Delft University of Technology
Master of Science Thesis in Embedded Systems

Dynamic Cache Replacement Policy Selection Using Experts

Kwoon San Lam

TU Delft Delft University of Technology

Dynamic Cache Replacement Policy Selection Using Experts

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Kwoon San Lam
koen.lam@student.tudelft.nl
koen.lam@live.nl

30-03-2022

Author
Kwoon San Lam (koen.lam@student.tudelft.nl)
(koen.lam@live.nl)

Title
Dynamic Cache Replacement Policy Selection Using Experts

MSc Presentation Date
13-04-2022

Graduation Committee

dr. ir. Fernando Kuipers (chairman)	Delft University of Technology
dr. George Iosifidis	Delft University of Technology
prof. dr. ir. Geert Leus	Delft University of Technology

Abstract

In recent years, researchers proposed several universal caching policies. These universal caching policies aim to work well with any request sequence. However, with this universal well-working property, these caching policies sometimes do not work as well as conventional caching policies such as Least-Recently-Used and Least-Frequently-Used. This thesis introduces an online learning approach to dynamically switch between caching policies. The dynamical switching selects the most suitable caching policy for the request sequence and thus ensures the best of both worlds, working well under any request sequence and having comparable performance to the conventional caching policies. The Dynamic Expert Caching (DEC) policy designed in this thesis uses algorithms from the expert selection problem to achieve the dynamic caching policy selection. The DEC policy is evaluated using various datasets where it is shown to have comparable performance to the best caching policies in the simulations. The DEC policy is also adapted to work in a network of caches, named MultiDEC. In this setting, MultiDEC has significant advantages over other universal caching policies where each caching node can have a different caching policy that is optimal.

Preface

Caching is fundamental in today's infrastructure and is being applied in more and more places. Online learning techniques have seen tremendous success in other fields and thus they are also applied in the field of caching.

This period where I worked on my thesis has been the most difficult time of my life. I lost the rhythm that physically going to the university brought. I struggled to be efficient with my time. I felt my life has been put on pause. There were even times I thought this project was not going anywhere.

That is why I'm incredibly grateful the finish line is finally visible. I would like to thank George Iosifidis for his guidance during this project. To Mirza Mrahorović, thank you for all the times we worked together. My time in Delft would not have been the same without you. I really appreciate your support when I needed it. To Zakaria Abdellaoui and Tim Sweering, thank you for being my partners during my master's courses. Lastly, I would like to thank my family for supporting me during this difficult period. It has taken way longer than it should've had but now I'm very happy it is coming to an end.

Kwoon San Lam

Delft, The Netherlands
30th March 2022

Contents

Chapter 1

Introduction

Mobile data traffic is growing dramatically each year with a forecasted annual growth rate of 46% from 2017 to 2022 according to a recent white paper of Cisco [16]. A major contributor to this growth is the consumption of streaming video-on-demand content which will represent up to 82% of the total IP traffic.

Small cell architectures are proposed to support this traffic increase. These small base stations located closer to the user have a short-range and can increase the network throughput by an order of magnitude [35]. However, to obtain the same coverage many small base stations are required and all of them might need high-speed backhaul connectivity. This can cause the backhaul capacity to become the system bottleneck [44]. A solution is to have a small cache at each of these small base stations. This cache can store data that can be reused for future requests and thus reduce the backhaul load.

Caching is a well-established technique in for example content distribution networks. However, caching at the small base stations has some additional challenges. Small base stations usually have a short coverage, a small cache, and few users. This causes the request distribution to be highly non-stationary which in turn results in caching policies designed for the internet to be less effective [3].

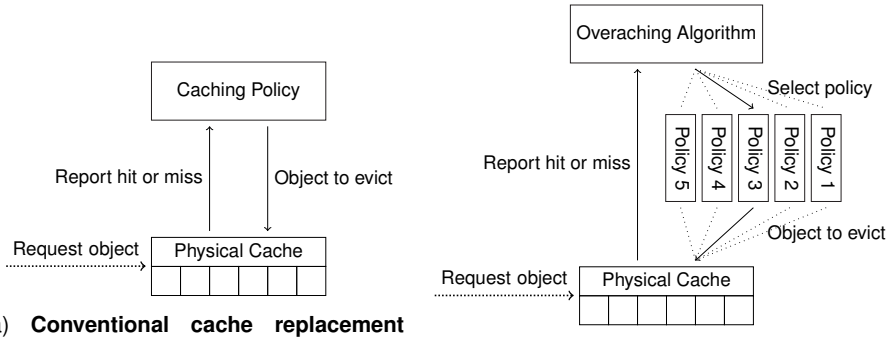
In this challenging setting Paschos et al. [39] propose to study caching policies in a model-free setting with no assumption on the request sequence. This means that the request sequence can be anything and even a request sequence designed to degrade the performance of the caching policy. Paschos et al. propose the OGA caching policy for a single cache and BSA for a cache network as two universally well-working caching policies for this setting. Following this work, many other researchers propose universally well-working caching policies [10, 45].

1.1 Problem Statement

Universally well-working caching policies perform competently under any request distribution however in some cases conventional caching policies perform better. Therefore, the question becomes:

How can a universally well-working caching policy be improved in scenarios where conventional caching policies work better?

A conventional cache replacement cycle works as follows. A request comes to the cache. The cache reports either a hit or miss, and the caching policy determines



(a) **Conventional cache replacement cycle**

(b) **Cache replacement cycle with experts**

Figure 1.1: **Diagrams that show the cache replacement cycle for different types of caching policies.**

which objects to evict. This cycle is illustrated in Figure 1.1a.

The basic idea of this thesis is to add an overarching algorithm that selects different caching policies depending on their performance as seen in Figure 1.1b. However, this basic design requires some modifications to be practical.

1.2 Contributions

The contributions of this thesis are summarized as follows:

- **Caching simulator:** For this thesis a trace-driven simulator is built that can simulate caching policies with different request traces. The source code of this simulator is available online [31].
- **DEC caching policy for a single cache:** The caching policy designed in this thesis for a single cache situation with any request distribution setting is the DEC caching policy. It incorporates algorithms of the expert selection problem into a caching policy to dynamically select the best caching policy for the request distribution. The DEC policy always selects the most suitable caching policy in the simulations.
- **MultiDEC caching policy for caching networks:** For caching networks MultiDEC is designed. MultiDEC is a version of DEC adapted for caching networks and performs significantly better than BSA [39] in the simulations.
- **New versions of OGA and BSA:** Using some ideas of the DEC caching policy, versions of OGA and BSA are designed that only cache entire internet objects named Integral OGA and Integral BSA. The original OGA and BSA policies assume that objects can be cached as arbitrarily small chucks. However, this assumption is not valid in practice as some metadata is required for each chunk [45]. The newly designed Integral OGA and BSA exhibit similar performance compared to the original OGA and BSA in the trace-driven simulations.

1.3 Thesis Synopsis

Chapter 2 presents the background and related work. Chapter 3 describes the simulator that is used throughout this thesis. Chapter 4 presents the design of the

DEC caching policy. Following it, the DEC policy is adapted for caching networks in Chapter 5. Chapter 6 describes new versions of OGA and BSA that use a rounding step inspired by DEC. Lastly, the thesis ends with the conclusion and future work in Chapter 7.

Chapter 2

Background and Related Work

The chapter starts with the cache replacement problem in Section 2.1. The conventional caching policies are described in Section 2.2. The intelligent caching policies are discussed in Section 2.3. A couple of caching policies for a network of caches setting are described in Section 2.4. The expert selection problem is explained in Section 2.5. The algorithms for the expert selection problem are briefly discussed in Section 2.6. Some applications of the algorithms for expert selection problem are presented in Section 2.7. Lastly, some content popularity models are explained in Section 2.8.

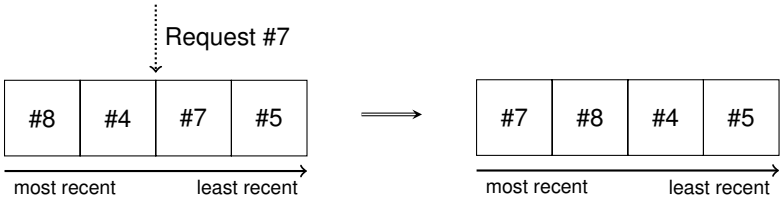
2.1 Cache Replacement Problem

2.1.1 Computer Caching

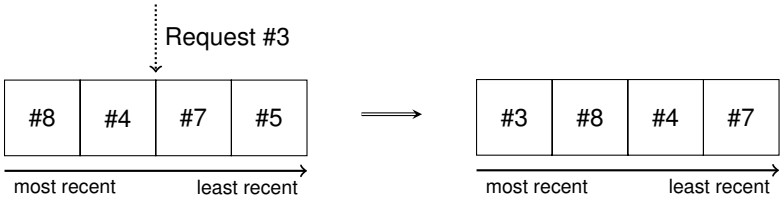
The memory hierarchy is fundamental in the computer architecture for the performance of the system [47]. Due to the faster operation of the CPU compared to the memory, small memory units operating near the speed of the CPU are used. This small memory unit, also known as the cache, holds a subset of the information in the memory. A cache hit happens when the CPU requests information that is available in the cache. It is called a cache miss when the information is not available in the cache. In this case, the information has to be requested from the memory. At a cache miss, the requested information is also stored inside the cache depending on the cache replacement policy. This exploits temporal locality, that is the fact that recently requested information is prone to be requested again in the future. An important aspect of caching is the cache replacement problem, that is, what to evict from the cache when the cache is full. For this, there are several caching policies available such as LRU and LFU.

2.1.2 Web Caching

Caching finds its way into web networks to counter the growing load on the internet [41]. This technique has the benefits of reducing the network bandwidth usage, user-perceived delays, and load on the origin server [19]. In the traditional web caching hierarchy, there are multiple proxy caches and reverse proxy caches between the user and the origin server [36]. Newer network architectures, such as the edge networks



(a) **An example of a cache hit with the LRU policy.**



(b) **An example of a cache miss with the LRU policy.**

Figure 2.1: **Examples that show the behavior of the LRU policy.**

[49], aim to place the caches closer to the user. These networks promise to elevate some of the burdens of video traffic becoming a significant portion of the total internet [38]. One of those architectures is the femtocaching architecture [44].

In the femtocaching architecture, there is the base station and multiple small cell access points referred to as helpers. These helpers can be placed much closer to the user and all have to small storage capacity for caching at the edge. This solves the issue of the backhaul capacity becoming the bottleneck as the number of helpers increases.

2.2 Conventional Cache Replacement Policies

There are multiple classifications of the caching policies available. Wong [50] classifies them as follows: recency-based policies, frequency-based policies, size-based policies, function-based policies, and randomized policies. In recent years, there is another class of caching replacement policies, that is, the intelligent cache replacement policies [6]. The following subsections will briefly discuss recency-based and frequency-based policies. The intelligent caching policies require their own section as the caching policy designed in this thesis is a part of this category.

2.2.1 Recency-based Policies

Recency-based policies use recency as the main factor in their replacement decision. The classical caching policy of this category is Least Recently Used (LRU). This policy evicts the object that is the least recently used in the cache when the cache is full.

An example of a cache hit is given Figure 2.1a to illustrate the behavior of LRU. Here, the group of blocks represents the cache. Inside each block, there is the object currently in the cache. When there is a request for object #7, there is a cache hit, and object #7 is moved to the most recent position in the cache. Figure 2.1b gives the example at a cache miss. The request for object #3 is not in the cache, therefore it is moved to the most recent position in the cache, and object #5 is evicted.

Many policies belong to the category of recency-based policies. A few examples are given here. The Pitkow/Recker policy [40] first removes objects based on the day. When the cache is full or at the end of the day, all the oldest objects, i.e., the objects with the same highest number of days between now and last requested, are evicted. If all the objects are within the same day as when the cache is full, then the object with the largest size is evicted. It thus uses recency based on the day for objects older than a day, and size otherwise.

LRU-MIN [2] incorporates the size of the requested object together with the recency. At a cache miss, it starts evicting objects in the cache that are in size the same as or larger than the requested object with the LRU rule. If there is still no place for the requested object in the cache, it evicts objects that are in size the same as or larger than half of the requested object. This happens until there is enough place in the cache for the requested object.

LRU-THOLD [2] modifies LRU with the rule that it only caches objects larger than a certain threshold. Even if there is room in the cache for the object, it is not cached if it is larger than the threshold.

2.2.2 Frequency-based Policies

Frequency-based policies use frequency as the main factor in their replacement decision. Least Frequently Used (LFU) is the classical caching policy in this class. LFU can be implemented either where only the requests of the objects in the cache are counted, In-Cache-LFU, or where all the requests of the whole catalog are counted, Perfect-LFU [13]. The In-Cache-LFU implementation requires less memory than Perfect-LFU.

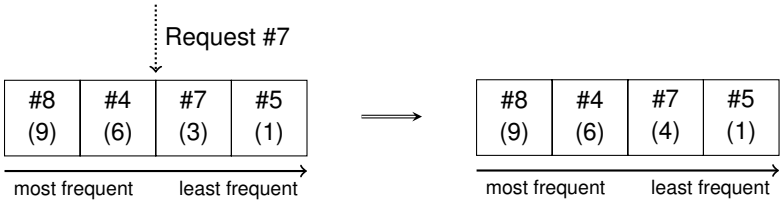
The behavior of LFU for both In-Cache-LFU and Perfect-LFU at a cache hit is given with an example in Figure 2.2a. The number in the parenthesis represents the count of the corresponding object, e.g., object #8 is requested nine times. After the request of object #7, there is a cache hit, and the count of object #7 is increased from 3 to 4. A cache miss occurs in Figure 2.2b. Object #3 is requested and is not in the cache. Therefore, the count of object #3 is increased and object #5 is evicted.

As seen, LFU behaves differently than LRU. Both LRU examples result in the requested object being in the highest position in the cache at that moment, i.e, all other objects have to be evicted before that object will. With LFU in the same situation, the position of the requested object does not change with the cache hit example or becomes the lowest position at the cache miss example.

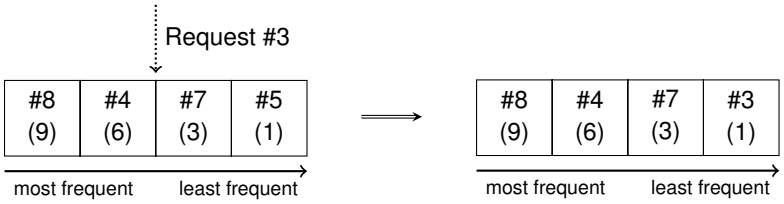
There are again various policies belonging to the frequency-based policy category. A few of them are described here. LFU-DA [8] incorporates a dynamic aging mechanism into LFU to reduce the problem of cache pollution. Cache pollution happens when very popular objects during a short period stay too long in the cache. The dynamic aging mechanism consists of an inflation factor L. In addition to increasing the count of the requested object, the inflation factor is added to the frequency value. The inflation factor L is updated to the frequency value of the evicted object each time an eviction happens.

The α-aging policy [51] uses a periodic linear aging function in addition to LFU. This function consists of the value $\alpha \in [0, 1]$. At each clock tick, the frequency counts are multiplied with α. Both α and the clock tick are tunable values. For example, [51] uses α of 0.9 and a clock tick every half an hour.

Window-LFU [30] is a more practical implementation of Perfect-LFU. It examines only the requests in a time window instead of all past requests. This makes it have



(a) **An example of a cache hit with the LFU policy.**



(b) **An example of a cache miss with the LFU policy.**

Figure 2.2: **Examples that show the behavior of the LFU policy.**

significantly less cost due to the shorter history measurements but still with similar cache hit rates.

