Machine Learning Algorithms for Caching Systems

Online Learning for Caching with Heterogeneous miss-costs

Robert Valentin Vadastreanu

Supervisor(s): Georgios Iosifidis, Naram Mhaisen, Fatih Aslan

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 23, 2024

Name of the student: Robert Valentin Vadastreanu
Final project course: CSE3000 Research Project
Thesis committee: Georgios Iosifidis, Naram Mhaisen, Fatih Aslan, Neil Yorke-Smith

Abstract

This paper presents an adaptive per-file caching policy designed to dynamically adjust caching decisions based on the importance of the requested files. It relies on the Online Gradient Ascent (OGA) algorithm, which treats the caching problem as an online optimization problem. This methodology ensures minimal regret by continuously optimizing caching configurations in response to real-time request sequences. The caching configurations are optimized after every request using a constant learning rate. Due to the fact that the trends of requested files can change, we will introduce two new algorithms that change the learning rate at every request to increase the adaptability. We will present two new algorithms, the Universally Adaptive Caching (UAC) algorithm and the Adaptive Per File Caching Algorithm (APFC), and we will present scenarios to highlight their performances.

1 Introduction

Caching is a universal strategy to improve efficiency by keeping frequently needed items easily accessible. This principle can be applied to various fields in computer science, such as databases, but also outside the field, such as library management or household organization. In computer science, caching[1] is a technique used to store data that is requested frequently in a temporary storage area to improve the system's performance. The caching memory is a small, high-speed storage area close to the CPU. Because of its size, an efficient caching policy is trivial for a system's performance.

A caching policy is a set of rules determining which data should be stored in the cache memory at every time slot. Its objective is to maximize the number of cache hits to minimize latency. *A cache hit* occurs when the requested data is found in the cache memory, allowing faster access than retrieving it from the main memory. *A cache miss* is the opposite situation when the requested data is not stored in the cache memory. An illustration of how cache works can be seen in Figure 1.

The caching problem, first identified in the 1950s, continues to pose significant challenges in modern computing. The increasing volume of data and the evolving patterns of data requests necessitate the development of new techniques and strategies.

Some classical caching methods [2] are Least Recently Used (LRU), which removes the least recently used items from the cache when it is full, First In First Out (FIFO), in which the oldest items are removed when the cache reaches its maximum capacity, and Least Frequently Used (LFU) in which the least frequently used file is removed whenever the cache become full. These previous methods follow a static set of rules regardless of the request pattern and do not adapt to the different importance of the files, which can depend on the distance to the server storing that file or on traffic. Considering this, the caching problem started to be cast as an online linear optimization [5]. A new policy called Online Gradient Ascend (OGA) was introduced that adapts to the trends and the patterns of the requests with different importance, using a constant learning rate. Even if the online learning methods are generally pessimistic because they measure performance in an adversarial way, the performance is better in practice.

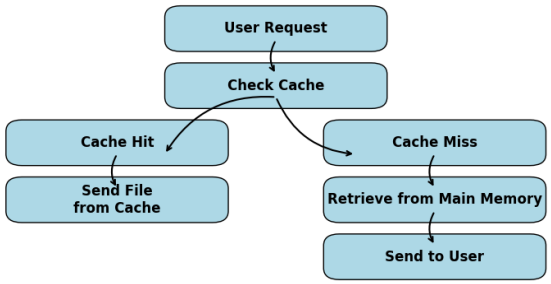

Figure 1: Caching diagram

The OGA adjusts the next caching configuration based on the direction of the gradient. It uses a step size η, representing the change made at every step. This variable is fixed and depends on the upper bound of the importance of the files. This paper will provide a scenario where the OGA algorithm will perform poorly because of its step size. This makes use of a high variance of the importance of the files.

As proved in [5], the OGA algorithm has optimal performance compared with the classical algorithms, but its performance is inversely proportional to the importance of the files. The performance is high when the requests have a small variance in importance. Still, the situation changes when we introduce a high variance for the files' importance because the algorithm's step size will be very small so that it will converge very slowly. This led to the realization that this current algorithm is insufficient for handling scenarios with varying importance for the files. Because of that, we need to design another algorithm that addresses this issue and adapts its step size according to the importance of the files, even if those can change dramatically over time.

