Department of Computer Science and Engineering

Course Code: CSE360

Course Title: Computer Architecture

Section: 01

Semester: Summer 24

Submitted to:

Dr. Md. Nawab Yousuf Ali

Professor

Department of Computer Science and Engineering

East West University

Submitted by:

Name: Umme Habiba

ID: 2021-3-60-047

Name: Sakib Hasan

ID: 2022-1-60-098

Name: Lamia Binte Zaman

ID: 2021-3-60-066

Submission date :24 September, 2024

Objective

1. **Maximize Cache Hit Rate**:
   - Ensure that frequently accessed or valuable data remains in the cache, leading to more cache hits and reducing the need to fetch data from slower memory.
2. **Minimize Cache Pollution**:
   - Prevent cache space from being consumed by rarely used or irrelevant data, which can displace more important data and lower the cache's overall efficiency.
3. **Reduce Cache Thrashing**:
   - Avoid frequent eviction and reloading of the same data, which happens when the cache is overwhelmed by data churn. The goal is to maintain a stable set of frequently used data in the cache.
4. **Improve System Performance**:
   - By reducing cache misses, the policy should improve the overall speed and efficiency of the system, lowering the latency involved in accessing memory.
5. **Adapt to Workload Dynamics**:
   - The policy should be flexible and adaptive to changing access patterns in workloads, dynamically adjusting to prevent performance degradation due to changes in data usage.
6. **Efficient Cache Resource Usage**:
   - Make the best use of limited cache resources by carefully selecting which data to cache, evict, and retain, ensuring the cache serves its purpose without waste.

What is Cache Pollution and Thrashing?

Cache Pollution:

Cache pollution occurs when irrelevant or low-utility data is loaded into the cache, displacing more frequently accessed or useful data. This reduces the overall efficiency of the cache, as important data may be evicted prematurely in favor of less useful data.

Cache Thrashing:

Cache thrashing happens when the system continually loads and evicts data from the cache without effectively using it. This typically occurs when the working set of data (the active data the system is using) exceeds the cache size, causing constant data eviction and reloading.

Benefits of reducing cache pollution and thrashing

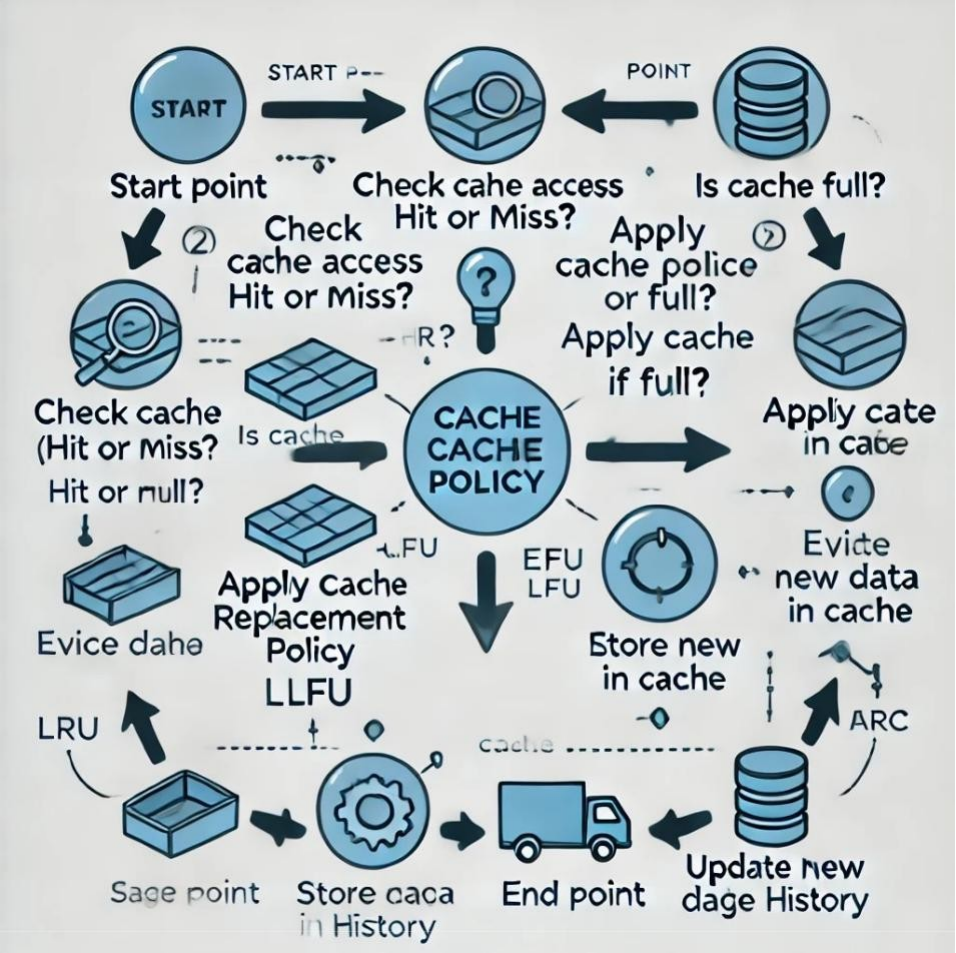
- Reducing cache pollution ensures that valuable data is retained in the cache, leading to more cache hits. Cache hits provide fast access to frequently used data without needing to fetch it from slower main memory.