2.3 Intelligent Cache Replacement Policies

The category of intelligent cache replacement policies consists of policies that leverage machine learning techniques into their cache replacement decision. The conventional caching policies are designed empirically. Therefore, the performances of those are depended on the request distribution. In the real world, there is rarely a stationary request distribution. Especially with the caching at the edge where the user base quickly changes and there are relatively few requests compared to proxy caches, there is a need for more sophisticated caching policies [39].

The intelligent cache replacement policies aim to work well with any request distribution and to adapt depending on the request sequence. The intelligent cache replacement policies can be divided further into two subcategories: the data-driven caching policies and the online convex optimization (OCO) based caching policies.

2.3.1 Data-driven Caching Policies

Data-driven caching policies use the large availability of web access logs for their design. Data-driven machine learning techniques have seen tremendous success in the fields of image recognition [28], speech recognition [1], robotics [4], and even surpassing humans in the game of Go [46]. As such, these techniques are also applied to the field of caching.

Calzarossa and Valli [15] propose caching policies based on fuzzy logic. In ordinary set theory an item can be for example either hot or cold, whereas, with fuzzy sets, the item can be for example 70% hot. Therefore, with fuzzy logic, more sophisticated rules can be implemented. Here, the size, time, and frequency of each web object is characterized using membership functions with labels as `LOW`, `MEDIUM`, or `HIGH`, etc. With this characterization, fuzzy conditional statements of `if-then` are designed.

These determine the probability of replacement RP. In the trace-driven simulations this policy results in more cache hits than LRU. However, it is questionable how well this policy works when the request distribution changes, as it probably requires a redesign of the fuzzy rules and membership functions.

Neural Network Proxy Cache Replacement (NNPCR) [17] uses a multi-layer feed-forward artificial neural network in the caching policy. The neural network is used to predict the likelihood of a future request for a web object, and the objects that are the least likely to get a future request are evicted. It consists of 3 inputs, 2 hidden layers, and 1 output. The inputs represent recency, frequency, and size. The authors experimented with various numbers of nodes on the hidden layer, varying between 1 and 10 nodes. Different configurations perform better depending on the cache size. However, in the simulations, the best configurations of NNPCR perform comparably to the more conventional LFU.

Ali and Shamsuddin [5] splits the cache into a short-term and a long-term cache. The short-term cache uses LRU. When an object in the short-term cache has been requested more than a certain threshold, it is moved to the long-term cache. In the long-term cache, an Adaptive Neuro-Fuzzy Inference System (ANFIS) is used. A Neuro-Fuzzy Inference System (NFIS) is a neural network that functions similarly to a fuzzy inference model. An Adaptive Neuro-Fuzzy Inference System adds the capability to extract the membership function parameters from a data set. When objects have to be evicted from the long-term cache, they are classified as either cachable or uncachable with the ANFIS. Uncachable here means that the object is unlikely to be requested within a specific time window. If the objects in the long-term cache are all cachable, or all uncachable, then LRU is used. With web traces, this policy can obtain a hit ratio higher than LRU and LFU.

Sadeghi et al. [42] use the reinforcement learning framework to learn the popularity dynamics of the user requests. The popularity dynamics are modeled as a Markov model. To deal with the unknown transition probabilities, a Q-learning caching algorithm is developed. With simulations, this Q-learning algorithm is shown to learn the optimal policy. However, this method required substantial memory and computation.

2.3.2 OCO-based Caching Policies

This category of caching policies uses the framework of Online Convex Optimization (OCO) [52] for the design of the caching policy. OCO differs from convex optimization in that the cost of each decision is not known beforehand.

The online convex optimization framework, or more generally, the online learning framework has been applied successfully in other fields, such as spam filtering [43], online routing [9], and portfolio selection [33]. In recent years, OCO is also applied to the caching problem.

Paschos et al. [39] cast the caching problem as an Online Linear Optimization (OLO) problem. Here, they do not assume the request distribution and look for a caching policy that works well for any request distribution, even one where an adversary requests objects that aim to degrade the system's performance. In this case, the hit rate metric is not as useful as an adversary can always request objects that are not in the cache. Therefore, a new metric is introduced, which is regret. With the regret metric, the aim is to optimize the performance against the best static cache configuration in hindsight. For this new metric, Paschos et al. prove a lower bound for

any caching policy. They also introduce the Online Gradient Ascent (OGA) policy that matches this regret lower bound for a single cache situation.

Bhattacharjee et al. [10] improve on the lower bound of [39] by deriving a tight sub-linear regret lower bound for both the single cache situation and the bipartite caching network. They also propose a new caching policy named Follow the Perturbed Leader (FTPL). FTPL works similar to Perfect-LFU, with the addition of some noise added to the frequency counts at the replacement decision. This noise is to ensure that the algorithm works well in any situation, as it is possible to have request sequences that ensure zero hits with Perfect-LFU. This FTPL policy has the benefit compared to OGA of [39] that it works with whole files in the cache instead of the partial files of OGA and has near-optimal regret.

Si Salem et al. [45] use the Online Mirror Descent (OMD) algorithm for their caching policy. They show the OGA algorithm of [39] as a special instance of the OMD algorithm and prove that OMD has no time-averaged regret.

2.4 Cache Replacement Policies for a Network of Caches

Caching in a network of caches is a more challenging setting than with a single cache. In a network of caches, the individual caches can overlap in range with each other. Therefore, one then has to decide how to route a request to the caches as there can be multiple caches capable of serving the request. Furthermore, the diversity of objects in the caches is also important due to this overlap as the more diverse the caches are the higher chance a user in this overlap gets a cache hit. Moreover, in the network of caches, it can more beneficial to select one cache above another. The cache can have benefits that range from bandwidth economization, Quality of Service improvement, shorter transmission range, etc. This benefit of one cache compared to another cache can be represented as the utility of the cache where the higher the utility the more beneficial it is to use the cache.

Giovanidis and Avranas [22] extent the LRU caching policy for the network of caches. Here, they propose the mLRU caching policy. The mLRU caching policy has different variants which perform better depending on the traffic model. One of those variants works as follows. Each cache updates according to the LRU caching policy. A request is served by the cache with the highest utility. When no caches have the requested object, a single cache is selected to store this requested object. This selection can happen with different criteria or for instance at random. When the requested object is in multiple caches, all caches that contain this object are updated with the information of the request. The mLRU policy is compared to the normal LRU rule without this sophisticated update rule and is shown to have a relative gain in hit ratio up to 70%.

Leonardi and Neglia [32] propose an LRU based caching policy that is very similar to mLRU with the key difference in how the individual caches are updated. The designed LazyLRU only updates the individual caches when there is exactly one replica of the requested object in the caching network. With a network of cache in a realistic configuration based on a T-Mobile network in Berlin and a real content request trace from Akamai Content Delivery Network, the authors show that the LazyLRU can have up to 20% higher hit ratio than the mLRU policy.

Paschos et al. [39] present the Bipartite Supergradient Algorithm (BSA) for the network of caches. It uses the OGA policy for the individual caches and updates them using a supergradient. The routing happens according to the maximization of the utility. In practice, this means that the highest utility caches containing the reques-

ted object are selected to serve the request. In the simulations, they compare BSA with mLRU and LazyLRU and show that BSA can outperform the best heuristic in the simulations by 45.8%.

2.5 Expert Selection Problem

In the expert selection problem [34], N_e experts give advice. One has to choose which advice of the experts to follow. After choosing, the cost of that advice is revealed and the loss occurred. The objective is to make as few mistakes as the best expert in hindsight.

The expert selection problem is modeled as follows. The advice/prediction is represented by $a \in \mathcal{A}$ with $\mathcal{A}$ the set of all possible advice. The weight of an expert i is described by $W(i)$. The set $S(a)$ contains all experts that give the same advice a. The total number of experts is N_e. The cumulative number of mistakes at time T of expert i is denoted as $M_T(i)$. This notation is based on the notation of [27].

2.6 Expert Selection Problem Algorithms

There are many existing algorithms for the expert selection problem. A few of them are described in the following subsections.

2.6.1 Weighted Majority

The Weighted Majority (WM) [34] algorithm works as follows. Each expert makes a prediction. For each possible prediction, all the weights of the experts with that same prediction are summed up. The prediction with the highest cumulative weights is chosen for that iteration. After the prediction, the loss is revealed and the weights of the experts with the incorrect prediction are decreased. This happens according to (2.1) [27]. The full algorithm is described in Algorithm 1.

$$W_{t+1}(i) = \begin{cases} W_t(i) & \text{if expert } i \text{ was correct} \\ W_t(i)(1-\epsilon) & \text{if expert } i \text{ was incorrect} \end{cases} \tag{2.1}$$

Algorithm 1: Weighted Majority

for $t = 1 \dots T$ **do**
  $a_t = \arg\max_{a \in \mathcal{A}} \sum_{i \in S_t(a)} W_t(i)$;
  Loss revealed;
  foreach *expert* i **do**
    $W_{t+1}(i) = \begin{cases} W_t(i) & \text{if expert } i \text{ was correct} \\ W_t(i)(1-\epsilon) & \text{if expert } i \text{ was incorrect} \end{cases}$
  end
end

2.6.2 Randomized Weighted Majority

The Randomized Weighted majority (RWM) [34] algorithm is a modification of WM. Instead of deterministically choosing which advice to follow by the weighted majority, the algorithm chooses based on a probability distribution as (2.2). The weight update step is the same as with WM as seen in Algorithm 2.

$$p_t(i) = \frac{W_t(i)}{\sum_{k=1}^{N_e} W_t(k)} \tag{2.2}$$

Algorithm 2: Randomized Weighted Majority

for $t = 1 \dots T$ **do**
  $i_t = i$ with probability $p_t(i) = \frac{W_t(i)}{\sum_{k=1}^{N_e} W_t(k)}$;
  Loss revealed;
  foreach *expert* i **do**
    $W_{t+1}(i) = \begin{cases} W_t(i) & \text{if expert } i \text{ was correct} \\ W_t(i)(1-\epsilon) & \text{if expert } i \text{ was incorrect} \end{cases}$
  end
end

2.6.3 Hedge

The Hedge algorithm [20] is a more general version of RWM. Here, when an expert is incorrect, a loss occurs. This loss can be continuous, unlike RWM and WM where the loss is discrete. With this change, the weights are updated differently according to (2.3) [27] and the algorithm can be found in Algorithm 3.

$$W_{t+1}(i) = W_t(i)e^{-\epsilon l_t(i)} \tag{2.3}$$

Algorithm 3: Hedge

for $t = 1 \dots T$ **do**
  $i_t = i$ with probability $p_t(i) = \frac{W_t(i)}{\sum_{k=1}^{N_e} W_t(k)}$;
  Loss l_t revealed;
  foreach *expert* i **do**
    $W_{t+1}(i) = W_t(i)e^{-\epsilon l_t(i)}$;
  end
end

2.6.4 Follow the Leader

The Follow the Leader (FTL) algorithm for the expert problem selects the experts with the fewest mistakes. The algorithm is in Algorithm 4.

Algorithm 4: Follow the Leader

for $t = 1 \dots T$ **do**
  $i_t = \arg\min_i M_t(i)$;
  Loss revealed;
  foreach *expert* i **do**
    $$M_{t+1}(i) = \begin{cases} M_t(i) & \text{if expert } i \text{ was correct} \\ M_t(i) + 1 & \text{if expert } i \text{ was incorrect} \end{cases}$$
  end
end

2.6.5 Follow the Perturbed Leader

The downside of the FTL algorithm is that it does not fair well in an adversarial situation, that is, when an adversary deliberately wants to degrade the performance, then it is possible to have 100% mistakes. This is due to the deterministic behavior of FTL. The Follow the Perturbed Leader (FTPL) [18] aims to counter that by introducing some noise to the expert selection. The FTPL algorithms is in Algorithm 5.

Note that the FTPL algorithm is quite general and is here applied for the expert selection problem. FTPL is also adapted as a caching policy [10].

Algorithm 5: Follow the Perturbed Leader

for $t = 1 \dots T$ **do**
  $i_t = \arg\min_i M_t(i) + \mathcal{N}(0, 1)$;
  Loss revealed;
  foreach *expert* i **do**
    $$M_{t+1}(i) = \begin{cases} M_t(i) & \text{if expert } i \text{ was correct} \\ M_t(i) + 1 & \text{if expert } i \text{ was incorrect} \end{cases}$$
  end
end

2.6.6 Shrinking Dartboard

Geulen et al. [21] notice that RWM is not suitable for online buffering problems due to too many switching between experts. Therefore, they introduce the Shrinking Dartboard (SD) algorithm that aims to reduce the number of switches while having the same regret bounds as RWM. This is accomplished by only having a chance to switch when the weight of the currently chosen expert is lowered which significantly reduces the number of switching between experts. However, the chance to choose an expert remains the same which is proven by the authors. The pseudocode of SD is in Algorithm 6.

2.6.7 Mixing Schemes

Bousquet and Warmuth [11] notice long recovery times for experts in situations where bad performing experts suddenly perform better. This long recovery time is due to the

Algorithm 6: Shrinking Dartboard

for $t = 1 \dots T$ **do**
  $i_t = i_{t-1}$ with probability $\frac{W_t}{W_{t-1}}$ else $i_t = i$ with probability $p_t(i) = \frac{W_t(i)}{\sum_{k=1}^{N_e} W_t(k)}$;
  Loss revealed;
  foreach *expert* i **do**
    $$W_{t+1}(i) = \begin{cases} W_t(i) & \text{if expert } i \text{ was correct} \\ W_t(i)(1-\epsilon) & \text{if expert } i \text{ was incorrect} \end{cases}$$
  end
end

multiplicative updates which cause the weight to become very small. They propose a set of schemes, the mixing schemes, to prevent those weights to become too small too quickly. The mixing update of these schemes happens after the loss update according to (2.4). The authors propose the following mixing schemes, Fixed-Share To Start Vector, Fixed-Share to Uniform Past, Fixed-Share to Decaying Past. The idea of these schemes is to combine the current weight vector with some small part of another weight vector. Fixed-share To Start Vector uses the starting weight vector. Fixed-Share to Uniform Past uses the past averaged weight vector. Fixed-Share to Decaying Past scheme uses a weighted past average weight vector where recent weight vectors contribute more than past weight vectors.

$$\boldsymbol{W}_{t+1} = (1-\alpha)\boldsymbol{W}_t + \alpha \boldsymbol{V}_{t-1} \tag{2.4}$$

where $\boldsymbol{V}$ is either the starting weight vector, past averaged weight vector, or the decaying past weight vector, and α is the mixing parameter.

2.7 Experts Framework Applications

2.7.1 Adaptive Caching Using Multiple Experts

The Adaptive Caching Using Multiple Experts (ACME) [7] algorithm is a proposal for a caching policy where that uses multiple caching policies as experts. In ACME, the experts vote on which objects to keep, and based on the weighted votes of all experts, the objects to keep in the cache are decided. For the weight adjustments, the experts are simulated with the requested object and the weights are lowered depending on the resulting simulated cache miss or hit. ACME is implemented in [23] wherein simulations it is shown that it can adapt to the best expert. However, there is no known regret bound for ACME as ACME does not use a standard Experts framework algorithm for expert selection. This is in contrast to the DEC policy designed in this thesis, where the algorithms of the Experts framework can be used. Therefore, the DEC policy has to benefit that it can be improved when a newer algorithm of the Experts framework is developed.

2.7.2 Experts for Hard Disk control

Helmbold et al. [29] apply the Experts framework to the problem of spinning down the hard disk. Mobile computing devices spin down the hard disk to save energy consumption. However, spinning the hard disk back up consumes a lot of energy. Therefore, there is the problem of when to spin the hard disk down as spinning it down too early can lead to frequent power state switches. This in turn might result in more energy consumption compared to not spinning down the hard disk. The algorithm designed in [29] uses fixed time-outs as experts. The time-outs determine how long the hard disk has to be idle before spinning it down. The time-out that is used is the weighted average of the experts' time-outs. The loss is calculated by comparing the predictions of the experts to the optimal prediction. The weights are updated using the Hedge loss update. This algorithm can perform better than the best fixed-time out and wastes on average only 88% of the energy that is wasted by the best-fixed time-out.

