

An Intelligent Cache Management for Data Analysis at CMS

Mirco Tracolli[1,2,3(✉)], Marco Baioletti[1(✉)], Diego Ciangottini[3(✉)], Valentina Poggioni[1(✉)], and Daniele Spiga[3(✉)]

[1] Università degli Studi di Perugia, Perugia, Italy
{marco.baioletti,valentina.poggioni}@unipg.it
[2] Università degli Studi di Firenze, Florence, Italy
[3] INFN Sezione di Perugia, Perugia, Italy
{mirco.tracolli,diego.ciangottini,daniele.spiga}@pg.infn.it

Abstract. In this work, we explore a score-based approach to manage a cache system. With the proposed method, the cache can better discriminate the input requests and improve the overall performances. We created a score based discriminator using the file statistics. The score represents the weight of a file. We tested several functions to compute the file weight used to determine whether a file has to be stored in the cache or not. We developed a solution experimenting on a real cache manager named XCache, that is used within the Compact Muon Solenoid (CMS) data analysis workflow. The aim of this work is optimizing to reduce maintaining costs of the cache system without compromising the user experience.

Keywords: Cache · Optimization · LRU · Intelligent system · Big data · Data science workflow

1 Introduction

The Compact Muon Solenoid (CMS [5]) collaboration is one of the four major experiments at the Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN). The experiments that involve the detector and the physics simulations, such as the Monte Carlo simulations, create a considerable amount of data every year. These data are kept on disk (Hard Disk Drives, HDDs) and tape distributed storage systems. HDDs are used for storing analysis data of physicist users because they are much faster than tape, but they are more expensive and hence disk space is limited. The analysis workflow implies that the jobs run on the site (tier) where the requests are made. The current storage system is hierarchical organized (tier levels, Fig. 1) and centrally managed, hence it is not autonomous and it cannot react as a dynamic environment.

The next decades will be characterized by the LHC upgrade to HL-LHC (High Luminosity Large Hadron Collider). As a consequence, an increase of factor 20 is expected for the storage requirements while, on the computing side,

© Springer Nature Switzerland AG 2020
O. Gervasi et al. (Eds.): ICCSA 2020, LNCS 12250, pp. 320–332, 2020.
https://doi.org/10.1007/978-3-030-58802-1_24

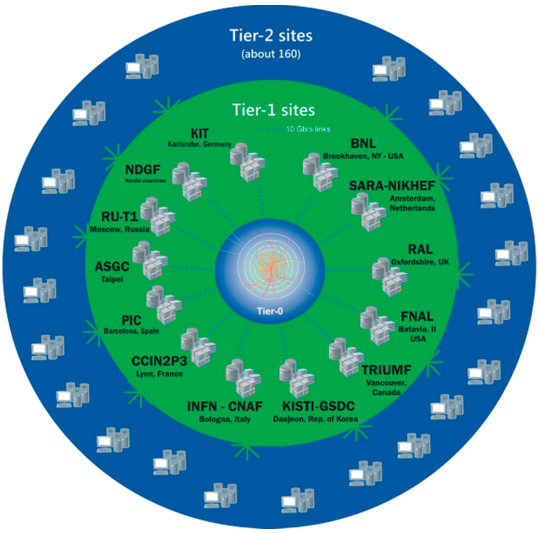

Fig. 1. Current data distribution model [4]

the estimation is about a 30x CPUs necessity. Moreover, we foresee a shift in resources provisioning towards the exploitation of dynamic solutions using private or public clouds and also High-Performance Computing facilities. In this scenario, a key point is an evolution and optimization of the amount of space that is custodial and the CPU efficiency of the jobs that run on "storage-less" resources. In particular, jobs can be instrumented to read data from a geographically distributed cache storage. The cache system will appear as a distributed and shared file system populated with the most requested data; in case of missing information data access will fallback to the remote access. Furthermore, in a possible future scenario based on the data-lake model (Fig. 2), it is reasonable to imagine that many satellite computing centers might appear and disappear dynamically as needed. In this sense, an intermediate and auto-sufficient layer against a centrally managed storage might be a keystone to maintain unaltered the user experience.

