Operating Systems CS 232

Page replacement algorithms

Understanding various page replacement algorithms by implementing in C++

Introduction :

The page replacement algorithm decides which memory page is to be replaced. The process of replacement is sometimes called swap out or write to disk. Page replacement is done when the requested page is not found in the main memory (page fault). Below we have implemented the four replacement algorithms along with explanation of their implementation, complexity analysis and inference.

Structure of the test cases :

We have considered three important test cases for each of the algorithms :

a) No locality test case (random pages) :

Our first test has no locality, which means that each number is a random page within the set of accessed pages. This allows us to test the efficiency of the algorithms for randomly accessed pages. Though the possibility of such a case occurring in real life is very low. We vary the table size from 5 (small) to 10 (all the pages covered) .

b) 80 - 20 test case :

This test case exhibits locality. In this 80 % of the references are made to the 20 % of all the pages, and 20 % of the references are made to the rest 80 % of the pages. This approximates the real life usage of the page replacement algorithms, due to the principle of locality.

c) Looping test case :

This test case can be used to measure the **worst case time** of algorithms such as LRU and FIFO. These algorithms, under a looping-sequential workload, kick out older pages, unfortunately, due to the looping nature of the workload, these older pages are going to be accessed sooner than the pages that the policies prefer to keep in cache. This can lead to even zero page hits. For example, in our examples, only when the table is very large (such as 15) can we get proper hits for LRU and Approx LRU.

Complexity :

In this project, we will be calculating the complexity of the page replacement algorithm specifically.The worst case time complexity here refers to the time needed to find whether a page exists or not, and if not , suggest which page to replace. The time for the actual page replacement, and the memory retrieval is dependent on the hardware.

The number of page faults occurring for a sequence of pages also is very useful for determining the usefulness of a replacement algorithm.

1. Least Recently Used algorithm using stack :

Description :

We implement least recently used algorithm using stack.whenever a page is found move it to the top of the stack.whenever page fault occurs pop a page from the stack and push the new page.

Implementation :

since we have to check whether the page is in the stack or not, we used a vector whose size is number of frames to implement this.

- If the vector is empty we just push the page into the vector
- If the page vector is not full and page is not found ,then we push the page ito the vector
- If the page is found,then move it to the top(end) of the vector
- If the page is not found and vector is full then pop the page from the vector and push the new page into the vector.

Complexity :

Since we are giving 'n' number of pages as our input, for each page we have to do search operation on the vector whose size is f .So the number of steps(iterations) this algorithm requires is o(f) Where f is number of frames for one page.

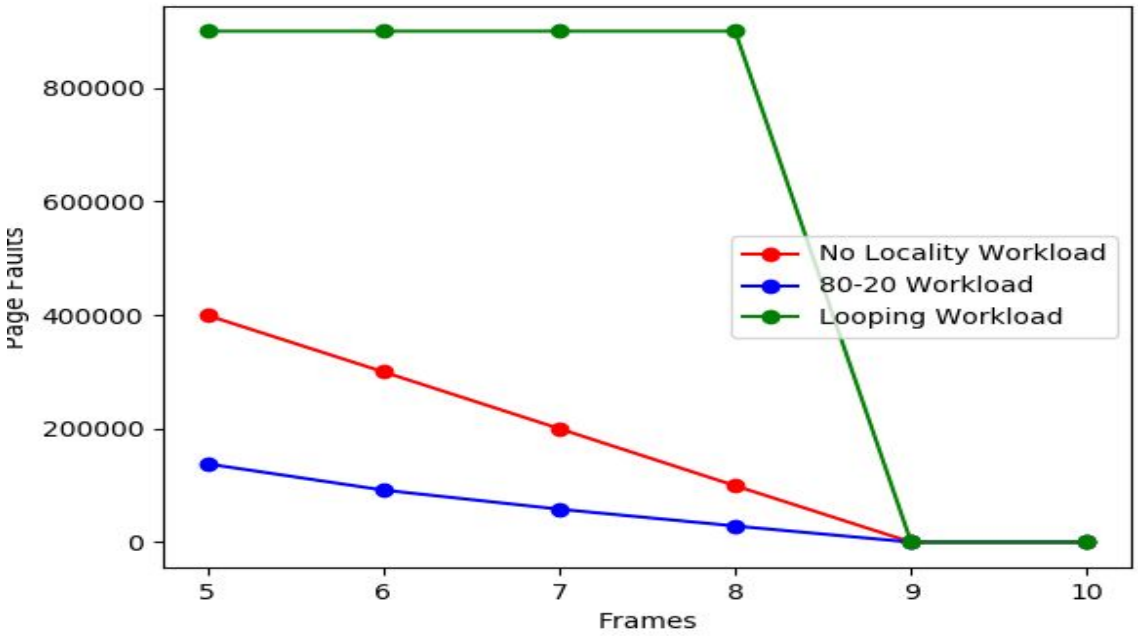
The worst case running time of the algorithm is O(f) i.e, running time of 1 page and where f=number of frames.

