Integral Caching using Online Mirror Descent in a Networked Context

QUENTIN JOHANNES OSCHATZ[1]
Supervisor(s): Georgios Iosifidis[1]**, Tareq Si Salem**[2]**, Naram Mhaisen**[1]
[1]**EEMCS, Delft University of Technology, The Netherlands**
[2]**Inria, Université Côte d'Azur, France**
22-6-2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

This paper explores algorithms to optimize networked caching, where requests for files can be handled by a local cache instead of a remote server. Caches work collaboratively to prevent redundant caching, and each new batch of file requests is used to update the entire network. Data is cached in an integral manner, meaning that only discrete files or chunks can be stored, not fractions of them. Bipartite networks are studied, though the proposed model supports arbitrary network topologies. The approach is based upon an Online Mirror Descent (OMD) policy, which has been shown to have sublinear regret in single cache scenarios.

1 Introduction

With the ever-expanding traffic on the internet, methods that decrease loads on servers and provide both lower latency when requesting information and overall increased system performance are essential. Caching is one such method, employing a network of small storage severs called "caches", which are deployed close to users and can serve requests directly. These devices generally do not have the capacity of storing all data that could be requested, leading to the requirement for algorithms to determine which files to store on a cache. Patterns in web traffic specifically can be exploited to both design and evaluate these algorithms.

With caching being both ubiquitous and essential, a great amount of research on both the performance of existing algorithms and the development of new methods has been performed. Traditionally, a large part of existing research focuses on modelling caching as a stochastic problem [1]. However, some newer work, including the algorithms presented in [2] and [1], model caching in an adversarial setting. This approach assumes that requests are generated by an adversary—instead of being sampled from a stochastic function. The performance of an online algorithm can then be evaluated using a metric called *regret*, calculated using the cost incurred by the algorithm under test over a time horizon $\mathcal{T}$ compared to the cost incurred by an optimal, static cache state over the same time horizon (a more thorough explanation is provided in section 2.2) [1]. An algorithm is said to be *no-regret* if its regret grows sublinearly compared to time, which effectively means that its time-averaged regret becomes negligible given enough time has passed, i.e., the policy experiences, on average, at most the same costs as the static optimum with hindsight knowledge [1].

The authors of [1] proposed a no-regret caching policy based upon an online version of the mirror descent algorithm. Their work was based on previous research performed in [2], where the authors modelled caching as an Online Convex Optimization (OCO) problem. Generalizing this method, the algorithm developed in [1] is able to handle batch processing of multiple requests in each time slot and integral caching, where only discrete chunks or files—but not fractions of them—can be stored in caches.

Building upon this online mirror descent (OMD) approach, the work presented in this paper aims to modify the base OMD algorithm in order to apply it to multi-cache networks. In such a setting, multiple caches are able to serve user requests, with each user connected to a subset of all caches. Additionally, caches may be connected to each other in a variety of topologies, such as in a hierarchical, tree-like structure. The goal of the new algorithm is to take the entire network topology into account when updating each cache state, reducing the chance of superfluous duplicates.

The paper structure will now be outlined. Section 2 includes both a short formal description of the caching problem, as well as a summary of related work—including the OMD algorithm. Following this, section 3 will delineate key improvements made to the OMD algorithm proposed in [1], followed by a presentation of results in section 4. Then, section 5 will include a discussion of the gathered findings, followed by a reflection on ethical implications of the research in section 6. Lastly, section 7 will provide a brief conclusion and proposal for future avenues of research.

2 Background

In order to accurately describe a caching policy, a formal model of a caching system must first be established. Once that model has been described, related work can be addressed to both give a summary of previous work on which this paper is based upon, as well as a clear line of reasoning as to why the algorithm presented in this paper contributes to the field.

2.1 Problem Description

Cache Model. The set of items that can be requested is represented by the catalog $\mathcal{N} = \{1, 2, ..., N\}$, with a cache being of size $k < N$. The system works within discrete time slots $t \in \{1, 2, \ldots, T\}$. Each request is modeled as $x_t \in \mathcal{X}$, where valid requests consist of a vector the size of N, with each entry representing the amount of requests present for that specific item n. Mathematically, it is defined as follows:

$$\mathcal{X} = \left\{ x \in \{0, 1, \ldots, B\}^N : \sum_{n=1}^{N} x_n \leq B \right\},$$

where B is the maximum number of requests at at any time slot t.

The cache state is constructed from a time-dependent vector $y_t \in \{0, 1\}^N$, which denotes whether a cache contains file n at time slot t. Furthermore, with the additional constraint placed on the cache in terms of total capacity, a state y_t is considered a valid cache state if and only if it is a member of set

$$\mathcal{Y} = \left\{ y \in \{0, 1\}^N : \sum_{n=1}^{N} y^n \leq k \right\}.$$

Some caching policies described in section 2.2 define cache state slightly differently, allowing for the elements of y_t to take any value in the interval $[0, 1]$ instead of only 0 or 1. Such policies are said to be "fractional" or "continuous", whereas the model defined by this paper is referred to as "integral" or "discrete". The former allow arbitrarily small chunks of files to be cached, while the later only allows entire files or fixed-size chunks of them to be stored.

Caching Policy. A caching policy δ maps a set of past requests $x_1, ..., x_{t-1}$ and configurations $y_1, ..., y_{t-1}$ to a new configuration $y_t(\delta) \in \mathcal{Y}$ at every time slot t.