This paper introduces two upgrades of this online caching method. Firstly, we will introduce an algorithm that changes the step size at every step, solving the previous scenario. In this approach, we will show that

[1] Caching explained: https://aws.amazon.com/caching/

[2] https://www.gem5.org/documentation/general_docs/memory_system/replacement_policies/

the policy's performance will no longer depend on the upper bound of the files' importance. Secondly, the step size will become independent for each file, depending on each file's importance in the system. This will improve the first version of our algorithm, so at every step, the step size will rely only on the requested file itself, not on all the previous requests.

The following article is organized as follows: section 2 analyzes other caching policies' performances based on other research made. Section 3 introduces the system model, followed by the problem statement. Section 4 explains the universally adaptive caching algorithm and the adaptive per-file caching algorithm. The following section presents some experiments of those algorithms with some plots. Section 6 presents some ethical aspects of this research. In the last section, the result of this experiment will be concluded.

2 Related work

2.1 Classical Caching Policies

As also presented in the introduction, there are some classical caching policies such as LRU, LFU, and FIFO. This section will present related work that analyzes those policies' performances, along with the Online Gradient Ascent Algorithm (OGA). Due to its limited size, the performance of a cache policy needs to be adequately measured. In this section, we will use a metric called cache hit ratio, which represents the percentage of hits made during the request. In [4], it is shown that LRU policy achieves a high hit ratio for some request patterns that favor items to be re-requested shortly after their initial access. This happens because the LRU policy ensures that these recently accessed items remain in the cache. This optimal performance can be observed in the simulations where the user access patterns follow a Zipf distribution. Examples of requests distributed according to Zipf's law are web pages [2] and YouTube videos [6]. Also, in [4], it is shown that under a stationary request pattern, LFU outperforms LRU. There are also some optimizations of the LRU, such as multi-LRU [7], which has higher performances than the classical LRU.

In the comparison between LRU and FIFO, there were significant changes. Initially, LRU was considered better than FIFO under the independent reference mode as written in [14]. After some changes in the modern request patterns and trends, a revision of this paper appeared [3], which claims that FIFO outperforms LRU.

After this analysis, we can conclude that classical algorithms have some strengths in specific request patterns. However, they do not adapt to changing trends because they follow a strict set of rules, so they do not achieve high performance on all request patterns.

2.2 Caching as an Online Algorithm

The limitation presented in the previous subsection gave the motivation to model caching as an online learning problem [12], [15]. The caching was studied as an online algorithm in several ways, such as an online gradient ascent (OGA) caching policy in [5], as an online mirror ascent [13], or using sub-modular policies [8]. There are other approaches to the caching problem as an online problem, such as in [10], where the problem is solved using a recommender system.

Regarding the aspect of adaptability, the OGA Algorithm solves this weakness. It is proved that it has optimal performance [5]. The OGA Algorithm outperforms LRU and LFU by 20% in some scenarios. In the other scenarios, its performance stays close to the best among them. This is the basis of our new adaptive caching algorithm.

The OGA Algorithm is an online problem solved with a constant step size. In [11], it is shown that online learning with fixed steps is less efficient than online learning with adaptive steps. This paper will treat caching as an online problem that will be solved with adaptive steps.

3 System model and problem statement

System model

In a caching system, we need to decide what files to store in the high-speed caching memory close to the CPU. In our model, we consider a finite set of files $\mathcal{F} = \{1, 2, \ldots, N\}$, of length N and cache that can store at most C files, $C \leq N$.