T(n)= Time taken to run the n pages.

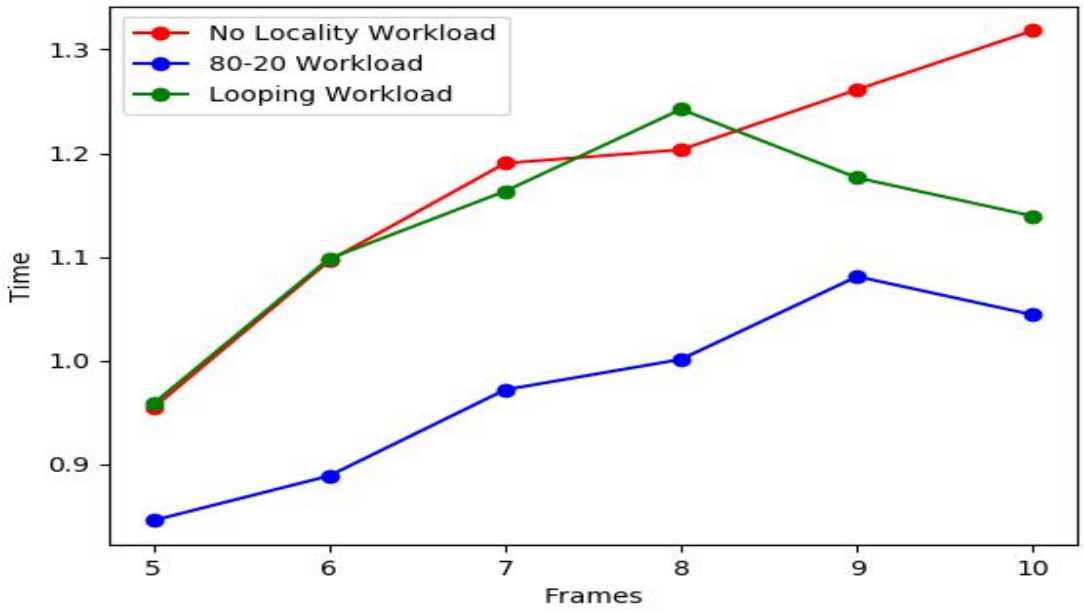
T(n) = Time taken to traverse cache + Time taken to check the page is there or not + Time to replace the page.

T(n)=O(n*f) where n=no of pages and f= no of frames

Page fault graphs :



Time graphs :



2. Least Recently Used algorithm using counter :

Description :

Every page entry has a corresponding counter .Ever time a page is referenced through this entry,copy the clock into this counter. When a page is need to be change see the counters which has minimum count and change that.

Implementation :

we have used a vector of size number of frames for pages and a corresponding vector of size number of frames to store the counter values to implement this algorithm

- set counter = 0 at the beginning.
- after taking each page as input increment the counter

If the page is not found and vector is not full then add the page to the vector and also add the counter value to the corresponding frame in the counter vector.

