CERN Summer Student 2022

Caching for Analysis Workloads of the ATLAS LHC experiment

Size-Aware Caching Policies

Student:

Sergiu Bogdan Popescu

Supervisors:

Olga Chuchuk
Markus Schulz

Table Of Contents

1. Introduction and Starting Point
2. LRU Threshold
3. GDSF - Greedy Dual Size Frequency
4. ML Caching Policy
5. Conclusions

Introduction and Starting Point

My 2022 CERN Summer Student project has been based on the **study of cache management** in large, geographically distributed systems used for scientific computation. In particular, we considered as a use-case the Large Hadron Collider (LHC) located at CERN (the European Center of Nuclear Research), which is the world's largest particle accelerator. This unique and complex facility allows scientists to study elementary particles and fundamental laws of physics while posing specific storage and computing challenges. To solve them, scientists and engineers have developed the Worldwide LHC Computing Grid (WLCG), which is a federation of large and small computing centers distributed around the globe (in total, around 170 sites located on 5 different continents).

Motivation

Currently, most data analysis in WLCG is performed in the so-called "local grid analysis" mode, where data expected to be processed is moved/replicated to the processing site before the computation starts. This mode requires manual intervention of the physics groups to curate the data transfer process and leads to an abundance of data replicas throughout the WLCG, which is incompatible with large data volumes generated by the future HL-LHC.

For these reasons, there has been considered a remote computation model, in which the processing unit automatically fetches data over the WLCG network based on users' requests. In this way, there is no need to couple the CPU resources with large (and expensive) permanent storage systems. Instead, working data can be placed in **small-sized caches**, implemented through simple storage systems with a low level of service. Therefore we focused our work on **new caching policies** that exploit specific knowledge of the transferred data and compare them with already existing state of the art caching strategies.

Considering this computing architecture, we also have to keep in mind the data that is fetched by the processing unit needs to always be present when the computation happens. Thus, in this specific scenario, the cache policies that also have an selective admission (not all the requested files are admitted into the cache) do not fit our requirements. Some of the caching policies I consider in this work also have the selection admission. We either studied them as they are to evaluate the potential first, or adapted them to admit all the requested files.

Contributions

During my work here, apart from understanding the WLCG architecture and the existing cache algorithms that are used to optimize its data transfer, I mainly focused on implementing new caching policies. In total I have worked on **3 different strategies regarding cache implementation**.

The first 2 were based on well known papers which describe state of the art policies that have been proven to bring adequate results on large variable sets of data. The first policy is presented in **AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery**

Network and the second one in **Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy.**

Furthermore, taking in account the importance of the file details when it comes to designing a new caching policy, and the large amount of characteristics that data files generated at CERN can have, we also came to the conclusion that a machine learning based caching policy could bring satisfactory results. As a consequence of that, I successfully implemented a full ML pipeline: preprocessing and feature engineering the data, selecting a ML algorithm, selecting a ML model, training the model and using the model for building a cache policy. This would conclude the 3rd strategy.

Previous work

The main purpose of my CERN summer student project was to assist my team, particularly my supervisor, in the study of caching policies. All the study and implementation I did, had as starting point the work my supervisor has done so far in this area. It is described in full detail in the following paper: **Caching for dataset-based workloads with heterogeneous file sizes.** In the following paragraphs I will summarize the key elements of it:

A three-month trace of user accesses of the ATLAS experiment at the CERN data center (the largest of the WLCG sites) has been extracted. This trace has been analyzed, highlighting key findings relevant to the analysis and design of caching systems.

My supervisor then compared the performance on this trace of widely used caching policies, like LRU, 2-LRU [1–3] and clairvoyant optimal policies [4] that minimize either the File Miss Ratio (FMR) or the Byte Miss Ratio (BMR).

Metrics of interest

The standard metrics used to evaluate the cache performance is the File Hit Ratio (FHR), and the Byte Hit Ratio (BHR). They can be defined as follows: FHR = $Ncache / N$, BHR = $Vcache / V$ where N is the total number of requests (or the trace length), V is the total volume of the catalog, i.e., the total size in bytes of all files which have been requested at least once, $Ncache$ is the number of files retrieved from the cache (the total number of hits), and $Vcache$ is the total number of bytes served by the cache. File Miss Ratio (FMR) and Byte Miss Ratio (BMR) are calculated as: FMR = 1 − FHR, BMR = 1 − BHR.

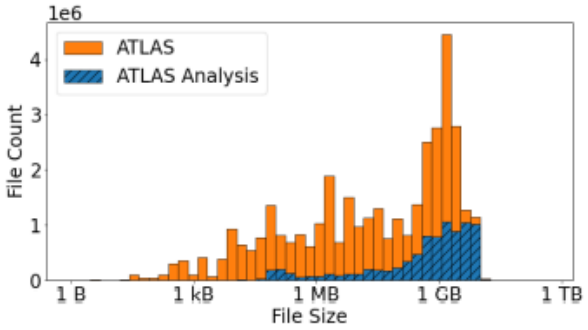
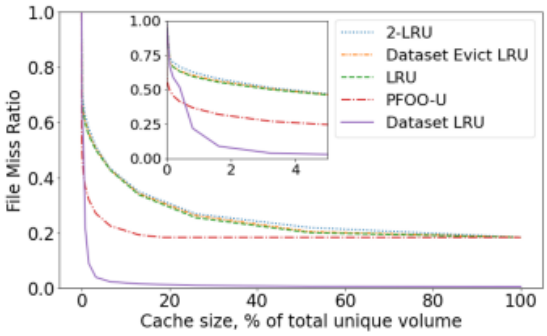
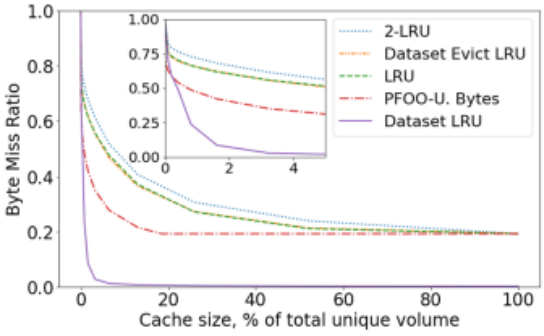


Figure 1: File size distribution.