A request is made at every time slot. It will be encoded as a one-hot vector of length N. If $\mathbf{x}_t^k = 1$, then at time t, the kth file is requested. Every request has the form:

$$\mathbf{x}_t = (x_t^i \mid i \in \mathcal{F} \text{ and } x_t^i \in \{0, 1\})$$

All requests are chosen from the set:

$$\boldsymbol{\mathcal{X}} = \{\mathbf{x} \in \{0, 1\}^N \mid \sum_{i=1}^{N} x^i = 1\}$$

At every time slot, we need to decide our caching configuration. We will encode a caching configuration as a vector of length N with the sum of elements at most N. Every element in the vector represents the amount of the file to be stored in the cache memory at time t. It has the form:

$$\mathbf{y}_t = (y_t^i \mid i \in \mathcal{F} \text{ and } y_t^i \in [0, 1])$$

All the caching configurations are chosen from the set:

$$\boldsymbol{\mathcal{Y}} = \{\mathbf{y} \in [0, 1]^N \mid \sum_{i=1}^{N} y^i \leq C\}$$

A cache hit happens when the requested file is already stored in the cache. In the opposite case, the requested file is not stored in the cache, so we have a cache miss. A cache hit at time t respects the property $\mathbf{x}_t \mathbf{y}_t \neq \mathbf{0}$. The cache miss at time t respects $\mathbf{x}_t \mathbf{y}_t = \mathbf{0}$.

The importance of the files at time t is encoded in the weights vector $\mathbf{w}_t$ of length N, where every position represents how important a file is at time t. This is the form of the weights vector:

$$\mathbf{w}_t = (w_t^i | i \in \mathcal{F} \text{ and } w_t^i > 0)$$

We define the weighted request as being the element-wise product between the request vector and the weights vector:

$$\mathbf{g}_t = \mathbf{w}_t \odot \mathbf{x}_t$$

The utility function represents how efficient the caching configuration was at time t. It is defined as:

$$f(\mathbf{x}_t, \mathbf{y}_t) = \sum_{i=1}^{N} w_t^i x_t^i y_t^i$$

Consider that we have T requests, $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_T)$ and our caching policy $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, ...\mathbf{y}_T)$. The following function measures the utility of this caching policy over these requests:

$$U(\mathbf{X}, \mathbf{Y}) = \sum_{t=1}^{T} f(\mathbf{x}_t, \mathbf{y}_t)$$

We define a static caching policy as a policy that does not change the caching configuration over time. Every time slot uses the same caching configuration. The static policy that uses $\mathbf{y}$ as caching configuration is:

$$\mathbf{C}(\mathbf{y}) = (\mathbf{y}_1, \mathbf{y}_2, ...\mathbf{y}_T),\ \mathbf{y}_i = \mathbf{y}, \forall i \in \{1, 2, ...\ T\}$$

The best static caching policy $\mathbf{Y}^\star = C(\mathbf{y}^\star)$ is the static policy that achieves the highest utility until the horizon T. The argument is defined as the following:

$$\mathbf{y}^\star = \arg\max_{\mathbf{y} \in \boldsymbol{\mathcal{Y}}} U(\mathbf{X}, \mathbf{C}(\mathbf{y}))$$

The performance of our caching policy will be measured in a metric called the regret, which compares the policy with the best static policy. It is defined as:

$$R_T(\mathbf{X}, \mathbf{Y}) = U(\mathbf{X}, \mathbf{Y}^*) - U(\mathbf{X}, \mathbf{Y})$$

Problem statement

This paper aims to introduce new caching algorithms based on the OGA algorithm. In the following, we will explain how the OGA algorithm works and its limitations, as demonstrated in [5].

Because we cannot influence the incoming requests, we will redefine the utility function with only one parameter, the vector $\mathbf{y}_t$. This also applies to the regret function:

$$f(\mathbf{y}_t) = \sum_{i=1}^{N} w_t^i x_t^i y_t^i$$

$$R_T(\mathbf{Y}) = U(\mathbf{X}, \mathbf{Y}^*) - U(\mathbf{X}, \mathbf{Y})$$

The OGA updates the caching policy with the formula:

$$\mathbf{y}_{t+1} = \Pi_{\mathcal{Y}}(\mathbf{y}_t + \eta_t \nabla \mathbf{f}_t)$$

where η_t is the learning rate, and $\Pi_{\mathcal{Y}}(\mathbf{z}) = \arg\max_{\mathbf{y} \in \mathcal{Y}} \|\mathbf{z} - \mathbf{y}\|$ represents the projection of the vector $\mathbf{z}$ onto $\mathcal{Y}$, and $\nabla \mathbf{f}_t = (\frac{\partial \mathbf{f}_t}{\partial y_t^i}, \forall i \in \{1, 2, ..., N\}) = \mathbf{w}_t \odot \mathbf{x}_t$, where $\odot$ represents the element-wise product between 2 vectors. An example of a projection algorithm that can be used in this scenario can be found in [5].

