BTech Project

Mentor : Dr Shweta Jain

Group Members - Raj Gupta (2020CSB1116)
Ayushi Patel (2020CSB1080)

TOPIC

Fair Allocation of Cache in In-Memory Systems

IDEA

In cloud computing environments, multiple users and applications often share the same pool of resources, including memory. When it comes to memory caching, it is important to allocate the cache space fairly among the users and applications, so that each user can get their fair share of the performance benefits.

Our aim is to deduce an algorithm that gives fair allocation of cache along with incorporating the idea of public and private cache that has been introduced in the field of Computer Architecture.

We want the algorithm to be proportion fair and efficient and also provide strategy proofness.

Desirable properties for a good Algorithm

1. **Isolation guarantee:** by sharing the caches, each user should get no fewer files in memory than it would have had with an isolated and evenly partitioned cache.

2. **Strategy-proofness:** a user cannot have more files in memory at the expense of another by lying about its demand of caching a file.

3. **Pareto efficiency:** it is not possible to cache more files for a user without evicting files of another.

Opportunistic Fair Sharing of Cache (OpUS)

It solves the problem of free riding often encountered in shared systems.

- N
- M
- a_j
- $p_{i,j}$

$$U_i(\mathbf{a}) = \sum_{j=1}^{M} a_j p_{i,j}$$

Virtual Utility :

$$V_i(\mathbf{a}) = \log U_i(\mathbf{a})$$

Stage 1 : strives to share caches for high efficiency without suffering harmful manipulations using Vickrey-Clarke-Groves (VCG) mechanism.

Stage 2 : decide whether to go with allocation found in first step or not.

Algorithm 1 OpuS: **Op**portu**n**istic **S**haring for high efficiency

procedure OPUS($\{p_{i,j}\}$) ▷ $\{p_{i,j}\}$: caching preference
    $(\mathbf{a}^*, \{T_i\}) \leftarrow$ VCG_PF($\{p_{i,j}\}$) ▷ Seek to share cache
    if Provides_IG($\mathbf{a}^*, \{T_i\}$) **then**
        return $(\mathbf{a}^*, \{T_i\})$ ▷ Settle on cache sharing
    else
        return isolated allocation $\bar{\mathbf{a}}$ ▷ Reduce to isolation

procedure PROVIDES_IG($\mathbf{a}^*, \{T_i\}$)
    for all user i **do**
        if $T_i > \bar{T}_i$ **then** ▷ $\bar{T}_i$: break-even tax following (6)
            return False
    return True

procedure VCG_PF($\{p_{i,j}\}$) ▷ VCG-PF mechanism
    $\mathbf{a}^* \leftarrow$ PF allocation that solves (2)
    for all i **do**
        $\mathbf{a}^*_{-i} \leftarrow$ PF allocation that solves (2) w/o user i's presence
        $T_i \leftarrow \sum_{k \neq i} V_k(\mathbf{a}^*_{-i}) - \sum_{k \neq i} V_k(\mathbf{a}^*)$
    return $(\mathbf{a}^*, \{T_i\})$

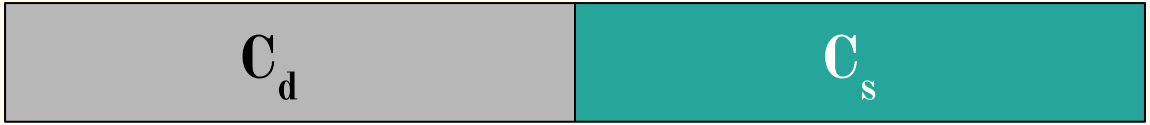
Concept of Private and Public Cache

PRIVATE/DEDICATED CACHE : This is dedicated to solely to the one user and no user can access the items present in private cache of any other user.

PUBLIC /SHARED CACHE : This is also known as shared cache and is shared among all the users. Users compete for it.

HOT ITEMS : Items that are frequently accessed by the one user are considered as hot items for that user.

COLD ITEMS : Items that are not often accessed by the one user are considered as hot items for that user.



Dedicated/Private Cache

Shared/Public Cache

We are talking about Last level cache that has the total capacity of C. This is further divided into C_d (dedicated/private cache) and C_s (shared/public cache).

The items that are hot for a certain user should be moved into the dedicated cache and cold item should be evicted from it.

Dedicated cache is divided proportionally among all the users

Hybrid Cache Architecture

This incorporates an approach for the optimized use of the cache space while satisfying the performance requirement for individual tenants.

Hard requirement: the minimal cache hit rate that should be satisfied at any time for the tenant.

Soft requirement: the desired cache hit rate as long as the cache resources are currently sufficient (not mandatorily but preferably).

