CHAPTER ONE

BADKGROUND AND PROBLEM FORMULATION

1.1 Background

In recent days, there is rapid growth in memory size but access time is comparatively slow. To minimize the gap between CPU speed and memory speed, the concept of caching arises. The cache is a small amount of very fast associative memory. Caching is a technique in which a fast memory is used from which access time is very fast as compared from secondary storage. To locate a memory address within a cache, a mapping function is used. There are three types of mapping functions- direct mapping, fully associative and set associative [9]. If each block has only one place it can appear in the cache, the cache is said to be direct mapped. If a block can be placed anywhere in the cache, the cache is said to be fully associative. And, if a block can be placed in a restricted set of places in the cache, the cache is set associative. A cache is set of group of blocks in the cache. A block is first mapped onto a set, and then the block can be placed anywhere within that set.

At a time computers running multiple processes, each with its own address space. So, it would be too expensive to dedicate a full address space worth of memory for each process. Hence, there must be a means of sharing a smaller amount of physical memory among many processes. To do this, Fotheringham [15] devised virtual memory which divides physical memory into small blocks and allocates them to different processes. Handling the virtual memory is one of the challenging tasks of memory management.

1.1.1 Virtual Memory

Virtual memory works silently and automatically without any intervention from the application programmer. The virtual memory is a method which automatically manages the two levels of the memory hierarchy represented by main memory and secondary storage. It moves required program into physical main memory for their execution and part of it not currently being execution are stored in secondary storage. Virtual memory makes the task of programming much easier because the programmer no larger needs to worry about the amount of physical memory available. The main memory is efficiently managed by virtual memory with treating it as cache keeping only the active data or program in main memory, and transferring data back and forth between disk and main memory as required during program execution. Virtual memory provides uniform address

space to each process for memory management and protects the address space of each process from corruption by other processes. Due to different advantages and efficient technique, virtual memory is a popular and widely used memory management technique in the computer systems.

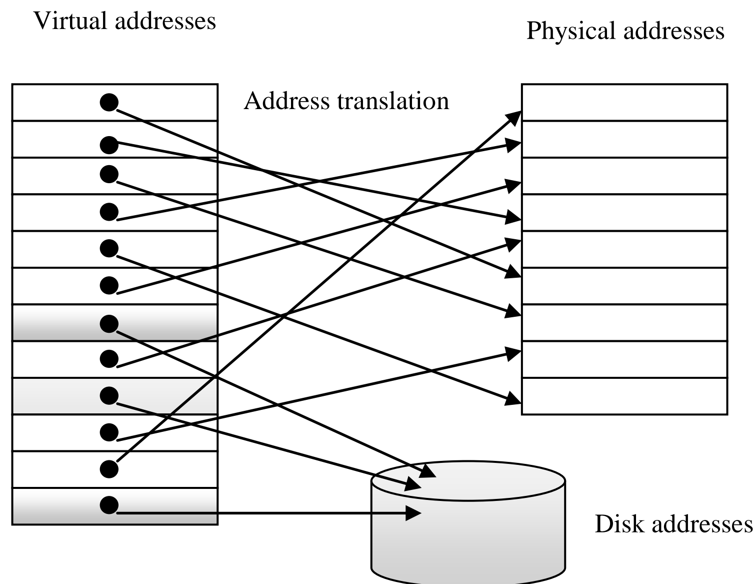


Figure 1.1 Virtual memory management

1.1.2 Memory Hierarchy

The performance of the computer system largely depends upon the memory. So, the memory is a major component of the computer system. There are varieties of memory devices are available in market which vary on their response time, cost, reliability, capacity etc. Memory Hierarchy is the ranking of memory devices so as to achieve higher performance with in the limited storage capacity. The computer architecture uses the term memory hierarchy when discussing performance issues in computer architectural design, algorithm predictions, and the lower level programming constructs such involving locality of reference. Memory hierarchy consists of different levels according to speed and cost.

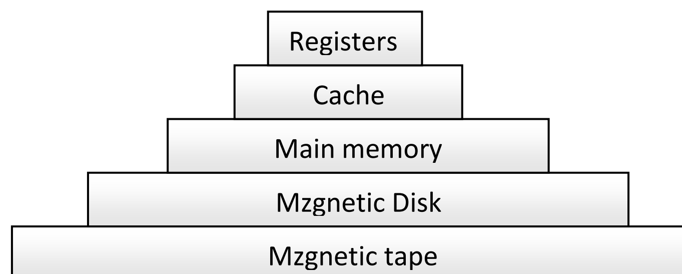


Figure 1.2 General memory hierarchy.

Figure 1.2 shows the arrangement of memory devices of computer system in which faster memory is at the top level and that of slower memory is at the bottom level. However, overall performance of computer system depends upon management and organization of such memories. The memory management is done by OS according to different policies followed by it. Besides real memory OS uses virtual memory to speed up the overall performance of the computer system. Different types of memory available are divided into two groups-primary memory and secondary memory. The main memory is a volatile memory usually too small and stores needed data and program temporally. The secondary storage is non-volatile memory considered as an extension of main memory and can store large quantities of data permanently.

1.1.3 Page Table Structure

A virtual address is translated to corresponding physical address before the memory can be used. This address translation is made in every memory reference and done by special hardware called Memory Management Unit (MMU). Virtual memory system maps virtual addresses onto physical addresses with the help of page table. Using this mapping information MMU performs translation. If the given virtual address is not mapped to the main memory, MMU traps the operating system. This trap is called page fault and gives the operating system an opportunity to bring the desired page from the secondary storage and page table is updated. A typical page table entry includes most important field called page frame number which is used as an index to find the entry for that virtual page. Page table contains Present/absent bit. When this bit is 1, then the entry is valid otherwise (i.e.0) invalid and required page is not in memory and can't be used. Page table includes modified and referenced bits that keep track of page usage.

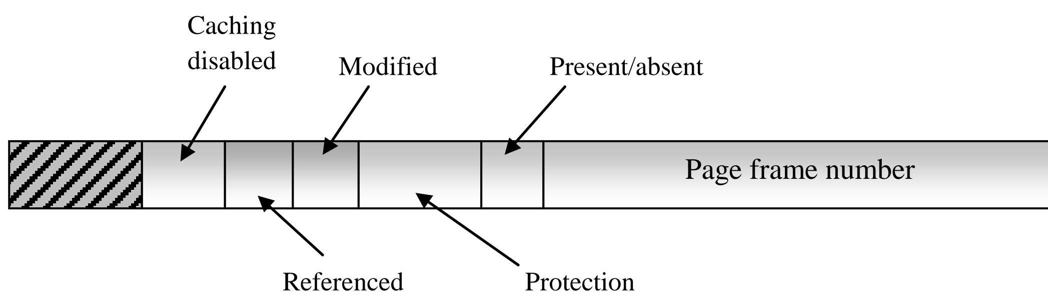


Figure 1.3 A typical page table structure [1].

1.1.4 Paging

Paging is the process of memory management in which memory is allocated in the non contiguous form, i.e. the program is break into block of fixed size known as page and also the main memory also break down into block of same size known as frame or page frame. In paging a computer can store and retrieve data from secondary storage for use in main memory and avoids the considerable problem of fitting memory chunks of varying sizes onto the backing store. Each page fits within page frame because generally page and frame size are equal. Initially, program is executed after loading only a few pages in memory. During program execution, if the references page is not currently in main memory then OS creates trap is called page fault. To increase the degree of multiprogramming the program or data that is not currently in main memory is required to be brought into main memory as needed and, some other page should be removed from memory to allocate the space for incoming page.

1.1.5 Demand Paging and Prepaging

In demand paging only those pages are brought into main memory which is required during program execution. At that time, when a program needs other pages it will swap out the unused pages from the main memory and swap in the desired page. Thus, it is possible to execute the program though the space available is not sufficient to bring the whole program into main memory. One of the problem arises due to demand paging is page fault which is caused due to required page not found in main memory for which we require swapping.

In prepaging strategy, only needed pages are brought once in the main memory before program execution then, program is executed. With pre-paging, pages other than the one demanded by a page fault are brought in. Prepaging attempts to guess which pages are needed next by placing them to main memory before they are needed. In general cases, it is very difficult to make accurate guesses of page usage and demand paging is generally accepted as a better choice. The selection of such pages is done based on common access patterns, especially for secondary memory devices.

1.1.6 Page fault Handling

As mentioned earlier, typically page fault occurs when a process references a page that is not present in the main memory. This page faults are triggered by the CUP and handled

by the page fault handler. The procedure for handling this page fault is straightforward which is shown in figure 1.4.

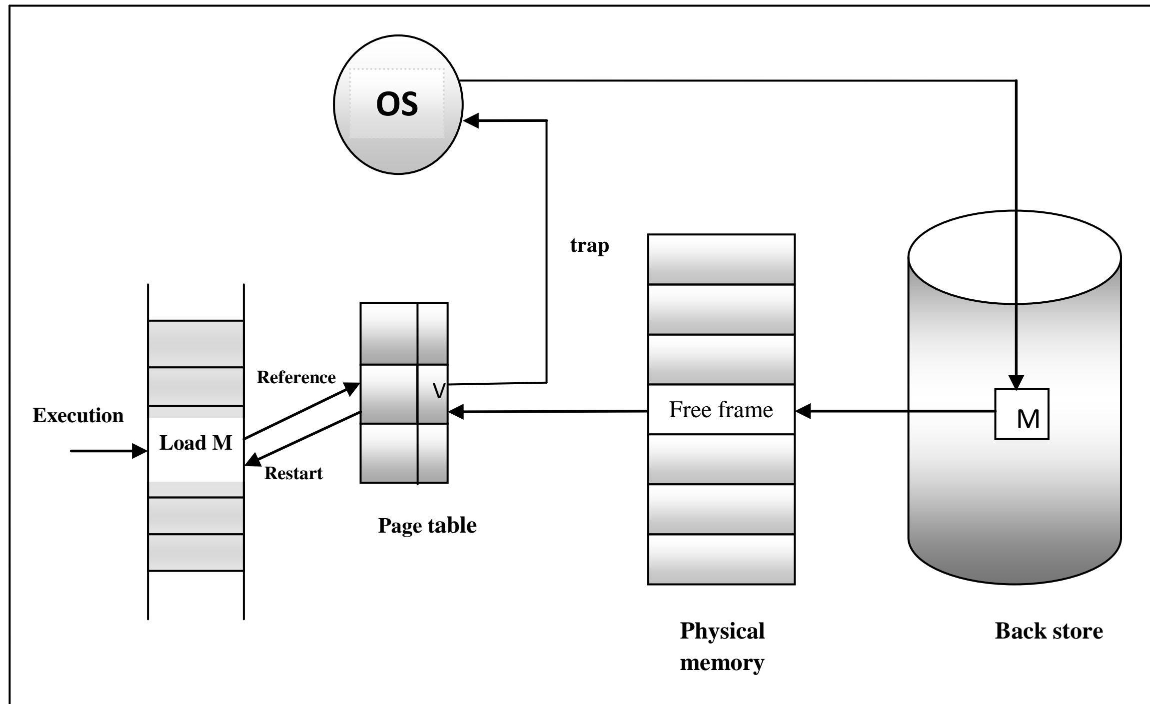


Figure 1.4 Page fault handling processes.

In page fault handling process, page fault handler checks the page table to determine whether the reference of the requested page is valid or invalid. If the reference is valid then the requested page is loaded. If reference bit is invalid then MMU creates trap to the OS then the missing page is swap in to main memory from the backing store and page table entry for referenced page is updated to valid. Now, OS restarts the instruction to execute properly. In the case when memory is full and the reference page has invalid bit in the page table then OS decides which page is to be removed according to the page replacement algorithms. If the page to be removed is modified while in memory, it must be rewritten to the disk so that the disk copy remains up to date. If, however, the page has not been changed (e.g., it contains program text), the disk copy is already up to date, so no rewrite is needed.

1.1.7 Page Replacement algorithms

The page replacement is the process of choosing a page frame to replace when a page fault occurs and, the page frame chosen for the replacement is called victim frame. The algorithm used by operating system to find victim frame is called page replacement algorithm. The most operating system has to choose a page frame to remove from main memory to make room for page that has to be brought in from secondary storage when

page fault occurs. The page to be read in just overwrites the page being evicted. While it would be possible to pick a random page to evict at each page fault, system performance is much better if a page that is not heavily used is chosen. If a heavily used page is removed, it will probably have to be brought back in quickly, resulting in extra overhead. Much work has been done on the subject of page replacement algorithms, both theoretical and experimental [8].

