SIMULATION AND ANALYSIS
OF
CACHE REPLACEMENT ALGORITHMS

By

Archana Godavarthy
Sheela Lakshminarasimhachar
Sukesh Gopinathan

COEN 283, Spring 2010
# Table of Contents

Table of Figures..................................................................................................................3

1 Introduction..........................................................................................................................4
  1.1 Objective ..........................................................................................................................4
  1.2 Problem Statement ..........................................................................................................4
  1.3 Relevance to Operating System......................................................................................4
  1.4 Drawbacks of existing approach ....................................................................................4
  1.5 Our approach ..................................................................................................................4
  1.6 Area or Scope of Investigation .......................................................................................5

2 Theoretical Bases and Literature Review..........................................................................5
  2.1 Definition of the problem ...............................................................................................5
  2.2 Theoretical background of the problem ........................................................................5
  2.3 Related research to solve the problem ..........................................................................5
  2.4 Advantages / Disadvantages of those research ..............................................................5

3 Hypothesis............................................................................................................................6

4 Methodology ..........................................................................................................................6
  4.1 Generation of input data ...............................................................................................6
  4.2 Algorithm design ...........................................................................................................6
  4.3 Language and Tools used .............................................................................................7
  4.4 Prototype .......................................................................................................................7
  4.5 Generation of output .....................................................................................................7
  4.6 Test against hypothesis ...............................................................................................8
  4.7 Proof of correctness ......................................................................................................8

5 Implementation .....................................................................................................................9
  5.1 CAR Algorithm design ...............................................................................................9
  5.2 CAR Flowchart .............................................................................................................11
  5.3 CAR Algorithm Source Code .......................................................................................13

6 Data analysis and discussion .............................................................................................16
  6.1 Output generation ..........................................................................................................16
  6.2 Output analysis ..............................................................................................................18
  6.3 Output Vs Hypothesis ...................................................................................................20
7 Conclusion and Recommendations .................................................................21
  7.1 Summary and Conclusions .........................................................................21
  7.2 Recommendations for future studies ..........................................................21

8 Bibliography ........................................................................................................22

9 Appendices ..........................................................................................................23
  9.1 Simulator flowchart ......................................................................................23
  9.2 Simulator source code ..................................................................................24
  9.3 Input listing ...................................................................................................41

Table of Figures

Figure 1: CAR Directory Structure ......................................................................9
Figure 2: CAR Algorithm Flowchart .................................................................11
Figure 3: Replace() Function Flowchart ............................................................12
Figure 4: Simulator output (default) .................................................................16
Figure 5: Simulator output with “D” option – Initial Screen .......................17
Figure 6: Simulator output with “D” option – Final Screen ........................17
Figure 7: Workload 1 Graph ..............................................................................18
Figure 8: Workload 2 Graph ..............................................................................19
Figure 9: Workload 3 Graph ..............................................................................20
Figure 10: Simulator Flowchart .........................................................................23
1 INTRODUCTION

1.1 Objective
To implement replacement algorithms that overcomes some of the disadvantages of LRU. To provide comparative analysis of these algorithms against LRU. This implementation is generic to all types of caches (virtual memory, databases, file systems, storage controllers).

1.2 Problem Statement
The optimal replacement algorithm is the benchmark for the replacement algorithms, since it achieves the best performance. But it is practically not possible to implement. LRU is an approximation to optimal replacement algorithm but it has many disadvantages. Many algorithms have tried to achieve performance close to optimal replacement algorithm by using LRU as the building block and still continue to suffer from one or more drawbacks of LRU.

1.3 Relevance to Operating System
Requirements for page replacement algorithms have changed due to differences in operating system kernel architectures. In particular, most modern OS kernels have unified virtual memory and file system caches, requiring the page replacement algorithm to select a page from among the pages of both user program virtual address spaces and cached files. Similar replacement algorithms are used for different types of caches.

1.4 Drawbacks of existing approach
The following are some of the major disadvantages of LRU and most of the other cache replacement algorithms:

1. High latency to evict an unused page/cache line.
2. It does not factor in the 'frequency' and 'spatial locality'.
3. LRU is not scan resistant.
4. In virtual memory context, overhead of moving a page to the MRU position on every page reference.
5. Single global lock contention in a multithreaded environment.

These are some of the factors that affect the performance to a great extent which need to be addressed.

1.5 Our approach
Our approach is inspired by Clock with Adaptive Replacement (CAR) and Adaptive Replacement Cache (ARC) algorithm that overcomes the above mentioned disadvantages. We are first planning to implement CAR algorithm and as a further contribution, do filtering of pages in cache based on spatial locality.
1.6 **Area or Scope of Investigation**

Implement a replacement algorithm that improves the performance of LRU and other algorithms that use LRU as the building block. Our scope is limited to simulation of the algorithms using static input data. It does not involve real-time testing.

2 **THEORETICAL BASES AND LITERATURE REVIEW**

2.1 **Definition of the problem**

LRU based replacement algorithms have lot of shortcomings which hinders the performance of cache to a great extent. There is a need for an algorithm that overcomes some of the major drawbacks as mentioned above.

2.2 **Theoretical background of the problem**

Cache replacement algorithms were a hot topic of research and debate in the 1960s and 1970s. That mostly ended with the development of sophisticated LRU approximations and working set algorithms. Since then, some basic assumptions made by the traditional cache replacement algorithms were invalidated, resulting in a revival of research.

2.3 **Related research to solve the problem**

Our solution is based on the algorithms proposed in the following research papers.

Clock with Adaptive Replacement algorithm (CAR)

Counter-Based replacement algorithm

- Penalty-based replacement algorithm
- Exploiting Spatial Locality in Data Caches

2.4 **Advantages / Disadvantages of those research**

Advantages of CAR:

- CAR removes cache hit serialization problem of LRU and ARC.
- CAR has very low overhead on cache hits and is simple to implement
- CAR is self-tuning and has high performance
- CAR is scan-resistant and has low space overhead less than 1%

Disadvantages of CAR:

- Does not take care of temporal filtering i.e. for certain workloads, we have to impose more stringent demarcation between pages of long term and short term utility.
Advantages of Penalty-based replacement algorithm

Penalty-based replacement algorithm improves the hit rate and overall cache performance in a system where load and store operations are quite predictable. This scheme can be used in conjunction with other replacement algorithms to further improving performance.

Disadvantages of Penalty-based replacement algorithm

Penalty-based replacement algorithm will give positive results only when the cost of replacement is correctly predicted. Since predictions cannot be done correctly 100% of the time, this algorithm gives negative results in some scenarios which need to be further researched.

3  HYPOTHESIS
Our approach outperforms LRU by overcoming the above mentioned disadvantages.

4  METHODOLOGY

4.1  Generation of input data
We will replicate some of the real-life scenarios of page/cache references using static files for input data. We will create a tool to generate the input files containing the cache references. Our cache simulator will be run with these input files.

4.2  Algorithm design
We have selected CAR as the basic algorithm. We will enhance this algorithm to take into account the temporal and spatial locality of references. Below is the pseudo code for CAR algorithm.

