Towards Efficient and Robust Caching: Investigating Alternative Machine Learning Approaches for Edge Caching

By Hoda Torabi,

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree of Master of Science

GRADUATE PROGRAM IN COMPUTER SCIENCE

York University
TORONTO, ONTARIO

December, 2023

Abstract

This study introduces HR-Cache, a caching framework designed to enhance the efficiency of edge caching. The increasing complexity and variability of traffic classes at edge environments pose significant challenges for traditional caching methods, which often rely on simplistic metrics. HR-Cache addresses these challenges by implementing a learning-based strategy grounded in Hazard Rate ordering, a concept originally used to establish cache performance upper bounds. By employing a lightweight supervised machine learning model, HR-Cache learns from HR-based caching decisions and predicts the 'cache-friendliness' of incoming requests, identifying 'cache-averse' objects as priority candidates for eviction.

Our experiment results demonstrate HR-Cache's superior performance. It consistently achieves 2.2–14.6% greater WAN traffic savings compared to the LRU strategy and outperforms both heuristic and state-of-the-art learning-based algorithms, while adding minimal prediction overhead. Though designed with the considerations of edge caching limitations, HR-Cache can be adapted with minimal changes for broader applicability in various caching contexts.

Acknowledgements

I would like to extend my deepest gratitude to my supervisors, Dr. Hamzeh Khazaei and Dr. Marin Litoiu, for their unwavering support and guidance. Their invaluable advice and guidance were instrumental in the success of my graduate studies.

I also want to thank my friends and colleagues from the Performant and Available Computing Systems (PACS) Lab and the Adaptive Software Research Lab. Their support has been an important part of this journey, greatly contributing to my research and personal growth.

A heartfelt thank you to my parents, for always loving me and supporting me. Without them none of this would be possible.

Finally, to Farnood, who believed in me when I didn't. I'm eternally grateful for his support throughout this journey.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Figures **vii**

List of Tables **viii**

Declaration of Authorship **ix**

1 Introduction **1**

1.1 Overview 1

1.2 Motivation 4

1.2.1 Reducing WAN Traffic 4

1.2.2 Limitations of Existing Learning-based Methods 5

1.3 Thesis Contributions 5

1.4 Thesis Organization 7

2 Background **8**

2.1 Limitations of Existing Methods 8

2.2 Hazard Rate-Based Upper Bound 12
2.3 Hazard Rate Function 14
2.4 Gradient Boosted Decision Trees 15
2.4.1 Fundamental Mechanics of GBDT 15
2.4.2 Algorithmic Workflow 15
2.4.3 Applications and Advantages 16

3 Related Work **17**
3.1 Learning-based Caching 17
3.2 Distributed Caching Systems 19
3.3 Machine Learning for System Efficiency and Improvement Beyond Caching 21

4 HR-Cache: Intelligent Caching **24**
4.1 Hazard Rate Estimation 25
4.2 Learning From HRO 27
4.3 Design of HR-Cache 29
4.3.1 Training Data 30
4.3.2 ML Architecture 31
4.3.3 The HR-Cache Policy 33
4.4 Optimizations 34
4.4.1 Batched Predictions 34

5 Experimental Validation **36**
5.1 Implementation 36
5.1.1 C++ Language 37

5.1.2 LightGBM 37

5.2 Kernel Hazard Estimation Validation 39

5.3 Preliminary Evaluation 41

5.4 Workloads 42

5.5 State-of-the-art algorithms 43

5.6 Main Results 48

5.7 Prediction Overhead Optimization 51

5.8 Ablation Study 54

6 Conclusion **56**

6.1 Summary 57

6.2 Future Work 58

6.2.1 Distributed HR-Cache 58

6.2.2 Integration into Production Systems 60

Bibliography **62**

A Chapter 5 Supplement **70**

List of Figures

4.1 General Learning-Based Cache Architecture Overview 29

4.2 Architecture Overview of HR-Cache 33

5.1 HR-E Bound Comparison 40

5.2 IBM Trace Byte Miss under HR-Cache 41

5.3 Comparison of HR-Cache and SOA Algorithms 49

5.4 Comparison of Byte Miss Ratios for HR-Cache, the Best Performing Policy, and LRU 52

5.5 Improvement of HR-Cache over the SOA under Ablation Study . . 55

List of Tables

5.1 Hyperparameters of the LightGBM Model 39
5.2 IBM Web Access Trace Collected from a Gateway Router 39
5.3 Summary of the traces used in our evaluation. 42
5.4 Comparison of Prediction Batch Sizes and Prediction Times for LRB and HR-Cache Algorithms 53
5.5 Prediction Overhead Reduction For Wiki 2018 53

A1.1 Performance Comparison on Wikipedia 2018 Trace 70
A1.2 Performance Comparison on Wikipedia 2019 Trace 71
A1.3 Performance Comparison on CloudPhysics Trace 71
A1.4 Performance Comparison on EU Trace 72

Declaration of Authorship

I, Hoda Torabi, hereby declare that this thesis titled, "Towards Efficient and Robust Caching: Investigating Alternative Machine Learning Approaches for Edge Caching", and the work presented herein are solely my own efforts. Parts of this thesis have formed the basis of a paper submitted to the 15th ACM/SPEC International Conference on Performance Engineering (ICPE) which is currently under review.

Chapter 1

Introduction

1.1 Overview

The proliferation of smart devices, coupled with advancements in wireless communication technologies, has given rise to a multitude of multimedia applications ranging from video streaming and gaming to virtual/augmented reality. These applications are increasingly prevalent in domains such as the 5th Generation Mobile Networks (5G), Internet of Things (IoT), and Internet of Vehicles (IoV) [1], contributing to a substantial rise in network traffic. This surge in traffic is leading to increased user latency and exerting considerable pressure on backhaul links, which are crucial for connecting local base stations to the Internet. Furthermore, the growing demand for high-throughput and stringent network performance due to these advanced applications poses significant challenges to the existing infrastructure of conventional wireless networks.

To address the surging data traffic and meet the demanding performance requirements of these applications, Multi-access Edge Computing (MEC) has emerged as a key solution [2]. MEC enhances network performance by introducing computing and caching capabilities at the network's edge. Notably, network edges are equipped with numerous edge servers to store content that users frequently request. This allows users to access a wide range of content directly from edge servers rather than remote cloud servers, significantly reducing latency in content retrieval and easing the load on network traffic. This approach, known as edge caching, is pivotal in various areas. In the context of 5G, popular content can be cached at various Base Stations (BSs), such as Small Base Stations (SBSs) and Macro Base Stations (MBSs). In the IoT sphere, content can be cached in smart devices, edge gateways, or localized IoT servers to facilitate quicker access and processing. For IoV, content can be stored at Roadside Units (RSUs) and within vehicles. Additionally, extending Content Delivery Network (CDN) services towards the mobile edge can help enhances user Quality of Experience (QoE) while reducing the load on backhaul and core network infrastructures [2, 3]. The relevance of this approach was highlighted by trends in CDN usage. In 2017, CDNs accounted for 56% of Internet traffic, with projections at that time estimating an increase to 72% by 2022 [4]. While this forecast is from a past perspective, it underscores the anticipated significance of localized content delivery in managing the increasing demands of internet traffic globally.

Since caches are typically situated on high-cost storage media with limited capacity, their sizes are generally much smaller than the sizes of the datasets they serve. This discrepancy makes the selection of data to be cached a critical decision. An efficient cache is one that stores data deemed most useful, thereby fulfilling a greater number of requests directly from the cache instead of relying on backend storage systems. The effectiveness of a cache is commonly evaluated using two metrics: the hit ratio, which is the proportion of requests served directly from the cache, and the byte hit ratio, which measures the fraction of data bytes served from the cache in relation to the total bytes requested. When a cache reaches its capacity limit, it employs an eviction algorithm to determine which data to retain and which to remove. The choice and effectiveness of this eviction algorithm play a pivotal role in maintaining high hit and byte hit ratios, thereby ensuring the cache operates at optimal efficiency. As such, this problem has been extensively studied since the advent of the internet. Caching strategies have evolved from basic heuristic methods like Least Recently Used (LRU), which evicts the oldest data first, to intricate algorithms that combine frequency and recency (e.g. Hyperbolic) and others that use a composition of frequency and object size (e.g. GDSF). Despite extensive research, most production systems—such as those employed by Akamai [5], Memcached [6], and NGINX [7]—commonly implement LRU variants as their standard caching algorithm. Yet, these may not be ideally suited to the particular demands of edge caching, which contends with limited cache sizes and the unpredictable nature of user requests [8]. The challenge in designing effective caching algorithms is that workload characteristics, like object access patterns or request processes, are not constant and often change over time. Thus, a heuristic that performs well for one workload scenario may falter in another, or fail to adapt when access patterns evolve, underscoring the necessity for flexible caching strategies that can overcome these challenges. The recent progress in Machine Learning (ML) has paved the way for significant improvements in cache algorithms, addressing these challenges. ML techniques, known for their robust data handling and precise pattern recognition capabilities, are being increasingly utilized in various caching contexts. For instance, reinforcement learning has been applied to predict content popularity, aiding in proactive caching strategies as discussed in [9]. Additionally, the use of supervised learning techniques in designing caching algorithms, particularly for making informed replacement decisions, represents a less explored yet promising area, as seen in [10]. This research is centred around caching schemes that incorporate learning-based approaches, leveraging ML to enhance cache performance and efficiency.

1.2 Motivation

1.2.1 Reducing WAN Traffic

Edge caching is strategically deployed to manage a significant portion of internet traffic locally, aiming to enhance economic viability and mitigate potential impacts on the broader Internet infrastructure. The effectiveness of these edge servers in local content caching is crucial, as they act as a primary buffer against extensive traffic influx. In instances where the edge cache fails to deliver requested content, known as cache misses, the data must be retrieved via the wide-area network (WAN), incurring additional load and potential delays. The growing volume of data handled by edge caches has notable financial implications for network service providers, especially when traffic at the edge is not optimally managed. Ensuring low latency is vital for small, latency-sensitive items, while for larger content like videos and large downloads, the focus shifts to reducing traffic management costs and averting congestion at critical points in the network [11]. Consequently, one of the primary objectives of edge caching is to increase the proportion of data served directly from the cache – a measure known as the byte hit ratio (BHR). This not only ensures efficient network operation but also plays a key role in cost containment and maintaining network integrity.

1.2.2 Limitations of Existing Learning-based Methods

Edge caching effectively brings popular content closer to users, enhancing storage resource optimization by reducing service latency and minimizing redundant data transmissions across backhaul links. However, while machine learning techniques have made significant strides in caching schemes, learning-based caching approaches still confront various challenges. These challenges underscore the necessity to explore and develop more advanced and varied machine learning strategies for caching, aiming to overcome existing limitations and further enhance caching efficiency. Chapter 2 of this thesis provides an overview of these challenges, particularly those relevant to our proposed method. This targeted analysis shapes the design considerations of our approach, ensuring that our development is informed by the most relevant issues in the field.

1.3 Thesis Contributions

To achieve the above objectives, this research proposes a new learning-based caching policy to improve caching performance while conforming to the limitations of edge networks. The accuracy and effectiveness of the proposed scheme is demonstrated by comprehensive experimental results with real-word and synthetic datasets. The main contribution of this thesis are:

- Development of HR-Cache Using HRO Upper Bound: We introduce HR-Cache, a novel caching policy leveraging the hazard rate ordering upper bound (HRO) principle from [12], translating the prefetching-associated principle into actionable insights for real-time cache decision-making.