If the page is not found and vector is full then replace the page that has least counter value

If the page is found then update its corresponding counter value in the counter vector.

Complexity :

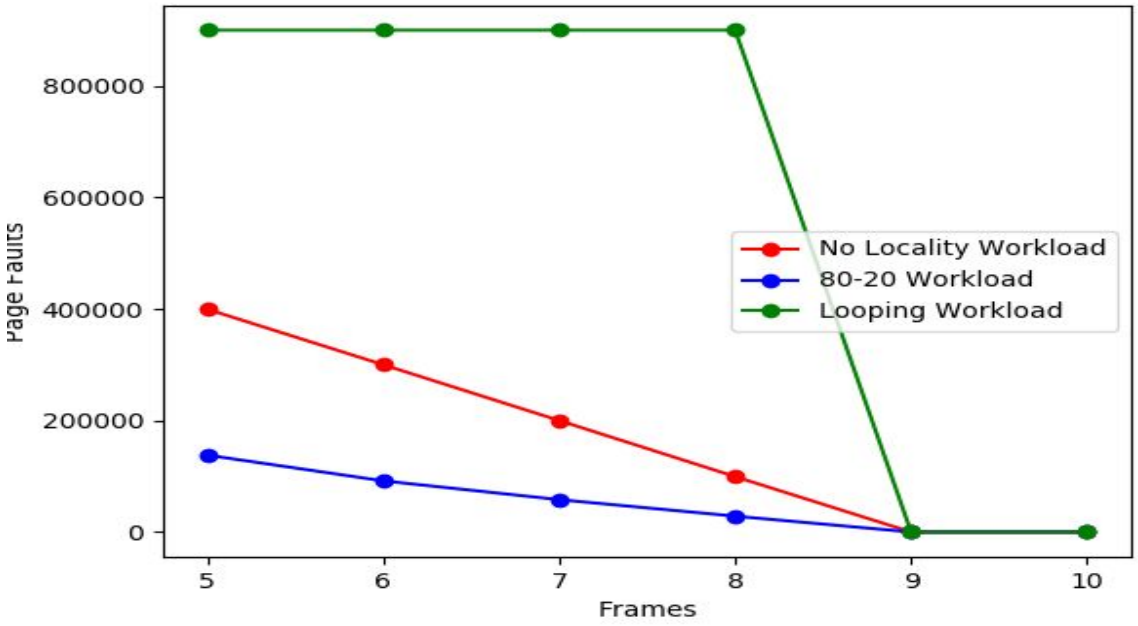
In this algorithm also we are giving 'n' number of pages as our input, for each page we have to do update two vectors i.e.., counter and frames which takes time in order of number of frames i.e, O(f)

T(n)= Time taken to run the n pages.

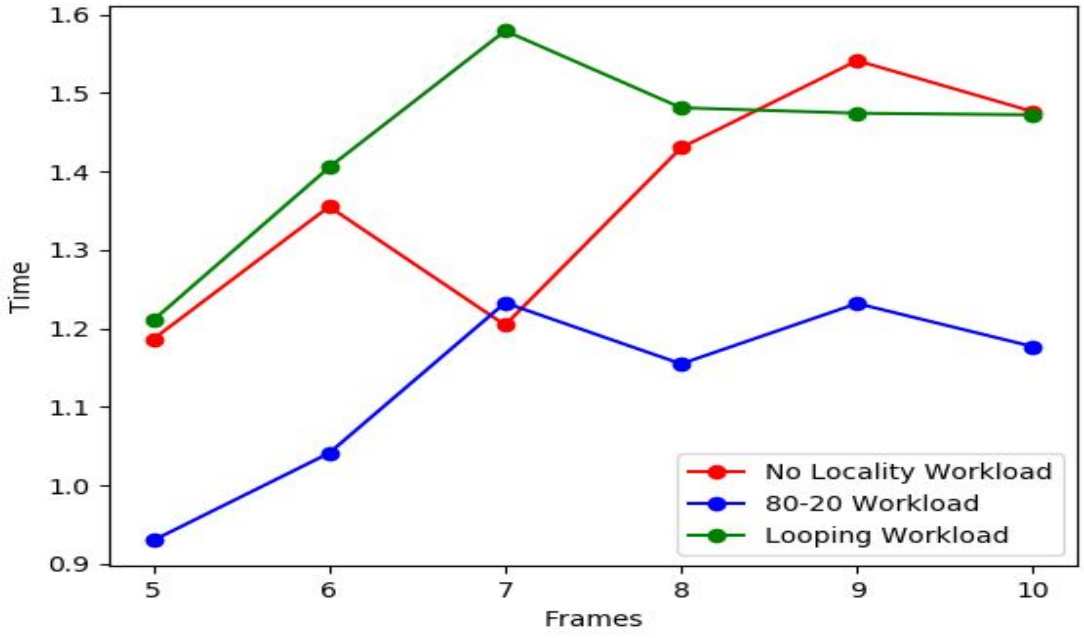
The worst case running time of n pages in the algorithm is O(n*f).