CAR Algorithm:

INITIALIZATION : Set p = 0 and set the lists T1 , B1 , T2 , and B2 to empty.
CAR( x)
 I NPUT: The requested page x.
 if (x is in T1 ∪ T2 ) then /* cache hit */
        Set the page reference bit for x to one.
 else /* cache miss */
        if (|T1 | + |T2 | = c) then /* cache full, replace a page from cache */
            replace()
            /* cache directory replacement */
        if (x is not in B1 ∪ B2 ) and ((|T1 | + |B1 | = c)) then
            Discard the LRU page in B1 .
    elseif ((|T1 | + |T2 | + |B1 | + |B2 | = 2c) and (x is not in B1 ∪ B2 )) then
            Discard the LRU page in B2 .
endif
endif
/* cache directory miss */
if (x is not in B1 ∪ B2 ) then
    Insert x at the tail of T1 . Set the page reference bit of x to 0.
/* cache directory hit */
elseif (x is in B1 ) then
    Adapt: Increase the target size for the list T1 as: p = \min \{p + \max \{1, \frac{|B2|}{|B1|}\}, c\}
    Move x at the tail of T2 . Set the page reference bit of x to 0.
/* cache directory hit */
else /* x must be in B2 */
    Adapt: Decrease the target size for the list T1 as: p = \max \{p - \max \{1, \frac{|B1|}{|B2|}\}, 0\}
    Move x at the tail of T2 . Set the page reference bit of x to 0.
endif
endif
replace()
found = 0
repeat
    if (|T1 | >= \max(1, p)) then
        if (the page reference bit of head page in T1 is 0) then
            found = 1;
            Demote the head page in T1 and make it the MRU page in B1 .
        else
            Set the page reference bit of head page in T1 to 0, and make it the tail page in T2 .
        endif
    else
        if (the page reference bit of head page in T2 is 0), then
            found = 1;
            Demote the head page in T2 and make it the MRU page in B2 .
        else
            Set the page reference bit of head page in T2 to 0, and make it the tail page in T2 .
        endif
    endif
until (found)

4.3 **Language and Tools used**
C programming language, MS Visual Studio 8

4.4 **Prototype**
Standalone Cache simulator executable supporting CAR, LRU and CLOCK replacement algorithms.

4.5 **Generation of output**
The cache simulator will be run using the statically generated input files. The same set of input files will be used to run the algorithms. On executing the simulator, it prints out the
cache performance measurements of the selected algorithm in terms of hit-miss ratio, which will be tabulated for further analysis and comparison.

4.6 Test against hypothesis

The simulator will be run using various algorithms using the statically generated input data. We will generate graphs using the output that is produced by these algorithms and a comparative analysis will be done on it.

4.7 Proof of correctness

The correctness of the algorithm is the improvement in the hit ratio compared to LRU or other LRU-based algorithms. Since the simulator will be run on a predefined set of input data, the correctness of the new algorithm will be demonstrated only for this input set.
5 IMPLEMENTATION

5.1 CAR Algorithm design

The CAR directory consists of four linked lists T1, T2, B1 and B2. The link lists T1 and T2 implements CLOCK policy and B1 and B2 implements LRU policy. The figure below shows the CAR directory structure.

![Figure 1: CAR Directory Structure](image)

The CLOCKS T1 and T2 contain those pages that are in the cache and the lists B1 and B2 contain history pages that were recently evicted from the cache. The CLOCK T1 captures “recency” while the CLOCK T2 captures “frequency”. The lists B1 and B2 are simple LRU lists. Pages evicted from T1 are placed on B1, and those evicted from T2 are placed on B2. The algorithm strives to keep B1 to roughly the same size as T2 and B2 to roughly the same size as T1. The algorithm also limits $|T1| + |B1|$ from exceeding the cache size. The sizes of the CLOCKS T1 and T2 are adapted continuously in response to a varying workload. Whenever a hit in B1 is observed, the target size of T1 is incremented; similarly, whenever a hit in B2 is observed, the target size of T1 is decremented. The new pages are inserted in either T1 or T2 immediately behind the clock hands which are shown to rotate clockwise. The page reference bit of new pages is set to 0.

Upon a cache hit to any page in $T1 \cup T2$, the page reference bit associated with the page is simply set to 1. Whenever the T1 clock hand encounters a page with a page reference bit of 1, the clock hand moves the page behind the T2 clock hand and resets the page reference bit to 0. Whenever the T1 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B1. Whenever the T2 clock hand encounters a page with a page reference bit of 1, the page reference bit is reset to
Whenever the T2 clock hand encounters a page with a page reference bit of 0, the page is evicted and is placed at the MRU position in B2.

The Data structures used for the CAR algorithm are as follows:

### CAR Directory

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1_List</td>
<td>List of the blocks contained in the T1 (these blocks are present in the cache)</td>
</tr>
<tr>
<td>T2_List</td>
<td>List of the blocks contained in the T2 (these blocks are present in the cache)</td>
</tr>
<tr>
<td>B1_List</td>
<td>List of the blocks contained in the B1</td>
</tr>
<tr>
<td>B1_List</td>
<td>List of the blocks contained in the B2</td>
</tr>
<tr>
<td>p</td>
<td>Target size for T1</td>
</tr>
</tbody>
</table>

### T1 List

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Pointer to the head node in T1</td>
</tr>
<tr>
<td>Tail</td>
<td>Pointer to the tail node in T1</td>
</tr>
<tr>
<td>Size</td>
<td>Number of nodes in T1</td>
</tr>
</tbody>
</table>

### T2 List

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Head</td>
<td>Pointer to the head node in T2</td>
</tr>
<tr>
<td>Tail</td>
<td>Pointer to the tail node in T2</td>
</tr>
<tr>
<td>Size</td>
<td>Number of nodes in T2</td>
</tr>
</tbody>
</table>

### B1 List

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>Pointer to the node at LRU position in B1</td>
</tr>
<tr>
<td>MRU</td>
<td>Pointer to the node at MRU position in B1</td>
</tr>
<tr>
<td>Size</td>
<td>Number of nodes in B1</td>
</tr>
</tbody>
</table>

### B2 List

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LRU</td>
<td>Pointer to the node at LRU position in B2</td>
</tr>
<tr>
<td>MRU</td>
<td>Pointer to the node at MRU position in B2</td>
</tr>
<tr>
<td>Size</td>
<td>Number of nodes in B2</td>
</tr>
</tbody>
</table>

### Node

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemAddress</td>
<td>Address of the block in the main memory/disk</td>
</tr>
<tr>
<td>CacheAddress</td>
<td>Address of the memory in the cache</td>
</tr>
<tr>
<td>Reference</td>
<td>Reference bit, which is set when the cache line is accessed</td>
</tr>
<tr>
<td>nextNode</td>
<td>Pointer to the next node</td>
</tr>
<tr>
<td>prevNode</td>
<td>Pointer to the previous node</td>
</tr>
</tbody>
</table>
5.2 CAR Flowchart

Figure 2: CAR Algorithm Flowchart
Figure 3: Replace() Function Flowchart
5.3 CAR Algorithm Source Code

/* CAR Data Structure */
// Node structure in T1/T2/B1/B2
struct Node
{
    long MemAddress;  // Address of block in main memory
    long CacheAddress;  // Address of block in cache memory
    bool Reference;  // Reference bit in T1/T2
    Node *nextNode;
    Node *prevNode;
};

// Linked list for T1/T2
struct List_T
{
    Node *Head;
    Node *Tail;
    long Size;  // |T1| or |T2|
};

// Linked list for B1/B2
struct List_B
{
    Node *LRU;
    Node *MRU;
    long Size;  // |B1| or |B2|
};

// Cache directory containing T1/T2/B1/B2 and p
typedef struct t_CAR_Directory
{
    List_T T1_List;
    List_T T2_List;
    List_B B1_List;
    List_B B2_List;
    long p;  // Target size of T1
} t_CAR_Directory;