The diameter of a set of vectors is defined as the maximum distance between two vectors in that set. In our case, in the space of all possible caching configurations, the diameter is:

$$\text{diam}(\mathbf{Y}) = \begin{cases} \sqrt{2C} & \text{if } 0 < C \leq \frac{N}{2}, \\ \sqrt{2(N-C)} & \text{if } \frac{N}{2} < C \leq N. \end{cases}$$

The learning rate of the OGA algorithm is defined as:

$$\eta_t = \frac{\text{diam}(\mathbf{Y})}{L\sqrt{T}}, \forall t \in \{1, 2...T\}$$

,

where L represents the upper bound of the norm of the gradient: $\|\nabla \mathbf{f}_t\|$

As proved in [5, Theorem 2], the OGA algorithm's regret has an upper bound of:

$$R_T(\boldsymbol{OGA}) \leq diam(\mathbf{Y}) L\sqrt{T}$$

As we can observe, L is a multiplication factor in the upper bound of the regret. This means that if we use a pattern where some files have a big variance in weight, the regret will be big, meaning that the policy performance will be low. This is happening because $\frac{1}{L}$ is a factor in the learning rate. The learning rate will be very low for L being significantly bigger than most of the requests' weight, implying that the OGA will learn very slowly only because there is a possibility of requesting a file of high importance.

This paper aims to introduce a per-file weighted cache algorithm that obtains a smaller upper bound for the regret because it can converge faster. Firstly, the paper will introduce an intermediate algorithm that adapts the learning rate at every time slot and performs better than the OGA in the pattern presented earlier. This intermediate algorithm can be improved to have a unique learning rate per file so that the gradient updates independently for each file. These two algorithms will solve the small adaptability of the OGA algorithm in some specific request patterns.

4 Adaptive Caching Algorithms

This section proposes two new online caching algorithms, which adapt their learning rates to the request patterns. We will show that in experiments, those methods lead to better performance than using the constant learning rate. The first subsection presents the Universally Adaptive Caching Algorithm, while the second presents the Adaptive per-file Caching Algorithm.

4.1 Universally Adaptive Caching Algorithm(UAC)

The Universally Adaptive Caching Algorithm updates the learning rate based on the weights at every request. The improvement from the OGA algorithm is that it is not dependent on the upper bound for the weighted request. For our algorithm, we will set the learning rate:

$$\eta_t = \frac{\sqrt{2}\text{diam}(\mathbf{Y})}{2\sqrt{\sum_{i=1}^{t} \|\mathbf{g}_i\|^2}}$$

It is proved in [9, Eq. 15] that for solving an online learning algorithm, using the previous learning rate which is adapted to our caching problem, we obtain the bounds:

$$R_T(\boldsymbol{UAC}) \leq \sqrt{2}diam(\mathbf{Y})\sqrt{\sum_{t=1}^{T} \|\mathbf{g}_t\|^2}$$

Now, we will compare the upper bound of the regret of the OGA algorithm with the upper bound of the regret of the UAC algorithm. We will start with the fact that L is the upper bound for the norm of a weighted request:

$$\|\mathbf{g}_t\| \leq L \ \forall t \in \{1, 2, ...T\}$$

If we raise to the square, we obtain:

$$\|\mathbf{g}_t\|^2 \leq L^2 \ \forall t \in \{1, 2, ...T\}$$

We can sum this inequality for all $t \in \{1, 2, ...T\}$ and obtain:

$$\sum_{t=1}^{T} \|\mathbf{g}_t\|^2 \leq L^2 T$$

Now, we can get the square root and multiply with $\sqrt{2}diam(\mathbf{Y})$ both sides:

$$\sqrt{2}diam(\mathbf{Y})\sqrt{\sum_{t=1}^{T} \|\mathbf{g}_t\|^2} \leq \sqrt{2}diam(\mathbf{Y})L\sqrt{T}$$