T(n) = Time taken to traverse cache + Time taken to check the page is there or not in first vector + Time taken to update counter vector+Time to replace the page.

Page fault graphs :



Time graphs :



3. Approximate Least Recently Used algorithm using Clock method :

Description :

This method tries to get the benefits of using LRU, while at the same time, tries to reduce the performance issues and implementation difficulties. Here we maintain a reference bit for each page in the cache. When a page fault occurs and we need to evict a page from the cache, we will check the reference bit of each of the pages currently present in FIFO order. If the reference bit is set, then we will clear it and search for the next page in order. When we find a page whose reference bit is not set, we will replace that page.

Implementation :

- An array is used for the page table.
- A second array called second_chance[ ] is used to hold the reference bit of a page in memory.
- We use a pointer to store the location of the recently swapped page, to implement the FIFO nature. The pointer moves through the array in a circular fashion, to find which page to replace.

Complexity :

The worst case for Approximate LRU is when the reference bits of all the pages are set. Then, the pointer moves across the whole cache one by one in circular way, and all the reference bits are set one by one to zero. Finally a page is evicted and replaced.

Let frame size = n.

In case of page hit, the time taken would be the time taken to traverse the array, which represents the cache.

Then, for successful search for page, time would be n.

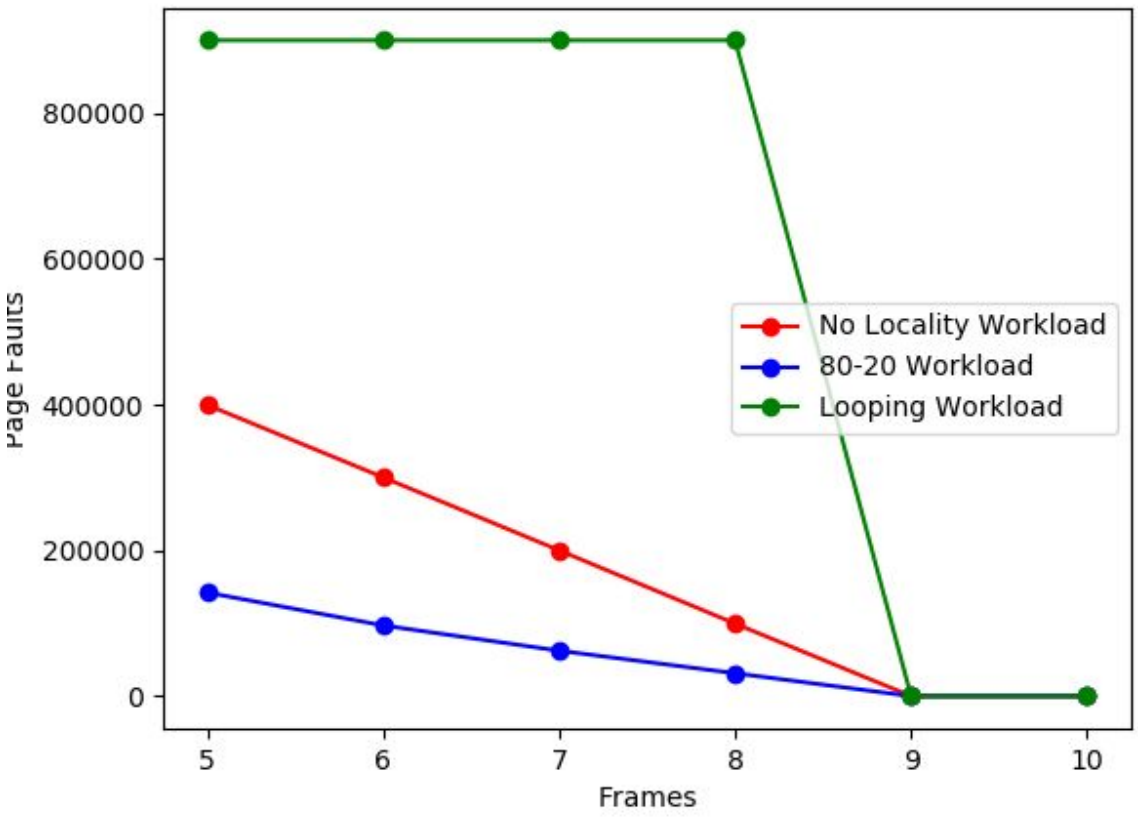
In case page fault occurs and we need to find which page to replace, in worst case, we have to search the reference bits of all existing pages.