Networked Caches. When modeling a caching system containing multiple caches, requests come from a source $i \in \mathcal{I}$ for an item n at time t. The information is then served by one or more caches $j \in \mathcal{J}$, or the repository server π. The definition of a request x_t is therefore expanded to be a matrix containing $\mathcal{I}$ rows and N columns, with $x_{t,i,n} = l$ representing l requests for item n from source i. The cache state y_t is similarly extended, with an additional dimension added to represent multiple caches.

A weight variable $w_{i,j,n}$ represents the cost associated with transferring a file over the link between source i and cache j for item n, being infinite if there is no link. Cost is a value associated with each file transfer, with caching algorithms attempting to minimize the overall cost of the system. Its formula is algorithm-specific, and can be influenced by many factors, such as bandwidth, and can also be tuned for reasons such as load balancing. This will inform the system which links it should prefer.

Modelling more complex topologies, such as a hierarchical network, can be achieved by creating virtual caches that represent a given path of caches. For instance, given a hierarchical topology with two caches j_1 and j_2, if j_2 is the "higher" cache and j_1 the user-facing cache, getting data from j_2 is modeled as receiving files from a cache $j_{1,2}$, with a link weight of $w_{i,1,n} + w_{i,2,n}$.

2.2 Related Work

The caching problem has a long history and many algorithms have been developed to address it, such as Least Recently Used (LRU) and Least Frequently Used (LFU). Both algorithms function in very similar ways. Every time a request is made, they first check if the needed data is already cached. If so, it is served to the user. Should the data not be cached, it is requested from the main server, and subsequently added to the cache. Whenever adding a new data chunk would violate the cache size constraint k, another item must be removed. Determining what data to remove is done according to the eviction policy, which is where the two algorithms diverge: LRU evicts data that has been used least recently, whereas LFU removes information that has been requested least frequently.

Both LRU and LFU work well under certain request patterns, but perform very poorly on others [2]. Specifically, the former works well for requests sampled from a distribution with a shifting popularity — i.e., the likelihood of certain data being requested — but under-performs on patterns with fixed popularity, while LFU performs well under fixed popularity [2]. In short, choosing the correct algorithm to use requires knowledge of the future request pattern, which, in practice, diminishes their usefulness.

More recently, machine learning has been used to attempt to achieve better performance and eliminate the need to choose the "correct" algorithm. In particular, offline policies which use historical request data as training data have been developed [3], [4]. While these methods have been shown to be effective in certain cases, they suffer from a similar flaw to LRU and LFU: they assume that request patterns remain similar over time. Should the request pattern not adhere to historical trends, the model requires re-training. Additionally, in order to create a proper training set, a large set of previous requests must be compiled and properly formatted, which can be expensive and time-consuming, or even prohibitive should no historical record exist. While some methods exist to mitigate this, such as using attributes of the catalog to predict popularity [4], they do not address the fundamental issue of dealing with changes to historical trends.

In [5], the authors propose a model-free caching policy that does not make assumptions about the request pattern, enabling general use and minimum performance guarantees. This algorithm, Online Gradient Ascent (OGA), is based upon Online Convex Optimization (OCO) [5]. It employs gradient descent at each time slot, taking a step in the direction of the gradient of a utility function

$$U_{x_t}(y_t) = \sum_{n=1}^{N} w_n x_{t,n} y_{t,n} \tag{1}$$

where w is a vector of weights for each file, which can—for instance—be defined uniformly to maximize the raw cache hit ratio [5]. The "step" is added to a continuous intermediate cache state. Before being able to convert the intermediate cache to the proper cache state y_t, it must be ensured that the new cache state does not violate the size constraints. For this purpose, the intermediate state is projected onto the set $\mathcal{Y}$, which ensures that k is not exceeded while also choosing the closest valid state to the current "ideal" state. The gradient always "points" towards what would be optimal for the current request, and the existing intermediate cache is a summary of all previous gradient steps.

The OGA algorithm, contrary to policies such as LRU and LFU, is able to perform well even in an adversarial setting [5]. In such an environment, requests are not simply sampled from some distribution, but rather deliberately picked by an adversary in an attempt to achieve a cache miss [5].

In order to more properly benchmark algorithms in terms of pattern agnosticism and resistance to adversarial attacks, *regret* is used [5]. *Regret* compares the cost of a given algorithm to that of an optimal static cache policy created with hindsight, meaning that it is generated after every request is known [5]. Formally, it can be described as follows:

$$\text{Regret}_T(\delta) = \sup_{\{x_1,\ldots,x_t\}\in\mathcal{X}^T} \left\{ \sum_{t=1}^{T} C_{x_t}(y_t(\delta)) - \sum_{t=1}^{T} C_{x_t}(y*) \right\}, \tag{2}$$

where T is the time horizon, δ is the caching policy to be evaluated, and $y*$ is the optimal static cache state [1]. Ideally, caching policies would see their regret grow in a sublinear manner when compared to time, which means that the caching policies would experience—on average—no more cost than the static optimum with hindsight knowledge [2]. The result of using such a metric is that it gives a more applicable standard of comparison. Should an adversary generate a sequence of abruptly shifting, unpredictable cache requests—which turns caching into a mostly luck-based affair, both the static policy and the algorithm will perform poorly.

On the other hand, very predictable patterns will allow for high-scoring static policies, and, therefore, a high bar for the algorithm to pass. In short, the regret metric allows robust evaluation of policies based on how well they can learn and predict a broad range of request patterns. The OGA algorithm has been shown to have the mentioned ideal sublinear regret growth over time [5].