(a) File Miss Ratio



(b) Byte Miss Ratio

The performance of the caching policies was evaluated by simulating the cache behavior using the same three-month trace reflecting the request process. In each case, the initial state of the system is an empty cache. The size of the cache decreases in size by log2, therefore, for smaller size caches, we have more data regarding the behavior of the cache policy.

These plots show that among the reactive techniques we considered, LRU remains the best option, thus in future comparisons we will use the LRU performance as a starting point.

LRU Threshold

The first cache policy that we are interested in is called **AdaptSize** and is described in **AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network**.

Most major content providers use **Content Delivery Networks (CDNs)** to serve web and video content to their users.

This paper proposes the AdaptSize cache policy: an adaptive, size-aware cache admission policy for **CDN** that achieves a high FHR, even when object size distributions and request characteristics vary significantly over time. At the core of AdaptSize is a novel Markov cache model that seamlessly adapts the caching parameters to the changing request patterns. This algorithm is based on finding the **best C threshold** regarding size, for using it as an **admission criteria** (files with size lower than C can be added). The paper describes that for their use case (CDN) the value of C is **not constant during time** (it changes during the day, depending on what content is requested by the user), therefore they try to find the optimal value after each T period of time, using Markov chains and probability.

WLCG Scenario

As described in the Introduction Paragraph, our work at CERN is based on studying new ways of implementing the cache policy for WLCG, thus improving the transfer of data between different nodes of the computing grid. Even though the type and size of the files that are transferred inside the WLCG varies a lot, it still is not as close to the type of data that a CDN, as it is described in the paper above, has to manage. Therefore, we decided that before we implement the AdaptSize caching strategy (which seems to be quite hard), it would be useful to test the caching policy that stands at the base of AdaptSize. As a result we implemented LRU with an admission policy based on size threshold, thus only files with a size below that can be admitted. We will call this caching policy **LRU Threshold**.

Implementation of LRU Threshold

We used the same algorithm as with LRU, but we add a condition that states the following: ***a file is admitted only when its size < threshold***. The policy was tested with different sizes of the threshold variable: 100MB, 1GB, 5GB and 10GB.

Results

As shown in Figure 9 and Figure 10, the results are rather unexpected. Not only is there no improvement of FMR or BMR compared to LRU when using any of the thresholds listed before, but except for the case when the threshold is 10GB (when it behaves similarly to LRU), the performance of this cache policy gets worse as the size gets smaller.

When we reduce the size of the threshold we start to see that the performance goes

down and at some point the FMR and BMR go flat (this happens because the cache has enough space to store more files but all the files with size < threshold are already in the cache, therefore no new files are added).