2.7.3 File Prediction Using Multiple Experts

Brandt [12] proposes an algorithm that uses the Experts framework to do file prefetching. The pool of experts consists of file prefetch algorithms that predict the successor for a file. There is also a special null expert that advocates to not prefetch. This expert is useful when the prediction accuracy of the other experts is low. The expert that is chosen to follow happens according to the FTL algorithm for experts. The weights are adjusted based according to the correctness of the prediction. The null expert is correct when none of the experts made a correct prediction. This algorithm can predict correctly up to 98% of the time in the simulations.

2.8 Content Popularity Models

For modeling the content distribution and the request sequences there are many different models available. Content is defined in [37] as a piece of reusable information, e.g, videos files, music files, web pages, etc.

2.8.1 Zipf's Law

A popular model for content distribution is a Zipf-like distribution. Zipf's law states that the probability of requesting the n'th most popular object is inversely proportional to its rank in the frequency table [14]. This is expressed as $p_n \propto n^{-1}$ in [37]

For example, the probability of requesting the second most popular object is about half of the most popular object as seen in Figure 2.3.

The Zipf-like distribution, also called power-law distribution, differs in the exponent as seen in (2.5) ((2.1) in [37]). Depending on the application there are different estimates for the parameter τ. For example, for web traffic τ is estimated to be between 0.64 and 0.83 [14], for Youtube video traffic to be between 0.70 and 0.85 [48], and for Wikipedia pages to be between 0.50 and 0.75 [26].

$$p_n = \frac{n^{-\tau}}{\sum_{j=1}^{N} j^{-\tau}}, \quad n = 1 \dots N \tag{2.5}$$

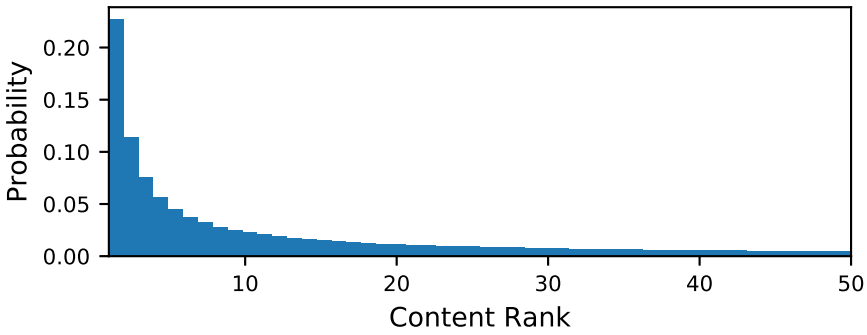


Figure 2.3: **An example that shows the probability distribution of the Zipf's law. The most common object is twice as likely as the second most common object.**

2.8.2 Independent Reference Model

The independent reference model (IRM) is a model for the request sequence [14]. Each request is drawn independently from a power-law distribution. This model is widely used although it has some limitations. One limitation is the treatment of every request being independent. Therefore, the common phenomenon of temporal locality cannot be captured using this model. Another limitation is the assumption that the popularity of content does not change over time which is not the case in the real world.

2.8.3 Random Replacement Model

The random replacement model (RRM) modifies IRM with an additional step where according to a replacement rate a randomly chosen object is swapped with an object of lower popularity for all subsequent requests. For example, with a replacement rate of 4, approximately every 4 requests, an object is randomly chosen, e.g, object #80, then all future requests of #80 are swapped with a request for another randomly chosen object, e.g., object #75, and vice versa. This additional step aims to model the varying content popularity.

Chapter 3

Simulator

This chapter describes the simulator that is used for trace-driven simulations. The simulator is built specifically for this thesis to simulate and compare caching policies with various traces. Some caching policies described in Chapter 2 are also simulated. The results show that the universally well-working caching policies do not always perform better than the conventional caching policies.

3.1 Implementation

The trace-driven simulator is written in Python 3. Each caching policy is implemented as a separate class. A trace consists of an array of integers. Each integer represents a request with the corresponding integer as the object id. A simulation starts with generating an initial cache. This initial cache consists of random objects and is used as the initial cache for all the caching policies. After generating the initial cache, a request for an object is simulated with the trace together with the behavior of each caching policy. This cycle continues until there are no requests left in the trace. The simulator is available publicly in [31].

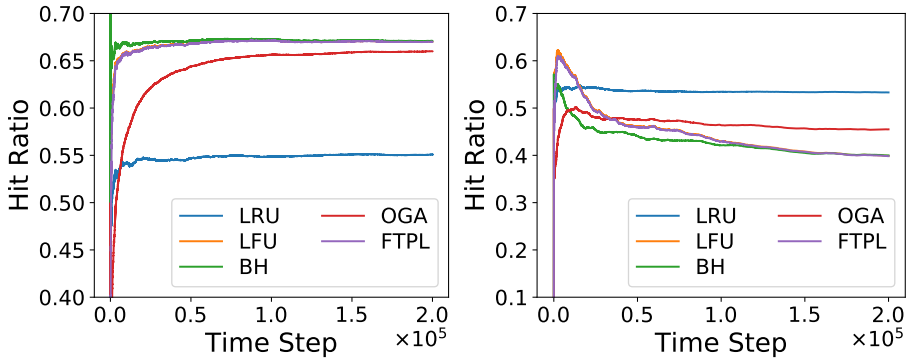
3.2 Traces

The following traces are used for the simulations. It consists of two synthetic traces and two traces from real-world applications. The relative cache size is 30% for all traces.

Independent Reference Model (IRM) The IRM trace has a power-law exponent of 0.8, a sample size of 2000000, and a catalog size of 100.

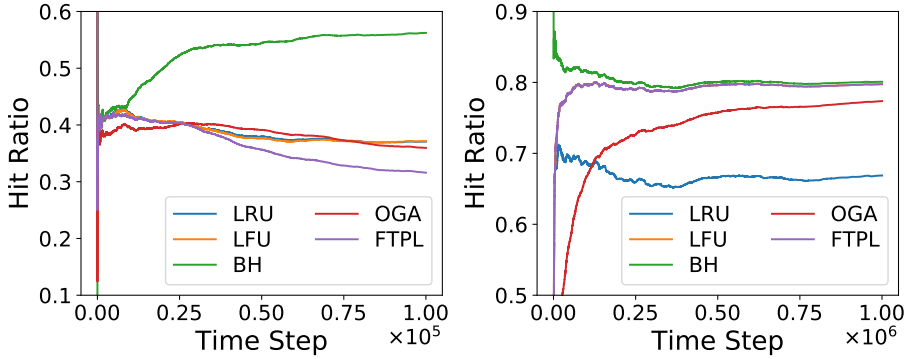
Random Replacement Model (RRM) The RRM trace has a power-law exponent of 0.8, a sample size of 2000000, a catalog size of 100, and a replacement rate of 4.

YouTube The YouTube trace from [39] contains requests for various YouTube videos. This dataset consists of 100000 samples and has a catalog size of 62538.



(a) **IRM** (b) **RRM**

Figure 3.1: **Simulations with various caching policies on synthetic traces.**



(a) **YouTube** (b) **MovieLens**

Figure 3.2: **Simulations with various caching policies on real-world traces.**

MovieLens The MovieLens trace use the MovieLens 1M dataset [24, 25]. This benchmark is also used in [10]. There are 1000209 samples in this trace with a catalog size of 3953.

3.3 Results

In this section, a selected few caching policies are simulated. These caching policies consist of LRU and LFU as the conventional caching policies. LFU is implemented as the Perfect-LFU variant. For the intelligent caching policies based on the OCO framework, FTPL and OGA are chosen for the reason that these caching policies aim to work well in any situation.

The results of the simulations are plotted in Figure 3.1 and 3.2. It can be observed that the caching policies based on the OCO framework, i.e, FTPL and OGA, do not always perform the best. This is the most notable in the RRM trace, where LRU has a significantly better hit ratio. Therefore, there is room for another caching policy with a hit ratio closer to the conventional caching policies while still working well with any request sequence.

Chapter 4

Dynamic Expert Caching Policy

In this chapter, a new caching policy is introduced named Dynamic Expert Caching policy or DEC for short. DEC incorporates the Experts framework algorithms into a caching policy. It consists of an overarching algorithm that dynamically selects an expert based on its current performance. Each expert represents a different caching policy. With it, DEC can be easily extended and improved with newer caching policies to achieve better hit rates in any situation. In contrast to previous works, DEC uses the algorithms from the Experts framework directly such that those algorithms can also be easily integrated into DEC when newer algorithms of the Experts framework are introduced. Another benefit is that the performance guarantees of the Experts framework algorithms within DEC are still valid and this results in DEC having no time-averaged regret similar to OGA and FTPL. Additionally, there is an optional mixing step within DEC that can improve the performance in situations where different caching policies perform better depending on the section of the trace. This step has the downside of voiding the regret bounds, however, DEC with mixing still has good performance in the simulations.

This chapter starts with the system model and the performance metrics in Section 4.1 and 4.2. Next, the design of the DEC policy is described in Section 4.3. Following it, the theory on the performance guarantees of DEC are derived in Section 4.4, and the results of simulations are shown in Section 4.5. Lastly, the mixing step is introduced and discussed in Section 4.6.

4.1 System Model

The single cache model consists of a local cache connected to an origin server. The origin server has access to the entire library of size N_f. The local cache has capacity C with $C \ll N_f$. The objects in the library are all assumed to be uniform in size.

At each time slot t a request arrives. With each request, at most 1 object can be requested. The request is modeled as a one-hot encoded vector $\boldsymbol{x}_t$ with an 1 if the corresponding object is requested and 0 otherwise. This is represented in the following set.

$$\mathcal{X} = \left\{ \boldsymbol{x} \in \{0,1\}^{N_f} | \mathbb{1}^\mathsf{T} \boldsymbol{x} = 1 \right\} \tag{4.1}$$

where $\mathbb{1}$ is the vector of all ones.

The cache configuration is modeled as the vector $\boldsymbol{y}_t$. The cache can either store entire objects or partial objects. The cache configuration vectors that only store entire objects are drawn from the following set.

$$\mathcal{Y} = \left\{ \boldsymbol{y} \in \{0,1\}^{N_f} | \mathbb{1}^\mathsf{T} \boldsymbol{y} \leq C \right\} \tag{4.2}$$

where C is the cache size.

The cache configuration vectors that store partial objects are drawn from the following set.

$$\mathcal{Y}^c = \left\{ \boldsymbol{y}^c \in [0,1]^{N_f} | \mathbb{1}^\mathsf{T} \boldsymbol{y}^c \leq C \right\} \tag{4.3}$$

The single cache model is modeled very similarly as in [39].

4.2 Performance Metrics

4.2.1 Hit Rate

A classical performance metric is the average hit rate. The total number of hits, i.e, times that a requested object is found in the cache, can be calculated using (4.4). The average hit rate or hit ratio is obtained by dividing this by the total time elapsed T.

$$f_T(\{\boldsymbol{x}_t\}_1^T, \{\boldsymbol{y}_t\}_1^T) = \sum_{t=1}^{T} \boldsymbol{x}_t^\mathsf{T} \boldsymbol{y}_t \tag{4.4}$$

4.2.2 Regret

In the model-free setting, it is possible for an adversary to specifically request objects that are not in the cache. Therefore, for any caching policy, it is possible to have a hit rate of zero. The regret metric, introduced to the caching problem by [39], aims to provide a meaningful performance metric even in this scenario. It measures the performance of the caching policy against the best static caching configuration in hindsight as seen in (4.5).

$$R_T(\{\boldsymbol{x}_t\}_1^T, \{\boldsymbol{y}_t\}_1^T) = \mathbb{E}\left[\sum_{t=1}^{T} f_t(\boldsymbol{x}_t, \boldsymbol{y}^*) - \sum_{t=1}^{T} f_t(\boldsymbol{x}_t, \boldsymbol{y}_t) \right] \tag{4.5}$$

where $f_t(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{x}^\mathsf{T} \boldsymbol{y}$ and $\boldsymbol{y}^* = \arg\max_{\boldsymbol{y} \in \mathcal{Y}} \sum_{t=1}^{T} f_t(\boldsymbol{x}_t, \boldsymbol{y})$.

4.3 Design

The design of the DEC policy uses the basic premise of an overarching algorithm that dynamically selects a caching policy depending on its performance as described in Figure 1.1b. It utilizes an expert selecting algorithm of the Experts framework as the overarching algorithm with each expert representing a different caching policy. The following subsections describe the design of each part in more detail.

4.3.1 Expert Design

The caching policies are used as experts in the following way. A caching policy can be seen as a function that gives a ranking to each object. Objects with a high rank are kept in the cache whereas objects with a lower rank are discarded. The advice of each expert is thus the ranking of the objects.

The only requirement for the ranking is that it has order. The scale of the values does not matter as in this design the rank of the objects is only compared with each other and not with the rankings of the other experts.

4.3.2 Loss Function

The loss of each expert is calculated when a request for an object comes. For each expert, the rank of this requested object is looked up. A loss of 1 occurs when the rank of the requested object is not within the top τ_r highest rank objects and 0 otherwise. This τ_r is a tunable parameter, and if it is set to the cache size, then the loss resembles a cache miss.

4.3.3 Expert Caching Policy Selection

The caching policies that are chosen to be experts are LRU, FTPL, and OGA. This is because LRU gives a ranking based on recency, FTPL gives a ranking based on frequency, and lastly, OGA combines both frequency and recency. Note, however, that by viewing caching policies as algorithms that only give a ranking, the expert OGA is different that the caching policy OGA. The details are expanded upon in Chapter 6, however, the summary is that by only caching the top τ_r objects that OGA considers the most important, the OGA policy is discretized such that it becomes the Integral OGA caching policy of Chapter 6. Therefore, the experts actually consist of LRU, FTPL, and Integral OGA.

4.3.4 Overarching Algorithm Selection

In this design of incorporating the Experts framework into a caching policy, all the algorithms given in Section 2.6 except WM can be applied in this design as WM combines the advice of the experts. Therefore the available Experts framework algorithms suitable for DEC are:

- Follow The Leader (FTL)
- Follow The Perturbed Leader (FTPL)
- Randomized Weighted Majority (RWM)
- Hedge
- Shrinking Dartboard (SD)

As explained in Section 2.6, the main benefit of Hedge compared to RWM is that it works with continuous losses. However, in this design, the loss is discrete and either 1 or 0. Therefore, in this situation, both algorithms can give the same decrease in weights if the parameter ϵ is tuned for it, and thus are equivalent.