While possessing sublinear regret is an extremely important feature, the OGA algorithm comes with a few key constraints that are not ideal, especially in the setting explored by this paper. First, it can only process one request per time slot, which is especially detrimental during periods of high demand. After each request, both the gradient and projection must be recalculated, which can be especially demanding in a multi-cache setup with multiple users requesting files. Second, the additive nature of the update rule used can slow down responsiveness to sudden, extreme changes of file popularity [1]. It will take a number of requests for the additions to previously unpopular items to overcome the existing margin of the previously popular items. Third, the algorithm proposed by [5] is a fractional policy, whereas this paper aims to develop an integral algorithm.

In [1], the authors present a generalized version of the OGA algorithm based on Online Mirror Descent (OMD). This policy enables requesting multiple files per time slot and allows usage of a broader range of update functions [1]. Specifically, it allows the use of a mirror map, an essential function in mirror descent algorithms [1]. Such algorithms presume that variables and gradients live in two different spaces, linked by the mirror map [1]. Gradient updates are performed in one space, then the change is translated using the mirror map to the variables [1]. This leads to faster convergence in several cases connected to caching [1]. The authors of [1] developed a specific instance of OMD using a negative-entropy mirror map, which brings both superior regret performance when compared to OGA in cases where request batches contain many different files, while also allowing for performance optimization via the Bregman projection. This instance also uses a multiplicative update rule instead of an additive one, increasing the reactivity to sudden popularity changes [1].

A notable aspect of OCO-based approaches is their default behaviour of continuous, partial caching [5]. Partial caching refers to being able to store only needed fractions of files, which has been studied in the context of media caching [6]. As a result of the gradient descent step used in OCO algorithms, resulting cache states are fully continuous, meaning that the algorithm assumes infinitely small fractions of files can be stored [5]. In practice, this is rarely the case. Not only are files constructed from discrete bits, but splitting arbitrary file types may also not always be possible, especially concerning information such as file metadata. Additionally, the upsides of partial caching—being able to store only relevant parts of large files—can also be achieved by creating discrete file chunks, and applying integral caching methods to them [6]. Using any deterministic rounding policy to generate an integral cache state from the result of an OCO algorithm, however, will have a grave impact on performance guarantees [1].

Symbol	Definition
v_t	Integral cache state of all caches at timeslot t
y_t	Fractional cache state of all caches at timeslot t
y_*	Optimal static cache state in hindsight
x_t	A matrix of the number of requests from source i for file n at timeslot t
w	A matrix of weights from each source i to each cache j regarding file n
$\mathcal{J}$	The set of all caches in the network
$\mathcal{I}$	The set of all request sources in the network
$\mathcal{N}$	The catalog of all files
T	The time horizon
η	The step size
u	The subgradient of the utility function
$C_x(v)$	The cost attributed to a certain request batch under a certain cache state
$U_x(v)$	The utility attributed to a certain request batch under a certain cache state
z	A matrix of fractions representing the amount of data transferred to source i from cache j regarding file n

Table 1: A summary of the used notation

In order to derive an integral caching algorithm from an OMD algorithm, a rounding step is added. Since [1] proves that any deterministic rounding algorithm has a strongly negative impact on performance in adversarial settings, as it would allow an adversary to have perfect knowledge of the cache state and thereby always request non-cached files, randomness is introduced. Specifically, the online coupled rounding, described in [1], is used to derive discrete cache states while not overwriting the internal, continuous state used for mirror descent.

Moving to a multi-cache environment, [5] introduced a variant of the OGA that would function in a bipartite network. However, this algorithm has the same constraints placed upon it as the single-cache variant. Outside the realm of OCO-based caching, [7] described a bipartite model to be used by a Follow the Perturbed Leader (FTPL) policy. Even though it was designed for a different algorithm, its basic concepts can be adapted for an OMD-based approach.

3 OMD: Connected and Integral

Caches frequently operate as a network, with bipartite or hierarchical topologies being common [8]. As shown in section 4.2, in such settings, optimal results require that the caches make network-optimal decisions, not just individually-optimal decisions. Therefore, it is not sufficient to simply run an OMD-variant on each cache in the network.

3.1 Multi-Cache Utility

In order for the algorithm to take the network into account when updating cache states, the links between request sources and caches must be modeled. For this purpose, a weight matrix $w \in \mathcal{I} \times \mathcal{J} \times \mathcal{N}$ is introduced, which represents both the presence of links, as well as encoding which links should be preferred by the network over others.

With the weights matrix representing the links, the update function of the OMD algorithm must be modified to make use of this new variable. However, in addition the update function—i.e. the gradient of the cost function—needing to take into account links, it must also prevent superfluous caching. A file should not be cached by multiple caches if the requests can be served by one of the caches, and the cost function should reflect this. In [7], this is solved by adding a term to the function that negates gained cost for any additional, reachable cache storing an already-cached file. The final utility (which can also be converted to cost) U of a given cache state v_t (v_t is the integral cache state, different from the fractional y_t) can then be calculated using the cost of retrieving the requested files from the remote server $C(0)$, minus the savings incurred by requests served by caches $C(v_t)$:

$$U(v_t, x_t) = \sum_{n=1}^{N} \sum_{i \in \mathcal{I}} x_{t,i,n} \left(C(0, i, n) - C(v_t, i, n) \right), \quad (3)$$

$$\text{where } C(v_t, i, n) = \sum_{j \in \mathcal{J}} (w_{i,j,n} - w_{i,j-1,n}) P(t, i, j, n) \quad (4)$$

$$\text{and } P(t, i, j, n) = 1 - \min \left\{ 1, \sum_{k=1}^{j-1} v_{t,k,n} \right\}. \quad (5)$$