A Content Delivery Network (CDN) has many affinities with the cache architecture design, but the cache system has a finer grade infrastructure: the cache layer could be divided into regions (a composition of federated servers) in which each cache not foreseen duplicate files within its federated servers. The cache storage will be, by definition, "non-custodial". Thus, it reduces overall operational costs. To get an idea on the dimensions of the problem we consider, from statistic analysis, that a single cache could manage thousands of requests per day and have a realistic dimension starting from 100 Terabytes of storage.

The main actor in this scenario is XRootD [2,12], a modular, fast, low-latency and scalable file server. This service could be configured in several ways, the main ones are as Redirector and as Data Server. A Redirector can communicate with

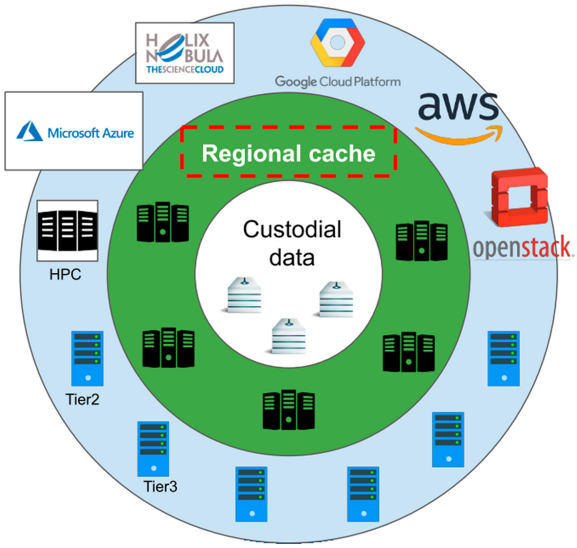

Fig. 2. Data lake model [4]

other Redirectors to resolve a user request. To resolve the request, the Redirector has to find the requested file into one of the available Data Servers and it has to make contact between the user and that server. A Data Server simply serves data from the memory and it implements the data caching part. The Data Server uses the Least Recently Used (LRU) replacement policy as default.

The main reason to choose XRootD is that it has integrated the CMS workflow in a transparent way for the end-users. In addition, the XRootD service supports a plugin system with which we can interact directly with the cache

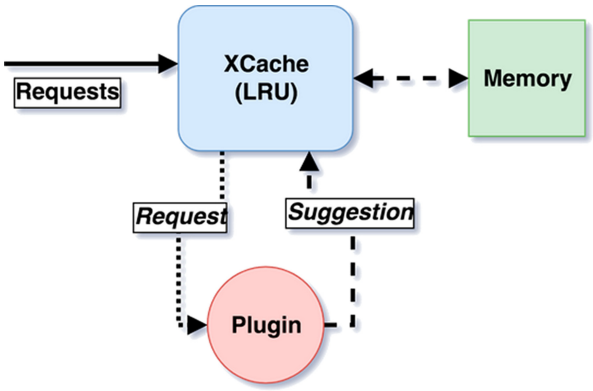

Fig. 3. XRootD plugin schema

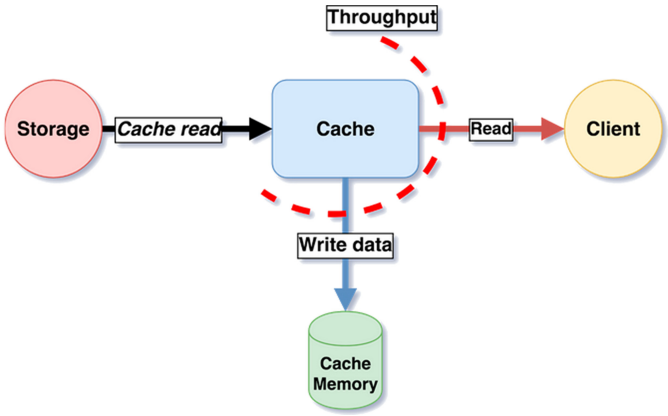

Fig. 4. Schema of the environment

system. The target of this work is to interact with this system finding a method to suggest which files to store or not into the cache (Fig. 3).

