CHAPTER 1
BACKGROUND AND PROBLEM FORMULATION

1.1 Background

Before exploring or diagnosis of any research it is necessary to study the basic terminologies which are related to the topic. In the same manner, hereby, some of the basic terminologies which are relevant to this study are discussed in the following sections.

1.1.1 Memory Management

Memory is an important and limited resource in computer which is to be managed carefully. The act of managing and organizing computer memory is known as memory management. Memory is basically categorized into two types depending upon its location from processor namely primary memory and secondary memory. Primary memory is volatile computer memory, which is accessed frequently by processor whereas secondary memory is non-volatile memory which is not often accessed by processor. When a process to be executed, it must be loaded into main memory because access time of primary memory is far better than secondary memory. When the program size is larger than the available main memory that is too big to fit on main memory there arise problem which needs to be handled by memory management. The solution to overcome this problem may be to increase significantly the amount of available memory but this would have penalty in the cost of the whole computer system. In this context, the solution is to split program into large number of small logical sections called overlays (pages) [19].The execution starts from overlay 0. After executing overlays 0, it calls another overlay. In such way that operating system keeps active overlays for execution in the primary memory and rests are stored in secondary memory. The process of moving required data from secondary memory to main memory is called swapping in and the process of moving unused data to secondary memory is called swapping out. Although, swapping in and out is done by the operating system and the partition of large program into overlays is done by programmer which adds extra burden to the programmer and this working process seems too much time consuming. Therefore, the concept of virtual memory

is developed. Now, the operating system uses virtual memory system to solve the overlay problem. The concept of virtual memory is defined more clearly in the following section.

1.1.2 Virtual Memory

Virtual memory is a memory management technique for executing large program or multiple programs which cannot be accommodated in main memory entirely. It moves required data to the main memory for execution and rest of it which is not currently being in execution is stored in secondary memory. The part of the secondary memory that is reserved for virtual memory is called swap space. The whole process of virtual memory system is depicted in figure 1.1.

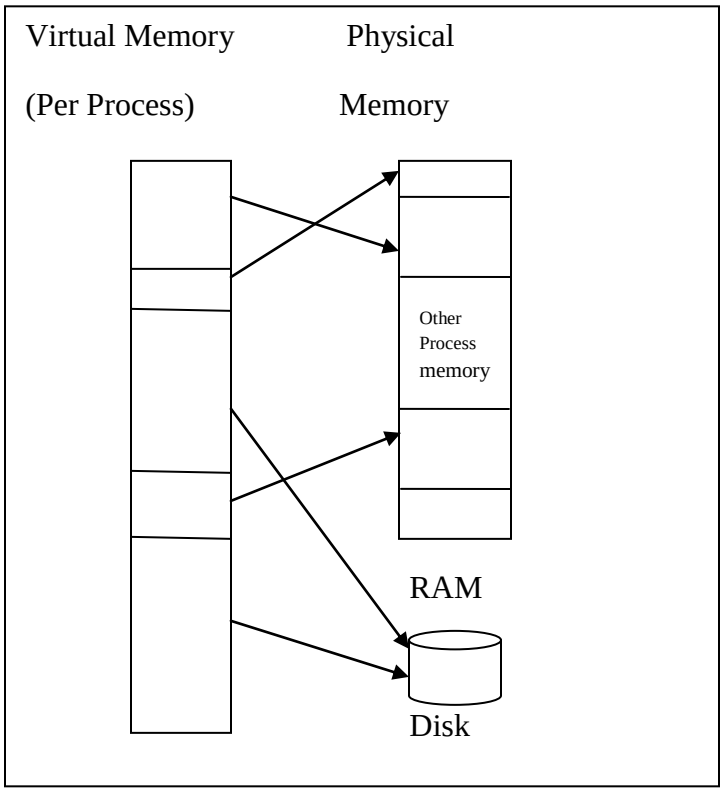

Figure 1.1: Virtual Memory System

Program and data are stored in number of memory cells. Each cell has a unique identifier called address. The ordered set of address is called address space. During the program execution, the CPU generates address by accessing main memory is called virtual address. The set of all virtual (logical) addresses generated by a program is a virtual address space; the

set of all physical addresses corresponding to these virtual addresses is a physical address space. This is then presented to memory management unit (MMU), and is responsible for run time mapping from virtual address to physical address.

1.1.3 Paging

Paging is one of the most popular memory management schemes to implement a virtual memory management which divides main memory into sufficient large number of blocks or units called page frames [17,19]. On the other hand, virtual address space is also divided into blocks called pages having same size as that of page frames. This equality facilitates to move any page from secondary memory to any page frame in main memory. Each page in the virtual address space must be mapped to appropriate page frames by translating their virtual address to the corresponding physical address. This process of translation is called address translation that is done with every new memory references. Usually, special hardware memory management unit (MMU) is used to make this translation. The mapping between virtual address spaces to physical address space is stored in a table named page table. When the given logical address does not map to the physical address then MMU traps to the operating system which is called page fault. The operating system moves some pages to secondary memory to make room for reference page and updates the page table, as the MMU does not know anything about present/absent status of pages in main memory. By paging, the memory allocates to the blocks that are requested. Thus, the problem of external fragmentation is completely eliminated. Internal fragmentation exists where there is unused memory within a page. Larger pages make for smaller page tables and use the TLB more efficiently but create more internal fragmentation.

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 block and finally replacement procedure decides which page to be swapped out when required page have to be brought in. Further, paging algorithm can 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 better choice because its further uses but pre-paging is not in real use because it requires prediction of page uses which is impossible to predict [13].

1.1.4 Page Replacement Algorithm

Operating system uses paging for managing virtual memory and virtual memory makes use of page replacement algorithm for efficient access of pages. These are responsible for allocation of requested pages in main memory by replacing the unused pages to secondary memory. If the requested page is not available in main memory then it encounters page fault. In case of page fault, it removes or replaces unused page from main memory with requested page. The effectiveness of page replacement algorithm depends on its selection procedure of unused pages to deportation. This means the performance is much better if a page that is not heavily used is chosen for replacement. If the heavily used is removed, it will probably have to be brought in quickly, resulting in extra overhead.

The main goal of page replacement algorithm is to minimize the number of page fault. 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 criterion 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 [17]. Therefore, to determine the number of page fault the number of page frames used there should also be known. Different types of page replacement algorithm and their working behavior is described in more detail in chapter 2.

The performance of a page replacement algorithm depends on the program behavior which demonstrates how reference strings are generated by program. Reference strings are either generated randomly, or by tracing the paging behavior of a system and recording the page number for each logical memory reference. Behavior of program depends on the access pattern of references which further depends upon working set and locality of reference.

1.1.5 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 much computation will not be performed, finally it suffers from thrashing [5, 19]. The entire algorithm based on principle of working set is not in real use but the informal concept of working set is still being used.

1.1.6 Locality of Reference

The allocation of limited set of pages in main memory during particular time of execution is called locality [5]. The algorithm that exhibits weak locality impacts the cache performance optimization. So, the locality of reference is most important principle in virtual memory system as it is found in currently used page algorithm. Page replacement algorithm employs two types of locality namely temporal locality and spatial locality. Temporal locality is based on the time which refers the memory location referenced at any point in time likely to be reference again in fixed time interval. Spatial locality is based on space which refers the memory location referenced once then program will be referenced again by nearby memory location.

1.1.7 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.

1.1.7.1 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 1,2,3,4 are reference block then their cyclic pattern will likely to be 1,2,3,4, 1,2,3,4, and 1,2,3,4.

1.1.7.2 Probabilistic Pattern

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

1.1.7.3 Temporally Clustered Pattern

A temporally clustered pattern has the property that block 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.

1.1.7.4 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.

1.1.8 Performance Metrics

Performance metrics refers criteria for measuring the performance of any system. 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.

1.1.8.1 Page fault count

Page fault count can be measured by counting total number of page faults occurred between the some intervals of references.

1.1.8.2 Hit Rate and Hit Ratio

This is the rate of hitting the page in main memory out of total referenced pages which is calculated by using formula

$$HR = 100 - MR \quad \text{(1.1)}$$

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.

1.1.8.3 Miss Rate and Miss Ratio

Miss rate (MR) can be calculated by using formula

$$MR = 100 \times ((PF - NDR) / (REF - NDR)) \quad \text{(1.2)}$$

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 [15].

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

1.2 Rationale of the study

Day by day the performance gap between memory and processor gets wider. Consequently, various solutions are proposed to cope with this issue. A high performance cache replacement algorithm is one method to reduce cache misses for speeding up computer system. Various page replacement algorithms have been researched in recent years [19, 15]. To date, LIRS (Low Inter Reference Recency Set) is currently known as very effective method follows the original assumption of LRU algorithm which is based on the principle that recently accessed block will be re-accessed very soon in future. Although LRU seems to be simple and easy to implement, it cannot gain better performance because of stored limited history information. Also the adaption of all scales of reference pattern such as file scanning, cyclic pattern (loop-like), access pattern that is marginally larger than the cache size, access of different frequency blocks is difficult. In this context the concept of LIRS algorithm was proposed to overcome all the limitations of traditional LRU algorithm.

The main idea behind the LIRS algorithm is to reduce the number of page faults by using IRR (Inter-Reference recency set) as recording history information of each block. Where, IRR of the block refers number of distinct access between two accesses of the same block in the reference sequence. Another factor, Recency refers to the number of distinct accesses between last references to the current time used to evaluate IRR value of accessed block. The algorithm assumes current large IRR value of a block generates next large IRR value for accessed block. So, the block with largest IRR is selected for eviction [7].

LIRS algorithm classifies each reference block into two categories [7]: LIR block and HIR block. The mapping process of referenced block to physical cache is depicted at figure 1.2. A block which associates low IRR value is taken as LIR block and block with high IRR value is taken as HIR block. In the same manner, cache is also partitioned into two parts C_{Lirs} and C_{Hirs}. C_{Lirs} contains set of LIR blocks and have higher probability of being referencing in near future and re-accessing these blocks always guaranteed to be hit. C_{Hirs} contains set of HIR blocks that have the less probability of usability in future and accessing those blocks encounters page fault. The algorithm dynamically maintains data blocks in the cache; LIR blocks in the cache as much as possible and HIR value block out of the cache as page fault. That's why this algorithm is called LIRS algorithm.

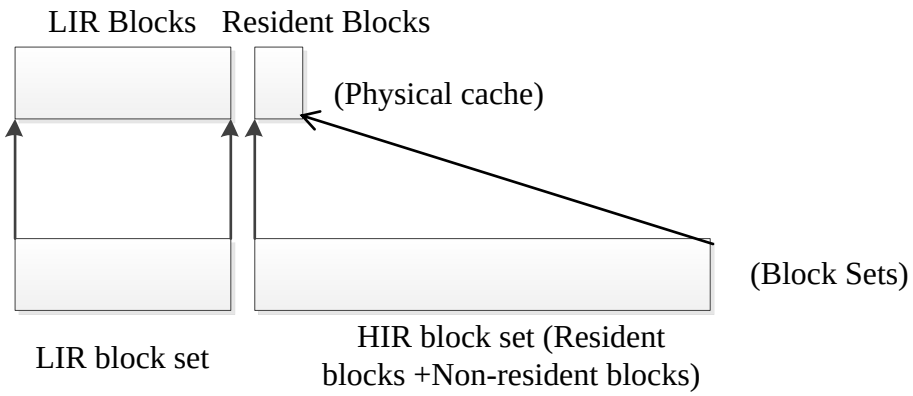

Figure 1.2 Mapping process of Logical memory to physical cache.