We can observe that the left term represents the regret of the UAC algorithm, while the right contains the regret of the OGA algorithm:

$$R_T(\boldsymbol{UAC}) \leq \sqrt{2}R_T(\boldsymbol{OGA})$$

In this case, we approximated all the weights to the upper bound L. In a real-world scenario, this approximation is too harsh, so the regret of the UAC algorithm will stay lower than the regret of the OGA algorithm. In the evaluation section, we compare these two methods. A pseudocode for the UAC algorithm, which handles a stream of requests, is displayed as Algorithm 1.

Algorithm 1 Universally Adaptive Caching Algorithm (UAC)

```
y_1 ← 0                                   // Initialize caching policy
G_0 ← 0                                   // Initialize weighted sum
D ← Diameter                              // Calculate the diameter of Y
for t = 1, 2 ... do
    g_t ← w_t ⊙ x_t                       // Calculate weighted request using element-wise product
    G_t ← G_{t-1} + ||g_t||^2             // Update weighted sum
    η_t ← D√2 / (2√G_t)                   // Get Learning Rate
    y_{t+1} ← Π_Y(y_t + η_t g_t)
end for
```

4.2 Adaptive per file Caching Algorithm(APFC)

As we can observe in the previous subsection, the UAC's learning rate becomes smaller and smaller from time slot to time slot. This is happening because:

$$\eta_t \geq \eta_{t+1}, \forall t \in \{1, 2, ...T-1\}$$

For the proof, this is equivalent to:

$$\frac{\sqrt{2}\text{diam}(\mathbf{Y})}{2\sqrt{\sum_{i=1}^{t} \|\mathbf{g}_t\|^2}} \geq \frac{\sqrt{2}\text{diam}(\mathbf{Y})}{2\sqrt{\sum_{i=1}^{t+1} \|\mathbf{g}_t\|^2}}, \ \forall t \in \{1, 2, ...T-1\}$$

We can perform some basic calculations and obtain the following:

$$\sqrt{\sum_{i=1}^{t} \|\mathbf{g}_t\|^2} \leq \sqrt{\sum_{i=1}^{t+1} \|\mathbf{g}_t\|^2}, \ \forall t \in \{1, 2, ...T-1\}$$

After we raise to the square and subtract, we obtain:

$$\|\mathbf{g}_{t+1}\|^2 \geq 0, \ \forall t \in \{1, 2, ...T-1\}$$

Because the last step is always true, we demonstrated that the learning rate becomes smaller from time slot to time slot. So, it can become very low in an advanced part of a run of the algorithm. This implies that updates to the caching configurations will become significantly low in some situations. An example is a change of trends: for example, in the first period, there are some popular files, and over a few time slots, their popularity decreases, and there will be a new trend. In this case, the transition will be very slow. Using this intuition, we can improve the UAC algorithm by using learning rates per individual file.

In [11], the performance of the AdaGrad algorithm is presented, which is an Online Convex Optimization algorithm. It uses a learning rate for each dimension of each of the coordinates of a 1-dimensional vector. We will use this Online Convex Optimization algorithm in our caching problem to use a learning rate at time t for file i in order to create the *Adaptive Per File Caching Algorithm (APFC)*:

$$\eta_t^i = \frac{\sqrt{2}diam(\mathbf{Y})}{2\sqrt{\sum_{j=1}^{t}(w_j^i x_j^i)^2}}, \forall i \in 1, 2, ...N$$

If we plug in the values corresponding to our caching problems, considering that the diameter is 1 for each file space, in [11, Theorem 4.26], it is demonstrated that the following regret bound holds:

$$R_T(\boldsymbol{APFC}) \leq \sqrt{2}\sum_{i=1}^{N}\sqrt{\sum_{j=1}^{T}(w_j^i x_j^i)^2}$$

In [11], it is also demonstrated that using this learning rate, we will achieve a lower regret bound than setting the learning rate from the UAC algorithm. The evaluation section will compare these two algorithms in different scenarios. A pseudocode is depicted in Algorithm 2 below, which presents an application of the APFC algorithm over a stream of requests alongside the weights vector.

