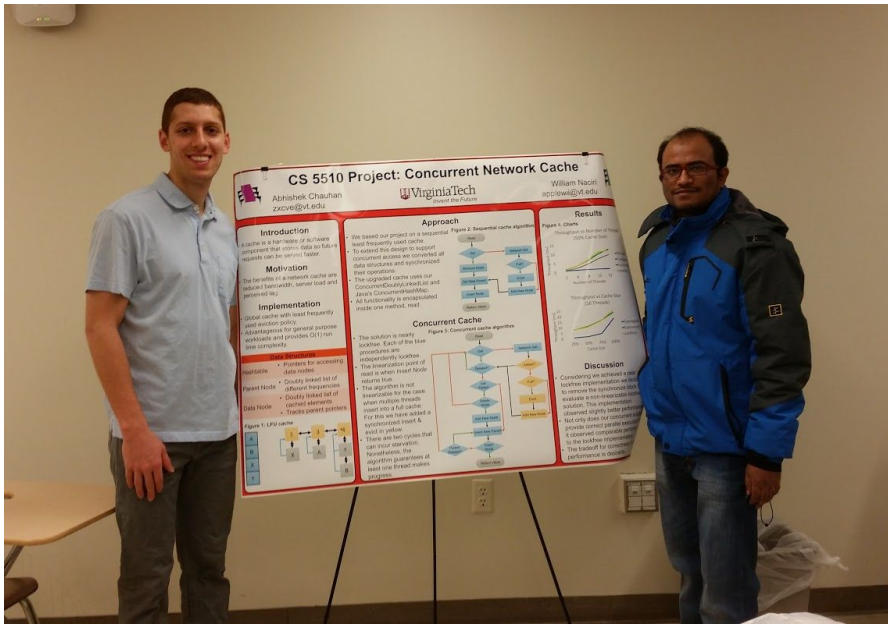
CS 5510: Multiprocessor Programming

Concurrent Network Cache

Abhishek Chauhan, William Naciri



Introduction

A cache is a hardware or software component that stores data so future requests for that data can be served faster. Cache performance is critical to the overall performance of a multiprocessor architecture.

Motivation

At best a typical cache miss might delay a processor for a hundred cycles. However, a network cache miss will delay a processor for thousands of cycles. Not only does caching improve network performance but can also reduce server load, and decrease perceived lag of the client. The greater the requests that can be served from the cache, the quicker system performance becomes.

In this class we have mostly discussed performance of algorithms on thread local caches. In our project we explore implementing a global cache with a network application. This is extremely beneficial for applications such as web browsers. Internet browsers use caching to store HTML web pages [1]. More importantly, most modern browsers are multithreaded. This implies the need for a thread-safe network cache. An excellent use case of our solution is to increase network performance for multiple web browser tabs.

Approach

Several authors have described web caching design in detail [2, 3] but merely give sequential implementations. The multiprocessor revolution is upon us and concurrent implementations are necessary. However, the sequential design is a good place to start our approach.

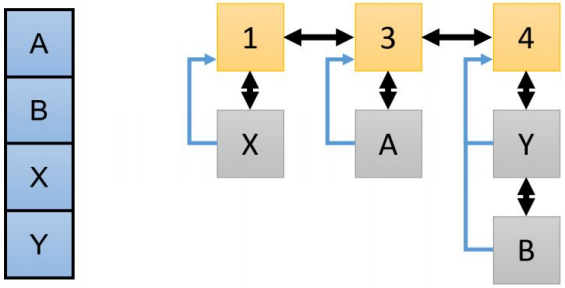
Web caches need to store most popular items. Typically, many static resources such as images, cascading style sheets, and javascript code can be cached for a fairly long time before being replaced by newer versions. These static assets are included on almost every page and must be served on every request. The benefits of caching assets are obvious but selecting an appropriate eviction policy is arguable. Our solution features a least frequently used eviction policy to retain the most popular items. To extend the sequential design to support concurrent access we converted all data structures and synchronized their operations. The upgraded cache uses our ConcurrentDoublyLinkedList and Java's ConcurrentHashMap. All functionality is encapsulated inside one method, read.

Implementation

Data Structures:

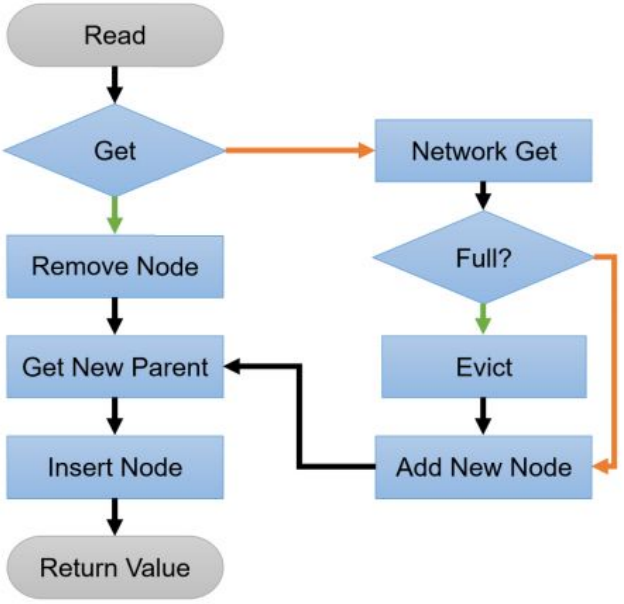
Data Structures	
Hashtable	• Pointers for accessing data nodes
Parent Node	• Doubly linked list of different frequencies
Data Node	• Doubly linked list of cached elements • Tracks parent pointers

Figure 1: LFU cache



The LFU Cache consists of a hash map which uses key as the address and the value as the node which contains the data for the queried address. This is required for O(1) time in finding the node . We are maintaining 2 linked lists; one on the access frequency and one for all elements that have the same access frequency. A hash table is used to access elements by key. A doubly linked list is used to link together nodes which represent a set of nodes that have the same access frequency (shown as yellow nodes in the diagram above). We refer to this doubly linked list as the frequency list. This set of nodes that have the same access frequency is actually a doubly linked list of such nodes (shown as grey nodes in the diagram above). We refer to this doubly linked list (which is local to a particular frequency) as a datanode list. Each node in the node list has a pointer to its parent node in the frequency list. Hence, nodes X will have a pointer back to node 1, nodes Y and B will have a pointer back to node 4 and so on.