/* CAR Replacement Algorithm Implementation */
bool CAR_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress)
{
    Node *pNode;
    int B_ListIndex;
    t_CAR_Directory *pDir = (t_CAR_Directory *)pDirectory;

    *CacheAddress = INVALID;

    if( (pNode = CAR_Search_T(pDir, MemAddress)) != NULL) /* Cache Hit if in T1 or T2 */
    {
        pNode->Reference = TRUE;
        return TRUE;
    }
    else /* Cache Miss not in T1 or T2*/
    {
        if( (pDir->T1_List.Size + pDir->T2_List.Size) == CACHE_SIZE) /* Cache Full */
        {
            *CacheAddress = CAR_Replace(pDir);

            // Cache Directory Replacement */
            if( CAR_Search_B(pDir, MemAddress, &B_ListIndex) == NULL)
            {
                if( (pDir->T1_List.Size + pDir->B1_List.Size) == CACHE_SIZE )
                {
                    /* Discarding a node from B1 */
                    CAR_DiscardLRU_B(&pDir->B1_List);
                }
            }
        }
    }
    return FALSE;
}
{
    /* Discarding a node from B2 */
    CAR_DiscardLRU_B(&pDir->B2_List);
}

/* Check Cache Directory */
if( (pNode = CAR_Search_B(pDir, MemAddress, &B_ListIndex)) == NULL)
{
    /* Cache Directory Miss */
    pNode = CAR_InsertToTail_T(&pDir->T1_List, MemAddress);
} else
{
    /* Cache Directory Hit */
    if( (B_ListIndex == 1) )
    {
        /* Found in B1 List */
        pNode = Min( pNode->p + Max(1, pNode->B2_List.Size/pNode->B1_List.Size), CACHE_SIZE);
        CAR_DiscardNode_B(&pDir->B1_List, MemAddress);
    } else
    {
        /* Found in B2 List */
        pNode->p = Max( pNode->p - Max(1, pNode->B1_List.Size/pNode->B2_List.Size), 0);
        CAR_DiscardNode_B(&pDir->B2_List, MemAddress);
    }
    pNode = CAR_InsertToTail_T(&pDir->T2_List, MemAddress);
}
pNode->Reference = FALSE;

/* Updating the cache */
if(*CacheAddress == INVALID)
{
    pNode->CacheAddress = CacheMemGetFreeLocation();
    *CacheAddress = pNode->CacheAddress;
} else
    pNode->CacheAddress = *CacheAddress;

return FALSE;

/* Replace Funcion */
long CAR_Replace(t_CAR_Directory *pDir)
{
    bool Found = FALSE;
    Node *pNode;
    long MemAddress;
    long CacheAddress;

    do
    {
        /* T1 size is more than max of target size and 1 */
        if(pDir->T1_List.Size >= (pDir->p > 1 ? pDir->p : 1) )
        {
            /* Looking for a node in T1 whose reference bit is 0 starting from LRU position */
            if(pDir->T1_List.Head->Reference == FALSE)
            {
                /* Found a node in T1 whose reference bit is 0 */
                Found = TRUE;
            }
        }
    while(Found == FALSE);

    return
}
MemAddress = pDir->T1_List.Head->MemAddress;
CacheAddress = pDir->T1_List.Head->CacheAddress;
CAR_DiscardHead_T(&pDir->T1_List);

/* Moving the node found from T1 to B1 */
pNext = CAR_InsertToMRU_B(&pDir->B1_List, MemAddress);

/* Removing the node from Cache */
pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;
}
else
{
  MemAddress = pDir->T1_List.Head->MemAddress;
  CacheAddress = pDir->T1_List.Head->CacheAddress;
  /* Reference bit of node in LRU position is T1 is not 0
     Move it to T2 and reset the reference bit */
  CAR_DiscardHead_T(&pDir->T1_List);
  pNode = CAR_InsertToTail_T(&pDir->T2_List, MemAddress);
  pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;
}
else
{
  /* T1 is less than target size, look for replacing nodes in T2 */
  if(pDir->T2_List.Head->Reference == FALSE)
  {
    /* Found a node in T2 whose reference bit is 0 */
    Found = TRUE;
    MemAddress = pDir->T2_List.Head->MemAddress;
    CacheAddress = pDir->T2_List.Head->CacheAddress;
    /* Moving the node from T2 to B2 and resetting the reference bit */
    CAR_DiscardHead_T(&pDir->T2_List);
    pNode = CAR_InsertToMRU_B(&pDir->B2_List, MemAddress);
    pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;
  }
  else
  {
    /* Could not find a node in T2 whose reference bit is 0 */
    MemAddress = pDir->T2_List.Head->MemAddress;
    CacheAddress = pDir->T2_List.Head->CacheAddress;
    /* Move it to the tail of T2 and reset its reference bit */
    CAR_DiscardHead_T(&pDir->T2_List);
    pNode = CAR_InsertToTail_T(&pDir->T2_List, MemAddress);
pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;
  }
}
while(Found == FALSE);
return CacheAddress;
}

Complete listing of the main simulator program, service functions used in CAR and
other replacement algorithms are shown in the Appendix section.
6 DATA ANALYSIS AND DISCUSSION

6.1 Output generation

The cache simulator is a standalone console application that reads the input memory references from an input file and prints the output to the console. The simulator supports three replacement algorithms that is user selectable. It also accepts the cache size as a parameter. It then reads the memory references from a static text file “input.txt” that contains the memory references, each separated by a single white space.

The cache simulator supports two modes of output generation based on the command line option. If no command line option is provided (default mode, without “D” option), the simulator prints the final hit ratio in percentage, for the selected algorithm and cache size. This mode is useful when running the simulator with large input file as it reduces the simulation time.

If the option “D” is provide in command line, the simulator prints the complete state of the cache directory and the cache contents for each memory reference. This mode is useful when analyzing the working of replacement algorithm.

Below are the snapshots of the simulator run with “D” option disabled and enabled.

![Figure 4: Simulator output (default)](image-url)
Figure 5: Simulator output with “D” option – Initial Screen

Figure 6: Simulator output with “D” option – Final Screen
6.2 Output analysis

The simulation was run using same input workload file for CAR, CLOCK and LRU replacement policies. Three simulation runs are shown in this report with the workload input listed in Appendix, Section 9.3.

The graphs below show the cache hit percentage for CAR, LRU and CLOCK for various cache sizes (unit size). It can be observed from the below graphs that CAR has a higher hit ratio compared to CLOCK and LRU.

