.

[8]. Silberschatz, A. Galvin, P. B. Gagne, G., "Operating System Concept, 8th edition", Wiley Student Edition, 2007.

[9]**.** Hong, W., "Study of page replacement Algorithm based on experiment." International conference on mechanical engineering and automation advances in biochemical engineering vol.10, 2012

[10] Parthey, D., "Analyzing Real-Time Behavior of Flash Memories", Chemnitz University of Technology, 2006.

[11] Paajanen, H., "Page Replacement in Operating System Memory Management", Uni of Jyvaskyla, 2007

[12]. Park, S. Jung, D. Kang, J. Kim, J. Lee, J., "CFLRU: Replacement Algorithm for Flash Memory", In: CASES'06, 2006.

[13]. O’Neil, E.J. O’Neil, P.E. Weikum, G., "The LRU-K Replacement. Algorithm for Database Disk Buffering", In: Proc.of SIGMO, 1993.

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

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

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

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

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

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

[20]. Gille, D., "Study of Different Cache Line Replacement Algorithms in Embedded Systems" ARM France SAS, 2007.

Bibliography

i. https://en.wikipedia.org/wiki/Flash_memory

ii. http://dl.acm.org/citation.cfm?id=2038735&dl=ACM&coll=DL&CFID=741904077&CFTOKEN=57026076

iii. Rawat, B. B., *Quantitative Evaluation of Buffer Replacement Algorithms for Flash Memory Based Systems, CDCSIT, Tribhuvan University, 2014.*

iv. Mahara, D. S., *A Comparative Evaluation of Buffer Replacement Algorithms LIRS-WSR and AD-LRU for Flash Memory Based Systems, CDCSIT, Tribhuvan University*, 2014.

v. Jiang, S. Zhang, X. Chen, F., "*CLOCK-Pro: An Effective Improvement of the Replacement*", ATEC ’05 Proceedings of the annual conference on USENIX Anuual Technical Conference, 2005, pp 35.

vi. Paajanen, H., "*Page Replacement in Operating System Memory Management*", Master's Thesis in Information Technology. University of Jyvaskyla. Department of Mathematical Information Technology, 2007.

Appendix 1: Random Trace Input Sample

1,8575	0,29923	0,49554	0,7786	1,1066	1,8672
0,17754	1,3050	1,18797	1,43211	0,9039	1,33281
0,33289	0,4043	1,6747	0,47655	1,6477	0,8337
0,3838	0,39113	0,31276	0,42213	1,41170	0,16815
0,19942	1,11686	1,786	0,919	0,23504	0,14078
1,25113	1,25837	1,42798	0,603	1,32354	0,23123
1,35145	0,4941	0,30971	0,4844	0,14280	1,38627
1,1939	0,7882	0,42594	0,44923	1,36795	1,3974
0,40780	0,39262	1,49503	1,29324	1,8732	1,39029
0,12831	1,32631	0,23075	0,26292	1,46002	0,24143
0,31724	0,36490	1,8717	1,31526	1,4880	0,45127
1,37162	1,11934	1,13521	1,38097	1,5637	1,6153
1,861	1,8851	1,988	1,39819	1,21680	0,21868
1,35912	1,16962	0,22467	0,30117	1,3496	1,3032
0,39216	0,37665	1,12586	1,14208	1,3220	0,49348
1,10863	1,23980	1,45284	1,27844	0,13282	0,27357
0,15454	0,41727	1,39329	1,8361	1,42670	0,2787
0,32425	0,15074	0,45058	1,16455	0,11669	1,41676
0,42141	1,19029	0,14795	1,5699	0,2716	0,45740
1,34769	1,1750	0,21120	0,10670	1,49749	1,12125

