
# COA Project

## Cache Replacement Policies

Team Details:

- CS22B1020: Deshna Thunga
- CS22B1081: Neha Kantheti
- CS22B1082: Nimisha Thallapally

# Contents

# 1 Cache

Caches store frequently used data for faster access. When full, they need to evict something. Least Frequently Used (LFU) removes data accessed the least, while Most Frequently Used (MFU) (not common) removes the most accessed data first. LFU avoids a specific issue but adds tracking overhead. The best policy depends on your workload's access patterns.

# 2 Cache Replacement Policies

In computing, caches are temporary storage areas that hold frequently accessed data or instructions closer to the processor for faster retrieval. When a cache reaches its capacity and new data needs to be stored, a cache replacement policy determines which existing data to evict to make room. This decision significantly impacts the cache's effectiveness.The replacement policy has to be chosen in such a way that the cache misses are reduced.

Some of the commonly used Cache Replacement Policies are FIFO, LRU, MRU and Random Replacement.

We have come up with two new Cache Replacement Policies which are Least Frequently Used(LRU) and Most Frequently Used(MRU).

## 2.1 Least Frequently Used (LFU) And Most Frequently Used(MFU)

Caches store frequently used data for faster access. When full, they need to evict something. Least Frequently Used (LFU) removes data accessed the least, while Most Frequently Used (MFU) (not common) removes the most accessed data first. LFU avoids a specific issue but adds tracking overhead. The best policy depends on your workload's access patterns.

## 2.2 Traditional Cache Replacement Policies:

In the traditional LRU policy, the victim block is chosen from the LRU position of the priority chain. The newly inserted block is placed in the MRU position and on a cache hit the cache block is promoted to the MRU position. As an example in the fig., on insertion of incoming block 'i' in to the priority chain, the block at the LRU position 'g' is removed from it and the insertion of block 'i' occurs in the MRU position.

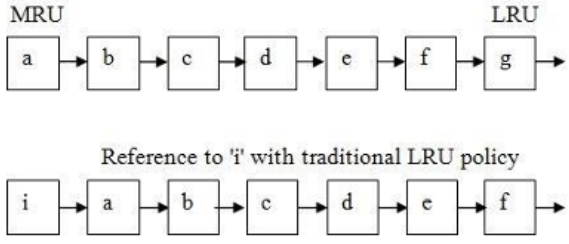


Fig. 1. Traditional LRU policy

Figure 1: Traditional LRU Policy

The re-reference count refers to the number of times a cache block has been referenced since it was last brought into the cache. It helps track the recent usage pattern of cache blocks. If the re-reference count exceeds a threshold, the cache block is promoted to the MRU position; otherwise, it remains at the LRU position. The threshold mentioned in this context is a predetermined value based on set duelling, used to determine whether a cache block should be promoted to the MRU (Most Recently Used) position or remain at the LRU (Least Recently Used) position. In set dueling some of the cache sets will be dedicated to follow LRU policy and some other sets will follow BIP policy and rest of the sets are known as follower sets. The follower sets choose that policy which incurs fewer misses in the above sets.

### 2.3 A Cache Replacement Policy Based on Re-reference Count

In this technique, the promotion policy similar in case of LIP is modified in such a way that if the re-reference count is greater than a threshold value, the cache block is promoted on to the MRU position else it is promoted to LRU position as shown in fig. Results show that under this modification L2 cache miss rate can be reduced.

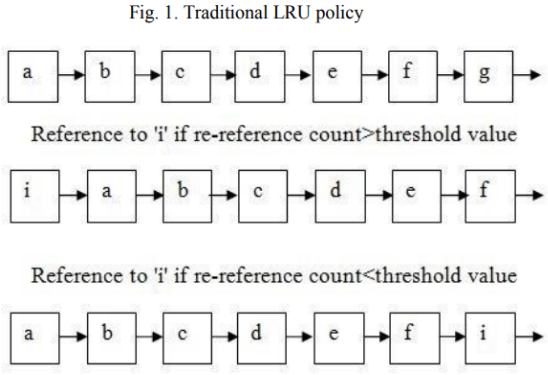


Figure 2: Traditional LRU Policy

# Algorithm

Input: SPEC CPU 2006 Benchmarks
Output: L2 Cache Miss Rate

1: For blocks undergone cache hit do
2: If blocks re-reference count > (14) then
3: Move to MRU position
4: Else
5: Move to LRU position
6: End if
7: End for
8: For blocks need to be inserted to cache do
9: Move to LRU position
10: End for
11: For blocks to be evicted from the cache do
12: Select from LRU position
13: End for

#### 2.3.1 NOTE

For the memory intensive workload, when the re-reference count is set to 4, there is a significant reduction in L2 cache miss rate. When the re-reference count is set to 14, there is a 60% improvement in L2 cache miss rate when compared to the traditional LRU policy. We've devised a strategy based on the research paper mentioned above, utilizing Least Frequently Used (LFU) and Most Frequently Used (MFU) cache replacement policies.