The choice in algorithms is hence between FTL, FTPL, RWM, and SD. An issue with only selecting the advice of one expert is too much switching between experts which can result in a policy that performs worse than either one of the experts. This is illustrated in the following situation. The trace that is used follows a zigzag pattern

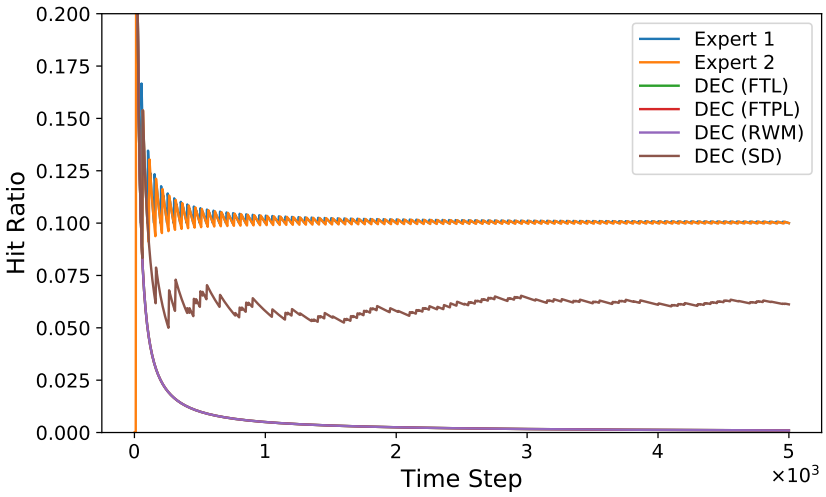
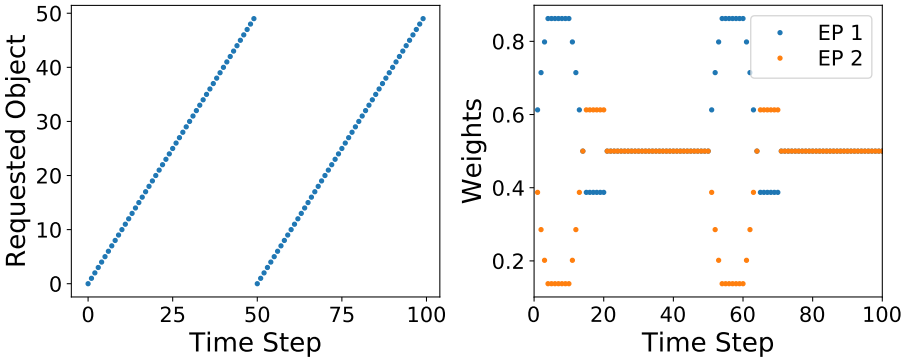


Figure 4.1: **A comparison between the different expert selection algorithms with an adversary trace. The best in hindsight solution is to stick with either one of the experts and not switch between experts. The FTL, FTPL, and RWM algorithms all switch too often leading to a very low hit ratio. The SD algorithm makes fewer switches and thus is closer to the best in hindsight solution.**

and can be found in Figure 4.2a. The experts consist of two Integral OGA caching policies with different initial caches, one has its initial cache consisting of objects $\{0, 1, 2, 3, 20\}$, and in the other one $\{10, 11, 12, 13, 14\}$. With this particular trace, Integral OGA prefers to keep its initial cache configuration and evicts everything else with a rule that is similar to LRU. The two Integral OGA experts will be referred to as Expert 1 and Expert 2. The result of a simulation of this configuration can be found in Figure 4.1.

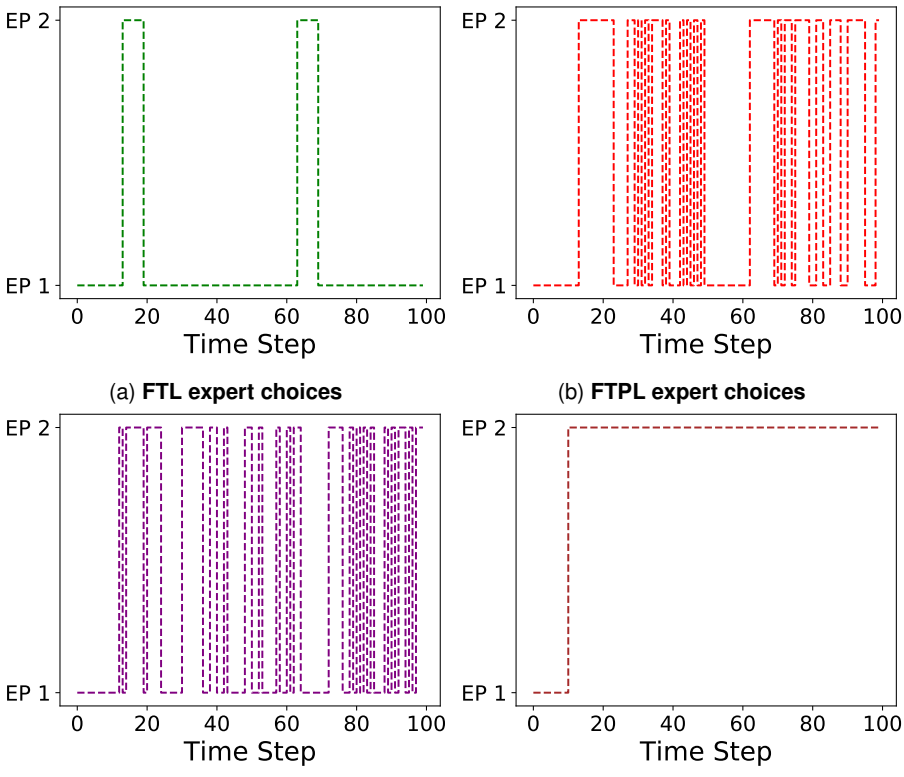
FTL, FTPL, and RWM all perform the same with an average hit rate approaching zero. This is because those algorithms switch too often between the two experts. SD performs better but not as good as a best in hindsight solution which is to stick with either Expert 1 or Expert 2.

The weights of the experts are plotted for the first 100 time steps in Figure 4.2b and are the same for all algorithms. The weights only change when either Expert 1 or Expert 2 have a hit. This happens for Expert 1 at requests $\{0, 1, 2, 3, 20\}$ and for Expert 2 at requests $\{10, 11, 12, 13, 14\}$. FTL always switches to the expert with the largest weight as seen in Figure 4.3a. In this scenario, FTL chooses Expert 1 when Expert 2 gets hits and Expert 2 when Expert 1 gets hits. Therefore, apart from the initial cache hits, there are zero hits. FTPL adds some noise to the weights, however, this causes many expert switches when the weights are equal as visible in Figure 4.3b. With RWM, the weights of both experts are very close to each other and thus it frequently switches between experts as seen in Figure 4.3c. This again results in very few hits. SD only has a chance to switch experts when the weights of the currently chosen expert decrease. Therefore, it does not switch unnecessary as seen in Figure 4.3d. For this reason, SD is chosen as the overarching algorithm in DEC.



(a) **Adversary trace** (b) **Expert weights**

Figure 4.2: **Adversary trace and the weights of the experts for the first 100 requests. The weights are the same for the different Experts framework algorithms.**



(a) **FTL expert choices** (b) **FTPL expert choices**

(c) **RWM expert choices** (d) **SD expert choices**

Figure 4.3: **The choices of the various Experts framework algorithms. EP is a shorthand notation for expert. The SD algorithm switches fewer than the other algorithms.**

4.3.5 Design Summary

A cycle in the DEC policy works as follows. When a request for an object comes, the internal values and the loss of each expert are updated. Next, an expert is selected using the Shrinking Dartboard algorithm if there is a cache miss. The ranking of this expert is used to determine whether or not to cache the requested object and which object to evict. Then, the cycle repeats. A diagram of this cycle is in Figure 4.4. The pseudocode of the DEC caching policy is in Algorithm 7.

Algorithm 7: Psuedocode of the DEC policy

foreach *Incoming object request* r **do**
  hit = true if r in cache else false;
  foreach *Expert* i **do**
    $$W_{t+1}(i) = \begin{cases} W_t(i) & \text{if Expert } i \text{ was correct} \\ W_t(i)(1-\epsilon) & \text{if Expert } i \text{ was incorrect} \end{cases};$$
    Update Expert i with request r;
  end
  Normalize weights;
  (Optional) Mixing update;
  if *Cache miss* **then**
    Select Expert e according to Shrinking Dartboard;
    Get ranking from Expert e;
    Lookup the ranking of each object in the cache;
    if *ranking of request* r *>= min(cache ranking)* **then**
      Replace object with the smallest ranking with request r;
    end
  end
end

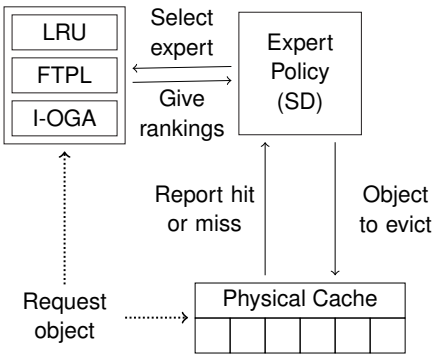


Figure 4.4: **A diagram that shows a cache replacement cycle of the DEC policy.**

4.4 Theory

For the DEC policy, the regret bound is derived as follows. Lemma 1 from [21] states that the probability of choosing an expert i with the Shrinking Dartboard algorithm is the same as with RWM. Therefore, the well-known bound on the expected number of mistakes in (4.6) is also the same. The DEC uses the Shrinking Dartboard algorithm without any modifications and thus also shares this bound.

$$\mathbb{E}[M_T] \leq (1+\varepsilon)M_T(i) + \frac{\ln N_e}{\varepsilon} \qquad \text{for any expert } i \tag{4.6}$$

where $M_T(i)$ is the cumulative number of cache misses at time T for expert i.

To obtain the regret bound of the DEC caching policy, the cumulative number of cache misses is rewritten into H_T, the cumulative number of cache hits at time T with (4.7). Therefore, the expected number of cache hits for the DEC policy is in (4.9).

$$H_T = T - M_T \tag{4.7}$$

$$T - \mathbb{E}[H_T] \leq (1+\varepsilon)(T - H_T(i)) + \frac{\ln N_e}{\varepsilon} \qquad \text{for any expert } i \tag{4.8}$$

$$\mathbb{E}[H_T] \geq (1+\varepsilon)H_T(i) - \varepsilon T - \frac{\ln N_e}{\varepsilon} \qquad \text{for any expert } i \tag{4.9}$$

Suppose the regret of an expert i is bounded a value B as in (4.10). Plugging (4.10) into (4.9) and simplifying it gives (4.13).

$$R_T(i) = H_T^* - H_T(i) \leq B \tag{4.10}$$

$$H_T(i) \geq H_T^* - B \tag{4.11}$$

where H_T^* is the cumulative number of hits at time T of the best static cache configuration in hindsight.

$$\mathbb{E}[H_T] \geq (1+\varepsilon)H_T(i) - \varepsilon T - \frac{\ln N_e}{\varepsilon} \geq (1+\varepsilon)(H_T^* - B) - \varepsilon T - \frac{\ln N_e}{\varepsilon} \tag{4.12}$$

$$R_T(\mathsf{SD}) = H_T^* - \mathbb{E}[H_T] \leq (1+\varepsilon)B + \varepsilon T + \frac{\ln N_e}{\varepsilon} \tag{4.13}$$

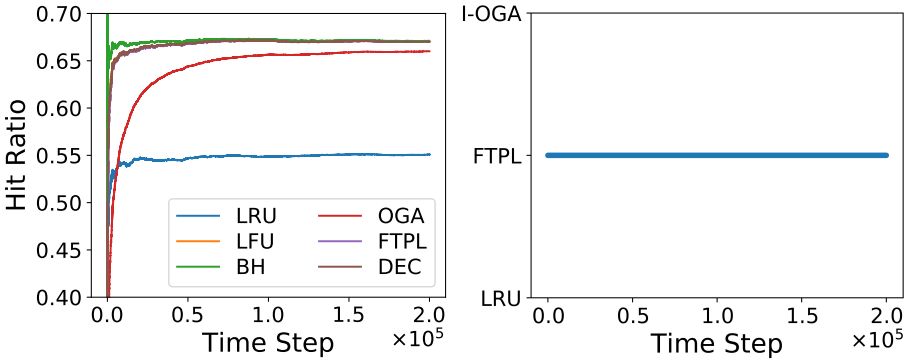
Of the three experts, LRU, FTPL, and Integral OGA, only FTPL [10] has a known regret bound in (4.14).

$$R_T(\mathsf{FTPL}) \leq 1.51(\ln N_f)^{\frac{1}{4}}\sqrt{CT} \tag{4.14}$$

Therefore, B can be substituted with this bound, and the regret bound of the DEC caching policy becomes (4.15).

$$R_T(\mathsf{DEC}) \leq (1+\varepsilon)(1.51(\ln N_f)^{\frac{1}{4}}\sqrt{CT}) + \varepsilon T + \frac{\ln N_e}{\varepsilon} \tag{4.15}$$

Taking the learning rate ε to be $\frac{1}{\sqrt{T}}$ the regret becomes (4.16). Therefore, this policy suffers sublinear regret as $\lim_{T\to\infty} R_T(\mathsf{DEC})/T = 0$ which is similar to FTPL and OGA.



(a) **Simulation Results** (b) **Expert Choices**

Figure 4.5: **A comparison between the DEC policy and other caching policies on the IRM trace. The DEC policy chooses the FTPL policy to follow which results in similar hit ratios as the frequency-based caching policies, LFU and FTPL.**

$$R_T(\text{DEC}) \leq (1+\sqrt{T})(1.51(\ln N_f)^{\frac{1}{4}}\sqrt{C}) + (1+\ln N_e)\sqrt{T} \tag{4.16}$$

4.5 Results

4.5.1 Experiment Setup

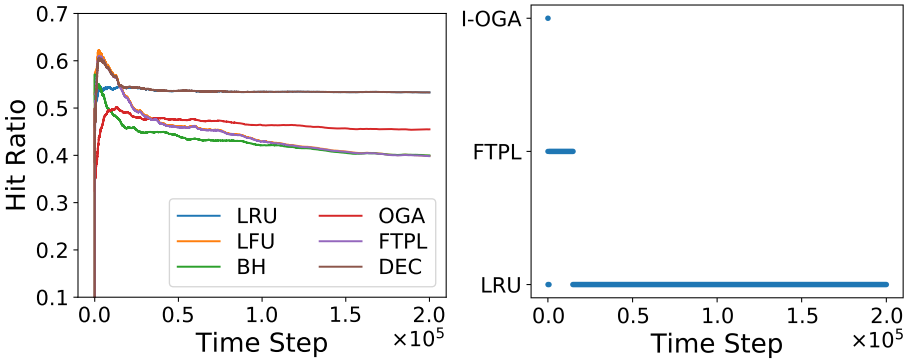
The DEC policy is simulated to compare with the existing caching policies. These consist of LRU, LFU, best static cache in hindsight (BH), OGA, and FTPL. All simulations use the same experimental setup as in Chapter 3. The learning rates for the policies that have one are set to values that minimize their regret. DEC is also compared to ACME, however, the performance gap is very small and is left out of the figures below for clarity's sake. Also, it is not the intention of this thesis to improve upon the previous design of combining Experts framework algorithms with caching but rather to introduce a different design such that the theory of the Experts framework is still valid. Nevertheless, the comparison with ACME together with the full-size figures of all the single cache simulations can be found in Appendix A.

4.5.2 IRM Trace

The results from the IRM trace together with the expert choices are plotted in Figure 4.5. LFU, FTPL, and the DEC all have similar hit ratios and approach the best static cache in hindsight. OGA also seems to perform well and the worst hit rates are with LRU. The DEC correctly selects FTPL as the best expert to follow as seen in Figure 4.5b.

4.5.3 RRM Trace

With the RRM trace in Figure 4.5, LFU and FTPL initially perform the best. However, after a short period, there are more random replacements that degrade the perform-



(a) **Simulation Results** (b) **Expert Choices**

Figure 4.6: **A comparison between the DEC policy and other caching policies on the RRM trace. The DEC policy first chooses FTPL and then switches to LRU. The resulting hit ratio is similar as LRU.**

ance as the popularity distribution changes. This causes LRU to have a higher hit ratio. OGA again performs well but not as well as the best policy. The DEC policy on the other hand first follows the FTPL expert and then later switches to LRU. However, as seen in Figure 4.6b, it switches around the point that the hit ratio of LRU becomes higher than FTPL. This suggests that the switch might happen too late and an earlier switch to LRU can improve the hit ratio. This is investigated in the next section.