So for finding the replacement page, time would be n.

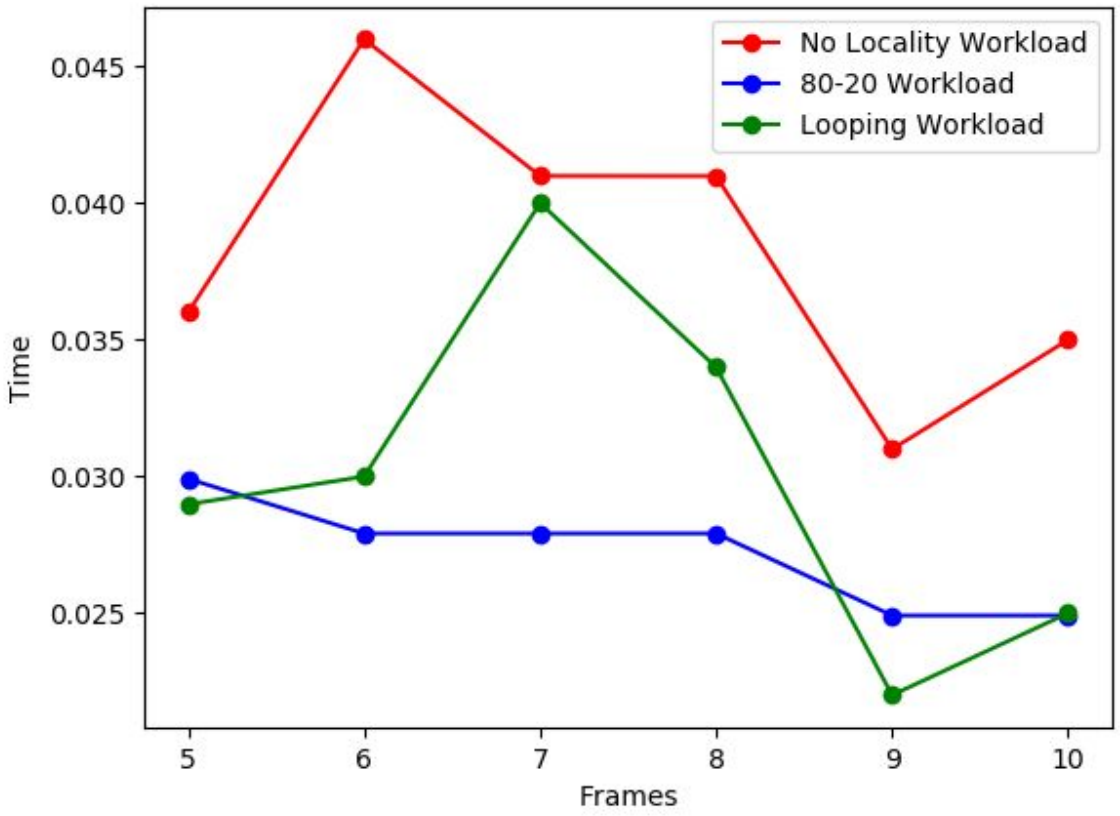
The worst case running time of the algorithm is O(n).

