University Of Thessaly

Mobile and Pervasive Computing

ECE515

PROJECT

Authors:
Lefkopoulou Eleni-Maria - 2557
Christodoulos Pappas - 2605

February 14, 2021

OUR : Optimal Update-based Replacement policy for cache in wireless data access networks with optimal effective hits and bandwidth requirements

Abstract

In mobile wireless access network, remote data is expensive concerning terms like bandwidth. It is very important to use a caching scheme that can reduce the amount of data transmission ,hence and bandwidth consumption. The paper that we have to study analyzing OUR policy , a cache replacement policy for wireless data network. We also compared the performance of the OUR, with different replacement policies and also compared their performance of their online and offline forms.

1 Introduction

The basic goal of OUR policy is to increase the hit ratio and to reduce the communication cost . Talking about cache hit let us explain the following terminology.

- **Cache miss :** is the situation when the accessed data object is not in cache .
- **Cache hit:** there are 2 different types of cache hit.
    1. when the latest version of the accessed data object is in cache , so we have a **valid cache hit.**
    2. when an older version of accessed data object is in cache , so we have an **invalid cache hit.**

When we calculate the cache hit ratio we calculate the valid cache hit or in other words the effective hit ratio.
Also let us explain the offline and online policy terminology :

- **Offline Policy :** The offline replacement policy is the policy which uses the actual system variables such as an items update frequency and access probability.
- **Online Policy :** As its name suggests, the replacement policies do not use the actual system variables, but instead they try to approximate them through different calculations.

1.1 Key terms

The hosted objects are identified as Oi where i = 1….N. If the data are not in the same size we consider that they split into fixed-sized smaller pieces. In our study the updates happened only on the server so the client must communicate with the server to learn about the latest version of an object. We need 2 specific metrics for OUR algorithm. First is μ i,j(t) which refers to the access frequency of data item j at client i (to wit how often the client i want to access the item j) and the second is the λ j(t) which refers to the update frequency of data item j at the server (to wit how often the server update the item j) . The μ is separated for its client but the λ is common for all clients and is calculated on the server.

1.2 Implemented API

get_from_cache : Check if the item is in cache
put_cache : Check if exists enough space to put the object else delete from cache the object that is not necessary based on the replacement policy that we use.
access_data : If the object is in cache but the timestamp is wrong we over-write the item data

1.3 OUR replacement Algorithm

Every replacement policy has some basic rules that define which element must be drop from cache its time. In OUR policy it is user=d the performance factor (PF) from the perspective of the client I for object j up to time t as follows :

$$PFi,j(t) = \frac{\mu i,j(t)^2}{\mu i,j(t) + \lambda j(t)}$$

Similarly long term pf is defined as :

$$PFi,j(t) = \lim_{x \to \infty} \frac{\mu i,j(t)^2}{\mu i,j(t) + \lambda j(t)}$$
$$= \frac{\mu i,j^2}{\mu i,j + \lambda j}$$

1.4 Replacement policies used in this paper

As we stated on the abstract, this paper work is a pure comparison of OUR cache replacement policy with some other important policies. Note that we implement their online and offline counterparts. So, in order to understand the different between these algorithms let us describe their role in the list bellow :

- **OUR** computes the PF for ll objects using the access and update history . If the objected that we want to add to cache has higher PF than that of any cached one , the object with the lower PF is evict from cache and the new one is added .
- **LRU** : Discards the least recently used items first. To be more specific, the data in the cache are stored in a queue. Whenever an item is accessed and is in the cache then it is moved on the head of the queue. Otherwise in case the cache is full then we dequeue an item from the queue and store the new item on its head.
- **PIX** : Instead of using the very important and simple P replacement policy, we use the PIX. PIX is often heavily used in pure broadcast environments. In order to import PIX in our system, we use the formula : $\frac{P_i}{\frac{S_i}{BW}}$, where P_i is the access probability of an item of a client, S_i is its size and BW is the bandwidth of the network. We used this replacement policy mainly because it takes into consideration the size of the data item, something that OUR ignores. As in every other policy, if the data item is not in the cache and the cache is full then we remove the item with the smallest PIX value which should be also smaller than the new items PIX value.