# 3 Least Frequently Used (LFU)

## 3.1 Concept

The LFU policy prioritizes removing data items in the cache that have been accessed the least number of times overall. The assumption is that data accessed less frequently is less likely to be needed again soon.

## 3.2 Implementation

LFU requires tracking access frequencies for each data item. This can be achieved using counters or timestamps. When a cache eviction is necessary, the item with the lowest access count is removed.

#### 3.2.1 Advantages

- Can be effective for workloads with a mix of access patterns, particularly for data that becomes less relevant over time (e.g., web browsing history).

#### 3.2.2 Disadvantages

- Tracking access frequencies can add overhead, especially for large caches or frequent updates.
- May not be ideal for scenarios where recently accessed data might still be needed even if accessed infrequently overall.

# 4 Most Frequently Used (LFU)

## 4.1 Concept

The MFU policy prioritizes removing data items in the cache that have been accessed the most number of times overall. The underlying idea is that data accessed more often is less likely to be needed again soon.

## 4.2 Implementation

Similar to LFU, MFU also requires tracking access frequencies for each data item. This can be achieved using counters or timestamps. When a cache eviction is necessary, the item with the highest access count is removed.

#### 4.2.1 Advantages

- Aims to keep the less frequently accessed data in the cache, assuming it's more likely to be needed again soon.

#### 4.2.2 Disadvantages

- MFU has a problem when a recently accessed item is evicted just before it's needed again, leading to a cache miss.
- Tracking access frequencies incurs overhead.

# 5 Code Snippets

## 5.1 LFU Implementation

```
#include <iostream>
#include<vector>
#include<unordered_map>
using namespace std;

void swap(pair<int, int>& a, pair<int, int>& b) {
    pair<int, int> temp = a;
    a = b;
    b = temp;
}

int parent(int i) {
    return (i - 1) / 2;
}
int left(int i) {
    return 2 * i + 1;
}
int right(int i) {
    return 2 * i + 2;
}

void heapify(vector<pair<int, int>>& v, unordered_map<int, int>& m, int i, int n) {
    int l = left(i), r = right(i), minim;
    if (l < n)
    minim = ((v[i].second < v[l].second) ? i : l);
    else
    minim = i;
    if (r < n)
    minim = ((v[minim].second < v[r].second) ? minim : r);
    if (minim != i) {
        m[v[minim].first] = i;
        m[v[i].first] = minim;
        swap(v[minim], v[i]);
        heapify(v, m, minim, n);
    }
}

void increment(vector<pair<int, int>>& v, unordered_map<int, int>& m, int i, int n) {
    ++v[i].second;
    heapify(v, m, i, n);
}

void insert(vector<pair<int, int>>& v, unordered_map<int, int>& m, int value, int& n) {
    if (n == v.size()) {
        m.erase(v[0].first);
        cout << "Cache block " << v[0].first << " removed.\n";
        v[0] = v[--n];
        heapify(v, m, 0, n);
    }
    v[n++] = make_pair(value, 1);
    m.insert(make_pair(value, n - 1));
    int i = n - 1;
    while (i && v[parent(i)].second > v[i].second) {
        m[v[i].first] = parent(i);
        m[v[parent(i)].first] = i;
        swap(v[i], v[parent(i)]);
        i = parent(i);
    }
    cout << "Cache block " << value << " inserted.\n";
}

void refer(vector<pair<int, int>>& cache, unordered_map<int, int>& indices, int value, int& cache_size) {
    if (indices.find(value) == indices.end()) {
        cout << "Cache Miss\n";
        insert(cache, indices, value, cache_size);
    } else {
        cout << "Cache Hit\n";
        increment(cache, indices, indices[value], cache_size);
    }
}

void printCache(const vector<pair<int, int>>& cache) {
    cout << "Cache contents:";
    for (const auto& block : cache) {
        cout << " " << block.first << "(" << block.second << ")";
```

```
        }
        cout << endl;
    }

    int main() {
        int cache_max_size = 4, cache_size = 0;
        vector<pair<int, int>> cache(cache_max_size);
        unordered_map<int, int> indices;

        cout << "Inserting 1 \n";
        refer(cache, indices, 1, cache_size);
        printCache(cache);

        cout << "Inserting 2 \n";
        refer(cache, indices, 2, cache_size);
        printCache(cache);

        cout << "Inserting 1 \n";
        refer(cache, indices, 1, cache_size);
        printCache(cache);

        cout << "Inserting 3 \n";
        refer(cache, indices, 3, cache_size);
        printCache(cache);

        cout << "Inserting 2 \n";
        refer(cache, indices, 2, cache_size);
        printCache(cache);

        cout << "Inserting 4 \n";
        refer(cache, indices, 4, cache_size);
        printCache(cache);

        cout << "Inserting 5 \n";
        refer(cache, indices, 5, cache_size);
        printCache(cache);

        return 0;
    }
```