Simulation 1:

![Graph showing cache hit percentage for CAR, CLOCK and LRU for various cache sizes.](image)

**Figure 7: Workload 1 Graph**

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>CAR (Hit %)</th>
<th>CLOCK (Hit %)</th>
<th>LRU (Hit %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>23.08</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>33.33</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>38.46</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>6</td>
<td>41.03</td>
<td>30</td>
<td>22.5</td>
</tr>
<tr>
<td>7</td>
<td>41.03</td>
<td>32.4</td>
<td>32.5</td>
</tr>
<tr>
<td>8</td>
<td>46.15</td>
<td>40</td>
<td>37.5</td>
</tr>
<tr>
<td>9</td>
<td>46.15</td>
<td>45</td>
<td>37.5</td>
</tr>
<tr>
<td>10</td>
<td>46.15</td>
<td>45</td>
<td>45</td>
</tr>
</tbody>
</table>

**Table 1: Data Table for Workload 1**
Simulation 2:

![Figure 8: Workload 2 Graph](image)

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>CAR (Hit %)</th>
<th>CLOCK (Hit %)</th>
<th>LRU (Hit %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>9.71</td>
<td>3.88</td>
<td>3.88</td>
</tr>
<tr>
<td>4</td>
<td>23.3</td>
<td>6.8</td>
<td>5.83</td>
</tr>
<tr>
<td>5</td>
<td>34.95</td>
<td>14.56</td>
<td>12.62</td>
</tr>
<tr>
<td>6</td>
<td>39.81</td>
<td>23.3</td>
<td>22.33</td>
</tr>
<tr>
<td>7</td>
<td>48.54</td>
<td>31.07</td>
<td>29.13</td>
</tr>
<tr>
<td>8</td>
<td>56.31</td>
<td>33.98</td>
<td>35.92</td>
</tr>
<tr>
<td>9</td>
<td>61.17</td>
<td>44.66</td>
<td>39.81</td>
</tr>
<tr>
<td>10</td>
<td>62.14</td>
<td>45.63</td>
<td>44.66</td>
</tr>
<tr>
<td>16</td>
<td>62.14</td>
<td>62.14</td>
<td>62.14</td>
</tr>
</tbody>
</table>

Table 2: Data Table for Workload 2
Simulation 3:

![Figure 9: Workload 3 Graph](image)

<table>
<thead>
<tr>
<th>Cache Size</th>
<th>CAR (Hit %)</th>
<th>CLOCK (Hit %)</th>
<th>LRU (Hit %)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>10.19</td>
<td>5.56</td>
<td>5.56</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>12.04</td>
<td>11.11</td>
</tr>
<tr>
<td>7</td>
<td>27.78</td>
<td>23.15</td>
<td>22.22</td>
</tr>
<tr>
<td>9</td>
<td>33.33</td>
<td>29.63</td>
<td>31.48</td>
</tr>
<tr>
<td>11</td>
<td>50.93</td>
<td>43.52</td>
<td>39.81</td>
</tr>
<tr>
<td>13</td>
<td>62.96</td>
<td>58.33</td>
<td>54.63</td>
</tr>
</tbody>
</table>

Table 3: Data Table for Workload 3

6.3 Output Vs Hypothesis

As explained in Section 6.2 and from the graphs, it is very clear that CAR outperforms both CLOCK and LRU replacement policies thus proving our hypothesis.
CONCLUSION AND RECOMMENDATIONS

7.1 Summary and Conclusions

In our project we have tried to demonstrate how CAR policy outperforms LRU and also how it combines best features of CLOCK and ARC by removing all the disadvantages of LRU. The following lines summarize the various advantages of CAR.

CAR has a very low overhead on cache hits.

CAR is simple to implement.

CAR is self-tuning.

The policy CAR requires no tunable, magic parameters. It has one tunable parameter \( p \) that balances between recency and frequency. The policy adaptively tunes this parameter in response to an evolving workload so as to increase the hit-ratio. A closer examination of the parameter \( p \) shows that it can fluctuate from recency (\( p = c \)) to frequency (\( p = 0 \)) and back, all within a single workload. In other words, adaptation really matters! Also, it can be shown that CAR performs as well as its offline counterpart which is allowed to select the best, offline, fixed value of \( p \) chosen specifically for a given workload and a cache size. In other words, adaptation really works! The self-tuning nature of CAR makes it very attractive for deployment in environments where no a priori knowledge of the workloads is available.

CAR is scan-resistant.

A scan is any sequence of one-time use requests. Such requests will be put on top of the list \( T_1 \) and will eventually exit from the cache without polluting the high-quality pages in \( T_2 \). Moreover, in presence of scans, there will be relatively fewer hits in \( B_1 \) as compared to \( B_2 \). Hence, our adaptation rule will tend to further increase the size of \( T_2 \) at the expense of \( T_1 \), thus further decreasing the residency time of scan in even \( T_1 \).

CAR outperforms LRU and CLOCK on a wide variety of traces and cache sizes.

7.2 Recommendations for future studies

Spatial locality is important for sequential accesses. More research can be done on extending CAR to take care of spatial locality.
8 BIBLIOGRAPHY


9 APPENDICES

9.1 Simulator flowchart

Figure 10: Simulator Flowchart
9.2 Simulator source code

/* CacheReplAlgSimulator.CPP */
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#define TRUE 1
#define FALSE 0
#define INVALID -1

// User Data Types

struct Node
{
    long MemAddress; // Address of block in main memory
    long CacheAddress; // Address of block in cache memory
    bool Reference; // Reference bit in T1/T2
    Node *nextNode;
    Node *prevNode;
};

struct List_T
{
    Node *Head;
    Node *Tail;
    long Size; // |T1| or |T2|
};

struct List_B
{
    Node *LRU;
    Node *MRU;
    long Size; // |B1| or |B2|
};

typedef directory containing T1/T2/B1/B2 and p
t_CAR_Directory
{
    List_T T1_List;
    List_T T2_List;
    List_B B1_List;
    List_B B2_List;
    long p; // Target size of T1
} t_CAR_Directory;

t_LRU_Directory
{
    Node *LRU;
    Node *MRU;
    long Size;
} t_LRU_Directory;

t_CLOCK_Directory
{
    Node *Hand;
} t_CLOCK_Directory;

typedef Cache memory structure
t_CacheMemory
{
    long MemAddress; // Address of block in main memory
Simulation and Analysis of Cache Replacement Algorithms

```c
long Valid;     // Check if data in cache is valid
long Dirty;    // Data in cache modified (or not)
)t_CacheMemory;

// Pointer to the cache replacement function
typedef bool (* t_pReplAlgFunction)(void *, long, long *);
typedef void (* t_pDisplayFunction)(void *);

// Global variables
/* CAR Directory */
t_CAR_Directory  CAR_Dir;
t_LRU_Directory     LRU_Dir;
t_CLOCK_Directory   CLOCK_Dir;

/* Cache Variables */
long CACHE_SIZE;    //Size of cache from user input
t_CacheMemory  *CacheMemory;    //Actual data representation in cache

// Function prototypes
/* General functions */
/* Cache memory Functions */
void CacheMemInit(void);      //Cache initialization function
long CacheMemGetFreeLocation(void);  //Function to find a free location in the cache
void CacheMemUpdate(long CacheAddress, long MemAddress);  //Function to update the Cache
void CacheDisplay(void);         //Function to display the cache details

/* CAR Functions */
void CAR_Init(t_CAR_Directory *Dir);      //Function to initialize the CAR_Directory structure
bool    CAR_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress);  //Main function implementing the CAR algorithm
long CAR_Replace(t_CAR_Directory *pDir);    //Replace function in the CAR implementation
Node *CAR_InsertToTail_T(List_T *pList, long MemAddress);     //Function to insert a node into T1/T2
Node *CAR_InsertToMRU_B(List_B *pList, long MemAddress);    //Function to insert a node into B1/B2
long CAR_DiscardHead_T(List_T *pList);      //Function to discard a node from T1/T2
long CAR_DiscardLRU_B(List_B *pList);        //Function to discard a node from B1/B2
void CAR_DiscardNode_B(List_B *pList, long MemAddress);    //Function to discard a node from B1/B2
Node *CAR_Search_T(t_CAR_Directory *pDir, long MemAddress);  //Function to search for a node in T1/T2
Node *CAR_Search_B(t_CAR_Directory *pDir, long MemAddress, int * Index);  //Function to search for a node in B1/B2
void CAR_Delete(t_CAR_Directory *pDir);     //Function to delete node from T1/T2/B1/B2
void CAR_Display(void *pDirectory);        //Function to display the CAR Directory structure
```
/* LRU Functions */
void LRU_Init(t_LRU_Directory *pDir);
bool LRU_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress);
Node *LRU_InsertToMRU(t_LRU_Directory *pDir, long MemAddress);
long LRU_DiscardLRU(t_LRU_Directory *pDir);
void LRU_Delete(t_LRU_Directory *pDir);
void LRU_Display(void *pDirectory);