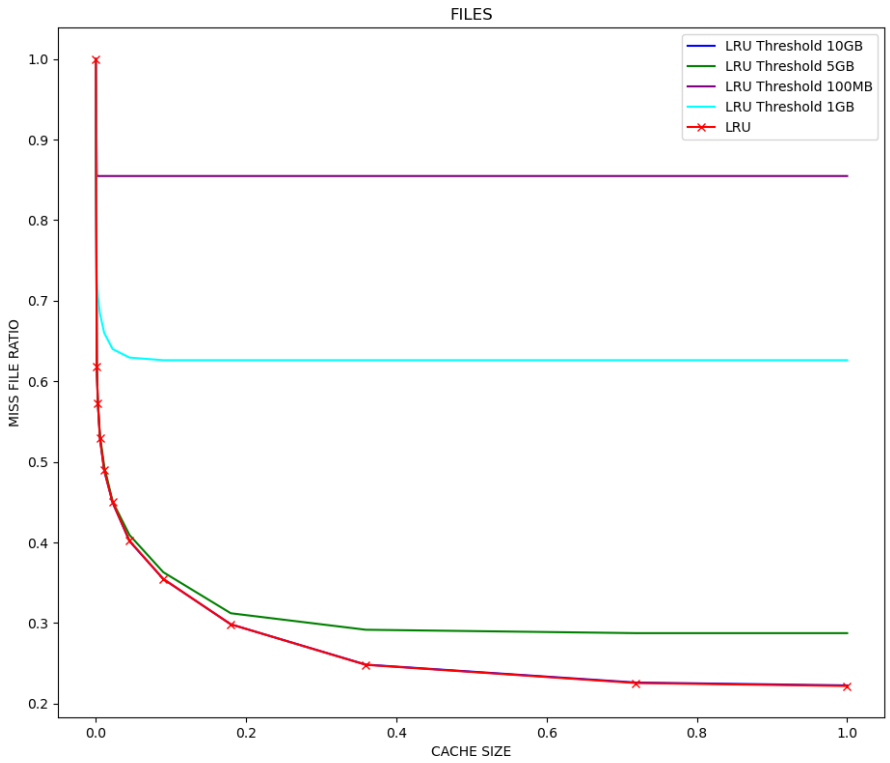


Figure 9

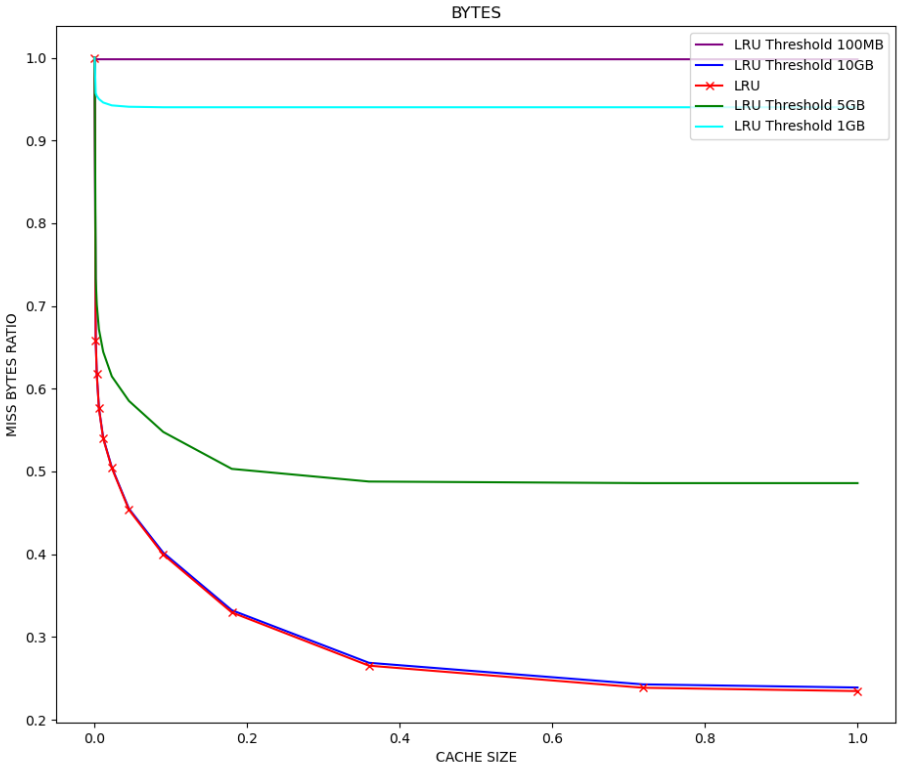


Figure 10

GDSF

Introduction

The second cache policy that we will discuss is called **Greedy Dual Size Frequency (GDSF)** and is based on one of the blueprints that are described in **Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy**. Amongst the several policies mentioned in this paper, we are most interested in is **GDSF**, as it combines the most important characteristics of a file when it comes to cache implementation: **size of the file**, **cost of retrieving the file** and **frequency** (the number of times a file that is in the cache has already been accessed).

Besides the original implementation of GDSF (which we are going to summarize in the next paragraph), we have also studied 2 slightly modified versions of it, thus in total we have 3 different implementation of the GDSF policy: **GDSF Base**, **GDSF Modified** and **GDSF Eviction Only**.

GDSF BASE

As the name would imply, GDSF Base is the original implementation of the GDSF policy, as it is described in the paper. This cache eviction and admission algorithm was designed to

deal with documents of variable size and variable retrieving costs for Web Proxy caching. Therefore, as it was already presented in the introduction paragraph, it takes into account file details such as: Size, Retrieving Cost and Frequency. Next, we will present how GDSF is implemented:

Let us consider a cache of size **Total**. Let **Used** be the amount of cache which is used to store cached files. At the beginning, Used = 0.

With each file **f** present in a cache we associate a frequency count **Fr(f)**: how many times it was accessed.

If a file **f** is not present in the cache, but is going to be cached, then it is assigned a frequency count of 1: Fr(f)=1.

To define which files are going to be replaced when the cache capacity is exceeded, we maintain a priority queue on files. The file **f** is inserted into the priority queue with a priority key **Pr(f)** computed in the following way:

$$Pr(f) = Clock + Fr(f) \times \frac{Cost(f)}{Size(f)} \qquad (1)$$

- **Clock** = represents an aging factor, which initially is equal to zero, but gets updated when there is an eviction of files. Usually it can be seen as the highest probability that has been removed so far with the exceptions of some cases (more on this matter when we present the rules of eviction)
- **Fr** = frequency of a file
- **Cost** = cost of retrieving a file form the main database
- **Size** = size of the file

Now, let us summarize the caching policy as a whole (for a more detailed description please see the paper).

1. ***If the requested file f is a hit in the cache then:*** this file request is served out of cache, the value of **Used** does not change, file frequency count **Fr(f)** is increased by one, the priority key **Pr(f)** is recomputed using formula (1). The old Pr(f) is deleted from the priority queue and recomputed **Pr(f)** is reinserted into the priority queue to reflect its new state.

2. If the requested file f is a cache miss then: this file request is served from the original server and may be cached by the cache on its way back from the original server to the client that requested this file. To do this, file frequency count **Fr(f)** is set to one, the priority key **Pr(f)** is computed using formula (1), file **f** is inserted into the priority queue with the computed key **Pr(f)** (even if the file would not fit into the cache database) and the amount of cache **Used** is reevaluated in the following way:

$$Used = Used + Size(f)$$

After that one of the following two situations takes place:

If **Used** < **Total** then this means that there is enough space to store the file **f**, and no files should be replaced. The file **f** is cached, and it completes this case.

If **Used > Total** then this means there is not enough space to store the file **f**, and some files should be replaced. Therefore, first of all, we identify a minimal set of files to evict using the following procedure: we parse the priority queue from the end (where the lowest priority files are stored) and add to a list all the files that have to be removed in order to reach a size of the cache lower than the maximum size (**Used < Total**). If the new file is among those files to be removed, we do not cache the new file. Otherwise, we remove all the files in the list and cache the new file. Furthermore, we only update the **Clock** variable in the case the new cache file **f** was **not** among those files which got evicted. In this case, the **Clock** variable takes the value of the highest priority that has been removed.

$$Clock = \max_{i=1}^{k} Pr(f_i) = Pr(f_k)$$

This completes the caching algorithm.

Different Variations of GDSF Base caching policy:

We also studied 2 different strategies for when to update the clock: the first one would consist of updating the clock just when the new file is evicted (not added to the cache) and, in the second strategy we update the clock each time an eviction has to be done (either the new file or other lower probability files are evicted).

Implementation

To test the caching policy described above, we implemented it using C++, choosing this programming language mainly due to the fast processing speed of the entire dataset (approx. 8 billions file requests). We implemented the cache using 2 data structures: an **unordered_map** and a **set**. In the map structure, we stored key-Value pairs that described file details. For the **Key** part we used the file ID, and for the **Value** part a structure in which details such as size, cost and frequency were saved. In the set structure, for every file, we stored a structure that was made of the file ID and file priority, and we ordered the set by the value of the priority field in every structure. We used the map for a fast insertion, erase and look-up times (~ O(Log N)) and the set for a fast insertion in a sorted structure (~ O(Log N)).