### Logic Behind the LFU Implementation

The C++ code above implements a Least Frequently Used (LFU) cache management strategy using a min-heap. This setup tracks the access frequency of each cache block, storing them in a vector and enabling quick lookups with an unordered map. Whenever data is accessed, the code checks for its presence in the cache. If not found, a "cache miss" occurs, leading to the insertion of this new block and potential eviction of the least used item if the cache is full. For cache hits, the access count of that block is incremented, and the heap structure is adjusted to keep the least accessed items at the front for easy eviction, i.e. , the root node has the least frequently used data item. This technique optimizes cache use by keeping frequently accessed data readily available, reducing unnecessary data fetching from slower main memory.

## 5.2 MFU Implementation

```
    #include <iostream>
    #include<vector>
    #include<unordered_map>
    using namespace std;

    void swap(pair<int, int>& a, pair<int, int>& b) {
        pair<int, int> temp = a;
        a = b;
        b = temp;
    }

    int parent(int i) {
        return (i - 1) / 2;
    }

    int left(int i) {
        return 2 * i + 1;
    }

    int right(int i) {
```

```
21          return 2 * i + 2;
22      }
23
24      void heapify(vector<pair<int, int>>& v, unordered_map<int, int>& m, int i, int n) {
25          int l = left(i), r = right(i), maxim;
26          if (l < n)
27          maxim = ((v[i].second > v[l].second) ? i : l);
28          else
29          maxim = i;
30          if (r < n)
31          maxim = ((v[maxim].second > v[r].second) ? maxim : r);
32          if (maxim != i) {
33              m[v[maxim].first] = i;
34              m[v[i].first] = maxim;
35              swap(v[maxim], v[i]);
36              heapify(v, m, maxim, n);
37          }
38      }
39
40      void increment(vector<pair<int, int>>& v, unordered_map<int, int>& m, int i, int n) {
41          ++v[i].second;
42          heapify(v, m, i, n);
43      }
44
45      void insert(vector<pair<int, int>>& v, unordered_map<int, int>& m, int value, int& n) {
46          if (n == v.size()) {
47              m.erase(v[0].first);
48              cout << "Cache block " << v[0].first << " removed.\n";
49              v[0] = v[--n];
50              heapify(v, m, 0, n);
51          }
52          v[n++] = make_pair(value, 1);
53          m.insert(make_pair(value, n - 1));
54          int i = n - 1;
55          while (i && v[parent(i)].second < v[i].second) {
56              m[v[i].first] = parent(i);
57              m[v[parent(i)].first] = i;
58              swap(v[i], v[parent(i)]);
59              i = parent(i);
60          }
61          cout << "Cache block " << value << " inserted.\n";
62      }
63
64      void refer(vector<pair<int, int>>& cache, unordered_map<int, int>& indices, int value, int& cache_size) {
65          if (indices.find(value) == indices.end()) {
66              cout << "Cache Miss\n";
67              insert(cache, indices, value, cache_size);
68          } else {
69              cout << "Cache Hit\n";
70              increment(cache, indices, indices[value], cache_size);
71          }
72      }
73
74      void printCache(const vector<pair<int, int>>& cache) {
75          cout << "Cache contents:";
76          for (const auto& block : cache) {
77              cout << " " << block.first << "(" << block.second << ")";
78          }
79          cout << endl;
80      }
81
82      int main() {
83          int cache_max_size = 4, cache_size = 0;
84          vector<pair<int, int>> cache(cache_max_size);
85          unordered_map<int, int> indices;
86
87          cout << "Inserting 1 \n";
88          refer(cache, indices, 1, cache_size);
89          printCache(cache);
90
91          cout << "Inserting 2 \n";
92          refer(cache, indices, 2, cache_size);
93          printCache(cache);
94
95          cout << "Inserting 1 \n";
96          refer(cache, indices, 1, cache_size);
97          printCache(cache);
98
99          cout << "Inserting 3 \n";
100         refer(cache, indices, 3, cache_size);
101         printCache(cache);
```

```
102
103         cout << "Inserting 2 \n";
104         refer(cache, indices, 2, cache_size);
105         printCache(cache);
106
107         cout << "Inserting 4 \n";
108         refer(cache, indices, 4, cache_size);
109         printCache(cache);
110
111         cout << "Inserting 5 \n";
112         refer(cache, indices, 5, cache_size);
113         printCache(cache);
114
115         return 0;
116     }
```