- Reducing thrashing prevents the system from constantly loading and evicting data, which otherwise results in excessive cache misses and high memory access times.

- Cache is a limited and expensive resource. Reducing pollution ensures that the cache stores data that will be reused, rather than irrelevant data that won't provide any benefit.

- Frequent memory access caused by cache misses increases power consumption, especially in mobile devices. Minimizing pollution and thrashing reduces the need to repeatedly access main memory.

Code Flow chart



Code Concept:

- LFU (Least Frequently Used): The cache evicts the least frequently used items. If two items have the same frequency, the least recently used item is evicted.
- LRU (Least Recently Used): Items accessed most recently are moved to the front of the cache list, and those accessed least recently are moved toward the back.
- Combination of LFU + LRU: When frequencies are tied, the cache behaves like an LRU cache, evicting the least recently used item.

Future Improvements:

•Advanced Eviction Policies: Consider an LFU (Least Frequently Used) policy alongside LRU for better control over hot and cold data.

•Segmentation: Divide the cache into segments for hot and cold data, where frequently accessed data is stored in the "hot" segment.

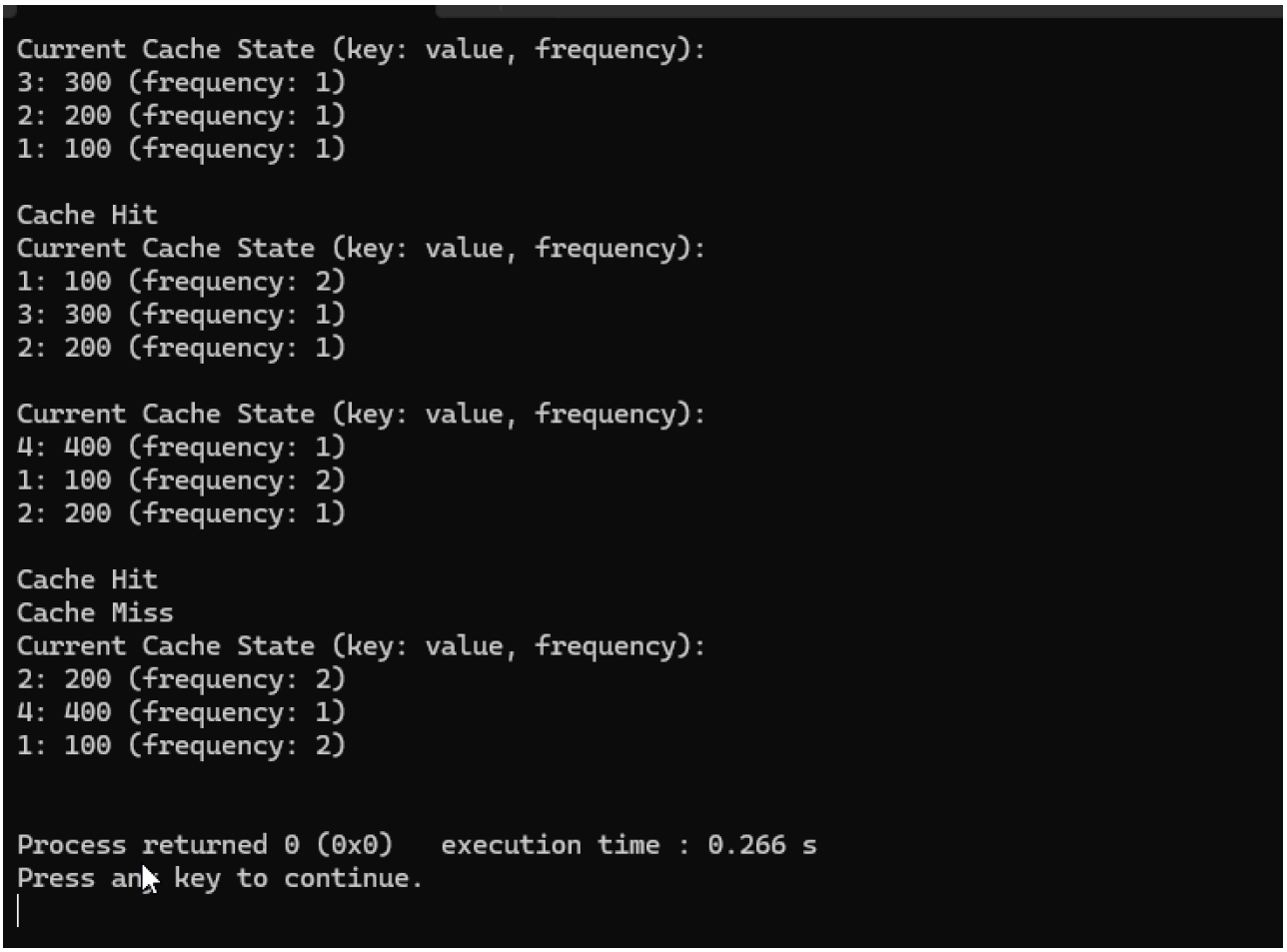
•TTL (Time-to-Live): Introduce expiration times for certain entries to remove stale data from the cache.