Algorithm 2 Adaptive Per File Caching Algorithm (APFC)

1: $\mathbf{y}_1 \leftarrow \mathbf{0}$ // Initialize caching policy
2: $\mathbf{G}_0 \leftarrow \mathbf{0}$ // Initialize weighted sum
3: $D \leftarrow$ Diameter // Calculate the diameter of $\mathcal{Y}$
4: **for** t = 1, 2 ... **do**
5: $\mathbf{g}_t \leftarrow \mathbf{x}_t \odot \mathbf{w}_t$ // Calculate the weighted request using element-wise product
6: $\mathbf{G}_t \leftarrow \mathbf{G}_{t-1} + \mathbf{g}_t \odot \mathbf{g}_t$ // Update weighted sum vector using element-wise product
7: $\eta_t^i \leftarrow \frac{D\sqrt{2}}{2\sqrt{\sum_{i=1}^{N} G_t^i x_t^i}}$ // Calculate Learning Rate for the requested file
8: $\mathbf{y}_{t+1} \leftarrow \Pi_{\mathcal{Y}}(\mathbf{y}_t + \eta_t^i \mathbf{g}_t)$
9: **end for**

5 Evaluation

This section compares the algorithms described in the paper. In the first subsection, we will highlight the differences in practice between the OGA and UAC using the MovieLens dataset. The second subsection will show a scenario where APFC's performance is better than UAC's.

5.1 OGA vs UAC

The MovieLens[3] data set was used for both figures for the simulation. In these simulations, we used the first 10000 reviews in chronological order from the MovieLens, where the requested file is the reviewed movie. The simulations from both figures use a total of $N = 497$ files and a fixed cache size of $C = 250$.

In Figure 2, we perform the simulation without considering the weight vector. To achieve this, we set it constant as a vector of ones. In this case, the learning rate of OGA will use the term $L = 1$. Considering that we know the horizon's value $T = 10000$, the number of files, and the caching size, we can also calculate the learning rate as $\eta \approx 0.222$. In this case, we can observe that the UAC and APFC perform similarly, while the OGA has a higher regret. This is happening because, in the early running stage, the UAC and APFC have high learning rates that decrease over time. The OGA uses a constant learning rate, updating the caching configuration more slowly.