- Integration of Machine Learning: We use lightweight machine learning to develop an effective caching algorithm based on the HRO principle to inform a caching policy based on the prediction of this model.

- Application of Non-Parametric Hazard Estimation: Our research uniquely applies non-parametric hazard estimation for the HRO principle, enhancing the effectiveness of the HRO principle, differing from the simplistic assumptions of prior work.

- Development in a C++ Trace-Driven Simulator: HR-Cache is developed within a C++ based trace-driven simulator, ensuring accurate evaluation of our caching framework and validating HR-Cache against a variety of workload scenarios.

- Comprehensive Testing with Diverse Traces: In departure from most works in edge caching that report performance using a single trace, our research conducts a comprehensive evaluation of HR-Cache using multiple workload traces, including real-world data and synthetic scenarios.

- Optimizations for Reduced Prediction Overhead: The thesis presents optimizations in HR-Cache, such as parallelism and strategic cache updating techniques, which allows us to significantly reduce prediction overhead and enhance efficiency compared to contemporary learning-based caching algorithms.

- Ablation Study: An ablation study validates the effectiveness of key contributions; the non-parametric hazard estimation, and our approach of translating the prefetching-associated principle of the HRO into actionable insights for real-time cache decision-making. The results of this demonstrate the effectiveness of our main contributions, affirming the soundness of our methodology in the context of HR-Cache.

1.4 Thesis Organization

The structure of this thesis is laid out in the following manner. Chapter 2 delves into the necessary background details, setting the stage for our research. Chapter 3 covers related research work. Chapter 4 is dedicated to the detailed design and implementation of HR-Cache. Chapter 5 presents the experimental validation and discussion of our results. Finally, Chapter 6 concludes the thesis, reflecting on our contributions and outlining potential directions for future research.

Chapter 2

Background

This chapter begins with a summary of significant works in the field, outlining the essential lessons that have informed the development of our framework. Subsequently, we explore the concept of the hazard rate upper bound as introduced in [12], which serves as a fundamental component of our methodology. Concluding this chapter, we present a detailed overview of Gradient Boosted Decision Trees, the supervised learning technique that plays a critical role in the architecture of our framework.

2.1 Limitations of Existing Methods

A popular line of research is utilizing reinforcement learning for cache decisions. Due to the large state-action space, these methods tend to be more complex and computationally demanding. Additionally, they can be sensitive to hyperparameters, making it challenging to fine-tune their performance. Furthermore, the delayed rewards common in reinforcement learning can result in slow reaction times in dynamic environments. This may hinder the algorithm's ability to adapt quickly to changing content popularity patterns or user behavior [13].

Another category of recent research focuses on leveraging theoretically optimal caching policies for developing learning-based methods. A significant point of reference here is the Belady optimal policy [14]. This algorithm operates on the principle of evicting the object that will be used furthest in the future, thereby minimizing miss rate. While Belady's algorithm provides an ideal strategy for cache replacement, its real-world application has been limited because it requires foreknowledge of future cache access patterns, which is generally not feasible. Nevertheless, this algorithm forms a basis for designing practical caching policies.

Hawkeye [15] was the first to introduce learning from the Belady's algorithm. Hawkeye employs a binary classification model to determine whether a cache line is likely to be reused (deemed 'cache-friendly') or not ('cache-averse'). Their policy prioritizes the eviction of cache-averse lines over cache-friendly ones. By using oracle labels for previous access patterns, Hawkeye effectively transforms cache replacement into a supervised learning challenge. Building upon Hawkeye's foundation, Glider [16] enhances this approach by integrating deep learning techniques to develop a more accurate predictor than its predecessor. However, it's important to note that both Hawkeye and Glider focus on hardware caches and are not directly applicable to software cache systems, particularly those handling variable-sized objects. Another work, Parrot [17] adopts an imitation learning approach to automatically learn cache access patterns by leveraging Belady's. Although effective, its computational demands can be significantly high.

Diverging from Parrot's methodology, LRB, as outlined in [10], employs a different strategy by predicting the next arrival times of object requests. This enables LRB to approximate Belady's algorithm through a supervised learning method. By learning the next access time for each object based on a multitude of features, LRB identifies and evicts objects predicted to have the furthest request time. This strategy has demonstrated enhanced performance over state-of-the-art caching algorithms in terms of byte hit ratios. However, LRB is not without its limitations. To closely emulate the optimal offline oracle, a system like LRB is required to predict the next access times for all objects in the cache, selecting for eviction the one with the most distant future request. This prediction process can be extremely resource-intensive for large caches. LRB mitigates this by limiting the inference to a sample of 64 objects for each eviction. Despite this optimization, the prediction overhead remains a significant computational burden. LRB's use of dynamic features means that prediction results are not reusable over time, necessitating fresh sampling and inference for every eviction. Reflecting this overhead, on a single CPU core, each eviction in LRB consumes 227.19 µs.[1] Consequently, this caps the eviction rate at a maximum of approximately 4,500 objects per second per core, rendering it less efficient for high-demand production environments.

LFO [13], another work employing supervised learning, first calculates the sequence of optimal caching decisions (OPT) for recent history using a min-cost flow model from [18], designed for optimal caching of variable-sized objects. Following this calculation, LFO applies manually-designed features and a gradient boosting decision tree to train a binary classifier for caching decisions. The classifier's prediction is then used to imitate the admission policy of OPT and serve as a ranking

[1]For 64 GB cache size, Wikipedia 2019 workload

metric for the eviction policy. However, the process of deriving optimal decisions based on the min-cost flow model is complex and computationally intensive, hindering LFO's ability to swiftly adapt to workload changes. Additionally, its design necessitates executing a prediction for every incoming request, further impacting its practical efficiency.

Inspired by similar principles to our work, LHR in [19] draws on the concept of the Hazard Rate bound from [12] to develop a learning-based caching policy. Unlike a direct adoption, LHR modifies this approach by constructing an online upper bound, which approximates the request process through a Poisson process. Under this assumption, the hazard rate remains constant and is equivalent to the request rate for each object. While this approach simplifies their model, it considerably narrows the applicability of LHR [20]; particularly in light of [12]'s demonstration that the HRO upper bound is effective for any stationary arrival process. Thus, LHR's reliance on the Poisson assumption potentially restricts the full exploitation of HRO's capabilities.

Given these limitations, it is crucial to investigate alternative machine learning approaches for caching that can overcome these challenges, while achieving a high byte hit ratio. In this research, we focus on exploring new learning-based caching policies that address the limitations of current algorithms and target the fundamental constraints of existing caching methods. It has been observed that there is a significant gap between the hit probabilities of state-of-the-art caching algorithms and upper bounds of optimal policies (OPT) such as the offline Belady algorithm [21], Flow-based offline optimal [18], and the more recent Hazard Rate

(HR) based upper bound [12]. Our goal is to leverage the Hazard Rate based upper bound to inform the design of learning-based caching methods. Considering the insights gained from the overview of existing works, our framework's design will be informed around these pivotal lessons:

1. **Utilization of HRO Bound:** Taking into account the limitations of LHR's Poisson assumption, our approach will seek to fully leverage the HRO bound's potential, avoiding oversimplified assumptions that could undermine the practicality and justification of using machine learning.

2. **Minimizing Prediction Overhead:** Addressing the challenge of high computational demands seen in methods like LRB, our framework will prioritize efficient prediction mechanisms to enhance scalability and performance.

3. **Decision-Making Process:** Considering the complexity of the LFO approach, we aim to create an efficient method for making caching decisions. This is important in fast-paced environments where models need regular updates and training. Our approach is designed for quick adjustments to stay up-to-date with frequent changes.

]

2.2 Hazard Rate-Based Upper Bound

The framework in [12] considers a caching system serving n distinct objects, possibly of different sizes, with a cache capacity of B bytes. In its most basic case, it considers a cache with capacity B, catering to requests for n distinct objects of equal size. In this context, [12] introduces the hazard rate based rule, termed as HR-E, which operates as follows:

- At any given time t, HR-E first determines the hazard rate function for each object.
- Then it places in the cache the B objects which have the largest hazard rates (ties between equal rates are broken randomly) .
- A request at time t is considered a 'hit' if the requested object is among those cached based on the aforementioned criteria.

They use this rule as a way to upper-bound various cache performance metrics including object hit and byte hit ratio.

They further extend this rule to obtain an upper bound on the byte hit probability for variable size objects. In this case, the authors adapt the hazard rate-based rule denoted as HR-FC to accommodate fractional caching, a strategy that permits the storage of a fraction of an object. Specifically, the rule at any time caches objects with the highest hazard rates until an object cannot fit. For the object that cannot be fully fit due to limited remaining cache capacity, only a sufficient number of bytes required to reach the cache limit are stored. In the case for equal-sized objects, the HR-E rule serves as an upper bound on the cache hit probability for non-anticipative caching policies, while HR-FC serves as an upper bound on the cache byte hit probability, which is the metric we are interested in. Throughout this work, we will collectively refer to these rules as the the 'HRO' rule (hazard rate ordering rule) for consistency and ease of reference.

The work in [22] derives closed form expressions for the upper bound under some specific object request arrival processes, such as Poisson Process, On-Off Request Process, Markov Modulated Poisson Process, and Shot Noise Model.

2.3 Hazard Rate Function

Let us consider the sequential times at which object i is accessed as $\{\tau_{ik} \mid k \in \mathbb{Z}\}$. The time interval between consecutive requests—namely, the kth and $(k-1)$th requests—for the same object i is termed X_{ik} and computed as $\tau_{ik} - \tau_{i(k-1)}$, for $k \geq 1$. By default, τ_{i0} is set to zero. The sequence $\{X_{ik}\}_{k\geq 1}$ is assumed to form a stationary point process, with the cumulative distribution function (c.d.f) for the inter-arrival time given as $F_i(t) = \mathrm{P}(X_{ik} \leq t)$, and its corresponding density function is represented as $f_i(t)$.

The hazard rate function, denoted as $\lambda_i(t)$, associated with $F_i(t)$ is defined as follows:

$$\lambda_i(t) = \frac{f_i(t)}{1 - F_i(t)}, \quad t \in [0, F_i^{-1}(1)], \tag{2.1}$$

Here, the hazard rate function is the conditional density of the occurrence of an object request, given the realization of the request process over $[0, t)$ [23]. It is noteworthy that the hazard rate function's meaning can vary based on its application context. For example, in survival analysis, the hazard rate quantifies the conditional probability of an item's failure, given that it has remained functional up to a specific time point. In caching terminology, we can treat failure/death of an item as an object being requested.

2.4 Gradient Boosted Decision Trees

Gradient Boosted Decision Trees (GBDT) are a potent and widely-utilized machine learning technique, particularly known for their efficacy in handling tabular data. This ensemble learning method, primarily used for regression and classification tasks, leverages the concept of boosting in conjunction with decision tree algorithms, optimizing an arbitrary differentiable loss function through iterative enhancements.

2.4.1 Fundamental Mechanics of GBDT

GBDT operates on the principle of sequentially integrating multiple weak learners—specifically, decision trees—to construct a comprehensive and accurate model. Each weak learner in this context is a basic model that provides predictions slightly better than random chance. The sequential integration of these trees is meticulously designed to rectify the residual errors from the preceding aggregate of trees, with the objective loss function varying based on the task - be it regression, classification, or others.

2.4.2 Algorithmic Workflow

1. **Initial Model Establishment**: GBDT begins with an elementary model, often a simple mean of the target values for regression problems or log odds for classification challenges. This initial model sets the foundation for subsequent refinement.