This caching policy (together with its variations) was applied to a dataset representing **3 month of file requests** (the files that were requested stored data from the ATLAS experiment at CERN). The input data for the caching policy is presented as a request trace obtained after the preprocessing of the original log files coming from the CERN Data Center, and contain the following characteristics of the files: **ID** and **Size**. You may observe that there is no cost for the files, that is because for this initial testing we will consider that the cost of retrieving a file is the same for all files, thus we consider **Cost = 1**.

Results

The results of the caching policy are shown in Figure 1 and Figure 2. We compare the performance of the GDSF Base variations with LRU's performance.

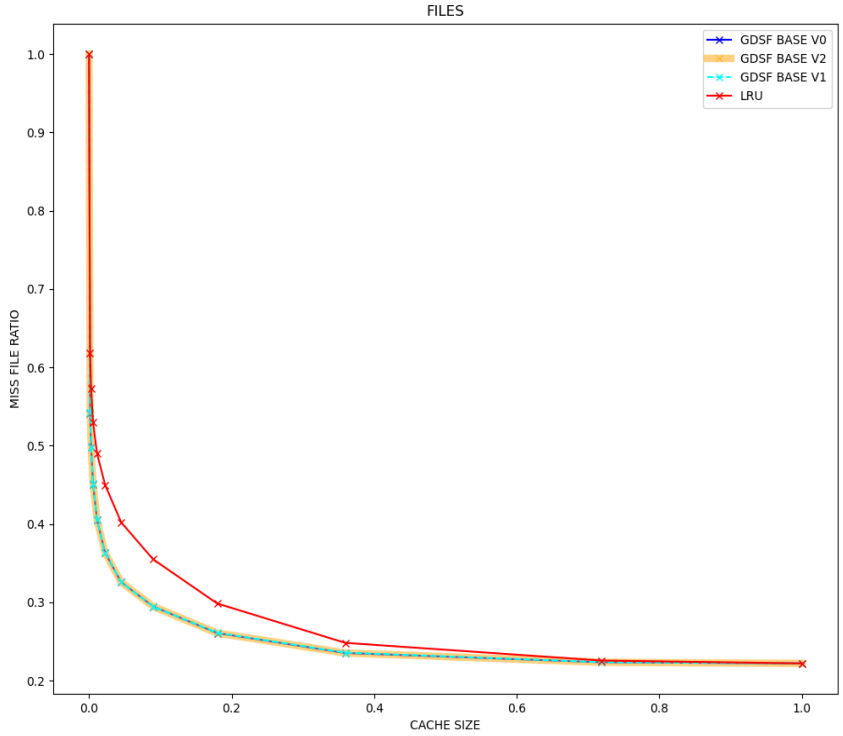


Figure 1

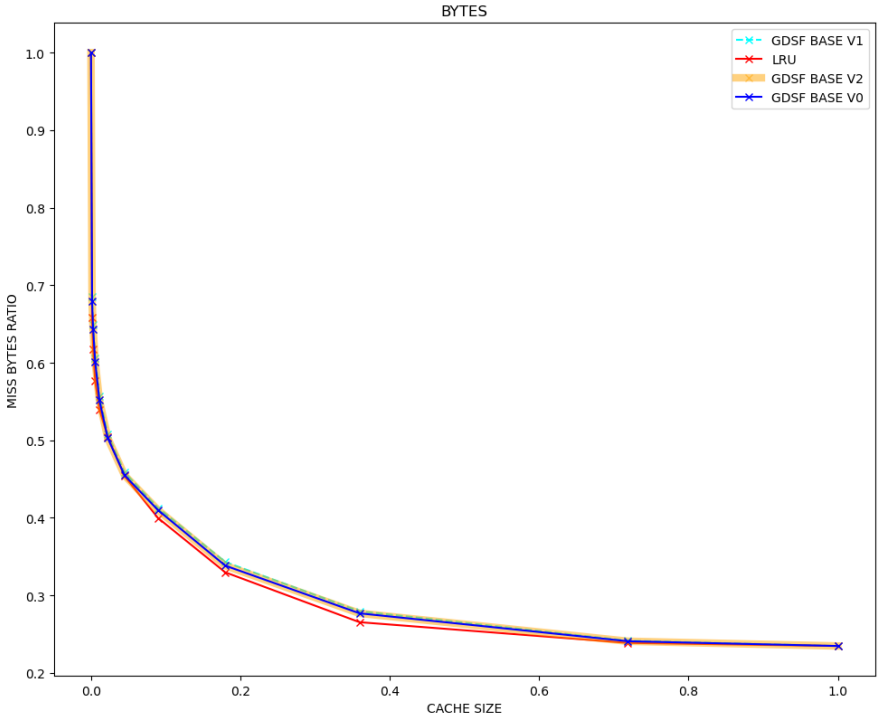


Figure 2

In the Figure 1 and Figure 2 we will refer to the GDSF Base and its variations in the following way:

- GDSF Base V0 = GDSF Base
- GDSF Base V1 = GDSF Base but the clock is updated when the new file is removed (in the opposite case of GDSF Base V0)
- GDSF Base V2 = GDSF Base but the clock is updated in both cases.

In **Figure 1** is presented the file miss ratio (FMR) and in **Figure 2**, the miss byte ratio (BMR). It is clear that all the variations of GDSF Base behave better than LRU regarding the OMR and worse regarding the BMR. As can probably be observed from the plots above, GDSF Base and its 2 variants are extremely similar to one another (the plots vary just with 10^-4). The best one of them is **GDSF Base V1**.

GDSF Modified

This version is derived from the GDSF Base algorithm by changing the way the Clock variable is calculated. For **GDSF Modified** the Clock variable does not take the value of the highest priority that was removed, but instead the Clock is increased by that value.

$$CLOCK \mathrel{+}= \max(Pr(f_i))$$

This modification is meant to make the caching policy more ‘’aggressive’’ regarding the eviction of the old files, thus making it more similar to LRU.