The cost is calculated for each item being requested, with $x_{t,i,n}$ being 0 for all files not being requested. To evaluate the cost of a certain cache state, $w_{i,j,n} - w_{i,j-1,n}$ represents the cost reduction received if—given both cache j and $j-1$ have cached a file n—requesting that file from j instead of $j-1$. $w_{i,0,n}$ is equal to 0, since that cache does not exist. This not only ensures that the file is retrieved from the optimal cache, but also that unconnected caches are not used.

The second term ensures that cached files are only rewarded once. $\sum_{k=1}^{j} v_{t,k,n}$ sums up the total percentage of file n that is stored in caches $1, \ldots, j$ at time t, while the $\min$ function caps the sum at a maximum of 1. As long as connected caches have not cumulatively stored the entire file, the minimum function remains below 1, and, as such, $P(t, i, j, n) > 0$. Since this algorithm is performed on integral caches, $v_{t,j,n}$ will always be either 1 or 0. As soon as the first cache j is found to store the requested file n, for all following caches, the $\min$ function will return 1, resulting in $P(t, i, j, n)$ equalling 0. If the caches are sorted in such a way that the most "ideal" servers (i.e., those with the lowest weights) are searched first, then this equates to always choosing the most cost-optimal cache in cases where multiple caches are available.

In implementation, the cost function can be simplified massively. For instance, has already been noted that, as soon as a cache is found to store the requested file, all subsequent rewards go to 0 and can thus be ignored. Additionally, all weights of caches inspected before that cache is found can also be ignored. This can be demonstrated through an example, for instance using a system made up of 3 caches, with second cache storing the requested item. The cost function $C(v_t, j, n)$ would then be evaluated as follows:

$$\begin{aligned} C(v_t, i, n) &= (w_{i,3,n} - w_{i,2,n}) \cdot P(t, i, 3, n) + \\ &\quad (w_{i,2,n} - w_{i,1,n}) \cdot P(t, i, 2, n) + \\ &\quad (w_{i,1,n} - w_{i,0,n}) \cdot P(t, i, 1, n) \\ &= (w_{i,3,n} - w_{i,2,n}) \cdot 0 + \\ &\quad (w_{i,2,n} - w_{i,1,n}) \cdot 1 + \\ &\quad (w_{i,1,n} - w_{i,0,n}) \cdot 1 \\ &= (w_{i,2,n} - w_{i,1,n}) + (w_{i,1,n} - w_{i,0,n}) \\ &= w_{i,2,n} - w_{i,1,n} + w_{i,1,n} - 0 \\ &= w_{i,2,n} \end{aligned}$$

This same chain of cancelling terms works for any number of caches and weight configurations, resulting in the far simpler cost function

$$C(v_t, i, n) = \min_{j \in \mathcal{J}} \left\{ w_{i,j,n} v_{t,j,n} \right\} \quad (6)$$

3.2 Multi-Cache Subgradient

In order for an OMD-based policy to be implemented, a function must be provided which will direct the cache state towards a currently-optimal state based on the newest set of requests. Traditionally, this is function is derived directly from the cost function by merely taking its gradient. In this case, however, the presence of a $\min$ operation in the cost function complicates matters: $C_x(v_t)$ no longer has a single, defined gradient at every point. Instead, there is a point where the $\min$ operator caps the function, leading to a sharp change in direction and gradient. At that intersection point, multiple gradients exist. For this reason, a subgradient must be used instead.

This subgradient is created using a piecewise function, which always returns only one unique result. Crucially, it maintains the reward gain cap of the utility function, becoming 0 for file entries that, while requested, are already fully cached in a connected cache.

$$c_t \in \partial_{x_t} C_x(v_t) = \begin{cases} \sum_{i \in \mathcal{I}} w_i x_t, & \text{if } \sum_{j \in \mathcal{J}} \sum_{n \in \mathcal{N}} v_{t,j,n} \leq 1 \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

3.3 Rounding

Due to the base OMD policy returning fractional cache states, rounding is used to derive integral cache states. In [1], the authors proved that any deterministic rounding policy would lead to the dissolution of the regret bounds ensured by the OMD policy. Intuitively, this can be understood by considering the adversarial setting: if the rounding can be predicted with perfect accuracy, an adversary can always request a file that is not cached at all, resulting in 0 utility. In a fractional setting, this is not as much of an issue, since the caches can store small chunks of every file, meaning that some utility will be gained.

For this policy, the Online Coupled Rounding algorithm presented in [1] is used. After the OMD updates and projects the caches, rounding is applied individually on each cache. For simplicity, the random variable used to prevent determinism is drawn only once, and is the same for all caches.