With our customized suggestion we target to maximize the throughput of the cache. As you can see in the schema in Fig. 4, we want to minimize the data written by the cache and maximize the amount of data read from the clients.

2 Related Works

As we said in the previous chapter, the introduced data science workflow has many affinities with the CDN networks and also with the web content caching. The purpose of this intermediate layer is to decreases user-perceived latency, to reduces network bandwidth usage, and to reduces loads on the origin servers. Thus, the cache plays a very important role in the whole picture and we need to design an efficient cache replacement algorithms to achieve better performances.

The LRU algorithm is a simple queue where the items are ordered with respect to the times of the last access. Thus the most recently accessed item is placed on top. When a file is requested and it is already in the queue, it is moved to the top. If we need to remove a file, we start from the tail of the queue and we remove the files that are not too much accessed in recent history. Such a policy is known to under-perform in the big data environment as it was initially designed for evicting fixed-size pages from buffer caches[1].

Anyhow, the main characteristics that influence the replacement process are still the same [3,7,11] and they are related only to the file (or object) requested F:

- recency: time of (since) the last reference to F
- frequency: number of requests to F
- size: the size of F

We can add also two more features such as the cost, that represents the price to pay to store F, and some information relative to the access to F, such as the latency or the distance.

Recently, there are several Machine Learning techniques that are used to improve this kind of cache. For instance, in [11] a Deep Neural Network is trained in advance to better manage the real-time scheduling of cache content into a heterogeneous network. In [9] a Deep Recurrent Neural Network is used to predict the cache accesses and make a better caching decision.

Also, other techniques, like the Gradient Boosting Tree, are used in [6] to automate cache management of distributed data.

There are also Deep Reinforcement Learning approaches like [10].

In this article, we will not use a specific Machine Learning technique, but the problem size that we tackle is much higher than the articles mentioned before, both in terms of the number of files and domain-specific features (related to physic data analysis).

3 Data Analysis

The source data used in this work are request logs of user analysis. These logs have information about user requests, such as the filename, etc. The information contained in this database refers to the data popularity research made in CMS[8]. The collected data are archived per day and we focused the experiments on a specific time span, the whole year 2018. Also, to have a realistic simulation of a single cache used within the current data analysis workflow, we decided to filter the requests by region, for example, we choose only the requests coming from Italy (Fig. 5).

