Computer Systems 2

CEN 502

Project 3 Report

Cache Replacement Policies

Aman Karnik

ASU ID: 1209536091

INDEX

1. INTRODUCTION:

A CPU cache is a cache used by the central processing unit (CPU) of a computer to reduce the average time to access data from the main memory. The cache is a smaller, faster memory which stores copies of the data from frequently used main memory locations. Most CPUs have different independent caches, including instruction and data caches, where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, etc.). The process is termed as caching and it is one of the oldest and fundamental concept that has extended today to modern computing. Whenever a requested item is available in cache it is termed as a cache hit else when the item is not present in the memory it is classified as a cache miss. After a cache miss is encountered the element is brought into the cache memory from the secondary memory so that it can be retrieved faster during future accesses.

Caches are relatively small in size as compared to secondary memories and in contrast more expensive too. Caches find applications in almost all computing devices today and its major fields of applications are databases, storage systems, web servers, processors, disk drives, RAID controllers, operating systems and so on.

There are many cache replacement policies in use today some of which are:

- Least Recently Used (LRU) – The cache memory discards the items that are least recently used.
- Most Recently Used (MRU) – The cache memory discards the item that is most recently used.
- Random Replacement (RR) – A random element is selected and discarded to make space for a newer element in this cache replacement policy.
- Adaptive Replacement (ARC) – The cache lists are dynamically arranged such that two lists along with two more ghost lists making a total of four lists are maintained such that the elements are arranged in cache such that less cache misses are obtained. It is considered to be an improvement over LRU policy.
  Moreover there are many other policies like Segmented LRU, Pseudo LRU and Least Frequently Used (LFU) etc.

In this project we have implemented the Least Recently Used (LRU) replacement policy and the Adaptive Replacement Cache (ARC) policy.

2. Design and Analysis of LRU cache replacement policy:

The LRU cache replacement policy operates on the principle of replacing the least recently used page in the cache memory. It is one of the most elementary, standard cache replacement policy in use today. The policy is efficient in its own ways but there have been some policies that build up on the framework set by LRU and improve on the overall functionality.

2.1 LRU Cache Algorithm:

- Define the variables and a Linked List for the basic structure of a cache memory.
- Two command line arguments are defined for the cache size and the file name which are provided in the script file.
- Evaluate the trace file line by line and split the respective fields into address and block length.
- If page present in cache, cache_hit+1 else cache_miss+1
- If cache hit then remove page from that location and move to most recently used (MRU) position.
- If cache miss then remove last page from cache and bring the requested page to cache at the MRU position.
- Calculating the mem_accesses and the hit_ratio
- Display the cache hits,cache misses and the hit ratio.

Program Flow:

The file is run by accepting command line arguments of cache size and filename which are predefined. The user can also be prompted for the cache size and the full path of the file manually by uncommenting the code. Each text data line read one by one and the data fields are split into the corresponding address (first field) and the block length (second field). The page is checked in the cache memory and the corresponding cache hits and cache misses are recorded. The memory accesses are then calculated and the hit ratio is calculated and displayed.

The number of memory accesses = Number of cache hits + Number of cache misses

Hit ratio = Number of cache hits / Number of memory accesses

2.2 Implementation:

Data structures used:

The LRU policy is implemented using an empty Linked List as the basic data structure for defining a cache. The standard java.util.LinkedList class and its methods are utilized for the structuring and operation of the cache block.

Some of the functions used by the Linked List in the program are:

Object.remove (int index) - Removes the element at the specified position in the list. Throws NoSuchElementException if the list is empty.

void addFirst (Object o) - Inserts the given element at the beginning of the list.

Object removeLast () - Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.

Variable declaration:

The cache_hit, cache_miss and mem_accesses are all declared as long integer so as to tackle the large values to handle during accesses to files like P4.lis.

The hit_ratio is declared as a float variable and later calculated using typecasting values for cache_hit and mem_accesses.

Rounding Off:

The variable hit_ratio has been rounded off to two digits using a java class .DecimalFormat and then displayed using the same.

The syntax for the command is as follows:

DecimalFormat df = new DecimalFormat("#.##");

*System.out.println("The hit ratio of the tracefile after rounding off:"+**df.format(hit_ratio)**+ "%");*

Where hit_ratio is a variable declared to store the hit ratio of the cache policy.

3. Design and Analysis of ARC cache replacement policy:

Adaptive Replacement Cache (ARC) is a page replacement algorithm with better performance than LRU (least recently used) developed at the IBM Almaden Research Center. This is accomplished by keeping track of both frequently used and recently used pages plus a recent eviction history for both. The ARC replacement policy is an improvement over the traditional LRU policy. The simplicity of the policy and dynamically being able to adapt to changing workloads is the main feature of this caching policy. The ARC policy maintains two lists L1 and L2 which are subcategorized into T1, B1 and T2, B2 where L1=T1+B1 and L2=T2+B2. The cache list T1 is used for recency and T2 is used for frequency whereas two ghost lists B1 and B2 are used for holding the keys of discarded elements. The size of T1 and T2 dynamically varies according to the pages encountered. T1 holds pages accessed once and T2 holds pages accesses at least twice. The ghost lists are used to hold the elements discarded from the respective lists.

The main functionality of ARC is on the basis of functioning with a dynamic workload and arranging the cache sizes respectively.

3.1 ARC Cache Algorithm:

- The variables are declared and the four linked list are defined for the basic structure of a cache memory.
- Two command line arguments are defined for the cache size and the file name which are provided in the script file.
- Evaluate the trace file line by line and split the respective fields into address and block length.
- The Replace method is defined for checking if T1 is not empty and T1 is greater than a parameter p or page is in B2 and the size of T1 equals p.
  If the above is true, delete the LRU page in T1 cache and move to the MRU position in B1 else delete the LRU page in T2 and move to the MRU position in B2.
- For CASE I : If page is present in cache recency list T1, cache_hit++ , the page in T1 is located and moved to the most recent position in cache list T2.
  If the page is present in cache frequency list T2, cache_hit++, the page in T2 is located and moved to the most recent position in cache list T2.
- For CASE II: If page is not in T1 or T2 then cache_miss++ then check if the page is in B1 cache list, dynamically change the size of B1 and call the replace method to remove the last element in T1 and move it to the first position in B1.
  Move the page from B1 to the MRU position in T2 cache list.

- For CASE III: Check if the page is in B2, if yes then cache_miss++, dynamically change the size of B2 and call the replace function to remove the last element from T2 and store it as the first element in the ghost list B2.
  Move the page from B2 to the MRU position in T2 cache list.
- For CASE IV: If the page is not in any of the four lists T1, T2, B1 or B2 then cache_miss++. For Case IV (A) the size of T1+B1 is checked to be equal to the cache size. If yes then the length of T1 is checked if it's less than the cache size and if true then the LRU page in B1 is deleted and the replace method is called. If not then the ghost list B1 is empty and the LRU page from T1 is removed.
- For Case IV (B) the size of T1+B1 is less than the cache size. If the size of T1+T2+B1+B2 is greater than cache size, delete the LRU page in B2 if T1+T2+B1+B2= 2*cache size and call the replace method.
  Lastly bring the page to the cache by moving it to the MRU position in T1.

Program Flow:

The file is run by accepting command line arguments of cache size and filename which are predefined. The user can also be prompted for the cache size and the full path of the file manually by uncommenting the code. Each text data line read one by one and the data fields are split into the corresponding address (first field) and the block length (second field). The ARC algorithm is run by checking if the page is present in the cache or not. The corresponding cache hits and misses are recorded. The total number of memory accesses are calculated and the hit ratio is calculated and displayed using the following formula:

The number of memory accesses = Number of cache hits + Number of cache misses

Hit ratio = Number of cache hits / Number of memory accesses

3.2 Implementation

Data structures used:

The ARC policy is implemented using four empty Linked Lists comprising of T1 and T2 as the lists for recency and frequency and B1 and B2 as the ghosts lists. These four linked lists form the basic data structure for defining the cache. The standard java.util.LinkedList class and its methods are utilized for the structuring and operation of the cache block.

Some of the functions used by the Linked List in the program are:

Object.remove (int index) - Removes the element at the specified position in the list. Throws NoSuchElementException if the list is empty.

void addFirst (Object o) - Inserts the given element at the beginning of the list.

Object removeLast () - Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.

Object removeLast () - Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.

Object getLast () - Returns the last element in this list. Throws NoSuchElementException if this list is empty.

Variable declaration:

All the variables in ARC are declared as static so that they can be accessed from anywhere in the entire code. The cache_hit, cache_miss, mem_accesses and hit_ratio are declared of type double so as to tackle the large values to handle during accesses to files like P4.lis.

The hit_ratio is calculated using typecasting values for cache_hit and mem_accesses.

Rounding Off:

The variable hit_ratio has been rounded off to two digits using a java class .DecimalFormat and then displayed using the same.

The syntax for the command is as follows:

DecimalFormat df = new DecimalFormat("#.##");

*System.out.println("The hit ratio of the tracefile after rounding off:"+**df.format(hit_ratio)**+ "%");*

Where hit_ratio is a variable declared to store the hit ratio of the cache policy.

4. Screenshots for the cache replacement policies:

Running LRU and ARC for P3.lis with cache sizes 1024 and 2048:

```
-bash-3.2$ sh ./runP3.sh
Least Recently Used(LRU)
The size of the cache is:1024
The cache hits are:41051
The cache misses are:3871245
The hit ratio of the tracefile is:1.0492816%
The hit ratio of the tracefile after rounding off:1.05%
Adaptive Replacement Cache(ARC)
The cache hits are:44087.0
The cache misses are:3868209.0
The hit ratio of the tracefile is:1.1268830299377441%
The hit ratio of the tracefile after rounding off:1.13%
Least Recently Used(LRU)
The size of the cache is:2048
The cache hits are:45052
The cache misses are:3867244
The hit ratio of the tracefile is:1.1515489%
The hit ratio of the tracefile after rounding off:1.15%
Adaptive Replacement Cache(ARC)
The cache hits are:60276.0
The cache misses are:3852020.0
The hit ratio of the tracefile is:1.540681004524231%
The hit ratio of the tracefile after rounding off:1.54%
```

Running LRU and ARC for P4.lis with cache sizes 1024 and 2048:

```
-bash-3.2$ sh ./runP4.sh
Least Recently Used(LRU)
The size of the cache is:1024
The cache hits are:531000
The cache misses are:19245090
The hit ratio of the tracefile is:2.6850605%
The hit ratio of the tracefile after rounding off:2.69%
Adaptive Replacement Cache(ARC)
The cache hits are:531856.0
The cache misses are:1.9244234E7
The hit ratio of the tracefile is:2.689389228820801%
The hit ratio of the tracefile after rounding off:2.69%
Least Recently Used(LRU)
The size of the cache is:2048
The cache hits are:586135
The cache misses are:19189955
The hit ratio of the tracefile is:2.9638567%
The hit ratio of the tracefile after rounding off:2.96%
Adaptive Replacement Cache(ARC)
The cache hits are:588944.0
The cache misses are:1.9187146E7
The hit ratio of the tracefile is:2.9780609607696533%
The hit ratio of the tracefile after rounding off:2.98%
```

5. Conclusion:

The two cache replacement policies are run for the different cache configurations and the values obtained are tabulated as follows:

Cache configuration	Hit Ratio for LRU	Rounded value of hit ratio for LRU	Hit Ratio for ARC	Rounded off value of hit ratio for ARC
1024 for P3.lis	1.0492816%	1.05%	1.126883029%	1.13%
2048 for P3.lis	1.1515489%	1.15%	1.540681004%	1.54%
1024 for P4.lis	2.6850605%	2.69%	2.689389228%	2.69%
2048 for P4.lis	2.9638567%	2.96%	2.978060609%	2.98%

On observation we can estimate the ARC performs at a better rate than LRU on varying cache configurations. The hits encountered in ARC are greater than those obtained in an LRU cache policy. Since the efficiency of a cache memory is judged by its hit ratio, higher the cache hit ratio better is the cache replacement policy.

Since caching is an integral part of today's modern computing society, newer and newer caching policies are always being developed. LRU has been a traditional standalone policy and the basis for many newer policies. ARC is an improvement over LRU and many other newer replacement policies will soon be developed on the skeletal structure of the above two which may be more efficient than LRU and ARC.

6. References

1. Nimrod Megiddo and Dharmendra S. Modha, ARC: A Self-Tuning, Low Overhead Replacement Cache," USENIX Conference on File and Storage Technologies (FAST'03), San Francisco, CA, pp.115-130, March 31-April 2, 2003.

2. Computer Organization and Design: The Hardware Software Interface(4th Edition)- David A. Patterson and John L. Hennessey, Chap 5 Large and Fast Exploiting Memory Hierarchy Pg. 452- Pg. 548.

3. Computer Architecture: A Quantitative Approach(5th Edition)- David A. Patterson and John L. Hennessey, Chap 2 Memory Hierarchy Design Pg. 72- Pg. 131.

4. https://docs.oracle.com/javase/7/docs/api/java/util

5. http://www.stackoverflow.com

6. http://www.tutorialspoint.com/java