2. **Iterative Enhancement Process**: At each stage, GBDT constructs a new decision tree aimed at modeling the residual errors of the existing ensemble. These residuals represent the divergence between the current predictions of the ensemble and the actual observed values.

3. **Employing Gradient Descent for Optimization**: The boosting mechanism in GBDT utilizes the gradient descent algorithm to minimize the loss function. Each iteration fits a new tree to the negative gradient of the loss function, thereby incrementally steering the model toward reduced loss and improved accuracy.

4. **Regularization Techniques**: To mitigate overfitting and manage model complexity, GBDT integrates parameters like the learning rate and tree depth. The learning rate adjusts the influence of each new tree in the ensemble, while a restricted tree depth helps in curbing the model's complexity.

2.4.3 Applications and Advantages

GBDT have been effectively utilized in a wide array of data science competitions and practical scenarios, ranging from enhancing search engine algorithms to optimizing recommendation systems. Its scalability and capability to handle different data types, including missing values and mixed types (numerical and categorical), contribute to its popularity. This attribute, along with its lack of requirement for feature normalization, renders GBDT particularly suitable for a broad array of applications. Additionally, GBDT offers interpretability advantages over complex models like neural networks. In the specific context of caching, the effectiveness of GBDT is supported by studies like [10] and [13].

Chapter 3

Related Work

This chapter provides a comprehensive overview of existing literature in the domain of caching, with a particular focus on optimization and learning-based caching approaches. We then delve into studies that extend beyond conventional single-cache frameworks, exploring advancements in distributed caching systems. The latter part of this chapter is dedicated to a review of recent advancements in employing machine learning techniques for enhancing system efficiency and performance. This exploration will contextualize our work within the broader landscape of caching and machine learning applications in system optimization.

3.1 Learning-based Caching

In the study by [24], the authors investigate a machine learning-based approach for Web proxy caching, focusing on predicting the likelihood of an object being re-visited. To construct their training dataset, they extract pertinent features of Web objects from trace logs and proxy files. These features are then used to train models using labels: '1' for content that is re-requested and '0' otherwise.

The models are trained using Support Vector Machine (SVM) and Decision Tree algorithms, exploring their effectiveness in accurately making cache replacement decisions.

The authors in [25] investigate the problem of optimal content caching in a private wireless small cell base station (sBS) with limited backhaul. The authors model this problem as a multi-armed bandit (MAB) problem and propose three caching algorithms to efficiently learn unknown but time-invariant popularity profiles by balancing exploration and exploitation and subsequently cache the most popular files for a pre-defined time period.

In [26], a deep reinforcement learning (DRL) approach is proposed for making cache replacement decisions (whether or not to store the currently requested content in the cache, and if yes, to determine which local content to replace) at a base station with the goal of maximizing the cache hit rate in order to reduce the data traffic. The framework is built on the Wolpertinger architecture and trained using the deep deterministic policy gradient. Although the performance of the proposed framework shows both short-term and long-term cache hit rates improvement compared to LRU, LFU, and FIFO policies, the authors only consider equal-sized contents.

In PopCaching [27], a learning-based caching algorithm is proposed that predicts content popularity based on the context of requests. It employs a context vector with four dimensions, representing request frequencies in different time windows. Rather than directly learning content popularity, PopCaching learns the relationship between future popularity and request context using a dynamic partitioning approach. This method divides the context space into non-overlapping hypercubes, allowing PopCaching to exploit similarities in access patterns and adapt to evolving content popularity patterns. However, it only uses a very limited feature set as it relies only on request frequencies. This limited feature set might not capture all relevant factors affecting content popularity, potentially impacting prediction accuracy.

Recent research has increasingly concentrated on employing theoretically optimal caching policies as a foundation for developing learning-based approaches. This topic was elaborated upon in Section 2.1, highlighting our focus on integrating these methodologies.

3.2 Distributed Caching Systems

The mentioned works primarily concentrate on developing caching strategies for a single caching entity. Another prevalent scenario in edge networks, however, involves a network of interconnected caching nodes working together.

Earlier works in this regard mostly focus on utilizing traditional methods based on convex optimization or probability modeling to address the content placement problem for IoT/distributed networks. For instance, the study presented in [28] examines a cache cluster composed of multiple leaf-cache nodes and a parent-cache node either connected directly or via a parent node. The research focuses on determining the optimal content placement in cache nodes to minimize total bandwidth consumption, offering approximate solutions for specific scenarios. In another work [29], the authors present a general optimization problem to determine optimal content placement policies for caches installed across multiple levels of hierarchies, formulated as an integer optimization problem which takes into account the possibility of varying content demand patterns across leaf nodes. The study demonstrates that the problem is NP-Hard in its general form but can be expressed as the maximization of a submodular function subject to uniform matroid constraints.

Another line of work is based on learning algorithms such using machine learning or deep learning. For instance, in [30] the authors develop a video content caching scheme for cellular networks to enhance user QoE and reduce backhaul traffic. The approach uses an extreme learning machine approach for predicting global future content popularity based on human perception-informed features. The adaptive caching scheme combines mixed-integer linear programming optimization for cache placement at the initialization of the cache and an improved version of LRU (S3LRU) for cache replacement.

In [31], the authors introduce a reinforcement learning approach for proactive caching that accounts for the space-time popularity of user requests. They develop a Q-learning caching algorithm to learn the optimal policy while considering the unknown transition probabilities of the popularity dynamics. The small base stations (SBs) estimate their local popularity profiles and share them with the network operator, which aggregates them to estimate the global popularity vector. Their method adjusts the reward to balance tracking global trends and serving local requests, making it suitable for handling geographically and temporally variable cellular traffic. However, the Q-learning-based method might not be feasible for practical mobile edge computing systems due to the large state-action space involved. In [32], the authors propose a deep reinforcement learning approach based on hyper deep Q-networks (DQNs) for adaptive caching in a hierarchical content delivery network, which features a two-level network structure with one parent node and multiple leaf nodes, similar to the structure we will consider in Section **??**. In this setup, leaf nodes rapidly gather request information, while the parent node takes caching decisions at the beginning of each slow-timescale slot, based on the collected data from the leaf nodes. This method enables an effective balance between fast and slow timescales for more efficient caching decisions in a collaborative network. Finally, [33] model cache replacement as a Markov Decision Process (MDP) and propose a Double Deep Q-Network (DDQN) approach for handling this task. Additionally, they introduce a Federated Learning framework, allowing users to collaboratively learn a shared model while keeping raw data local. This framework enables edge nodes, such as base stations, to learn a global model by averaging local updates, thereby enhancing the efficiency and security of the caching process in the network.

3.3 Machine Learning for System Efficiency and Improvement Beyond Caching

Building on the exploration of machine learning applications in caching for system enhancement, the scope of ML and deep learning (DL) extends far beyond, permeating various other aspects of system performance and efficiency. This section delves into the diverse applications of these technologies across different system-related domains, highlighting how ML and DL are instrumental in advancing and optimizing system functionalities beyond caching solutions.

For instance, [34] employ deep neural networks (DNNs) for optimizing resource management in edge computing environments, enabling dynamic scheduling in distributed fog systems by estimating key Quality of Service (QoS) metrics. In another work, [35], the authors utilize recurrent neural networks to develop performance models for queueing networks, aiming to improve resource utilization based on it. For database efficiency, [36] leverage tree convolutional neural networks and reinforcement learning to optimize queries , while [37] apply machine learning techniques for effective database indexing. Similarly, In a study by Google [38], the application of machine learning in data center (DC) optimization is explored, demonstrating its pivotal role in enhancing operational efficiency. This research employs a neural network framework to accurately model and predict Power Usage Effectiveness (PUE), showcasing the capability of machine learning to effectively utilize existing sensor data in complex DC environments. In the study [39] at Microsoft, the authors delve into the intricacies of query optimization for big data systems, confronting the challenges posed by the increasing complexity of workloads and the nuanced dynamics of cloud environments. To navigate these complexities, they introduce "Microlearner" a machine learning-based optimizer tailored for Microsoft's extensive big data workloads. This optimizer is distinguished by its use of fine-grained learning models that train on subsets of workload data, enabling it to adeptly handle the diverse and evolving nature of cloud-based systems. By applying this method to Microsoft's SCOPE query engine, "Microlearner" aims to improve the process of query optimization, demonstrating a new approach to handling the intricacies of big data processing in a cloud context. In networking research, [40] propose a Robust statistical Traffic Classification (RTC) scheme for traffic classification in big data systems. This scheme is designed to address the challenge of identifying zero-day applications, which are previously unknown in traffic classification systems. They utilize a combination of supervised and unsupervised machine learning techniques to improve the accuracy of classifying both zero-day and known application traffic. The research focuses on enhancing traffic classification performance in the context of ever-evolving network traffic and applications. In another work, [41], the authors investigate the use of deep reinforcement learning for addressing resource management problems in systems and networking. They introduce DeepRM, a model that applies deep reinforcement learning techniques to the complex task of resource management, particularly focusing on the allocation and scheduling of tasks with varying resource demands. DeepRM is designed to learn optimal resource management strategies directly from experience, positioning it as a machine learning-driven alternative to conventional heuristic-based approaches in system and network resource management.

Chapter 4

HR-Cache: Intelligent Caching

In this section, we first discuss the application of non-parametric estimation for hazard rate calculation. We then explain how we harness the Hazard Rate Ordering rule as a fundamental principle for our learning-driven approach. Subsequently, the section outlines the crucial design choices that shape our HR-Cache framework. The chapter concludes by bringing together these concepts to present the complete HR-Cache framework. The primary goal of HR-Cache is to assess whether a requested object is cache-friendly or cache-averse. Upon a cache miss, the requested object is inserted into the cache; however, objects identified as cache-averse are placed in a candidate queue for potential future eviction. HR-Cache gives priority to evicting objects from this candidate queue, resorting to the main queue only when the candidate queue becomes empty.

4.1 Hazard Rate Estimation

To effectively implement hazard rate-based rule in our framework, we must first accurately determine the hazard rate function for each object. While this is relatively straightforward for synthetic data sets, it poses a significant challenge in real-world production settings. One approach to this challenge is approximating the inter-request times of objects using well-defined distributions, such as Poisson [19] or Generalized Pareto [12]. However, relying solely on these approximations could potentially diminish the benefits of leveraging machine learning in cache decision-making since these approximations may not be universally applicable across varying workloads and use-cases. Therefore, to calculate hazard rates that are adaptable to various workload trace distributions, we use the kernel hazard estimator proposed by [42]. We obtain this estimator by applying smoothing to the increments of the Nelson-Aalen estimator.

The Nelson-Aalen estimator is a non-parametric method used to estimate the cumulative hazard function in survival analysis. Unlike parametric methods, which make specific assumptions about the underlying hazard rate distribution, the Nelson-Aalen estimator does not require any such assumptions. We denote $H(t)$ as the cumulative hazard function at time t. The estimator is given by:

$$H(t) = \sum_{j:t_j \leq t} \frac{d_j}{n_j}$$

where t_j are the observed event times, d_j is the number of events at time t_j, and n_j is the number of subjects at risk just before time t_j. However, the Nelson-Aalen estimator results in a step function, which is not differentiable. Instead, kernel smoothing techniques are utilized to smooth the increments of the cumulative function estimate obtained by the Nelson-Aalen estimator [43]. The kernel hazard estimator we use takes the form:

$$\lambda(t) = \frac{1}{h} \sum_{i=1}^{n} K\left(\frac{t - t_i}{h}\right) \Delta H(t_i)$$

where $K(\cdot)$ is a kernel function (e.g., Epanechnikov kernel), h is the bandwidth, determining the width of the smoothing window, and $\Delta H(t_i)$ is the increment in the Nelson-Aalen estimate at time t_i, which is $\frac{d_i}{n_i}$.