3.4 The Algorithm

Algorithm 1 Bipartite OMD

Require: $\eta \in \mathbb{R}^+, y_1 \in \{k|k > 0\}^{\mathcal{J}\times\mathcal{N}}$
  for $t \leftarrow 1, 2, \ldots, T$ **do**
    $y_{t+1} \leftarrow y_t e^{(-\eta u_t)}$
    $y_{t+1} \leftarrow \Pi(y_{t+1})$
    $v_{t+1} \leftarrow$ Online Coupled Rounding(y_{t+1})
  end for

Assembling the parts, the final algorithm broadly resembles the base OMD policy with the negative-entropy map. The algorithm loops for each new batch of requests, starting each cycle by calculating the subgradient. Next, mirroring the update rule used by the authors of [1] for the OMD variant using the negative-entropy map, the subgradient is applied to the existing state via a multiplicative update rule.

$$y_t = y_t e^{-\eta u_t}. \tag{8}$$

After the update, the cache states will have likely moved outside of the set of constraints placed on them, meaning that they must be modified to become valid again. In order to maintain the effect of the update, a projection onto the set of valid cache states is performed using the function Π. Since the constraints (namely cache size) are enforced on a per-cache basis, this projection is also performed individually. The algorithm for this is identical to that used in [1] for the negative-entropy mirror map.

Lastly, the final cache states are derived using the rounding described in section 3.3. It should be noted here that the internal state used for the OMD calculations remains unchanged by the rounding; instead, merely its output is rounded. If the internal state were to also be rounded, the policy would loose most of its learning capabilities, with the subtle, gradual updates caused by incoming requests rounded to either no change or drastic change.

3.5 Extension: Simplified Network Model

In order to avoid enforcing the cap on reward gain for redundant caching within the cache update directly, it can be extracted as a constraint, massively simplifying the network. In [9], the authors accomplished this through the introduction of a new variable, z, which connects the cache states and propagates updates. For any request $x_{t,i,n}$ at time $t \in T$ from source $i \in \mathcal{I}$ for item $n \in \mathcal{N}$, $z_{i,j,n}$ holds what percent of the item is routed from cache $j \in \mathcal{J}$. The gradient update then acts on this new variable z_t, instead of directly on the cache state, using an modified version of the utility function Eq. 1, namely Eq. 9 and its gradient Eq. 10.

$$f_t(z_t) = \sum_{i\in\mathcal{I}} \sum_{j\in\mathcal{J}} \sum_{n\in\mathcal{N}} x_{t,i,n} z_{i,j,n} w_{i,j,n} \tag{9}$$

$$\nabla f_t(z_t) = (x_{t,i,n} w_{i,j,n})_{i,j,n\in\mathcal{I}\times\mathcal{J}\times\mathcal{N}}. \tag{10}$$

While updating z does not directly update cache states, several constraints connect the two, namely the ones presented below. The changes to z are therefore propagated to the cache states during the projection, since this step will modify y_t in order to adhere to the constraints.

$$\sum_{j\in\mathcal{J}} z_{i,j,n} \leq 1, \qquad \forall i \in \mathcal{I}, \forall n \in \mathcal{N}$$

$$z_{i,j,n} \leq y_{j,n}. \qquad \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, \forall n \in \mathcal{N}$$

This removes the need for a minimum function, and thereby the problem of multiple gradients, simplifying the entire process. Unfortunately, the very factor that allows for the simplification—the variable z—also makes rounding the result into an integral state a non-trivial problem. When the cache state y_t is rounded to an integral state, the constraints placed on the variables are no longer guaranteed to hold, and since, the performance guarantees no longer apply either. For this reason, this model is not used in this paper, but rather described as an avenue for potential future work.

4 Results

In order to properly present the results from benchmarking the modified OMD algorithm presented in section 3, this section first provides a brief description of the evaluation setup, after which the key findings are outlined.

4.1 Experimental Setup

In order to properly evaluate the developed algorithm, a simulation suite was developed in Python. The program employs an object-oriented design, with caching algorithms all inheriting from a base "cache" class. Additionally, data is fed into the model via classes inheriting from an overarching "trace" class. These data sources range from simply feeding a static array, to generating an arbitrary-length sequence of requests from a given seed using a certain generation method. Metrics, such as hit ratio and cost, are recorded by the caching algorithms. In order to streamline evaluation, especially for comparing several caching policies, a benchmarking class was written. This class, when given a JSON configuration file, will setup a simulation with the listed traces, caches, weights and other parameters. If the "seed" parameter is set, the same JSON file will always yield the same setup and results. Once complete, it will collect the gathered results, generate the static optimal cache used for calculating regret, and will finally record parameters such as the seed (especially when no particular seed was provided and thus one was generated randomly), plot the metrics of each algorithm, and save the raw results, plot and JSON file to a generated folder. The entire setup allows both reproduction of any particular run by feeding the used seed, and is extremely flexible due to its modular design.

In order to evaluate the described algorithm, a multitude of setups were used to benchmark different scenarios. Three different traces were used to generate request, each taken from [10]. First, a trace using a fixed popularity is used. This samples requests from a zipf distribution with a fixed s value of 0.6. Next, a sliding window trace is used, where the same zipf distribution is used, though its graph is shifted 5 times. This simulates a sudden change in what items are popular. Lastly, an adversarial trace is also used. Here, the circular

(aka oscillator) trace is used, which requests items in a circular manner, always requesting $(n+1) \mod N$ next, if n was just requested. This makes it particularly hard to gain a high utility, especially algorithms with only short-term or no learning. These policies usually discard older items due to them not being requested in favor of newer items, leading to and endless cycle of evicting files needed for upcoming requests in favor of items that will not be requested in the near future. In most cases, a catalog size of 1000 was used for the traces, as it provides an adequate amount of choice for both the requests and the caching policies, reducing the probability of repeated "lucky guesses". A time horizon of 1000 is used in most experiments for similar reasons.