Different Variations of GDSF Modified caching policy:

Furthermore, as this cache policy is very similar to the GDSF Base, we also implemented 2 other strategies, which, as for the previous cache policy, change the case when the Clock is updated. The first one would consist of updating the clock just when the new file is evicted (not added to the cache) and, in the second strategy we update the clock each time an eviction has to be done (either the new file or other lower probability files are evicted).

Implementation

Regarding the implementation, we used the same technologies and data structures as for the previous policy.

Results

The results of the caching policy are shown in Figure 3 and Figure 4. We compare the performance of the GDSF Modified variations with LRU’s performance.

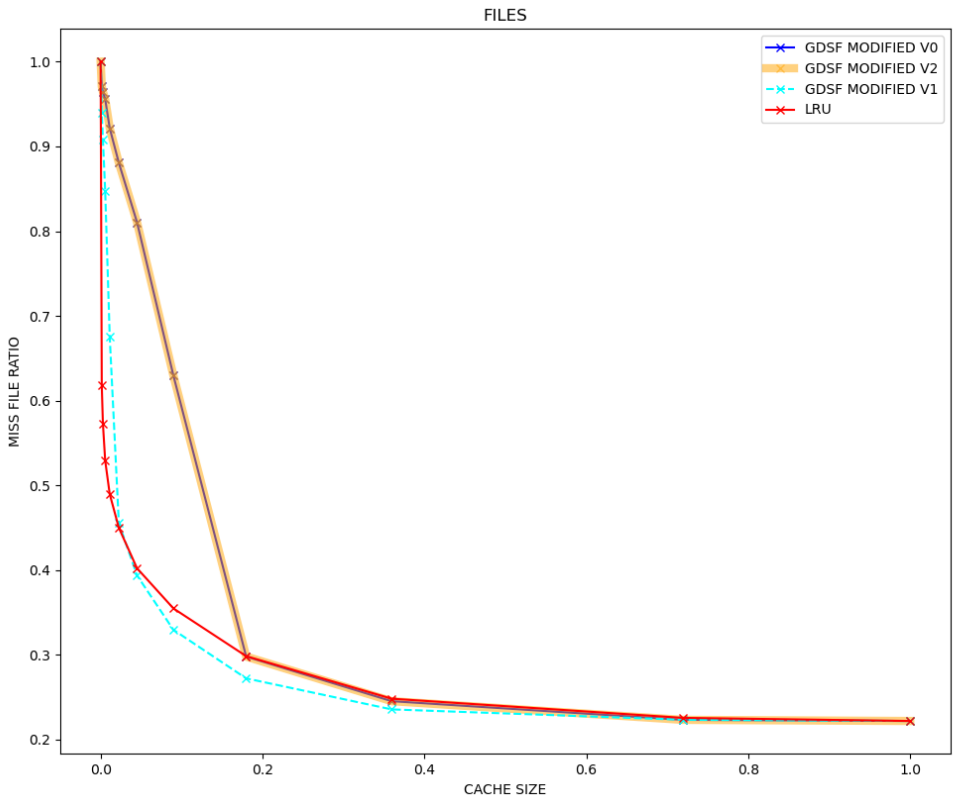

FILES
GDSF MODIFIED V0
GDSF MODIFIED V2
GDSF MODIFIED V1
LRU
MISS FILE RATIO
CACHE SIZE
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.0
0.2
0.4
0.6
0.8
1.0

Figure 3

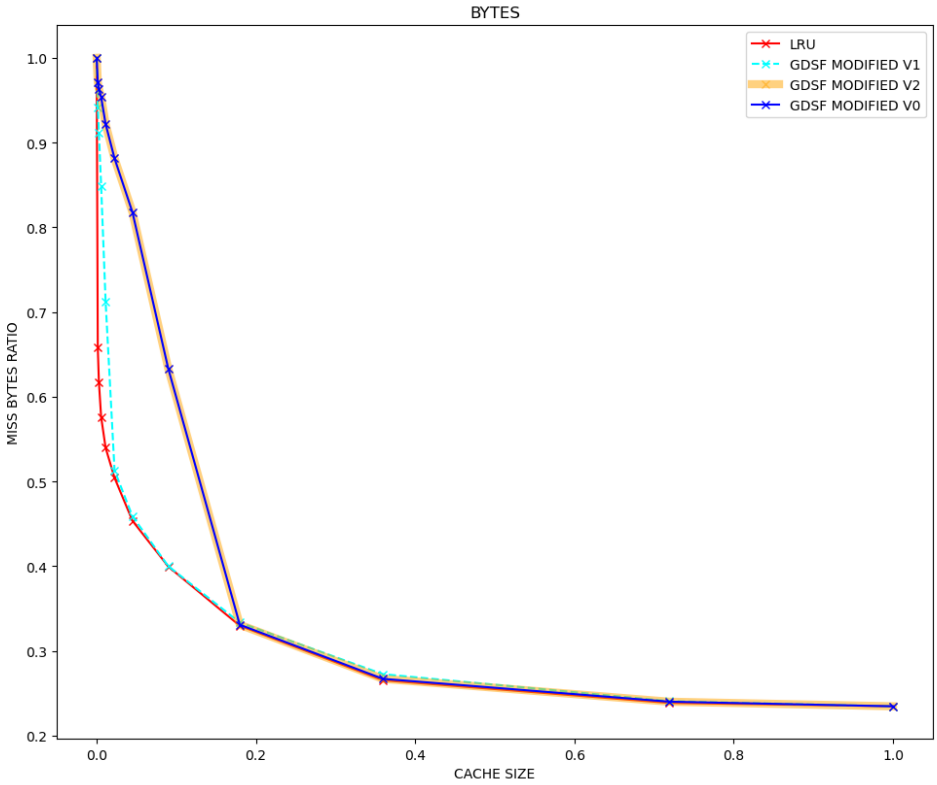


Figure 4

In the Figure 3 and Figure 4 we will refer to the GDSF Modified and its variations in the following way:

- GDSF Modified V0 = GDSF Modified
- GDSF Modified V1 = GDSF Modified but the clock is updated when the new file is removed (in the opposite case of GDSF Modified V0)
- GDSF Modified V2 = GDSF Modified but the clock is updated in both cases.