[3]https://grouplens.org/datasets/movielens/

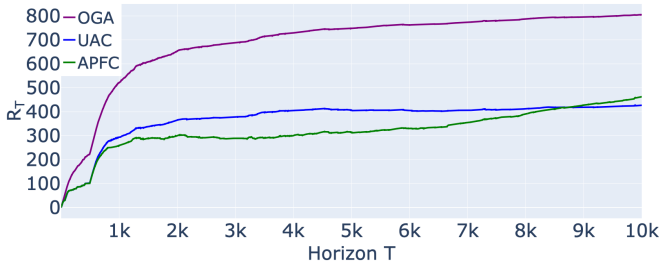

Figure 2: Unweighted simulation

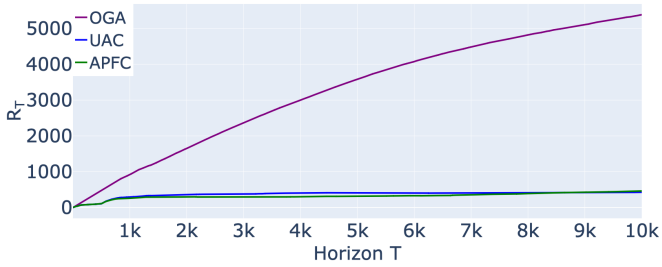

Figure 3: Weighted simulation

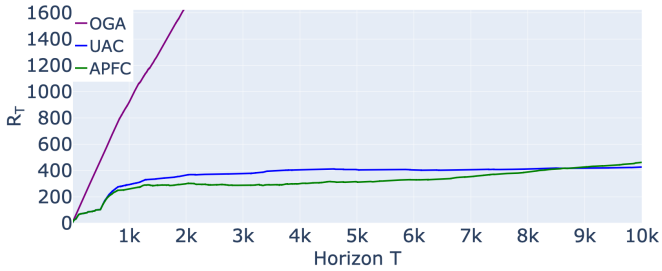

Figure 4: Weighted simulation zoomed

In Figure 3, the performance difference between OGA and these two algorithms becomes more evident by introducing weights. We will use a convention for the files' weight to be a minimum of 1 and a maximum of 15. This implies the upper bound is $L = 15$. Using the same steps previously, we can calculate $\eta \approx 0.0148$ in the OGA algorithm. As we can observe in the figure, the OGA is rising steeply in the first 10000 time slots. Because the learning rate changes from time slot to time slot, the UAC and APFC perform better, achieving a ten-times smaller regret. Figure 4 is a zoomed version of Figure 3, highlighting that UAC and APFC perform similarly in those cases.

5.2 UAC vs APFC

This subsection will present two scenarios where APFC outperforms UAC and OGA. APFC and UAC will outperform the OGA for the same reason as in the previous subsection. The purpose of this experiment is to compare UAC and APFC. Figures 5 and 6 represent different simulations of those algorithms using two request patterns with different trends. We define a *trend* as a period when a file is requested frequently compared to the other files. Both simulations are using $N = 400$ files.

In Figure 5, we perform a simulation using 100 trends having a cache size of $C = 100$. Every trend has 40 requests. In trend i, the file i is requested at least 25%, while the other files are randomly requested under the uniform distribution. In every trend, the first 75% of the requests have weight 1, while the last 25% have the weight 3. In the first 1000 iterations, the UAC and APFC have similar performances. This is happening because they are using similar learning rates. After that, we can observe a split; the regret of the APFC remaining lower because other trends are coming. This happens because the learning rate is high when a file without history is introduced. In UAC, the same learning rate is used at time t regardless of which file is requested. When new files become popular, the APFC performs better later in a simulation. It achieves a regret of around 230, while the UAC achieves a regret of around 350, which represents an increase of around 50%.

In Figure 6, we are using four cyclic trends, having a cache size of $C = 40$. For each cycle, we have 50 trends, each with 50 requests, of which at least 20% are for the popular file. The other 80% are randomly requesting a file using the uniform distribution. Each request has weight 1 in the first half of a trend, while in the second half, it is increased to 5. We plotted this experiment to show that even if the popular file has some history, like in cycles 2, 3, and 4, the APFC performs better. In this case, we can observe that the APFC obtains a negative regret at the horizon, meaning it performs better than the best static policy.

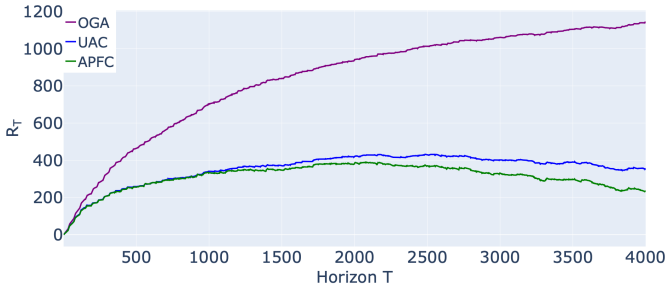

Figure 5: 100 trends with C = 100

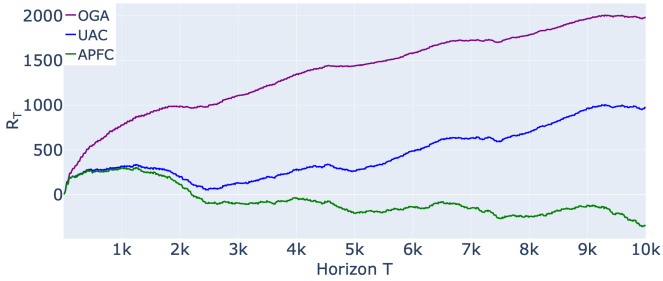

Figure 6: Cyclic trends with C = 40

6 Responsible Research

This section will discuss the ethical aspects of the research adhering to the Netherlands Code of Conduct for Research Integrity [1].