Cache misses have a negative impact on performance, as measured by Normalized Average Cost (NAC). This metric is calculated by adding all normalized cost incurred up until the current timeslot t, and then dividing that by the amount of time that has passed. Normalized cost refers to cost divided by the total number of requests received in that timeslot.

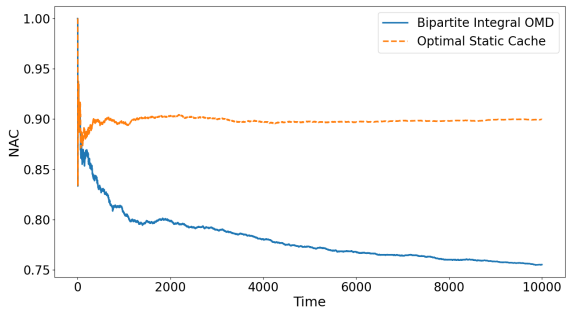
All experiments use the same network topology, constructed out of 2 sources and 3 caches, with caches 1 and 3 being connected to sources 1 and 2 respectively, and cache 2 being connected to both sources. All link weights are set uniformly. This setup was chosen for its balance between speed and variety. Speed here refers to the speed of the simulation, since each additional entity adds an immense amount of required processing, especially the already-slow projection step. With only 5 entities, it is able to test how a multi-cache policy is able to use network-wide collaboration to make better use of the total cache space, while also demonstrating that it correctly handles unlinked caches. The former is achieved by cache 2, which acts as a shared cache for both sources. For instance, with clever use, files common to both sources can be cached there instead of in both cache 1 and cache 3, freeing up more space in the network without decreasing performance. Handling of unlinked caches is also tested, mainly through the other 2 caches: only one source has access to them, meaning that the cache state there should tailor it to that source's pattern, and not that of the network. Other topologies, such as a hierarchical one, are not tested in this paper, but may be benchmarked in future work.

4.2 Findings

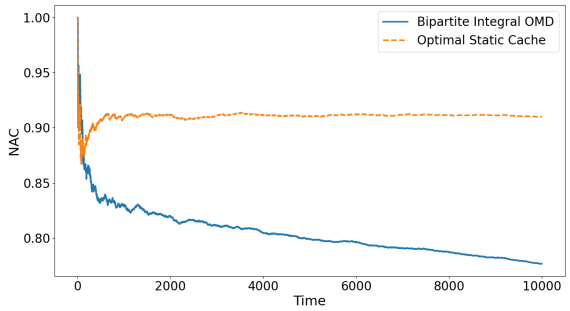
Observing the plots presented in figure 1, all three show the NAC of the OMD policy decreasing over time, converging to lower values. In addition, all three show the optimal static policy consistently scoring a higher NAC.

In figure 2, the results show the algorithm developed in this paper noticeably outperform two "naive" policies. The LFU algorithm does not manage a single cache hit, while the naive OMD policy evidently performs worse than this paper's algorithm, hovering close to the static optimal policy and converging to it over time.

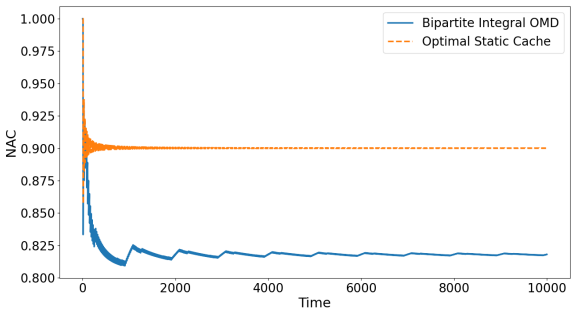
Investigating different step sizes, figure 3 shows different instances of the policy applied to the same fixed-popularity trace. The different step sizes lead to diverging convergence rates, as well as differing end results. All variations maintain a roughly similar shaped curve.



(a) Fixed popularity trace, catalog size 1000



(b) Sliding popularity trace, catalog size 1000



(c) Circular adversarial trace, catalog size 1000

Figure 1: Plotting the Normalized Average Cost of the Bipartite OMD policy over three different traces. In each plot, the topology consists of 2 sources and 3 caches (each of size 100), with cache 1 connected to source 1, cache 3 connected to source 2 and cache 2 connected to both.

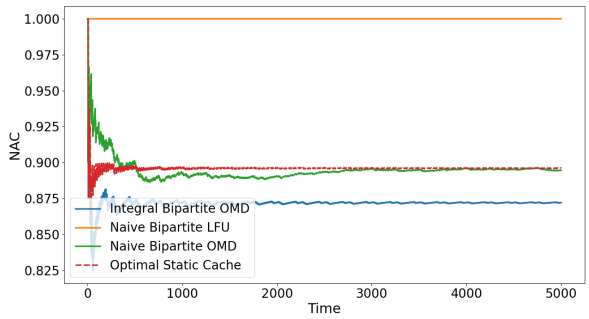


Figure 2: The developed OMD policy is tested against naive versions of OMD using negative-entropy and LFU. These naive variants only act on an individual cache basis, making decisions unaware and independent of each other. Topology is identical to figure 1, using the circular adversarial popularity trace with a catalog size of 250 and cache sizes of 25 each, since higher values were infeasible on available hardware due to performance reasons.