The Epanechnikov Kernel The Epanechnikov kernel is a non-parametric kernel function widely used in statistics, particularly in the fields of kernel density estimation and non-parametric regression. It is defined mathematically as follows:

$$K(u) = \begin{cases} \frac{3}{4}(1 - u^2) & \text{for } |u| \leq 1 \\ 0 & \text{for } |u| > 1 \end{cases}$$

Here, u represents the scaled distance from the point of estimation, and the kernel is only non-zero within the interval $[-1, 1]$.

The Epanechnikov kernel is favored in many statistical applications for several reasons:

- **Optimal MISE**: It minimizes the mean integrated squared error (MISE) in kernel density estimation, making it an efficient choice for accurate probability density function estimation.

- **Compact Support**: Its compact support ($|u| \leq 1$) ensures computational efficiency, as only a limited range of data points affect the estimation at each point.
- **Effective Distance Weighting**: The kernel naturally weights observations based on their distance, giving more influence to points closer to the target, which is embedded in its parabolic shape within its support.
- **Bias-Variance Trade-off**: It offers a desirable balance between bias and variance, which is crucial in statistical estimation.
- **Simplicity and Practicality**: The Epanechnikov kernel's simple form facilitates ease of implementation and understanding.
- **Reduced Boundary Bias**: The kernel's compact support can help mitigate boundary bias in density estimation, especially at data range edges.

4.2 Learning From HRO

Before diving into the learning process, it is crucial to make a key observation. We argue that since the HRO bound (Section 2.2) is derived in a pre-fetching manner, it does not directly correspond to cache decision at the time of request to an object. Specifically, the HRO rule assumes that at any time t, the objects with the highest hazard rates among all available objects, have already been pre-fetched and are present in the cache. Thus, the requested object at time t is considered a hit if it is among objects in the cache. Based on this, for object i to be considered as cached in the system, the previous request to object i must admit it to the cache. Or in other words, when the request at time t arrives, it can only be considered a hit if object i was already cached due to a prior request. We classify these earlier requests as cache-friendly, as they are the ones leading to hits. With this and the HRO rule as a backdrop, we are set to develop a learning-based caching strategy. Our approach employs a sliding window of past requests $W[k]$. Using the gathered requests in $W[k]$, we do three things:

1. First, using the inter-request times of objects in the window we calculate the increments of hazard rates for each object according to the Nelson-Aalen estimator, to be later smoothed by the kernel hazard estimator.

2. Second, we go over the requests in the window and mark them as hit/miss based on the HRO rule. Meaning for each request at time t, we compute the hazard rate at time t for every object within the window using the kernel estimator, and consider objects with the highest hazard rates in cache until one doesn't fit. If the object requested at time t is among the cached objects, it is considered as a hit; otherwise, it is considered as a miss.

3. Next, we examine the requests in the window once more: For each request i that was marked as a hit in the first step, we mark the previous request to i as cache-friendly. This provides us with a vector of cache decisions for requests in the window, which serves as the label data for our machine learning model training.

HR-Cache then trains a model that maps features to the decision derived in step

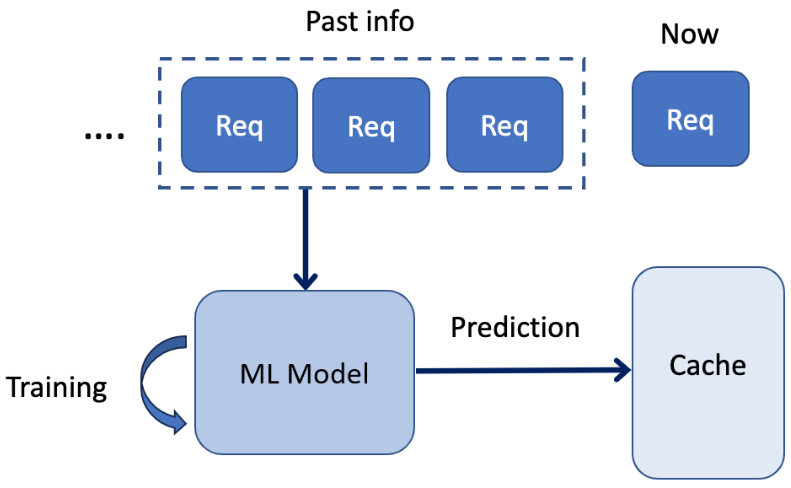


FIGURE 4.1: General Learning-Based Cache Architecture Overview

3. The trained model is subsequently used over the next window, $W[k+1]$, to inform cache decisions during which HR-Cache again records the requests.

4.3 Design of HR-Cache

This section presents the design details of HR-Cache, which uses ML to imitate the HRO algorithm. In developing an effective machine learning-based caching system, we focus on balancing the reduction of byte miss ratios with the feasibility of system implementation. This process entails addressing several interrelated design challenges:

1. **Generation of Training Data:** A key aspect of our approach is the dynamic generation of training datasets, using historical data. Given the variability in workloads over time, the system must be capable of regular retraining with new, accurately labeled datasets. This also involves determining

the appropriate amount of past data to use. While a larger dataset can potentially improve the training quality, it is imperative to optimize memory usage to ensure sufficient resources are available for caching.

2. **Machine Learning Architecture:** Selecting an effective ML architecture, this involves feature selection and prediction target.

3. **Caching Policy Implementation:** The final challenge involves the strategic use of the model's predictions in forming a caching policy. This step is critical in translating the insights gained from the ML model into effective caching decisions.

The subsequent sections will explore the specific design decisions made for HR-Cache, tackling each of these challenges to create a cohesive and efficient caching solution.

4.3.1 Training Data

An important design issue involves determining the optimal amount of past information to utilize. We adopt a sliding window approach, using the data within this window for hazard estimation, deriving the HRO cache decision, and model training. The choice of window size significantly impacts the system's effectiveness. A small window might result in few data for training or hazard rate estimations, while a window that is too large could lead to increased memory usage, as well as longer processing and training times. While some studies arbitrarily define their window sizes (e.g., [13] opts for a window of 1 million, [44] for the initial 10 million requests), [15] considers window size as a factor of cache capacity, $1\times$ represents a window that consists of accesses to k cache lines, where k is the capacity of the cache. We choose a 3× window, meaning the unique bytes of object requests in the window is three times the cache size as we find that this works well across all our experiments, however there is room for investigating how to set an optimal window size. In practice, the sliding window can encompass millions of objects, which presents significant challenges for the labeling process, particularly when reconstructing the HRO-Rule. To address this issue, HR-Cache employs a strategy of randomly sampling objects within the window to generate training samples. The sampling rate is automatically calibrated to ensure that the total number of operations stays within a manageable range, thus preventing the computational overhead from becoming prohibitive. In our C++ implementation, this adaptive approach has proven to be effective, yielding favorable results while keeping the computational demands at a reasonable level.

4.3.2 ML Architecture

This subsection describes the components of HR-Cache's ML architecture.

Features

When designing a machine learning model for cache decision, it is essential to choose relevant features that can help predict the optimal decision. Our chosen features encompass both the insights from past heuristics and the insight of recent learning-based caching policies. Traditional caching heuristics focus on individual metrics, such as object recency (as seen in LRU), its frequency (as in LFU), or object size. This is while learning-based methods allow us to incorporates a range of them. We consider the following features which can be derived in an online and robust manner.

1. **Delta series**: The time differences between consecutive requests for an object. Δ_1 indicates the amount of time since an object was last requested. Δ_2 indicates the time in between an object's previous two requests and so on, i.e., Δ_n is the amount of time between an object's n^{th} and $(n-1)^{\text{th}}$ previous requests. This can provide insights into the object's access pattern, which can help predict future requests. We use 32 deltas as our features.

2. **Decayed frequency**: Unlike simple frequency, decayed frequency accounts for the recency of requests by giving more weight to recent accesses. It calculates the fraction of requests for an object among all requests so far, but with a diminishing emphasis on older requests. This approach helps in capturing not just how often an object is requested, but also how its popularity or relevance changes over time.

3. **Static features**: These include unchanging characteristics of an object, such as its size and type. Static features can be useful due to their inherent correlation with different access patterns. For our implementation we only consider size among static features due to the availability of data in our traces.

Training HR-Cache

The primary objective of HR-Cache is to utilize its feature set to determine if an incoming request is conducive to caching ('cache-friendly') or not ('cache-averse'),

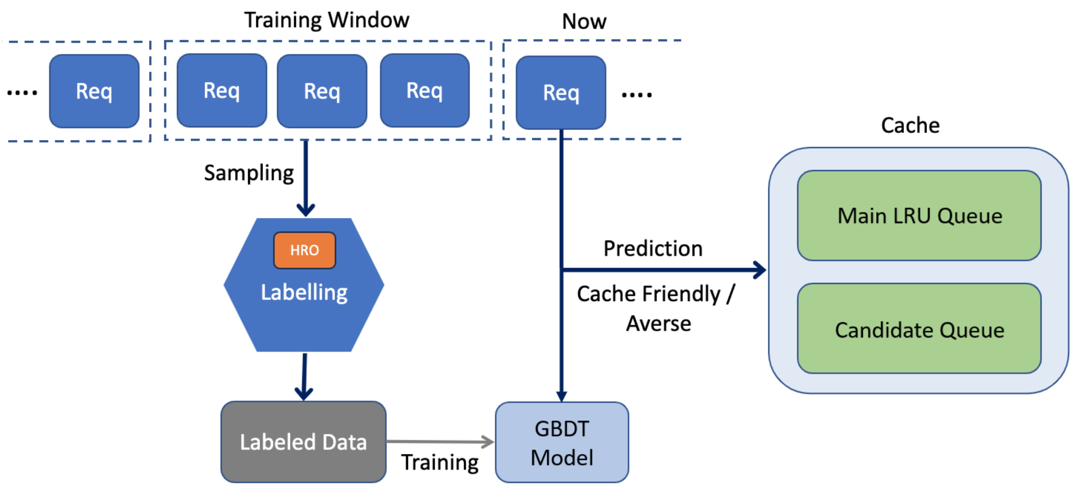


FIGURE 4.2: Architecture Overview of HR-Cache

in alignment with the Hazard Rate Ordering (HRO) rule. For this purpose, we have selected the Gradient Boosted Decision Trees (GBDT) model, as outlined in Section 2.4. This choice is grounded in the strengths and applicability of GBDT to our specific caching context. We incorporate logistic regression as the objective function within the GBDT framework, finding that it offers improved performance for the binary classification task central to our caching strategy.

4.3.3 The HR-Cache Policy

Putting it all together, we design a caching policy guided by our learned model. For every object request, our HR-Cache predictor outputs a decision indicating whether the object is cache-friendly or cache-averse. This decision guides how we update the cache as detailed in Algorithm 1. The goal is to manage objects so that cache-averse items end up in the candidate queue, while cache-friendly ones are placed in the main queue. The candidate queue consists of objects that the HR-Cache identifies as unlikely to lead to hits, hence prioritized for eviction.

Meanwhile, the main queue operates on an LRU basis, ensuring that if we need to evict from the main queue, it's the older objects that are removed.