C_{Hirs} part of cache is again classified into two parts namely resident block and non-resident block. Resident block keeps the track of those HIR blocks that are currently available in the cache. Such types of blocks may be evicted anytime whatever the value Recency is. And non-resident block keeps the metadata[1] of limited HIR blocks that are not currently in the cache. Status of a block of LIR and HIR might be interchanged only when the following condition is true.

$$Max(\text{Re}\,cency\ of\ \ LIR\,Block) > (\,New\,Born\ IRR\ of\ \ HIR\,Block\ ).$$

This policy uses new born IRR value of HIR block in case of changing the block status. In this case, the metadata of all accessed C_{Hirs} block cannot keep in the cache because it

[1] Metadata is data that provides information about accessed block.

performs pruning operation to minimize the space overhead. DIG algorithm [4] reveals a problem of using new born IRR value of C_{Hirs} that is the recently accessed block will never replace with frequently accessed block. By this, the performance of LIRS somehow degrades.

1.3 Problem Statement

Recency of an accessed block depends not only on its own reference activity, but on other blocks reference activity too. When the blocks recency value became a maximum then obviously it gets a large IRR value because IRR value of accessed block depends its own earlier recency. The LIRS algorithm presented in [7] directed to use a new born IRR value of C_{Hirs} for comparison with maximum recency value of C_{Lirs}. And it performs pruning operation while changing page status which plays key role in the performance degradation of LIRS algorithm.

When there is burst of first time (or fresh) block references, the size of stack grows unacceptably very large which makes cache polluted. To minimize the stack overhead, the entire HIR blocks having recency larger than the max recency of LIR blocks removed one by one. With such large limits, there is negative effect on LIRS performance by removing these removed HIR entries. As a consequence, it can store only history information of newly accessed block neither the history information of frequently accessed block. So, this thesis focuses on identity the impact of pruned on LIRS algorithm performance by comparing the max recency of LIR block with the minimum IRR of HIR block and integrating the new born IRR of accessed block and minimum IRR value of all C_{Hirs} to compare with maximum recency of all $C_{Hirs.}$

1.4 Objective

The objective of this study is to evaluate the impact of pruned data on LIRS algorithm by simulating it without violating its logic and compare the performance.

1.5 Motivations

Less time for page-ins, deeper history information about access to the pages and minimum number of misses are the key feature to recognize as a good algorithm. OPT is one, which gives bold logic for page replacement even it is not on a real use. Many near optimal page replacement algorithms have been found, but their complexity and restrictions restrict their implementation in real system.

Implementing a LRU is a successful idea due to its simplicity, flexibility and low overhead. By motivating from bold assumption of LRU a versatile LIRS algorithm was implemented. Its power of quantify to locality, way of selecting victim page and role of metadata on cache performance is the most important motivation for making a research in this field. Related research paper demonstrates the deployment of LIRS algorithm in different approach based on its principle. Before deploying the algorithm in a real system the question should be answered how much an algorithm stores deeper history information.

1.6 Thesis Organization

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

Chapter 2 consists of literature review which briefly reviews the related topics. Literature review includes summary of several traditional page replacement algorithms like Optimal, LRU, MRU, LRFU, 2Q etc. This chapter also contains the research methodology part which shows the flow of our research.

Chapter 3 is devoted to the program development steps of simulation. It includes detail design of the program.

Chapter 4 consists of data collection and analysis part which includes details about generating traces of memory references that shows trace driven input, output results with several analyzing graphs which are only tested for weak locality workloads.

Finally, the concluding remarks and further recommendations are outlined in chapter 5.

CHAPTER 2

LITERATURE REVIEW AND METHEDOLOGY

2.1 Literature Review

The brief introduction about page replacement is described in first chapter. There are so many algorithms developed and employed as the page replacement algorithm. Although it is infeasible to mention all of them, and all those are not seemed to be relevant to this dissertation work. Some of the main page replacement algorithms which are hugely used in the page replacement and relevant to this research are described below.

2.1.1 Optimal Page Replacement Algorithm

According to the author in [1], an optimal page replacement algorithm (OPT page replacement algorithm) suggests replacing a page that will unreferenced for a long period of time which is also known as Belady's MIN algorithm. This algorithm seems unpredictable for real systems because it requires knowledge of future reference sequence, which is really impossible to predict. This algorithm provides less page faults and never suffers from Belady's Anomaly[2]. As a result, OPT is used to compare the performance of other page replacement algorithms.

2.1.2 Random Page Replacement Algorithm

Random page replacement algorithm as its name suggests that it randomly chooses a page for eviction from a buffer without concerning it for future use. Its performance relies on type of block which has been evicted by algorithm. According to this algorithm performance benefits will be available if the evicted page is not frequently accessed otherwise the performance obviously degrades. Due to its random nature, in most sequences it produces unnecessary page faults and thrashed them unnecessarily. Therefore, the concept of random page replacement algorithm is not currently deployed due to issue of bad performance [19].

[2] Belady's anomaly is a problem that increases page fault rates while increasing number of page frames.

2.1.3 FIFO Page Replacement Algorithm

First in first out page replacement algorithm throws out the oldest page in case of page fault. All the referenced pages are kept in list such that the oldest page at the head and the most recently referenced page at the tail. On a page fault, the oldest page from head of list is selected to evict and new pages added to the tail of list. Each page has an equal lifetime in memory and there is discrimination among the pages on the basis of frequency. As mentioned in [15], it suffers from Belady's anomaly and cannot take advantages of locality trend.

2.1.4 NRU Page Replacement Algorithm

In the Not Recently Used page replacement algorithm [19], each page in the memory is classified into four classes according to referenced and modified bit. Class 0 contains neither referenced nor modified pages, class 1 contains not referenced but modified pages, class 2 contains referenced but not modified pages and Class 3 contains referenced and modified pages. NRU randomly selects the page from the least needed class when page fault occurred.

2.1.5 Recency/Frequency Based Page Replacement Algorithm

These entire page replacement algorithms make their cache replacement decision based on the Recency or Frequency or both. However, they are not able to exploit human desirable cache performance. Recency and Frequency Based Page Replacement Algorithm is further categorized into different categories which are described below:

2.1.5.1 LRU Page Replacement Algorithm

Least Recently Used (LRU) algorithm assumes that page referenced at any instance will probably be used very soon. This policy exploits principle of locality for many common memory access patterns [7]. All the access to the block is recorded and replaces a page that has not been used for long time. The limitation of this algorithm is its implementation because it requires additional hardware. Two types of implementations are feasible which are Counter and Stack.

Counter: Counter is just like dummy variable which value is increments when memory is accessed and the current value of this counter is stored in the page table entry of the page. Then finding the LRU page involves simple searching the table for the page with the smallest counter value.

Stack: LRU keeps all referenced pages into a sorted list which is sorted on the basis on their accessed time from reference pattern and the list is called LRU stack. The only one recency factor is used to record history information which is evaluated by maintaining LRU list. The recency factor is updated as per entry of new page in LRU stack.

There is no any mechanism to differentiate frequently accessed and rarely accessed blocks. If frequently accessed block has slightly large reuse distance it cannot be fit into main memory, this time it ignores usability of a block. And it shows anomalous behavior in many workloads [7].

2.1.5.2 LFU Page Replacement Algorithm

Least Frequently Used is another classic algorithm which keeps track of the frequency of each block by counting the number of times references with other page. The term frequency is used to estimate the probability of next page to be referenced. According to this algorithm page with the least frequently used counter is replaced with new page. The problem arises when a frequently accessed block at initial will be not needed again for long time staying in cache and restricting new data to be cached in. Aging is one method to avoid such types of cache pollution.

2.1.5.3 MRU Page Replacement Algorithm

Most Recently Used page replacement algorithm works with reverse principle of LRU algorithm which removes the most recently used resource first. This algorithm is considered best when the reference pattern is highly unpredictable. It can be implemented like as LRU by maintaining LRU stack. MRU is mostly applied in the database memory caches.

2.1.5.4 LRFU Page Replacement Algorithm

Least Recently/Frequently Used page replacement inherits the advantages of LRU and LFU algorithm. According to this algorithm each block contains CRF (combined recency and frequency) value and any block with minimum CRF C(x) is victim when new block is encountered [10].

$$C(x) = \begin{cases} 1 + 2 -\curlywedge C(x) & if\ x\ is\ reference\ in\ time\ t \\ 2 -\curlywedge C(x) & otherwise. \end{cases} \quad \text{..... (2.1)}$$

Where, $\curlywedge$ is tunable parameter which is defined as…..if definition or physical meaning, mentioned here. The performance of CRF is critically depended on the choice of $\curlywedge$.

2.1.5.5 EELRU Page Replacement Algorithm

EELRU-Early eviction LRU [18], works on the basis of the position of LRU queue and memory reference pattern rather than the memory size. EELRU outperforms over LRU by default and page is chosen to evict based on the aggregate recency information. So the hit rate of EELRU algorithm seems better than the simple LRU. Recency information on EELRU is also used to maintain the information of resident and nonresident pages. It detects the potential sequential access patterns analyzing the reuse of pages.

2.1.6 Enhanced LRU algorithm

2.1.6.1 LRU-k algorithm

LRU-K algorithm [14] 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. When the value of k is large then it discriminates frequently accessed and infrequently accessed block. The problem arises when more than one block has undefined backward distance. In this case, the supplementary policy is applied to overcome this problem.

2.1.6.2 2-Q Algorithm

The main intuition of 2Q algorithm [9] is the detection of real hot pages and removes 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 hotlist managed by LRU stack. The first queue is further partitioned into two sets F_{in} and F_{out}. The first referenced block is placed in F_{in} list while F_{out} contains only information of a missed block. The re-accessed block on F_{in} list 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.

2.1.6.3 ARC algorithm

ARC algorithm keeps track of both frequently and recently used pages along with history data regarding eviction. ARC uses two types of LRU lists L_1 and L_2 to manage the pages. L_1holds pages accessed only once and L_2keeps 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 $|T_1+T_2|=c$. where c is the cache size. Suppose $|T_1|=p$ then $|T_2|=c-p$. The parameter p is dynamic and it may be incremented and decremented based on the respective size of two sets B_1 and B_2. Same parameter controls the replacement point in L_1 and L_2 [11].

2.1.7 CLOCK Based algorithms

CLOCK, CAR, CART 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 [6]. CLOCK based algorithms hold the information regarding how frequently block has been accessed and page has been accessed but these algorithms have limitation that unable to detect an access pattern.

2.1.7.1 CLOCK algorithm

In CLOCK, replacement is done by inspecting the oldest page. The reference bit of block is set whenever it is referenced. If reference bit is zero, the page is replaced with new one and the hand is advanced one position. If reference bit is one then the bit is reset to zero and advance to the next page. This process continues till a page with reference bit set to zero.

2.1.7.2 CAR

CAR [3] is a variant of ARC based on clock algorithm. Like ARC, it uses two lists to hold information of each page, but is controlled by pointer like as in Clock. The scanned pages replaced from first queue and their information is maintained in data structure. CAR is self-tuning algorithm. It uses both the recency and frequency of page and balanced between them automatically.