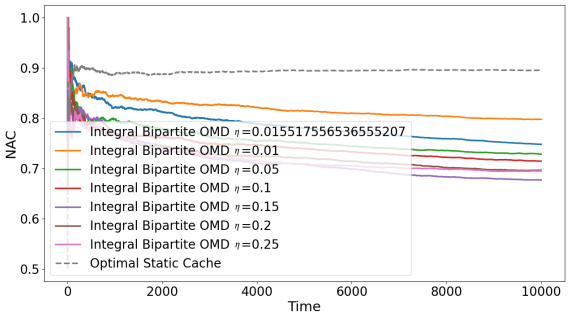


Figure 3: Different step sizes used for the Bipartite OMD policy. Topology is identical to figure 1, using the fixed trace with a catalog size of 1000.

5 Discussion

From the results presented in section 4.2, it can be concluded that the bipartite OMD algorithm performs well in a variety of settings, even in the face of adversarial requests. It not only consistently bested the optimal static policy, but shows clear signs of learning, especially in figures 1a and 1a, where the NAC is seen to decrease consistently, with the decrease slowing as the network converges. Figure 1c shows more abrupt changes, likely due to the unusual request pattern.

Recalling the regret metric introduced in earlier sections, it can be observed that, using the NAC of both the Bipartite OMD and the optimal static cache, the time-averaged regret can be calculated. As described earlier, a key feature of algorithms that perform well under adversarial request patterns is that their regret grows sublinearly compare to time. In other words, as $t \to \infty$, the time-averaged regret should become negligible. Looking at figure 1, it can be seen that, in all three cases, the cost of the OMD variant either decreases or remains at parity with that of the optimal static policy. Therefore, for high values of t, this difference, and thereby the regret, approach 0.

Figure 2 is especially relevant for this paper. It clearly demonstrates that networked policies vastly outperform distributed, independent instances of caching algorithms.

The performance of the LFU network is no surprise, given that the trace is adversarial in nature and manages to "break" the LFU in a similar manner in a single cache scenario as well. Nevertheless, it demonstrates, should a series of requests follow such a trace, a network relying on a fully distributed set of LFU policies would suffer devastating performance loss.

Looking at the naive OMD policy, it can be observed that it is able to serve some number of requests. However, when comparing it to the networked algorithm, it becomes obvious that the latter is able to more reliably produce cache hits.

Summarizing the results, the experiments seem extremely promising, demonstrating the algorithm's potential in a networked setting. It performs consistently in the range of experiments presented, and the performance implies that further study of the policy may be fruitful. However, it must be noted that only a handful of scenarios have been tested and presented here. Theoretical bounds on the regret of this algorithm have yet to be developed and proven, meaning that there is still substantial work to be done before this algorithm can receive the desirable label of a "no-regret policy".

6 Responsible Research

In 2010, the Yale Law School Roundtable on Data and Code Sharing released a document addressing what they called a "credibility crisis" present in the field of computational science [11]. A key emphasis of the remaining document was the importance of being able to reproduce findings presented in scientific articles in order to allow for easy peer review and validation of results [11]. For this reason, reproducibility was a key consideration throughout the process. In the case of this paper and the work presented within, this tenet is baked into the core design of the both the implementation of the algorithm and the developed simulation software.

All parameters of a simulation run, including the seed used for all random sampling and data generation, are configurable through a JSON file. After every run, a folder is created that stores a CSV text file of the raw results, a plot of the results over time, and a copy of this configuration file. This ensures that experiments can be reproduced seamlessly while only needing to share the source code and the appropriate parameters file. Since the data is also generated deterministically from the seed, it is shared directly with the the network topology and other environment settings. This results in experiments being trivial to reproduce for any reader of this paper.

Related to this, the source code of the simulator as well as the parameters file for all experiments performed in this paper are fully open source, in line with the recommendations laid forth in [11]. In addition, to prevent dependency clash and to ensure that the exact same setup is available for other researchers, a tool named "Pipenv" [12] is used, which captures the entire Python environment, including exact versions of Python and all dependencies. Through it, users can easily and reliably recreate the exact environment used in the paper.

Next to reproducibility, the Netherlands Code of Conduct for Research Integrity defines an entire range of principles as best practices for responsible research [13]. These cover a broad variety of aspects, from virtues such as honesty and scrupulousness, which are, of course, held in high regard by the authors of this work, over transparency (which has already been addressed in the previous paragraphs), to independence and responsibility [13]. These principles are then used to give rise to a set of standards [13].

A portion of these standards reflect the need claims made to be substantiated and presented honestly [13]. The work in this paper is not yet fully mature; while the algorithm has been developed, implemented, and empirical results have been collected, the theoretical work proving bounds on performance has yet to be completed. The paper attempts to honestly present the accuracy of its empirical results, emphasizing that the results are not fully conclusive.

Lastly, it should be noted that the author has not, in fact, faked any of the presented results, nor intentionally left out data that contradicted findings.

7 Conclusion

This paper set out to develop a variant of the integral OMD caching policy for multi-cache settings. For this, a new variable was introduced to represent links between caches, and the utility function was modified to prevent files from being cached in more places than necessary. As a result, the gradient used in previous work had to be exchanged for a subgradient, as the new utility function is not necessarily differentiable.

A small set of empirical results were collected in order to test the developed policy in a multi-cache setting, as well as to compare it to naive, completely distributed approaches. In all instances, the collected data shows promise, showing that the algorithm retains sublinear regret in all three of the tested cases, as well as comfortably outperforming two naive policies.