The area of page replacement policy is the probably the most studied area of memory management. When all of the frames in main memory are occupied and it is necessary to bring in a new page to satisfy a page fault, the page replacement algorithm determines which page currently in main memory to be replaced. All of the page replacement algorithms have as their objective that the page that is removed should be the page least likely to be referenced in the near future. Because of the principle of locality, there is often a high correlation between recent referencing history and near future referencing pattern. Thus, most page replacement algorithms try to predict future behavior on the basic of past behavior. One tradeoff that must be considered is that the more elaborate and sophisticated the replacement policy, the grater the hardware and software overhead to implement it. The page replacement policy is also used in a Web server to improve the performance of web service to employ web caching mechanism. The web caching caches popular heavily used objects at location close to the clients, so it is considered one of the effective solutions to avoid web service bottleneck, reduce traffic over the internet and improve scalability of the web system [12].

A page replacement algorithm can be either global or local. Local replacement means that replacement candidates are searched only from pages that belong to the page faulted process. This effectively means that, if the working set of the process does not fit to the memory reserved for it, the process will page fault constantly. This generally leads to poor usage of memory resources. In global page replacement all processes compete for the main memory and the replacement algorithm automatically adapts to changing working set sizes of running processes [8].

Static page replacement algorithms share frames equally among processes. It splits m frames to n users such that each user gets m/n frames. For example, if we have 100 frames and 5 processes then each process will get 20 frames. But, some applications require more frames than others. Compare a database program of 127k and a small

student process of 10k. One solution to this problem is to decide the number of frames at initial load time according to program size, or priority.

Dynamic paging algorithms share frames according to needs rather than equally. Some processes need more frames than others and sometimes a process needs more frames than other times. Change of locality when change of function. Some localities require more pages and Change of localities requires more pages. These issues can be addressed with dynamic page replacement algorithm. Although it is apparent that Dynamic Algorithms are more versatile in their ability to deal with locality changes and the natural occurrence of working page set changes, their complexity makes them a reality only for large-scale systems.

One would naturally expect the behavior of static paging algorithms to be linear; after all, they are static in nature. Instinct tells us that by increasing the available physical memory for storing pages, and thus decreasing the needed number of page replacements, that the performance of the algorithm would increase. However, with most simple algorithms this is not necessarily the case. In fact, by increasing the available physical memory, some algorithms such as FIFO can decrease in page fault performance seemingly at random, as evidenced by [15]. This occurrence is referred to as Belady's Anomaly.

1.1.8 Performance Metrics

An offline performance of the algorithm can be determined using page fault count, hit rate, miss rate etc. Page hit occurs when the requested block is available in memory. If it doesn't occur in memory during program execution then, page fault occurs. Generally, increase in memory size reflects increase in hit rate. Higher hit rate leads better performance of the algorithm than higher miss rate.

1.1.7.1 Page Fault Count

Page fault count is also known as page fault frequency (PFF) which is the number of occurrence of page faults between some intervals of reference. A better page replacement algorithm has always less number of page fault frequency.

1.1.7.2 Hit Rate & Hit Ratio

Hit rate is the percentage calculation of hit ratio and calculated by using formula:

$$HR = 100 - MR$$

Where HR is the hit rate and MR is the miss rate. Hit ratio is calculated by dividing total number of hits by total references and further by subtracting miss ratio from 1.

1.1.7.3 Miss Rate & Miss Ratio

Miss rate (MR) is the percentage calculation of miss ratio and calculated by using formula:

$$MR = 100 \times ((NPF - NDP) / (TNPR - NDP))$$

Where, NPF is the number of page faults, NDP is the number of distinct pages referenced and TNPR is the total number of pages referenced [3]. Miss Ratio is calculated by dividing total number of page fault by total references ignoring the distinct references.

1.1.9 Program Behavior

The performance of page replacement algorithm depends on several factors. The page replacement algorithm shows different behaviors according to different memory reference patterns as well as locality of reference and working set.

1.1.9.1 Memory Reference Pattern

The operating system accesses different memory locations with different reference manner within time is called memory reference pattern. There are different types of memory reference patterns that influence on the performance of the page replacement algorithms. Some of the memory reference patterns are: cyclic pattern, correlated access pattern, probabilistic pattern, temporally clustered patterns which are described below.

1.1.9.1.1 Cyclic Access Pattern

In cyclic patterns (references), all blocks are accessed repeatedly with a regular interval or period such as looping. For example, let p1, p2, p3, p4 be the memory pages used then, cyclic pattern is of the form p1, p2, p3, p4, p1, p2, p3, p4, p1, p2, p3, p4, p1, p2, p3, p4. Here loop executes five times.

1.1.9.1.2 Correlated Pattern

The memory reference pattern is said to be correlated if frequently accessed memory locations at particular place are repeated after some time duration. Sequential scan can be taken as an example of correlated pattern. For example, let p1, p2, p3 are the memory pages frequently accessed then, correlated pattern is of the form p1, p2, p3, p4, p5, p6, p7, p8, p1, p2, p3, p10, p11, p12, p13, p15, p1, p2, p3, p16, p17, p18,…so on.

1.1.9.1.3 Probabilistic Pattern

The memory reference pattern is said to be probabilistic if particular memory block has a stationary reference probability and all other blocks are accessed independently with the associated probabilities [8]. For example, let p1, p2, p3 be the memory pages frequently accessed then, probabilistic pattern is of the form p1, p2, p4, p5, p6, p1, p7, p8, p2, p10, p11, p12, p13, p3, p7, p14, p1, p5, p3, p6, … so on.

1.1.9.1.4 Temporally Clustered Pattern

Temporally clustered patterns are such references in which blocks accessed more recently are the once more likely to be accessed in the near future. For example, temporally clustered pattern is of the form p1, p2, p1, p3, p2, p4, p3, p1, p2, p5, p6, p7, p8, p9, p10, ...so on.

1.1.9.2 Locality of Reference

In computer system, locality of reference, also known as principle of locality, is the phenomenon of the same data value or related storage locations being frequently accessed. Locality is merely one type of predictable behavior that is used by many page replacement algorithms to predict about future references. There are two basic types of reference locality. Temporal locality refers to the reuse of specific data within relatively small time durations. In temporal locality, 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 near future. The spatial locality is property of page reference pattern in which 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. Furthermore, locality of reference may strong and weak. In strong locality of reference, there is high probability of reference of memory location again in the near future. Locality of reference is said to be weak if there is low probability of same memory location to be referenced in near future. There are different types of algorithms that take locality of reference for choosing victim frame when page fault occurs. Some of them are for weak locality of reference and some of them are for strong locality of reference [15].

1.1.9.3 Working Set

The working set is the set of pages that are physically in memory at any one time and frequenty accessed by an applications or set of applications. Page fault will occur if a working set is not available in cache. The operating system constantly swaps pages out to

disk to make room to swap in pages that an application wants to access. If swapped out pages are in working set then, it will almost immediately be needed again and have to swap in from disk meaning that there is regular occurrence of page fault. This is referred to as thrashing. Thrashing leads slowing down the execution of a program because CPU spends more time to handle it. The page replacement algorithm is based on the working set and basic idea is to find a page that is not in working set and evict it [1].

1.2 Introduction

The memory management is a major concern in today's computer system. There are several page replacement algorithms with own advantages and disadvantages. Among them Least Recently Used (LRU) is a popular, simple, flexible and low overhead page replacement algorithm. LRU is a recency based algorithm in which replacement of victim frame is based on recency. That is, LRU replaces page that is not accessed for longest time. Recency is defined as the virtual time difference between the current time and last time when the oldest block is accessed. LRU is the best algorithm for good locality of reference but it shows worse performance with weak locality of reference. Reason behind it is that LRU makes bold assumption that a block that has not been accessed for the longest time would wait for the longest time to be used again. Here are some representative examples reported in the research literature, to illustrate how LRU poorly behaves:-

a. Under the LRU, a burst of references to infrequently used blocks such as "sequential scans" through a large file may cause replacement of commonly referenced blocks [17]. Suppose a memory consists of 51 page frames. Consider a program in which there is a regularly repeated access to 50 pages and once in every 25 references there is a reference to a new page. In such case LRU replaces the frequently used page frame.
b. For a cyclic (loop-like) pattern of accesses to a file that is only slightly larger than the cache size, LRU always mistakenly evicts the blocks that will be accessed for the longest time [6]. For example, let us consider memory consist of 4 page frames and consider a program which repeatedly references 5 pages in cyclic order. Let P1, P2, P3, P4, and P5 are the pages which are referenced repeatedly as: P1, P2, P3, P4, P5, P1, P2, P3, P4, P5, P1, P2, P3, P4, P5, ……so on. Here, when the program references the age P5, the native LRU replaces page P1, because it is

the oldest unused page in memory. However, this page (i.e.P1) will be used in the next reference. So this behavior of LRU is undesirable.

In above mentioned cases, if the "frequency" of each page reference is taken into consideration, it will perform better in the case where workload has weak locality. The Least Recently Used (LFU) uses the "frequency" for the prediction. This algorithm keeps track of the number of references to each page, and the page selected for replacement is the page that has the least number of references. In another word, this policy is based of the presumption that the page that has been more frequently referenced in the past is more likely to be referenced in the near future.

The LRU and LFU are two streams of independent algorithms. To overcome the merits and demerits of both algorithms, there are several page replacement algorithms have been developed. Least Recently Frequently Used (LRFU) page replacement algorithm combines both "recency" and "frequency" factors on the replacement decision [4].

1.2.1. Problem Statement

Generally, in most page replacement algorithms, the page fault rate decreases as the memory size increases. But in some algorithms just opposite i.e. increasing in memory size leads to increase in page fault rate. This unexpected result is known as anomaly. The LRFU page replacement algorithm also shows anomalous behaviors some times. But LRU and LFU algorithm never shows anomalous behavior. This study mainly focused to identify the reasons behind the anomalous behavior of LRFU and adopted the algorithm so that anomalous behavior could be avoided.

1.2.1. Objectives

The objectives of this research work are:

1. To analyze LRFU algorithm to identify causes of its anomalous behavior.
2. To list sample workloads causing anomalous behavior of LRFU algorithm.
3. To adopt LRFU page replacement algorithm so that anomalous behavior of the algorithm can be removed.

1.3 Motivation

The operating system manages the computer hardware. The memory management is a major concern to computer system and it is not only the burden of today's computing devices. It has been researched for decades. Whatever variety of storage devices found in

today's market is the great achievement of computer science. But still computer memory is the limited source which directly hampers the performance of computing system. Performance gain can be achieved by increasing the capacity of primary storage. Expectation of customer is to decrease cost price with sufficient working memory. It is not possible to gain performance without managing memory logically for its usability. Varieties of techniques had been tried for this achievement. Among such techniques paging is the successful one. Page replacement algorithm is the main part of paging technique because deciding the victim page is a very tough job. Optimal page replacement algorithm is the best one. But it can be only simulated since references should be known earlier, which is not possible in most of the real systems.

Many near-optimal replacement schemes have been found, but their complexity and various practical considerations tend to limit the effectiveness of the algorithms implemented in real systems. Implementing LRU is a successful idea due to its simplicity, flexibility and performance gain. But still LRU shows worse performance with weak locality workloads. It is better if an algorithm could work as LRU comparatively equivalent to computational complexity as well as it could solve the problem on weak locality workloads. For this locality of workloads LFU shows somewhat better than earlier LRU. LRFU algorithm combines both LRU and LFU to overcome merits and demerits which shows better performance than earlier both algorithms. But the performance of LRFU algorithm largely depends on the control parameter λ. An optimal value of the control parameter $\lambda=1$ that results optimal performance of LRFU. However, LRFU shows anomalous behavior sometimes according to different workloads [10].

1.4 Report Structure

The background and introduction part of this dissertation work have been mentioned in this chapter including concept of virtual memory, page replacement algorithm and other related basic terms. This chapter has also included problem definition and objectives of this dissertation work. The research methodology and literature review part of this dissertation work are described in chapter 2. Several page replacement algorithms such as LFU, LRU-k, FBR, LIRS, 2Q, ARC, SEQ, EELRU etc. have mentioned in literature review part of this chapter. Chapter 3 contains detail description and tracing of algorithm which have been studied and implemented during study.

Chapter 4 consists of program development steps of our simulation. It includes detail design of the program such as flowcharts. It also includes details about the data structures and programming language used to build the simulation. Chapter 5 consists of data collection and analysis part which includes output results with several analyzing graphs which are tested for probabilistic pattern, cyclic pattern, temporally clustered pattern and mixed pattern workloads. Chapter 6 consists of conclusion of this whole dissertation work and the future work which shows guidelines for further research.

CHAPTER 2

LITERATURE REVIEW AND METHODOLOGY

2.1. Research Methodology

Research is common parlance refers to a search for knowledge. It is an aggregate of activities carried out by using scientific methods in sequential order to describe an event or phenomenon or arrive or draw conclusion about the happening of an event or phenomenon on the basic of available information about a factor of elements involved in the events or phenomenon. The system of collecting data for research projects is known as research methodology. The validity of research data is a important factor in research methodology.

The topics memory management and design has been studied from the early generation of computer. Page replacement algorithm is one of the major strategies to manage memory efficiently. All data collected are primary data, which are traces of page references. This dissertation work is based on trace driven simulation. Output information gathered is analyzed in a quantitative approach. Finally conclusion is drawn with the help of analyzed data which is generalized form. This work takes different types of workloads such as cyclic pattern, probabilistic pattern, temporally clustered pattern and mixed pattern.

2.2. Literature Review

An optimal page replacement algorithm [13] is an ideal replacement algorithm that has the lowest page-fault rate and will never suffer from anomaly. An optimal page replacement algorithm replaces the page that will not be used for the longest period of time. Use of this page replacement algorithm guarantees the lowest possible page fault rate for a fixed number of frames. Unfortunately, this algorithm is difficult to implement, because it requires future knowledge of the reference string, so this algorithm can
not be realized in practice.

The management of virtual memory is a central issue in today's computer system. LRU is a most commonly used page replacement algorithm in various forms. It is used for the management of virtual memory, file buffer caches, and data buffers in database systems. Many enhancement and modifications have been made to overcome the problems of LRU. Different page replacement algorithms can be categorized and explained in the next section.

2.1.1. Replacement Algorithms Based on User-Level Hint

Application-controlled file caching and application-informed prefetching and caching [5] are the algorithm based on user-level hints. By taking user-level hints, applications are hinted during caching and pre-fetching which rely on users understanding of data access patterns. These algorithms identify frames less likely to be re accessed in the near future based on the hints provided by user programs. To provide appropriate hints, programmers need to understand the data access patterns, which add to the programming burden [18].

2.1.2. Replacement Algorithms using Deeper History Information

LRU uses the limited history information for the replacement so that it behaves worse performance than other simple algorithms. This is due to limited history information it uses. Therefore, to overcome limitations of LRU, several algorithms were proposed that use more "deeper" history information than LRU.

2.1.2.1 LFU Page Replacement Algorithm

Least Frequently Used (LFU) algorithm replaces the page that has been used least frequently. This algorithm is based on the presumption that the block that has been more frequently referenced in the past is more likely to be referenced in the near future. The motivation for this algorithm is that some pages are accessed more frequently than others so that the reference counts can be used as an estimate of the probability of a page being referenced. Instead of using a single recency factor as LRU, LFU defines additional information of frequency which is equal to number times the page used with each page. This frequency is calculated throughout the reference stream by maintaining counting information. Frequency count leads to serious problem after a long duration of reference stream. Because when the locality changes, reaction to such certain change will be extremely slow. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less than the pages associated with the previous program. Since the context has changed, the pages swapped out will most likely be needed again soon which leads to thrashing. One way to remedy this is to use a popular variant of LFU, which uses frequency counts of a page since it was last loaded rather than since the beginning of the page reference stream. Each time a page is loaded, its frequency counter is reset rather than being

allowed to increase indefinitely throughout the execution of the program. LFU still tends to respond slowly to change in locality [18].

2.1.2.2 LRU-k Page Replacement Algorithm

The LRU-K algorithm makes the decision about the victim frame based on the time of reference of kth-to-last reference to the block [6]. It ignores the recency of the k-1 references. For example, assume that {4, 9, 16, 18, 20} is the reference history of a block, then LRU-4 would treat its decision on the time of fourth-to-last reference of this block (i.e. 9). If this time is least over all blocks present in the memory, then the algorithm replaces this block. LRU-K algorithm violates the rule of thumb that the more recent behavior predicts the future better. For example, let reference histories of blocks a and b are {1, 3, 10, 20} and {2, 3, 5, 20} respectively. Then LRU-3 treats both a and b equally because their 3rd-to-last reference time is same (i.e.3). Intuitively, block b have greater possibility of being replaced. Since, the block a is references recently than block b (10 vs.5).

2.1.2.3 FBR Page Replacement Algorithm

Frequency Based Replacement (FBR) algorithm [11] is a hybrid replacement algorithm which combines both LRU and LFU algorithms in order to capture the benefits of both algorithms. It maintains the LRU ordering of all blocks in the memory, but the replacement decision is primarily based upon the frequency. It divides LRU list into three sections: new, middle, and old. For every page in memory, a counter is also maintained. On a page hit, the hit page is moved to the MRU position; moreover, if the hit page was in the middle or the old section, then its reference count is incremented. The key idea known as factoring out locality was that if the hit page was in the new section then the reference count is not incremented. On a page miss, the page in the old section with the smallest reference count is replaced. A limitation of the algorithm is that to prevent cache pollution due to stale pages with high reference count but no recent usage the algorithm must periodically resize (rescale) all the reference count. The algorithm also has several tunable parameters, namely, the size of all three sections, and some other parameters C_{max} and A_{max} that control periodic resizing.

2.1.2.4 2Q Page Replacement Algorithm

2Q replacement algorithm [19] is based on the modification that the cold pages would recently removed from the memory. It uses one FIFO queue Al_{in} and two LRU lists, Al_{out} and A_m. It places a block in Al_{in} on the first access and promotes the block to A_m on the second access. It replaces a block in Al_{in} and put the block's identifier in Al_{out} if Al_{in} has more than fixed number of blocks. Otherwise, it replaces a block in A_m.

2.1.2.5 LIRS Page Replacement Algorithm

Low Inter-Reference Recency Set (LIRS) algorithm [17] is another important replacement algorithm which is suitable for weak locality of references. It minimizes the deficiencies of LRU algorithm using Inter-Reference Recency (IRR) history instead of just access recency for making a replacement decision. IRR of a page is the number of other pages accessed between two consecutive references to the page. The algorithm assumes the existence of some behavior inertia and, according to the collected IRRs, replaces the page that will take more time to be referenced again. This means that LIRS does not replace the page that has not been referenced for the longest time, but uses the access recency information to predict which pages have more probability to be accessed in a near future. Pages with smaller IRR values are favored than those with larger IRR values.

2.1.2.6 ARC Page Replacement Algorithm

An Adaptive Replacement Cache (ARC) algorithm [14] combines recency and frequency by using two lists L1 and L2. The first list contains pages that have been seen once recently, while other list contains pages that have been seen at least twice recently. Therefore, list L1 is thought of as "recency" and L2 as "frequency". For the replacement, replace the LRU page in L1, if $|L1|=c$; otherwise, replace the LRU page in L2. The basic idea is to divide L1 into top T1 and bottom B1 and to divide L2 into top T2 and bottom B2. The pages in T1 and T2 are in the cache and in the cache directory, while the history pages in B1 and B2 are in the cache but not in the cache. The pages evicted from T1 (resp. T2) are put on the history list B1 (resp.B2). The algorithm sets a target size p for the list T1. During replacement, replace the LRU page in T1, if $|T1|\geq p$; otherwise replace the LRU page in T2. ARC algorithm dynamically adjusts their sizes depending which factor is more important.

2.1.3. Replacement Algorithms Based on Detection and Adaptation of Access Regularities

Algorithms that are adapted themselves by carefully observing the page reference pattern at run time can gain better performance than former.

2.1.3.1 SEQ Page Replacement Algorithm

An SEQ replacement algorithm [7] is an adaptive version of the LRU algorithm. It shows better performance in the case of linearly sequential memory accesses, than the original LRU algorithm. An SEQ replacement algorithm detects long sequences of page faults and applies MRU replacement to such sequences. The goal is to avoid LRU flooding, which occurs when a program accesses a large address space range sequentially. If a program accesses an address range once, LRU would page out useful pages that would be accessed again; if the range is larger than physical memory, LRU would page out the pages in the order in which they are accessed and thus perform poorly. If no sequences are detected, SEQ performs LRU replacement.

2.1.3.2 EELRU Page Replacement Algorithm

Early Eviction LRU (EELRU) is a simple adaptive page replacement algorithm. LRU exhibits worse performance for regular access patterns over more pages than the main memory can hold (e.g., large looping). To eliminate this problem of LRU, EELRU algorithm is arises which uses only the kind of information needed by LRU-how recently each page has been touched relative to the others [20]. EELRU is not affected by high-frequency behavior (e.g., loops much smaller than the memory size), chooses pages to evict in a way that respects both the memory size and aggregate memory-referencing behavior of the program. Moreover, by an aggregate analysis, EELRU cannot quickly respond to the changing access patterns. EELRU performs LRU replacement by default but diverges from LRU and evicts pages early when it notices that too many pages are being touched in a roughly cyclic patterns can be reliably detected using recency information. EELRU maintains some kind of information for reference pages as resident and nonresident pages.

2.1.4. AI Based Page Replacement Algorithms

Recent adaptive algorithms use Artificial Intelligence techniques in order to help them in the adaptation. For example the FPR [2] and FAPR [16] algorithms apply fuzzy inference techniques to manage the replacement priorities of the resident pages. All these algorithms bring important conceptual benefit to the traditional page replacement algorithms, but they also present more complex implementations. In many cases additional data structures to hold nonresident pages are necessary increasing space requirements. Some algorithms require data update in every memory access, making impracticable its real implementation.

CHAPTER 3

PAGE REPLACEMENT ALGORITHMS STUDIED

3.1. Least Recently Frequently Used (LRFU) Algorithm

LFU and LRU algorithms that consider either frequency or recency but LRFU algorithm takes into account both the frequency and recency of references in its replacement decision. LRFU algorithm associates a value with each page frame. This value is called the CRF (Combined Recency and Frequency) value and quantifies the likelihood that the page would be referenced in the near future. Each reference to a block in the past contributes to CRF value and a reference's contribution is determined by a weighting function F(x), where x is the time span from the reference in the past to the current time. In general, F(x) should be decreasing function to give more weight to more recent references and, therefore, a reference's contribution to the CRF value is proportional to the recency of the reference. Weighting function that subsumes both LRU and LFU is given by:

$$F(x)=(1/p)^{\lambda x}$$

Where, x is difference between current time and the time of reference in the past, p>=2, and λ is a control parameter which ranges from 0 to 1 [4]. The performance of the LRFU algorithm depends on the parameter λ [10]. LRFU algorithm replaces a block whose CRF value is minimum and CRF value of each page is calculated as follows:

Definition 1 Assume that the system time can be represented by an integer value by using a system clock and that at most one block may be referenced at any one time. The CRF value of a block b at time t_{base}, denoted by $C_{tbase}(b)$, is defined as [4]-

$$C_{tbase}(b)=\sum_{i=1}^{k} F(t_{base}-t_{bi})$$

Where F(x) is a weighting function and $\{t_{b1}, t_{b2}, \ldots, t_{bk}\}$ are reference times of blocks b and $t_{b1}<t_{b2}<\ldots<t_{bk}<=t_{base}$.

For example, assume that block b was referenced in the past at times 2, 4, 7, and 9, and that the current time (tc) is 10. Then, the CRF value of the block at current time tc is denoted by $C_{tbase}(b)$ and, is computed as:

$$C_{tbase}(b)=F(10\text{-}2)+F(10\text{-}4)+F(10\text{-}7)+F(10\text{-}9)$$

$$=F(8)+F(6)+F(3)+F(1)$$

Property 1: If F(x)=c for all x where c is a constant, then the LRFU algorithm replaces the same block as the LFU algorithm [4].

In case of LFU algorithm, the replacement decision is made on the basis of frequency of reference. It means that LFU algorithm does not discriminate a block in accordance to the time of reference (it only counts the frequency of reference). Therefore, CRF value of any block, if LRFU algorithm were managed to follow LFU algorithm, would be independent of time span, i.e., F(x) = c, where c is a constant, for all time span x. To be more precise, F(x) should give the reference count of a block. So, F(x) = 1, for all x.

Property 2: If F(x) satisfies the following condition, then the LRFU algorithm replaces the same block as the LRU algorithm [4].

$$\forall i,\ F(i) > \sum_{j=i+1}^{k} F(j) \text{ for any k where } k \geq i + 1$$

A weighting function $F(x)=(1/2)^{\lambda x}$ satisfies both property 1 and property 2 [4]. Spectrum generated by LRFU algorithm according to the function $F(x)= (1/2)^{\lambda x}$ is given below:

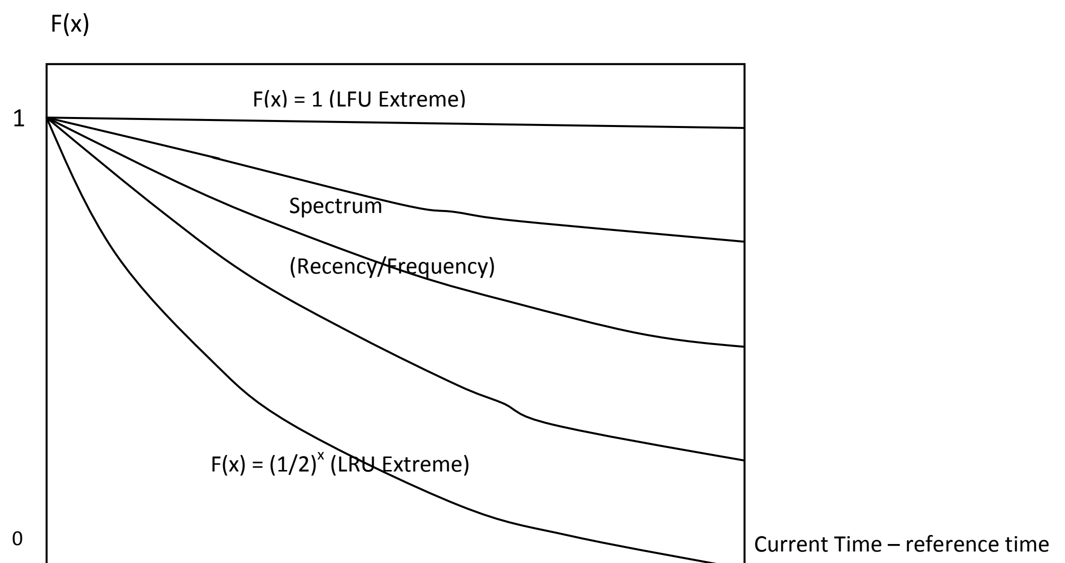


Figure 3.1 Spectrum of LRFU according to weighting function $F(x)= (1/2)^{\lambda x}$, where x=(current time-reference time).

In general, computing the CRF value of a block requires the reference times of all the previous references to the block. This obviously requires unbounded memory and, thus algorithm may not be implementable. This necessitates re-computing the CRF value of every block at each time step, makes the algorithm unimplementable. So, to reduce storage and computational overheads, two properties: F(x+y)=F(x)F(y) and F(x+y)=F(x)+F(y) have been identified [4]. Using this property, the CRF value at the time of the K'th reference can be computed using the time of the (K-1)'th reference and CRF value at that time as:

$$C_{tbk}(b)=F(0)+F(\delta)C_{tbk-1}(b)$$

Where $C_{tbk}(b)$ is CRF value of block b at the time of K'th reference, $C_{tbk-1}(b)$ is CRF value of block b at the time of (K-1)'th reference and $\delta = tc - t_{bk}$[4]. This implies that, at any time the CRF value can be computed using only two variables for each block.

3.1.1. Algorithm

1. If b is already in the buffer cache
    - 1.1. then
    - 1.2. $CRF_{last}(b)= F(0)+ CRF(b)$
    - 1.3. LAST (b) $=t_c$
    - 1.4. Restore (H,b)
2. Else
    - 2.1. Fetch the missed block from the disk
    - 2.2. $CRF_{last}(b)= F(0)$
    - 2.3. LAST (b) $=t_c$
    - 2.4. Victim = ReplaceRoot (H, b)
3. End if
4. End
5. Restore (H, b)
    - 5.1. If b is not a leaf node
        - 5.1.1. Then
        - 5.1.2. Let smaller be the child that has smaller CRF
        - 5.1.3. If CRF(b) > CRF (smaller)
            - 5.1.3.1. Then
            - 5.1.3.2. Swap (H, b, smaller)
            - 5.1.3.3. Restore (H, smaller)

5.1.4. End if

5.2. End if

6. End Restore
7. Replace root (H . b)

7.1. Victim=H . root

7.2. H . root =b

7.3. Restore (H , b)

7.4. Return victim

8. End Replace root
9. CRF (b)

9.1.Return $F(t_c - LAST(b)) * CRF_{last}(b)$

10. End CRF

3.1.2. LRFU Tracing

Suppose reference string and virtual time is represented by integer value are given in the following table as:

Time	0	1	2	3	4	5	6	7	8	9	10
Reference String	5	10	7	3	9	15	3	6	7	15	16

Assume that number of page frames in memory are 7 and weighting function $F(x)=(1/2)^{\lambda x}$, where $\lambda=1/8$. Each LRFU node (b) contains three fields: first field contains page number, middle field contains CRF value of that page, $CRF_{last}(b)$ and last field contains last access time of the page, LAST(b).

Step 1: At current time (tc)=0, page 5 is referenced which is not in memory (page fault).

CRF(5)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(5)=tc=0

5	1	0

Step 2: At current time(tc)=1, referenced page=10 which is not in memory (page fault).

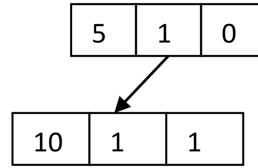
CRF(10)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(10)=tc=1



Step 3: At current time(tc)=2, referenced page=7 which is not in memory (i.e. page fault)

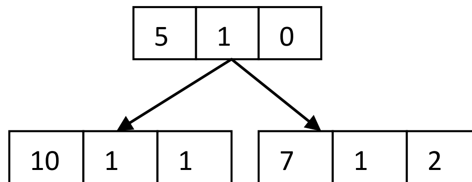
CRF(7)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(7)=tc=2



Step 4: At current time(tc)=3, referenced page=3 which is not in memory (i.e. page fault)

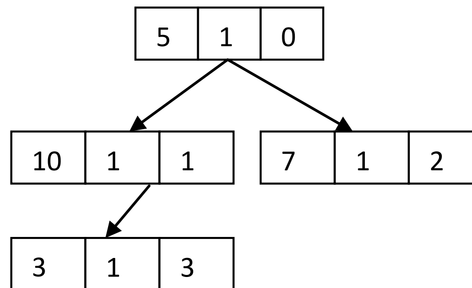
CRF(3)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(3)=tc=3



Step 5: At current time(tc)=4, referenced page=9 which is not in memory (i.e. page fault)

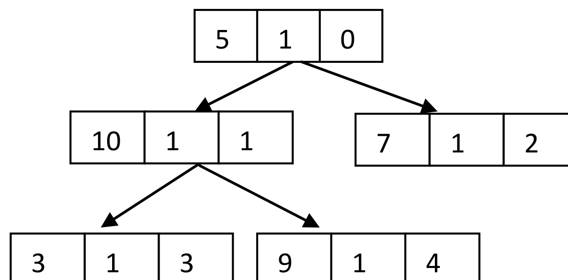
CRF(9)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(9)=tc=4



Step 6: At current time(tc)=5, referenced page=15 which is not in memory (i.e.page fault)

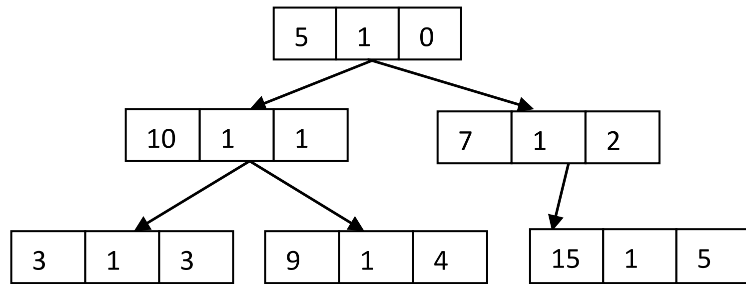
CRF(15)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

=1

LAST(15)=tc=5



Step 7: At current time(tc)=6, referenced page=3 which is already in the memory (i.e. page hit)

CRF (3)=F(0)+ CRF (3)

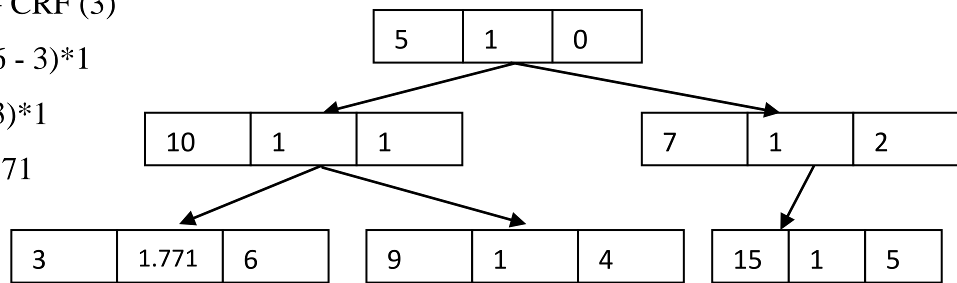
=1+ F(6 - 3)*1

=1+ F(3)*1

=1+0.771

=1.771

LAST(3)=tc=6



Step 8: At current time(tc)=7, referenced Page=6 which is not in memory (page fault)

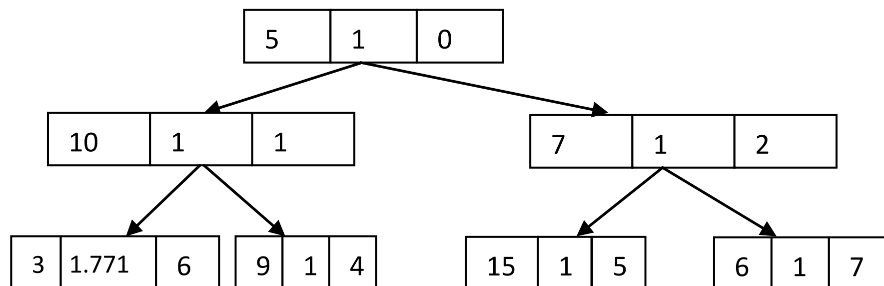
CRF(6)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

=1

LAST(6)=tc=0



Step 9: At current time(tc)=8, referenced page=12 which is not in memory(i.e.page fault).

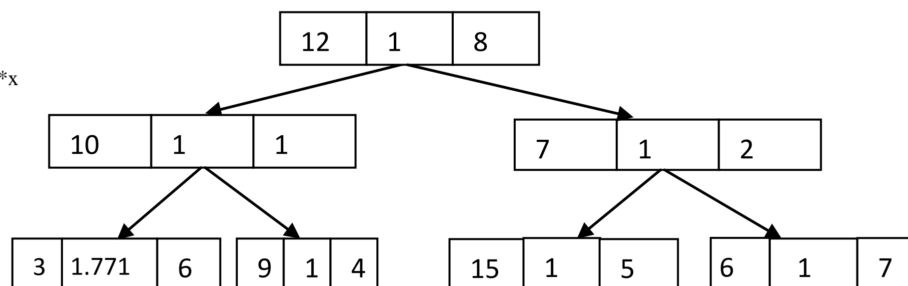
CRF (12)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

=1

LAST(12)=tc=8



Step 10: At current time(tc)=9, referenced page=15 which is already in the memory (i.e.page hit)

CRF (15)=F(0)+ CRF (15)

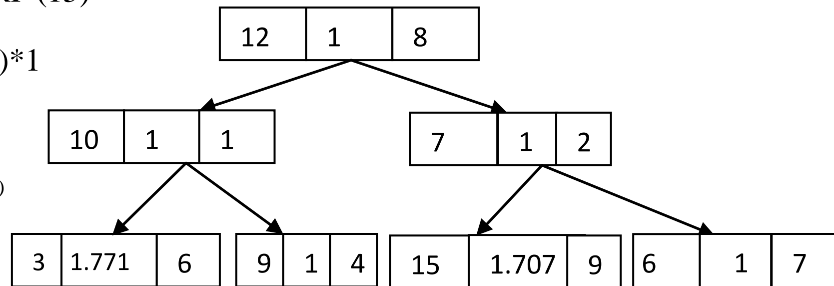
=1+ F(9 - 5)*1

=1+ F(4)

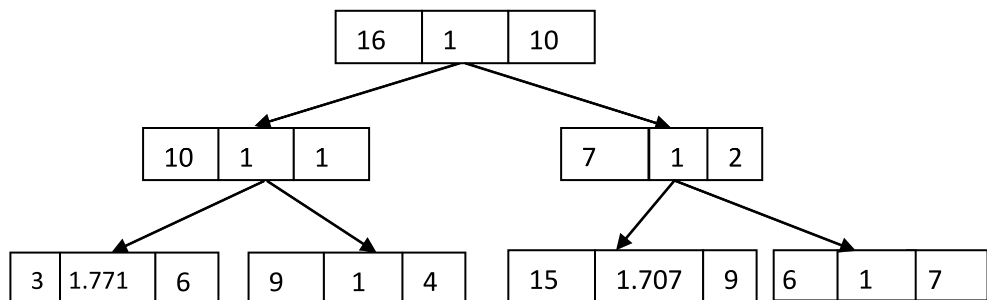
$=1+(1/2)^{(1/2)}$

=1.707

LAST(15)=tc=9



Step 11: At current time(tc)=10, referenced page=16 which is not in the memory. At this time memory is full and hence, a page should be removed from the memory to make room for the fetched page. For this, swap root node with last node and discard this last node from the heap. Then, insert new node at this position and perform restore operation on each of non- leaf node of the heap.



3.2. Adapted LRFU algorithm

FIFO algorithm replaces the page that has been referenced earlier i.e. the oldest page has been replaced when page fault occurs. This algorithm is easy to understand and implement but it shows anomalous behavior. Here, anomalous behavior is an unexpected result when increasing page frame number leads to increase in page fault rate. It is shown by study and analysis that LRFU algorithm also shows this type of anomalous behavior sometimes [12]. In LRFU algorithm each node associates a CRF value. A node with minimum CRF value at the root is removed and newly fetched block is placed at the root. This root node is swapped with last node and restore operation is done to obtain minimum

CRF value block at root. Thus, LRFU removes recently referenced block when CRF values of blocks are equal. Some adaptations have been made in LRFU algorithm so that such anomalous behavior can be avoided.

Adapted LRFU replaces the smallest CRF value with least recently used block. In Adapted LRFU, min-heap is made according to CRF value and Last reference time. Here, the root of min-heap contains a node with minimum last access time if CRF values of nodes are equal. Therefore, when page faults occurs the root of the min-heap is removed. This change has been made in Adapted LRFU algorithm and is the key point of my dissertation work.

3.2.1 Algorithm

1. If b is already in the buffer cache
    - 1.1. then
    - 1.2. $CRF_{last}(b)= F(0)+ CRF(b)$
    - 1.3. LAST (b) =t_c
    - 1.4. Restore (H.b)
2. Else
    - 2.1. Fetch the missed block from the disk
    - 2.2. $CRF_{last}(b)= F(0)$
    - 2.3. LAST (b) =t_c
    - 2.4. Victim = ReplaceRoot (H, b)
3. End if
4. End
5. Restore (H, b)
    - 5.1. If b is a leaf node then goto step 6
    - 5.2. If b is not a leaf node
        - 5.1.1. Set flag=0
        - 5.1.2. Let smaller be the child that has smaller CRF value
        - 5.1.3. If CRF(b) > CRF (smaller)
            - 5.1.3.1. flag=1 then goto 5.1.4
        - 5.1.4. If Flag=0
            - 5.1.4.1. Iet smaller be tha child that has equal CRF value to parent
            - 5.1.4.2. Check Last(b)>Last(smaller)

5.1.4.3. Goto 5.1.5

5.1.5. If(smaller!=i)

5.1.5.1. Swap (H, b, smaller)

5.1.5.2. Restore (H, smaller)

6. End Restore
7. Replace root (H , b)
   7.1. Victim=H . root
   7.2. H . root =b
   7.3. Restore (H , b)
   7.4. Return victim
8. End Replace root
9. CRF (b)
   9.1.Return $F(t_c - LAST(b)) * CRF_{last}(b)$
10. End CRF

3.2.2 Adapted LRFU Tracing

The tracing of adapted LRFU algorithm takes same reference string, page frame number, weighting function and value of control parameter λ as mentioned in previous tracing section 3.1.2.

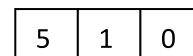
Step 1: At current time (tc)=0, page 5 is referenced which is not in memory (i.e. page fault).

CRF(5)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}=1$

LAST(5)=tc=0



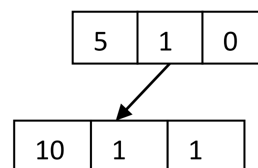
Step 2: At current time(tc)=1, referenced page=10 which is not in memory (i.e. page fault).

CRF(10)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}=1$

LAST(10)=tc=1



Step 3: At **c**urrent time(tc)=2, referenced page=7 which is not in memory (i.e. page fault).

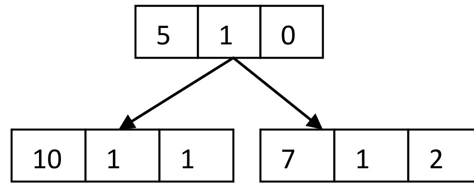
CRF(7)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(7)=tc=2



Step 4: At **c**urrent time(tc)=3, referenced page=3 which is not in memory (i.e. page fault).

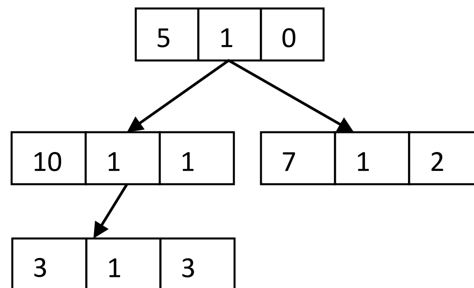
CRF(3)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(3)=tc=3



Step 5: At **c**urrent time(tc)=4, referenced page=9 which is not in memory (i.e. page fault).

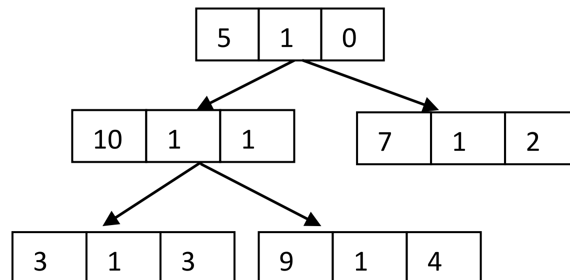
CRF(9)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(9)=tc=4



Step 6: At current time(tc)=5, referenced page=15 which is not in memory (i.e. page fault).

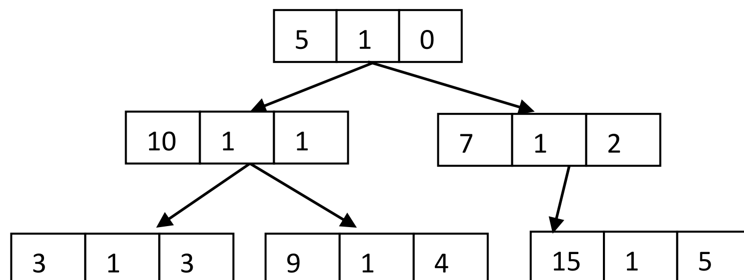
CRF(15)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

$=1$

LAST(15)=tc=5



Step 7: At current time(tc)=6, referenced page=3 which is already in the memory (i.e.page hit).

CRF (3)=F(0)+ CRF (3)

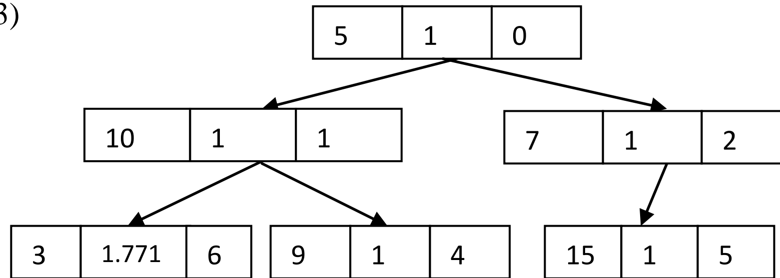
=1+ F(6 - 3)*1

=1+ F(3)*1

=1+0.771

=1.771

LAST(3)=tc=6



Step 8: At current time(tc)=7, referenced Page=6 which is not in memory (i.e. page fault).

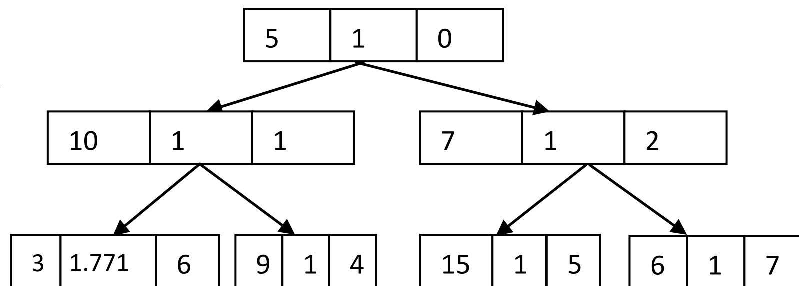
CRF(6)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

=1

LAST(6)=tc=0



Step 9: At current time(tc)=8, referenced page=12 which is not in the memory (i.e.page fault).

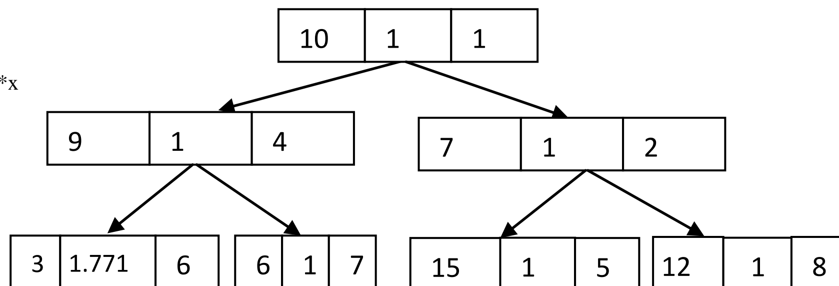
CRF (12)=F(0)

$= (1/2)^{(1/8)*x}$

$= (1/2)^{0}$

=1

LAST(12)=tc=8



Step 10: At current time(tc)=9, referenced page=15 which is already in the memory (i.e.page hit)

CRF (15)=F(0)+ CRF (15)

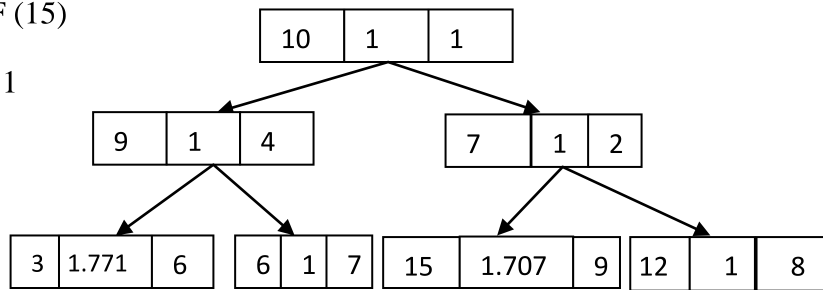
=1+ F(9 - 5)*1

=1+ F(4)

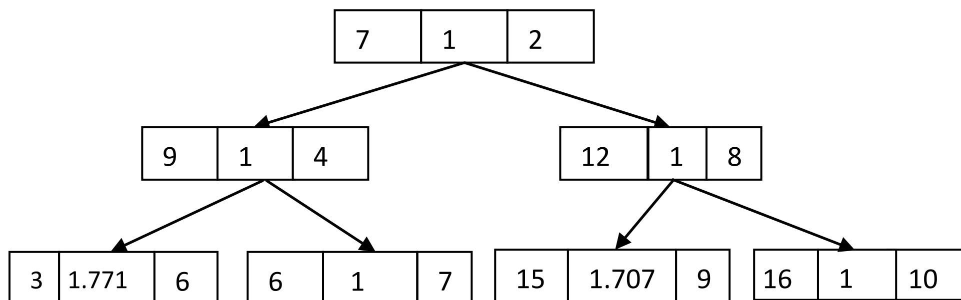
$=1+(1/2)^{(1/2)}$

=1.707

LAST(15)=tc=9



Step 11: At current time(tc)=10, referenced page=16 which is not in the memory. At this time memory is full hence, a page should be removed from the memory to make room for the fetched page. For this, swap root node with last node and discard this last node frome heap. Then, insert new node at this position and perform restore operation on each of non- leaf node of the heap.



CHAPTER 4

IMPLEMENTATION

4.1. Tools used

4.1.1. Programming Language

Java Programming Language is used for the implementation of proposed algorithms. Java is a general-purpose, concurrent, class-based object oriented programming language. It is portability language means that any program written in java language can run similarly on any other configuration of hardware and software platform. End-users generally use a Java Runtime Environment (JRE) installed on their own machine for standalone java applications, or in a web browser for java applets. Furthermore, Java is a robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage-collected language ease the programmer's virtually all memory management problems. Java also incorporates the concepts of exception handling which captures series errors and eliminates any risk of crashing the system.

4.1.2. NetBeans IDEs

IDE is an open source Integrated Development Environment (IDE), written in the Java programming language for developing primarily with java. The NetBeans platform allows applications to be developed from a set of

modular software components called modules. The NetBeans project consists of a full-featured open source IDE and a rich client application platform, which can be used as a generic framework to build any kind of application.

4.2. Data Structure Used

4.2.1. Heap

A Heap is a partially ordered complete binary tree. The complete binary tree is a binary tree in which each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right. To say that a heap is partially ordered is to say that there is some relationship between the value of a node and the values of it's children. This property is known as heap order property. In a **min-heap**, the value of a node is less than or equal to the values of its children. In a **max-heap**, the value of a node is greater than or equal to the values of its children. Consequently, smallest value (largest value) in a min-heap (max-heap) is at the root of heap.

The LRFU uses the min-heap data structure to maintain the ordering of blocks according to their CRF values. The root of the heap contains the smallest CRF value block. Since, LRFU replaces the root of the heap when page fault occurs. The algorithm first checks whether the requested block b is in the buffer cache. If it is, the algorithm recalculates its CRF value, updates the time of the last reference, and, if needed, restores the heap property of the sub-heap rooted by b. In the other case where the block is not in the buffer cache, the missed block is fetched from disk and its CRF value and the time of the last reference are

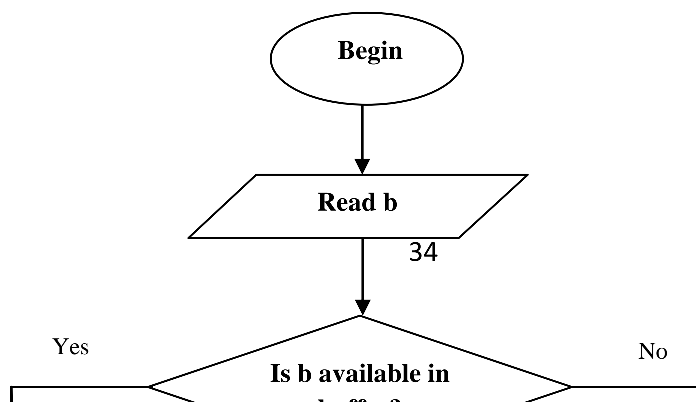
initialized. Then, the root block of the heap is replaced with the newly fetched block and the heap property is maintained.

Similarly, the Adapted LRFU also uses the min-heap data structure to maintain the blocks according to their CRF values and last reference times. The root of the min-heap contains the block whose CRF value and last reference time are smallest. The reference time is represented by real integer value. Since, Adapted LRFU replaces the root of the heap when page fault occurs. And other processes are same as LRFU takes but CRF values and last reference times of blocks are taken in consideration during restore operation.

Structure of LRFU and Adapted LRFU node

```
public class LRFUNode
        {
          int pn;                    //page  number
          int fn;                    //frame number
          int last;                  //last reference time
          double CRFlast;            //CRF value of last referenced page
          boolean isresident;
        }
```

4.3 Flowcharts



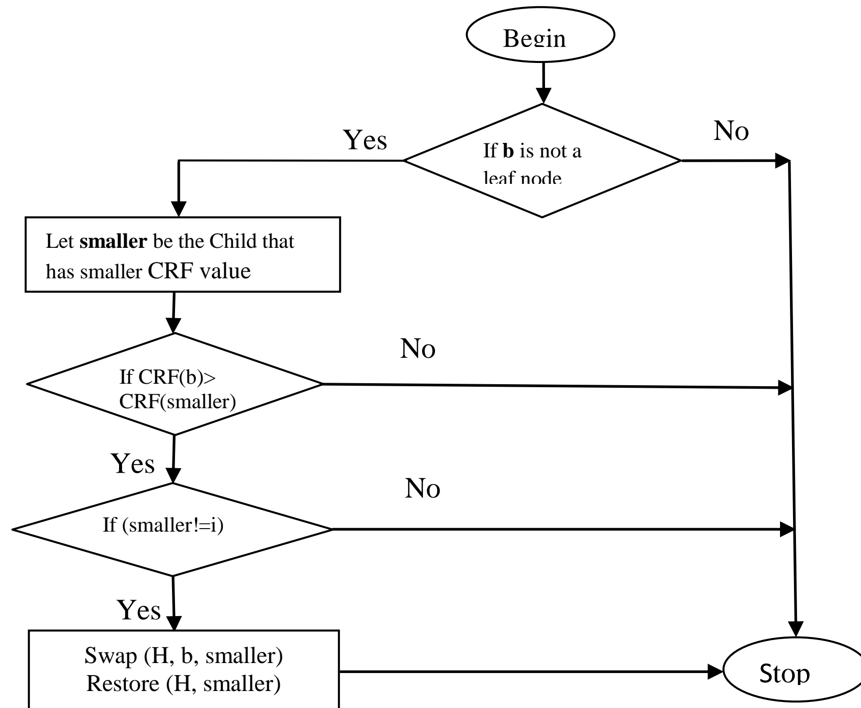
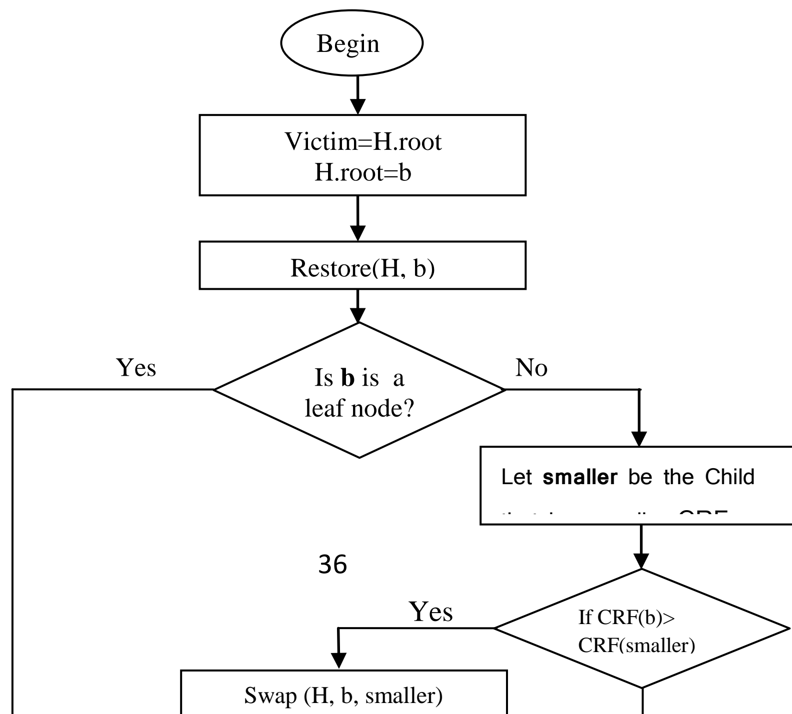
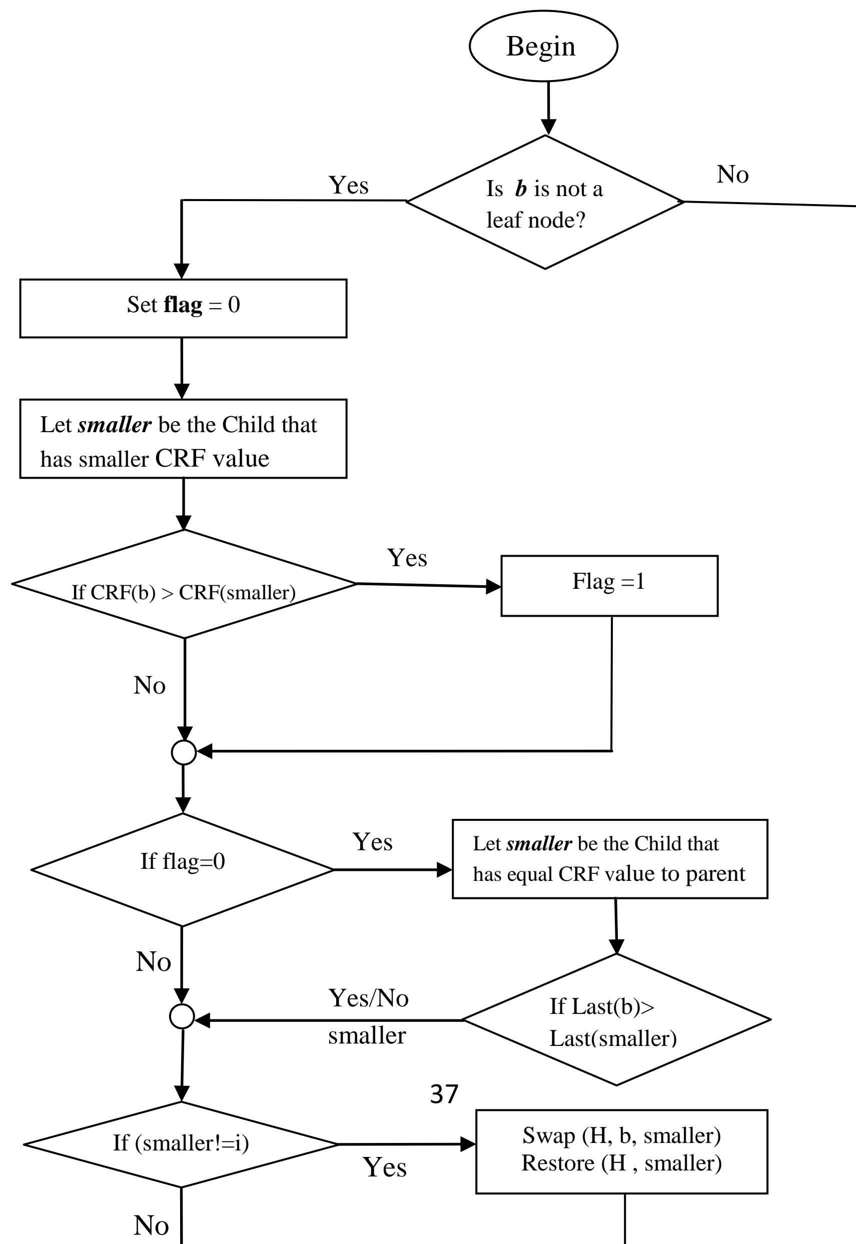


Figure 4.3 Restore Operation for LRFU





Begin
Is b is not a leaf node?
Yes
No
Set flag = 0
Let smaller be the Child that has smaller CRF value
If CRF(b) > CRF(smaller)
Yes
Flag =1
No
If flag=0
Yes
Let smaller be the Child that has equal CRF value to parent
No
If Last(b)> Last(smaller)
Yes/No
smaller
If (smaller!=i)
Yes
Swap (H, b, smaller)
Restore (H , smaller)
No

Fiqure 4.3 Restore Operation for

4.4 Memory Traces

In this dissertation work, following real memory traces are used during data collection-

a. **Cs-** is an interactive c source program examination tool trace. The total size of C programs used as input is roughly 5 Mb [17].
b. **Cpp-** is GNU C compiler pre-processor trace. The total size of C programs used as input is roughly 11 Mb [17].
c. **2_pools-** is a synthetic trace which simulates application behavior. It contains 100,000 references [17].
d. **Sprite** is from the Sprite Network file system, which contains request to a file server from client workstations for a tow-day period [17].

e. Multi is obtained by executing two workloads cs and cpp together [17].

4.5 Sample Test Case (sprite)

0 1 2 3 4 4 5 6 7 8 9 10 11 12 13 14 9 15 16 17 9 18 9 19 9 20 21 22 23 24 25 26 15 16 4 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 5 86 87 88 89 2 3 90 91 92 93 94 95 96 97 98 99 100 8 101 54 59 4 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 82 83 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 129 129 130 131 132 133 133 133 133 134 134 134 134 135 135 136 136 136 136 136 136 137 137 137 138 138 138 139 139 139 139 139 139 139 139 139 139 140 140 140 140 140 140 141 141 141 141 141 142 142 142 142 143 143 144 144 145 145 145 145 145 146 146 146 146 146 146 146 146 146 146 147 147 147 147 147 148 148 148 149 149 149 149 149 149 149 149 149 150 150 150 150 150 150 151 151 151 152 152 152 153 153 153 154 154 154 154 154 155 155 155 133 133 133 135 135 135 135 135 135 135 135 135 156 156 156 156 156 156 156 156 156 156 156 156 156 157 157 157 158 158 158 158 158 158 158 158 158 159 159 159 159 159 159 159 159 159 160 160 160 160 160 160 161 161 161 161 161 161 161 161 161 161 161 136 136 136 136 162 162 162 163 164 164 164 164 164 164 165 165 165 166 166 166 167 167 167 168 168 168 138 138 138 138 138 138 138 138 138 139 139 139 142 142 142 144 144 144 144 144 169 169 169 145 145 145 170 170 170 171 171 171 172 151 151 173 174 174 159

CHAPTER 5

DATA COLLECTION AND ANALYSIS

5.1. Data Collection

Data is a main source of information. First of all sample input is designed through hand tracing experiments such that the anomalous behavior can be verified with this. Besides this different real memory traces will be given as input to the simulated algorithms and hit rate of the algorithms is calculated. This Dissertation work will be solely based on primary data. These primary data are generated by the simulated page replacement algorithms. In this study, different types of workloads such as cyclic pattern (cs), probabilistic patterns (2_pools and cpp), temporally clustered pattern (sprite) and mixed

pattern (multi) are used as input. Number of page fault is collected with real memory traces in the interval of 1 varying memory size 8 to 32 for the purpose of analyzing anomalous behavior. In addition, data is also collected with real memory traces in the interval of 64 varying memory size 64 to 1024 for the purpose of analyzing performance.

5.1.1. Analyzing Anomalous Behavior of LRFU

5.1.1.1. Sample Input Causing Anomalous Behavior

Suppose a sample input as

2 1 5 4 2 6 1 5 5 5 6 1 5 5 5 6 1

Let memory size varies from 2 to 10 page frames in the interval of 1.

Memory Size	Number of page faults	
	LRFU	Adapted LRFU
2	11	12
3	6	8
4	10	6
5	5	5
6	5	5
7	5	5
8	5	5
9	5	5
10	5	5

Table 5.1 Number of page faults with sample input.

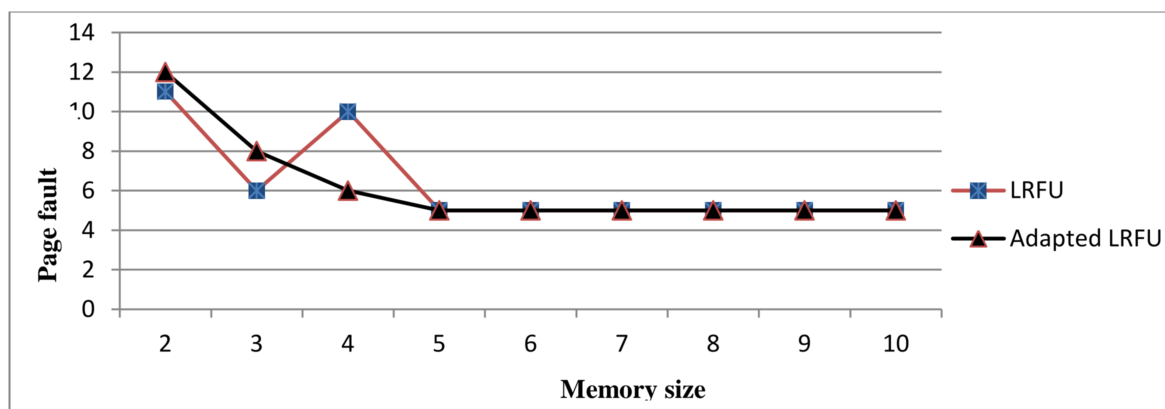


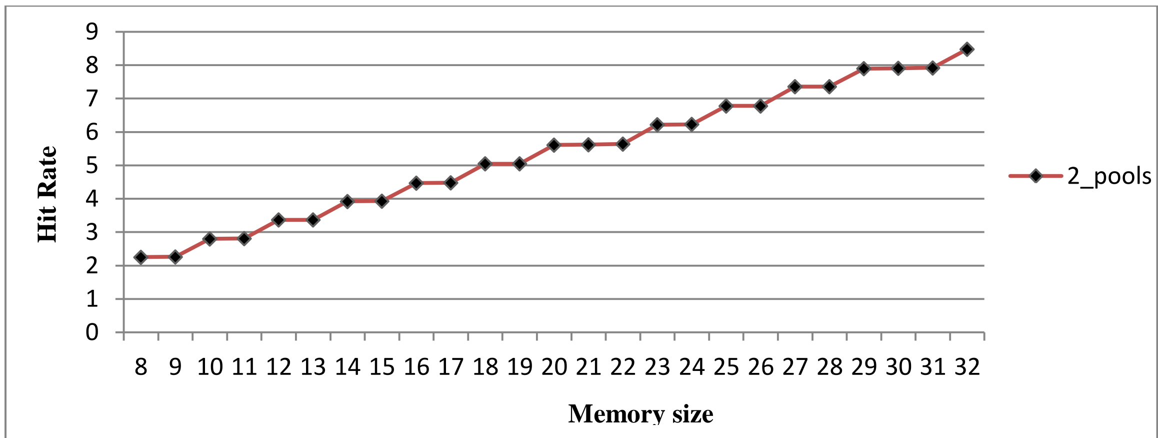
Figure 5.1 Graph for table 5.1

The graph of above figure 5.1 shows that the LRFU shows anomalous behavior sometimes. Generally, number of page faults decreases as the memory size increases. But reversely, LRFU shows that increase in memory size leads also increase in number of page faults (memory size=3, no. of page fault=6 and memory size=4, no. of page fault=10). This is because LRFU replaces recently accessed page when CRF values of memory pages are equal. The line graph of Adapted LRFU (Fig.5.1) shows that increasing in memory size leads decrease in page faults. It means that Adapted LRFU does not show anomalous behavior.

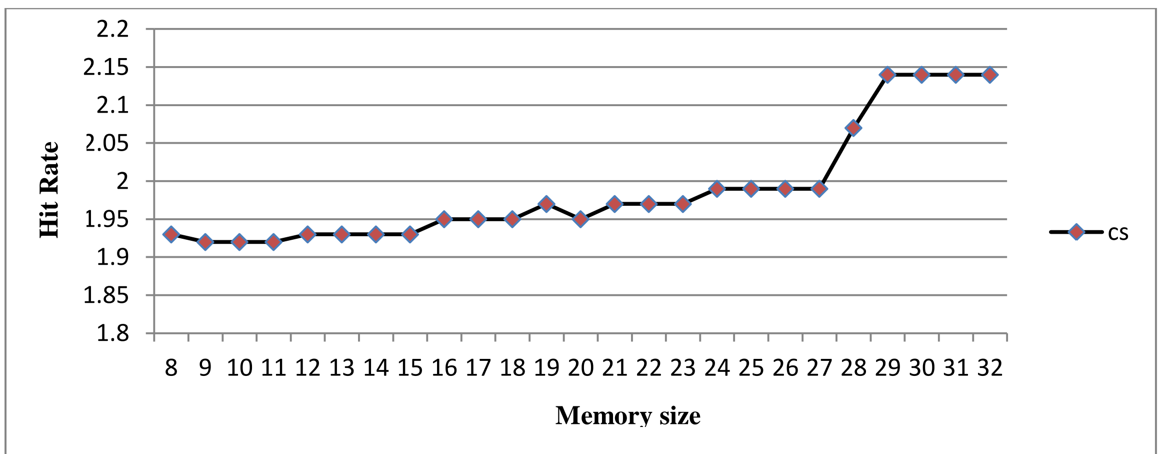
5.1.1.1. Analysis with Real Memory Traces

Memory Size	LRFU							
	2_pools		Cs		Sprite		Multi	
	Page Fault	Hit Rate	Page Fault	Hit Rate	Page Fault	Hit Rate	Page Fault	Hit Rate
8	97971	2.25	6677	1.93	129436	3.59	26162	0.72
9	97966	2.26	6678	1.92	129355	3.66	26154	0.76
10	97478	2.80	6678	1.92	129204	3.78	26169	0.69
11	97470	2.81	6678	1.92	129059	3.89	26154	0.76
12	96964	3.37	6677	1.93	129049	3.90	26141	0.82
13	96961	3.37	6677	1.93	128961	3.97	26113	0.96
14	96469	3.92	6677	1.93	128802	4.09	26113	0.96
15	96464	3.93	6677	1.93	128719	4.16	26079	1.12
16	95970	4.47	6676	1.95	128597	4.25	26078	1.13
17	95965	4.48	6676	1.95	128604	4.25	26150	0.78
18	95450	5.05	6676	1.95	128403	4.41	26150	0.78
19	95448	5.05	6675	1.97	128470	4.35	26150	0.78
20	94939	5.61	6676	1.95	128336	4.46	26154	0.76
21	94937	5.62	6675	1.97	128277	4.51	26151	0.78
22	94922	5.64	6675	1.97	128129	4.62	26135	0.85
23	94395	6.22	6675	1.97	128015	4.71	26103	1.01
24	94392	6.23	6674	1.99	127936	4.77	26073	1.15
25	93894	6.78	6674	1.99	127915	4.79	26073	1.15
26	93888	6.78	6674	1.99	127825	4.86	26065	1.19
27	93373	7.36	6674	1.99	127733	4.93	26065	1.19
28	93368	7.36	6670	2.07	127628	5.02	26058	1.23
29	92881	7.90	6666	2.14	127644	5.00	26057	1.23
30	92876	7.91	6666	2.14	127487	5.13	25993	1.54
31	92871	7.92	6666	2.14	127469	5.14	25993	1.54
32	92362	8.48	6666	2.14	127352	5.23	25986	1.58

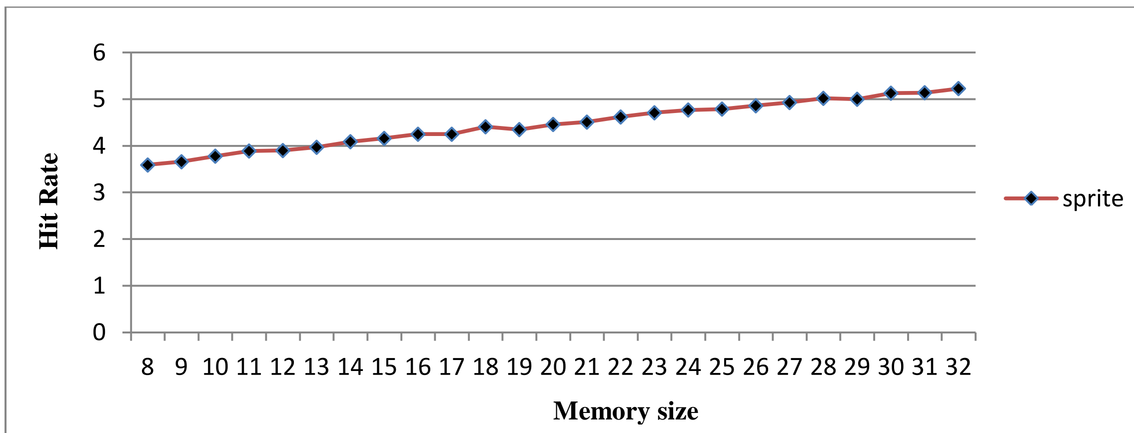
Table 5.2 Page faults and hit rates with 2_pools, cpp, sprite and multi.



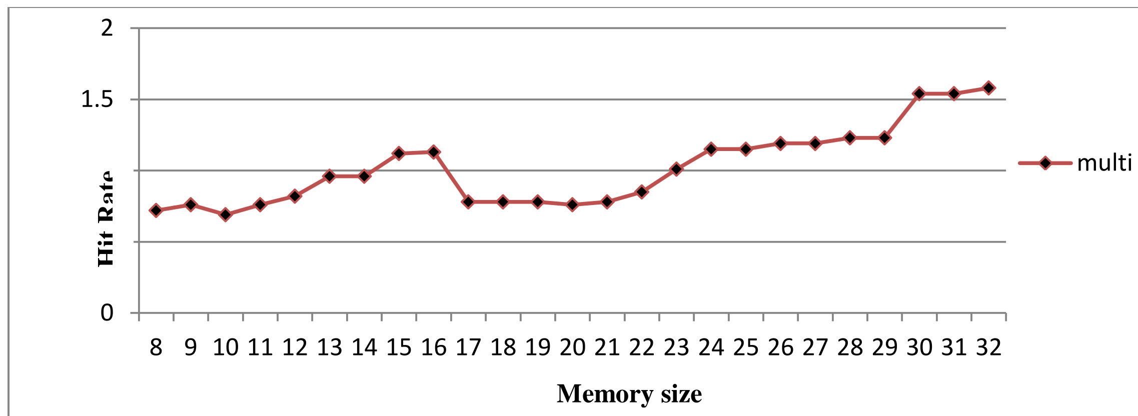
(a)



(b)



(c)



(d)

Figure 5.2 (a) Hit rates with 2_pools.(b) Hit rates with cs.(c) Hit rates with sprite and (d) Hit rates with multi.

Above graph 5.2(a) shows that the hit rates increases as the memory size also increases. Thus LRFU does not show anomalous behavior with 2_pools. But LRFU shows anomalous behavior with cs, sprite and multi because hit rate sometimes decreases with increase in memory size (see fig.5.2(b), fig.5.2(c), fig.5.2(d)) with them. Thus, LRFU shows anomalous behavior with real memory traces also.