### Logic Behind the MFU Implementation

The C++ code sets up a cache system using a max-heap to implement the Most Frequently Used (MFU) cache replacement strategy. The program uses a vector to represent the heap, where each element is a pair of a cache block's identifier and its access frequency. An unordered map tracks each block's position in the heap for quick access. Whenever a block is accessed (via the refer function), the program checks if it's already in the cache. If it's not, this triggers a "cache miss" and the block is added to the heap. If the cache is full, the most frequently used item, which is at the top of the heap, is removed to make space for the new one. If the block is found in the cache ("cache hit"), its access frequency is incremented and the heap is adjusted to reflect the change. This method optimizes cache usage by ensuring that the most frequently accessed data is evicted first in the hope that it will not be needed again, while less frequently accessed data is kept.

# 6 Output

## 6.1 LRU OUTPUT

```
PS E:\COA> g++ lfu.cpp -o lfu
PS E:\COA> ./lfu.exe
Inserting 1
Cache Miss
Cache block 1 inserted.
Cache contents: 1(1) 0(0) 0(0) 0(0)
Inserting 2
Cache Miss
Cache block 2 inserted.
Cache contents: 1(1) 2(1) 0(0) 0(0)
Inserting 1
Cache Hit
Cache contents: 2(1) 1(2) 0(0) 0(0)
Inserting 3
Cache Miss
Cache block 3 inserted.
Cache contents: 2(1) 1(2) 3(1) 0(0)
Inserting 2
Cache Hit
Cache contents: 3(1) 1(2) 2(2) 0(0)
Inserting 4
Cache Miss
Cache block 4 inserted.
Cache contents: 3(1) 4(1) 2(2) 1(2)
Inserting 5
Cache Miss
Cache block 3 removed.
Cache block 5 inserted.
Cache contents: 4(1) 5(1) 2(2) 1(2)
PS E:\COA>
```

Figure 3: Output of C++ LFU Code

### 6.2 MRU OUTPUT

```
PS E:\COA> g++ mfu.cpp -o mfu
PS E:\COA> ./mfu.exe
Inserting 1
Cache Miss
Cache block 1 inserted.
Cache contents: 1(1) 0(0) 0(0) 0(0)
Inserting 2
Cache Miss
Cache block 2 inserted.
Cache contents: 1(1) 2(1) 0(0) 0(0)
Inserting 1
Cache Hit
Cache contents: 1(2) 2(1) 0(0) 0(0)
Inserting 3
Cache Miss
Cache block 3 inserted.
Cache contents: 1(2) 2(1) 3(1) 0(0)
Inserting 2
Cache Hit
Cache contents: 1(2) 2(2) 3(1) 0(0)
Inserting 4
Cache Miss
Cache block 4 inserted.
Cache contents: 1(2) 2(2) 3(1) 4(1)
Inserting 5
Cache Miss
Cache block 1 removed.
Cache block 5 inserted.
Cache contents: 2(2) 4(1) 3(1) 5(1)
PS E:\COA>
```

Figure 4: Output of C++ MFU Code

## 7 Conclusion

In conclusion, when choosing between MFU (Most Frequently Used) and LFU (Least Frequently Used) cache replacement strategies, it's essential to consider your workload. MFU targets the most accessed data but can be complex to implement, especially if recent access is a better predictor of future use. LFU, on the other hand, focuses on removing less frequently accessed data. However, both strategies come with a disadvantage—they add overhead to the system. Therefore, exploring well-established alternatives that strike a balance between recency and access frequency is crucial for optimizing your cache's performance while managing overhead.

## 8 References

**Referenced Paper:** A Cache Replacement Policy Based on Re-reference Count Sreya Sreedharan Department of Computer Science and Engineering Rajagiri School of Engineering and Technology Kochi, India
**Referenced Site:** IEEE Xplore, GeeksforGeeks
**Links:** LFU and MFU Code Reference