Appendix 2: Read Most Trace Input Sample

0,18468	0,15112	0,11839	0,12275	0,22024	0,28294
0,34788	1,16270	0,40554	0,48986	1,40558	0,7946
0,16056	0,36671	0,583	0,20596	0,3330	0,2748
0,48011	0,18313	0,25881	0,10714	1,4020	0,36907
0,3932	0,45554	0,8353	0,5776	0,15769	0,5078
0,25030	0,37062	0,22762	0,39721	0,9032	0,27022
1,16770	0,14371	1,8476	0,29714	1,33398	1,14669
0,21941	0,48320	0,2452	1,27136	0,26794	0,37419
0,29929	0,2162	0,21562	0,8363	0,17359	0,12382
0,38177	0,27188	0,49935	0,41959	0,6901	0,8955
0,7153	0,24383	0,38578	0,3181	0,46199	0,43073
0,18859	0,9427	0,24493	1,11001	0,45489	0,4139
0,3268	0,23379	0,1086	0,25168	0,49561	0,37292
0,28072	0,39283	0,48678	0,8592	0,37635	0,31386
0,9315	0,24410	0,12792	1,39601	0,1892	0,15131
0,28505	0,16370	0,589	0,8962	0,46703	0,44501
0,49719	0,4033	0,14086	0,26716	0,25444	0,40518
0,44878	0,21778	0,23292	0,39017	0,43238	0,6139
0,18673	0,45129	0,9893	0,11701	0,11309	0,49892
0,6906	0,16800	0,30087	1,18572	0,30281	0,22521

Appendix 3: Write Most Trace Input Sample

1,23302	1,27730	1,47528	1,34617	1,16350	1,35557
1,13471	0,25933	1,7678	1,29231	1,22821	1,29788
1,14359	0,23356	0,23761	0,5727	0,28702	1,47754
1,21971	1,32532	1,48844	1,27015	1,6451	1,17588
1,20334	1,5623	1,29099	0,34690	1,32037	1,17838
1,31838	0,2742	0,3500	1,40669	1,47186	1,42179
1,49621	1,14485	1,26389	1,19798	1,18653	1,38931
1,28811	1,21326	1,2172	1,28686	1,45641	1,32941
1,3897	1,36952	1,11354	1,20363	1,16091	1,38931
1,43040	1,29226	0,6325	1,34354	1,29841	1,32941
1,27843	1,15461	1,42822	1,44407	1,10516	1,47935
1,32634	0,16820	1,26389	1,24634	1,7979	1,48613
1,27524	1,22017	1,5102	1,16787	1,168	1,42451
1,16779	0,9035	1,34110	1,38452	1,30352	1,13432
1,43505	1,25265	0,21757	1,42688	1,3196	1,25523
1,25441	1,11212	1,20043	1,40239	1,46214	1,16903
1,44214	1,33356	1,43395	1,1854	1,30409	1,30117
1,2239	1,27032	1,14065	0,4638	1,43742	1,34689
1,21654	1,14152	1,16337	1,42761	1,48075	1,32151
1,40382	1,15124	1,43311	1,26164	1,26852	1,49445

Appendix 4: Zipf Trace Input Sample

1,8550	0,14300	0,5565	1,40	0,631	1,5
0,3609	0,1	1,69	0,3	0,50	0,47719
1,654	0,16	1,7723	1,47	0,44415	1,8
1,17913	0,18	0,39491	0,6	1,800	1,889
0,145	1,1167	0,2020	0,8631	0,1847	0,345
0,2550	0,7	1,1	0,7375	0,1353	0,1136
1,5970	1,27473	0,16	0,9649	0,115	0,242
0,2461	0,47	1,17217	0,3530	0,28497	1,10958
1,33806	0,127	1,3717	0,21	1,2611	0,1178
0,17	0,286	1,3294	1,508	0,697	0,17
0,1	0,35	1,31	1,8	1,1728	1,3
0,17	1,1	1,40143	0,16	0,1	1,20063
0,7186	0,63	0,49198	1,20349	1,32	0,1992
1,370	0,15	0,15221	1,4506	0,57	0,4
0,159	0,17	0,191	1,279	1,358	1,7485
0,10290	0,574	0,49491	1,111	0,522	1,1406
0,54	1,1815	1,2842	0,1472	1,4	0,168
0,4	0,173	1,2797	0,2768	0,612	1,87
1,40078	0,6	0,25825	0,36002	0,2599	1,602
0,481	0,9172	0,7165	0,168	1,2	0,1638

Appendix 5: Source Code of Basic CFLRU Algorithm

```
Basic CFLRU Algorithm:
namespace ThesisLastUpdate
{
    //this for to show page status
    public enum Pagestatus { dirty, clean };

    //node structure
    class CflruNode
    {

        public CflruNode prev { get; set; }
        public object Data { get; set; }
        public CflruNode next { get; set; }
        public Pagestatus status;

        public CflruNode()
        {
            Data = null;
            prev = null;
            next = null;
        }

        public CflruNode(object data, Pagestatus Pstatus)
        {

            status = Pstatus;
            Data = data;
            prev = null;
            next = null;
        }

        // returns entire list
        public CflruNode(Pagestatus statuss, CflruNode datavalue)
            : this(statuss, null, datavalue, null)
        {
        }

        //this constructor fills all parameters on their respective fields
        public CflruNode(Pagestatus statuss, CflruNode previousnode, CflruNode datavalue,
CflruNode nextnode)
        {
            status = statuss;
            prev = previousnode;
            Data = datavalue;
            next = nextnode;
        }
    }


    /// <summary>
    /// Class for List of nodes with all necessary operation in the list
    /// </summary>
    class cache
    {
        public CflruNode firstnode { get; set; }
        public CflruNode lastnode { get; set; }
        public string name;

        /// <summary>
        /// Initialization of the list
        /// </summary>
        /// <param name="listname"></param>
        public cache(string listname)
        {
            name = listname;
            firstnode = null;
            lastnode = null;
        }
        public cache()
            : this("list")
        { }

        /// <summary>
        /// Insert node at the firts position of the list
        /// </summary>
        /// <param name="insertNode"></param>
        public void InsertAtFront(CflruNode insertNode)
        {
            if (IsEmpty())
            {
                firstnode = lastnode = insertNode;
            }
            else
            {
                insertNode.next = firstnode;
                firstnode.prev = insertNode;
                firstnode = insertNode;
            }

        }

        /// <summary>
        /// To check whether the list is empty
        /// </summary>
        /// <returns></returns>
        public bool IsEmpty()
        {
            return firstnode == null;
        }


        /// <summary>
        /// To search the node in the list
        /// </summary>
        /// <param name="node"></param>
        /// <returns></returns>
        public CflruNode findnode(CflruNode node)
        {
            CflruNode newnode = new CflruNode();

            for (CflruNode temp = firstnode; temp != lastnode.next; temp = temp.next)
            {
                if (temp.Data.Equals(node.Data))
```

```
            {
                newnode = temp;
            }
        }
        return newnode;
    }

    /// <summary>
    /// Search the node in the List
    /// </summary>
    /// <param name="nodetobeSearched"></param>
    /// <returns></returns>
    public bool search(CflruNode nodetobeSearched)
    {
        bool checkpoint = false;

        if (IsEmpty())
        {
            checkpoint = false;
        }

        else
        {
            for (CflruNode temp = firstnode; temp != lastnode.next; temp = temp.next)
            {
                if (temp.Data.Equals(nodetobeSearched.Data))
                {
                    checkpoint = true;
                }
            }
        }
        return checkpoint;
    }

    public bool CheckCleanPage(int noOfStepsTomove)
    {
        bool checkpoint = false;
        int count = 0;
        if (IsEmpty())
        {
            checkpoint = false;
        }

        else
        {
            for (CflruNode temp = lastnode; temp != firstnode.next; temp = temp.prev)
            {
                if (noOfStepsTomove > count)
                {
                    if (temp.status.Equals(Pagestatus.clean))
                    {
                        //Remove node from here
                        checkpoint = true;
                    }
                }
                else
                    break;
                count++;
                }
            }
            return checkpoint;
        }


        //get least recent used block
        public CflruNode GetLeastUsedCleanNode(int length)
        {
            CflruNode newnode = new CflruNode();
            var count = 0;
            for (CflruNode temp = lastnode; temp != firstnode.prev; temp = temp.prev)
            {
                if (length > count)
                {
                    if (temp.status.Equals(Pagestatus.clean))
                    {
                        newnode = temp;
                        break;
                    }
                    count++;
                }
                else
                    break;
            }
            return newnode;
        }

        /// <summary>
        /// Moving the Current page to the first position of the list
        /// </summary>
        /// <param name="node"></param>
        public void MoveAtFirsPos(CflruNode node)
        {
            if (node.prev == null)
            {
                //Do nothing since the node itself is first node
            }

            else if (node.next == null)
            {
                node.prev.next = null;
                lastnode = node.prev;
                node.next = firstnode;
                node.prev = null;
                firstnode.prev = node;
                firstnode = node;

            }
            else
            {
                node.prev.next = node.next;
                node.next.prev = node.prev;
                node.next = firstnode;
                node.prev = null;
                firstnode.prev = node;
                firstnode = node;
            }
```

```
    }

    /// To delete the node tempom the list
    /// <param name="delenode"></param>
    public void Deletenode(CflruNode delenode)
    {
        for (CflruNode temp = firstnode; temp != lastnode.next; temp = temp.next)
        {
            if (temp.Data.Equals(delenode.Data))
            {
                if (temp.next == null)
                {
                    lastnode = lastnode.prev;
                    lastnode.next = null;
                    break;
                    //temp.prev = null;
                }
                else if (temp.prev == null)
                {
                    firstnode = firstnode.next;
                    firstnode.prev = null;
                    break;
                    //temp.next = null;
                }
                else
                {
                    temp.next.prev = temp.prev;
                    temp.prev.next = temp.next;
                    //temp.prev = null;
                    //temp.next = null;
                    break;
                }
            }
        }
    }

    public void RemoveLastNode()
    {
        lastnode = lastnode.prev;
        lastnode.next = null;
    }
    // To show the Current status of the all nodes in the list
            public void showstatus()
    {
        if (IsEmpty())
        {
            Console.WriteLine("Empty " + name);
        }

        else
        {
            Console.WriteLine("\t");
        }
    }
}
//Main class starts from here
class LFLRU : CflruNode
{
        public static void Main()
        {
            double mediumValue; //var will determine the range of MRU and LRU.
            int totalsize, i;//hold the user inputed cache size
            int np = 0, pageCount = 0, pagehit = 0, pagefault = 0, distinctpages = 0,
wrcount = 0;
            Console.WriteLine("Algorithm: CLFLRU");
            Console.WriteLine("Cache size:");
            totalsize = int.Parse(Console.ReadLine());//reads user inputed cache size
            mediumValue = totalsize / 2;
            //check whole and double number, if it is whole number then do nothing
otherwise it makes whole number with ceiling function.
            if (Math.Floor(mediumValue) != mediumValue)
            {
                mediumValue = Math.Ceiling(mediumValue);
            }
            CflruNode node;// temporary node

            using (StreamReader r = new StreamReader("D:\\data-random 100k references-50k
pages.txt")) // give file directory here, reads file based on this directory
            {
                cache newlist = new cache();
                string refpage;
                List<string> pageList = new List<string>();

                while ((refpage = r.ReadLine()) != null || (refpage = r.ReadLine()) !=
"") // reads entire lines included within file
                {
                    #region
                    if (refpage == null)
                        break;
                    var distinguishedpageNPageStatus = refpage.Split(',');//splits input
string by , and convert this into array
                    object data = distinguishedpageNPageStatus[1];

                    if (!pageList.Any(x => x.Contains(data.ToString())))
                    {
                        pageList.Add(data.ToString());
                        distinctpages++;

                    }
                    Console.WriteLine("Page:" + data);
                    if (distinguishedpageNPageStatus[0].Trim() == "0")
                        node = new CflruNode(data, Pagestatus.clean); // creates new
instance
                    else
                        node = new CflruNode(data, Pagestatus.dirty);
                    pageCount++;
                    //if page is exist in the list
                    if (newlist.search(node))
                    {
                        pagehit++;
                        //Find that particular node in the list
                        CflruNode CurrentPage = newlist.findnode(node);

                        if (node.status == Pagestatus.dirty && CurrentPage.status ==
Pagestatus.clean)
                            CurrentPage.status = Pagestatus.dirty;
                        //Move  Current page at the begining of the list
                        newlist.MoveAtFirsPos(CurrentPage);
                    }
                    //page in not in the list
                    else
                    {
                        pagefault++;
                        np++;
                        //admit the newly acccessed block to the MRU list till there is
no more free slot
                        if (np <= totalsize)
                        {
                            newlist.InsertAtFront(node);
                            //    newlist.showstatus();
                        }
                        //promote the newly accessed
                        else
                        {
                            int noOfStepsTomove = totalsize -
Convert.ToInt32(mediumValue);
                            if (newlist.CheckCleanPage(noOfStepsTomove))
                            {
                                //get least used clean page
                                var cleanBlock =
newlist.GetLeastUsedCleanNode(noOfStepsTomove);
                                //Remove clean node
                                newlist.Deletenode(cleanBlock);
                                //Add new at the front
                                newlist.InsertAtFront(node);
                            }
                            else
                            {
                                wrcount++;
                                //Remove Last node from list
                                newlist.RemoveLastNode();
                                //Add new one at the front
                                newlist.InsertAtFront(node);
                            }
                        }
                    }
                    #endregion

                }
            }

            Console.WriteLine("Total Number of Pages:" + pageCount);
            Console.WriteLine("Total Number of distinct Pages:" + distinctpages);
            Console.WriteLine("Totol Number Of Page fault:" + pagefault);
            Console.WriteLine("Totol Number Of Page Hit:" + pagehit);
            Console.WriteLine("Totol Number Of Write Count:" + wrcount);
            Console.ReadLine();
        }


    }

}
```

Appendix 6: Source Code of Moidifed CFLRU Algorithm

```
Modified CFLRU Algorithm:
namespace CFLRU_DirtyHot_CleanHot
{
    class CflruDirtyHotCleanHot : CflruOperationBase
    {
        public CflruDirtyHotCleanHot(int posTomove)
            : base(posTomove)
        {
        }

        public override void MovePageToMruRegion(ref LinkedList<CflruNode> cache)
        {
            var lastPage = cache.LastOrDefault();
            if (lastPage != null && ((lastPage.Status == PageStatus.Dirty &&
lastPage.Flag == PageFlage.Hot) || (lastPage.Status == PageStatus.Clean && lastPage.Flag
== PageFlage.Hot)))
            {
                lastPage.Flag = PageFlage.Cold;
                var nodelist = cache.Find(cache.ElementAt(PosTomove));
                cache.RemoveLast();
                if (nodelist != null) cache.AddBefore(nodelist, lastPage);
                MovePageToMruRegion(ref cache);
            }
        }
    }
}

namespace CFLRU_DirtyHot
{
    //node structure
    public class CflruNode
    {

        public CflruNode Prev { get; set; }
        public string Data { get; set; }
        public CflruNode Next { get; set; }
        public PageStatus Status;
        public PageFlage Flag;
        public CflruNode(string data, PageStatus pstatus)
        {
            Status = pstatus;
            Data = data;
            Flag = PageFlage.Cold;
            Prev = null;
            Next = null;
        }
    }

    public enum PageStatus
    {
        Dirty,
        Clean
    };
    public enum PageFlage
    {
        Hot,
        Cold
    };
}

namespace CFLRU_DirtyHot_CleanHot
{
    public class Algorithm : CflruBase
    {
        public static void Main()
        {

            Console.WriteLine("Algorithm: CFLRU Dirty Hot Clean Hot");
            Console.WriteLine("Cache size:");
            var totalsize = int.Parse(Console.ReadLine());
            var mediumValue = CalculateMediumValue(totalsize);
            var cache = new LinkedList<CflruNode>();

            using (var r = new StreamReader("D:\\data-random 100k references-50k
pages.txt"))
            {
                string refpage;
                while ((refpage = r.ReadLine()) != null) // reads entire lines included
within file
                //foreach (var refpage in InputString)
                {
                    var node = SetNode(refpage);
                    PageCount++;
                    if (cache.Any(x => x.Data == node.Data))
                    {
                        Pagehit++;
                        var currentPage = cache.FirstOrDefault(x => x.Data == node.Data);
//Find that particular node in the list
                        cache.Remove(currentPage);

                        if (node.Status == PageStatus.Dirty && currentPage.Status ==
PageStatus.Clean)
                            node.Status = PageStatus.Dirty;
                        node.Flag = PageFlage.Hot;

                        cache.AddFirst(node);//Move Current page at the begining of the
list
                    }
                    else //page in not in the list
                    {
                        Np++;
                        Pagefault++;
                        if (Np <= totalsize)
                        {
                            cache.AddFirst(node); //admit the newly acccessed block to
the MRU list till there is no more free slot
                        }
                        else //promote the newly accessed