In **Figure 3** is presented the file miss ratio (FMR) and in **Figure 4**, the miss byte ratio (BMR). **GDSF Modified V1** behaves better than the other 2 variants of GDSF Modified regarding the FMR, and also for BMR, but just for cache sizes up to 20% of the total size of the database.

Concerning the performance difference between LRU and GDSF Modified V1, when it comes to FMR, the letter one has better results excluding the really small cache sizes. As for BMR, the GDSF version has slightly worse results.

GDSF Evict Only

Due to the specific usage of the cache that we described in the Introduction paragraph, in practice all the requested files will have to be admitted to the cache. Therefore, we are only interested in the pure eviction policies (the admission part of the policy is dismissed).

As both GDSF policies presented before include some sort of file admission rules, we have to modify the algorithm to accept all the files that are requested to the cache. As a result, we bring the following changes:

- When a file that is not in the cache is requested, we compute its probability, but we do not insert it yet.
- We look to see if there is enough space in the cache:
    - If there is enough space, the file is cached and also included in the priority queue.
    - If there is not enough space, we remove as many low priority files as we have to in order to make room for the new file. After this step is completed, the new file is cached and also included in the priority queue.

Implementation

Regarding the implementation, we used the same technologies and data structures as for the previous policies.

Results

The results of the caching policy are shown in Figure 3 and Figure 4. We compare the performance of the GDSF Evict Only with LRU's performance.

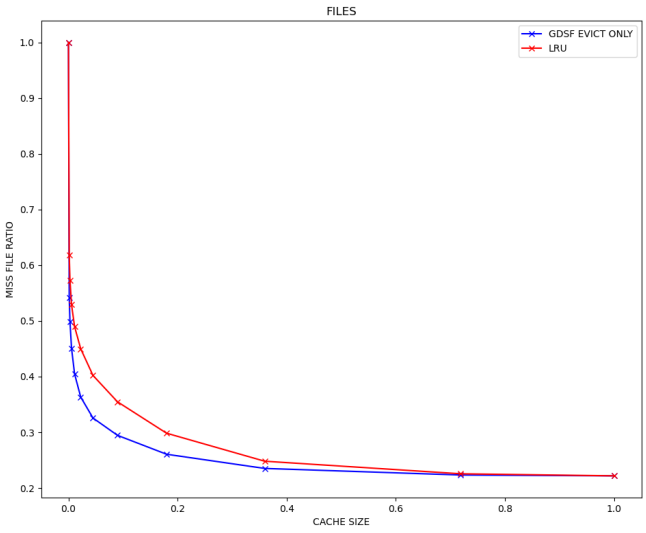


Figure 5

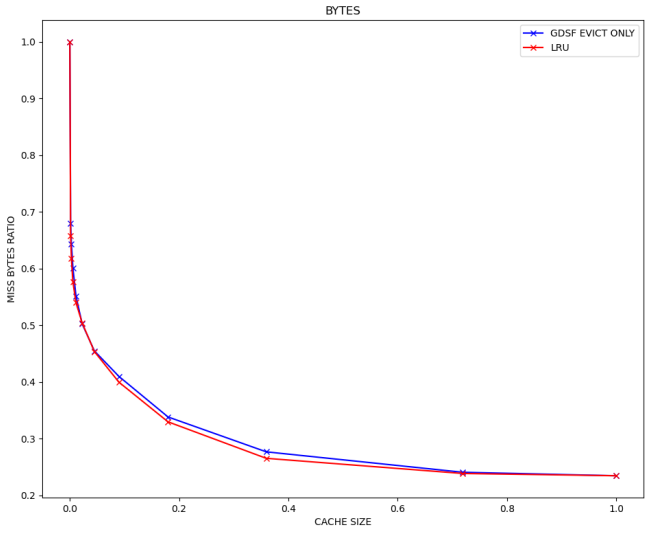


Figure 6

In **Figure 5** is presented the file miss ratio (FMR) and in **Figure 6**, the miss byte ratio (BMR). GDSF Evict Only behaves better than LRU when it comes to FMR, but rather worse regarding BMR.

Best Version of GDSF

Now we will plot the best version of all 3 GDSF caching policies, in order to observe which one behaves the best. The policies that will be plotted are: **GDSF Base V1**, **GDSF Modified V1**, **GDSF Only Evict** and **LRU**.