Throughout our study, we ensured that all real-world data used was publicly available and properly cited. We utilized the MovieLens dataset, a common resource in research for testing machine learning algorithms, which is FAIR (Findable, Accessible, Interoperable, and Reusable). For the second experiment, we wrote how the data can be produced, including the distributions used.

The algorithms used, UAC and APFC, are described in detail, including a pseudocode for each, ensuring transparency and enabling reproducibility. The results of the experiments are impartially analyzed based on the plots displayed in the paper.

7 Conclusions and Future Work

In conclusion, we highlighted the performance of two new online caching algorithms, which keep track of the importance of the files, Uniform Adaptive Caching (UAC) and Adaptive Per File Caching (APFC), alongside a comparison between them. Our main goal was to prove that the APFC and UAC algorithms improve the OGA algorithm. In a comparison between UAC and APFC, we showed that APFC adapts better when we have a dynamic environment where the weights of the files change frequently. In other experiments, the APFC and UAC performed similarly.

Extending these two new algorithms to work in real-time systems could provide practical benefits for future work. Another idea is to integrate a machine learning model to predict file request patterns, which could adapt the learning rate and further enhance caching efficiency.

Because UAC and APFC showed high adaptability to different request patterns compared with the OGA algorithm, which outperforms the classical offline caching policies such as LRU, LFU, or FIFO as written in [5], these algorithms may promise an improvement in the caching systems.

References

[1] Association of Universities in the Netherlands (VSNU). Netherlands Code of Conduct for Research Integrity, 2018.

[2] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like distributions: Evidence and

implications. In *INFOCOM '99*, volume 1, pages 126–134, March 1999.

[3] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat. It's time to revisit LRU vs. FIFO. In *Proceedings of the 12th USENIX HotStorage Workshop*, Berkeley, CA, USA, July 2020.

[4] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, 2012.

[5] L. Vigneri G. Iosifidis G. Paschos, A. Destounis. Learning to Cache with No Regrets. In *IEEE INFOCOM*, 2019.

[6] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: A view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement, IMC '07*, pages 15–28, New York, NY, USA, 2007. ACM.

[7] Anastasios Giovanidis and Alexandros Avranas. Spatial multi-LRU: Distributed caching for wireless networks with coverage overlaps. *arXiv preprint arXiv:1612.04363*, 2016.

[8] Yuyi Li, Tarek Si Salem, Giovanni Neglia, and Stratis Ioannidis. Online caching networks with adversarial guarantees. *Proc. ACM Meas. Anal. Comput. Syst.*, 5(3), 2021.

[9] H. Brendan McMahan. A survey of Algorithms and Analysis for Adaptive Online Learning. *Journal of Machine Learning Research*, 18(90):1–50, 2017.

[10] Naram Mhaisen, George Iosifidis, and Douglas Leith. Online Caching with Optimistic Learning. In *Proceedings of IFIP Networking*, 2022.

[11] Francesco Orabona. A Modern Introduction to Online Learning", 2023.

[12] Shai Shalev-Shwartz. *Online Learning and Online Convex Optimization*. Now Publishers Inc., 2012.

[13] T. Si Salem, G. Neglia, and S. Ioannidis. No-Regret Caching via Online Mirror Descent. In *Proc. of ICC*, 2021.

[14] J Van Den Berg and A Gandolfi. LRU is better than FIFO under the independent reference model. *Journal of applied probability*, 1992.

[15] M. Zinkevich. Online Convex Programming and Generalized Infinitesimal Gradient Ascent. In *ICML*, 2003.