In conclusion, the work presented in this paper shows a promising method of expanding the OMD algorithm to support multi-cache systems in an integral setting.

7.1 Future Work

Two immediate avenues of future work lie in theoretical validation. Determining the exact regret bounds and finding a formula for the optimal step size η could not be accomplished before the end of the project. Inspiration could be drawn from [2] and [7] for the regret bounds, and [1] for the step size. The former two papers already prove regret bounds for bipartite policies of similar nature (OGA and FTPL), while the latter deals with the optimal step size for non-networked OMD caches with a negative-entropy map, the same variant of OMD employed in this work. Finding the regret bounds would prove (or disprove) what the experimental findings imply, and provide numerical guarantees for the policy, while determining the optimal step size would likely boost performance.

Although the algorithm was tested in a variety of settings, extending the developed simulator in order to test the algorithm against real datasets with realistic topologies would give an insight as to how performance translates to actual implementations. Here, the average performance is especially interesting, which may not completely match the theoretical metrics gathered here. In order to gain even better performance in such settings, instead of using uniform weights for links, these can be adjusted to deliberately direct the network to prefer some caches over others for requests from a certain source. Reasons for this may include load balancing, geographic distance (which can influence latency) or bandwidth limitations.

In a similar vein, developing faster implementations of parts of or even the entire presented algorithm would substantially improve its feasibility in a real application. Currently, OCO libraries such as CVXPY[14] are used for procedures such as projection and generating the optimal static policy. While these packages are extremely useful in academic work, where speed is less of a concern, and being able to quickly change formulas is essential, they are not able to deliver the performance needed for large, real-time caching systems. In [1], the authors were able to develop a faster projection algorithm for their single-cache setup, which may be adapted for the networked setting.

Another area of future work concerns the simplified network model introduced in section 3. This model made use of a helper variable z that would propagate changes to the cache state, representing the percentage of a requested file retrieved from a specific cache in the network. With the variable acting as a bridge, should rounding be applied to the cache state in order to achieve integral caching, constraints placed on the relationship between the elements of the cache and z may no longer hold. For this reason, it was not incorporated into the algorithm, which used more complex sub-gradients. Future work may be able to develop a rounding technique which is able to derive an integral state from the fractional output of the simplified model without violating these constraints.

8 Acknowledgements

The author would like to thank Prof. Dr. Georgios Iosfidis, Tareq Salem and Naram Mhaisen for their repeated and invaluable guidance, both in regards to conducting the work as well as in the writing of this paper presenting it. Without them, this work would not exist.

References

[1] T. Si Salem, G. Neglia, and S. Ioannidis, "No-Regret Caching via Online Mirror Descent," *IEEE International Conference on Communications*, Jun. 2021, ISSN: 15503607.

[2] G. Paschos, G. Iosifidis, and G. Caire, *Cache Optimization Models and Algorithms*, 3–4. Now Publishers, Inc., Aug. 2020, vol. 16, pp. 156–345, ISBN: 9781680835960.

[3] E. Bastug, M. Bennis, E. Zeydan, M. A. Kader, I. A. Karatepe, A. S. Er, and M. Debbah, "Big Data Meets Telcos: A Proactive Caching Perspective," *Journal of Communications and Networks*, vol. 17, no. 6, pp. 549–557, Feb. 2016, ISSN: 12292370.

[4] S. M. S. Tanzil, W. Hoiles, and V. Krishnamurthy, "Adaptive Scheme for Caching YouTube Content in a Cellular Network: Machine Learning Approach," *IEEE Access*, vol. 5, pp. 5870–5881, 2017, ISSN: 2169-3536.

[5] G. S. Paschos, A. Destounis, L. Vigneri, and G. Iosifidis, "Learning to cache with no regrets," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 235–243.

[6] L. Wang, S. Bayhan, and J. Kangasharju, "Optimal chunking and partial caching in information-centric networks," *Computer Communications*, vol. 61, pp. 48–57, May 2015, ISSN: 01403664.

[7] D. Paria and A. Sinha, "LeadCache: Regret-Optimal Caching in Networks," *NeurIPS 2021*, 2021.

[8] Z. Li, G. Simon, and A. Gravey, "Caching Policies for In-Network Caching," in *2012 21st International Conference on Computer Communications and Networks (ICCCN)*, IEEE, Jul. 2012, pp. 1–7, ISBN: 978-1-4673-1544-9.

[9] N. Mhaisen, G. Iosifidis, and D. Leith, "Online Caching with no Regret: Optimistic Learning via Recommendations," Apr. 2022.

[10] Y. Li, T. Si Salem, G. Neglia, and S. Ioannidis, "Online Caching Networks with Adversarial Guarantees," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 5, no. 3, Dec. 2021, ISSN: 24761249. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/3491047.

[11] V. C. Stodden, "Reproducible Research: Addressing the Need for Data and Code Sharing in Computational Science," vol. 12, no. 5, pp. 8–12, 2010.

[12] *Pipenv: Python Dev Workflow for Humans*. [Online]. Available: https://pipenv.pypa.io/en/latest/.

[13] *Netherlands Code of Conduct for Research Integrity — NWO*. [Online]. Available: https://www.nwo.nl/en/netherlands-code-conduct-research-integrity.

[14] *CVXPY 1.2*. [Online]. Available: https://www.cvxpy.org/.