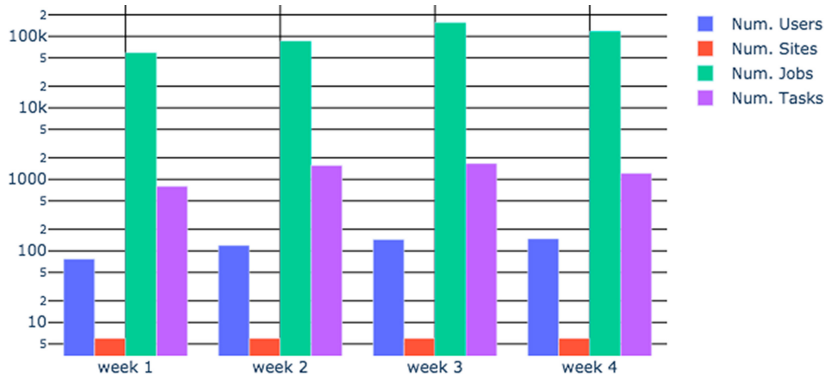

Fig. 5. Data general statistics of Italy requests in January 2018

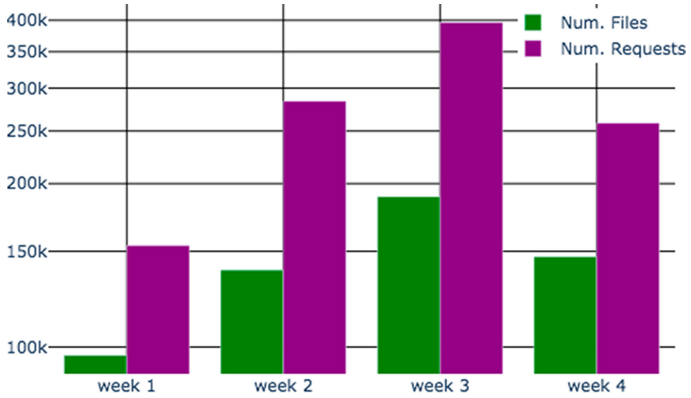

Fig. 6. Number of request statistics of Italy requests in January 2018

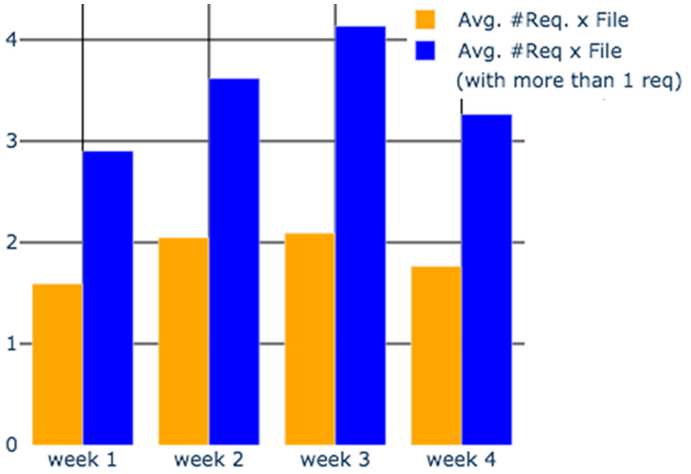

Fig. 7. Average number of request statistics of Italy requests in January 2018

A preliminary analysis of data gives us useful information about the dimension of the problem. In the region IT (Italy), the number of files requested is very high with respect to the number of users and sites where the jobs are run (Fig. 5, 6). The workflow proves that the user can launch tasks having more jobs, and those jobs can request several files. Nevertheless, the number of requests per file is not too high and the average value is small even if we not consider the files requested only 1 time Fig. 7.

However, this trend of the request statistics is confirmed in the other regions and, as a consequence, we expect to have comparable results regardless of the region.

4 Proposed Approach

Our approach is based on the concept of file weight (a score) which is used to determine if the cache has to accept or not the file. The policy is: a file with a smaller weight is more likely to be inserted into the cache. To accomplish this task, we collect statistics about the file to compute the score of a file f at time t, considering the following features:

- $numRequests(f, t)$: the number of requests (frequency) to f until the current time t
- $size(f)$: the file size
- $requestDelta(f, t)$: the average time difference between the previous file requests and the last one (a relative recency)

The average time difference is calculated on the last $k = 32$ requests (or on all the last requests, if the file has been requested $k < 32$ times), according to the following formula:

$$requestDelta(f,t) = \frac{\sum_{i=1}^{k-1} time(f,k) - time(f,i)}{k} \quad (1)$$

Where $time(f, i)$ is the time of the i–th request to f ($time(f, k)$ is the time of the last request).

The statistics to compute the score (or weight) of a file are maintained in a time window of 14 days. The threshold to accept a file is calculated on the median value of all the collected file weights.

We have selected 3 functional forms of the weight function which aggregate in different ways the three statistics $numRequests(f, t)$, $size(f)$, and $requestDelta(f, t)$.

Each of the 3 forms have α, β, γ as parameters

- Additive form

$$weight_A(f,t) = \alpha \cdot numRequests(f,t) + \beta \cdot size(f) + \gamma \cdot requestDelta(f,t) \quad (2)$$

- Additive-Exponential form

$$weight_E(f,t) = numRequests(f,t)^{\alpha} + size(f)^{\beta} + requestDelta(f,t)^{\gamma} \quad (3)$$

- Multiplicative form

$$weight_M(f,t) = numRequests(f,t)^{\alpha} \cdot size(f)^{\beta} \cdot requestDelta(f,t)^{\gamma} \quad (4)$$

The main measure we focus, as mentioned in the Sect. 1, is the *throughput*, defined in the following way:

$$throughput = \frac{readOnHitData}{writtenData} \quad (5)$$

where *readOnHitData* is the amount of Megabytes on which the cache had hit and the *writtenData* is the amount of files written (in Megabytes) into cache. However, this measure does not give the whole picture of the effect of our method on the cache system. Moreover, we cannot use only the hit rate measure because we take into account the size of the files. Consequently, we defined the following measures to test the impact of the new strategies:

– Cache disk space: this measure, named **Cost**, quantifies the work done by the cache: the sizes of the files deleted or written on the local storage and the files served in proxy mode. This latest scenario is influenced by the miss data. Hence, this measure can be written as follows:

$$Cost = writtenData + deletedData + readOnMissData \tag{6}$$

– Network bandwidth: to measure the impact of this constraint we simply take into account the miss data over the bandwidth of the cache. If the network is completely saturated, the cache cannot retrieve files anymore:

$$BandSaturation = \frac{readOnMiss}{cacheBandWidth} \tag{7}$$

– CPU efficiency: due to the log files we use, we have access to the CPU time and Wall time of each request. This allow us to measure the CPU efficiency as follow:

$$CPUEfficiency = \frac{\sum CPUTime}{\sum WallTime} \tag{8}$$

We calculate the upper and lower bound of the CPU efficiency using information from the logs. We collect the local CPU efficiency (file served from a tier, indicated in the log features) for the upper bound and the remote CPU efficiency for the lower bound. Then, we consider the difference (19% for the year 2018) to measure the CPU efficiency into the simulation, starting from the log CPU efficiency. We consider a file served by the cache as if it is local, otherwise, it will be a remote file, thus, we subtract the difference in percentage to simulate the loss in performance.

The simulation also takes care about the watermark behavior of the XCache: there are two watermarks, a higher and a lower watermark. When the size of the cache reaches the higher watermark, the cache removes some files until the lower watermark is reached. In our simulations, the cache size was 100 Terabytes and the watermarks was set respectively to 95% and 75% of the cache size.

We used the following algorithm to perform the tests:

To resume the Algorithm 1, we insert a file into the cache only if the file weight is less or equal the average weight of all files we have into the statistic table. We used this algorithm with each functional forms $weight_A, weigth_E, weight_M$ by considering the following values for the parameters α, β and γ: $\{0, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, 1, 2, 4\}$.

```
Data: log requests
Result: a cache simulation
Initialization of variables and statistic table
for each requested file f do
    t ← time of the request
    hit ← checkFileInCache(f)
    updateStatistics(f)
    if hit then
        updateLRUQueue(f)
    else
        if weight(f, t) ≤ avg(statsTable.weights) then
            insertWithLRU(f)
        end
    end
    checkWatermark()
end
```

Algorithm 1: Algorithm used to test the functional form of the weight function

5 Experimental Results

Table 1 shows the three measures *Throughput* (Eq. 5), *Cost* (Eq. 6), and *ReadOnHitData* for the LRU cache strategy (used as a baseline in the comparisons) and the best 10 combinations of parameters in each of the 3 functional forms (Eq. 2, 3, 4). The whole results are sorted by throughput, cost and read on hit amount. The main measure we take into account is the throughput because we want to optimize the cache work.

It is possible to see that all the proposed weight functions have a smaller cost with respect to LRU. They have a better throughput, so they write less data into the cache and they still make the clients read a considerable amount of data. They are still outperformed by LRU on *ReadOnHitData* value, but the global work made by the cache is considerably less.

The best functional form was the Additive with Exponential Form, having the best performances in terms of Throughput and Cost. Anyway, it is worth noting that the Additive Form reaches higher *ReadOnHitData* values. Moreover, by observing the parameter values, it is possible to deduce that the number of requests plays a secondary role. Indeed, the value of the α parameter of the best functions is 0.00 and only few of the results have a different value. Two possible explanations are that the range of the number of requests of a file is not comparable with the other two variables (the size of the file is in Megabytes and the delta time in minutes) or that α should take a much higher value to have a significant impact on the score.

During the simulation, we monitored the other measure values using several plots. We define a upper an lower bounds for the measures *BandSaturation* (Eq. 7) and *CPUEfficiency* (Eq. 8) using the data log information. The upper bound for *BandSaturation* is the entire network bandwidth we would like to simulate ($10Gbit$). The upper bound for *CPUEfficiency* is calculated by considering all the files present in the cache, while the lower bound is computed by considering all the files accessed by remote. In the following part we will see a specific view of the year of the top three functions.