4.5.4 YouTube Trace

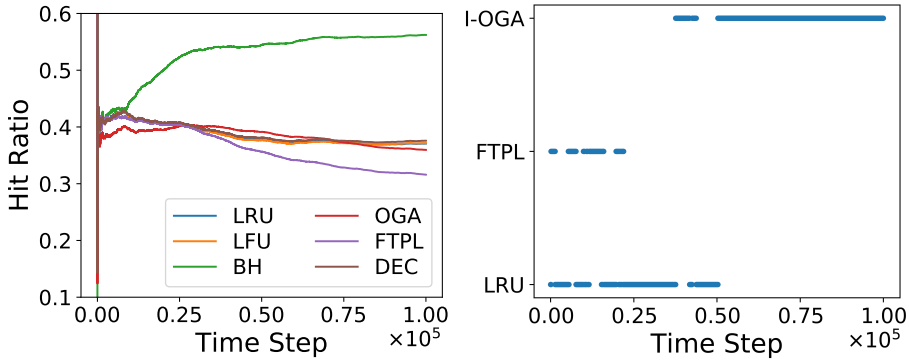
In Figure 4.7, the results of the YouTube trace are plotted. The policies all perform comparably, with FTPL performing significantly worse than the others. The difference between the caching policies and the best static cache in hindsight is also quite significant. In this trace, the DEC switches more often between experts than with the IRM or RRM trace. This is due to all experts having quite similar hit ratios. Another observation is that the DEC does not switch to OGA when OGA has the highest hit ratio. This is because the DEC uses Integral OGA as an expert instead of OGA. At that particular moment, Integral OGA performs worse than OGA as seen later in Figure 6.1c.

4.5.5 MovieLens Trace

The results of the MovieLens trace are similar to the IRM trace with FTPL, LFU, and DEC performing the best. The DEC policy also chooses the FTPL expert for almost all of the requests.

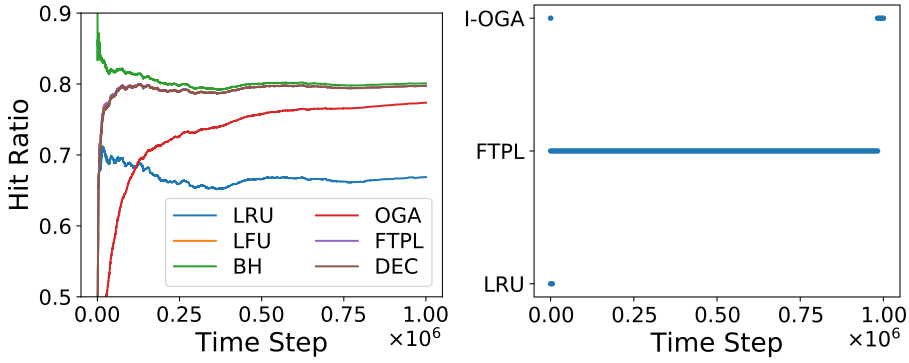
4.6 Mixing

As discussed before, the DEC policy might switch experts too late which can be observed in Figure 4.6a. This behavior can be explained when looking at the weights of each expert in Figure 4.9. The weights experts that do not do well quickly become



(a) **Simulation Results** (b) **Expert Choices**

Figure 4.7: **A comparison between the DEC policy and other caching policies on the YouTube trace. The DEC policy switches between FTPL and LRU at the start and later follows Integral OGA.**



(a) **Simulation Results** (b) **Expert Choices**

Figure 4.8: **A comparison between the DEC policy and other caching policies on the MovieLens trace. The results are similar to the IRM trace where DEC chooses the FTPL expert.**

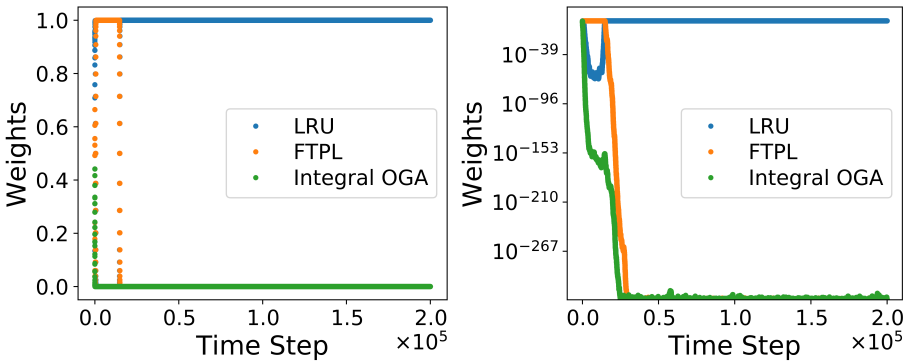
very small. This makes it that the recovery period is very long when the expert starts to do better and thus the slow expert switching.

Mixing can be used to overcome the problem of weights becoming too small. As explain in Chapter 2, there are three variations of mixing.

- Fixed-Share to Start Vector
- Fixed-Share to Uniform Past
- Fixed-share to Decaying Past

Here, Fixed-Share to Uniform Past is implemented with the same reasoning as in [23] as it can be implemented very efficiently compared to Fixed-Share to Decaying Past and still has good recovery performance. The mixing step can be implemented after normalizing the weights in Algorithm 7.

This additional mixing step also means that the bounds on regret from Section 4.4 are not valid anymore. Unfortunately, with the loss function used in the DEC policy,



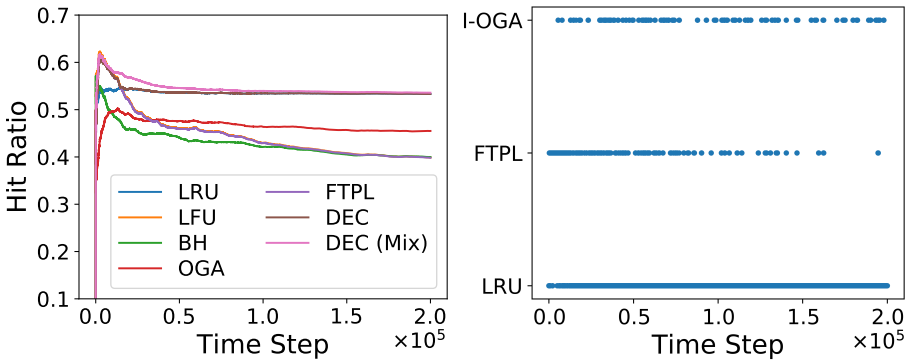
(a) **Linear weight axis** (b) **Logarithmic weight axis**

Figure 4.9: **Graphs that show the weights of each expert of DEC on the RRM trace. The weights are plotted using both a linear and a logarithmic weight axis. On the logarithmic scale, it can be observed that the weights become very small after some time.**

the regret bounds with mixing are unknown. The bounds for mixing in [11] required a loss function that only uses the prediction of each expert and the outcome. In this policy, the loss function also uses past information to calculate the loss and thus those bounds for mixing cannot be used.

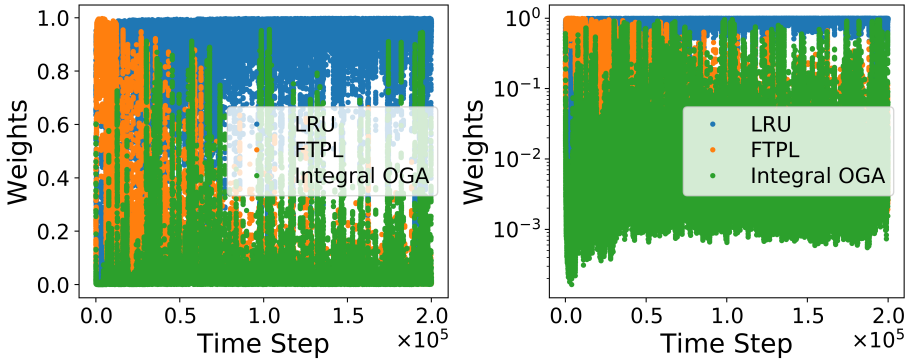
Nevertheless, in the simulation with RRM trace in Figure 4.10a with a mixing parameter of 0.005, there is a notable improvement in the hit ratio with mixing. The difference in hit ratio between with and without mixing is the largest at the point where best performing expert changes. This suggests that mixing does indeed improve the recovery speed of poor performing experts which is also apparent in the weights in Figure 4.11. The weights with mixing do not become as small as without mixing which explains the quick recovery.

The DEC policy with mixing is also simulated with the other traces, however, there is no noticeable improvement with those traces as the phenomenon of poor performing experts becoming good performing experts is not present there. These simulations can be found in Appendix A.



(a) **Simulation Results** (b) **Expert Choices**

Figure 4.10: **A comparison between the DEC policy with mixing and other caching policies on the RRM trace. With mixing, the DEC policy switches earlier from FTPL to LRU, which benefits the hit ratio.**



(a) **Linear weight axis** (b) **Logarithmic weight axis**

Figure 4.11: **Graphs that show the weights of each expert of DEC with mixing on the RRM trace. The weights are plotted using both a linear and a logarithmic weight axis. With mixing, the weights do not become as small as without mixing.**

Chapter 5

Multi Dynamic Expert Caching Policy

The DEC policy achieves good performance in the single cache situation as seen in the previous chapter. However, for the application of edge caching and in particular the femtocaching setting, the cache is part of a network of caches. Therefore, this chapter investigates the performance of DEC in the network of caches setting. This version of DEC in the network of caches is called MultiDEC.

First, the system model of the network of caches is discussed in Section 5.1. Following it, the design on how to incorporate DEC in the network of caches is presented in Section 5.2. At last, trace-driven simulations are done and the results are compared to other caching policies for the network of caches setting in Section 5.3.

5.1 System Model

The network of caches can be modeled as a weighted bipartite graph consisting of a set of users and a set of caches. The weights at the links represent the utility of the cache. The utility can model the benefit of using one cache over another cache which can range from bandwidth economization to quality of service improvements [39]. At each time step at most one user can request a single object. The objects are again all assumed to be uniform in size.

Figure 5.1 shows an example a bipartite graph. The caches are denoted as squares with a prefix of C and a number identifying the cache. The users are denoted as circles with a prefix of U and a number identifying the user.

5.2 Design

In the network of caches setting, an additional routing step is required with each request. This routing step determines which caches to serve the request. The same routing rule as BSA, mLRU, and LazyLRU is used for MultiDEC. This means that for each request, the accessible caches with the highest utility are first checked for the requested object until there is a cache hit with one of the caches or a cache miss with all of the accessible caches.

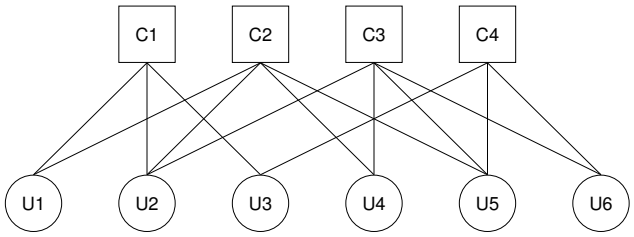


Figure 5.1: **Example diagram of a (weighted) bipartite graph. A square represents a cache. A circle represents a user. Each link can have a weight associated with it.**

The caches that are checked for the request are also updated with the information of the request. This is in contrast to mLRU or LazyLRU where the caches that are used to serve a request are not always updated with the information of the request. Note that when there is a cache hit, the remaining accessible caches are not checked for the requested object and consequently not updated with the information of the request as these caches have a lower utility.

This method of updating the caches is similar to the supergradient of the BSA policy when the positive values of the supergradient are used to determine which cache to update. However, compared to BSA, the caches in MultiDEC update independently from each other meaning that the state of one cache has no influence when updating another cache.

An individual cache of MultiDEC uses the same DEC policy as if it is in a single cache situation. This also means that each cache converges to the best in hindsight configuration for their visible requests over time. However, this is different than the best in hindsight configuration for the whole network as the routing of the requests is dependent on the content in the caches.

One example where the best in hindsight of the individual caches is not the best in hindsight of the caching network goes as follows. Suppose there are two users and two caches. User U1 is connected to C1 and user U2 is connected to both C1 and C2 as is illustrated in Figure 5.2. Both C1 and C2 have a cache size of 1. C1 has a utility of 1 for both U1 and U2. C2 has a utility of 2 for U2. The request sequence of U1 is $\{1, 1, 1\}$ and the request sequence of U2 is $\{1, 1, 1, 2, 2\}$. Based on these request sequences the best in hindsight for the individual caches C1 and C2 are both to cache object 1. This configuration results in a utility of $3 + 2 \times 3 = 9$. However, the best in hindsight for this network is actually for C1 to cache object 1 and for C2 to cache object 2. This configuration results in a utility of $6 + 2 \times 2 = 10$.

Therefore, MultiDEC is not guaranteed to converge to the best in hindsight configuration of the whole caching network. However, the simulations show that MultiDEC still has good performance compared to the other caching policies.

5.3 Results

5.3.1 Experiment Setup

The experiment setup for the network of caches is the same as in [39]. The caching network consists of 3 caches and 4 users. The users are connected to the caches

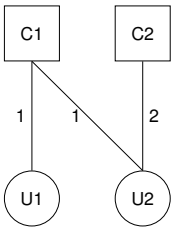


Figure 5.2: **An example of a caching network configuration where the best in hindsight of the individual caches does not match the best in hindsight of the caching network.**

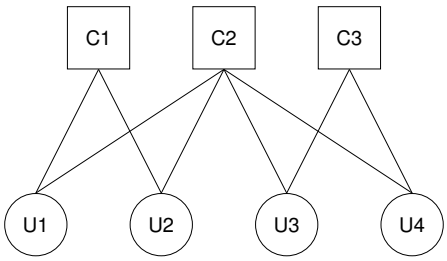


Figure 5.3: **A diagram that shows the caching network configuration used for the trace-driven simulations. Each link also has a weight or utility which is omitted from the diagram. All links leading to cache C1 have a utility of 1. All links leading to cache C2 have a utility of 2. All links leading to cache C3 have a utility of 100.**

as in Figure 5.3. The caches all have the same utility for every user, that is, C1 has a utility of 1, C2 has a utility of 2, and C3 has a utility of 100. At most one user can make a request each time step. The user that makes a request is uniformly chosen. All users have the same content popularity distribution.

The caches used for the simulation are MultiDEC, mLRU, LazyLRU, approximate best in hindsight ($\sim$BH), and BSA. The caching policies that have a learning rate, have it set to a value that minimizes the regret. The best in hindsight static cache configuration is approximated by calculating the best in hindsight for each cache individually. As explained before this is not the same as the best in hindsight cache configuration for the whole cache network. However, the calculations for the best in hindsight configuration for the cache network are much more computationally heavy as it is an NP-Hard problem [44], and therefore, I choose to approximate it with the best in hindsight for each cache. Nevertheless, this approximate best in hindsight is still competitive enough to give a good indication of the performance of MultiDEC.

The traces used for the simulations are still the traces as in the previous chapter, that is, the IRM trace, RRM, trace, YouTube trace, and the MovieLens trace. There is also a simulation done where the individual caches use the mixing variant of the DEC policy. However, the benefit with mixing is insignificant enough that it is omitted for clarity reasons. Still the results with mixing are in Appendix B.