/* CLOCK Functions */
void CLOCK_Init(t_CLOCK_Directory *pDir);
bool CLOCK_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress);
Node *CLOCK_Search(t_CLOCK_Directory *pDir);
void CLOCK_Delete(t_CLOCK_Directory *pDir);
void CLOCK_Display(void *pDirectory);

int main(int argc, char * argv[])
{
  t_pReplAlgFunction ExecReplacementAlg; //Variable for the function to implement a
  t_pDisplayFunction DisplayDirectory; //Cache content display function pointer
  void * Directory; //Pointer for directory structure
  FILE * pInputFile; //Input file
  int AlgOption; // Algorithm selection option
  char Selection[1024];

  long MemAddress, CacheAddress; //Addresses of memory locations in Main memory
  long NoOfReferences = 0; //and cache
  long HitCount = 0;

  bool EnableDirDisp = FALSE;

  printf("--------------------------------------------------------------------------\n");
  printf("SIMULATION AND ANALYSIS OF CACHE REPLACEMENT ALGORITHMS\n");
  printf("--------------------------------------------------------------------------\n");
  printf("COEN 283 Spring 2010\n");
  printf("Team: \n");
  printf(" Archana Godavarthy\n");
  printf(" Sheela Lakshminarasimhachar\n");
  printf(" Sukesh Gopinathan\n");
  printf("--------------------------------------------------------------------------\n");

  if(argc > 1)
  {
    EnableDirDisp = (strcmp(argv[1], "D") == 0) ? TRUE : FALSE;
  }
  else
  {
    printf("**Add D in command line to display directory contents\n");
  }

  while(TRUE)
  {
    NoOfReferences = 0;
    HitCount = 0;

    /* Select Algorithm */
    printf("C[AR]:1  LRU:2  CLOCK:3  Quit:0 \n");
    printf("Select Replacement Algorithm :\n");
    scanf("%s", Selection);
    if(strcmp(Selection, "0") == 0) exit(0);
    AlgOption = atoi(Selection);

    if(AlgOption == 1)
    { /* Pseudo Code for CAR */
      /* Compute
       */
    }
    else if(AlgOption == 2)
    { /* Pseudo Code for LRU */
      /* Compute
       */
    }
    else if(AlgOption == 3)
    { /* Pseudo Code for CLOCK */
      /* Compute
       */
    }
  }
}
Simulation and Analysis of Cache Replacement Algorithms

```c
switch(AlgOption)
{
    case 1: // CAR
        ExecReplacementAlg = CAR_AlgorithmExec;
        Directory = &CAR_Dir;
        DisplayDirectory = CAR_Display;
        CAR_Init(&CAR_Dir);
        break;
    case 2: // LRU
        ExecReplacementAlg = LRU_AlgorithmExec;
        Directory = &LRU_Dir;
        DisplayDirectory = LRU_Display;
        LRU_Init(&LRU_Dir);
        break;
    case 3: // CLOCK
        ExecReplacementAlg = CLOCK_AlgorithmExec;
        Directory = &CLOCK_Dir;
        DisplayDirectory = CLOCK_Display;
        CLOCK_Init(&CLOCK_Dir);
        break;
    default:
        printf("ERROR: Invalid Selection. Try again...\n\n");
        continue;
}
printf("Cache Size: ");
scanf("%d", &CACHE_SIZE);
/* Allocate cache memory array */
CacheMemory = new t_CacheMemory [CACHE_SIZE];
/* Initialize Cache memory */
CacheMemInit();
/* Error handling for the input file */
if( (pInputFile = fopen("input.txt", "r")) == NULL)
{
    printf("Unable to open input file\n");
    exit(0);}
/* Parsing through the input file */
while(!feof(pInputFile))
{
    fscanf(pInputFile, "%d\n", &MemAddress);
    if(EnableDirDisp == TRUE)
    {
        printf("Mem Ref: %4d \n\n", MemAddress);
    }
    NoOfReferences++;
    /* Calling the replacement algorithm */
    if(ExecReplacementAlg(Directory, MemAddress, &CacheAddress) == FALSE)
    {
        /* Updating the cache memory */
        CacheMemUpdate(CacheAddress, MemAddress);
    }
    else HitCount++;
    if(EnableDirDisp == TRUE)
    {
        DisplayDirectory(Directory);
        CacheDisplay();
        printf("----------------------------------------\n");
    }
}
```
/ * Printing out statistics */
printf("=====================================================\n");
printf("\n");
printf("No. of References : %d\n", NoOfReferences);
printf("Total Hit Count : %d\n", HitCount);
printf("Percentage Hit : %.2f\n", ((double)HitCount/(double)NoOfReferences)*100.0);
printf("\n");
printf("=====================================================\n");
printf("\n");
/* Delete all memory allocations */
CAR_Delete(&CAR_Dir);
LRU_Delete(&LRU_Dir);
CLOCK_Delete(&CLOCK_Dir);
delete CacheMemory;
fclose(pInputFile);
}
return 0;
}

//////////////////////////////////////////////////////////////////////////
/* Cache initialization */
void CacheMemInit(void)
{
    for(int i=0;i<CACHE_SIZE;i++)
    {
        CacheMemory[i].Valid = FALSE;
        CacheMemory[i].Dirty = FALSE;
        CacheMemory[i].MemAddress = INVALID;
    }
}

/* Finding a free location in cache */
long CacheMemGetFreeLocation(void)
{
    for(int i=0;i<CACHE_SIZE;i++)
    {
        if(CacheMemory[i].Valid == FALSE)
            return i;
    }
    return -1;
}

/* Updating the cache */
void CacheMemUpdate(long CacheAddress, long MemAddress)
{
    if(CacheAddress < 0 || CacheAddress >= CACHE_SIZE)
    {
        printf("ERROR: Address out or range in Cache Memory !!!\n");
        return;
    }
    CacheMemory[CacheAddress].MemAddress = MemAddress;
    CacheMemory[CacheAddress].Valid = TRUE;
}

/* Displaying the Cache directory */
void CacheDisplay(void)
{
    printf("Cache  : ");
    for(int i=0;i<CACHE_SIZE;i++)
    {
        if(CacheMemory[i].Valid == TRUE)
            printf("%4d ", CacheMemory[i].MemAddress);
    }
    printf("\n");
}
Simulation and Analysis of Cache Replacement Algorithms

/////////////////////////////////////////////////////////////////////////
//                                CAR
/////////////////////////////////////////////////////////////////////////

/* Initialize CAR Directory */
void CAR_Init(t_CAR_Directory *pDir)
{
    pDir->T1_List.Head = NULL;
pDir->T1_List.Tail  = NULL;
pDir->T1_List.Size = 0;
pDir->T2_List.Head = NULL;
pDir->T2_List.Tail  = NULL;
pDir->T2_List.Size = 0;
pDir->B1_List.LRU  = NULL;
pDir->B1_List.MRU  = NULL;
pDir->B1_List.Size = 0;
pDir->B2_List.LRU  = NULL;
pDir->B2_List.MRU  = NULL;
pDir->B2_List.Size = 0;
pDir->p     = 0;
}

/* CAR Replacement Algorithm Implementation */
bool CAR_AlgorithmExec(void *pDirectory, long MemAddress, long * CacheAddress)
{
    Node *pNode;
    int B_ListIndex;
t_CAR_Directory *pDir = (t_CAR_Directory *)pDirectory;

    *CacheAddress = INVALID;

    if( (pNode = CAR_Search_T(pDir, MemAddress)) != NULL) /* Cache Hit if in T1 or T2 */
    {
        pNode->Reference = TRUE;
        return TRUE;
    }
    else /* Cache Miss not in T1 or T2*/
    {
        if( (pDir->T1_List.Size + pDir->T2_List.Size) == CACHE_SIZE) /* Cache Full */
        {
            *CacheAddress = CAR_Replace(pDir);
            /* Cache Directory Replacement */
            if( CAR_Search_B(pDir, MemAddress, &B_ListIndex) == NULL)
            {
                if( (pDir->T1_List.Size + pDir->B1_List.Size) == CACHE_SIZE )
                {
                    /* Discarding a node from B1 */
                    CAR_DiscardLRU_B(&pDir->B1_List);
                }
                {
                    /* Discarding a node from B2 */
                    CAR_DiscardLRU_B(&pDir->B2_List);
                }
            }
            /* Check Cache Directory */
            if( (pNode = CAR_Search_B(pDir, MemAddress, &B_ListIndex)) == NULL)
            {
                /* Cache Directory Miss */
                pNode = CAR_InsertToTail_T(&pDir->T1_List, MemAddress);
            }
        }
    }
    else

Page 29 of 41
{ /* Cache Directory Hit */
    if (B_ListIndex == 1) {
        /* Found in B1 List */
        pDir->p = Min(pDir->p + Max(1, pDir->B2_List.Size/pDir->B1_List.Size), CACHE_SIZE);
        CAR_DiscardNode_B(&pDir->B1_List, MemAddress);
    } else {
        /* Found in B2 List */
        pDir->p = Max(pDir->p - Max(1, pDir->B1_List.Size/pDir->B2_List.Size), 0);
        CAR_DiscardNode_B(&pDir->B2_List, MemAddress);
    }
    pNode = CAR_InsertToTail_T(&pDir->T2_List, MemAddress);
}

pNode->Reference = FALSE;

/* Updating the cache */
if (*CacheAddress == INVALID) {
    pNode->CacheAddress = CacheMemGetFreeLocation();
    *CacheAddress = pNode->CacheAddress;
} else {
    pNode->CacheAddress = *CacheAddress;
}

return FALSE;
}

/* Replace node in cache - T1/T2 */
long CAR_Replace(t_CAR_Directory *pDir)
{
    bool Found = FALSE;
    Node *pNode;
    long MemAddress;
    long CacheAddress;

    do {
        /* T1 size is more than max of target size and 1 */
        if (pDir->T1_List.Size >= (pDir->p > 1 ? pDir->p : 1)) {
            /* Looking for a node in T1 whose reference bit is 0 starting from LRU position */
            if (pDir->T1_List.Head->Reference == FALSE) {
                /* Found a node in T1 whose reference bit is 0 */
                Found = TRUE;
                MemAddress = pDir->T1_List.Head->MemAddress;
                CacheAddress = pDir->T1_List.Head->CacheAddress;

                CAR_DiscardHead_T(&pDir->T1_List);

                /* Moving the node found from T1 to B1 */
                pNode = CAR_InsertToMRU_B(&pDir->B1_List, MemAddress);

                /* Removing the node from Cache */
                pNode->CacheAddress = CacheAddress;
                pNode->Reference = FALSE;
            } else {
                MemAddress = pDir->T1_List.Head->MemAddress;
                CacheAddress = pDir->T1_List.Head->CacheAddress;
            }
        } else {
            MemAddress = pDir->T1_List.Head->MemAddress;
            CacheAddress = pDir->T1_List.Head->CacheAddress;
        }
    }

    return Found;
}
/* Reference bit of node in LRU position is T1 is not 0 
Move it to T2 and reset the reference bit */

CAR_DiscardHead_T(&pDir->T1_List);

pNode = CAR_InsertToTail_T(&pDir->T2_List,MemAddress);
pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;

} else {
/* T1 is less than target size, look for replacing nodes in T2 */

if(pDir->T2_List.Head->Reference == FALSE)
{
/* Found a node in T2 whose reference bit is 0 */

Found = TRUE;

MemAddress   = pDir->T2_List.Head->MemAddress;
CacheAddress = pDir->T2_List.Head->CacheAddress;

/* Moving the node from T2 to B2 and resetting the reference bit */
CAR_DiscardHead_T(&pDir->T2_List);

pNode = CAR_InsertToMRU_B(&pDir->B2_List, MemAddress);
pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;

} else {
/* Could not find a node in T2 whose reference bit is 0 */

MemAddress   = pDir->T2_List.Head->MemAddress;
CacheAddress = pDir->T2_List.Head->CacheAddress;

/* Move it to the tail of T2 and reset its reference bit */
CAR_DiscardHead_T(&pDir->T2_List);

pNode = CAR_InsertToTail_T(&pDir->T2_List, MemAddress);
pNode->CacheAddress = CacheAddress;
pNode->Reference = FALSE;

}

}while(Found == FALSE);

return CacheAddress;

} /* Discarding the node from T1/T2 */

long CAR_DiscardHead_T(List_T *pList)
{
long MemAddress;
Node *pNode;

if(pList->Head != NULL)
{
/* Discarding the node at the head position in T1/T2 */

MemAddress = pList->Head->MemAddress;
pNode      = pList->Head->nextNode;
delete pList->Head;

/* If there are more nodes in T1/T2, change the Head position */
if(pNode != NULL)
{
pList->Head = pNode;
pList->Head->prevNode = NULL;
}
else /* No more elements are present in T1/T2 */
{
pList->Head = NULL;
}
pList->Tail = NULL;
}

pList->Size--;
return MemAddress;
}

return INVALID;
}

/* Discard node in LRU position of B1/B2 */
long CAR_DiscardLRU_B(List_B *pList)
{
  long MemAddress;
  Node *pNode;

  if(pList->LRU != NULL)
  {
    /* Discarding the node at the LRU position in B1/B2 */
    MemAddress = pList->LRU->MemAddress;
    pNode = pList->LRU->nextNode;
    delete pList->LRU;

    /* If there are more nodes in B1/B2, change the LRU position */
    if(pNode != NULL)
    {
      pList->LRU = pNode;
      pList->LRU->prevNode = NULL;
    }
    else /* No more nodes are present in B1/B2 */
    {
      pList->LRU = NULL;
      pList->MRU = NULL;
    }

    pList->Size--;
    return MemAddress;
  }

  return INVALID;
}

/* Function to discard a particular node in B1/B2 */
void CAR_DiscardNode_B(List_B *pList, long MemAddress)
{
  Node *pNode = pList->LRU;

  /* The node to be deleted is the only node in the B list */
  if( (pList->LRU == pList->MRU) && (pNode->MemAddress == MemAddress) )
  {
    delete pList->LRU;
    pList->LRU = NULL;
    pList->MRU = NULL;
    pList->Size = 0;
    return;
  }

  while(pNode != NULL)
  {
    Node *tNode;

    /* Found the node to be deleted */
    if(pNode->MemAddress == MemAddress)
    {
      tNode = pNode->prevNode;
      /* Node to be deleted is in LRU position */
      if(tNode == NULL)
      {

Page 32 of 41
pList->LRU = pNode->nextNode;
pList->LRU->prevNode = NULL;
} else /* Node to be deleted is NOT in LRU position */
{
    if (pNode->nextNode == NULL) /* Node to be deleted is in MRU position */
    {
        pList->MRU = tNode;
pList->MRU->nextNode = NULL;
    } else /* Node to be deleted is NOT in MRU position */
    {
        (pNode->nextNode)->prevNode = tNode;
tNode->nextNode = pNode->nextNode;
    }
}
delete pNode;
pList->Size--;
break;
} pNode = pNode->nextNode;
} /* Insert a node to tail of T1/T2 */
Node *CAR_InsertToTail_T(List_T *pList, long MemAddress)
{
    Node *newNode = new Node;
    newNode->MemAddress = MemAddress;
    newNode->nextNode = NULL;

    if (pList->Head == NULL)
    {
        /* Inserting into an empty list */
pList->Head = newNode;
pList->Tail = newNode;
    } else
    {
        /* Inserting into a list containing only one element */
        if (pList->Head == pList->Tail)
        {
            pList->Head->nextNode = newNode;
        }
pList->Tail->nextNode = newNode;
    newNode->prevNode = pList->Tail;
pList->Tail = newNode;
}
pList->Size++;

    return newNode;
}

/* Insert a node to MRU position of B1/B2 */
Node *CAR_InsertToMRU_B(List_B *pList, long MemAddress)
{
    Node *newNode = new Node;
    newNode->MemAddress = MemAddress;
    newNode->nextNode = NULL;

    if (pList->LRU == NULL)
    {
        /* Inserting into a list containing only one element */
pList->LRU = newNode;
pList->MRU = newNode;
    }
}  
else  
{  
/* Inserting at the MRU position */  
if(pList->LRU == pList->MRU)  
{  
pList->LRU->nextNode = newNode;  
}  
pList->MRU->nextNode = newNode;  
newNode->prevNode = pList->MRU;  
pList->MRU = newNode;  
}  
pList->Size++;  
return newNode;  
}  
/* Search for memory reference in CAR directory (T1/T2) */  
Node *CAR_Search_T(t_CAR_Directory *pDir, long MemAddress)  
{  
/* Searching for the node in T1 */  
Node *pNode = pDir->T1_List.Head;  
while(pNode != NULL)  
{  
/* Found the node */  
if(pNode->MemAddress == MemAddress)  
return pNode;  
pNode = pNode->nextNode;  
}  
/* Searching for the node in T2 */  
pNode = pDir->T2_List.Head;  
while(pNode != NULL)  
{  
/* Found the node */  
if(pNode->MemAddress == MemAddress)  
return pNode;  
pNode = pNode->nextNode;  
}  
return NULL;  
}  
/* Search for memory reference in B1/B2 */  
Node *CAR_Search_B(t_CAR_Directory *pDir, long MemAddress, int * Index)  
{  
/* Searching for node in B1 */  
Node *pNode = pDir->B1_List.LRU;  
*Index = 1;  
while(pNode != NULL)  
{  
if(pNode->MemAddress == MemAddress)  
return pNode;  
pNode = pNode->nextNode;  
}  
/* Searching for node in B2 */  
pNode = pDir->B2_List.LRU;  
*Index = 2;  
while(pNode != NULL)  
{  
if(pNode->MemAddress == MemAddress)  
return pNode;  
pNode = pNode->nextNode;  
}
/ * Node not found in B1 or B2 */
*Index = 0;
return NULL;
}

/* Remove all memory allocation for LRU directory */
void CAR_Delete(t_CAR_Directory *pDir)
{
    Node *pNode;
    Node *pDelNode;

    pNode = pDir->T1_List.Head;
    while(pNode != NULL)
    {
        pDelNode = pNode;
        pNode = pNode->nextNode;
        delete pDelNode;
    }

    pNode = pDir->T2_List.Head;
    while(pNode != NULL)
    {
        pDelNode = pNode;
        pNode = pNode->nextNode;
        delete pDelNode;
    }

    pNode = pDir->B1_List.LRU;
    while(pNode != NULL)
    {
        pDelNode = pNode;
        pNode = pNode->nextNode;
        delete pDelNode;
    }

    pNode = pDir->B2_List.LRU;
    while(pNode != NULL)
    {
        pDelNode = pNode;
        pNode = pNode->nextNode;
        delete pDelNode;
    }

    CAR_Init(pDir);
}

/* Display CAR directory content */
void CAR_Display(void *pDirectory)
{
    t_CAR_Directory *pDir = (t_CAR_Directory *)pDirectory;

    /* Displaying the T1 list */
    Node *pNode = pDir->T1_List.Head;
    printf("    T1 : ");
    while(pNode != NULL)
    {
        printf("%4d|%d ", pNode->MemAddress, pNode->Reference);
        pNode = pNode->nextNode;
    }
    printf("\n");

    /* Displaying the B1 list */
    pNode = pDir->B1_List.LRU;
    printf("    B1 : ");
    while(pNode != NULL)
    {
        printf("%4d ", pNode->MemAddress);
        pNode = pNode->nextNode;
    
}
printf("\n\n");
/* Displaying the T2 list */
pNode = pDir->T2_List.Head;
printf("    T2 :");
while(pNode != NULL)
{
    printf("%4d|%d ", pNode->MemAddress, pNode->Reference);
pNode = pNode->nextNode;
}
printf("\n");
/* Displaying the B2 list */
pNode = pDir->B2_List.LRU;
printf("    B2 :");
while(pNode != NULL)
{
    printf("%4d ", pNode->MemAddress);
pNode = pNode->nextNode;
}
printf("\n\n");
/* Find minimum of two integers */
long Min(long a, long b)
{
    if(a < b) return a;
    return b;
}
/* Find maximum of two integers */
long Max(long a, long b)
{
    if(a > b) return a;
    return b;
}

/////////////////////////////////////////////////////////////////////////
//                                LRU
/////////////////////////////////////////////////////////////////////////
/* Initialize LRU directory */
void LRU_Init(t_LRU_Directory *pDir)
{
pDir->LRU = NULL;
pDir->MRU = NULL;
pDir->size = 0;
}
/* LRU Replacement Algorithm Implementation */
bool LRU_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress)
{
    Node *pNode;
t_LRU_Directory *pDir = (t_LRU_Directory *)pDirectory;
    if( (pNode = LRU_Search(pDir, MemAddress)) != NULL)
    {
        /* HIT */
        LRU_MoveToMRU(pDir, pNode);
        return TRUE;
    }
    else
    {
        /* MISS */
        if(pDir->Size == CACHE_SIZE)
        {
            /* Cache full */
            *CacheAddress = LRU_DiscardLRU(pDir);
            pNode = LRU_InsertToMRU(pDir,MemAddress);
            pNode->CacheAddress = *CacheAddress;
if(cache not yet full)
    pNode = LRU_InsertToMRU(pDir, MemAddress);
    pNode->CacheAddress = CacheMemGetFreeLocation();
    *CacheAddress = pNode->CacheAddress;
else

return FALSE;