2 System implementation details

2.1 Protocol implementation

OUR replacement policy, is a policy originally formulated for pure push system architecture. On these schemes, the client whenever needs to access data, sends a request for that data item in the server and the server replies fulfilling the need of the client for the data item. The cache comes in great significance as it can reduce not only the interaction between the client and the server and thus reducing bandwidth of the network, but also reducing the access time of a data. However, for non read only files, comes the problem of checking the validity of the file stored in cache. In order to validate the files, the client

before accessing the item, polls the server by sending him the timestamp of the data item and also its id. The server checks if the timestamp of the file is identical of that of the client. If that is true, then the server replies with an acknowledgement. Otherwise the server replies with the updated data item and also the items new timestamp. This protocol does not require the server to have any knowledge about the clients and their state, making it stateless. However one important drawback, is that the network bandwidth is loaded by too many polling packets. This can be solved by relaxing the consistency requirements.

2.2 Calculation of replacement policies variables

Another important aspect when implementing a cache replacement scheme is the evaluation of its variables. Many policies such as P , OUR or MIN-SAUD, use in their cost functions variables such as the size of the file, its update frequency or its access frequency. In order to evaluate a_i we use the formula $a_i = \frac{K}{(T-T_k)}$, where K is an integer, T is the current time and T_k is the time of the Kth most recent access of the item i. To implement this we use for each item stored in cache an array of size of K which holds the K most recent access times. Whenever we access the item we dequeue the oldest time , calculate the new a_i and then we enqeue the new time. Another formula that could be used is $a_i = e \cdot \frac{1}{(T-T_{lastaccess})} + (1-e)a_i$.

The update frequency is somewhat more complicated. Sure, the formulas been used for the evaluation of the access frequency of items can also be used for the update frequency. However, if it is implemented naively, we can observe that the client in a polling invalidation scheme can never be sure about the update frequency of the item. This is, because he is unable to learn how many times an item was been updated between its last and its current access. One easy way to solve this is instead of using an actual timestamp, the server keeps a logical timestamp for each data item. Whenever an item is changed, its timestamp is incremented by one. The formula we chose to implement for finding the item update frequency considering that the update of each item follows poisson distribution is $u_i = \frac{Number\_of\_Updates}{(T-T_{first\_update})}$. This can be calculated be the client or by the server biggybacking the item's update frequency on each poll or retreival. Here we have to mention that we implement a much more simplified form of the above formula, which is just the $\frac{Updates_i}{T}$. This formula gives us much better estimations in our experimental evaluation, although it is true that it is not practical, because the update and access rates do not always remain the same through the time.

2.3 Replacement Algorithm

As we stated before, when the data item is not in the cache, and the cache is full, then a data item must be replaced. This is not a problem when the data are of the same size, as we could just replace the new data with the victim one. However a problem emerges when data are not of the same size. For example after removing the victim item there is a chance that there will be again not enough space to insert the new item. Moreover, the algorithm we need in order to select the best set of victim items is NP-complete, and thus its implementation is by no means efficient. For this reason we use a simple and efficient heuristic of removing the items that have the smallest cost until we get enough free space.