Algorithm 1 HR-Cache Policy

```
procedure UpdateCache(object, lookupTable)
    Perform lookup for object in lookupTable
    if object is in cache (Hit) then
        if object is in Candidate Queue then
            if predicted as Cache-friendly then
                Change mode to Main Queue
                Move object from Candidate to Main Queue
            end if
        else if object is in Main Queue then
            if predicted as Cache-friendly then
                Promote object to MRU in Main Queue
            else if predicted as Cache-averse then
                Change mode to Candidate Queue
                Move object to Candidate Queue
            end if
        end if
    else                                          ▷ Request not in cache (Miss)
        if predicted as Cache-friendly then
            Add object to Main Queue
        else
            Add object to Candidate Queue
        end if
    end if
end procedure
```

4.4 Optimizations

4.4.1 Batched Predictions

The basic HR-Cache needs to predict cache-friendliness of objects as each request arrives. To take advantage of the architectural strengths of multi-core processors in contemporary CDN and edge servers, we implement data parallelism in our cache decision-making. This modification permits parallel predictions for B requests simultaneously. The chosen batch size, B, plays a critical role in balancing parallelism and miss ratio. A small B fails to fully utilize the potential of parallelism, while an excessively large B can lead to delayed predictions and negatively affect the miss ratio. We selected a batch size of B = 128, finding it optimal for harnessing parallelism without affecting our miss ratio. For instance, in our experiments, which do not account for object retrieval overhead, a batch size of B = 128 enabled an increase in throughput from handling 11,828 requests per second to 98,404 requests per second, while maintaining cache performance efficiency on the Wiki 2019 trace.

Chapter 5

Experimental Validation

This section describes traces, the setup of our experiments, the competing algorithms, and the parameter settings of HR-Cache. Unless otherwise noted, the reported results for HR-Cache are based on its default operation settings, which include batch-mode inference with a batch size of 128. We conduct trace-driven experiments to evaluate the performance of HR-Cache against a broad spectrum of state-of-the-art caching algorithms. Our analysis primarily focuses on two key questions: First, we examine how the byte miss ratio of HR-Cache compares with that of other state-of-the-art research systems across a variety of traces and cache sizes. Second, we assess how HR-Cache performs in relation to the state-of-the-art (SOA) learning-based cache mechanisms, particularly in terms of prediction overhead.

5.1 Implementation

We developed our framework in C++ to accurately assess our framework's miss ratios by replaying cache requests from traces. We use the LightGBM library for the GBDT model training.

5.1.1 C++ Language

In the development of our caching framework, the choice of programming language was pivotal. We opted for C++ due to its well-acknowledged high-performance capabilities, which are essential in handling the intensive computational demands of caching mechanisms. It also offers fine-grained control over memory and system resources, a feature critical in optimizing the performance of caching algorithms. This control allows for the efficient handling of large datasets and high-speed data processing, ensuring that our implementation operates with minimal latency and maximum throughput.

C++'s combination of advanced features such as templating and object-oriented programming significantly contributed to the development of a robust and scalable caching algorithm. Its capabilities for both low-level manipulation and high-level abstraction were instrumental in crafting an efficient, resource-manageable caching solution. Furthermore, C++'s compatibility with a variety of optimization algorithms and techniques greatly enhanced our method's performance and reliability. These attributes render C++ particularly suitable for high-performance computing applications like ours, and facilitate the potential integration into existing caching frameworks.

5.1.2 LightGBM

LightGBM, Light Gradient Boosting Machine, developed by Microsoft as part of their Distributed Machine Learning Toolkit, stands as a prominent gradient boosting framework based on decision tree algorithms, designed for efficiency and high performance, especially in large-scale machine learning tasks.

One of the key features of LightGBM is the Gradient-based One-Side Sampling (GOSS), which retains instances with large gradients and performs random sampling on instances with small gradients. This approach not only maintains accuracy but also significantly improves computational speed. Alongside, Exclusive Feature Bundling (EFB) is employed to reduce the dimensionality of data. EFB bundles mutually exclusive features, thereby reducing the number of features without losing valuable information, further enhancing the efficiency of the model.

Handling large datasets is one of LightGBM's strengths, attributed to the implementation of both GOSS and EFB. This makes LightGBM a practical choice for big data applications, where traditional gradient boosting methods may not be feasible due to memory constraints. Additionally, LightGBM supports parallel learning and is optimized for distributed computing, including GPU support, which offers faster computations necessary for large-scale machine learning tasks.

Another significant advantage of LightGBM is its native support for categorical features. This capability simplifies the data preprocessing pipeline, as it can manage categorical features without requiring extensive pre-processing to convert them into numerical values. This feature sets LightGBM apart from many other boosting frameworks that lack such native support.

In terms of performance, LightGBM often surpasses other gradient boosting frameworks. The combination of GOSS and EFB, along with other algorithmic optimizations, contributes to this enhanced performance. LightGBM is also known for its flexibility and ease of integration into various programming environments. It supports high-level programming languages such as Python and R, and importantly for our application, offers native support for C++. This compatibility with C++ is particularly beneficial for scenarios requiring the performance optimizations and robustness provided by the language.

Hyperparameter	**Value**
Learning Rate	0.1
Max Depth	50
Number of Trees	100
Max Number of Bins	255
Objective	logistic regression

Table 5.1: Hyperparameters of the LightGBM Model

5.2 Kernel Hazard Estimation Validation

We use a real-world IBM trace from [12], to test the validity of the non-parametric hazard estimator. Details about the trace are provided in Table 5.2.

Trace length	3.7 million
Unique objects	5638

Table 5.2: IBM Web Access Trace Collected from a Gateway Router

For our experiment, we derive the upper bound on hit probability using the HR-E ordering rule with three different estimators. The first method employs the non-parametric estimator introduced earlier in section 4.1. The study by [12] effectively estimated the hazard rate for each object in the IBM trace, assuming a Generalized Pareto distribution for inter-request times. We include the HR-E upper bound calculated under their estimator for validation. Additionally, we explore the HR-E upper bound assuming request processes follow a Poisson process. For further comparison, we also present the object hit probabilities attained by the LRU and Belady's algorithms. The results of these comparisons, run under 3 different cache sizes, is illustrated in Figure 5.1.

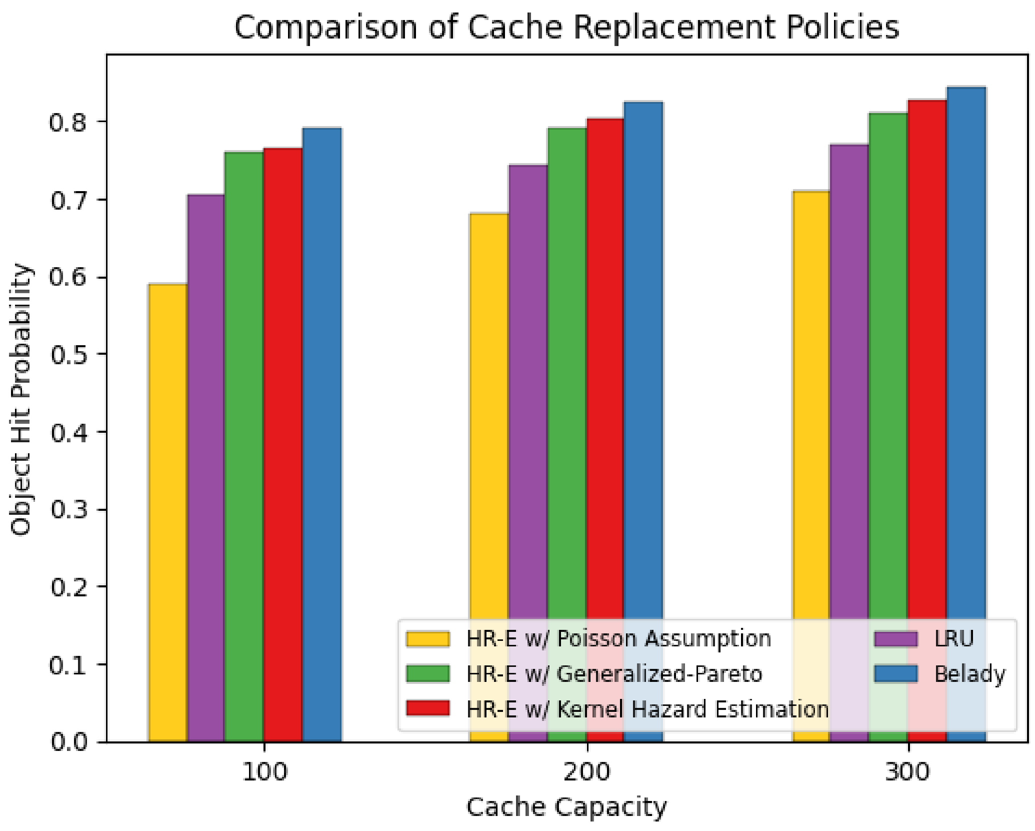


FIGURE 5.1: HR-E Upper Bound Comparisons and Hit Probabilities for LRU and Belady's Algorithms Across Three Cache Sizes.

As can be seen, the kernel hazard estimation method we use gives us an upper bound that aligns with the expected bound derived using the 'good' parametric estimator of the Generalized Pareto distribution. This confirms that kernel hazard estimation is indeed suitable for our use case. As anticipated, the simplistic nature of the Poisson assumption results in LRU outperforming it. Moreover, our results reaffirm the tighter bound achieved by the HR-E rule compared to the Belady algorithm, consistent with the findings in [12].

5.3 Preliminary Evaluation

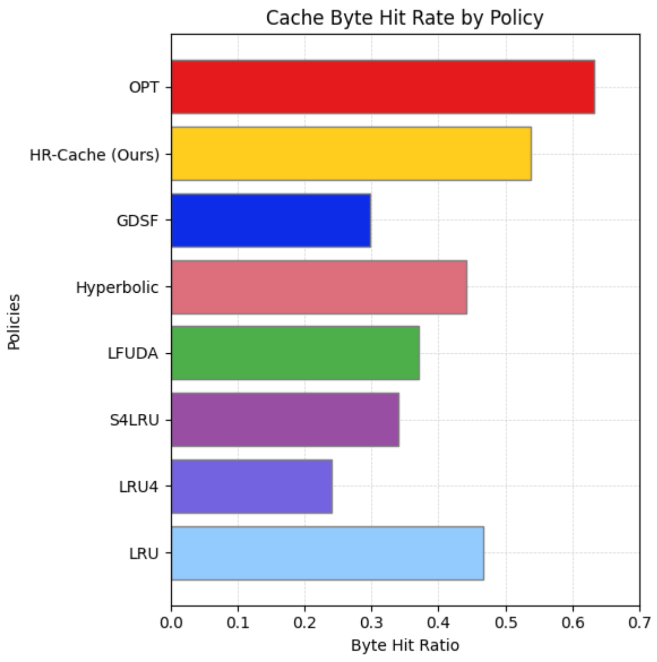


FIGURE 5.2: Comparison of HR-Cache to State-of-the-Art Heuristic Caching Systems For the IBM trace.

For a preliminary evaluation, we use the IBM request trace to assess the effectiveness of our learning framework without the sampling and training window considerations. Given the trace's limited length, we use the initial one million requests to derive HRO decisions as outlined in Section 4.2. Subsequently, we train a model based on these decisions and apply the HR-Cache policy to evaluate the byte hit ratio on the remainder of the trace. As depicted in Figure 5.2, HR-Cache demonstrates its effectiveness by achieving a Byte Hit Ratio that surpasses the state-of-the-art heuristic policies, even within the limited range of this relatively short trace.

5.4 Workloads

Table 5.3: Summary of the traces used in our evaluation.