Sequential Design:



Initially, the LFU cache starts off as an empty hash map and an empty frequency list. When the first element is added, a single element in the hash map is created which points to this new element (by its key) and a new frequency node with a value of 1 is added to the frequency list. It should be clear that the number of elements in the hash map will be the same as the number of items in the LFU cache. A new node is added to 1's frequency list. This node actually points back to the frequency node whose member it is. For example, if X was the node added, then the node X will point back to the frequency node 1. Hence the runtime complexity of element insertion is O(1).

When this element is accessed once again, the element's frequency node is looked up and its next sibbling's value is queried. If its sibbling does not exist or its next sibbling's value is not 1 more than its value, then a new frequency node with a value of 1 more than the value of this frequency node is created and inserted into the correct place. The node is removed from its current set and inserted into the new frequency list's set. The node's frequency pointer is updated to point to its new frequency node. For example, if the node A is accessed once more then it is removed from the frequency list having the

value of 3 and added to the frequency list having the value of 4. Hence the runtime complexity of element access is O(1).

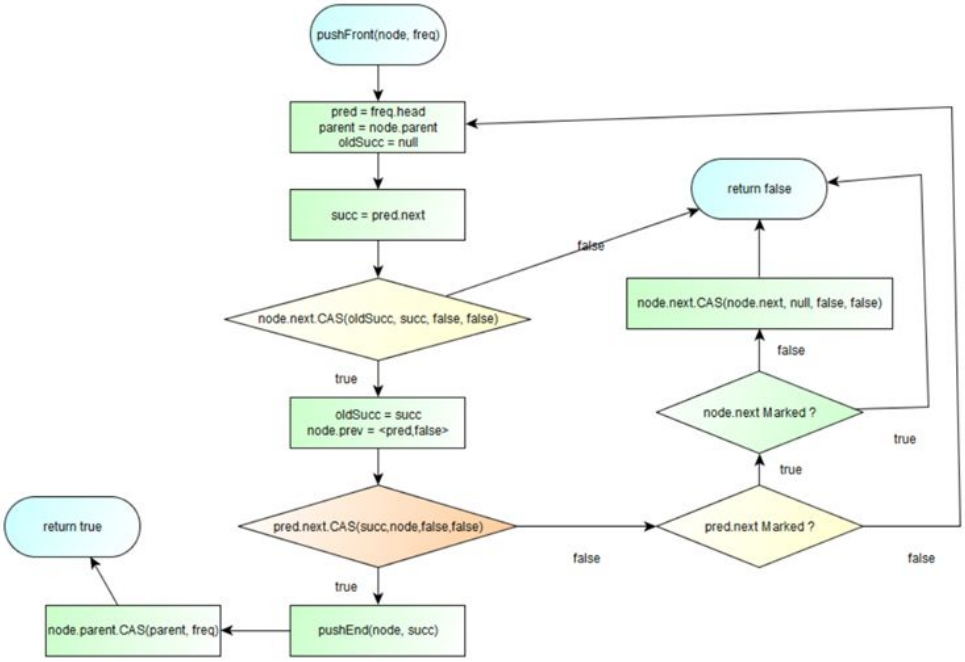
When an element with the least access frequency needs to be deleted, any element from the 1st (leftmost) frequency list is chosen and removed. If this frequency list's node list becomes empty due to this deletion, then the frequency node is also deleted. The element's reference from the hash map is also deleted. Hence the runtime complexity of deleting the least frequently used element is O(1).

Concurrent Cache:

The concurrent cache uses ConcurrentHashMap from java and ConcurrentDoublyLinkList[4] by Sundell. We will briefly explain ConcurrentDoubllyLinkList which we have implemented. The lock free doubly linklist uses next pointer as main pointer and treats the prev pointer as auxiliary pointer. It is fine to have delayed prev pointer update as all synchronization uses next pointer update as the successful/failed operation. We discuss all the APIs of doubly link list before going through the concurrent cache design.

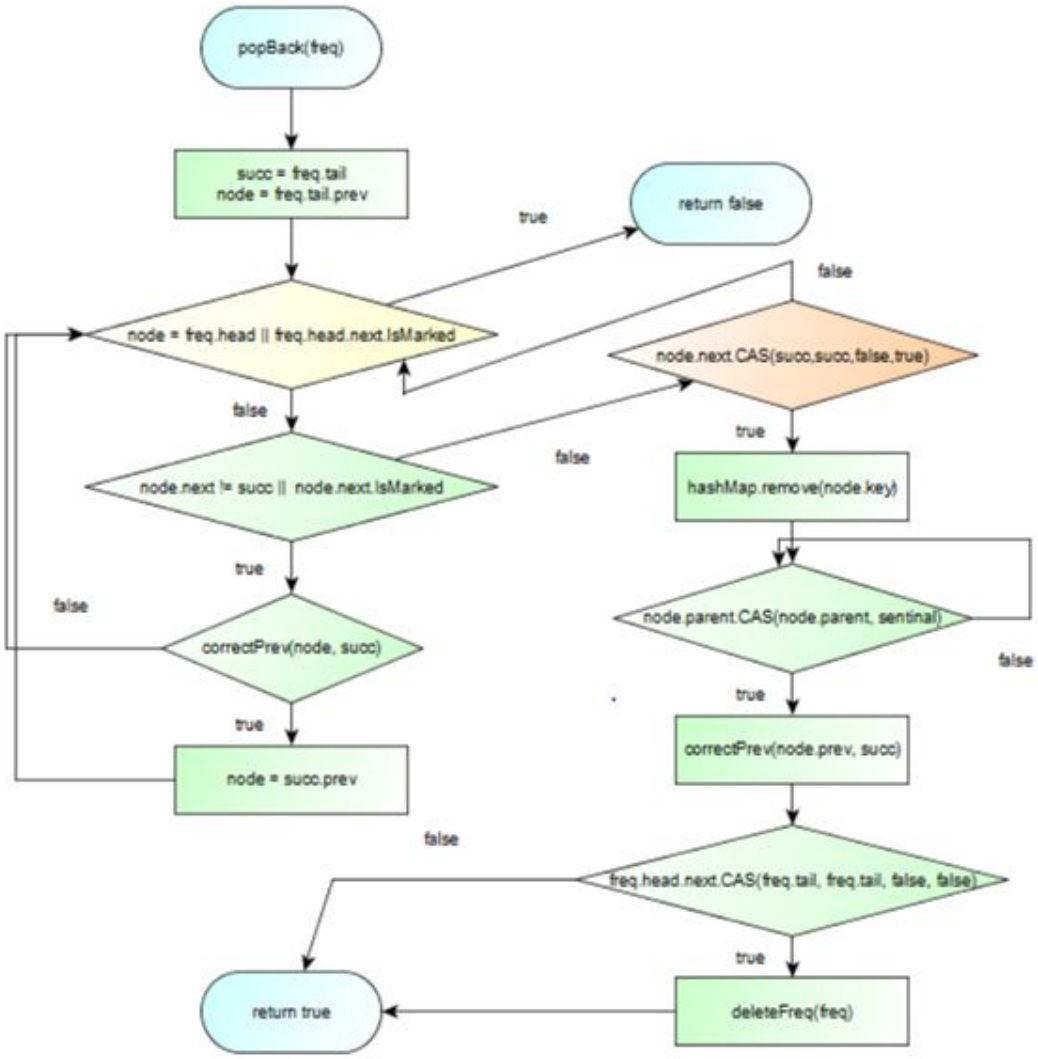
Operations for dataNode list:

PushFront:



The responsibility of this API is to insert the dataNode after the head pointer of the parent frequency node. This API can fail when the parent is deleted or the node being pushed has been deleted. If the CAS for head.next to succ fails then it is retried if parent is not deleted on the assumption that some other node might have succeeded in putting the node. If the head.next to succ CAS succeeds then a helper function pushEnd is called to fix the prev pointer for the succ. This is followed by swinging the parent pointer to the new parent.

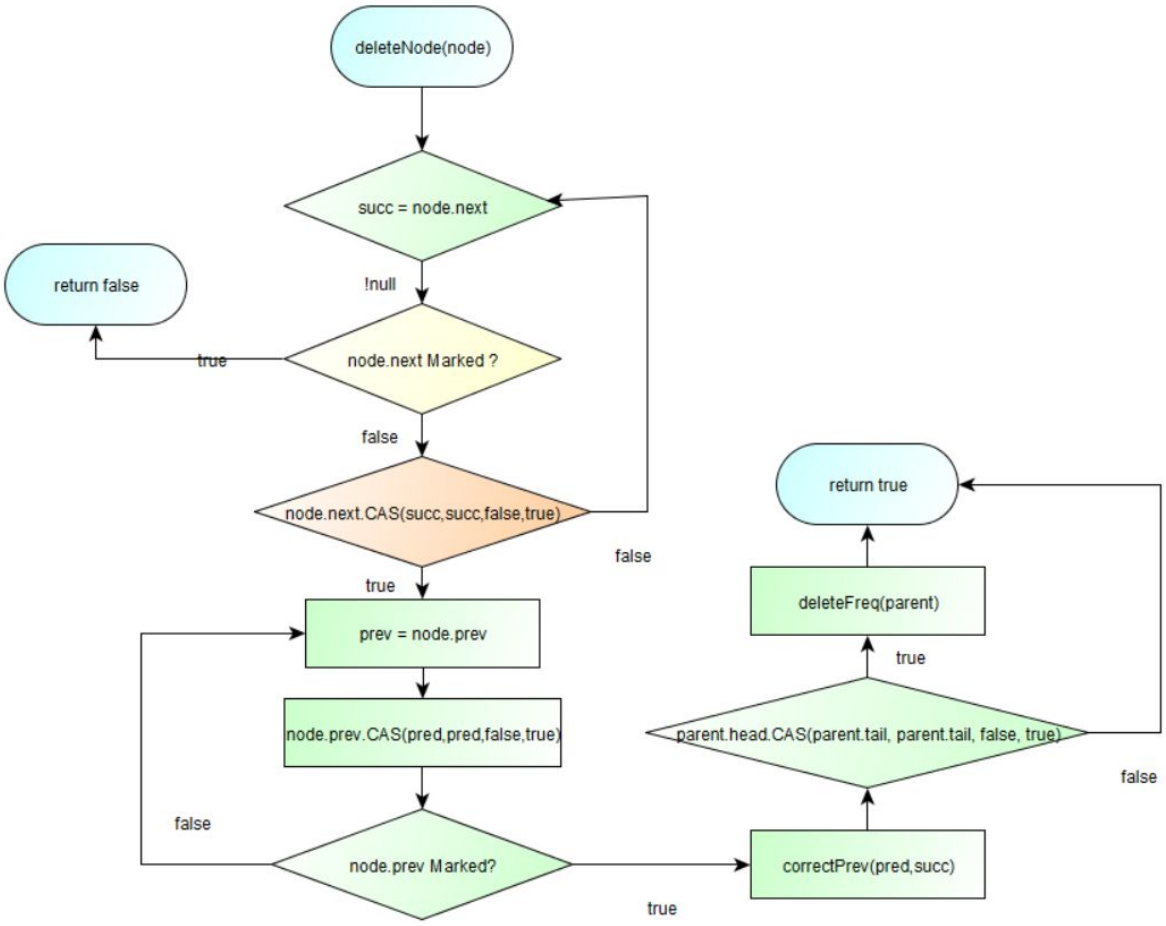
PopBack:



The responsibility of this API is to remove the datNode from the tail pointer of the parent frequency node. This API is called by evict to pop a node from the the first frequency list. This API can fail if the freq list has been deleted or there is no node in the freq list(can happen due to delayed insert or delayed delete). Once a node is popped it checks if that is the last node in the freq list, if true then it deletes the freq list. After popping the node, it tries to fix prev pointer for tail by calling correctPrev for tail. This call will also perform physical deletion for the node just popped. Also if node being popped is no longer the

last node, then it retries the operation by fixing prev pointer for tail pointer(calls correctPrev) and tries to pop the returned node from correctPrev.