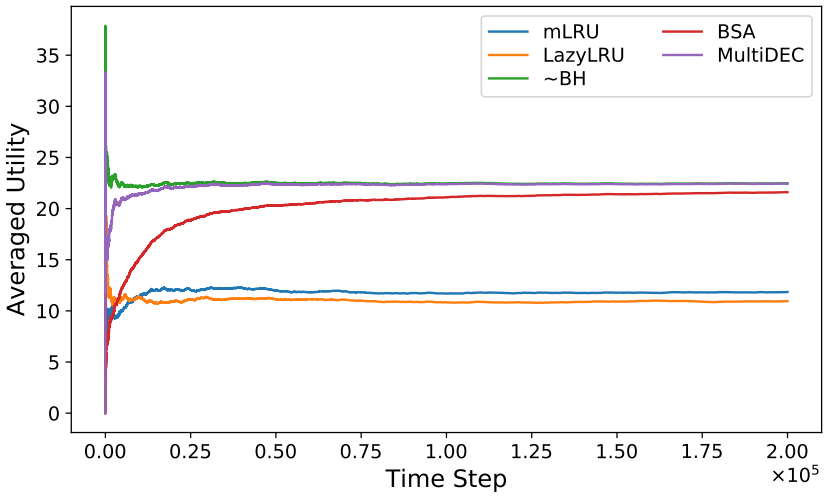
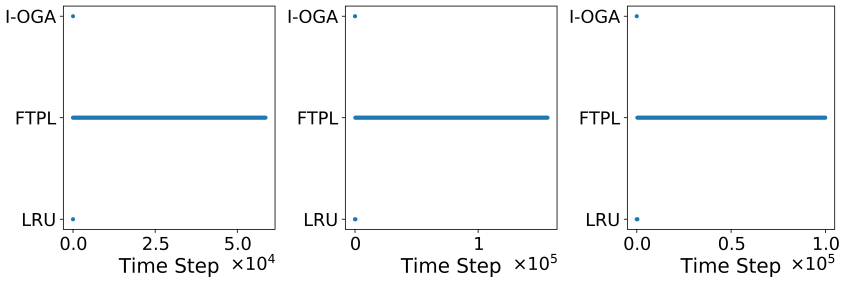


Figure 5.4: **A comparison between the MultiDEC policy and other caching policies for a caching network on the IRM trace. The MultiDEC has a noticeable higher averaged utility than mLRU and LazyLRU.**



(a) **Cache 1** (b) **Cache 2** (c) **Cache 3**

Figure 5.5: **Expert choices of the MultiDEC caches for the IRM trace. Each cache chooses FTPL which is similar as with the single cache situation.**

5.3.2 IRM Trace

The results from the IRM trace are in Figure 5.4. The experts that each cache chooses are in Figure 5.5. As expected for the IRM trace all the three DEC policies quickly choose the FTPL expert to follow. This results in MultiDEC having the highest utility compared to the other caching policies and approaches the utility performance of the approximate best in hindsight cache configuration.

5.3.3 RRM Trace

The results of the RRM trace in Figure 5.6 are a bit more interesting. Again, MultiDEC outperforms the other caching policies. This time the three caches do not choose the same experts as seen in Figure 5.7. Cache 1 follows the FTPL for most of the trace and switches to Integral OGA for the last part of the trace. Cache 2 and 3 choose LRU as the expert to follow. The LRU policy is the best performing caching policy for the

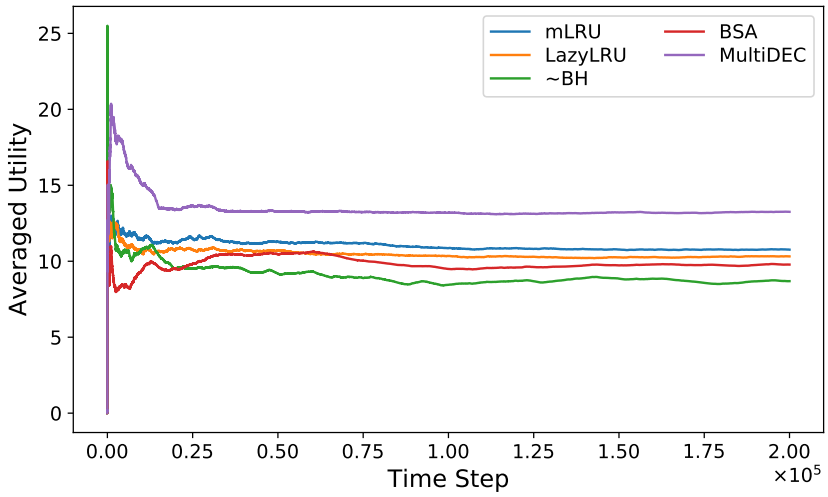
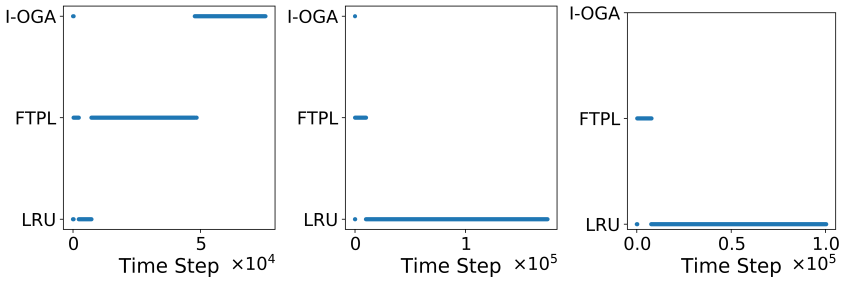


Figure 5.6: **A comparison between the MultiDEC policy and other caching policies for a caching network on the RRM trace. The MultiDEC policy has a significantly higher averaged utility over the other caching policies.**



(a) **Cache 1** (b) **Cache 2** (c) **Cache 3**

Figure 5.7: **Expert choices of the MultiDEC caches for the RRM trace. The difference in expert choices shows the benefits of MultiDEC.**

RRM trace in the single cache situation as seen in the previous chapter. One possible explanation for the different expert choices of Cache 1 is that the request sequence from the point of view of Cache 1 does not follow the RRM model due to the routing and therefore it results in different expert choices. This particular simulation shows the benefit of dynamic caching policy selection in a caching network as each cache can have a different caching policy that is optimal.

5.3.4 YouTube Trace

With the YouTube trace in Figure 5.8, the approximation of the best in hindsight static cache configuration significantly outperforms the caching policies which are similar to the single cache situation. However, with the dynamic caching policies, MultiDEC performs the best. The experts that are chosen have a similar result as with the RRM trace, where Cache 2 and 3 roughly choose the same experts and Cache 1 chooses different experts as seen in Figure 5.9.

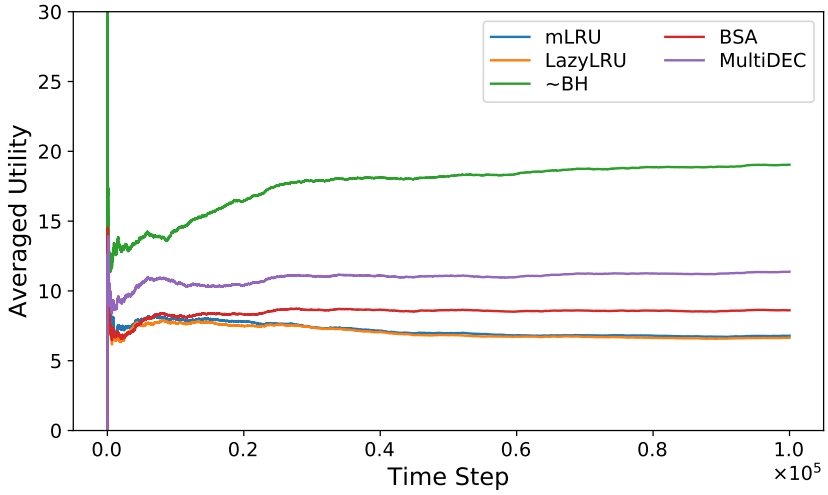
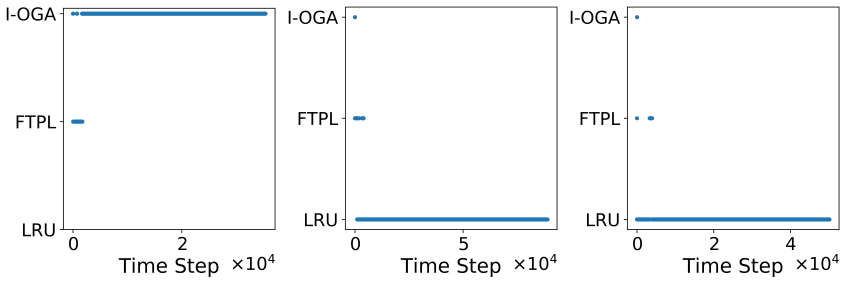


Figure 5.8: **A comparison between the MultiDEC policy and other caching policies for a caching network on the YouTube trace. MultiDEC performs better than the other reactive caching policies but not as good as the approximation of the best in hindsight static cache configuration.**



(a) **Cache 1** (b) **Cache 2** (c) **Cache 3**

Figure 5.9: **Expert choices of the MultiDEC caches for the YouTube trace. As with the RRM trace, Cache 1 chooses differently than Cache 2 and Cache 3.**

5.3.5 MovieLens Trace

The results of the MovieLens trace in Figure 5.10 are similar to IRM, where MultiDEC outperforms the other caching policies and approaches the approximation of the best in hindsight caching configuration. The interesting part, however, is in the choices of experts in Figure 5.11. Here, Cache 1 and 2 follow a similar pattern of first choosing FTPL and then switching to Integral OGA which is surprising given the fact that in the single cache situation FTPL has a significantly higher hit ratio than OGA.

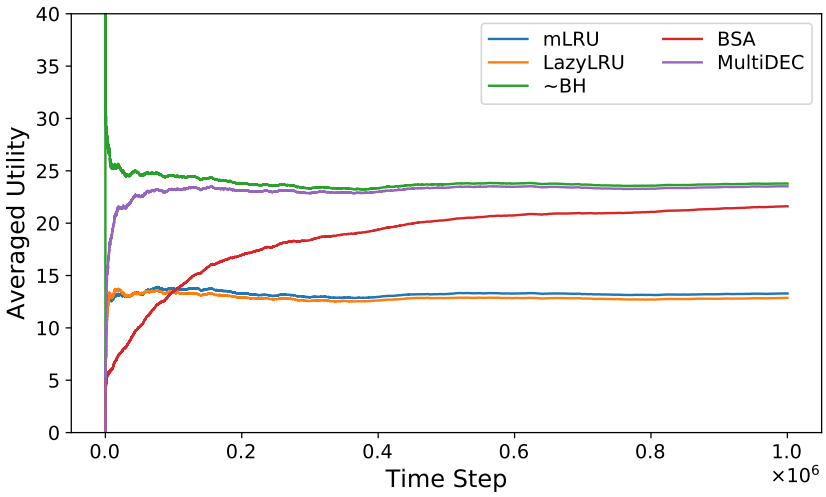
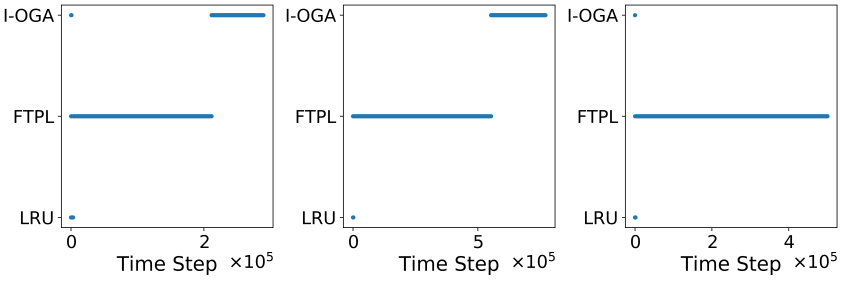


Figure 5.10: **A comparison between the MultiDEC policy and other caching policies for a caching network on the MovieLens trace. The results are similar to the IRM trace where MultiDEC has a significantly higher averaged utility of mLRU and LazyLRU.**



(a) **Cache 1** (b) **Cache 2** (c) **Cache 3**

Figure 5.11: **Expert choices of the MultiDEC caches for the MovieLens trace. The expert choices of Cache 1 and Cache 2 are different from the results of the single cache situation.**

Chapter 6

Integral OGA and Integral BSA

The OGA and BSA policy of [39] both have the assumption that objects can be cached as arbitrary small chunks. This assumption is not feasible in practice and consequently begs the question of whether it is possible to design versions of OGA and BSA that only cache entire objects. This chapter describes those versions of OGA and BSA, named Integral OGA and Integral BSA. Integral OGA and Integral BSA deploy an additional rounding step that is inspired by the DEC policy. These versions behave very similarly to the original caching policies in the simulations and achieve similar cache hit rates.

The chapter starts with the design of Integral OGA in Section 6.1. Then, I discuss whether the original regret bounds are still valid in this version of OGA in Section 6.2. Following it, I present some simulations that show performance similarities between the OGA and Integral OGA in Section 6.3. Lastly, I discuss the Integral BSA policy in Section 6.4.

6.1 Design

The original OGA caching policy updates the cache by calculating the gradient of (4.4) and adding a scaled version of it to the cache vector y_t. This however will make it that more objects are cached than there is storage available. Therefore, this caching vector is projected onto the feasible plane such that the caching configuration is within its constraints.

Integral OGA uses an additional step to round up the continuous caching vector of OGA. The caching vector of OGA can be viewed as a ranking where objects that are cached with a higher percentage are deemed with a higher ranking. Therefore, Integral OGA caches these objects with a high ranking wholly until the cache is full. For example, when the caching configuration of OGA is $\boldsymbol{y}^c = \begin{bmatrix} 0.2 & 0.6 & 0 & 0.8 & 0.4 \end{bmatrix}^T$ and the cache size is 2, then Integral OGA rounds it to $\boldsymbol{y} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \end{bmatrix}^T$.

This rounding projection is described mathematically in (6.1). The pseudocode of the Integral OGA algorithm can be found in Algorithm 8.

$$y = \Pi_{\mathcal{Y}}(\boldsymbol{z}) = \arg\max_{\boldsymbol{y} \in \mathcal{Y}} \boldsymbol{y}^T \boldsymbol{z} \tag{6.1}$$

Algorithm 8: Integral OGA caching policy

Input: Stepsize η
for $t = 1 \dots T$ **do**
  Check hit or miss based on y_t;
  Calculate gradient ∇f_t ;
  Calculate continuous caching vector $y^c_{t+1} = \Pi_{\mathcal{Y}^c}\left(y^c_t + \eta \nabla f_t(x_t)\right)$;
  Calculate discretized caching vector $y_{t+1} = \Pi_{\mathcal{Y}}\left(y^c_{t+1}\right)$;
end

6.2 Theory

The original OGA caching policy with a stepsize of $\eta = \frac{diam(\mathcal{Y}^c)}{L\sqrt{T}}$ has a regret bound of $R_T(OGA) \leq diam(\mathcal{Y})L\sqrt{T}$ [39]. The basis of the proof is the non-expansiveness property of the Euclidean projection. However, in Integral OGA, there is the additional rounding operation that does not have this non-expansiveness property. Another problem with the rounding up operation of (6.1) is that there are sometimes multiple solutions available when elements of the continuous input vector z have the same value. The resulting vector of the rounding up operation in that situation is then implementation dependent. Therefore, the bounds of OGA are not valid for Integral OGA.