T(n) = Time taken to traverse cache + Time taken to check the second chance array + Time to replace the page.

Page fault graphs :



Time graphs :



Since the implementation of approximate clock is much simpler, the time taken to find the hit page, or the time required to find the page to remove would be very small. A slight issue is that approximate LRU is not as efficient as LRU in case of page faults, but it is very close to LRU.

No locality test case :

Approximate LRU does not have any advantage here as it relies on principle that pages referenced recently would be referenced again in the future.

80-20 test case :

Since this test relies on locality, LRU has good performance as 80 % of the pages are the 20 % hot pages.

Looping test case :

This test case signifies the worst case scenario for LRU and FIFO. Since we may remove a page much earlier than we would like , if we face the removed page again, it leads to a lot of page faults, sometimes causing even 0 page hits. Only when the cache can store more than 10 pages (total number of pages) , there are page hits, as we are able to store the pages until the next time they are referenced.

4. Not Frequently Used Algorithm using aging registers :

Description :

The not frequently used (NFU) page replacement algorithm requires a counter, and every page has one counter of its own which is initially set to 0. Whenever a page is referenced the counter of that page is incremented. The counters keep track of how frequently a page is used. In the traditional implementation, the drawback was that the frequency was always increasing with time, and there was no information showing whether the page was used recently or before a long time. So we implement aging, where the counter is divided (right shifted every clock cycle) so that the counter values decrease for the pages which have not been referenced recently.

Implementation :

- An array is used for the page table.
- Another array called frequency is used to store the counter values.
- When searching for a page, if a page is present, then the frequency counter value is shifted right and 1 is added in the MSB of the counter, to indicate that the page was referenced recently.
- If a page is not present, then page fault occurs.
- In page fault, we try to find the best page to replace. Here the best page would be the page with the least frequency.
- After placing a page, the value of the frequency is made 1.
- During every clock cycle, the values in the frequency arrays are decreased exponentially to show that they were not called recently.

Complexity :

Consider the number of frames as n,

Then in case of page hit, we have to traverse the whole array, so the complexity is O(n).

In case the page is not present then we have to traverse the frequency array to find which page to replace, which is again of complexity of O(n).