		Wiki 2018	**Wiki 2019**	**Cloud Physics**	**EU**
Total Requests		84 million	90 million	27 million	100 million
Unique Objects Requested		7 million	11 million	8 million	41 million
Total Bytes Requested		2.6 TB	3.4 TB	360 GB	100 TB
Unique Bytes Requested		0.75 TB	1 TB	86 GB	38 TB
Request Obj Size	**Mean**	34 KB	41 KB	14 KB	1 MB
	Max	674 MB	558 MB	1 MB	7 MB

Our evaluation uses a set of four distinct traces to create a diverse testing environment for the HR-Cache system. This includes two public CDN production traces from Wikipedia for the years 2018 and 2019 [10], a public trace from CloudPhysics [45], and a synthetic trace generated using the JEDI tool in [46, 47]. The selection of these traces aims to represent the performance of HR-Cache across a wide spectrum of real-world and synthetic workloads. Detailed descriptions of each trace source are as follows:

1. **Wikipedia Traces (2018 and 2019)**: These traces are sourced from Content Delivery Network nodes in a metropolitan area in 2018 and 2019, respectively. They mainly consist of web and multimedia content, including images and videos, catering to Wikipedia pages. To reflect the typical environment of edge caches, our evaluations on these traces are conducted

with cache sizes of 16 GB, 32 GB, 64 GB, and 128 GB, aligning with the characteristics of smaller cache sizes often found in edge caches [8].

2. **CloudPhysics Trace**: A Block I/O trace from [45], capturing the activity of VMware virtual disks. This trace introduces a more diverse workload for our analysis, extending beyond the typical CDN scenarios. In our analysis of this trace, we chose cache sizes of 1 GB, 4 GB, 8 GB, and 16 GB, reflecting common configurations in virtual machine environments.

3. **EU Synthetic Trace**: This trace is generated using the JEDI tool [46] which produces traces that have similar caching properties and object-level properties as original production traces. We use the "eu" traffic class which is tailored to replicate the traffic patterns observed in an Akamai's production CDN, specifically those serving content related to social media. For this trace, we use cache sizes of 256 GB, 512 GB, 1 TB, and 2 TB. This decision is based on the trace's large working set size, where smaller cache sizes would not be effective or meaningful for performance analysis.

Table 5.3 summarizes key properties of the four traces.

5.5 State-of-the-art algorithms

In our evaluation, HR-Cache is compared with eleven state-of-the-art caching algorithms: LRB, LRU, LRU-4, S4LRU, GDSF, LFUDA, AdaptSize, Hyperbolic, LHD, LeCaR, and UCB. To enhance readability, we only present the results for the six best-performing algorithms compared to LRU, but give an overview of each algorithm here. Detailed results for all the algorithms are available in the supplementary chapter A, providing a comprehensive comparison. These six best-performing algorithms can be divided into two categories: 1) learning-based algorithms, which include LRB [10] (covered in section 2.1), LeCaR [48], and UCB [49]; and 2) heuristics-based algorithms, comprising LRU-4 [50], LFUDA [51], and S4LRU[52].

LRU (Least Recently Used): LRU is a widely used caching policy that evicts the least recently used items first. In an LRU cache, when a new item needs to be inserted and the cache is full, the item that hasn't been accessed for the longest time is removed to make space. This approach is based on the idea that items used recently are more likely to be needed again soon. LRU is popular due to its simplicity and reasonable assumption about access patterns in many scenarios, but it may not always be the most efficient choice, especially in systems where access patterns change rapidly or are not well-predicted by recent usage.

LRU-K: The LRU-K algorithm is an extension of the traditional Least Recently Used (LRU) caching strategy [50]. In LRU-K:

- "K" refers to a predefined number that indicates how many past references of each object the algorithm keeps track of.
- When an object is accessed, the time of access is recorded. The algorithm maintains the last K timestamps of accesses for each object.
- The object with the oldest K-th reference is the one replaced when a new object needs to be loaded into the cache and there's no more space.

This allows LRU-K to discriminates between objects with different levels of reference frequency. In this comparison, we choose K=4 and as such will refer to the algorithm as LRU-4.

S4LRU (Segmented LRU): S4LRU algorithm is an advanced variation of the traditional LRU caching mechanism, designed to improve cache hit rates by segmenting the cache into four distinct queues, numbered 0 to 3 [52]. This structure offers a more dynamic approach to handling cache items based on their access patterns. Upon a cache miss, when a requested item is not in the cache, S4LRU places this item at the start of queue 0. This is the entry level for new or less recently accessed items. If an item is accessed while it's already in the cache (a cache hit), it gets promoted to the beginning of the next higher-level queue. For items in queue 3, the highest level, a cache hit simply moves them to the front of the same queue. Each queue is allocated a fixed portion, precisely one-fourth, of the total cache size. To maintain this allocation, if a queue exceeds its capacity due to a new addition, the item at the end of that queue is demoted to the head of the immediately lower queue. The eviction process begins at queue 0, where items at the tail are removed from the cache when space needs to be freed.

GDSF (Greedy-Dual Size Frequency): GDSF is a caching policy that extends Greedy-Dual by considering both the size and frequency of items. In GDSF, each item in the cache is assigned a priority value based on a combination of its size, the cost of fetching it (e.g., from a disk), and the frequency of its access. The priority of an item increases with each access, reflecting its frequency of use. Additionally, smaller items are given higher priority since they consume less cache space, allowing more items to be stored in the cache. When the cache is full and a new item needs to be added, the algorithm evicts the item with the lowest priority value to make space. This approach ensures that frequently accessed items, especially those that are smaller and hence more efficient to store, are retained in the cache for longer periods. On the other hand, larger items or those that are nfrequently accessed are more likely to be evicted.

LFUDA (Least Frequently Used with Dynamic Aging): The Least Frequently Used with Dynamic Aging (LFUDA) algorithm is an enhancement of the Least Frequently Used (LFU) caching strategy. It addresses a key flaw of LFU - the preference for older, possibly outdated data. LFUDA operates by counting each object's access frequency, similar to LFU, but introduces a dynamic aging mechanism that gradually reduces the access count of each object over time. This aging effect makes older objects more susceptible to eviction, even if they were frequently accessed in the past. As a result, LFUDA adapts more effectively to changing access patterns, ensuring that the cache remains aligned with the most current and relevant data. This approach balances the simplicity of LFU with the need to accommodate temporal shifts in data access trends.

AdaptSize: AdaptSize[53] is a caching system that reduces the probability of caching large objects, so as to increase the hit rate of smaller, more frequently accessed ones by using a Markov cache model for size-aware cache admission policy.

Hyperbolic: The Hyperbolic caching algorithm is a variant of caching policy designed to improve upon traditional approaches like Least Recently Used (LRU) and Least Frequently Used (LFU). It selects items for eviction based on a combination of how frequently and how recently they were accessed, using a hyperbolic function.

LHD: LHD [54] operates by forecasting the hit density of each object, which is the expected number of hits per unit of space it consumes. It achieves this through conditional probability modeling, which assesses objects' potential contribution to the overall cache hit rate. By continuously monitoring objects and their characteristics (such as age, frequency, application id, and size), LHD can dynamically adjust its eviction strategy to align with evolving application workloads.

LeCaR (Learning Caching Replacement): LeCaR [48] is a learning-based caching policy that combines the benefits of both recency (LRU) and frequency (LFU) based approaches. It uses reinforcement online learning with regret minimization to adaptively adjust the importance of recency versus frequency under varying access patterns, aiming to optimize cache performance dynamically.

UCB: UCB [49] is a learning-based algorithm that applies the Upper Confidence Bound algorithm from reinforcement learning for cache management.

Experimental Setup

All experiments are run on a Google Cloud server with 24 E2-v CPUs (12 shared physical cores) and 64 GB of RAM. Unless specified otherwise, the reported results for HR-Cache are based on the settings that HR-Cache operates in batch-mode inference with a batch size of 128. We also set the frequency decay factor to 0.9 for the decayed frequency feature.

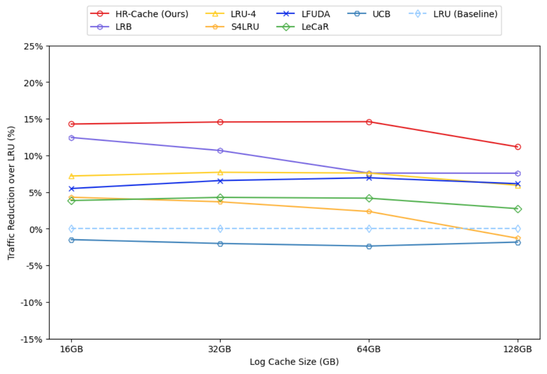
We note that the LRB algorithm was run using its default window parameter. The longer duration of this default memory window, in comparison to the lengths of our traces and the sizes of our caches, might have a bearing on its performance. However, any such influence is expected to be advantageous, which contributes to a balanced comparison in our study.

In all our experiments, the initial training window, during which HR-Cache reverts to LRU, is considered a warm-up phase. We report the metrics for HR-Cache and other algorithms after this period. Notably, LRB starts its training ahead of our framework, and thus, this warm-up phase provides enough time for its training phase to start.

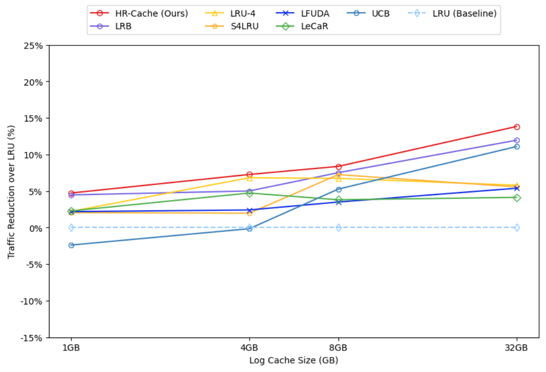
5.6 Main Results

We compare HR-Cache with the caching algorithms detailed in Section 5.5, across various cache sizes. Our primary metric for comparison is the percentage of traffic offload, which indicates the reduction in traffic from downloading content via backhaul links in comparison to LRU. Figure 5.3 shows the reduction in wide-area network (WAN) traffic for each algorithm, relative to LRU, across different cache sizes and the four traces. To ensure a comprehensive analysis, we have also included a comparison of the byte miss ratios for HR-Cache and the best-performing policy for each trace in Figure 5.4.

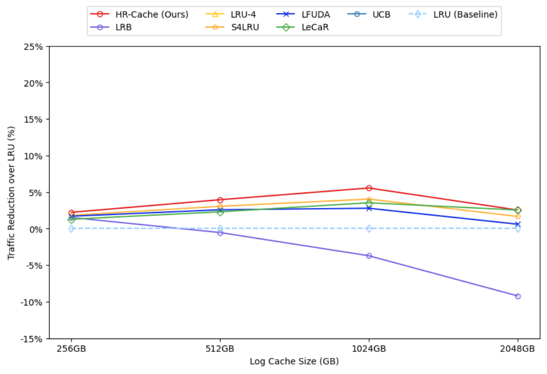
HR-Cache consistently outperforms existing state-of-the-art algorithms, securing the lowest byte miss ratios across various combinations of traces and cache



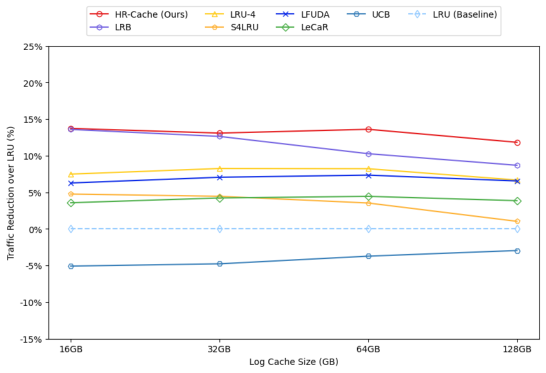
(a) Wikipedia 2018



(b) CloudPhysics



(c) EU Synthetic



(d) Wikipedia 2019

Figure 5.3: WAN Traffic Reduction Compared to LRU Across Various Cache Sizes for HR-Cache and Seven Leading Algorithms. HR-Cache Consistently Achieves 2.2–14.6% Greater WAN Traffic Savings than LRU, Outperforming the SOA Alternatives.

sizes. The sole exception is observed with the EU Synthetic, size 2048, where HR-Cache achieves performance equivalent to that of LeCar. On average, HR-Cache reduces WAN traffic by over 9.7% compared to LRU, with reductions ranging from 2.2–14.6%. Its robust performance is evident across all traces, unlike other algorithms that lack consistent improvements across varying traces and cache sizes.