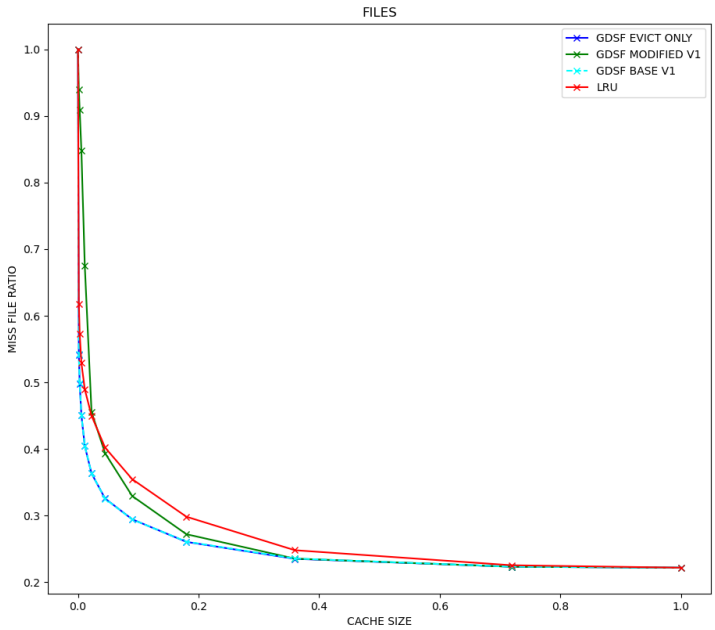


Figure 7

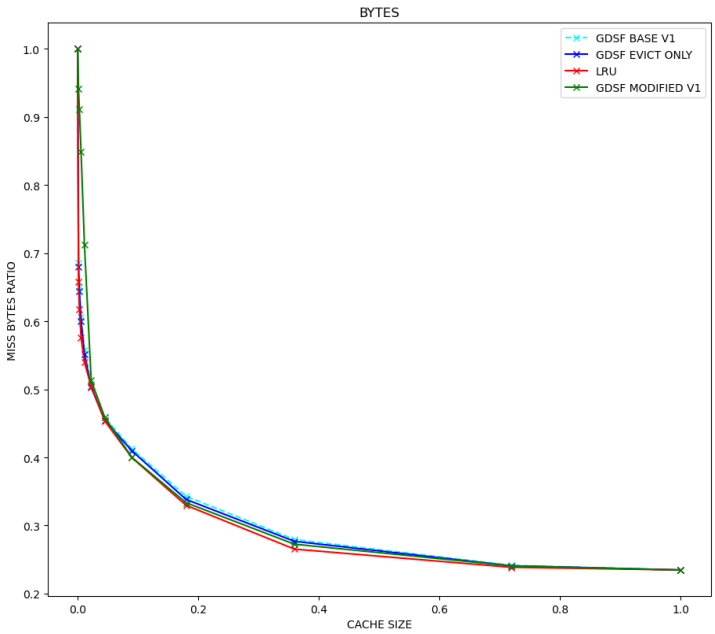


Figure 8

Concerning the FMR the best cache policy is GDSF Base V1 (closely followed by GDSF Evict Only). As for BMR, LRU is the best caching policy, but amog GDSF policies, the **GDSF Modified V1** is leading.

Machine Learning Cache Policy

Reasoning

As described in the Introduction Paragraph, the files that are requested by the processing units through WLCG have a large amount of details that can be used, such as: ID, size, acces time, dataset, file type, group of users, etc. A key observation, that can be deduced from the performance of the cache policies presented before, is that the more details an algorithm takes into account, usually the better the results are. Nevertheless, this idea only stands if the right relations between the data characteristics are considered.

Because a caching policy that is designed especially for the data requested through WLCG is hard to create, and even harder to find, a promising idea would be to train an ML model and use it inside a cache policy.

Cache policy based on ML model

Purpose

The main purpose of my work regarding the ML caching policy was to create the full ML pipeline. This consists of: **data farming**, **data processing and feature engineering**, **selecting a ML algorithm**, **selecting a ML model**, **training the model**, **building the cache policy** and finally **getting the results**.

Caching Policy

As presented in previous versions, once the cache gets full, we need a strategy for file evictions. Therefore, based on the information of the files in the cache and using a NN model, we compute the predicted number of times a file is to be accessed in a future time interval (in our case, the prediction is done for the next 10 days). Using this number, we order the cache by it, and remove the files with the lowest value until we reduce the size of the cache to the desired level.

Implementing the pipeline

Data farming

The data is already farmed, having a 3 month trace of file requests related to the ATLAS experiment. The input data is stored in a JSON file with the following fields: Id, Size, Request time.

Data processing

The entire process is based on 2 time intervals:

- A separation time interval **T**, which is used to divide the processing of the request trace into steps
- A future prediction time interval **TF**, which is use to determine how long into the future we will try to predict (10 days)

We read through the trace until the timestamp of the newest requested file is equal or greater than the current n*T (n is a natural number, initially 0 and it is incremented by one with every step).

Once a request of a file has been read, we save its data in a metadata structure.

The data we save is the following:

- ID
- Size
- First acces of the file (this will always be the same)
- The last acces of the file (initially is equal to the first acces of the file)
- Number of requests so far (initially is set to 0).
- Number of future accesses in an already defined future time interval FT (initially zero)

If the file appears multiple times in this interval [0, n*T], we update its last access to the latest one and increment the number of requests by one for every new appearance.

After we complete this step, we go back to reading the trace up until we reach a timestamp equal or greater than n*T + TF. While we read the requests, if any of the files requested in this ‘’future’’ time interval is already in the metadata, we increase the number of future requests by one for that particular file.

Every time we finish this step, we again process (feature engineer) the data from the metadata structure and print it into a file.

Feature engineering the data:

For every file in the metadata structure we build the following set of data:

- Size
- Difference between n*T and first access
- Difference between n*T and last access
- Number of access so far
- Number of accesses in a future time interval (TF)

We take this data and print it into a JSON file. The process is repeated by increasing the value of ‘’n’’ until n*T + FT <= last timestamp.

Algorithm Selection

The learning process is a supervised one, based on regression with a neural network. We chose to implement the pipeline with AutoKeras, a Tensorflow API, which has proven extremely helpful in finding the right algorithm. We separated the data into sets: a training set (60% of the total processed data) and testing set (30% of the total processed data). AutoKeras helps us in finding the best algorithm just by giving the type of the model and the training set. For this to work, we have to specify the percentage of the training set that is going to be used for testing different models. This part of the training data is called the Validation Set and it represents 30% of the training set.