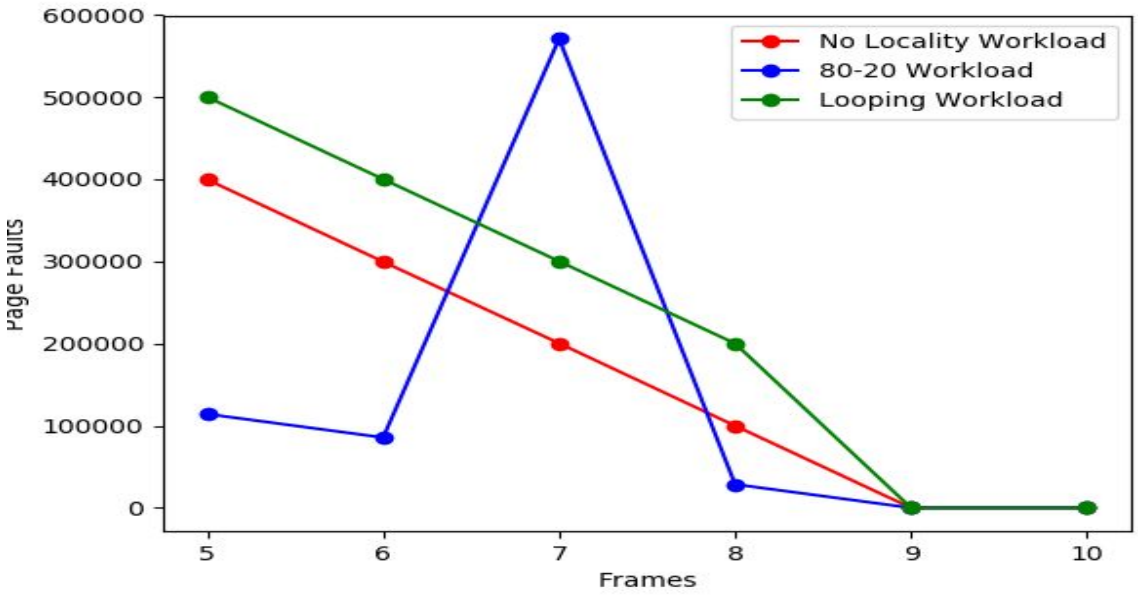
During every clock cycle, we have to divide the frequency values, which is also of O(n) complexity.

Under worst case, we will be doing all the three activities, leading to the worst case complexity of O(n).

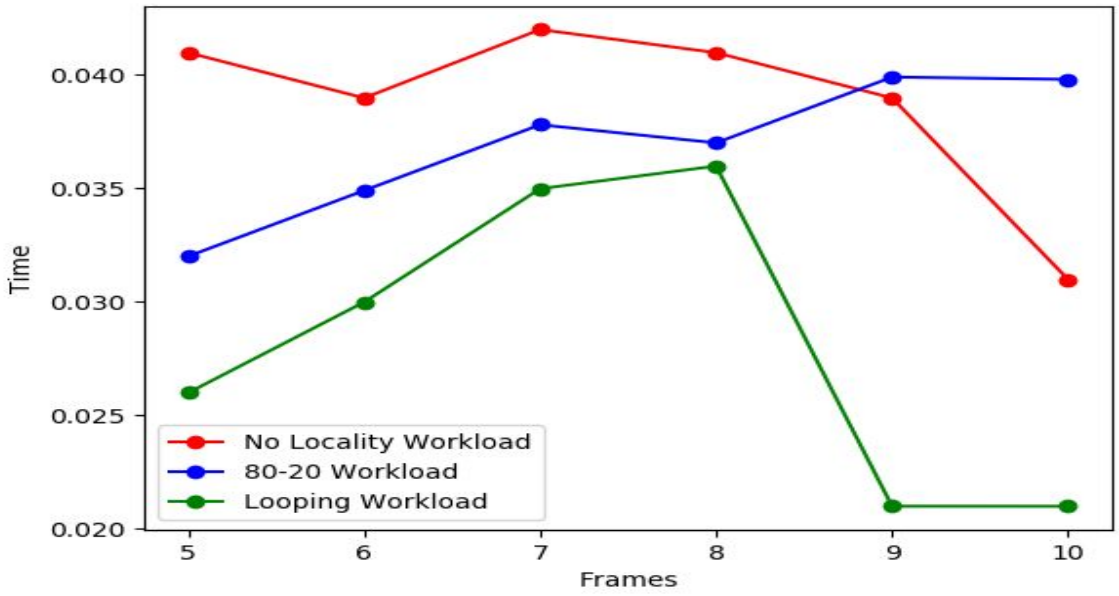
If page fault occurs, the time required to access the memory would be,

T(n) = Time taken to traverse cache + Time taken to divide the frequency array + Time taken to find evicted page + Time to replace the page .

Page fault graphs :



Time graph :



Conclusion :

The four page replacement algorithms have been discussed and their complexity has been calculated. The advantages of the algorithms have been discussed and their performance has been analysed.