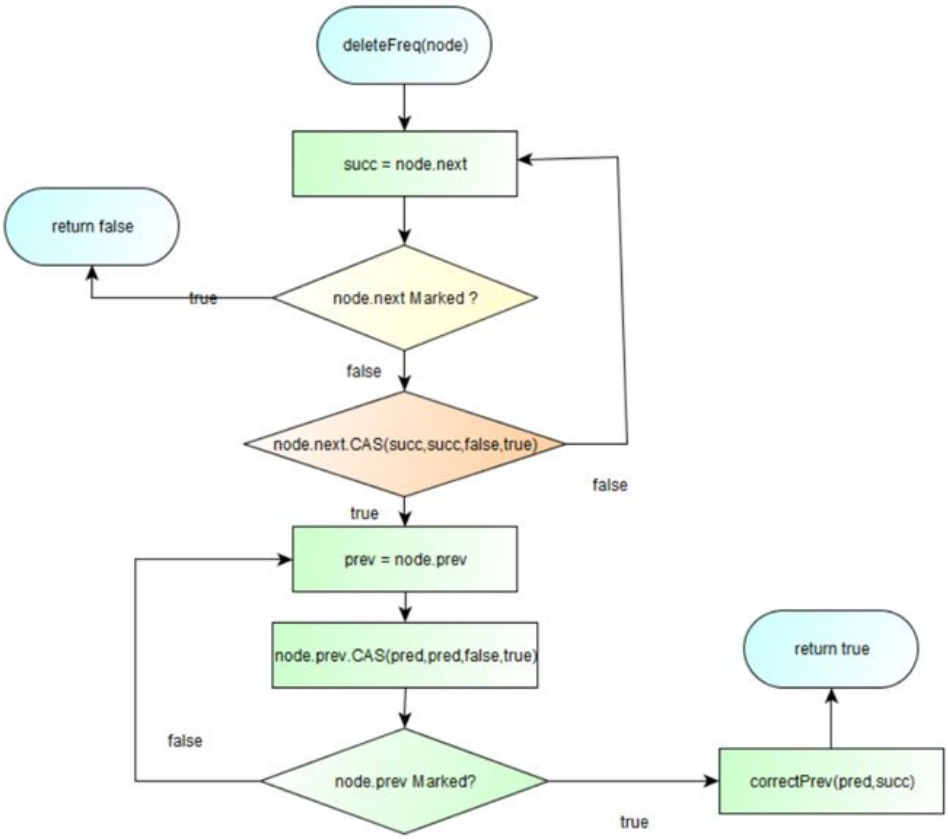
DeleteNode:



The responsibility of this API is to try to delete the given node. This is called when we want to update the node from previous parent to new parent due to new access for the node. This will return false when the node is deleted. It tries to perform logical deletion by marking the next pointer of the node. Once next pointer is marked, it marks prev pointer. This is followed by call to correctPrev to update the prev pointer of succ node which also performs physical deletion of the node. It also checks if it was the last node in the freq list and deletes the freq list accordingly before returning true.

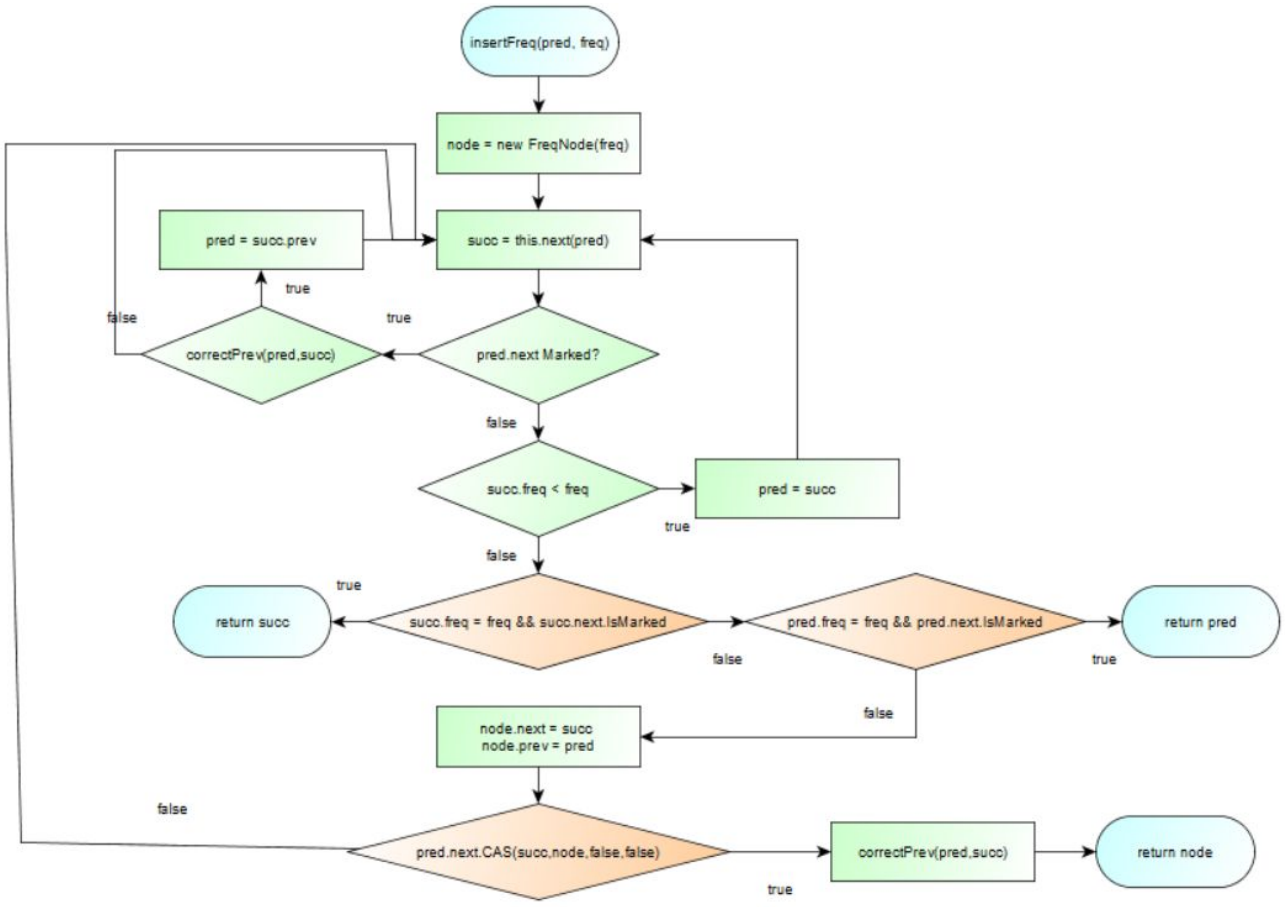
Operations for Freq list:

DeleteFreq:



The responsibility of this API is to try to delete the given frequency node. This is called when we want to delete a frequency node as the frequency node is empty. This will return false if the node is already deleted. It tries to perform logical deletion by marking the next pointer of the node. Once next pointer is marked, it marks prev pointer. This is followed by call to correctPrev to update the prev pointer of succ node which also performs physical deletion of the node.