For instance, LRU-4 improves performance over LRU in 3 of the workloads, but completely underperforms in the EU traces, resulting in a significant 16-24% increase in traffic over LRU (not depicted in the plot due to being below the y-axis). On the other hand, UCB generally underperforms compared to the other algorithms, with a notable exception in CloudPhysics at 16 GB, where it closely rivals HR-Cache and LRB. Shifting focus to LeCaR and LFUDA, these algorithms consistently outperform LRU, yet they do not manage to surpass the effectiveness of other top-performing policies. LRB, on the other hand, exhibits strong results on the Wiki traces, however, it performs the same or falls short in comparison to HR-Cache even where it performs best. Moreover, LRB is outperformed by heuristic algorithms in an instance of CloudPhysics and EU Synthetic and undergoes a significant decrease in effectiveness in the EU Synthetic trace, particularly as cache sizes increase.

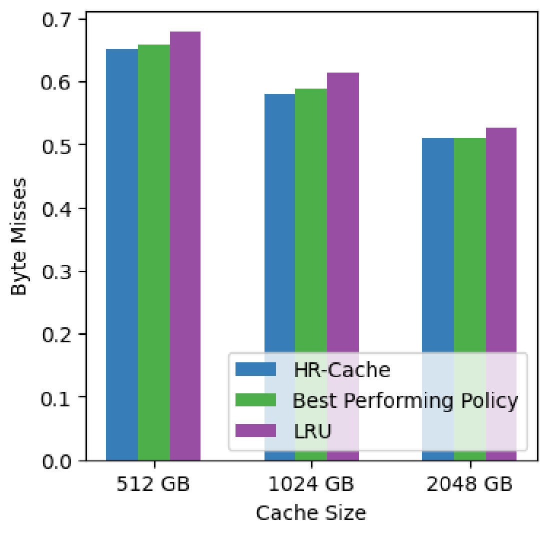
Furthermore, it is important to note that the pattern of WAN traffic reduction achieved by HR-Cache does not consistently correlate with cache capacity. For instance, in the EU Synthetic trace, we observe that the traffic reduction effectively doubles when moving from 256 GB to 1 TB. Conversely, in CloudPhysics, HR-Cache's reduction over LRU generally shows an increasing trend, yet there are instances where the improvement trend inversely declines. This variability suggests that the traces used in our study encompass a diverse array of request patterns, influencing the performance dynamics of HR-Cache differently across scenarios.

Overall, these results suggest that heuristic-based algorithms excel with specific patterns but falter with others. A similar trend is observed among the learning-based algorithms we evaluated. UCB generally underperforms across the board, and LeCaR struggles to match the performance of state-of-the-art alternatives. LRB, although demonstrating strengths in certain scenarios, does not consistently show improvement, underscoring the variability in its efficacy. Figure 5.4 demonstrates this observation as well, showing that HR-Cache consistently outperforms the best performing policy across traces, whereas no single policy consistently emerges as the best across all scenarios.

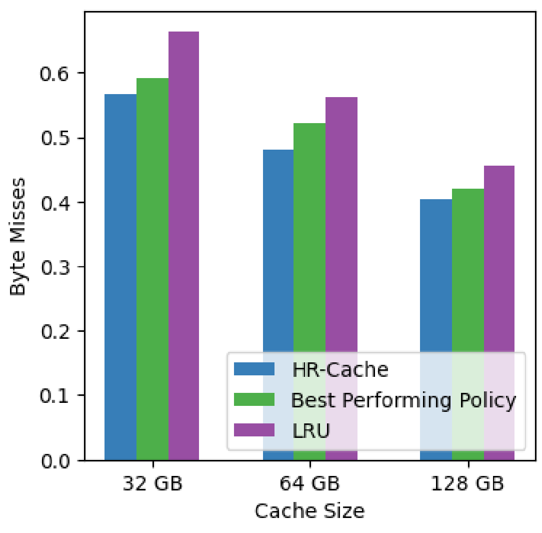
5.7 Prediction Overhead Optimization

In this section, we analyze the additional prediction overhead introduced by HR-Cache in comparison with the state-of-the-art LRB algorithm. To understand the source of this overhead in both LRB and HR-Cache, we examine this overhead for the Wiki 2018 trace.

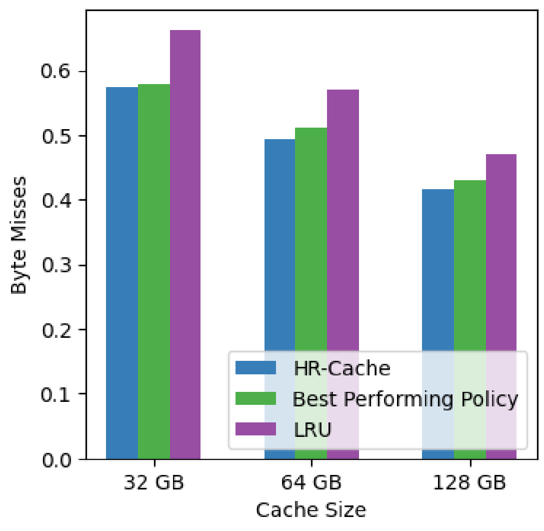
LRB incurs prediction overhead by running predictions on 64 samples for each eviction event. In contrast, HR-Cache requires a prediction for each incoming request to determine cache-friendliness. However, HR-Cache's batch mode significantly reduces this requirement by enabling inference on every 128 requests, rather than on each individual request. As both frameworks utilize the GBDT



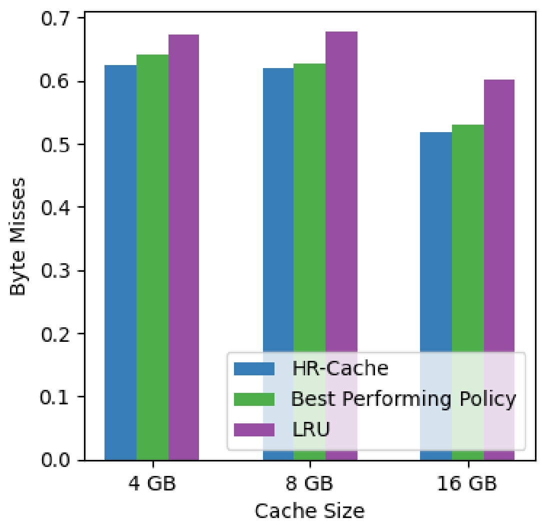
(a) EU Synthetic



(b) Wiki 2018



(c) Wiki 2019



(d) CloudPhysics

Figure 5.4: Comparison of Byte Miss Ratios for HR-Cache, the Best Performing Policy, and LRU

model, we measure the inference time for batches of 64 (LRB's eviction candidate count) and 128 (HR-Cache's inference batch size) inputs, respectively. The results of these measurements is found in Table 5.4.

Given that LRB is required to run predictions with every eviction event, its

	Prediction batch	Prediction time (μs)
LRB	64	183
HR-Cache	128	220

Table 5.4: Comparison of Prediction Batch Sizes and Prediction Times for LRB and HR-Cache Algorithms

prediction overhead is directly tied to the object miss ratio. For our analysis, we assume the best-case scenario for LRB, where only one object needs to be evicted per cache miss.

Under the Wikipedia 2018 workload for cache sizes of 64 GB and 128 GB, LRB is required to run predictions for 18% and 13% of requests, respectively. In contrast, HR-Cache with a batch size of 128, effectively runs predictions for only 1/128 of requests. Taking this and the measured inference times into account, this translates to a prediction overhead reduction by factors of 19.2x and 13.8x for cache sizes of 64 GB and 128 GB, respectively, when compared to LRB.

Table 5.5: Prediction Overhead Reduction For Wiki 2018

Alg.	Miss Ratio		Pred Time ($\mu s/req$)		Reduction Factor	
	64 GB	128 GB	64 GB	128 GB	64 GB	128 GB
LRB	0.18	0.13	32.94	23.8	-	-
HR-Cache	-	-	1.72	1.72	19.2x	13.8x

Another aspect of overhead comes from the process of feature building. HR-Cache constructs one feature per request, while LRB, in contrast, needs to build 64 features on each object miss. This difference results in a significant reduction of overhead for HR-Cache. Specifically, under the Wiki 2018 workload for cache sizes of 64 GB and 128 GB, HR-Cache achieves a reduction in feature-building overhead by factors of 11.5x and 8.3x, respectively, compared to LRB.

To illustrate HR-Cache's computational burden, consider the Wiki 2018 trace with a cache size of 64: replaying 84 million requests, conducting frequent training and inference, and updating our cache based on these predictions, takes approximately 13 minutes, which is more than acceptable given the inter-arrival request rates for objects.

5.8 Ablation Study

In Section 4.2, we discussed how hit or miss outcomes determined by hazard ordering may not directly correspond to cache decisions. This is because Hazard Rate Ordering (HRO) assumes objects with the highest hazard rates are always prefetched and available in the cache whenever a request occurs at time t. Therefore, if a request for an object at time t is a hit, we previously classify it as cache-friendly in its last request.

We also noted that modeling the request process as a Poisson process is a simplification, even though it offers a less complex method for calculating hazard rates. Under this process, the hazard rate remains constant. In our study, we conduct an ablation analysis on three of the traces, where we remove the look-back option in one scenario. In another, we operate HR-Cache under the Poisson assumption, as opposed to using non-parametric kernel hazard estimation.

The results from our ablation analysis, presented in Figure 5.5, offer valuable insights into the effectiveness of our method. Notably, the inclusion of the look-back option in HR-Cache significantly impacts its performance across all traces.

This finding underscores the validity of our initial assumption regarding the importance of translating the pre-fetching nature of the HRO rule into cache decisions. It also affirms the soundness of our approach in HR-Cache for labeling requests, demonstrating the method's efficacy in enhancing cache performance across various scenarios.

When operating under the Poisson assumption, we observe a notable reduction in performance for the Wiki 2018 trace, while the impact on the CloudPhysics and EU traces was comparatively minimal. This variance in outcomes underscores the hypothesis that the Poisson model's assumptions may not align seamlessly with the diverse realities of real-world trace data. This highlights the importance of our approach in HR-Cache, which opts for a more nuanced and adaptable hazard rate estimation method that is able to handle a broader range of caching scenarios.