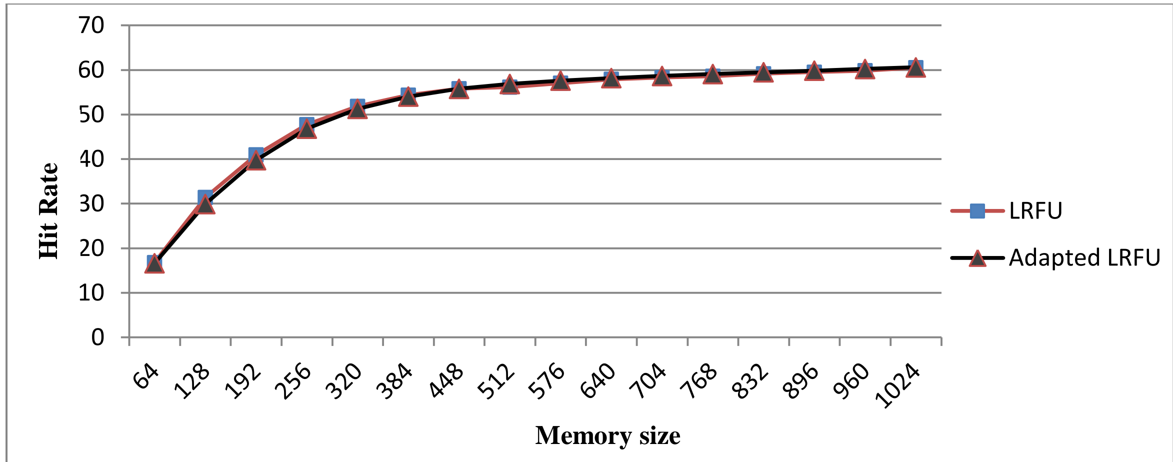
5.1.2. Performance Analysis

5.1.2.1. Test Result for 2_pools, cpp, sprite and multi

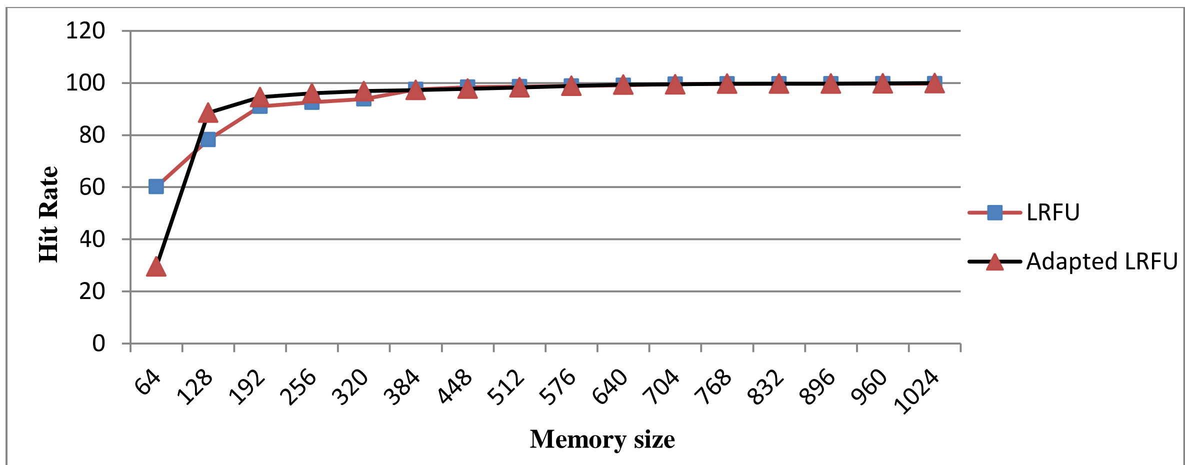
Memory Size	**LRFU Hit Rates**				**Adapted LRFU Hit Rates**			
	2_pools	cpp	sprite	multi	2_pools	cpp	sprite	Multi
64	16.80	60.22	6.22	10.65	16.64	29.59	16.71	7.62
128	31.41	78.27	8.32	23.68	29.93	88.61	30.08	30.25
192	40.97	91.03	10.61	37.41	39.77	94.56	40.72	47.85
256	47.74	92.55	12.72	41.28	46.84	96.06	53.36	51.26
320	51.84	93.88	14.36	46.85	51.29	96.84	64.27	53.58
384	54.36	97.52	14.70	49.34	54.04	97.32	73.02	54.48
448	55.80	98.40	14.89	51.65	55.77	97.81	79.14	55.22
512	56.16	98.58	15.14	53.75	56.86	98.30	82.65	55.58
576	57.05	98.82	16.51	57.25	57.62	98.91	86.95	56.19
640	57.91	99.08	21.95	59.09	58.18	99.36	89.21	56.36
704	58.28	99.55	35.10	64.54	58.65	99.46	91.19	56.69
768	58.60	99.65	43.05	66.33	59.09	99.74	92.59	56.54
832	59.15	99.68	46.65	70.14	59.49	99.76	93.81	57.53

896	59.50	99.68	52.61	72.25	59.85	99.76	94.63	57.62
960	59.85	99.73	53.25	74.15	60.22	99.82	95.38	57.70
1024	60.52	99.73	53.97	76.17	60.58	99.92	95.43	57.85

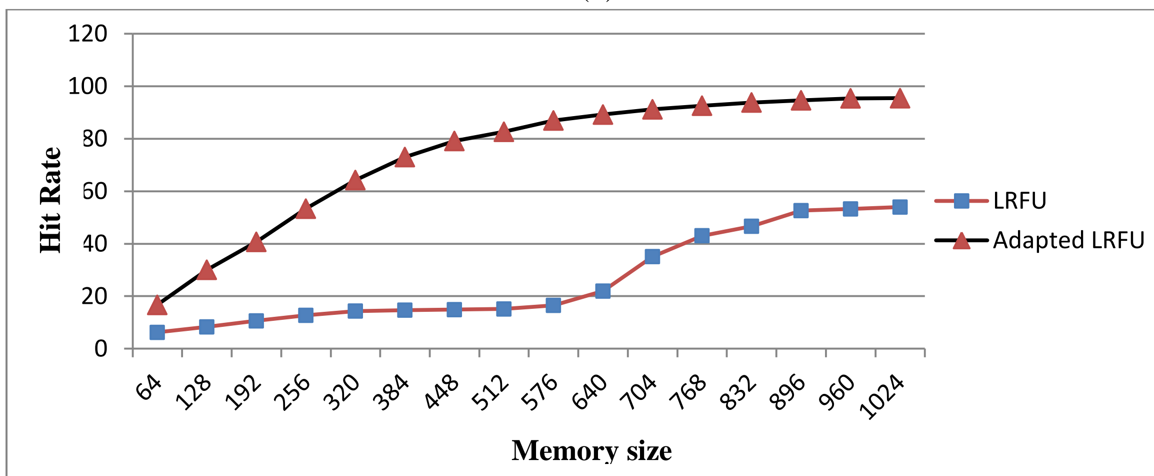
Table 5.3 Hit Rates with 2_pools, cpp, sprite and multi.



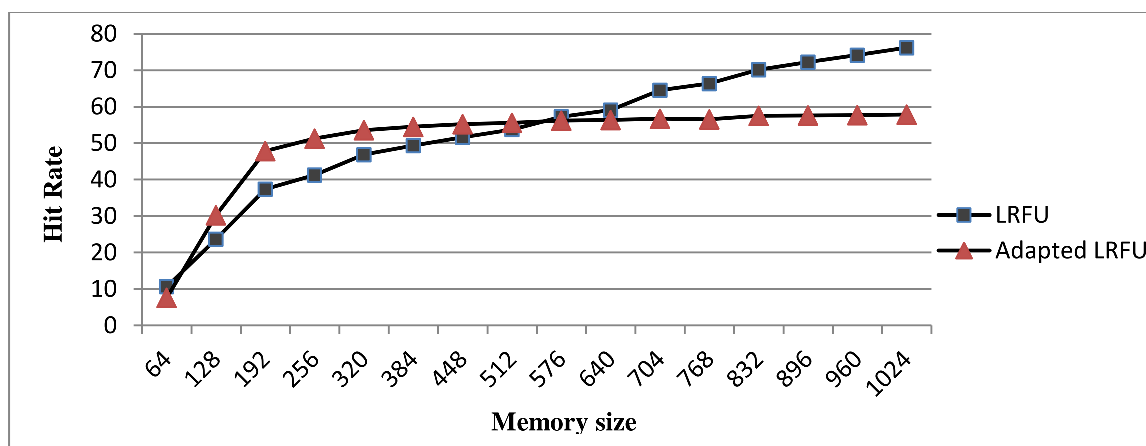
(a)



(b)



(c)



(d)

Figure 5.3 (a) Hit rates with 2_pools (b) Hit rates with cpp (c) Hit rates with sprite and (d) Hit rates with multi.

The performance of adapted LRFU is comparative with LRFU (see Fig.5.3 (a), Fig.5.3 (b), and Fig.5.3 (d)). Further, with strong locality of workload the performance of Adapted LRFU is better than LRFU (see Fig.5.3(c)).

CHAPTER 6

CONCLUSION AND FURTHER STUDY

6.1. Conclusion

Anomalous behavior of page replacement policies is the subject of research since second generation of computer. The class of algorithms that never shows anomalous behavior is called stack algorithms. LRFU is a page replacement policy that suffers from anomaly. Until to now none of the research had listed the sample input with which LRFU show anomaly.

This dissertation successfully lists a sample workload where LRFU shows anomalous behavior and also modifies the existing LRFU such that anomalous behavior can be avoided. Besides, this study used real memory traces cs, 2_pools, sprite and multi to experiment them with LRFU and showed that anomaly can also be seen with real memory traces. Finally, the dissertation compares LRFU and Adapted LRFU with real memory traces cpp, 2_pools, sprite and multi and showed that LRFU and Adapted LRFU had comparative performance. Thus, the dissertation successfully verified anomalous behavior of LRFU and able to remove this anomalous behavior.

6.2. Future Work

Even, if there are lots of studies that evaluate the impact of control parameter λ on performance. The correlation between λ and function F(x) is not much studied. Therefore, it is one of the interesting future work. Further, the performance evaluation of Adapted LRFU in other areas such as database cache replacement etc is another area of further study.

References

[1] A. S. Tanenbaum, 2008, "Modern Operating Systems (Prentice Hall Second Edition)", pp 175-248.

[2] Bagchi, S., Nygaard M., 2004, "A Fuzzy Adaptive Algorithm for Fine Grained Cache Paging". 8th International Workshop (SCOPES'04), Netherlands, pp 200-213.

[3] B. Subedi, "An Evaluation of Page Replacement Algorithm Based on Low Inter-Reference Recency Set (LIRS) Scheme on Weak Locality Workloads", Master's

Dissertation in Computer Science and Information Technology, Tribhuvan University, Central Department of Computer Science and Information Technology.

[4] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, 1996, "LRFU (Least Recently/Frequently Used) Replacement Policy: A Spectrum of Block Replacement Policies," IEEE Trans. Computers, March, pp. 1-24.

[5] E. G. Coffman and P.J. Denning, 1973, "Operating Systems Theory". Prentice-Hall.

[6] E. J. O'Neil, and G. Weikum, 1993, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", proc. 1993 ACM SIGMOD Int'l Conf. Management of Data, pp.297-306, May.

[7] G. Glass and P. Cao, 1997, "Adaptive Page Replacement Based on Memory Reference Behavior," Proc. 1997 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, May, pp. 115-126..

[8] H. Paajanen, 2007, "Page Replacement In Operating System Memory Management", Master's Thesis in Information Technology, University of Jyvaskyla, Department of Mathematical Information Technology, October 23.

[9] John L. Hennessy and David A. Patterson, 2010, "Computer architecture A Quantitative Approach", Mogan Kaufmann Publisher fourth edition, pp, C1-C58.

[10] Joshi Gyan Prakash, 2004, "Calculation Of Control Parameter λ That Results Into Optimal Performance In Terms Of Page Fault Rate In The Algorithm Least Recently Frequently Used (LRFU) For Page Replacement", Master's Thesis, Tribhuvan University, Central Department of Computer Science and Information Technology.

[11] J. T. Robinson and N.V. Devarakonda, 1990, "Data Cache Management Using Frequency-Based Replacement," Proc. 1990 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, , May, pp. 134-142.

[12] K Muralidhar, and N. Geethanjali, 2012, "Fuzzy Replacement Algorithm for Browser Web Caching". International Journal of Engineering Research and Applications (IJERA) ISSN: 2248-9622, Vol. 2, pp.3017-3023.

[13] L. A. Belady, 1966, "A Study of Replacement Algorithms for a Virtual-Storage Computer", IBM SYSTEM JOURNAL, pp. 78-101, vol. 5, No. 2.

[14] Megiddo, N. and Modha, D. S., 2003, "ARC: A Self-Tuning, Low Overhead Replacement Cache", In Proceedings of the USENIX Conference on File and Storage Technologies (FAST'03), San Francisco, pp 115-130.

[15] Silberschatz, A., Galvin, P. B., & Gagne, G., 2004, “Operating system concepts (7th Edition)”. Wiley.Stuart.

[16] Sabeghil, M. and Yaghmaee, M. H., 2006, “Using fuzzy logic to improve cache replacement decisions”. IJCSNS International Journal of Computer Science and Network.

[17] S. Jiang and X. Zhang, 2002, “LIRS: An Effective Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance,” Proc. SIGMETRICS, PP. 31-42.

[18] Song Jiang and Xiaodong Zhang, 2005, “Making LRU Friendly to Weak Locality Workloads: A Novel Replacement Algorithm to Improve Buffer Cache Performance”, IEEE Transactions on Computers, Vol. 54, and No. 8, August, pp 939-952.

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

[20] Y. Smaragdakis, S. Laplan, and P. Wilson, 1999, “EELRU: Simple and Effective Adaptive Page Replacement”, Proc. 1999 ACM SIGMETRICS Conf. Measuring and Modeling of Computer Systems, May, pp. 122-133.