Table 1. Test results grouped by function family.

cache				**Throughput**	**Cost**	**ReadOnHitData**
LRU				0.483708	178349611	26327729
Form	α	β	γ	**Throughput**	**Cost**	**ReadOnHitData**
Additive	0.00	0.50	2.00	0.643959	173774548	18793149
Additive	0.00	1.00	4.00	0.643959	173774548	18793149
Additive	0.00	0.50	0.67	0.642360	173757770	18580175
Additive	0.00	0.33	2.00	0.640429	174542198	18623528
Additive	0.00	0.67	4.00	0.640429	174542198	18623528
Additive	0.33	1.00	4.00	0.640004	174010336	18556491
Additive	0.00	0.67	1.00	0.638810	174277036	18672706
Additive	0.00	0.33	0.50	0.638810	174277036	18672706
Additive	0.00	0.33	1.00	0.637943	174116598	18833412
Additive	0.00	0.67	2.00	0.637943	174116598	18833412
AdditiveExp	**0.00**	**0.50**	**0.33**	**0.737599**	**150626930**	**13428482**
AdditiveExp	0.00	0.33	0.33	0.722364	160052405	15912505
AdditiveExp	0.00	0.50	0.50	0.715685	161956500	16153019
AdditiveExp	0.00	0.67	0.67	0.696146	161609673	15457155
AdditiveExp	0.00	0.33	0.00	0.693033	158801312	15735244
AdditiveExp	0.00	0.67	0.50	0.68763	156525555	15012763
AdditiveExp	0.33	0.67	0.67	0.681264	162213056	15011936
AdditiveExp	0.00	0.50	0.00	0.671477	163232309	16836007
AdditiveExp	0.00	0.33	0.50	0.668889	168519018	16758717
AdditiveExp	0.00	0.67	0.33	0.665731	164003690	17574075
Multiplicative	0.00	0.33	0.00	0.693033	158801312	15735244
Multiplicative	0.00	0.50	0.00	0.671477	163232309	16836007
Multiplicative	0.33	0.33	0.33	0.665997	173205320	17519780
Multiplicative	0.00	0.67	0.33	0.664488	172794102	17472908
Multiplicative	0.33	0.00	0.33	0.663508	173055914	17577131
Multiplicative	0.00	1.00	0.33	0.658037	173540416	17428691
Multiplicative	0.00	0.00	0.33	0.657345	172689485	17367026
Multiplicative	0.00	0.33	0.33	0.655531	172804247	17339478
Multiplicative	0.00	0.67	0.00	0.654745	168199648	18575368
Multiplicative	0.33	0.67	0.33	0.652380	173510455	17457726

As a first result, the higher throughput (Fig. 8, Eq. 5) of the functions during the time is maintained in all the simulations. This demonstrates that the constraint to write less and maintain a high output towards the clients is respected.

Besides, we considered during the tests that the available bandwidth was $10Gbit$ and we saw that the miss files have, as consequence, a heavier load on the network. In fact, the bandwidth is always higher than the LRU cache (Fig. 9, Eq. 7). It is possible to understand, from the plot, that the limit of 100% is overcome in several points. Hence, there is not enough bandwidth to satisfy all

the traffic requested by the clients. The greater request of the network due to the miss files is not a good point for the weight functions and we need better control over this aspect.