InsertFreq:



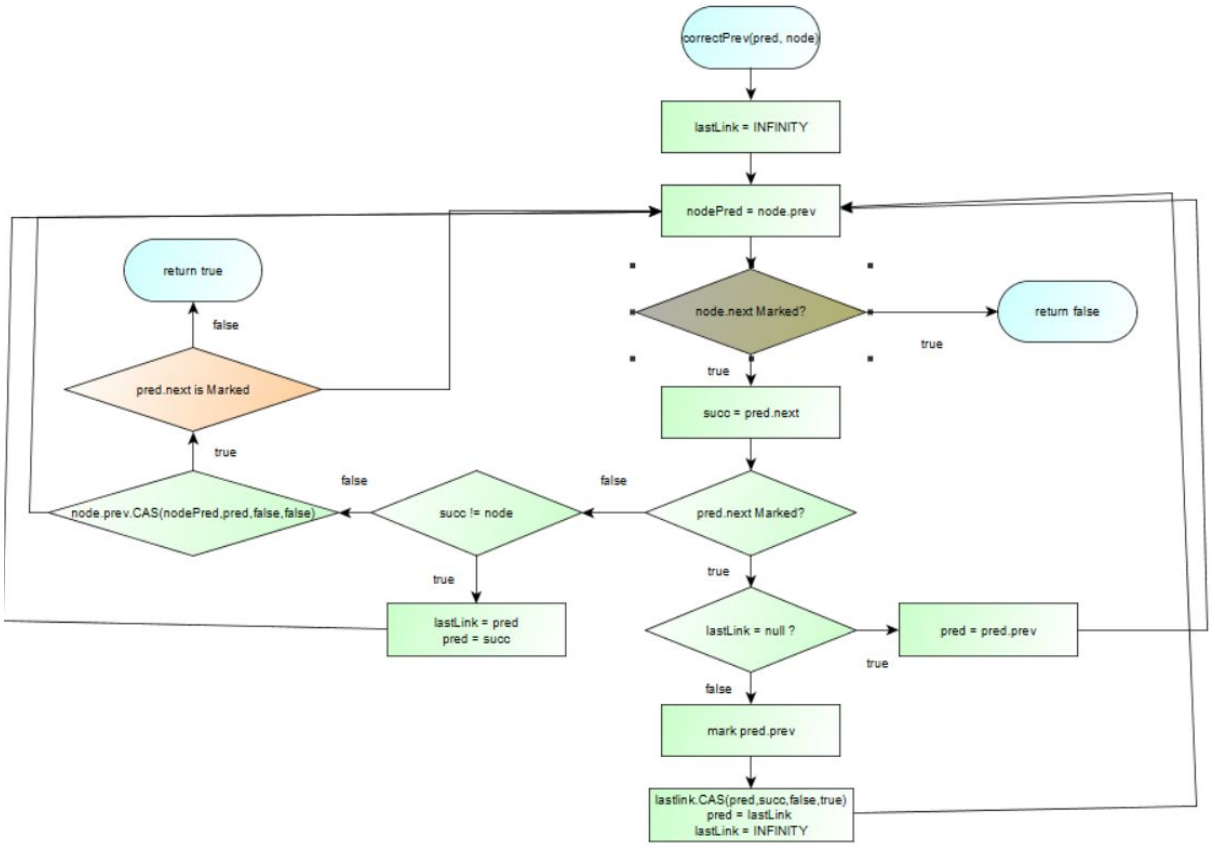
The responsibility of this API is to insert the node if it is not present. If found then it returns the found node. It tries to get an unmarked successor using helper function next. Once it finds the succ, it checks if pred is marked or not. If pred is marked then it tries to get a valid predecessor using correctPrev API. Once we have a valid succ and pred, we check the freq of succ to find the correct spot of insertion. If we find succ to have >= freq value then we break out of loop. If any of the succ and pred are unmarked and is equal to freq then we return the node. If not, then we try CAS over pred.next from succ to node to insert our new node. If this succeeds then we return the node inserted otherwise we retry.

Helper functions:

Next:

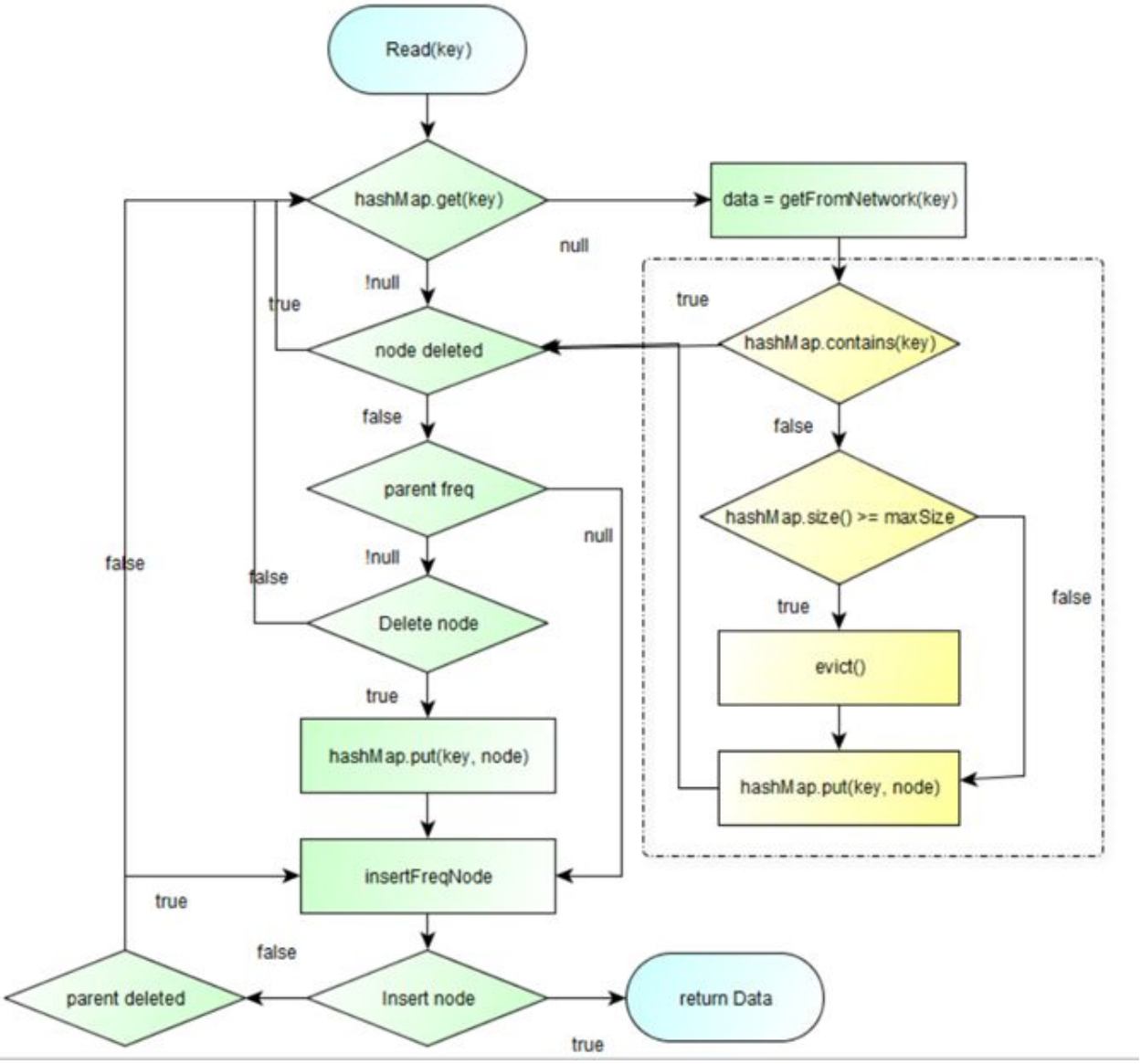
This function tries to find an unmarked successor . If it finds any marked node in the path then it performs a physical deletion of the node after marking the pred pointer.

CorrectPrev:



The correctPrev API is responsible for updating the previous pointer for a given node and also snips away the deleted nodes. Hence, any deleter just marks the node to be deleted and call correctPrev for physical deletion. It will return false if it found the node whose prev pointer is to be fixed to be deleted. It tries to move towards the head direction unless it finds an unmarked node. After that it moves in tail direction and snips away all the logically deleted nodes. After updating the prev pointer for the node, it checks if the pred is logically deleted or not. If it is deleted then it will retry all the operation again.

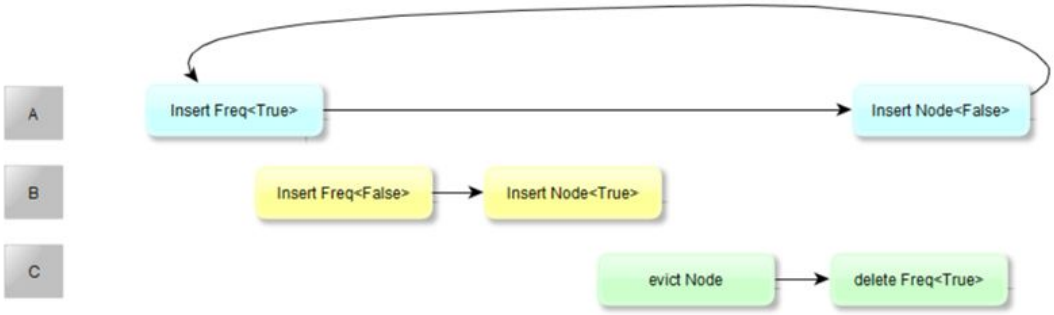
Concurrent Cache Design:



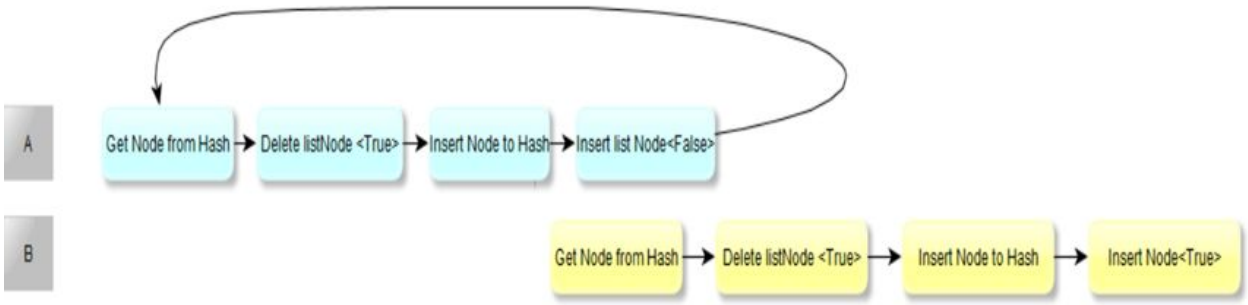
Any thread performing read checks whether the node is present in the hashMap. If it is not found, then we read data from network. After reading the data we try to take lock and check if cache is full. If the cache is full then we perform eviction. After evicting a node we add our node in the hashMap. After inserting the node, we reach the same spot where a thread which found the node in hashMap would be. The next step is to check if the node is deleted. If the node is deleted, then we retry the entire operation. If the node is not deleted, then we read the parent pointer. If parent pointer is null, then it means that this node is being inserted to freq node 1. If it is not null, then we try to delete the node. If deletion fails then we-try everything we have done. If deletion succeeds then we

will try to push a new node with same key and value in the hash Map. This is required as we are not doing node reuse, hence we allocate new node for each new parent. This step is followed by pushing the new parent frequency node. Once that node is pushed, we will try to insert our node in the list. If the insertion fails and the new parent got deleted then we retry by inserting new parent. If my node itself got deleted by some other thread, then I have to retry everything from beginning.