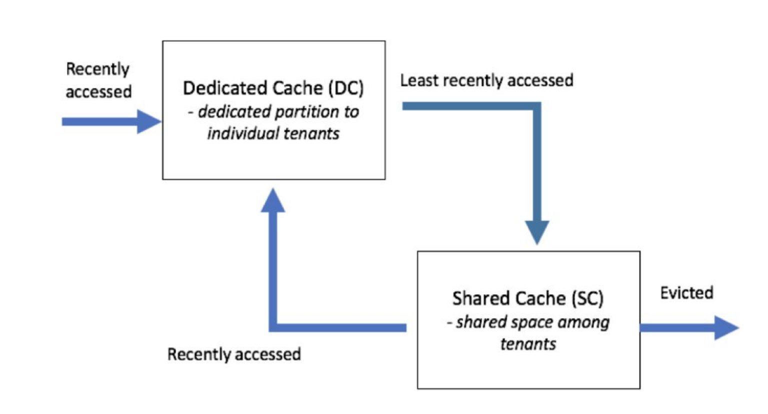
Notation	Description
C	Total cache capacity in the system
U_k	User k in the tenant set U ($u_k \in U$)
H_k	Minimal cache hit rate for u_k (hard requirement)
S_k	Desired cache hit rate for u_k (soft requirement)
h_k	Measured cache hit rate for u_k
g_k	$g_k = h_k - S_k$
G	$G = min(h_k - S_k)$ for all k

Objective 1: For any k, $h_k \geq H_k$

Objective 2: Maximize $G = \text{mink}(h_k - S_k)$ for all k

If $G >= 0$ then the soft/desired requirements are satisfied for everyone or else if $G < 0$ it means that there exists at least one user who has a measured hit rate smaller than the desired one.

The user first utilized its dedicated cache and in case it doesn't have space available it moves to the shared cache memory.



Steps in the Algorithm

- A user u_i wants to cache a file, it looks for space in DC_i and if enough space is available, the file is cached into the dedicated cache.
- If DC_i is full, then we look in shared cache, if it is also full then we find a victim file from shared cache for eviction. Then this new empty slot is assigned to the file we want to cache.
- Now, a swapping takes place between DC_i and SC to keep the hot item in the dedicated cache.
- A victim item is chosen from DC_i that is moved to the slot in SC occupied by the new item, and the new item is stored in the victim slot in DC_i.

Input: a new cache item c_i for u_i;
if *Hit from* DC_i **then**
    return;
end
if DC_i *is not full* **then**
    Insert c_i to an empty slot in DC_i;
    return;
end
if *Hit from* SC **then**
    Find a victim from DC_i;
    Swap the hit slot in SC and the victim slot in DC_i;
    return;
end
if SC *is not full* **then**
    Insert c_i to SC;
    Find a victim from DC_i;
    Swap the inserted slot in SC and the victim slot in DC_i;
    return;
end
$X = \{\text{all active tenant IDs}\}$;
$j = \text{argmax}_{k \in X}(g_k = h_k - S_k)$;
while *true* **do**
    if u_j *occupies any slot in* SC **then**
        Find a victim from SC occupied by u_j;
        Evict the victim;
        Insert c_i to the victim slot in SC;
        Find a victim from DC_j;
        Swap the inserted slot in SC and the victim slot in DC_j;
        break;
    else
        $X = X - \{j\}$;
        $j = \text{argmax}_{k \in X}(g_k = h_k - S_k)$;
    end
end

Prediction of Hot/Cold items

Zipfian distribution is used for analysing the data access pattern that will predict whether a particular item is hot or cold.

- It is a probability distribution that describes the frequency of elements in a dataset. In a Zipfian distribution, the frequency of any element is inversely proportional to its rank.
- Mathematically, if f(k) represents the frequency of the element at rank k, and s is a parameter that determines the distribution's steepness, Zipf's law is expressed as:

$$\boxed{\mathbf{f(k) = 1/k^{s}}}$$

Where k is the rank of the element, and s is the Zipf parameter. The larger the value of s, the steeper the distribution.

Problem Statement

Devise an optimal algorithm for fair allocation of cache where cache has two components - Shared and Dedicated Cache.

Proposed Algorithm

- When a user wants to cache a file, first put it directly into the Shared cache according to the OpUS Algorithm.
- With a probability p, put the same block in it's dedicated cache.
- A block/file will be evicted with the hybrid cache architecture algorithm
- Algorithm will be be fair share and each user will have its locally optimal usage of its cache.

COMPARISON

OPUS	Hybrid Cache Architecture	Proposed Algorithm
• Strategy Proof • Proportional Fair • Solves Free Riding	• It incorporates the concept of dedicated cache and shared cache that guarantees proportional fairness. • Also incorporates the hit rate .	• Has Dedicated and Shared cache • Satisfies the desired properties for a fair cache allocation algorithm and hit rates.

Future Enhancements

- We aim to implement the algorithm using an appropriate simulator so that we can generate results for some cases.
- In order to get better efficiency how to divide files between Dedicated and Shared cache. Formulate a more efficient way.

Thank You