Code:

```
#include <iostream>
#include <unordered_map>
#include <list>

class Cache {
    struct CacheEntry {
        int key;
        int value;
        int frequency;
    };

    int capacity;
    std::list<CacheEntry> cacheList; // stores cache entries in order of usage
    std::unordered_map<int, std::list<CacheEntry>::iterator> cacheMap; // key to cache entry

    void touch(std::list<CacheEntry>::iterator it) {
        CacheEntry entry = *it;
        cacheList.erase(it);              // remove entry from current position
        entry.frequency++;                // increase frequency of access
        cacheList.push_front(entry);      // move entry to the front
        cacheMap[entry.key] = cacheList.begin();  // update map to new position
    }

public:
    Cache(int cap) : capacity(cap) {}

    int get(int key) {
        if (cacheMap.find(key) == cacheMap.end()) {
            std::cout << "Cache Miss\n";
            return -1;
        } else {
            std::cout << "Cache Hit\n";
            touch(cacheMap[key]); // update LRU position and frequency
            return cacheMap[key]->value;
        }
    }

    void put(int key, int value) {
        if (cacheMap.find(key) != cacheMap.end()) {
            // If key exists, update value and move to front
            cacheMap[key]->value = value;
            touch(cacheMap[key]);
        } else {
            // If cache is full, evict the least frequently used or least recently used entry
            if (cacheList.size() == capacity) {
                // find the least frequent item in the list (if frequencies are tied, LRU is used)
                auto minIt = cacheList.end();
```

```
            int minFreq = INT_MAX;
            for (auto it = cacheList.begin(); it != cacheList.end(); ++it) {
                if (it->frequency < minFreq) {
                    minFreq = it->frequency;
                    minIt = it;
                }
            }
            if (minIt != cacheList.end()) {
                cacheMap.erase(minIt->key); // remove entry from map
                cacheList.erase(minIt);     // remove entry from list
            }
        }

        // Add new cache entry
        cacheList.push_front({key, value, 1});
        cacheMap[key] = cacheList.begin();
    }
}

void displayCache() {
    std::cout << "Current Cache State (key: value, frequency):\n";
    for (const auto& entry : cacheList) {
        std::cout << entry.key << ": " << entry.value << " (frequency: " << entry.frequency << ")\
n";
    }
    std::cout << std::endl;
}
};

int main() {
    Cache cache(3); // Cache with capacity 3

    cache.put(1, 100);
    cache.put(2, 200);
    cache.put(3, 300);
    cache.displayCache();

    cache.get(1); // Access key 1
    cache.displayCache();

    cache.put(4, 400); // Insert key 4, this should evict key 2 (least frequently accessed)
    cache.displayCache();
    cache.get(2); // Access key 2 (cache miss since it was evicted)
    cache.get(3); // Access key 3 (cache hit)
    cache.displayCache();

    return 0;
}
```

Output:



Conclusion:

The code implements an LFU (Least Frequently Used) cache with LRU (Least Recently Used) eviction for tied frequencies. It efficiently handles cache operations using a combination of a linked list and a hashmap. The `get()` and `put()` methods ensure quick access and insertion, while evictions prioritize removing the least frequently accessed or least recently used items when the cache reaches its capacity.