CEN 502 FALL 2015

Project # 3 Report

Implementation of an LRU cache replacement algorithm and the ARC cache replacement algorithm on real-life workloads

OMKAR SALVI
ASU ID : 1209536104

LRU replacement Policy:

Implementation-

1) Data structures used:

- **Queue –** The cache is implemented as a queue. A Queue is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of pages a cache can store i.e. cache size in terms of number of pages. The most recently used pages will be near front end and least recently pages will be near rear end. The queue will maintain this requirement of keeping LRU and MRU pages at extreme ends. Because of this to flush a page from cache or inserting a new page into cache becomes easier.

- **Doubly Linked List –** To implement the queue I have used doubly linked list. The Linked List was selected for implementing the queue because It takes O(1) time i.e. constant time to insert or remove a node from list. In this project as we are implementing the cache. Cache will have insertion and deletion operations all the time.
  Every page in the cache is represented as a node in the doubly linked list. A class is created to define a node. Each node in the linked list will have 3 members –
  1. Data (page number) : Integer
  2. Next node : object of node class
  3. Previous node : object of node class

  The class for doubly linked list will have 3 members –
  1. First node : object of node class to hold start of the queue
  2. Last node : object of node class to hold end of the queue
  3. Current size of the Linked list : Integer

- **Hash table –**
  - ➢ A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.
  - ➢ Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash *collisions*—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way. For this a list of entries with the same index is maintained. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.
  - ➢ In my program, a Hash with page number as key and the corresponding node in the linked list as value is implemented. Hash table is used to know whether a page is present in the cache i.e. queue. When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue. If the required page is not in the memory, we bring that in memory.

  - ➢ In simple words, we add a new node to the front of the queue and update the node corresponding to that page in the hash table. If the queue is full, i.e. queue size equal to cache size, we remove a node from the rear of queue, and add the new node to the front of queue.

2) Representation of data:
   - The page numbers were stored as integer data. The starting block numbers from input trace file were stored in an arraylist 'startblk[]' and number of blocks in input trace file were stored in another arraylist 'numofblks[]'.
   - An integer variable 'page' was initialized to hold the page number which is being accessed at a time.
   - The variables hit and miss of type integer were used to calculate the number of cache hits and cache misses.
   - Hit ratio is stored in variable of type double as the hit ratio will be a decimal value.

3) Hit ratio logic:
   - To count the number of hits and misses I have initialized 2 variables called 'hit' and 'miss' of type double. Whenever hash table finds an entry for the accessed page I increment the variable 'hit'. Similarly whenever hash table does not find an entry for the accessed page I increment the variable 'miss'. At the end to calculate hit ratio I divide the variable 'hit' with the sum of variables 'hit' and 'miss' and typecast the result in double. Then multiply it by 100 to get percentage hit ratio. Then rounded off the result till 2 decimal point precision using Math library.

4) Features to improve complexity:
   - **Hash table**
     - ➢ Whenever a page is accessed we need to check whether that page is present in cache or not. Now our implementation of cache is based on doubly linked list. And to search for a node in Linked list takes O(n) time, where n is the size of the linked list. Therefore for each page access it needs to scan through whole linked list which is time consuming.
     - ➢ In addition large number of pages is being accessed by our input file and thus the time required to use the whole data in the code will be very large. The worst case scenario is when the page is not present in the cache. Program will search whole linked list and at the end it will know that page is not present in the cache.
     - ➢ To alleviate this situation I used hash table. So hash table will detect whether a page is present in the cache or not. Searching for a page in hash table will take constant time. If a page is not present in the cache program will not waste time in searching for it in linked list and take the appropriate action.
     - ➢ If page is found in the hash table, it returns the node holding that page. Thus we found the page present in cache. Then we can take the required action.
     - ➢ To search for a page in hash table takes O(1) i.e. constant time. Thus program will complete in less time and time complexity will be reduced.