Starvation in our design:

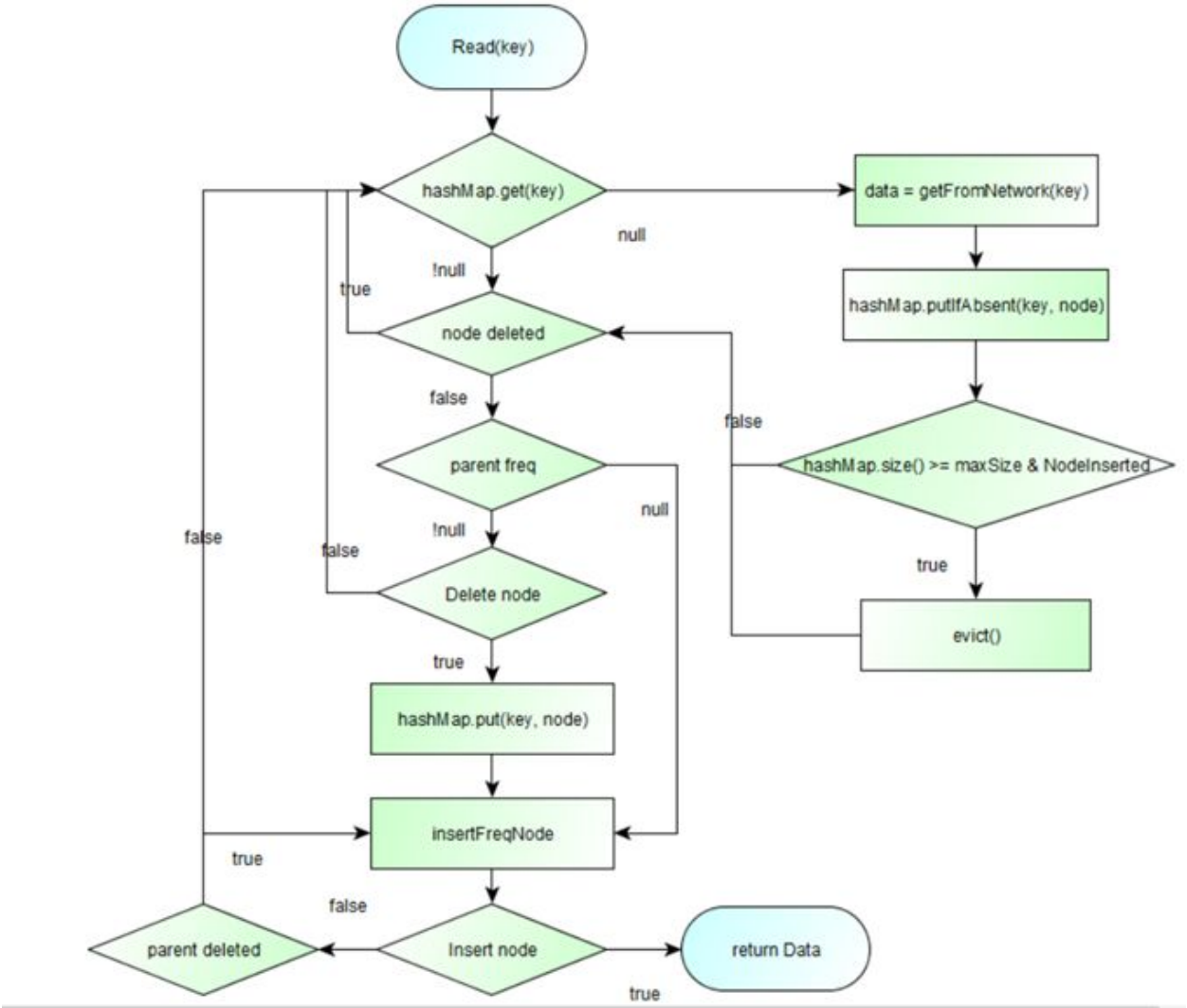


A node can starve if its new parent node is getting deleted everytime due to eviction or a thread which tried to move another node from same parent causing the parent to delete due to empty parents.



A node can starve if its node is getting deleted everytime by some other thread who is trying to modify the same key.

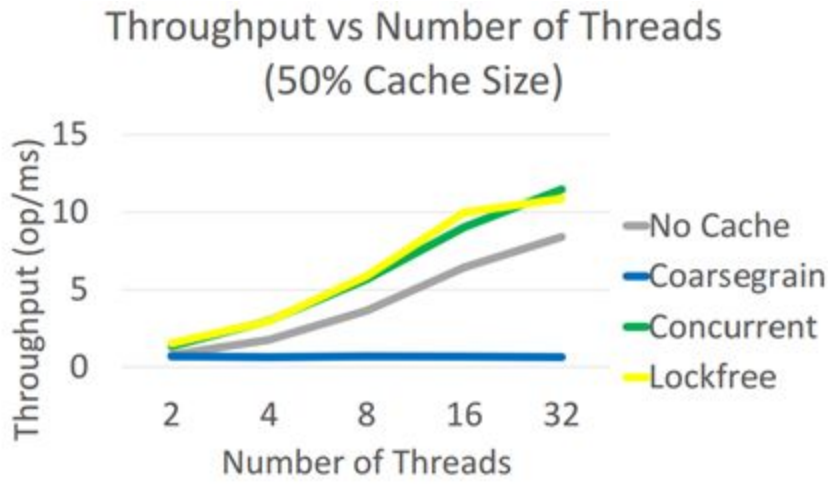
LockFree NonLinearizable LFU:

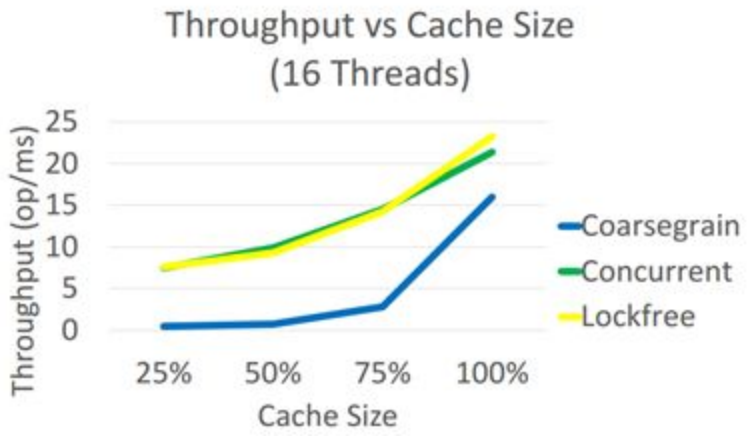


The idea is to improve the performance by removing the synchonized block from earlier implementation. To achieve that we implemented a non-linearizable version which pushes node before performing eviction. This was done to reduce the unnecessary evictions caused by multiple threads coming up with the same key. This implementation requires extra space of thread count in the hashMap to allocate extra node pushes in the hashMap.