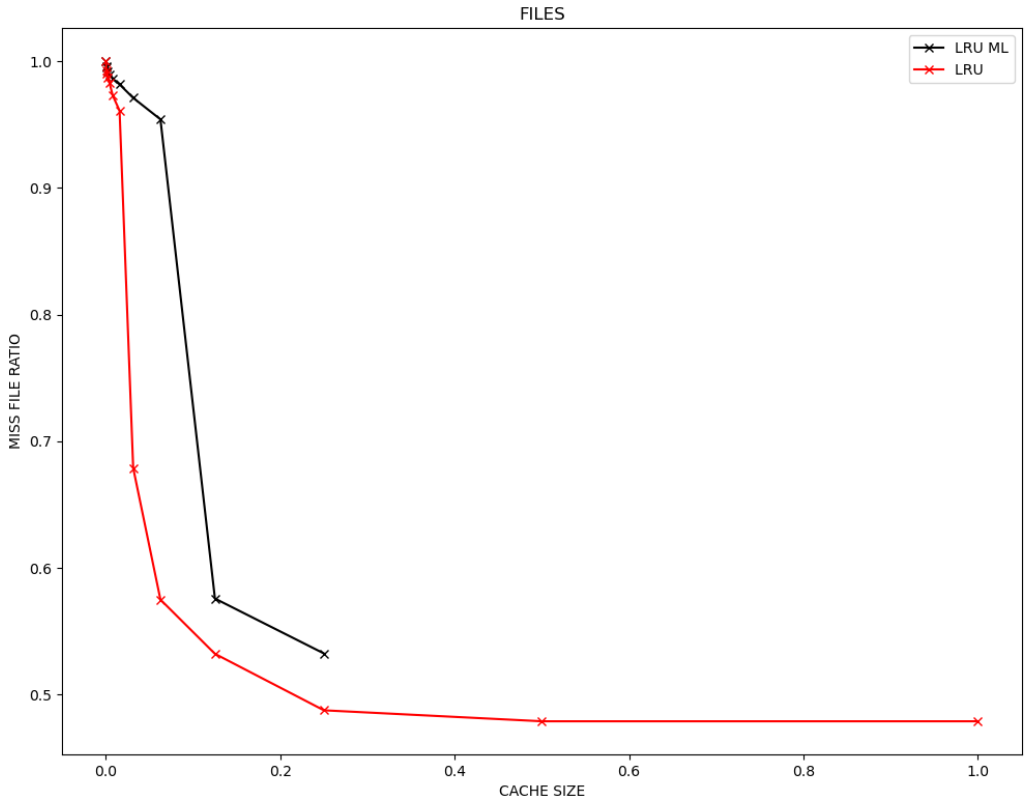
We tested 10 different models and the best one was represented by a NN with 5 layers, one for normalization, and the other 4 for processing, each of them having between 1000 and 2000 neurons.

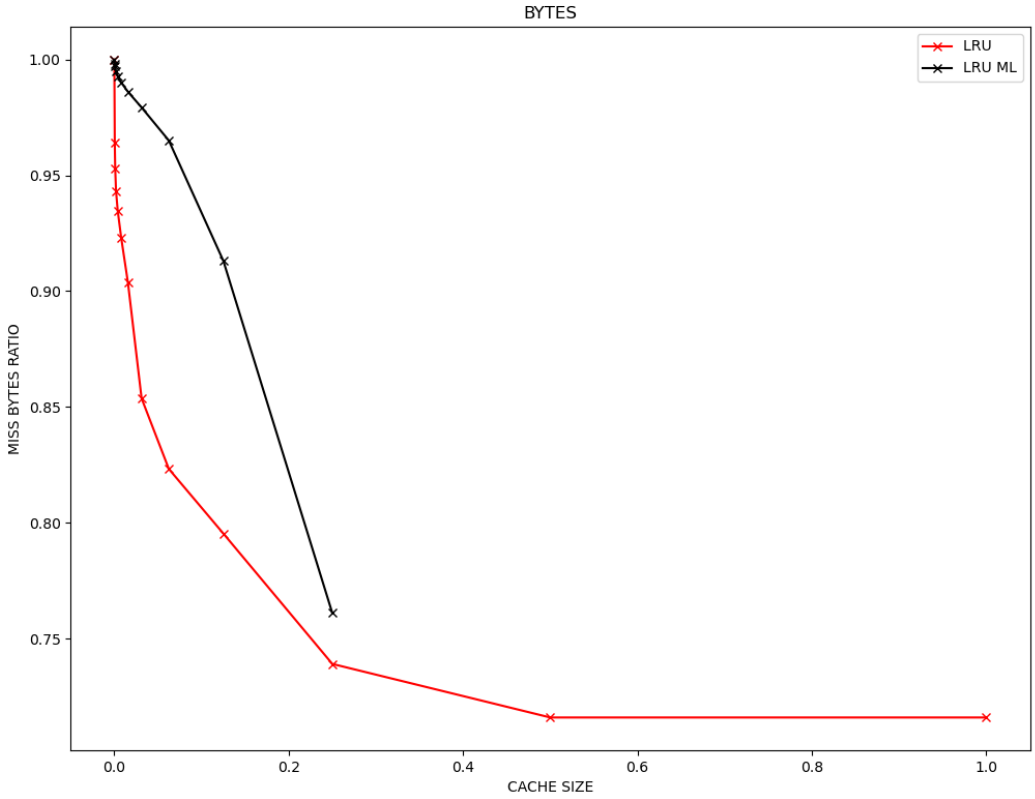
Encountered Problems

As processed data has a large size (approx. 13 GB), it was not possible to use all of it to train the model, because the size of the RAM memory of the GPU was not big enough to store it. The only solution was to split the data in multiple pieces and train the model sequentially with all of them. A point worth mentioning here is that the model selection was only done using the first piece of data, therefore it may not be the most optimal model.

Once the model was trained, the next step was to use it inside the cache policy, which was planned to be implemented in C++. Unfortunately, transferring the model from Python to C++ requires a lot of time, as CAPI has to be used. Because it is known how time consuming this would take, and as the end of my summer program was approaching, we took the decision to implement the caching policy in Python. As a result, testing the cache policy was a really slow process. Thus, we only tested it using 300000 requests out of the total 8 billions present in the trace. Also, we only tested the cache just for sizes up to 25 % of the total size required to store all the requested files

Results





As it can be seen, the LRU ML behaves worse than normal LRU. Considering that the purpose of this ML approach was rather to implement the full pipeline and not to outperform the previous caching policies, and also that there are many ways to improve the implementation that I have done so far, LRU ML seems at least promising for future development.

Conclusions

During my summer student program I successfully implemented and tested 3 size aware caching policies: LRU Threshold, GDSF and ML Caching Policy. Out of these 3 the best results were provided by the GDSF algorithm. This policy also represents a possible solution for future remote processing sites from WLCG, due to its no selective admission strategy. The ML algorithm, even though it does not perform better than LRU, can be improved in many ways, thus being a solid option for future research.

References

[1] Caching for dataset-based workloads with heterogeneous file sizes

[2] AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network

[3] Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy