CEN 502 PROJECT 3

INTRODUCTION

Caching is an area of a computer’s memory devoted for temporarily storing recently used information. Data is stored on the local hard drive in order to make it faster for the user to access it, which helps improve the efficiency of the computer and its overall performance.
Most caching occurs without the user knowing about it. For example, when a user returns to a Web page they have recently accessed, the browser can pull those files from the cache instead of the original server because it has stored the user’s activity. The storing of that information saves the user time by getting to it faster, and lessens the traffic on the network. [1]

Caching is one of the oldest and the most fundamental concepts in modern computing. The cache is much faster than the auxiliary memory, but it is also much more expensive. [2] The project involves implementation of two replacement policies: LRU (Least Recently Used) replacement algorithm and the ARC (Adaptive Replacement Cache) replacement algorithm on real-life workloads. We compute and compare the hit ratios of each of the two algorithms for a given cache size. The language used for the implementation of this project is Java, and the data structure used are Linked List.

LEAST RECENTLY USED (LRU)

Accessing large amounts of data, is deemed too slow, a common speed up technique is to keep a small amount of the data in a more accessible location known as a *cache*. The first time a particular piece of data is accessed, the slow method must be used. However, the data is then stored in the cache so that the next time you need it you can access it much more quickly. For example, a database system may keep data cached in memory so that it doesn't have to read the hard drive. Or a web browser might keep a cache of web pages on the local machine so that it doesn't have to download them over the network.
In general, a cache is much too small to hold all the data you might possibly need, so at some point you are going to have to remove something from the cache in order to make room for new data. The goal is to retain those items that are more likely to be retrieved again soon. This requires a sensible algorithm for selecting what to remove from the cache. One simple but effective algorithm is the Least Recently Used, or LRU, algorithm. When performing LRU caching, you always throw out the data that was least recently used.
As an example, let's imagine a cache that can hold up to five pieces of data. Suppose we access three pieces of data—A, B, and C. As we access each one, we store it in our cache, so at this point we have three pieces of data in our cache and two empty spots. Now suppose we access D and E. They are added to the cache as well, filling it up. Next suppose we access A again. A is already in the cache, so the cache does not change; however, this access counts as a use, making A the most recently used. Now if we were to access F, we would have to throw something out to make room for F. At this point, B has been used least recently, so we throw it out and replace it with F. If we were now to access B again, it would be exactly as the first time we accessed it: we would retrieve it and store it in the cache, throwing out the least recently used data—this time C—to make room for it. [3]

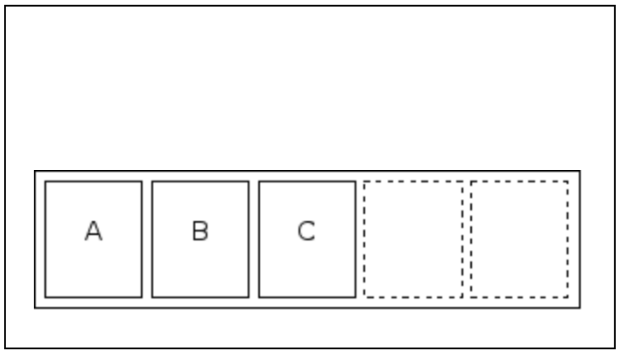


Figure 1: Cache after A, B, C

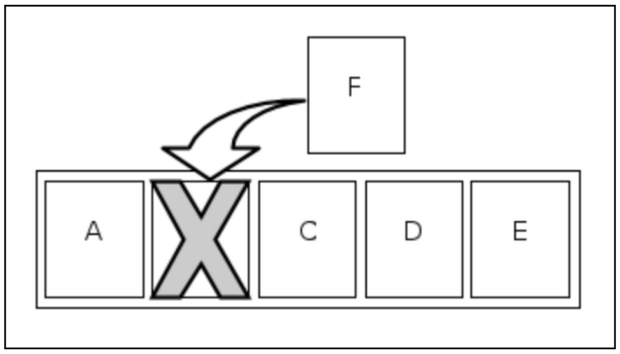


Figure 2: Cache after A, B, C, D, E, A, F

IMPLEMENTATION OF LRU

For the project first we implement the LRU replacement policy. In this policy, the real-life workload files are read and stored in the computer cache. The size of the cache as well as the file to be read are taken from the user as a command line argument. The cache is characterized by two important terms – the LRU and the MRU (Most Recently Used). The file to be read consists of four fields out of which only the first two fields are taken into consideration. A data structure (Linked List) is used to store the data that is read from the file, that is, the linked list acts as the cache for the policy. The size of this cache is the value that is entered by the user as a command line argument. While reading every page of the file, a search in the cache is initiated which checks whether the current page already exists in the cache or not. If it already exists, then that page is shifted from its position to the MRU position of the cache. But if the page does not exist in the cache, then the page from the LRU position of the cache is removed and the non-existing page is stored at the MRU position of the cache. The policy also checks the length of the cache (linked list). If the link list is full and there is no more space in the cache to include any more pages, then the LRU page is deleted/removed from the cache and then the new page is loaded in the cache at the MRU (Most Recently Used) position. The program executes till the time the file is not empty. After the entire file is read, the hit ratio is calculated. A hit counter is incremented every time the current page read, already exists in the cache.

ALGORITHM:

```
INPUT: file to be read
INPUT: cache size
Create the Linked List
Open and Read the trace file line by line
While(the file contains pages){
        String[]  line = Currentline.split(“ “);
        int sb1 = Convert to Integer and store the first field
        int sb2 = Convert to Integer and store the second field
        for(int i = sb1; i<(sb1+sb2);i++){
                If(list contains the current page){
                        Cache_hit++
                        Remove the current page
                        And it to MRU
                Else(if list.size()>=cache_size){
                        remove the last page
                        add the current page to MRU
                }
                else(){
                        add page to MRU
                }
                cache_miss++
        }
close file
memory access = cache_hit + cache_miss;
hit ratio = hit / mem_acc
Print the results
}
}
```

ADAPTIVE REPLACEMENT CACHE (ARC)

Adaptive Replacement Cache (ARC) is a page replacement algorithm with better performance than LRU (least recently used) This is accomplished by keeping track of both frequently used and recently used pages plus a recent eviction history for both. Basic LRU maintains an ordered list (the cache directory) of resource entries in the cache, with the sort order based on the time of most recent access. New entries are added at the top of the list, after the bottom entry has been evicted. Cache hits move to the top, pushing all other entries down.

ARC improves the basic LRU strategy by splitting the cache directory into two lists, T1 and T2, for recently and frequently referenced entries. In turn, each of these is extended with a *ghost* list (B1 or B2), which is attached to the bottom of the two lists. These *ghost* lists act as scorecards by keeping track of the history of recently evicted cache entries, and the algorithm uses *ghost* hits to adapt to recent change in resource usage. As an entry is evicted into a *ghost* list its data is discarded. The combined cache directory is organized in four LRU lists:

1 T1, for recent cache entries.
2 T2, for frequent entries, referenced at least twice.
3 B1, *ghost* entries recently evicted from the T1 cache, but are still tracked.
4 B2, similar *ghost* entries, but evicted from T2.

T1 and B1 together are referred to as L1, a combined history of recent single references. Similarly, L2 is the combination of T2 and B2. [4]

IMPLEMENTATION OF ARC

Next we implement the ARC replacement policy. In this policy, the real-life workload files are read and stored in the computer cache. The size of the cache as well as the file to be read are taken from the user as a command line argument. Here, four data structures (Linked List) are created to store the pages read from the file. The four link lists are T1, T2, B1, B2 which stand for 'Top 1', 'Top 2' and 'Bottom 2', 'Bottom 2'. The four data structures actually combine to form lists L1 and L2. L1 is thus, a sum of B1 and T1 while L2 is the sum of B2 and T2. While reading the current page, the program checks for the four cases – whether the current page exists in T1 or T2, whether the page exists in B1, whether the current page exists in B2 or whether the current page does not exist in any of the data structures. Each case has its own set of instructions to be executed. After the entire file is read, the hit ratio is calculated. A hit is considered if and only if the current page that is being read already exists in T1 or T2. If it does, then it is counted as a hit. If not, it is counted as a miss.

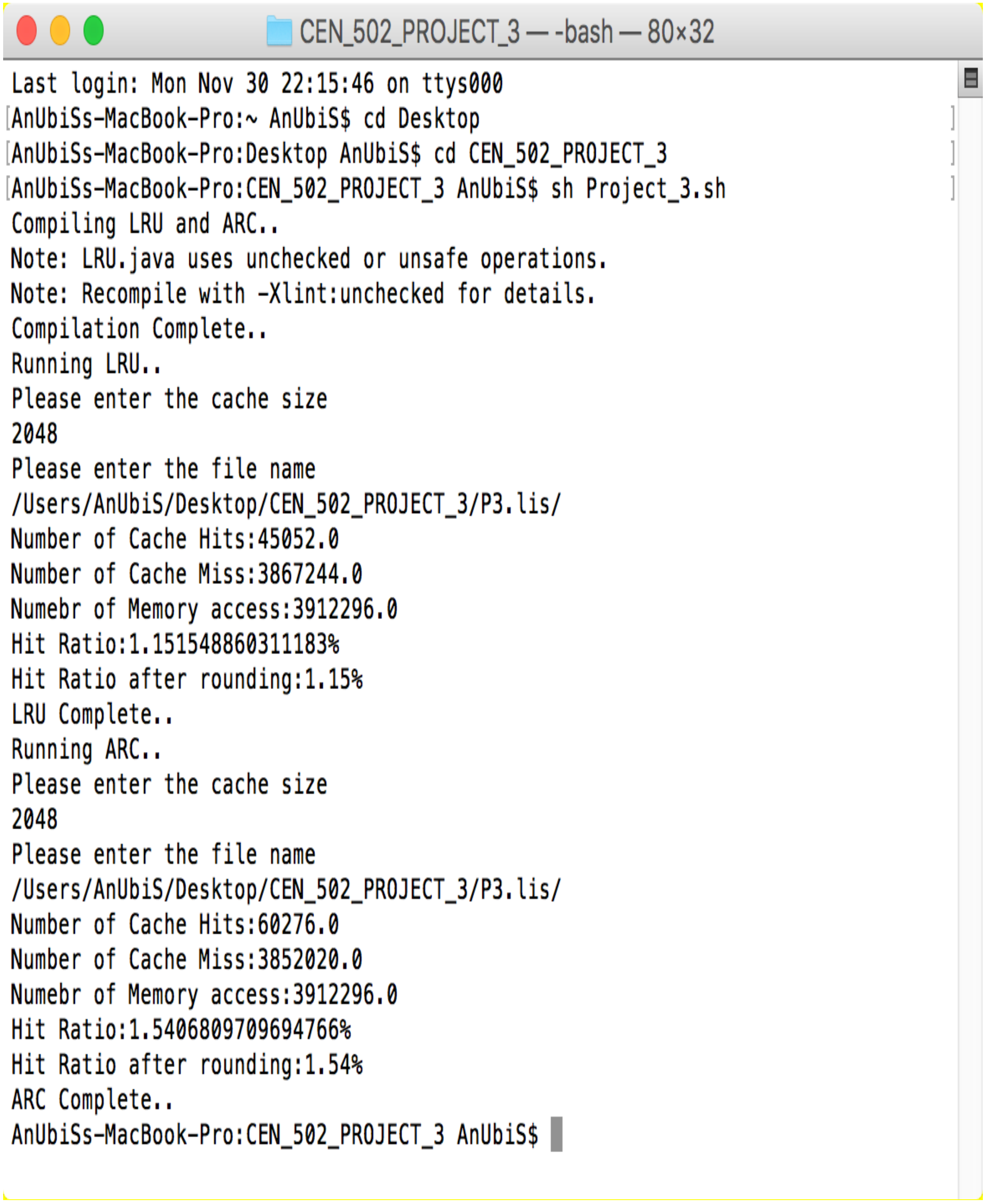
ALGORITHM:

```
INPUT: File to be read
INPUT: cache size
Create 4 linked lists and set them to empty
While(the file contains pages){
        String[]  line = CurrentLine.split(" ");
        int sb1 = Convert to Integer and store the first field
        int sb2 = Convert to Integer and store the second field
        for(int i = sb1; i<(sb1+sb2);i++){

        //CASE 1
        if (T1 or T2 conatins current page) {
                cache_hit++
                move current page to MRU of T2
        }

        //CASE 2
        if (B1 contains current page) {
                cache_miss++
                delta1 = |B2|/|B1|
                p = min{p+delta1, size}
                Replace(current page,p)
                Move current page from B1 to MRU of T2
        }

        //CASE 3
        if (B2 contains current page) {
                cache_miss++
                delta2 = |B1|/|B2|
                p = min{p-delta1, 0}
                Replace(current page,p)
                Move current page from B2 to MRU of T2
        }

        //CASE 4
        if (T1 or T2 or B1 or B2 does not contain current page) {
                if(T1+B1=size){
                        if(|T1|<size){
                                Delete LRU in B1
                                Replace (current page, p)
                        }
                                Delete LRU in T1
                }
        }
        else{
                if(|T1|+|T2|+|B1|+|B2|>=c){
                        if(|T1|+|T2|+|B1|+|B2|=2*size){
                                Delete LRU in B2
                        }
                        Replace(current page,p)
                }
        Move current page to MRU of T1
        }
close file
memory access = cache_hit + cache_miss;
hit ratio = hit / mem_acc
Print the results
}
}
```

DESIGN AND ANALYSIS

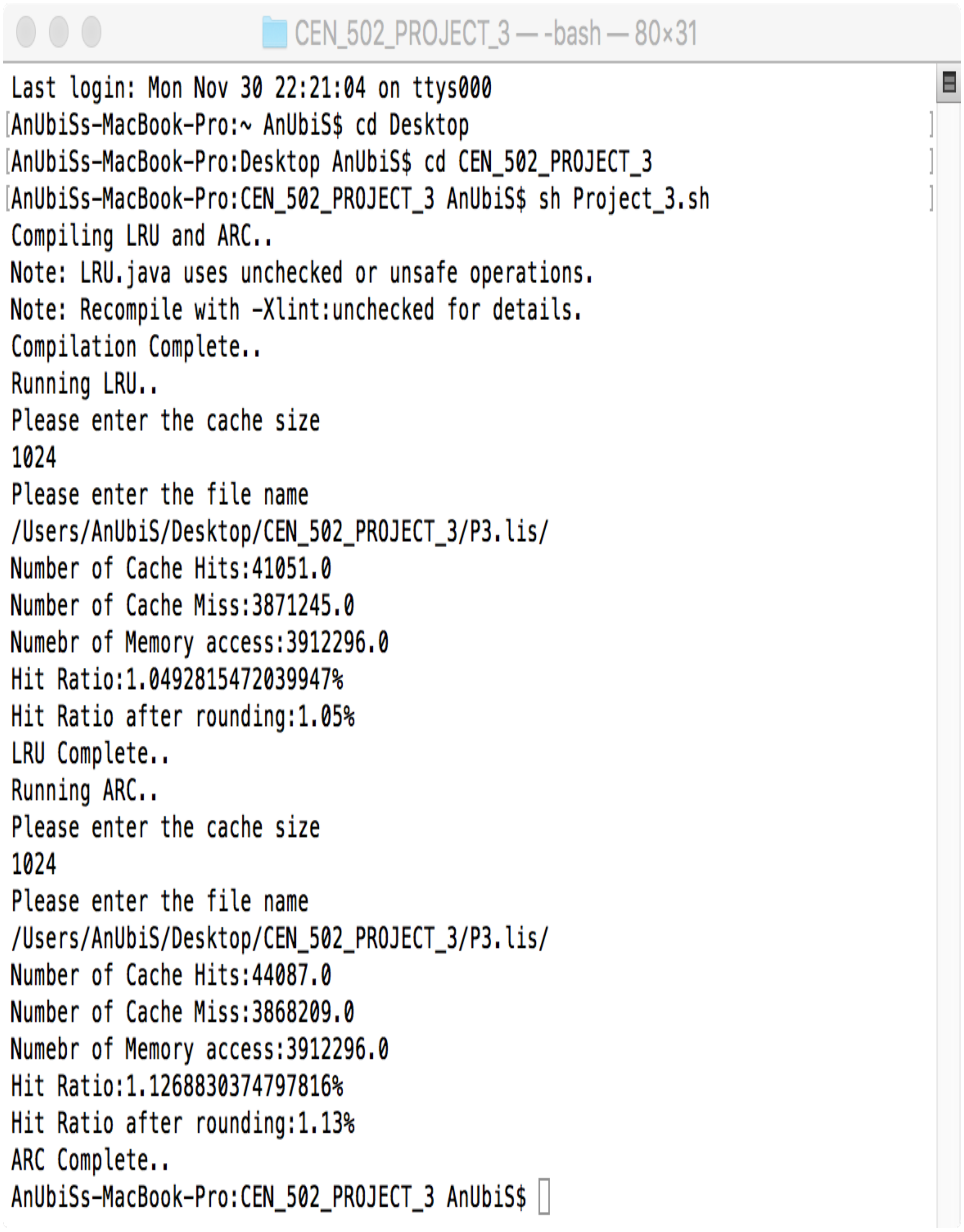
1. Cache size and the file to be read is taken from the user.
2. For both the policies Linked Lists are used as the data structures to store the pages.
3. The number of hits, number of misses, number of memory access, hit ratio are calculated using the data type double.
4. I have used the predefined java class text.DecimalFormat to round of the final output to two decimal place value.
5. The calculations done are:
   - Memory Access = Hits + Misses
   - Hit Ratio = (Hits/Memory Access) * 100