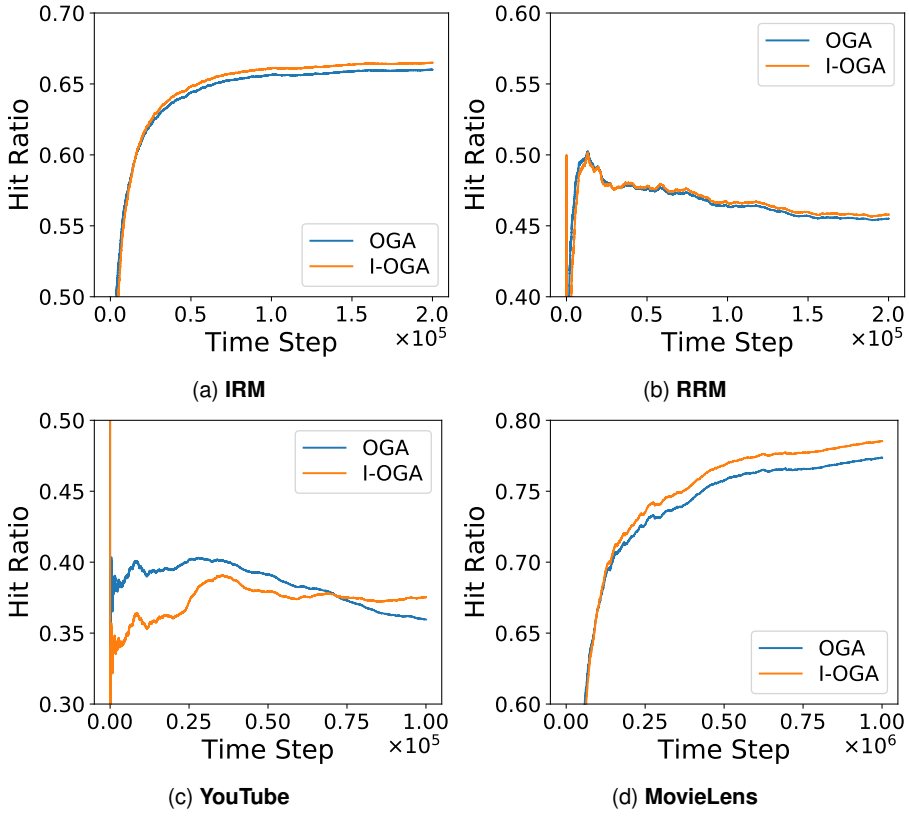
6.3 Results

The Integral OGA caching policy is simulated and compared with the OGA caching policy in Figure 6.1. The same traces as in Chapter 3 are used with the same configurations. In the IRM, RRM, and MovieLens trace it performs very close to OGA with even a slightly higher hit ratio. In the YouTube trace, the difference in hit ratio is greater. Nevertheless, it can be concluded that Integral OGA is a good enough approximation of OGA when the cache can only cache entire objects.

6.4 Integral BSA

The variant of OGA for a network of caches is BSA. This version also has the requirement that the objects can be cached in arbitrary small chunks. For the integral variant of BSA, named Integral BSA, the OGA caching policy in the individual caches is replaced with Integral OGA. For the calculation of the supergradient, the continuous internal values of Integral OGA are used. However, for the caching decisions, the values obtained after the rounding step are used. Again, due to the rounding step not having the non-expensiveness property, the theory of no regret theory of BSA is not valid with Integral BSA. Nevertheless, the simulations show that Integral BSA is a good approximation of BSA.

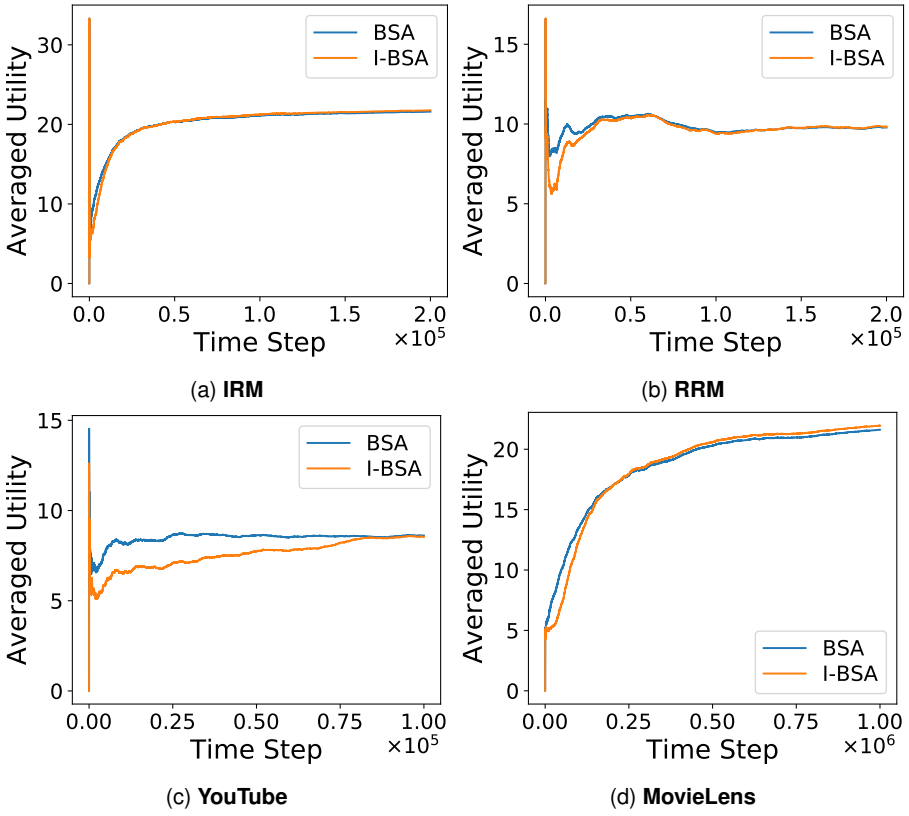
For the experiments, the same setup as in Chapter 5 is used. The results of the simulations are in Figure 6.2. Integral BSA has a utility that is close to BSA in all simulations.



(a) **IRM** (b) **RRM**

(c) **YouTube** (d) **MovieLens**

Figure 6.1: **Simulation results with Integral OGA and OGA. The Integral OGA variant performs very similarly to the original OGA.**



(a) **IRM** (b) **RRM**

(c) **YouTube** (d) **MovieLens**

Figure 6.2: **Simulation results with Integral BSA and BSA. Integral BSA performs very closely to the original BSA except on the YouTube trace.**

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, I have designed a no-time-averaged regret, i.e., a universal caching policy that still works comparable to conventional caching policies such as LRU and LFU. Previously designed universal caching policies sacrifice some performance in terms of cache hits in order to work well for any request sequence.

The Dynamic Expert Caching (DEC) policy designed in this thesis uses an overarching algorithm to dynamically switch between caching policies. DEC uses the Shrinking Dartboard algorithm from the Experts framework as the overarching algorithm due to this algorithm not making unnecessary switches. The Shrinking Dartboard algorithm has a theoretical bound on the number of mistakes between it and the best expert in hindsight. The experts in DEC are selected caching policies. The regret bound of DEC is formulated using the bound of the Shrinking Dartboard algorithm and the bounds of the caching policies. The resulting regret bound of DEC also exhibits the no-time-averaged regret property.

I have compared DEC to other caching policies using trace-driven simulations. The results show that DEC always has a similar cache hit ratio as the best caching policy in the simulation.

The mixing variant of DEC ensures faster switching between caching policies when different caching policies work better for different sections of the trace. In the random replacement model trace, this mixing variant of DEC does have a notable improvement over the regular DEC variant. One of the downsides of the mixing variant of DEC, however, is that the regret bound of DEC is not valid for it.

DEC is also adapted for inside a caching network, named MultiDEC. The simulations show significant improvements over other caching policies for caching networks such as BSA. MultiDEC also shows the benefit of dynamically selecting a caching policy as in the simulations the best caching policy for each cache in the caching can differ from each other.

Lastly, I have designed new versions of the existing caching policies OGA and BSA. These named Integral OGA and Integral BSA use an additional rounding step to cache entire objects instead of partially caching objects in the original versions. In the simulations, the integral versions of OGA and BSA perform very similarly to the original versions.

7.2 Future Work

For this thesis, I managed to design a caching policy that can dynamically switch between other caching policies. However, I still envision the following potential improvements:

- The DEC policy uses a discrete loss function for each expert. A fixed loss is given to an expert if the expert gives a low rank to the requested object. It could be interesting to use more information of the ranking for the loss of the expert. I have tried to use the full ranking of an expert to determine the loss, however, that resulted in slower convergence to the best expert. A possible explanation could be that the lower part of the rankings that the experts give, is not a good indication of the performance differences between the experts. In other words, the bad predictions of the experts are all equally bad and should not be compared with each other. However, it can still be interesting to see if the loss function can be tweaked to differentiate between experts that are very confident in their prediction and those who are less confident. For example, the loss function could be changed to add an additional tier where there is no loss if the rank of the requested object is within $\frac{\tau_r}{2}$ of the highest ranking objects, a little loss if it is within τ_r, and some loss otherwise. This loss function might improve the performance of DEC in the random replacement model trace without resorting to the mixing variant.

- For the single cache simulations, there a very few traces that show a switch between experts. Therefore, it would be interesting to simulate the DEC policy in more situations where expert switching happens. This can also showcase the benefit of the mixing variant of DEC.

- Currently, the mixing variant of DEC and the integral versions of OGA and BSA have no regret bounds. This is due to the modifications that are made to them which causes the original proofs to not be valid anymore. It can be interesting to see if in the future a different approach to the proofs can be found such that these policies can have regret bounds.

- The simulations run with some assumptions that are not entirely practical, e.g, the assumption that all objects are uniform in size. It can be interesting to see if the caching policies designed in this thesis can be adapted to work with arbitrarily sized objects. This would, however, require a change in how the rankings are used for caching.

Appendix A

Single Cache Simulations

In this chapter, all the simulations for the single cache setting can be found. Each figure displays the results of every caching policies that is implemented for the single cache setting on a different trace.

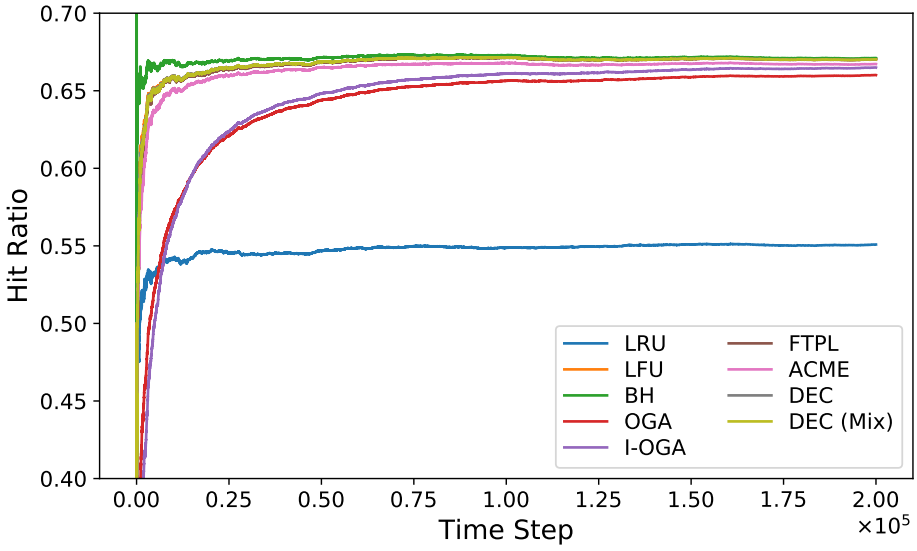


Figure A.1: **The results of every implemented caching policy for the single cache setting on the IRM trace.**

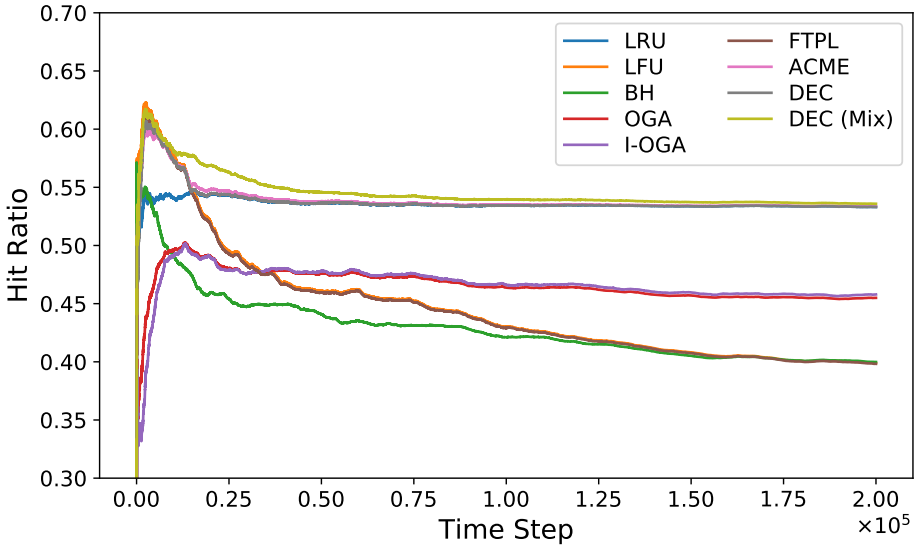


Figure A.2: **The results of every implemented caching policy for the single cache setting on the RRM trace.**

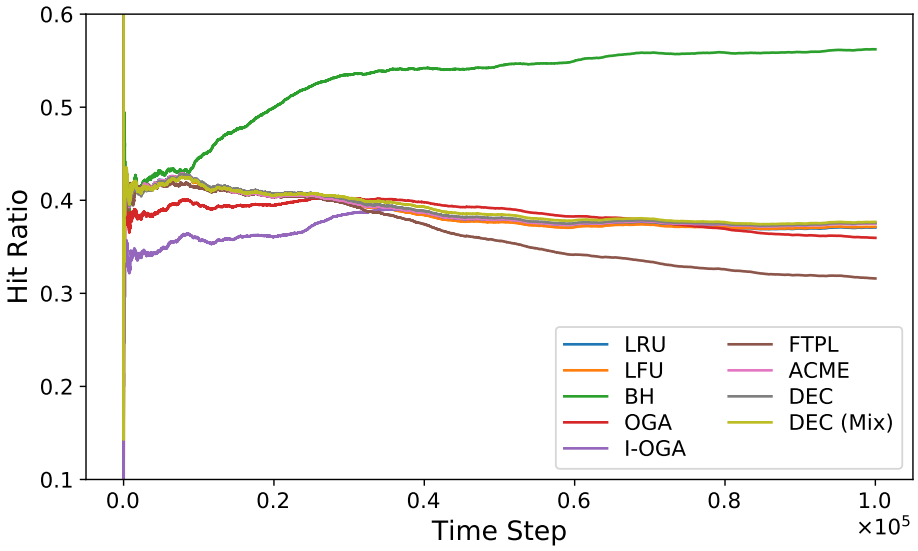


Figure A.3: **The results of every implemented caching policy for the single cache setting on the YouTube trace.**

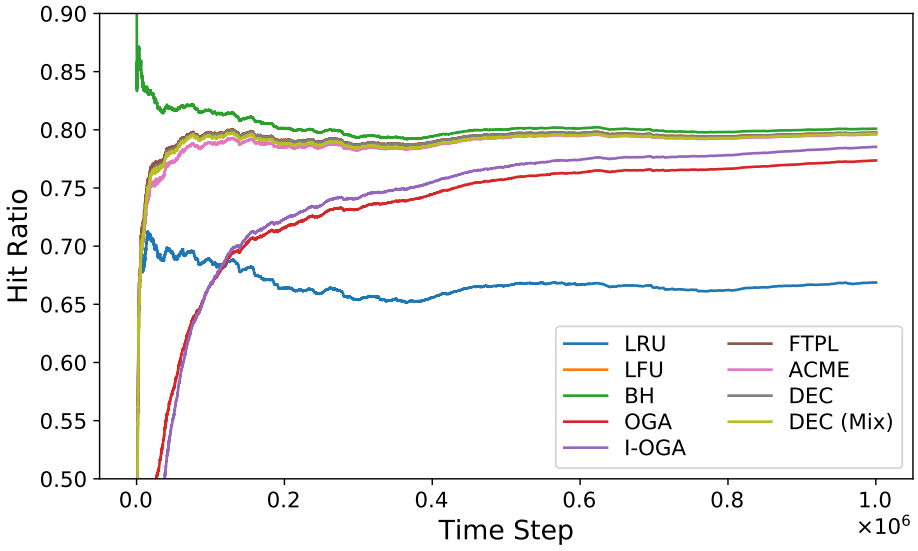


Figure A.4: **The results of every implemented caching policy for the single cache setting on the MovieLens trace.**

Appendix B

Network of Caches Simulations

In this chapter, all the simulations for the network of caches setting can be found. Each figure displays the results of every caching policies that is implemented for the network of caches setting on a different trace.