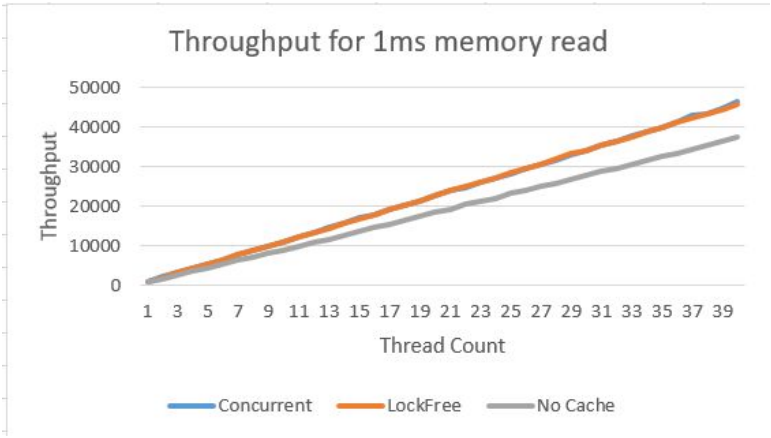
Evaluation

The below chart is with hard memory simulated as read over network. We can observe that coarsegrain remains flat due to sequential bottleneck. “No Cache” is faster than CoarseGrain as multiple threads can read concurrently over the network. Concurrent and LockFree have similar performance and has better performance than “No Cache” due to many Cache hits. Also it is evident that with more cache size we get more cache hits hence higher throughput.

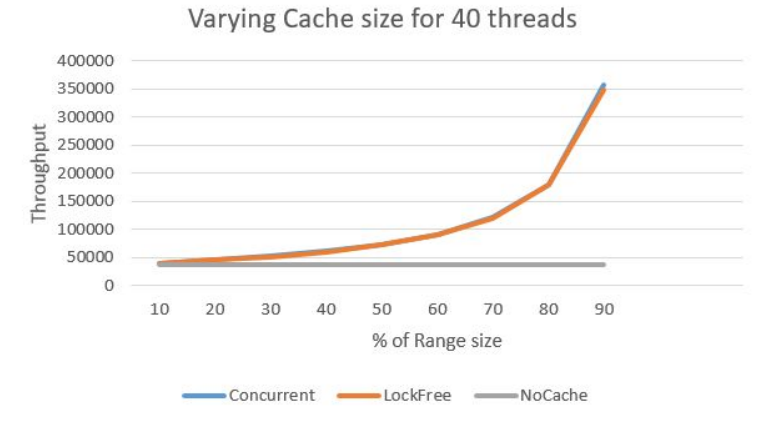




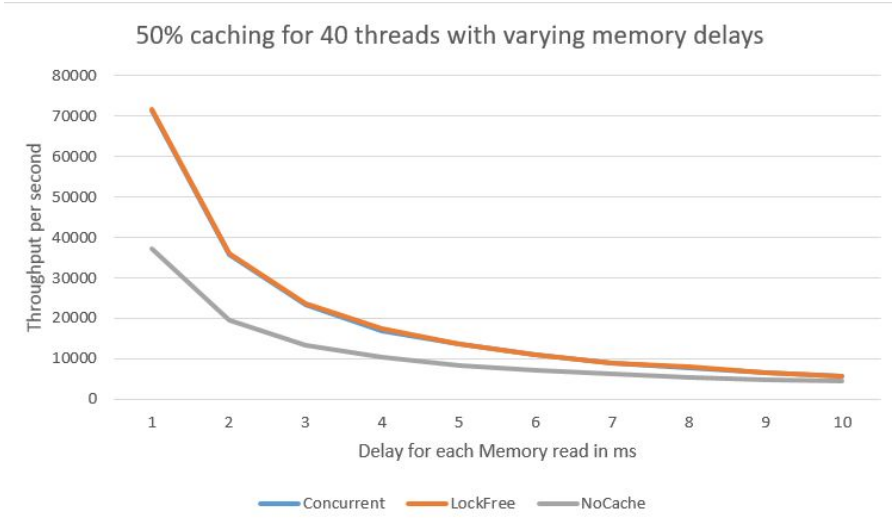
The below chart was taken 1ms delay from memory read resulted from cache miss. As seen the concurrent and lockFree version performs very well compared to no cache. A no cache implementation means direct memory read which has 1ms delay for every read. The lockfree and concurrent has same performance because the bottleneck in thread performance is not the yellow sync block in the figure, instead it is the starvation cycles the thread find themselves in. Hence, we dont get any better performance benefits among concurrent and lockfree design.



The below chart was taken by varying the cache from 10% to 90% of the size of the memory where cache miss causes 1ms read penalty. As it can be seen that the lockFree and Concurrent implementations scale very well with increasing cache size. The increase in throughput is due to less number of cache misses with bigger cache size.



The below chart is taken after varying the memory read delay from 1ms to 10ms. For higher delay we find bad throughput/second as most of the time is spent while reading the data during cache miss. For lower delay, we get good throughput when compared with no cache scenario as cache misses have lower time penalty.



Conclusion

The classic tradeoff in computing is time for space. However, in this course there is a third dimension: accuracy. We have experimented with all three. The coarsegrain throughput is limited due to the sequential bottleneck. The no cache solution is the baseline for our implementation. We observe higher throughput for small delay memory reads as the time spent in cache misses are less. We did not get any performance difference in the linearizable and nonlinearizable solutions as the major time is taken due to starvation cycles the thread find themselves in and hence the synchronization block has minimal performance impact. However, we can certainly simulate some pathogenic data set for which lockfree solution will outperform concurrent solution. But with a random dataSet we expect same performance for both of them.

Ultimately, the winner is the concurrent solution. It has the best overall performance with perfect accuracy. Our cache should be utilized to save resources and time.

References

[1] http://www.pctools.com/security-news/what-is-a-browser-cache/

[2] http://dhruvbird.com/lfu.pdf

[3] https://www.cs.princeton.edu/research/techreps/TR-622-00

[4] https://pdfs.semanticscholar.org/e87b/0601320b7a7ee0605f2a8748c45b94930793.pdf