3 Experimental Evaluation

3.1 Setup

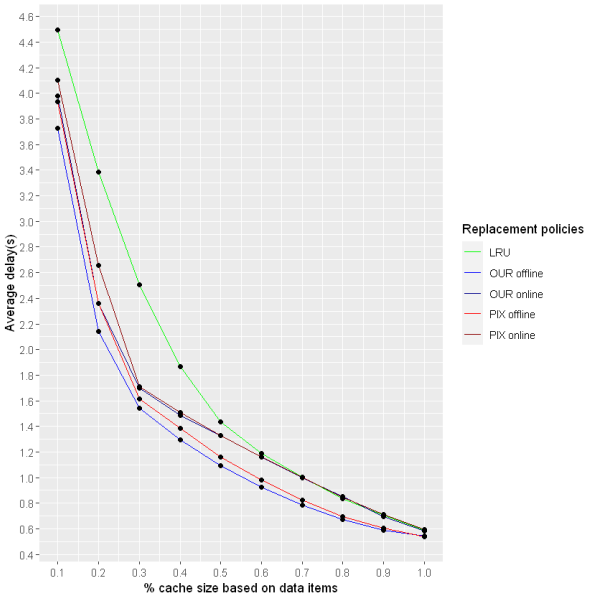
For the experimental evaluation, we created an server and an client. The server takes mainly as input the cardinarity of the data items. Then it generates update traces following a distribution. The distribution that we decided to follow is the zipfian distribution where the value a is given as input. Then following those traces, the server updates the files. The client, takes as input the cache replacement scheme, and cache size and the distribution from which the access stream is generated. From that stream, the client accesses data items and uses the cache. Moreover, we split the data into two distinct categories. The hot data which occupy the 20% of the data and are those data that been accessed more frequently by the client, and there rest of the data are the cold data that are not so frequently accessed by the client. Last but not least, the medium bandwidth is set to 1kb/s .

3.2 Experiments

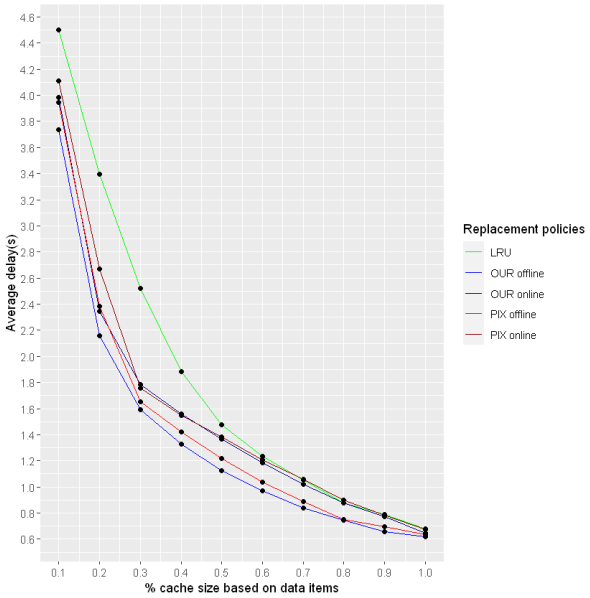
To begin with, on each experiment, there are 500 different items stored in the server, and the client makes 5000 accesses on those data. On the first experiment we test the hit rate of the different replacement algorithms and also their average access delay, when the data are on different sizes uniformly between 1kb through 10kb. The second experiment tests the cache hit rate and average access delay on data with the same 1kb size. The third and fourth experiments, test the hit rate and average access delay on data that are read only, and have correspondingly different and same sizes as the first two experiments.

Here you cam see the graphs for the different test caches.

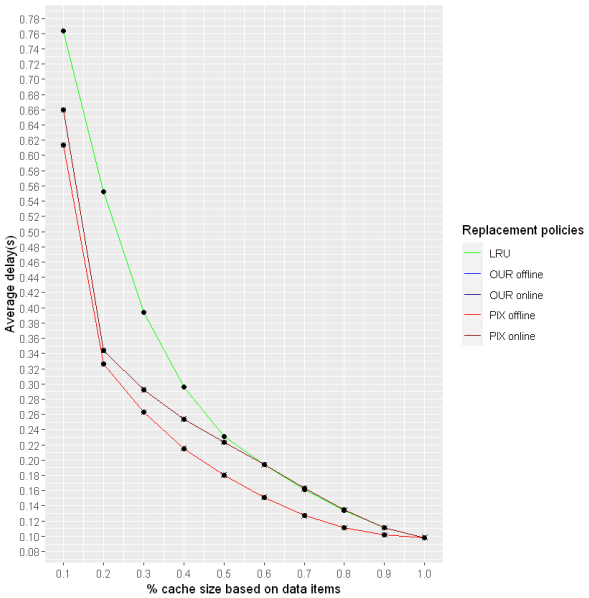
Delay-cahce size graphs for all policies