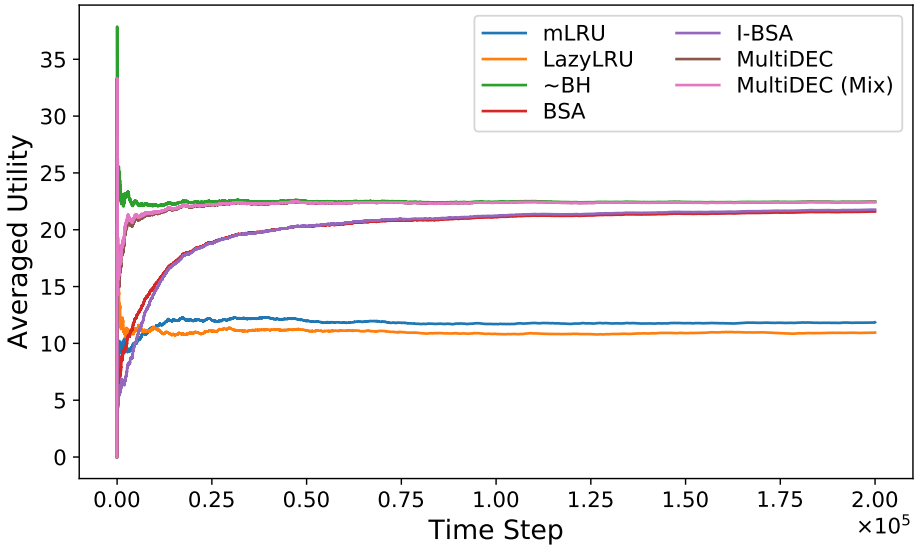


Figure B.1: **The results of every implemented caching policy for the network of caches setting on the IRM trace.**

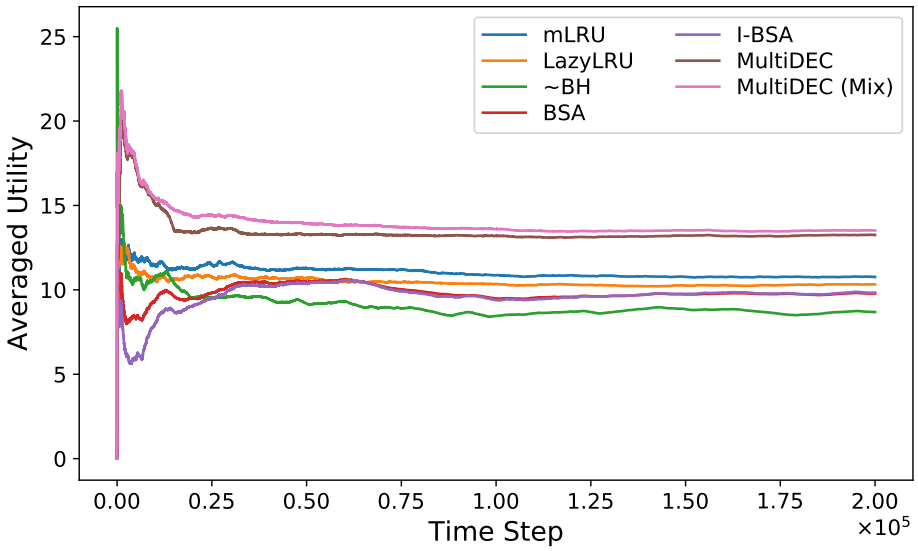


Figure B.2: **The results of every implemented caching policy for the network of caches setting on the RRM trace.**

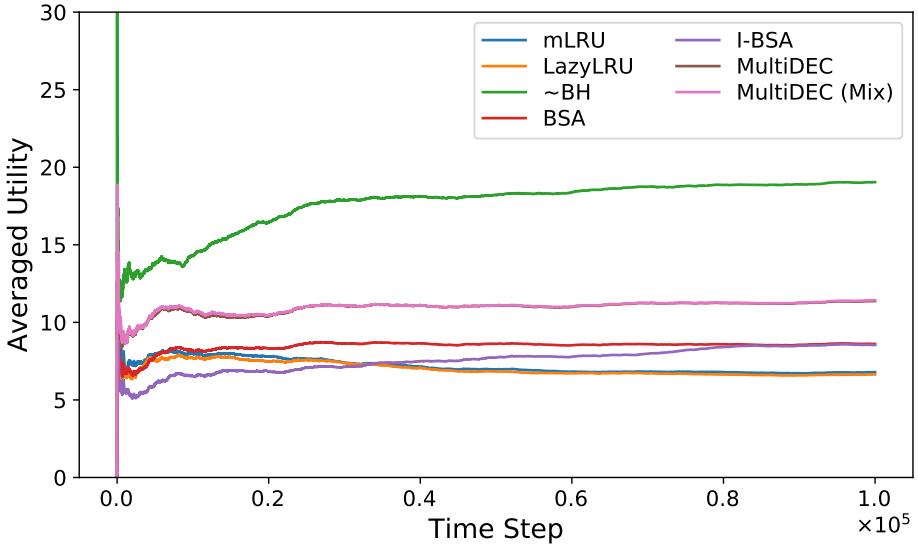


Figure B.3: **The results of every implemented caching policy for the network of caches setting on the YouTube trace.**

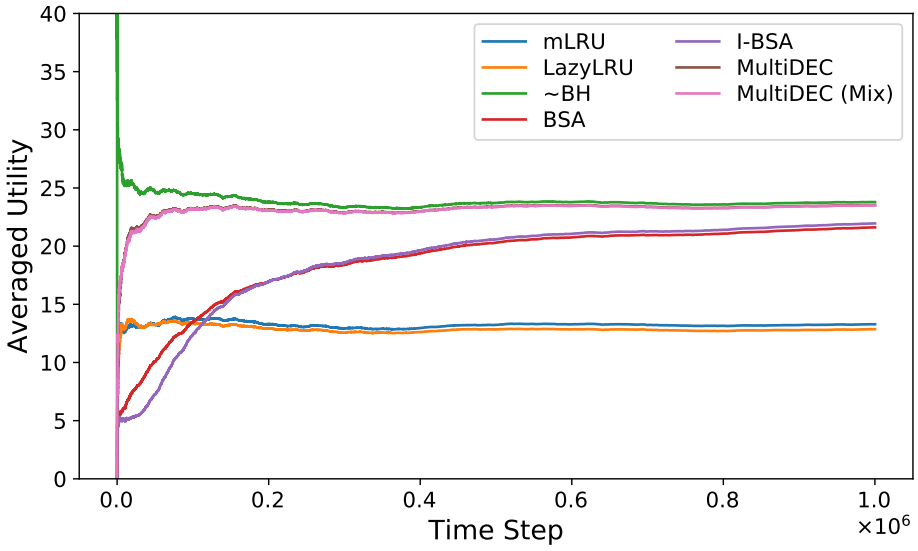


Figure B.4: **The results of every implemented caching policy for the network of caches setting on the MovieLens trace.**

Bibliography

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, Li Deng, Gerald Penn, and Dong Yu. Convolutional neural networks for speech recognition. *IEEE/ACM Trans. Audio, Speech, Language Process.*, 22(10):1533–1545, July 2014.

[2] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. *World Wide Web J.*, July 1995.

[3] Hasti Ahlehagh and Sujit Dey. Video-aware scheduling and caching in the radio access network. *IEEE/ACM Trans. Netw.*, 22(5):1444–1462, January 2014.

[4] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, October 2019.

[5] Waleed Ali and Siti Mariyam Shamsuddin. Intelligent client-side web caching scheme based on least recently used algorithm and neuro-fuzzy system. In *Int. Symp. Neural Networks*, pages 70–79, May 2009.

[6] Waleed Ali, Siti Mariyam Shamsuddin, Abdul Samad Ismail, et al. A survey of web caching and prefetching. *Int. J. Advance. Soft Comput. Appl*, 3(1):18–44, October 2011.

[7] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. Acme: Adaptive caching using multiple experts. In *Proc. Inf.*, volume 2, pages 143–158, January 2002.

[8] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *SIGMETRICS Perform. Eval. Rev.*, 27(4):3–11, March 2000.

[9] Baruch Awerbuch and Robert Kleinberg. Online linear optimization and adaptive routing. *J. Comput. Syst. Sci.*, 74(1):97–114, February 2008.

[10] Rajarshi Bhattacharjee, Subhankar Banerjee, and Abhishek Sinha. Fundamental limits on the regret of online network-caching. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(2), June 2020.

[11] Olivier Bousquet and Manfred K Warmuth. Tracking a small set of experts by mixing past posteriors. *J. Mach. Learn. Res.*, 3:363–396, November 2002.

[12] Karl S Brandt. *Using Multiple Experts to Perform File Prediction*. PhD thesis, University of California Santa Cruz, July 2004.

[13] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE Conf. Comput. Commun. (INFOCOM)*, volume 1, pages 126–134, March 1999.

[14] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE Conf. Comput. Commun. (INFOCOM)*, volume 1, pages 126–134, August 1999.

[15] Maria Carla Calzarossa and Giacomo Valli. A fuzzy algorithm for web caching. *Simul. Ser.*, 35(4):630–636, January 2003.

[16] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2017-2022, June 2019. White Paper.

[17] Jake Cobb and Hala ElAarag. Web proxy cache replacement scheme based on back-propagation neural network. *J. Syst. Softw.*, 81(9):1539–1558, September 2008.

[18] Alon Cohen and Tamir Hazan. Following the perturbed leader for online structured learning. In *Proc. Int. Conf. Mach. Inf. Process. Syst.*, pages 1034–1042, July 2015.

[19] B.D. Davison. A web caching primer. *IEEE Internet Comp.*, 5(4):38–45, July 2001.

[20] Yoav Freund and Robert E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *J. Comput. Syst. Sci.*, volume 904, pages 23–37. Springer Berlin Heidelberg, June 1995.

[21] Sascha Geulen, Berthold Vöcking, and Melanie Winkler. Regret minimization for online buffering problems using the weighted majority algorithm. In *Proc. COLT*, pages 132–143, June 2010.

[22] Anastasios Giovanidis and Apostolos Avranas. Spatial multi-lru caching for wireless networks with coverage overlaps. *SIGMETRICS Perform. Eval. Rev.*, 44(1): 403–405, June 2016.

[23] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Proc. Int. Conf. Neural Inf. Process. Syst.*, pages 1489–1496, January 2002.

[24] GroupLens. Movielens 1m dataset, February 2003. URL `https://grouplens.org/datasets/movielens/1m/`. Accessed: 22-10-2021.

[25] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4), December 2015.

[26] Gerhard Hasslinger, Konstantinos Ntougias, Frank Hasslinger, and Oliver Hohlfeld. Performance evaluation for new web caching strategies combining lru with score based object selection. In *Proc. 28th Int. Teletraffic Congr. ITC 2016*, volume 1, pages 322–330, September 2016.

[27] Elad Hazan. Introduction to online convex optimization. *Found. Trends Optim.*, 2 (3-4):157–325, August 2016.

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conf. CVPR*, pages 770–778, December 2016.

[29] David P. Helmbold, Darrell D. E. Long, Tracey L. Sconyers, and Bruce Sherrod. Adaptive disk spin—down for mobile computers. *Mob. Netw. Appl.*, 5(4):285–297, December 2000.

[30] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, May 2000.

[31] Koen Lam. Caching simulator, April 2022. URL `https://github.com/koenlam/CachingSimulator`.

[32] Emilio Leonardi and Giovanni Neglia. Implicit coordination of caches in small cell networks under unknown popularity profiles. *IEEE J.Sel. A. Commun.*, 36 (6):1276–1285, June 2018.

[33] Bin Li and Steven C. H. Hoi. Online portfolio selection: A survey. *ACM Comput. Surv.*, 46(3), January 2014.

[34] N. Littlestone and M.K. Warmuth. The weighted majority algorithm. *Inf. Comput.*, 108(2):212–261, February 1994.

[35] Dong Liu, Binqiang Chen, Chenyang Yang, and Andreas F. Molisch. Caching at the wireless edge: Design aspects, challenges, and future directions. *IEEE Commun. Mag.*, 54(9):22–28, September 2016.

[36] Ari Luotonen and Kevin Altis. World-wide web proxies. *Comput. Netw. ISDN Syst.*, 27(2):147–154, November 1994.

[37] Georgios Paschos, George Iosifidis, and Giuseppe Caire. Cache optimization models and algorithms. *Found. Trends Commun. Inf. Theory*, 16:156–343, January 2020.

[38] Georgios S. Paschos, George Iosifidis, Meixia Tao, Don Towsley, and Giuseppe Caire. The role of caching in future communication systems and networks. *IEEE J. Sel. Areas Commun.*, 36(6):1111–1125, September 2018.

[39] Georgios S. Paschos, Apostolos Destounis, Luigi Vigneri, and George Iosifidis. Learning to cache with no regrets. In *IEEE Conf. Comput. Commun. (INFOCOM)*, pages 235–243, April 2019.

[40] James Edward Pitkow and Mimi Recker. A simple yet robust caching algorithm based on dynamic access patterns. *Proc. the Second Int'l WWW Conf.*, pages 1039–1046, April 1994.

[41] Stefan Podlipnig and Laszlo Böszörmenyi. A survey of web cache replacement strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[42] Alireza Sadeghi, Fatemeh Sheikholeslami, and Georgios B. Giannakis. Optimal and scalable caching for 5g using reinforcement learning of space-time popularities. *IEEE J. Sel. Topics Signal Process.*, 12(1):180–190, February 2018.

[43] D. Sculley and Gabriel M. Wachman. Relaxed online svms for spam filtering. In *SIGIR 2019 - Proc. 30nd Int. ACM SIGIR Conf. Res. Dev. Inf. Retr.*, SIGIR '07, pages 415–422, July 2007.

[44] Karthikeyan Shanmugam, Negin Golrezaei, Alexandros G. Dimakis, Andreas F. Molisch, and Giuseppe Caire. Femtocaching: Wireless content delivery through distributed caching helpers. *IEEE Trans. Inf. Theory*, 59(12):8402–8413, September 2013.

[45] Tareq Si Salem, Giovanni Neglia, and Stratis Ioannidis. No-regret caching via online mirror descent. In *IEEE Int. Conf. Commun.*, pages 1–6, June 2021.

[46] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[47] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14(3):473–530, September 1982. ISSN 0360-0300.

[48] Stefano Traverso, Mohamed Ahmed, Michele Garetto, Paolo Giaccone, Emilio Leonardi, and Saverio Niccolini. Temporal locality in today's content caching: Why it matters and how to model it. *SIGCOMM Comput. Commun. Rev.*, 43(5): 5–12, November 2013.

[49] Shuo Wang, Xing Zhang, Yan Zhang, Lin Wang, Juwo Yang, and Wenbo Wang. A survey on mobile edge networks: Convergence of computing, caching and communications. *IEEE Access*, 5:6757–6779, March 2017.

[50] Kin-Yeung Wong. Web cache replacement policies: A pragmatic approach. *IEEE Netw.*, 20(1):28–34, January 2006.

[51] J. Zhang, R. Izmailov, D. Reininger, and M. Ott. Web caching framework: Aalytical models and beyond. In *IEEE Workshop on Int. Appl.*, pages 132–141, July 1999.

[52] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proc. of the Twentieth Int. Conf. on Int. Conf. on Mach. Learn.*, pages 928–935, August 2003.