RESULTS OBTAINED



```
Last login: Mon Nov 30 22:15:46 on ttys000
AnUbiSs-MacBook-Pro:~ AnUbiS$ cd Desktop
AnUbiSs-MacBook-Pro:Desktop AnUbiS$ cd CEN_502_PROJECT_3
AnUbiSs-MacBook-Pro:CEN_502_PROJECT_3 AnUbiS$ sh Project_3.sh
Compiling LRU and ARC..
Note: LRU.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Compilation Complete..
Running LRU..
Please enter the cache size
2048
Please enter the file name
/Users/AnUbiS/Desktop/CEN_502_PROJECT_3/P3.lis/
Number of Cache Hits:45052.0
Number of Cache Miss:3867244.0
Numebr of Memory access:3912296.0
Hit Ratio:1.151548860311183%
Hit Ratio after rounding:1.15%
LRU Complete..
Running ARC..
Please enter the cache size
2048
Please enter the file name
/Users/AnUbiS/Desktop/CEN_502_PROJECT_3/P3.lis/
Number of Cache Hits:60276.0
Number of Cache Miss:3852020.0
Numebr of Memory access:3912296.0
Hit Ratio:1.5406809709694766%
Hit Ratio after rounding:1.54%
ARC Complete..
AnUbiSs-MacBook-Pro:CEN_502_PROJECT_3 AnUbiS$ 
```

CACHE_SIZE = 2048



```
Last login: Mon Nov 30 22:21:04 on ttys000
AnUbiSs-MacBook-Pro:~ AnUbiS$ cd Desktop
AnUbiSs-MacBook-Pro:Desktop AnUbiS$ cd CEN_502_PROJECT_3
AnUbiSs-MacBook-Pro:CEN_502_PROJECT_3 AnUbiS$ sh Project_3.sh
Compiling LRU and ARC..
Note: LRU.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
Compilation Complete..
Running LRU..
Please enter the cache size
1024
Please enter the file name
/Users/AnUbiS/Desktop/CEN_502_PROJECT_3/P3.lis/
Number of Cache Hits:41051.0
Number of Cache Miss:3871245.0
Numebr of Memory access:3912296.0
Hit Ratio:1.0492815472039947%
Hit Ratio after rounding:1.05%
LRU Complete..
Running ARC..
Please enter the cache size
1024
Please enter the file name
/Users/AnUbiS/Desktop/CEN_502_PROJECT_3/P3.lis/
Number of Cache Hits:44087.0
Number of Cache Miss:3868209.0
Numebr of Memory access:3912296.0
Hit Ratio:1.1268830374797816%
Hit Ratio after rounding:1.13%
ARC Complete..
AnUbiSs-MacBook-Pro:CEN_502_PROJECT_3 AnUbiS$
```

CACHE_SIZE = 1024

CONCLUSION

LRU and ARC caching policies were successfully tested on real world work loads and implemented as desired. From the results it is very clear that the number of hits in ARC policy is more than that obtained in LRU policy for the same cache size and the same file. Thus the hit ratio also is more for ARC as compared to LRU.

REFERENCES

1. https://www.citrix.com/glossary/caching.html
2. ARC: A SELF-TUNNING, LOW OVERHEAD REPLACEMENT CACHE- Nimrod Megiddo and Dharmendra S. Modha
3. http://mcicpc.cs.atu.edu/archives/2012/mcpc2012/lru/lru.html
4. https://en.wikipedia.org/wiki/Adaptive_replacement_cache