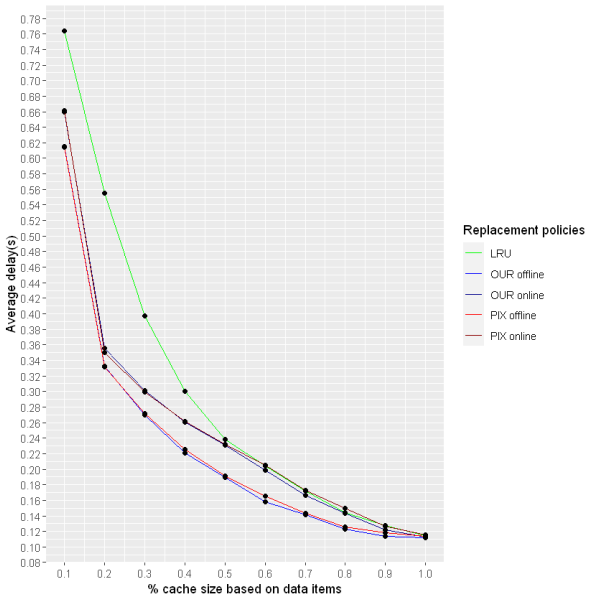
data with different sizes
read only



data with different sizes

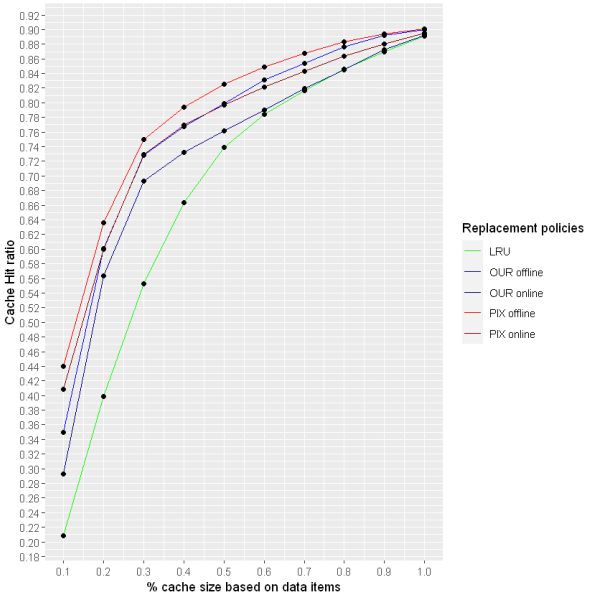


data with same sizes
read only

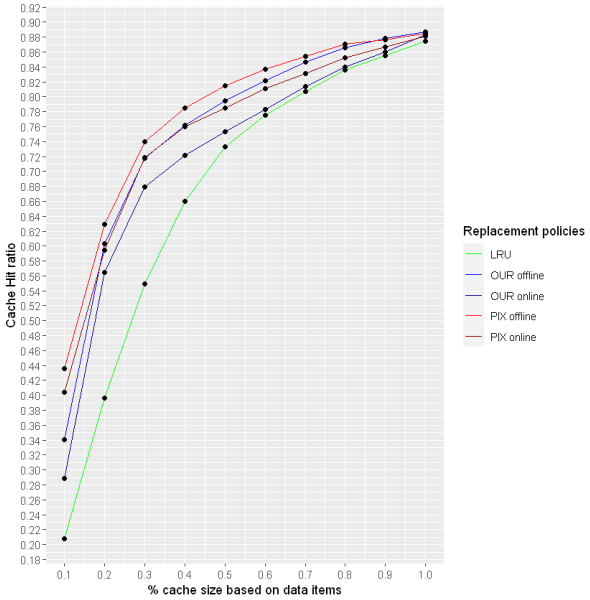


data with same sizes

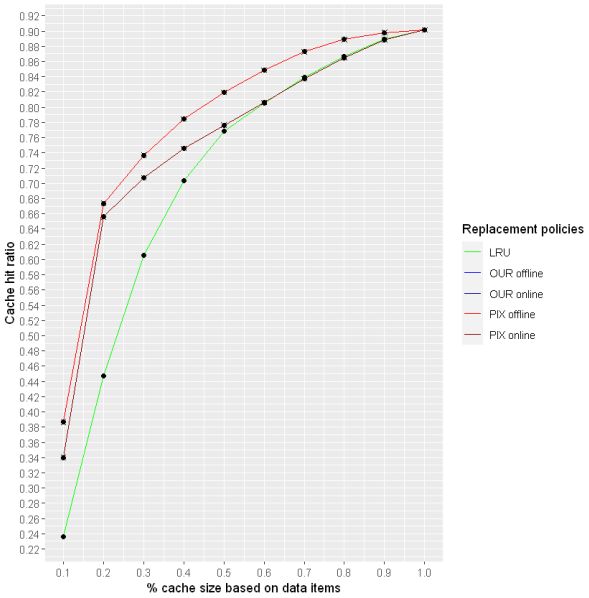
Hit rate -cahce size graphs for all policies



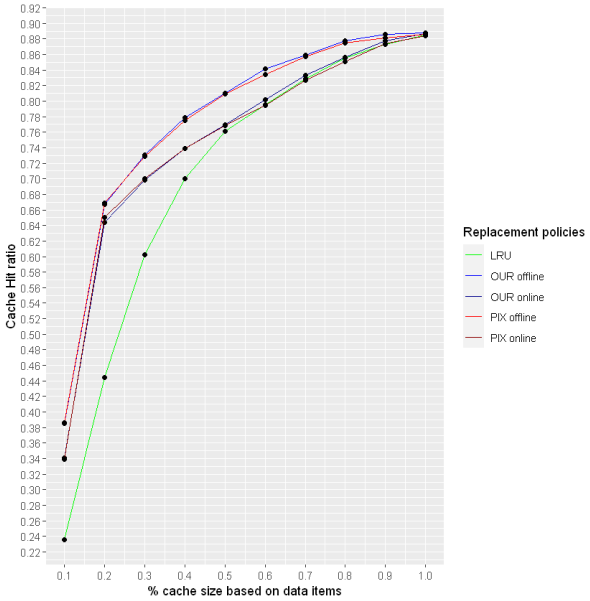
data with different sizes
read only



data with different sizes



data with same sizes
read only



data with same sizes

3.3 Discussion

First of all, as we could expect, the LRU lags behind in every experiment, and converges only with the other replacement algorithms when the cache size is large enough. Moreover we can see that the offline implementation of the algorithms, provide better results in contrast to the online ones. This is not an unexpected behavior, as offline implementations have a clear view of systems variables and thus provide more accurate results.

Something very interesting is the hit ratio of the PIX algorithms versus the OUR algorithms. Seemingly, we can see that PIX gives much better hit cache ratio especially when the algorithm is performed in different data sizes. However cache hit ratio should not fool us. If we take a closer look at the average delay graphs, we can see that OUR algorithms outperform PIX algorithms in term of average access delay. This happens mainly because PIX and OUR algorithms don't take into account the update rates of the data items and in result they have more invalid cache hits. So, in case the data access delay is negligible, which usually happens when we have a very fast internet connection, the PIX is preferred as it provides higher hit ratio. Otherwise we should use OUR algorithms.

Finally, we observe that for data of the same size, the performance of the OUR and PIX algorithm is roughly the same and especially when data are read only which is exact the same. If we take a closer look at the formulas of those algorithms, indeed, for same size, read only data, the two formulas become the same and are equal to the probability of access of a data item.