                        {
                            var lastPage = cache.LastOrDefault();
                            if (lastPage != null && ((lastPage.Status == PageStatus.Dirty
&& lastPage.Flag == PageFlage.Hot) || (lastPage.Status == PageStatus.Clean &&
lastPage.Flag == PageFlage.Hot)))
                            {
                                var opw = new
CflruDirtyHot(Convert.ToInt32(mediumValue));
                                opw.MovePageToMruRegion(ref cache);
                                cache.RemoveLast();
                                cache.AddFirst(node);
                            }
                            else
                            {
                                if (cache.Last().Status == PageStatus.Dirty)
                                {
                                    WriteCount++;
                                }
                                cache.RemoveLast();  //Remove Last node from list
                                cache.AddFirst(node); //Add new one at the front
                            }
                        }
                    }
                }
            }
            ShowResult();
            Console.ReadLine();
        }
    }
}

public abstract class CflruOperationBase
    {
        protected readonly int PosTomove;

        protected CflruOperationBase(int posTomove)
        {
            PosTomove = posTomove;
        }
        public abstract void MovePageToMruRegion(ref LinkedList<CflruNode> cache);


    }

public class CflruBase
    {
        public static int Np = 0, PageCount = 0, Pagehit = 0, Pagefault = 0, WriteCount =
0;
        private static int _distinctpages = 0;
        //public static string[] InputString = { "0,3", "1,1", "0,4", "0,2", "1,5",
"0,2", "1,1", "0,9", "1,3", "0,6", "1,12", "0,11", "0,10", "1,13", "0,14", "1,8" };
        static readonly List<string> List = new List<string>();
        public static double CalculateMediumValue(int totalSize)
        {
            double mediumValue = totalSize / 2;
            if (!Math.Floor(mediumValue).Equals(mediumValue))//check whole and double
number, if it is whole number then do nothing otherwise it makes whole number with
ceiling function.
            {
                mediumValue = Math.Ceiling(mediumValue);
            }
            return mediumValue;
        }
        public static CflruNode SetNode(string refpege)
        {
            var distinguishedpageNPageStatus = refpege.Split(',');//splits input string
by , and convert this into array
            var data = distinguishedpageNPageStatus[1];
            Console.WriteLine(data);
            DistinctPageCounter(data);
            return distinguishedpageNPageStatus[0].Trim() == "0" ? new CflruNode(data,
PageStatus.Clean) : new CflruNode(data, PageStatus.Dirty);// temporary node
        }

        private static void DistinctPageCounter(string page)
        {
            if (List.Any(x => x.Contains(page))) return;
            List.Add(page);
            _distinctpages++;
        }
        public static void ShowResult()
        {
            Console.WriteLine("Total Number of Pages:" + PageCount);
            Console.WriteLine("Total Number of distinct Pages:" + _distinctpages);
            Console.WriteLine("Total Number Of Page fault:" + Pagefault);
            Console.WriteLine("Total Number Of Page Hit:" + Pagehit);
            Console.WriteLine("Write Count:" + WriteCount);
        }
    }
```