2.1.7.3 CAR with Temporal Filtering (CART)

CART is a variation of CAR which includes additional mechanism to filter out correlated reference pattern which seems undesirable in CAR. These filters classify pages as long utility and short term utility. Least recently used pages from list are evicted [3].

2.1.7.4 CLOCK –Pro

CLOCK- Pro [8] is an approximation of LIRS. Reuse distance, analogous to IRR of LIRS is used instead of recency of block to make replacement algorithm. A block with large reuse distance is considered as the cold block and block with minimum reuse distance is considered as the hot block. CLOCK-Pro preserves ghost cache holding recent replaced pages as an additional reference data structure.

2.1.8 Fuzzy Logic based cache replacement algorithm

Recent algorithm uses Fuzzy logic to improve cache performance decisions. Fuzzy logic is a way of computing based on "degrees of truth" rather than the usual "true or false". Fuzzy Inference Systems (FIS) consist of three stages: an input, processing and an output stage. Input stage mapped with frequency of references, recency of references and others. An intermediate processing stage is also called reference engine contains logic rule in the form

of If-Then statement [12]. Finally, output stage converts reference engine produced output into desired output value. FPR [16] and FAPR [2] algorithms apply fuzzy inference technique to make pages replacement decision.

Research related to replacement schemes for buffer and page cache management has been done for a long time. Among them most of page replacement schemes are based on tracing and utilizing history information. These schemes choose the victim block by using ‘‘deeper’’ history information. Replacement schemes based on tracing and utilizing history information choose the victim block by using ‘‘deeper’’ history information such as recency, frequency, and inter-reference gap. These schemes generally need low memory and timing overhead, and are simply easy to implement. LRU-K, 2Q, LRFU, and LIRS are the typical schemes found in this category.

The LIRS 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. Therefore, here proposed a scheme Derived LIRS which modifies and improves the LIRS by solving the problem of that assumption. The Derived LIRS is described in more detail in chapter 3.

2.2 METHEDOLOGY

As methodology is the systematic, theoretical analysis of the methods applied to a field of study, or the theoretical analysis of the body of methods and principles associated with a branch of knowledge. It, typically, encompasses concepts such as paradigm, theoretical model, phases and quantitative or qualitative techniques. As discussed in the earlier chapter this study is about LIRS page replacement algorithm and under this study different aspects of LIRS algorithm like IRR value, recency, are taken into consideration or excluded as necessary and their impact is analyzed for hit rate, miss rate and number of page faults.

This study is based on the trace driven simulation approach where all data collected are secondary data which are referenced from [7]. The implementation is done in simulator which is programmed in C# programming language in .Net Framework.

Thereafter, output information gathered is analyzed in quantitative approach with the help of table and graphs. Also impacts of taking into consideration the IRR value of reference pages, Recency are compared and analyzed. Finally, conclusion is drawn from those graphs and tabulated information.

CHAPTER 3

PROGRAM DEVELOPMENT AND IMPLEMENTION

3.1 Development Methodology and Tools

3.1.1 Programming Language Used

C# programming language in .Net Framework is used for simulating the LIRS page replacement algorithm. C# (Pronounced ‘see sharp’ or ‘c sharp’) is one of the many .Net programming language. It is simple, modern general purpose and object oriented programming language. Like other programming language it can be used to create a variety of application. An important point is that c# is a managed language; it requires that .NET Common Language Runtime (CLR) to execute. Essentially, as an application that is written in C# executes, the CLR is managing memory, performing garbage collection, handling exceptions, and providing many more services. The C# compiler produces Intermediate Language (IL), rather than machine language, and the CLR understands IL. .NET ships with a .NET Framework Class Library (FCL), which includes literally tens of thousands of reusable objects. This can help to write managed code for different purposes.

3.1.2 Data Structure Used

Doubly linked list is used to implement a LIRS algorithm. A doubly linked list is collection of sequential records called nodes. Where each node contains its current Recency value, IRR value and each node references the next and previous one. The advantages of using doubly linked list is that if items are inserted and deletion from the list, the doubly linked list is very fast. And another beauty is it can be traversed in both (forward and backward) more easily. The structure of doubly linked list is illustrated in figure below.

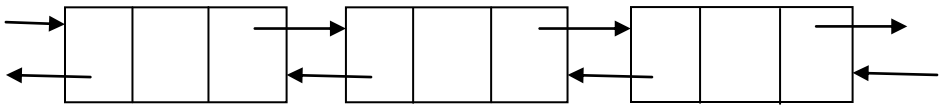

Figure 3.1 Structure of the Linked List

While implementing LIRS algorithm, two operations have to be performed- insertion new referenced node at front and moving of node from existed position to front position. If the existed block 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. Otherwise new nodes are linked to the front of list. And same doubly linked list is used for maintaining their page status. Thus, to implement LIRS algorithm efficiently the doubly linked list is selected.

Structure of LIRS Node

```
Class LIRSnode{
Public LIRSnode prev { get; set; }
public object Data { get; set; }
public LIRSnode next { get; set; }
public int Recency { get; set; }
public int IRR { get; set; }
public Pagestatus status;
public LIRSnode(){
        status = Pagestatus.NonResident_HIR;
        Data = null;
        prev = null;
        next = null;
    }
Public LIRSnode(Pagestatusstatuss, LIRSnodedatavalue): this(statuss, null, datavalue,
null){  }
public LIRSnode(Pagestatusstatuss, LIRSnodepreviousnode, LIRSnodedatavalue,
LIRSnodenextnode) {
        status = statuss;
        prev = previousnode;
        Data = datavalue;
        next = nextnode;
        }
  }
```

3.2 LIRS algorithm

LIRS algorithm takes into consideration the Inter reference recency set (IRR) of pages as the dominant function of eviction which distinguishes reference blocks into high IRR and low IRR. The number of LIR and HIR pages is chosen such that all LIR pages and small percentage of HIR pages kept in the cache. In the same manner, the cache is also partitioned into two partitions namely C_{LIRS} and C_{HIRS}. Where, C_{LIRS} contains the set of LIR blocks having a higher probability of being referencing in near future and re-accessing these blocks guaranteed to be hit. C_{HIRS} contains set of HIR blocks having less probability of being reference in future and accessing these blocks encounters page fault [7].

The policy functions as follows: when the *recency* of LIR block increases at any point and block of minimal IRR value get accessed from C_{HIRS}, then the status of these two blocks get interchanged. This means, a block with minimal HIR block promoted to C_{LIRS} and a block with higher *recency* demoted to C_{HIRS}. Initially, C_{LIRS} and C_{HIRS} are empty. Newly accessed blocks are admitted to C_{LIRS} then Resident (RC_{Hirs}) until there is not free slot in the cache. On accessing a block that already classified into C_{LIRS} encounters a cache hit and IRR value of accessed block becomes its earlier Recency. Similarly, on accessing a HIR block, there almost occurs cache miss because of smaller size of HIR block set. At this time, the status of a block might be interchanged. While changing status, following condition must be true.

$$Max(\text{Re}\,cency\ of\ \ LIR\ Block) > (\ New\,Born\ IRR\ of\ \ HIR\ Block)$$

If the maximum recency of LIR block is greater than the IRR of currently accessed HIR block, then the currently accessed resident block should be admitted to the C_{Lirs} and a block having maximum Recency in C_{Lirs} demoted to RC_{Hirs}. And currently accessed Non-resident block may be switched into LIR or Resident block based on their IRR value and block having maximum Recency may be demoted to Resident or non-resident block based on the recency. Otherwise existed block are unchanged. In every case of status change, the pruning operation is performed which is described in section 3.2.1. The algorithm placed in section 3.2.2 describes the complete policy of our model.

3.2.1 Pruning Operation

Pruning operation is performed during status change to minimize the cache overhead. While referencing new blocks, their history information is stored on list. But at some point, its size grows unexpectedly very large which makes cache polluted. As per the algorithm, the list contains metadata for the block with their recency less than the max recency of LIR block. The entire non-resident block having recency greater than the max recency of LIR block is removed one by one. After removing these unnecessarily used block decreases the size of list hence the list doesn't keep track of outdated references.

3.2.2 Algorithm

X- Requested page

Begin

Case A: If X is in Linked list.

Move X to the head of list.

Update Recency and set IRR value of X to its earlier recency

Case a: If X ϵ $C_{Lirs,}$ (encounters hit.)

Perform Pruning operation.

Case b: If X ϵ RC_{Hirs} U $NRC_{Hirs,}$

Case 1: If Max Recency of LIR block >New born IRR of HIR block

Case I: If X ϵ RC_{Hirs} (encounters hit.)

Replace X with a block having Max Recency value in C_{Lirs}

Perform Pruning operation.

Case II: Else X ϵ NRC_{Hirs} (encounters miss);

A block having min HIR switched with resident block having high recency.

A block having min HIR switched with LIR block having high recency.

Perform Pruning operation.

Case 2: Else Replace X with a block having max Recency value in RC_{Hirs}

Case B: Else X is not in Linked list.

Case a: If CLirs & $RC_{Hirs} = \Phi$

Insert X into C_{Lirs} then into RC_{Hirs}

Case b: Else C_{Lirs} & $RC_{Hirs} \neq \Phi$

Replace X with a block having max Recency value in RC_{Hirs}

End

3.2.3 Flowchart

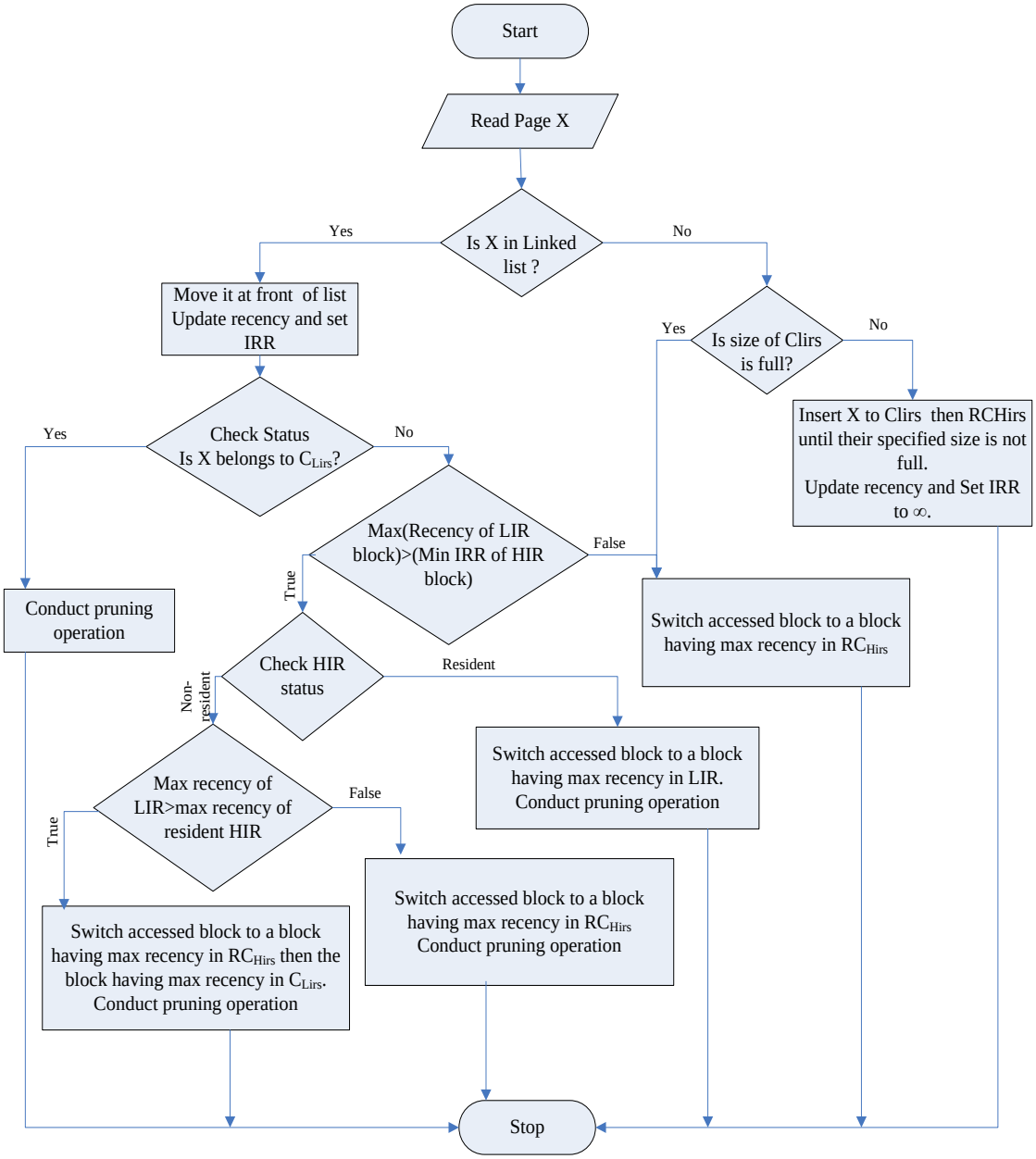

Figure 3.2 Flowchart of LIRS Algorithm

3.2.4 Tracing

Input Reference: 5 6 7 8 6 8 7 4 7 4 6

Cache Size: 3

Size of $C_{Lirs:}$ 2

Size of $C_{Hirs:}$ 1

On accessing a block 5

C_{Lirs}= {5}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	Recency	IRR
5	A		0	∞

Page fault.

Table 3.1.1 State at Virtual Time

On accessing a block: 6

C_{Lirs}= {5, 6}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	Recency	IRR
5	A			1	∞
6		A		0	∞

Page fault.

Table 3.1.2 State at Virtual Time 2

On accessing a block: 7

C_{Lirs}= {5, 6}

C_{Hirs} = {7}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	4	Recency	IRR
5	A				2	∞
6		A			1	∞
7			A		0	∞

Page fault.

Table 3.1.3 State at Virtual Time 3

On accessing a block: 8

C_{Lirs}= {5, 6}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	Recency	IRR
5	A					3	∞
6		A				2	∞
7			A			1	∞
8				A		0	∞

Page fault

Table 3.1.4 State at Virtual Time 4

On accessing a block: 6

C_{Lirs}= {6, 5}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	Recency	IRR
5	A						3	∞
6		A			A		0	2
7			A				2	∞
8				A			1	∞

Table 3.1.5 State at Virtual Time 5

On accessing a block: 8

C_{Lirs}= {8, 6}

C_{Hirs} = {5}

RC_{Hirs}= {5}

NRC_{Hirs}= {}

	1	2	3	4	5	6	Recency	IRR
5	A						3	∞
6		A		A			1	2
8			A		A		0	1

Status Change

Pruning operation also performed.

Table 3.1.6 State at Virtual Time 6

On accessing a block: 7

C_{Lirs}= {8, 6}

C_{Hirs} = {7}

RC_{Hirs}= {7}

NRC_{Hirs}= {}

	1	2	3	4	5	6	Recency	IRR
6	A		A				2	2
8		A		A			1	1
7					A		0	∞

Page fault and Pruning operation also performed

Table 3.1.7 State at Virtual Time 7

On accessing a block: 4

C_{Lirs}= {8, 6}

C_{Hirs} = {4, 7}

RC_{Hirs}= {4}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	7	Recency	IRR
5	A		A					3	2
6		A		A				2	1
4						A		0	∞
7					A			1	∞

Page fault

Table 3.1.8 State at Virtual Time 8

On accessing a block: 7

C_{Lirs}= {7, 6}

C_{Hirs} = {6, 4}

RC_{Hirs}= {6}

NRC_{Hirs}= {4}

	1	2	3	4	5	6	7	8	Recency	IRR
5	A		A						3	2
6		A		A					2	1
4						A			1	∞
7					A		A		0	1

Page fault

Table 3.1.9 State at Virtual Time 9

Total Number of Page fault: 7

3.3 Revised LIRS

This Revised LIRS algorithm modifies LIRS algorithm such that the data structure holds all LIR blocks and all the metadata of uncached block by avoiding pruning operations. Instead of using new born IRR value of HIR block, it uses minimum IRR value of HIR block for comparison with maximum recency of LIR block while changing page status. Like in LIRS algorithm, here two operations are possible- insertion of new accessed block to the data structure and moving on existed data blocks to the head of list. LIRS algorithm with minimum IRR tackles the problem of LIRS algorithm by maintaining all data in the cache. But during status change, this algorithm must perform some extra operation. Its behaviors and all operation can be described as:

On accessing a block that already classified into C_{Lirs} encounters a cache hit. The accessed block is moved to the head of list and recency value is updated based on their position. IRR value of accessed block becomes its earlier Recency.

On accessing a block that is already available in RC_{Hirs}, it encounter page hit and compares maximum recency of LIR block with minimum IRR value of HIR block for promotion. In such case, there is not only accessed resident block has a chance of promotion. Because it depends on the minimal value of RC_{Hirs} U NRC_{Hirs}. If the minimal value block is accessed block then accessed block promoted otherwise minimal value block gets promotion.

On accessing a block that is already classified on NRC_{Hirs}, encounters page fault. The algorithm keeps that block to the cache by comparing max recency of LIR block with minimum IRR of HIR block. Like RC_{Hirs} block, if the minimal value block is accessed block then accessed block is promoted otherwise a block having minimum IRR value and accessed block gets promotion altogether.

3.3.1 Algorithm

X- Requested page

Begin

Case A: If X is in Linked List

Move to the head of the list and update IRR to its earlier recency

Case a: If X ϵ $C_{Lirs,}$ hit occur.

Case b: X ϵ RC_{Hirs} U $NRC_{Hirs,}$

Case 1: If Max (Recency of LIR block) >Min (IRR value of HIR block)

Case I: If Min IRR block is RC_{Hirs}

Promote X to C_{Lirs} and demote Max Recency value block to RC_{Hirs}

Case II: If Min IRR block is accessed block and is in NRC_{Hirs}

Promote Min IRR value block and Demote Max Recency block to RC_{Hirs} or NRC_{Hirs} based on Recency

Case III: Else switch accessed block with a block having max recency on $RC_{Hirs.}$

Case 2: Else switch accessed block with a block having max recency on $RC_{Hirs.}$

Case B: Else X is not in linked list

Case a: If C_{Lirs} & $RC_{Hirs} = \Phi$

Insert X into C_{Lirs} then into RC_{Hirs}

Case a: If C_{Lirs} & $RC_{Hirs} \neq \Phi$

Case 1: If Max (Recency of LIR block) >Min (IRR value of HIR block)

Case I: If Min IRR block is RC_{Hirs}

Promote X to C_{Lirs} and demote Max Recency value block to RC_{Hirs}

Case II: If Min IRR block is accessed block and is in NRC_{Hirs}

Promote Min IRR value block and Demote Max Recency block to RC_{Hirs} or NRC_{Hirs} based on Recency

Insert accessed block to the RC_{Hirs} and demote resident block having maximum recency

End

3.3.2 Flowchart

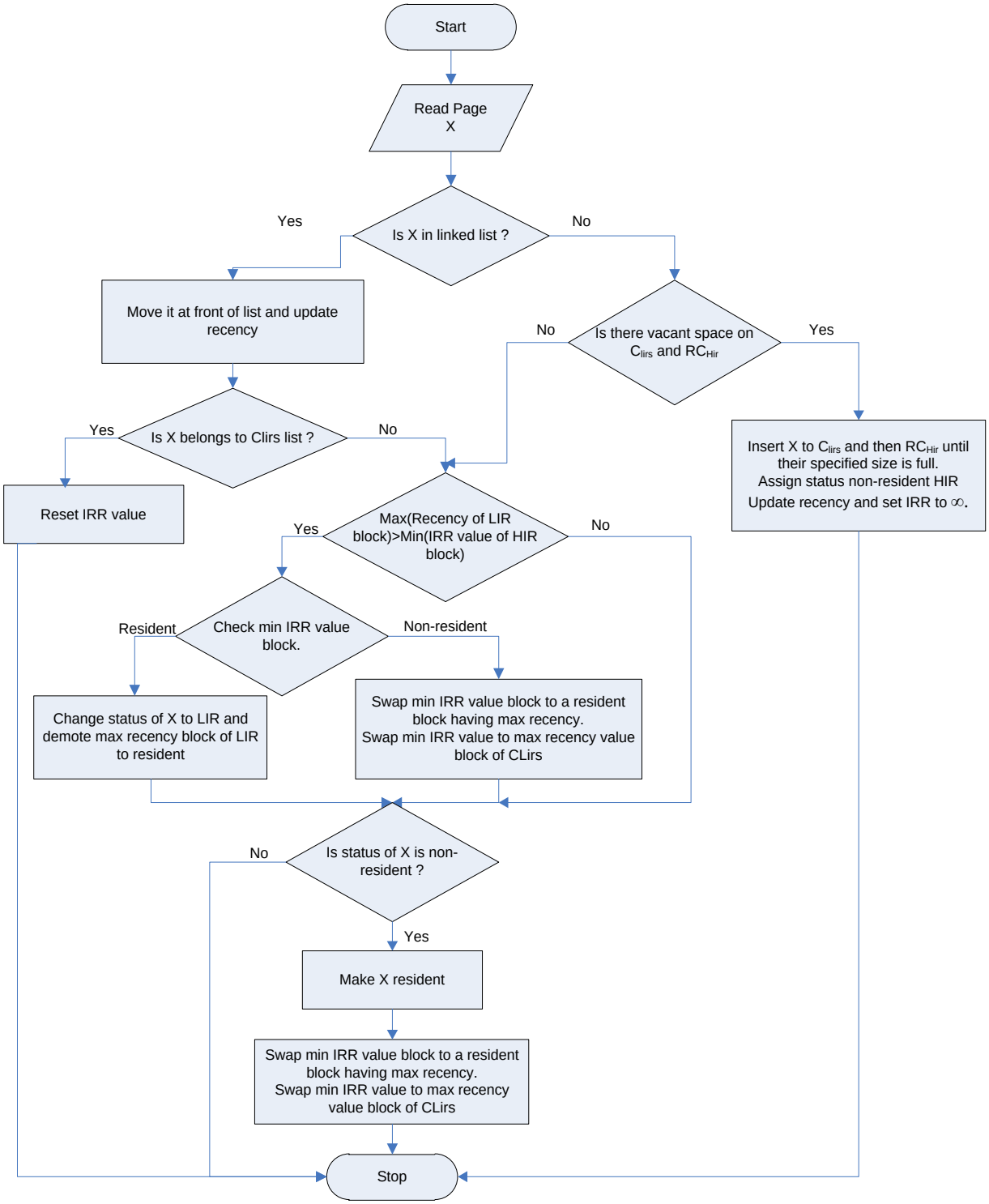

Figure 3.3 Flowchart of Revised LIRS Algorithm

3.3.3 Tracing

Input Reference: 5 6 7 8 6 8 7 4 7

Cache Size: 3

Size of $C_{Lirs:}$ 2

Size of $C_{Hirs:}$ 1

On accessing a block 5

C_{Lirs}= {5}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	Recency	IRR
5	A		0	∞

Page fault.

Table 3.2.1 State at Virtual Time 1

On accessing a block: 6

C_{Lirs}= {5, 6}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	Recency	IRR
5	A			1	∞
6		A		0	∞

Page fault.

Table 3.2.2 State at Virtual Time 2

On accessing a block: 7

C_{Lirs}= {5, 6}

C_{Hirs} = {7}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	4	Recency	IRR
5	A				2	∞
6		A			1	∞
7			A		0	∞

Page fault.

Table 3.2.3 State at Virtual Time 3

On accessing a block: 8

C_{Lirs}= {5, 6}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	Recency	IRR
5	A					3	∞
6		A				2	∞
7			A			1	∞
8				A		0	∞

Page fault

Table 3.2.4 State at Virtual Time 4

On accessing a block: 6

C_{Lirs}= {6, 5}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	Recency	IRR
5	A						3	∞
6		A			A		0	2
7			A				2	∞
8				A			1	∞

Table 3.2.5 State at Virtual Time 5

On accessing a block: 8

C_{Lirs}= {8, 6}

C_{Hirs} = {5, 7}

RC_{Hirs}= {5}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	7	Recency	IRR
5	A							3	∞
6		A			A			1	2
7			A					2	∞
8				A		A		0	1

Table 3.2.6 State at Virtual Time 6

On accessing a block: 7

C_{Lirs}= {8, 6}

C_{Hirs} = {7, 5}

RC_{Hirs}= {7}

NRC_{Hirs}= {5}

	1	2	3	4	5	6	7	8	Recency	IRR
5	A								3	∞
6		A			A				2	2
7			A				A		0	2
8				A		A			1	1

Page fault

Table 3.2.7 State at Virtual Time 7

On accessing a block: 4

C_{Lirs}= {7, 8}

C_{Hirs} = {4, 6, 5}

RC_{Hirs}= {4}

NRC_{Hirs}= {6, 5}

	1	2	3	4	5	6	7	8	9	Recency	IRR
5	A									4	∞
6		A			A					3	2
7			A				A			1	2
8				A		A				2	1
4								A		0	∞

Page Fault

Table 3.2.8 State at Virtual Time 8

On accessing a block: 7

C_{Lirs}= {7, 8}

C_{Hirs} = {4, 6, 5}

RC_{Hirs}= {4}

NRC_{Hirs}= {6, 5}

	1	2	3	4	5	6	7	8	9	10	Recency	IRR
5	A										4	∞
6		A			A						3	2
7			A				A		A		0	1
8				A		A					2	1
4								A			1	∞

Table 3.2.9 State at Virtual Time 9

Total Number of Page fault: 6

3.4 Derived LIRS

This approach modifies and improves standard LIRS algorithm in such a way that it can store history information of newly accessed block and frequently accessed block. The history information of newly accessed block is cached by comparing with max recency of C_{Lirs} block set and frequently accessed block is cached by comparing min IRR of all accessed block with max recency of C_{Lirs} block. Like Revised LIRS algorithm, it keeps metadata of all accessed C_{Hirs} block and avoids pruning operation. But during status change, at first it compare new born IRR of accessed block to the max recency of C_{Lirs} then minimum value of C_{Hirs} compares with max recency of C_{Lirs} if the first condition is false. Thus, the algorithm must perform some extra operation. Its behaviors and all operation can be described as:

Upon accessing a block that already classified into C_{Lirs} encounters a cache hit. The accessed block is moved to the head of list and recency value is updated based on their position. IRR value of accessed block becomes its earlier Recency.

When accessing RC_{Hirs} block encounters hit and gets new IRR equal to its recency. This new IRR value of accessed block compares with maximum recency of LIR block. If the new born IRR is found greater than the max recency of CLirs, then the accessed block gets promotion. Otherwise, max recency of CLirs is compared with evaluated minimum IRR of all CHirs block. Because it depend on the minimal value of RC_{Hirs} U NRC_{Hirs}. If the minimal value block is accessed block then accessed block promoted otherwise minimal value block and accessed block gets promotion altogether.

On accessing a block that is already classified on NRC_{Hirs}, it encounters page fault. Like RC_{Hirs} block, the maximum recency of LIR is compared with the new accessed non-resident block at first. If the maximum recency found greater, then accessed block swapped to RCHirs or NRC_{Hirs} based on the recency. Otherwise, the maximum recency of LIR is compared with the minimum IRR of HIR block. If the minimal value block is accessed block then accessed block is promoted otherwise a block having minimum IRR value and accessed block gets promotion altogether.

3.4.1 Algorithm

X-Requested page

Begin

Case A: If X is found in Linked list

Move X to front of list

Set recency and update IRR value by its earlier recency

Case a: If X has LIR status

Hit occurs in the cache and do nothing

Case b: Else if X ϵ RC_{Hirs} U $NRC_{Hirs,}$

Case 1: If max recency of LIR> New born IRR of accessed block

Case I: If accessed block is resident

Accessed block is switched with a block having max Recency value of LIR

Case II: Else if accessed block is non-resident

Accessed page switched with resident block having max recency.

LIR page with max recency is switched with accessed page.

Case 2: Else If Max Recency of LIR>min IRR of HIR block

Check the status of the block having min IRR

Case I: If the min IRR block has status resident

Switch Min IRR block to block having maximum recency in LIR block set

Case II: If Min IRR block is accessed block and is in NRC_{Hirs}

A block having min HIR switched with resident block having high recency.

A block having min HIR switched with LIR block having high recency.

Case 3: If accessed block is in NRC_{Hirs}.

Switch accessed block with a block having max recency on $RC_{Hirs.}$

Case B: Else X is not in linked list

Case a: If C_{Lirs} & $RC_{Hirs} = \Phi$

Insert X into C_{Lirs} then into RC_{Hirs}

Case b: If C_{Lirs} & $RC_{Hirs} \neq \Phi$

Case 1: If Max (Recency of LIR block) >Min (IRR value of HIR block)

Case I: If Min IRR block is RC_{Hirs}

Promote X to C_{Lirs} and demote Max Recency value block to RC_{Hirs}

Case II: If Min IRR block is accessed block and is in NRC_{Hirs}

Promote Min IRR value block and Demote Max Recency block to RC_{Hirs} or NRC_{Hirs} based on Recency

Insert accessed block to the RC_{Hirs} and demote resident block having maximum recency

End

3.4.2 Flowchart

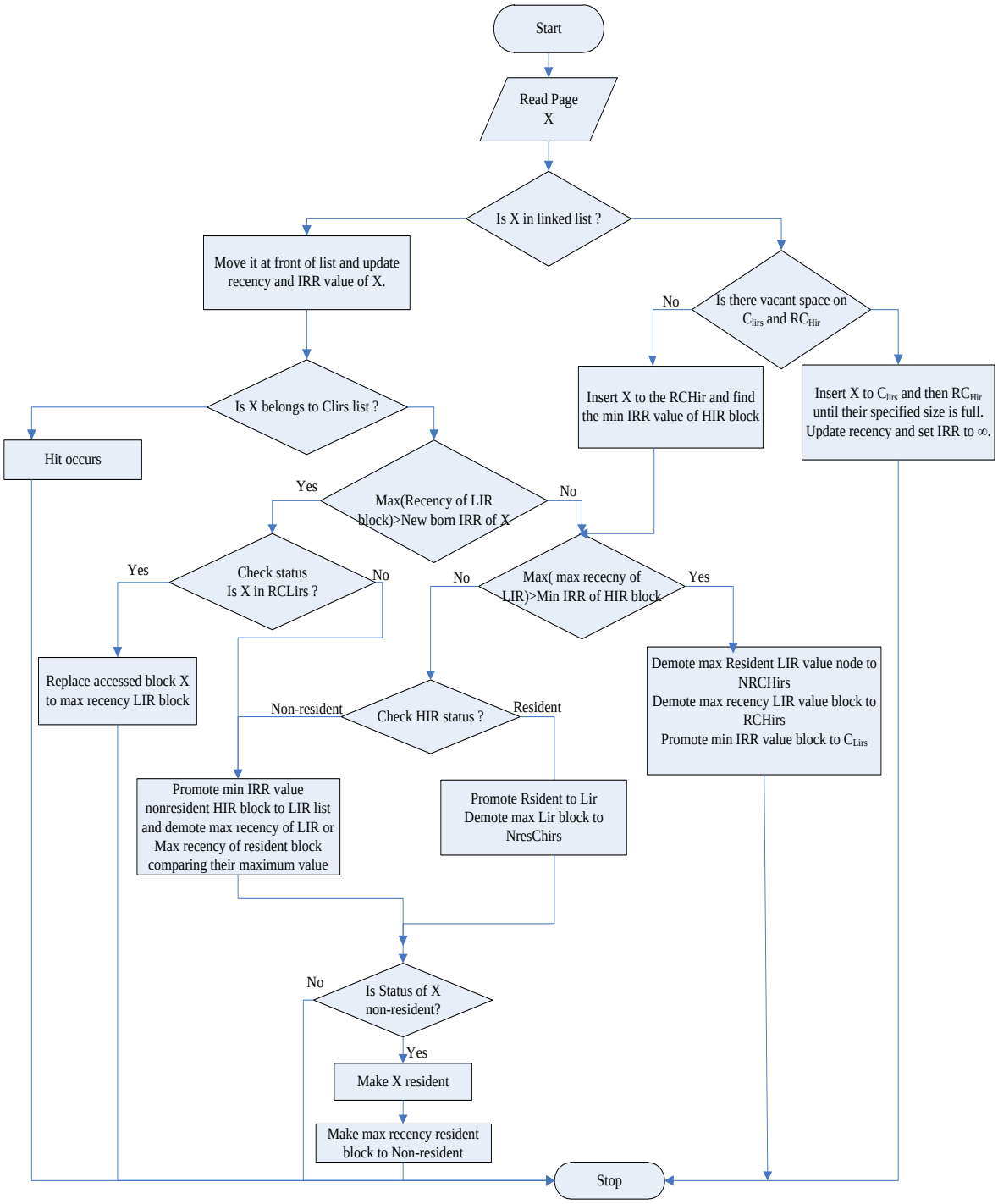

Figure 3.4 Flowchart of Derived LIRS Algorithm

3.4.3 Tracing

Input Reference: 5 6 7 8 6 8 7 4 7

Cache Size: 3

Size of $C_{Lirs:}$ 2

Size of $C_{Hirs:}$ 1

On accessing a block 5

C_{Lirs}= {5}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	Recency	IRR
5	A		0	∞

Page fault.

Table 3.3.1 State at Virtual Time 1

On accessing a block: 6

C_{Lirs}= {5, 6}

C_{Hirs} = {}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	Recency	IRR
5	A			1	∞
6		A		0	∞

Page fault.

Table 3.3.2 State at Virtual Time 2

On accessing a block: 7

C_{Lirs}= {5, 6}

C_{Hirs} = {7}

RC_{Hirs}= {}

NRC_{Hirs}= {}

	1	2	3	4	Recency	IRR
5	A				2	∞
6		A			1	∞
7			A		0	∞

Page fault.

Table 3.3.3 State at Virtual Time 3

On accessing a block: 8

C_{Lirs}= {5, 6}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	Recency	IRR
5	A					3	∞
6		A				2	∞
7			A			1	∞
8				A		0	∞

Page fault

Table 3.3.4 State at Virtual Time 4

On accessing a block: 6

C_{Lirs}= {6, 5}

C_{Hirs} = {8, 7}

RC_{Hirs}= {8}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	Recency	IRR
5	A						3	∞
6		A			A		0	2
7			A				2	∞
8				A			1	∞

Table 3.3.5 State at Virtual Time 5

On accessing a block: 8

C_{Lirs}= {8, 6}

C_{Hirs} = {5, 7}

RC_{Hirs}= {5}

NRC_{Hirs}= {7}

	1	2	3	4	5	6	7	Recency	IRR
5	A							3	∞
6		A			A			1	2
7			A					2	∞
8				A		A		0	1

Table 3.3.6 State at Virtual Time 6

On accessing a block: 7

C_{Lirs}= {7, 6}

C_{Hirs} = {8, 5}

RC_{Hirs}= {8}

NRC_{Hirs}= {5}

	1	2	3	4	5	6	7	8	Recency	IRR
5	A								3	∞
6		A			A				2	2
7			A				A		0	2
8				A		A			1	1

Page fault

Table 3.3.7 State at Virtual Time 7

On accessing a block: 4

C_{Lirs}= {7, 6}

C_{Hirs} = {4, 8, 5}

RC_{Hirs}= {4}

NRC_{Hirs}= {8, 5}

	1	2	3	4	5	6	7	8	9	Recency	IRR
5	A									4	∞
6		A			A					3	2
7			A				A			1	2
8				A		A				2	1
4								A		0	∞

Page Fault

Table 3.3.8 State at Virtual Time 8

On accessing a block: 7

C_{Lirs}= {7, 6}

C_{Hirs} = {4, 8.5}

RC_{Hirs}= {4}

NRC_{Hirs}= {8, 5}

	1	2	3	4	5	6	7	8	9	10	Recency	IRR
5	A										4	∞
6		A			A						3	2
7			A				A		A		0	1
8				A		A					2	1
4								A			1	∞

Table 3.3.9 State at Virtual Time 9

Total Number of Page fault: 6

CHAPTER 4
DATA COLLECTION AND ANALYSIS

To analyze the impact of pruned metadata on LIRS algorithm, three cases of the LIRS have been taken. All these three algorithms are implemented as offline replacement and demand paging policy. In this analysis, offline performance is measured and overhead analysis is ignored. Offline performance of replacement algorithm in these cases is measured as page fault count, hit rate and miss rate. The number of string with different patterns which is called trace data, are used as a real world performance indicator. There is different number of page references in each of these traces. In every cases cache size is maintained 1% for HIR and 99% for LIR. Each trace is tested in all three cases of LIRS simulator by varying the cache size from 4 to 1024.

4.1 Trace Data

Trace data are the reference string and consist of page reference to virtual address space. In this study, these reference strings are considered main sources for evaluating the impact of metadata on LIRS algorithm. Four types of traces pattern namely looping pattern, probabilistic pattern, temporally clustered and mixed pattern are used in this dissertation work which are mentioned in [7]. Here, ***cs*** and ***glimpse*** belongs to cyclic pattern, ***cpp*** belongs to probabilistic reference pattern, ***sprite*** belongs to temporally clustered pattern and ***multi1*** and ***multi2*** belongs to mixed reference pattern. These traces are considered typical and representation of application in that most of them is routinely used in other caching algorithm studies [7].

4.2 Testing and Analysis

The impact of pruned metadata on LIRS algorithm is analyzed on different patterns of traced data. To understand the impact of metadata, the hit ratio is used as measure factor to analyze the algorithm. Following graph and tables show the results and analysis of this study.

4.2.1 Replacement Performance on Looping Type Reference Pattern

As mentioned earlier this type of reference pattern contains ***cs*** and ***glimpse*** pattern which are analyzed individually for each flavors of LIRS algorithm. Traces cs has pure looping reference pattern where each block of reference pattern are almost accessed repeatedly within fixed interval where as glimpse has mixed looping pattern. The test result and corresponding graph is shown in Table 4.1 and figure 4.1.

4.2.1.1 CS Pattern

These are the relevant data with this traced pattern.

Total Number of References=6781

Number of distinct references=1409

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	6662	97.78%	2.22%	6662	97.78%	2.22%	6664	97.82%	2.18%
8	6660	97.74%	2.26%	6660	97.74%	2.26%	6660	97.74%	2.26%
16	6656	97.67%	2.33%	6660	97.74%	2.26%	6656	97.67%	2.33%
32	6629	97.17%	2.83%	6659	97.72%	2.28%	6629	97.17%	2.83%
64	6517	95.08%	4.92%	6567	96.01%	3.99%	6519	95.12%	4.88%
128	6263	90.35	9.65%	6316	91.34%	8.66%	4925	65.45%	34.55%
256	5757	80.93%	19.07%	5815	82.01%	17.99%	3636	41.45%	58.55%
512	4755	62.28%	37.72%	4795	63.03%	36.97%	2934	28.38%	71.62%
1024	2733	24.64	75.36%	2761	25.16%	74.84%	2064	12.19%	87.81%

Table 4.1: Result for CS Pattern of Reference Pages

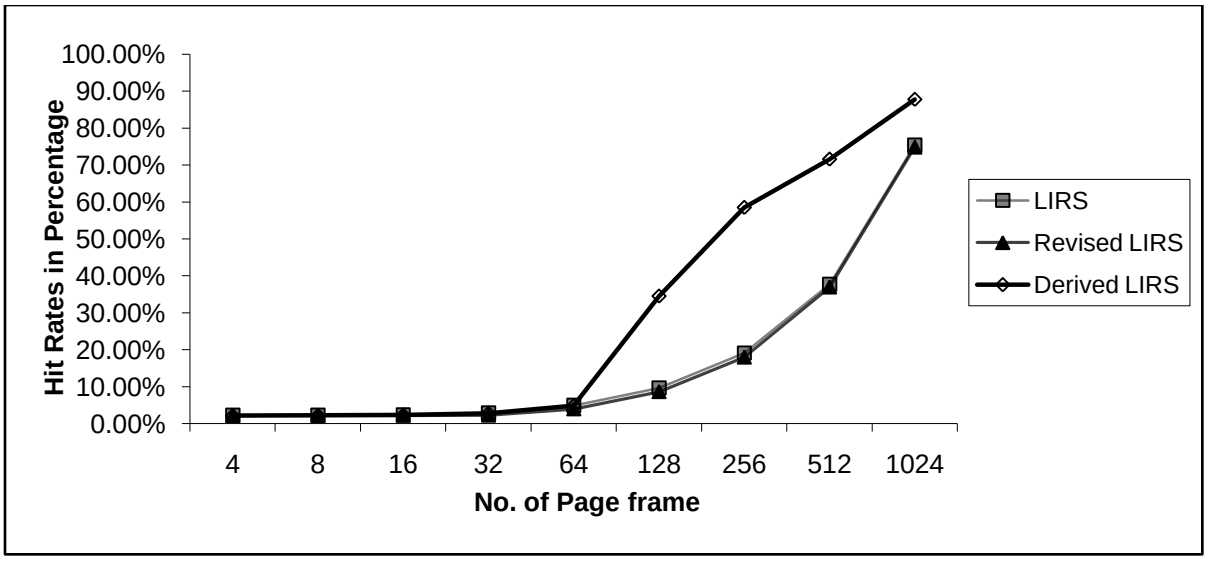

Figure 4.1: Graph Showing Result for CS Pattern of Reference Pages

4.2.1.2 Glimpse Pattern

Total Number of References=6016

Number of distinct references=2530

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	5956	98.27%	1.73%	5949	98.07%	1.93%	5956	98.27%	1.73%
8	5928	97.47%	2.53%	5934	97.64%	2.36%	5930	97.53%	2.47%
16	5904	96.78%	3.22%	5910	96.95%	3.05%	5906	96.84%	3.16%
32	5851	95.26%	4.74%	5857	95.43%	4.57%	5855	95.38%	4.62%
64	5723	91.59%	8.41%	5729	91.76%	8.24%	5727	91.70%	8.30%
128	5471	84.36%	15.64%	5470	84.33%	15.67%	5351	80.92%	19.08%
256	4963	69.79%	30.21%	4962	69.76%	30.24%	4716	62.70%	37.30%
512	3951	40.76%	59.24%	3950	40.73%	59.27%	3855	38.00%	62.00%
1024	2942	11.81%	88.19%	2942	11.81%	88.19%	2943	11.84%	88.16%

Table 4.2: Result for Glimpse Pattern of Reference Pages

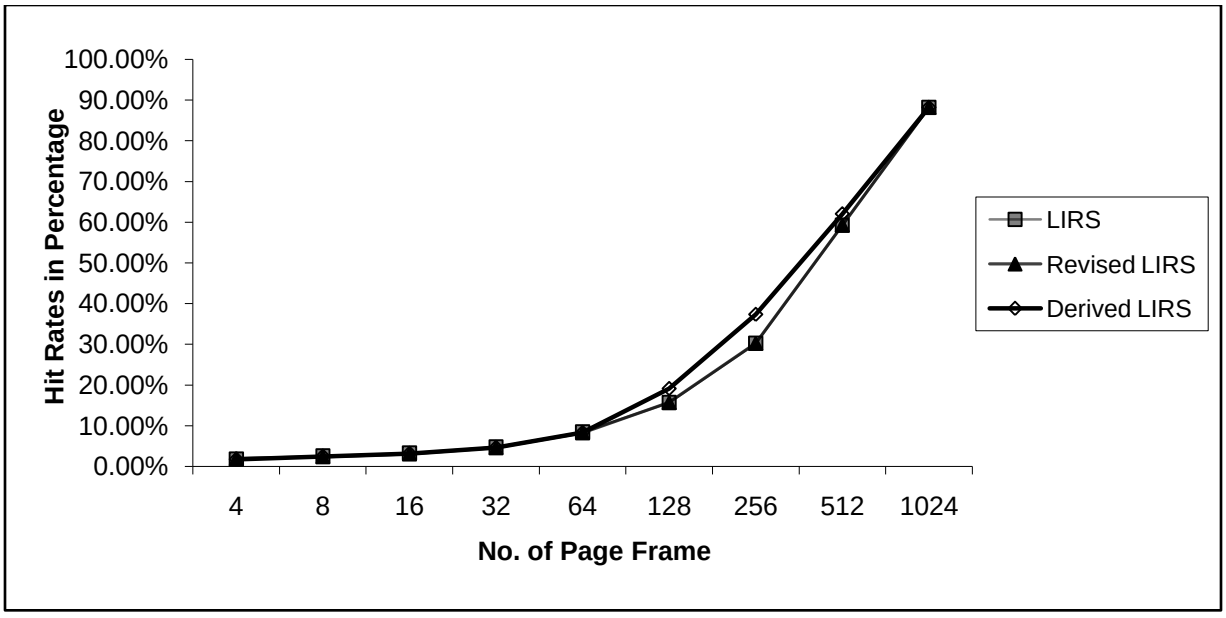

Figure 4.2: Graph Showing Result for Glimpse Pattern of Reference Pages

From fig. 4.1and 4.2, it can be concluded that the performance of LIRS, Revised LIRS and Derived LIRS have similar performance on both traces before the cache size reaches to 100. After the cache size increases to 100 page frames, Derived LIRS shows improved performance. In fact, the performance improvement is shown up to 39.48% on cs trace. The reason behind the improvement is to have maximum number of page frames than the number of block included in cyclic pattern. And another reason is the ability to hold the information of newly accessed block and minimum IRR blocks which help to make accurate prediction for future reference pages. On glimpse pattern, the performance is slightly improved. This is because the trace contains mixed looping pattern.