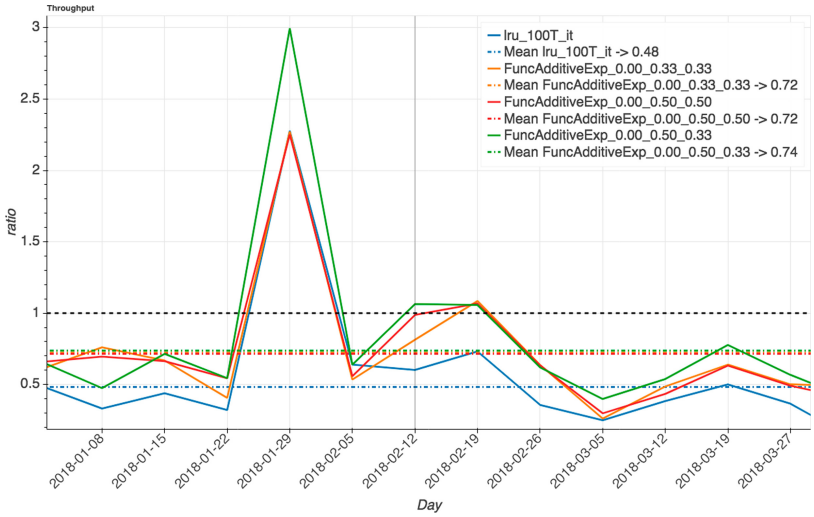

Fig. 8. Throughput detail of the period between January and March with the top 3 functions

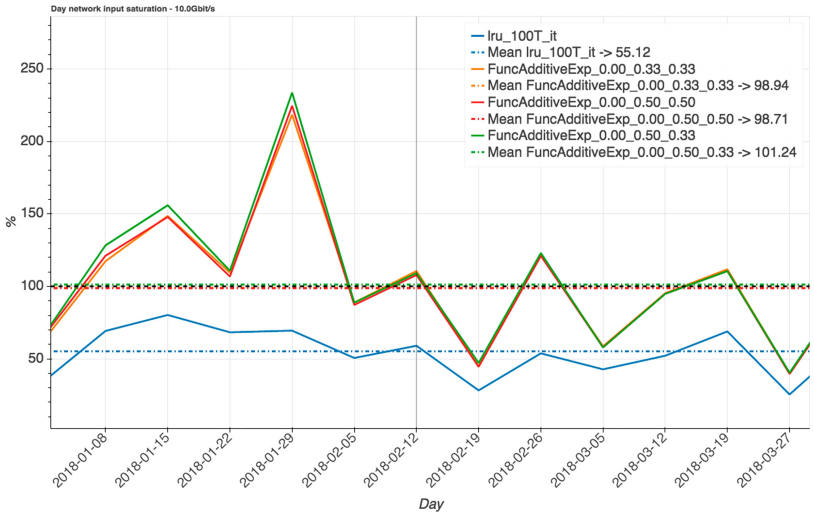

Fig. 9. Bandwidth detail of the period between January and March with the top 3 functions

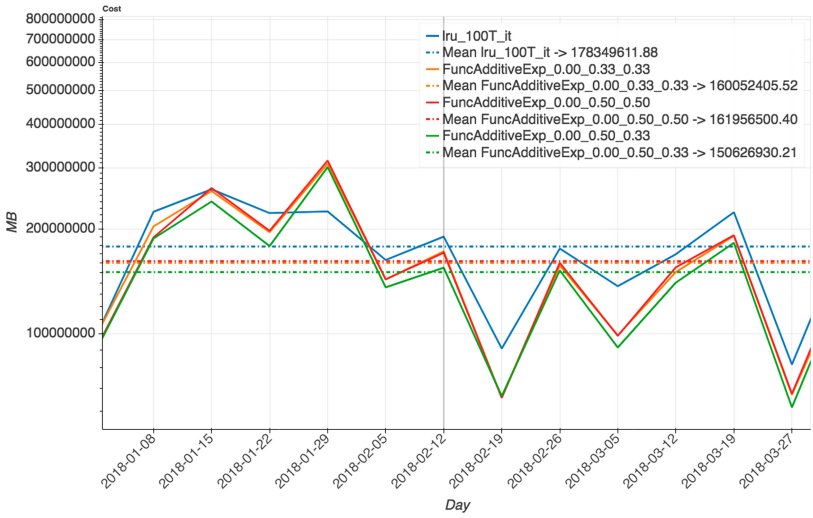

Fig. 10. Cost detail of the period between January and March with the top 10 functions