LRU Output for input trace file P4.lis :

Cache size	%Hit ratio observed	%Hit ratio from paper	Program Run Time
1024	2.69	2.68	< 10 sec
2048	2.96	2.97	< 10 sec
4096	3.32	3.32	< 10 sec
8192	3.65	3.65	< 10 sec
16384	4.07	4.07	< 10 sec
32768	5.24	5.24	< 10 sec
65536	10.76	10.76	< 10 sec
131072	21.43	21.43	< 10 sec
262144	37.28	37.28	< 10 sec
524288	48.19	48.19	< 20 sec

ARC replacement Policy:

Implementation-

1) Data structures used:
   - **Queue –** The cache is implemented as 2 queues. A Queue is implemented using a doubly linked list. The maximum size of the queues will be equal to 'p' and (c-p) where 'p' equal to adaptive parameter and 'c' equal to cache size in terms of number of pages. The once recently referred pages will be in queue with size 'p' and at least twice recently referred pages will be in queue with size 'c-p'. Most recently referred pages will be near front end and least recently referred pages will be near rear end in all the queues. The queue will maintain this requirement of keeping LRU and MRU pages at extreme ends. Because of this to flush a page from cache or inserting a new page into cache becomes easier.

   - **Doubly Linked List –** To implement the queue I have used doubly linked list. The Linked List was selected for implementing the queue because It takes O(1) time i.e. constant time to insert or remove a node from list. In this project as we are implementing the cache. Cache will have insertion and deletion operations all the time. Every page in the cache is represented as a node in the doubly linked list. A class is created to define a node. To implement the queue I have used doubly linked list.
     Each node in this list will have 3 members –
     1. Data (page number) : Integer
     2. Next node : object of node class
     3. Previous node : object of node class

   The class for doubly linked list will have 3 members –
     1. First node : object of node class to hold start of the queue
     2. Last node : object of node class to hold end of the queue
     3. Current size of the Linked list : Integer

- **Hash table –**

  - A hash table (hash map) is a data structure used to implement an associative array, a structure that can map keys to values. A hash table uses a hash function to compute an *index* into an array of *buckets* or *slots*, from which the desired value can be found.
  - Ideally, the hash function will assign each key to a unique bucket, but it is possible that two keys will generate an identical hash causing both keys to point to the same bucket. Instead, most hash table designs assume that hash *collisions*—different keys that are assigned by the hash function to the same bucket—will occur and must be accommodated in some way. For this a list of entries with the same index is maintained. The time for hash table operations is the time to find the bucket (which is constant) plus the time for the list operation.
  - In my program, a Hash with page number as key and the corresponding node in the linked list as value is implemented. Hash table is used to know whether a page is present in the cache i.e. queue. When a page is referenced, the required page may be in the memory. If it is in the memory, we need to detach the node of the list and bring it to the front of the queue. If the required page is not in the memory, we bring that in memory.
  - In simple words, we add a new node to the front of the queue and update the node corresponding to that page in the hash table. If the queue is full, i.e. queue size equal to cache size, we remove a node from the rear of queue, and add the new node to the front of queue.

2) Representation of data:
   - The page numbers were stored as integer data. The starting block numbers from input text file were stored in an arraylist 'startblk[]' and number of blocks in input text file were stored in another arraylist 'numofblks[]'.
   - An integer parameter 'page' was initialized to hold the page number which is being accessed at a time.
   - The lists of pages TOP 1 (T1), TOP 2 (T2), BOTTOM 1 (B1), BOTTOM 2 (B2) are implemented as the queues based on doubly linked list.
   - To search for a page in the each of these 4 queues I have implemented 4 Hash tables, one for each queue.
   - The variables ARC_hit and ARC_miss of type integer were used to calculate the number of cache hits and cache misses respectively.
   - Hit ratio is stored in variable of type double as the hit ratio will be a decimal value.

3) Rounding or truncation:
   - For calculation of adaptive parameter 'p' I have used rounding mechanism. I calculate the delta1 and delta2 variables as double type and then use rounding function defined inside 'Arc1' class to round off the value of delta to nearest integer.
   - That means if digit after decimal point is less than 5 then drop the decimal and return integer value of delta. Otherwise increment delta to nearest high integer and return.

4) Hit ratio logic:

- To count the number of hits and misses I have initialized 2 variables called 'hit' and 'miss' of type double. Whenever hash table finds an entry for the accessed page I increment the variable 'hit'. Similarly whenever hash table does not find an entry for the accessed page I increment the variable 'miss'. At the end to calculate hit ratio I divide the variable 'hit' with the sum of variables 'hit' and 'miss'. Then multiply it by 100 to get percentage hit ratio. This value is then rounded off to 2 decimal point precision by using round function from Math library.

5) Features to improve complexity:

- **Hash table**
  - ➢ Whenever a page is accessed we need to check whether that page is present in cache or not. Now our implementation of cache is based on doubly linked list. And to search for a node in Linked list takes O(n) time, where n is the size of the linked list. Therefore for each page access it needs to scan through whole linked list which is time consuming.
  - ➢ In addition large number of pages is being accessed by our input file and thus the time required to use the whole data in the code will be very large. The worst case scenario is when the page is not present in the cache. Program will search whole linked list and at the end it will know that page is not present in the cache.
  - ➢ To alleviate this situation I used hash table. So hash table will detect whether a page is present in the cache or not. Searching for a page in hash table will take constant time. If a page is not present in the cache program will not waste time in searching for it in linked list and take the appropriate action.
  - ➢ If page is found in the hash table, it returns the node holding that page. Thus we found the page present in cache. Then we can take the required action.
  - ➢ To search for a page in hash table takes O(1) i.e. constant time. Thus program will complete in less time and time complexity will be reduced.

ARC Output for input trace file P4.lis :

Cache size	%Hit ratio observed	%Hit ratio from paper	Program run Time
1024	2.69	2.69	< 10 sec
2048	2.98	2.98	< 10 sec
4096	3.49	3.50	< 10 sec
8192	4.16	4.17	< 10 sec
16384	5.76	5.77	< 10 sec
32768	11.23	11.24	< 20 sec
65536	18.54	18.53	< 20 sec
131072	27.45	27.42	< 20 sec
262144	40.19	40.18	< 20 sec
524288	53.37	53.37	< 20 sec

Problems in implementing the protocol:

- Null pointer error in maintaining the next and previous nodes of a node at head and tail of the queue is important. When adding a new node or removing an existing node we need to make changes in the neighboring node and also make sure to take special action for node at head and tail of the queue.
- Updating the head and tail of the queue at each insertion and removal of nodes at head and tail of the queue. This is because they define the start and end of the queue. Without maintaining them correctly we are bound to get incorrect operation.
- Maintaining size of the list at each access of the page i.e. maintaining the size of the cache
- While inserting and removing a node, updating both queue & hash table is necessary. Because search for a page is done in hash table while node operation is done on queue. Therefore both of them should be synchronized and one will reflect the changes made in another.

Conclusion:

- We have implemented a LRU replacement algorithm and also the ARC replacement algorithm on real-life workloads which are the traces of input capture disk accesses by databases, web servers, NT workstations, and a synthetic benchmark for storage controllers and calculated the hit ratio of the cache
- From the hit ratios observed we can conclude that ARC policy has better performance than that of LRU policy, since hit ratio of ARC is greater than that of LRU in most of the cases. This is because ARC is an adaptive algorithm which adapts the pages in cache as per the accesses of the pages by user.
- We also varied the cache size and observed the hit ratios. We can thus conclude from our observations that hit ratio increases with increase in cache size irrespective of the replacement policy used.
- Also the time required to implement the replacement algorithm increases as the cache size increases.

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] http://www.cs.princeton.edu/courses/archive/spr04/cos126/hello/linux

[3] http://coding-geek.com/how-does-a-hashmap-work-in-java/

[4] java/https://www.youtube.com/watch?v=r59xYe3Vyks&list=PLS1QulWo1RIbfTjQvTdj8Y6yyq4R7g-Al