AOS Assignment 5: Key value store Report

- Murali Kammili

In my assignment, I have implemented both Least Recently Used (LRU) and Adaptive Replacement Cache (ARC) management algorithms. My key-value store cache is of variable size and can be tuned based on the given input. The cache size is based on the parameters I supply in the program and memory for the cache is allocated using xmalloc() implemented in my last assignment. We can observe the performance of my implementation using cache monitoring operations. By increasing the cache size, I could achieve zero '0' errors for both sets and gets. However, it may not be practically feasible to have a large cache size such that no single request fails for all types of input.

In the kv_set(), i am storing the key-value pair in the cache as soon as I get the request. First, I check if the cache already contains a value for the same key. Then we just update the new value of the respective key. If the key is already not present in cache, I try to place the new key-value pair in the cache. Eventually our cache gets filled and there will be no more space to accommodate the set request. In such scenario, we find a victim key-value pair that has to be evicted to save the new key-value pair. This eviction policy differs from the algorithm implemented. In my LRU program, I am evicting the least recently used page among the keys of same hash index. For ARC, I am maintaining an extra cache of the same size as my main cache. This extra cache is split into two equal lists (ghost caches) which are represented as g1 and g2. These save the history of evicted pages from the main cache which is logically viewed as combination of T1 and T2. This abstraction is represented as an integer value {1 for T1, 2 for T2} in my program using a variable 'T' in 'xcache_k' structure. I have declared a global variable 'P', which allows us to maintain track of changes in T1 and T2. If any key is found to be accessed more than once, we change it from T1 to T2 and P is changed accordingly.

In the kv_get(), I am first obtaining the hash index to narrow the key search in the cache. If I have found the key, then I return the value associated with that key. If it is a miss, I am incrementing the get error count and further continue to search in the ghost caches. If we could not find the key in the main cache, then I am scanning the ghost cache g2. If I found the key in g2 then, I am evicting from T1 to accommodate one more extra key-value pair for T2. Variable P maintains how many caches are reserved for T2 at that moment. Similarly if I found the key in g1 then, I am evicting from T2 to accommodate one more extra key-value pair for T1. This eviction is again implemented using LRU.

In the kv_delete(), I am searching for the input key in cache and deleting the associated key-value pair by setting the values to null character. If I could not find the key, I am returning boolean value False.

In the kv_reset(), I am freeing all the allocated memory created using xmalloc() and resetting the values. In the end I am calling xheap_snapshot() which is implemented in the previous assignment to verify if all the allocated memory is freed up to be utilized again.

In kv_init( ), I am calling xmalloc_init() first to initialize the buffer pool. I am allocating required cache memory by calling xmalloc(). Note that ARC requires double the cache size required by LRU.

In get_cache_info(), I am returning the required variable by pre-computing the values in the related functions inside the program. These capture the given cache monitoring operations and return the result.

The most_popular_keys(), returns the ‘k’ most frequently used keys after eliminating the duplicates.

I tested my both programs on the given test cases coving few scenarios and I found my ARC implementation to be a better performer than LRU. In my program the maximum size of my key is 64 bytes and maximum size of the value will be 1 KB as mentioned in the assignment. If the size is beyond that I have written condition to ignore it rather than truncation.

Sample :

For cache size - 25*91 and on input file xlarge

Using LRU :

total_hits 82
total_accesses 100
total_set_success 2048
cache_size 2226176
num_keys 2034
total_evictions 14

Set errors: 0, Get errors: 18

Using ARC :

total_hits 100
total_accesses 100
total_set_success 2048
cache_size 2211854
num_keys 2034
total_evictions 14

Set errors: 0, Get errors: 0

For cache size - 25*92 and on input file xlarge

Using LRU and ARC:

total_hits 100
total_accesses 100
total_set_success 2048
cache_size 2226176
num_keys 2039
total_evictions 9

Set errors: 0, Get errors: 0