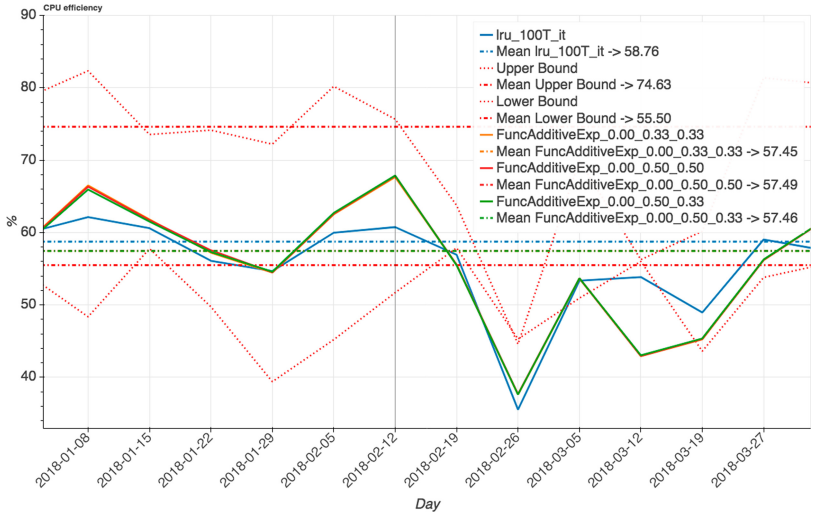

Fig. 11. CPU efficiency detail of the period between January and March with the top 3 functions

Instead, the cost measure (Fig. 10) demonstrates that the target of the function to do less work (from the cache perspective) is fulfilled. The LRU policy cannot compete with the weight functions in terms of the cost measure.

In the end, despite the bandwidth poor results, the CPU efficiency (Fig. 11) seems not to be compromised and still remaining the same. It is around 57% in all the proposed solutions.

6 Conclusions

This work puts a background on how to measure a different behavior of a smarter cache system. Better content management could lead to a more efficient cache, with extra regard to the domain-specific information. The target is to enhance all the measures without losing these improvements in cost and throughput. In particular, we want to decrease the gap between the current caches and the upper bound in the CPU efficiency.

An intelligent cache should have a better CPU efficiency to fulfill the clients expectations. A preliminary study on this aspect found a correlation between the amounts of data read when there is an hit and the CPU efficiency. Hence, the further studies will focus on improve the hit and miss ratio, to optimize the network utilization and, as a consequence, the global CPU efficiency.

References

1. Ali, W., Shamsuddin, S.M., Ismail, A.S., et al.: A survey of web caching and prefetching. Int. J. Adv. Soft Comput. Appl **3**(1), 18–44 (2011)
2. Bauerdick, L., et al.: Xrootd, disk-based, caching proxy for optimization of data access, data placement and data replication. J. Phys.: Conf. Ser. **513**(4), 042044 (2014)
3. Chen, T.: Obtaining the optimal cache document replacement policy for the caching system of an EC website. Eur. J. Oper. Res. **181**(2), 828–841 (2007)
4. Ciangottini, D.: Integrazione di una smart cache italiana federata per CMS. Talk at CCR, Italy (2019)
5. Collaboration, C.M.S., et al.: The CMS experiment at the CERN LHC. Jinst (2008)
6. Herodotou, H.: Autocache: employing machine learning to automate caching in distributed file systems. In: International Conference on Data Engineering Workshops (ICDEW), pp. 133–139 (2019)
7. Koskela, T., Heikkonen, J., Kaski, K.: Web cache optimization with nonlinear model using object features. Comput. Netw. **43**(6), 805–817 (2003)
8. Kuznetsov, V., Li, T., Giommi, L., Bonacorsi, D., Wildish, T.: Predicting dataset popularity for the CMS experiment. arXiv preprint arXiv:1602.07226 (2016)
9. Narayanan, A., Verma, S., Ramadan, E., Babaie, P., Zhang, Z.-L.: Deepcache: a deep learning based framework for content caching. In: Proceedings of the 2018 Workshop on Network Meets AI & ML, pp. 48–53 (2018)
10. Sadeghi, A., Wang, G., Giannakis, G.B.: Deep reinforcement learning for adaptive caching in hierarchical content delivery networks. IEEE Trans. Cogn. Commun. Netw. **5**(4), 1024–1033 (2019)
11. Tian, G., Liebelt, M.: An effectiveness-based adaptive cache replacement policy. Microprocess. Microsyst. **38**(1), 98–111 (2014)
12. XRootD. Xrootd homepage (2020). Accessed 4 Mar 2020