Node *LRU_Search(t_LRU_Directory *pDir, long MemAddress)
{
    Node *pNode = pDir->LRU;
    while(pNode != NULL)
    {
        if(pNode->MemAddress == MemAddress)
            return pNode;
        pNode = pNode->nextNode;
    }
    return NULL;

void LRU_MoveToMRU(t_LRU_Directory *pDir, Node *pNode)
{
    Node *pTempNode;
    if(pNode == NULL) return;
    if(pNode == pDir->MRU)
    {
        /* pNode already in MRU position. */
        return;
    }
    /* Remove pNode from current position */
    pTempNode = pNode->prevNode;
    if(pTempNode == NULL)
    {
        pDir->LRU = pNode->nextNode;
        pDir->LRU->prevNode = NULL;
    }
    else
    {
        (pNode->nextNode)->prevNode = pTempNode;
        pTempNode->nextNode = pNode->nextNode;
    }
    /* Add pNode to MRU position */
    pDir->MRU->nextNode = pNode;
    pNode->prevNode = pDir->MRU;
    pDir->MRU = pNode;
    pDir->MRU->nextNode = NULL;
}

Node *LRU_InsertToMRU(t_LRU_Directory *pDir, long MemAddress)
{
    Node *newNode = new Node;
    newNode->MemAddress = MemAddress;
    newNode->CacheAddress = INVALID;
newNode->nextNode = NULL;

if(pDir->LRU == NULL)
{
    /* List empty */
    pDir->LRU = newNode;
    pDir->MRU = newNode;
    newNode->prevNode = NULL;
}
else
{
    if(pDir->LRU == pDir->MRU)
    {
        pDir->LRU->nextNode = newNode;
    }
    pDir->MRU->nextNode = newNode;
    newNode->prevNode = pDir->MRU;
    pDir->MRU = newNode;
}

pDir->Size++;
return newNode;

/* Discard node at LRU position */
long LRU_DiscardLRU(t_LRU_Directory *pDir)
{
    long CacheAddress;
    Node *pNode;
    if(pDir->LRU != NULL)
    {
        CacheAddress = pDir->LRU->CacheAddress;
        pNode = pDir->LRU->nextNode;
        delete pDir->LRU;
        if(pNode != NULL)
        {
            pDir->LRU = pNode;
            pDir->LRU->prevNode = NULL;
        }
        else
        {
            pDir->LRU = NULL;
            pDir->MRU = NULL;
        }
        pDir->Size--;
        return CacheAddress;
    }

    return INVALID;
}

/* Display LRU directory content */
void LRU_Display(void *pDirectory)
{
    t_LRU_Directory *pDir = (t_LRU_Directory *)pDirectory;
    Node *pNode = pDir->LRU;

    printf("LRU-MRU: ");
    while(pNode != NULL)
    {
        printf("%d ", pNode->MemAddress);
        pNode = pNode->nextNode;
    }
    printf("\n");
}
/ Remove all memory allocation for LRU directory */
void LRU_Delete(t_LRU_Directory *pDir)
{
    Node *pNode = pDir->LRU;
    Node *pDelNode;

    while(pNode != NULL)
    {
        pDelNode = pNode;
        pNode = pNode->nextNode;
        delete pDelNode;
    }

    pDir->LRU = NULL;
    pDir->MRU = NULL;
    pDir->Size = 0;
}

/////////////////////////////////////////////////////////////////////////
//                            CLOCK
/////////////////////////////////////////////////////////////////////////

/* Initialize CLOCK directory */
void CLOCK_Init(t_CLOCK_Directory *pDir)
{
    Node *prevNode;
    Node *newNode = new Node;
    newNode->MemAddress   = INVALID;
    newNode->CacheAddress = INVALID;
    newNode->Reference    = FALSE;
    pDir->Hand = newNode;
    prevNode   = pDir->Hand;

    for(long i=0;i<CACHE_SIZE-1;i++)
    {
        newNode = new Node;
        newNode->MemAddress   = INVALID;
        newNode->CacheAddress = INVALID;
        newNode->Reference    = FALSE;
        newNode->prevNode     = prevNode;
        prevNode->nextNode    = newNode;
        prevNode = newNode;
    }

    /* Make it a circular list */
    prevNode->nextNode  = pDir->Hand;
    pDir->Hand->prevNode = prevNode;
}

/* Clock Replacement Algorithm Implementation */
bool CLOCK_AlgorithmExec(void *pDirectory, long MemAddress, long *CacheAddress)
{
    Node *pNext;
    t_CLOCK_Directory *pDir = (t_CLOCK_Directory *)pDirectory;

    if( (pNext = CLOCK_Search(pDir, MemAddress)) != NULL)
    {
        /* HIT */
        pNext->Reference = TRUE;
        return TRUE;
    } else
    {
        /* MISS */
        if( (*CacheAddress = CacheMemGetFreeLocation()) == INVALID )

        return FALSE;
    }
{ /* Cache full */
    while(pDir->Hand->Reference == TRUE)
    {
        pDir->Hand->Reference = FALSE;
        pDir->Hand = pDir->Hand->nextNode;
    }
    *CacheAddress = pDir->Hand->CacheAddress;
} else
{ pDir->Hand->CacheAddress = *CacheAddress;
} pDir->Hand->MemAddress = MemAddress;
    pDir->Hand = pDir->Hand->nextNode;
    return FALSE;
}

/* Search for reference in CLOCK Directory */
Node *CLOCK_Search(t_CLOCK_Directory *pDir, long MemAddress)
{ Node *pNode = pDir->Hand;
    do
    { if(pNode == NULL) break;
       if(pNode->MemAddress == MemAddress)
       return pNode;
       pNode = pNode->nextNode;
    } while(pNode != pDir->Hand);
    return NULL;
}

/* Remove all memory allocation for CLOCK directory */
void CLOCK_Delete(t_CLOCK_Directory *pDir)
{ Node *pNode = pD
    Node *pHand = pDir->Hand;
    Node *pDelNode;
    while(pNode != NULL)
    { pDelNode = pNode;
       pNode = pNode->nextNode;
       delete pDelNode;
       if(pNode == pHand) break;
    }
    pDir->Hand = NULL;
}

/* Display CLOCK directory contents */
void CLOCK_Display(void *pDirectory)
{ t_CLOCK_Directory *pDir = (t_CLOCK_Directory *)pDirectory;
    Node *pNode = pDir->Hand;
    printf("HAND...: ");
    do
    { if(pNode == NULL) break;
       printf("%d|%d ", pNode->MemAddress,pNode->Reference);
       pNode = pNode->nextNode;
    } while(pNode != pDir->Hand);
printf("\n");

9.3 Input listing

Workload 1 Input:
20 30 40 50 11 12 13 14 12 14 12 13 90 100 110 120 11 12 13 11 14 15 14
11 60 70 80 15 16 17 11 12 13 14 15 13 130 140 150

Workload 2 Input:
100 110 120 130 1 2 3 4 1 2 4 140 150 3 2 1 4 6 5 170 180 190 200 5 6 1 7
3 4 2 5 6 7 4 210 220 1 3 5 7 2 4 6 1 8 6 7 2 3 4 2 3 6 7 8 230 240 250
260 270 280 290 300 7 8 1 3 5 6 2 4 1 8 6 310 320 330 340 350 1 8 7 2 4 3
8 6 5 7 1 360 370 380 390 400 410 6 5 4 3 2 1 6

Workload 3 Input:
1 15 2 3 15 17 16 4 17 16 7 6 5 15 17 8 5 21 16 15 17 2 3 16 21 17 15 16
4 7 6 7 3 2 17 21 23 24 8 15 12 17 19 21 16 4 1 2 3 8 19 8 7 6 1 2 3 4 5
6 7 19 18 17 16 2 13 1 2 3 4 5 21 22 18 16 14 12 2 4 6 1 2 3 4 5 18 19 20
21 22 23 24 10 11 12 13 14 15 9 6 7 6 5 4 3 2 1