4.2.2 Replacement Performance on Probabilistic Type Reference Pattern

Traces, cpp exhibits probabilistic pattern. The test result and its corresponding graph is shown below.

Total Number of References=9047

Number of distinct references=1223

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	8695	95.50%	4.50%	9019	99.64%	0.36%	8921	98.38%	1.62%
8	8223	89.46	10.54%	8640	94.79%	5.21%	8604	94.33%	5.67%
16	7275	77.35	22.65%	8269	90.05%	9.95%	7741	83.30%	16.70%
32	5390	53.25	46.75%	6780	71.02%	28.98%	5818	58.72%	41.28%
64	3034	23.14	76.86%	4062	36.28%	63.72%	3137	24.46%	75.54%
128	1707	6.18	93.82%	2880	21.17%	78.83%	1333	1.40%	98.60%
256	1377	1.96	98.04%	1781	7.13%	92.87%	1236	0.52%	99.48%
512	1275	0.66	99.34%	1321	1.25%	98.75%	1264	0.16%	99.84%
1024	1228	0.06	99.94%	1232	0.11%	99.89%	1228	0.06%	99.94%

Table 4.3: Result for Cpp Pattern of Reference Pages

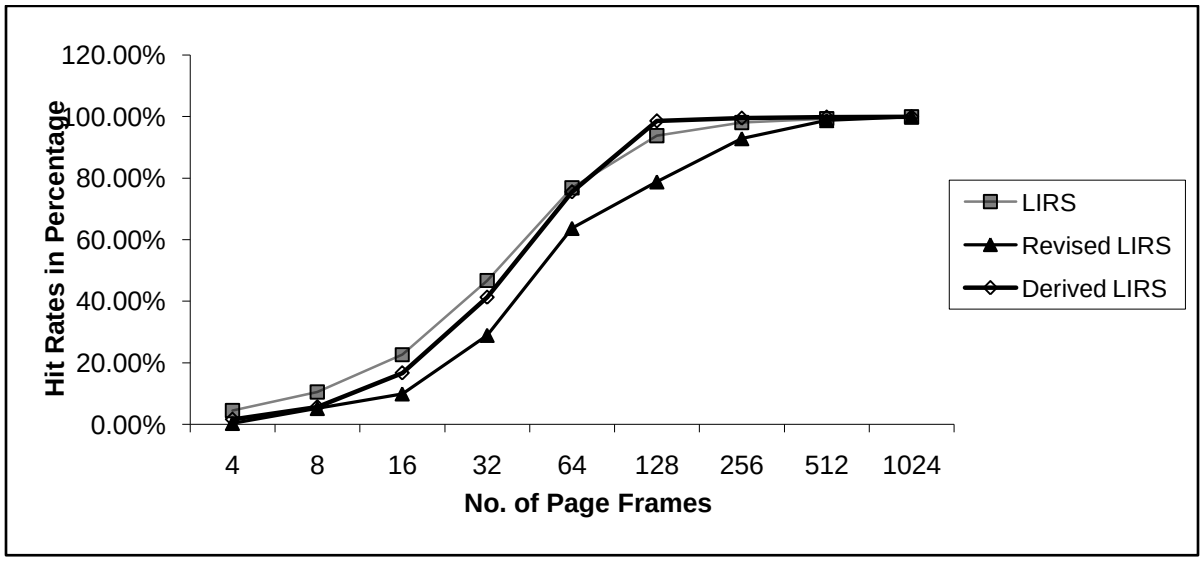

Figure 4.3: Graph Showing Result for cpp Pattern of Reference Pages

The hit rate curve generated for cpp reference pattern is shown in Figure 4.3. Before the cache size reaches 100 page frames, the performance of LIRS is better than Derived and Revised LIRS; this is due to the small cache size. In such case, the history of newly accessed block has fixed in the cache. However, the Revised LIRS shows poorer performance than

others because it replaces all newly referenced blocks after holding minimum IRR value blocks in the buffer cache for a short time. After cache size increases to 100 page frames, Derived LIRS is able to exploit good locality by holding the newly referenced blocks and frequently accessed blocks. Aftereffect, it gives better performance than LIRS and Revised LIRS.

4.2.3 Replacement Performance on Temporally Clustered Type Reference Pattern

Trace, ***sprite*** exhibits temporally clustered pattern. Where, pages are referenced more than the cache size with long interval. The hit rate curve and test result generated by LIRS and other cases of LIRS algorithm is shown Table 4.4 and Figure 4.4 respectively.

Total Number of References=133996

Number of distinct references=7075

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	129192	96.21%	3.79%	130857	97.52%	2.48%	128861	95.95 %	4.05 %
8	128662	95.79%	4.21%	130614	97.33%	2,67%	128289	95.50 %	4.50 %
16	127676	95.02%	4.98%	129925	96.79%	3.21%	127291	94.71 %	5.29 %
32	124232	92.30%	7.70%	128055	95.31%	4.69%	123314	91.58 %	8.42 %
64	113282	83.67%	16.33 %	123375	91.63%	8.37%	107328	78.98 %	21.02 %
128	91104	66.20%	33.80 %	110044	81.12%	18.88 %	57845	40.00 %	60.00 %
256	62650	43.78%	56.22 %	88991	64.54%	35.46 %	109639	19.19 %	80.81 %
512	30523	18.47%	81.53 %	66547	46.85%	53.15 %	8101	0.80%	99.20 %
1024	15741	6.82%	93.18 %	41229	26.90%	73.10 %	8031	0.75%	99.25 %

Table 4.4: Result for sprite Pattern of Reference Pages

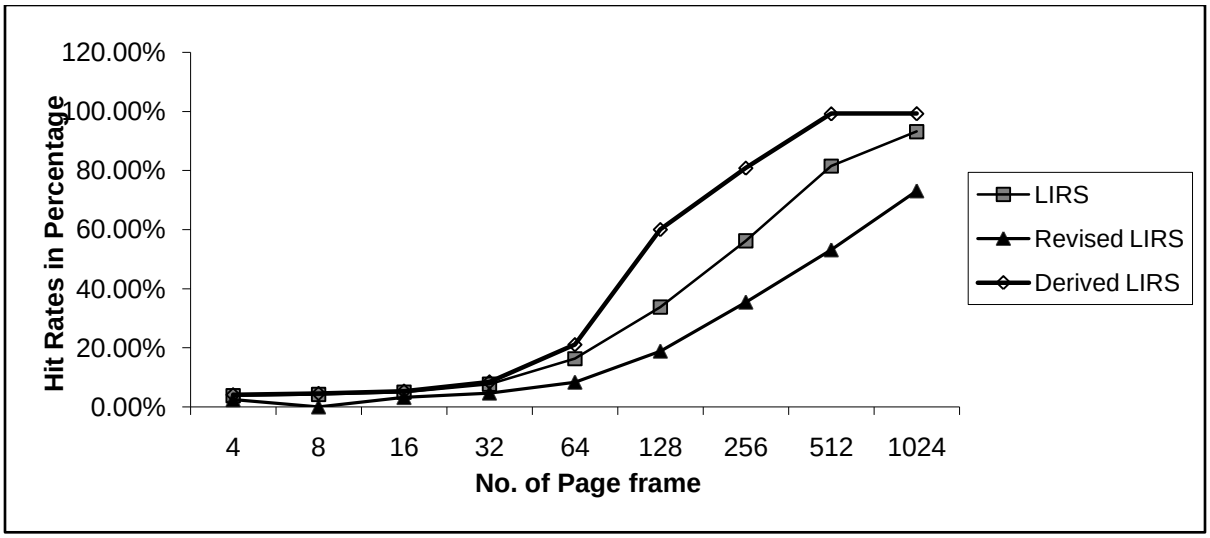

Figure 4.4: Graph Showing Result for sprite Pattern of Reference Pages.

The hit rate curve generated for Sprite reference pattern is shown in Figure 4.4.The Derived LIRS shows better than other cases. The property of temporally clustered reference pattern is that blocks accessed more recently are the likely to be accessed in the future. In such case, the Derived LIRS algorithm holds all these frequently accessed blocks and newly accessed blocks are evicted after short time. This is the main reason to have better performance than the LIRS. In case of LIRS algorithm, it keeps the newly accessed block in the cache, that's why the performance of LIRS degrades. Revised LIRS has the poorer performance because minimal IRR value occupied in the cache can't get entry newly accessed block for long time.

4.2.4 Replacement Performance on Mixed Type Reference Pattern

Traces, multi1 and multi2 exhibits mixed reference pattern. Multi1 is obtained by executing Cs and Cpp and Multi2 is obtained by executing Cs, Cpp and Postgress together. The test result and Performance of LIRS, Revised LIRS and Derived LIRS on Multi1 and Multi2 traces with different cache sizes are shown in following table and figures.

4.2.4.1 Multi1 Pattern

Total Number of References=15858

Number of distinct references=2606

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	15547	97.65%	2.35%	15836	99.83%	0.17%	15716	98.92%	1.08%
8	15115	94.39%	5.61%	15700	98.80%	1.20%	15458	96.98%	3.02%
16	14269	88.00%	12.00%	15309	95.85%	4.15%	14803	92.03%	7.97%
32	12459	74.35%	25.65%	14513	89.85%	10.15%	13346	81.04%	18.96%
64	9892	54.98%	45.02%	12936	77.95%	22.05%	10399	58.80%	41.20%
128	8540	44.77%	55.23%	11165	64.58%	35.42%	5442	21.40%	78.60%
256	8028	40.91%	59.09%	9253	50.15%	49.85%	3758	8.69%	91.31%
512	6990	33.11%	66.89%	8125	41.64%	58.36%	3514	6.85%	93.15%
1024	4933	17.55%	82.45%	5410	21.15%	78.85%	2968	2.73%	97.27%

Table 4.5: Result for multi1Pattern of Reference Pages

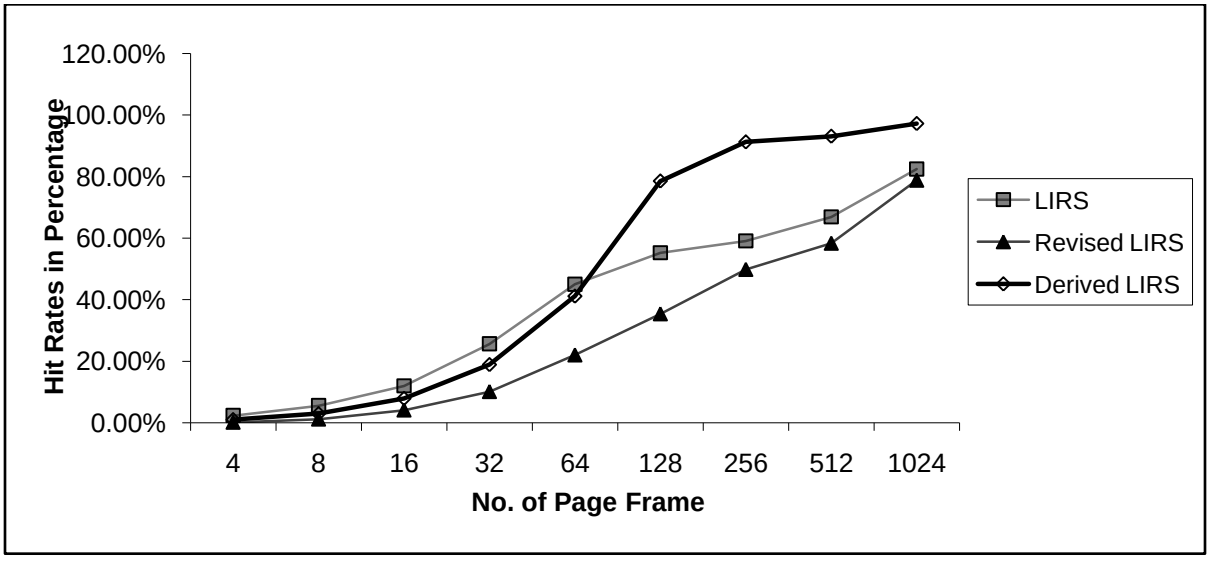

Figure 4.5: Graph Showing Result for Multi1Pattern of Reference Pages

4.2.4.2 Multi2 Pattern

Total Number of References=25972

Number of distinct references=5495

No. of page frames	LIRS			Revised LIRS			Derived LIRS		
	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate	Page Fault	Miss Rate	Hit Rate
4	25423	97.31	2.69%	25909	99.69%	0.31%	25638	98.36%	1.64%
8	24878	94.65	5.35%	25648	98.41%	1.59%	25325	96.84%	3.16%
16	24011	90.42	9.58%	25383	97.12%	2.88%	24910	94.81%	5.19%
32	22206	81.60	18.40%	25355	96.98%	3.02%	23930	90.02%	9.98%
64	19348	67.75	32.25%	24538	92.99%	7.01%	20850	74.98%	25.02%
128	16953	55.95	44.05%	22084	81.01%	18.99%	10717	25.50%	74.50%
256	14651	44.71	55.29%	18181	61.95%	38.05%	10250	23.22%	76.78%
512	12614	34.76	65.24%	15263	47.70%	52.30%	7885	11.67%	88.33%
1024	10661	25.22	74.78%	11926	31.40%	68.60%	7353	9.07%	90.93%

Table 4.6: Result for multi2Pattern of Reference Pages

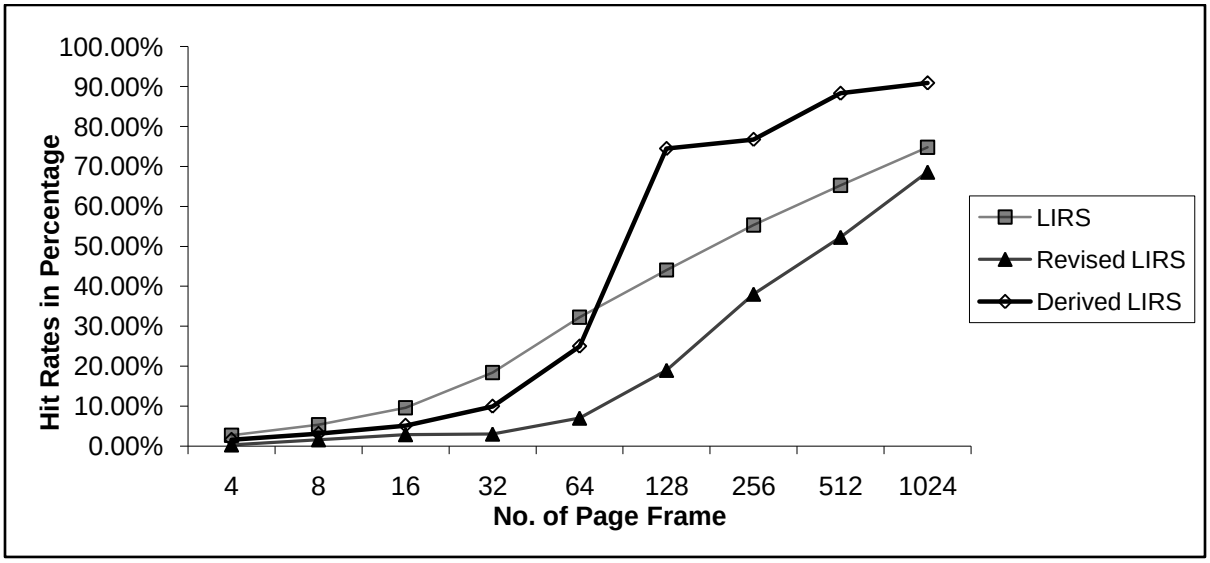

Figure 4.6: Graph Showing Result for Multi2Pattern of Reference Pages

From fig. 4.5 and 4.6, it can be noted that the hit rate of LIRS is better than those other two cases before the cache size reaches to 100 page frames. After the cache size increasing to 100, the performance is improved up to 32.22 % on trace multi1 and 30.45% on trace multi2.

CHAPTER 5

CONCLUSION AND FUTURE STUDY

4.1 Conclusion

The LIRS algorithm and its variants: Revised LIRS and Derived LIRS were implemented on C# programming language in .Net Framework. The tests were run on four types of traces pattern namely looping pattern, probabilistic pattern, temporally clustered and mixed pattern. These traces are considered typical and representation of application in that most of them is routinely used in other caching algorithm studies. As a result, we identify that the impact of pruned metadata on the performance of existed LIRS algorithm. When we use looping reference pattern, the performance of Derived LIRS and Revised LIRS is similar to that of LIRS algorithm. In the same way, on probabilistic pattern, the LIRS have better performance on small cache size. With increase in the cache size the Derived LIRS has better performance. However, Derived LIRS gives always better performance on temporally clustered reference pattern. And on the mixed type of reference pattern the Derived LIRS have performance enhancement compared to other cases with increase in cache size.

The modified LIRS algorithm named ***Derived LIRS*** employs several history information by using minimum IRR of HIR block along with retaining the advantage of LIRS algorithm. The policy decides more accurately than LIRS about page replacement. This is only due to storage of deeper history information, which is lost during stack pruning in LIRS. The modified scheme improves the performance in terms of hit ratio. In fact, the performance is improved up to (39.48%) compared to in case of LIRS for same traces.

Revised algorithm doesn't have improved performance on real traces. While avoiding pruning operation and taking minimum value HIR blocks, the dominance of unused blocks in the cache increases high. As a result, cache becomes polluted and they restrict the entry of newly accessed block in the cache. Therefore, there is no any role of pruned metadata on Revised LIRS. However, it works fine on certain cases where reference block have less variance on IRR values.

4.2 Future Work and Recommendation

As the consequences of this study, the performance of Derived LIRS algorithms seems better than other two cases of LIRS algorithm. The recommendations after this study are:

- Self-tunable size of LIR/HIR.: Size of HIR and LIR are fixed throughout this study. But making their size dynamic may fluctuate the final results and may be better than result extracted in this study.
- Computational complexity: All the results herby are based on the quantitative approach ignoring or neglecting computational aspects but its qualitative (or computational complexity) analysis is equally worthwhile.
- Mechanism to minimize data structure size: throughout this study, the metadata of all accessed block is maintained on the cache. As a result, cache overhead increased unexpectedly. So, here needs a mechanism which can minimize the cache overhead and increase the performance altogether.
- Input data used here are all secondary data which are referenced from other it would have been better if one generates its own input data pattern and analyses those data from enlisted LIRS algorithm.

References

[1] Belady, L. A., *A Study of Replacement Algorithms for a Virtual-Storage Computer*, IBM System Journal, Vol. 5, No. 2, pp. 78-101,1966.

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

[3] Bansal, S., and Modha, D. S. 2004. *CAR: Clock with Adaptive Replacement*, Conference on File and Storage Technologies (FAST 04), 2004.

[4] Choo, H., Lee, Y. J., and Yoo, S., *DIG: Degree of Inter-Reference Gap for a Dynamic Buffer Cache Management,* Information Sciences: an International Journal, Vol.176, pp. 1032-1044, 2006.

[5] Denning, P. J., *The Locality Principle,* Communication of the ACM, Vol.48, No. 7, 2005.

[6] Ding, X., Jiang, S., and Zhang, X., *BP-Wrapper: A System Framework Making Any Replacement Algorithms (Almost) Lock Contention Free*, IEEE International Conference on Data Engineering, pp. 369-380, 2009.

[7] Jiang, S. and Zhang X., *Making LRU Friendly to Weak Locality Workloads: Novel Replacement algorithm to Improve Buffer Cache Performance*, IEEE Transactions on Computers, Vol. 54, No. 8, 2005.

[8] Jiang, S., Chen, F., and Zhang, X. *CLOCK-Pro: An Effective Improvement of the CLOCK Replacement*. USENIX Annual Technical Conference, 2005.

[9] Johnson, T., and Shasha, D., *2Q: A Low Overhead High Performance Buffer management Replacement Algorithm*, proc. 20th Int'l Conf. Very Large Data Base, pp. 439-450, 1994.

[10] Lee, D., Choi, J., Kim J., Noh, S. H., Min S. L., Cho, Y., and Kim, C. S., *LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies*, IEEE Transactions on Computers, Vol. 50, No. 12, 2001.

[11] Megiddo, N. and Modha, D. S.,*ARC: A Self-Tuning, Low Overhead Replacement Cache*, Conference on File and Storage Technologies (FAST'03), 2003.

[12] Midorikawa, E. T., Piantola, R. L., and Cassettari, H. H., *On Adaptive Replacement Based on LRU with Working Area Restriction Algorithm*, SIGOPS Operating System Review, Vol. 42, No. 6, pp. 81-92, 2008.

[13] Nutt, G. J.,*Operating Systems: A Modern Perspective,*Second Edition, Addison-Wesley Longman Inc., 2000.

[14] O'Neil, E. J., O'Neil, P. E., and Weikum, G., *The LRU-K Page Replacement Algorithm for Database Disk Buffering,*Association for Computing Machinery, Special Interest Group on Management of Data (ACM SIGMOD), pp. 297-306, 1993.

[15] Paajanen, H., *Page Replacement in Operating System Memory Management,* Master's Thesis in Information Technology, University of Jyväskylä, Department of Mathematical Information Technology, 2007.

[16] Sabeghil, M. and Yaghmaee, M. H., *Using Fuzzy Logic to Improve Cache Replacement Decisions*, International Journal of Computer Science and Network Security (IJCSNS), Vol. 6, No. 3A, 2006.

[17] Silberschatz, A., Galvin, P. B., and Gagne, G., *Operating System Concepts*, Seventh Edition, John Wiley and Sons. Inc., 2004.

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

[19] Tanenbaum, A. S., *Modern operating System*, Third Edition, Prentice-Hall, 2008.

Bibliography

[1] Bhatt, P., *Operating Systems/Memory management*, Bangalore, 2004.

[2] Subedi, B., *An Evaluation of Page Replacement Algorithm Based on Low Inter Reference Recency Set Scheme on Weak Locality Workloads*, Master's Thesis, Tribhuvan University, Central Department of Computer Science and Information Technology, 2012.

[3] http://www.docstoc.com/docs/21106969/Role-of-OS-in-virtual-memory-management