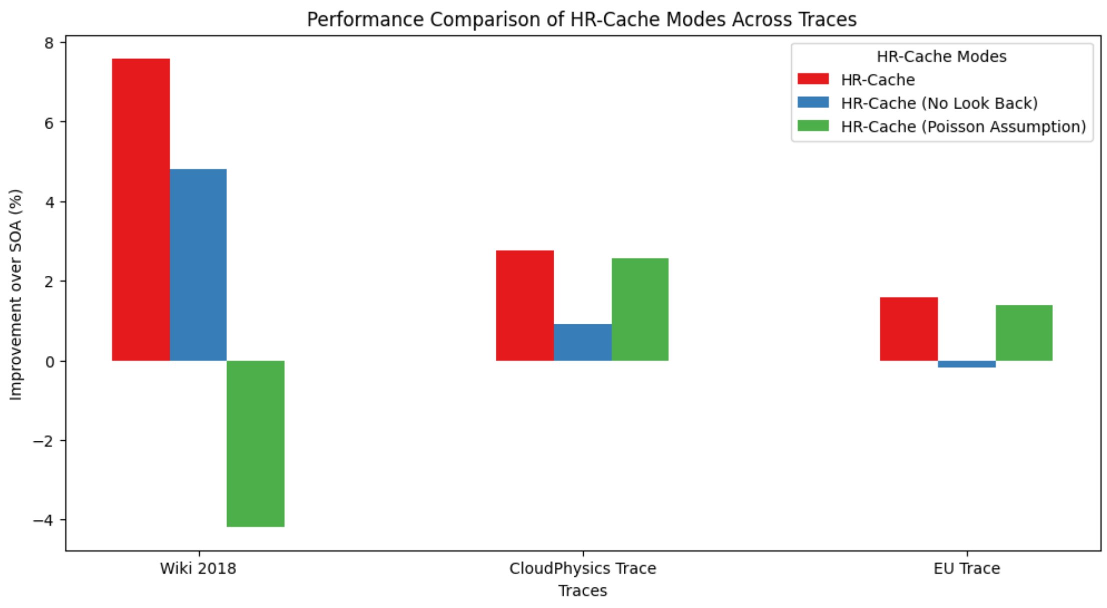


Figure 5.5: Improvement of HR-Cache over the SOA under Ablation Study

Chapter 6

Conclusion

This thesis has introduced and validated HR-Cache, a learning-based caching policy that synergizes non-parametric hazard rate estimation with supervised learning to enhance cache decision-making. Rooted in the hazard rate upper bound principle, HR-Cache has demonstrated its capability to surpass traditional heuristic methods and contemporary learning-based approaches, given the limited cache capacity and workload varying workload characteristics of edge caching environments. The cornerstone of HR-Cache is its ability to learn from hazard rate ordering decisions, enabling it to efficiently discern cache-averse objects and allocate them for eviction, thereby optimizing cache usage. This framework is composed of two integral components: first, the reconstruction of hazard rate ordering for a specific window of requests through kernel hazard estimation; second, the implementation of a decision tree classifier adept at predicting the cache-friendliness of incoming requests.

Our comprehensive evaluation of HR-Cache employed a range of real-world data traces, allowing us to rigorously compare its performance against a variety of leading caching strategies. The results of these evaluations were clear and compelling: HR-Cache not only markedly improved the byte hit rate over traditional LRU methods but also consistently outperformed a broad spectrum of state-of-the-art policies. Notably, it achieved these superior results while maintaining a minimal computational footprint, particularly when compared to the state-of-the-art learning-based caching policy.

6.1 Summary

Our extensive experiments using a diverse array of real-world and synthetic traces have underscored HR-Cache's adaptability and efficiency. The development of a C++ trace-driven framework has enabled accurate assessments of the HR-Cache performance under various scenarios. Moreover, the introduction of efficient resource utilization strategies, such as batched predictions, has been instrumental in reducing computational overheads, making HR-Cache viable for high-demand applications.

We have gained valuable insights into the framework's operational dynamics through an in-depth ablation study. These findings have confirmed the effectiveness of our approach in translating the pre-fetching nature of the Hazard Rate Ordering rule into practical cache decisions, thereby enhancing the overall performance of the caching system.

6.2 Future Work

We envision several promising directions for advancing the capabilities of HR-Cache. Two particularly compelling avenues for future exploration include deploying HR-Cache in distributed caching environments and integrating it into established production cache systems. These developments would not only broaden the scope of HR-Cache's applicability but also offer valuable insights into its performance and adaptability in more complex and demanding real-world scenarios.

6.2.1 Distributed HR-Cache

This work introduced HR-Cache, primarily focused on a single cache scenario. To explore its application in distributed environments, we identify two potential extensions:

Federated Learning Integration Federated learning, a method rapidly gaining traction, particularly in edge computing, offers an intriguing avenue for extending HR-Cache. This approach involves collaboratively learning a shared prediction model while keeping all the training data on the individual caches, aligning perfectly with the data privacy and locality principles of edge computing [55]. By adopting federated learning, each cache node can contribute to and benefit from a more comprehensive and diverse dataset, effectively overcoming the limitations of single-cache data availability. This collective learning process would enable the development of a global model leading to potentially more accurate and generalized cache decision-making.

However, this method isn't without its challenges, particularly the issue of non-IID (independent and identically distributed) data. In federated learning, data across different nodes is often not identically distributed, leading to variability in workload patterns [56]. This non-IID nature of data can result in a globally trained model that might not perform optimally for local scenarios, leading to subpar caching decisions. This discrepancy could diminish the benefits of a federated approach, necessitating a careful balance between global model training and local cache characteristics. Despite these challenges, federated learning integration for HR-Cache represents a fertile ground for future research which can potentially enhance the distributed caching strategies in edge environments.

Hierarchical Cache Application Another intriguing direction for future work involves exploring the application of the Hazard Rate Ordering (HRO) method within a hierarchical cache architecture such as those considered by [57, 58]. In such a setting, a multi-level cache, such as a two-level hierarchy, is considered. Implementing a cache replacement algorithm optimized for an isolated cache in this structure then might not yield optimal overall performance.

The adaptation of HR-Cache for a multi-level system would involve extending the principles of learning-based cache management to each level of the hierarchy. This process necessitates a thorough investigation into how the HRO method can be effectively applied in a network of caches. Such an exploration would not only include the development of individual cache models for each level but also a comprehensive strategy that optimizes cache utilization across the entire hierarchy. Furthermore, the interplay between cache levels in a hierarchical system could lead to complex dynamics that require advanced modeling techniques and careful consideration of inter-cache dependencies. Ultimately, the aim would be to create a robust, scalable caching framework that leverages the strengths of HR-Cache in a hierarchical setting. By addressing the nuances of cache networks, this extension has the potential to significantly enhance the efficiency and effectiveness of caching strategies, particularly in distributed and edge computing environments where hierarchical caching is prevalent.

These potential extensions of HR-Cache highlight the scalability and adaptability of the framework, opening avenues for enhancing caching strategies in distributed and edge computing scenarios.

6.2.2 Integration into Production Systems

The practical application and integration of HR-Cache into established caching systems, such as Redis [59], Memcached [60], Varnish [61], or Apache Traffic Server [62], represent another significant avenue for future work. This integration would enable a comprehensive assessment of HR-Cache in real-world environments, and would provide insights into its performance and scalability under actual operational conditions.

Integrating HR-Cache into these widely-used systems would involve some key steps:

- **Adaptation and Compatibility:** Modifying HR-Cache to ensure compatibility with the architecture and data structures of these systems. This would involve aligning HR-Cache's interfaces and data handling methods with those of the target systems to ensure seamless integration.

- **Latency and Overhead Analysis:** A crucial aspect of the integration process is assessing the latency and memory overhead introduced by HR-Cache. This analysis would provide valuable insights into the trade-offs between improved caching performance and resource consumption, which is critical for systems where low latency and efficient memory usage are paramount such as edge environments.

- **Scalability Assessment:** Understanding how HR-Cache scales with increasing data volumes and user requests is essential. This involves testing the system under high-load scenarios to ensure that HR-Cache maintains its efficiency and does not become a bottleneck.

The ultimate goal of this integration is to demonstrate the feasibility and benefits of deploying HR-Cache in real-world caching scenarios. This would allow us to validate the effectiveness of HR-Cache in a production setting and also pave the way for its wider adoption in industry-standard caching solutions.

Bibliography

[1] F. Yang, S. Wang, J. Li, Z. Liu, and Q. Sun, “An overview of internet of vehicles,” *China communications*, vol. 11, no. 10, pp. 1–15, 2014.

[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, “On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.

[3] T. Taleb, P. A. Frangoudis, I. Benkacem, and A. Ksentini, “Cdn slicing over a multi-domain edge cloud,” *IEEE Transactions on Mobile Computing*, vol. 19, no. 9, pp. 2010–2027, 2019.

[4] T. Barnett, S. Jain, U. Andra, and T. Khurana, “Cisco visual networking index (vni) complete forecast update, 2017–2022,” *Americas/EMEAR Cisco Knowledge Network (CKN) Presentation*, pp. 1–30, 2018.

[5] E. Nygren, R. K. Sitaraman, and J. Sun, “The akamai network: a platform for high-performance internet applications,” *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[6] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, *et al.*, “Scaling memcache at facebook,” in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pp. 385–398, 2013.

[7] W. Reese, “Nginx: the high-performance web server and reverse proxy,” *Linux Journal*, vol. 2008, no. 173, p. 2, 2008.

[8] D. Liu, B. Chen, C. Yang, and A. F. Molisch, “Caching at the wireless edge: design aspects, challenges, and future directions,” *IEEE Communications Magazine*, vol. 54, no. 9, pp. 22–28, 2016.

[9] M. S. ElBamby, M. Bennis, W. Saad, and M. Latva-Aho, “Content-aware user clustering and caching in wireless small cell networks,” in *2014 11th International symposium on wireless communications systems (ISWCS)*, pp. 945–949, IEEE, 2014.

[10] Z. Song, D. S. Berger, K. Li, A. Shaikh, W. Lloyd, S. Ghorbani, C. Kim, A. Akella, A. Krishnamurthy, E. Witchel, *et al.*, “Learning relaxed belady for content distribution network caching,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pp. 529–544, 2020.

[11] K. Mokhtarian and H.-A. Jacobsen, “Caching in video cdns: Building strong lines of defense,” in *Proceedings of the ninth European conference on computer systems*, pp. 1–13, 2014.

[12] N. K. Panigrahy, P. Nain, G. Neglia, and D. Towsley, “A new upper bound on cache hit probability for non-anticipative caching policies,” *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 7, no. 2-4, pp. 1–24, 2022.

[13] D. S. Berger, “Towards lightweight and robust machine learning for cdn caching,” in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pp. 134–140, 2018.

[14] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[15] A. Jain and C. Lin, “Back to the future: Leveraging belady’s algorithm for improved cache replacement,” *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 78–89, 2016.

[16] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying deep learning to the cache replacement problem,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 413–425, 2019.

[17] E. Liu, M. Hashemi, K. Swersky, P. Ranganathan, and J. Ahn, “An imitation learning approach for cache replacement,” in *International Conference on Machine Learning*, pp. 6237–6247, PMLR, 2020.

[18] D. S. Berger, N. Beckmann, and M. Harchol-Balter, “Practical bounds on optimal caching with variable object sizes,” *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 2, pp. 1–38, 2018.

[19] G. Yan, J. Li, and D. Towsley, “Learning from optimal caching for content delivery,” in *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pp. 344–358, 2021.

[20] X. Hu, E. Ramadan, W. Ye, F. Tian, and Z.-L. Zhang, “Raven: belady-guided, predictive (deep) learning for in-memory and content caching,” in *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pp. 72–90, 2022.

[21] A. V. Aho, P. J. Denning, and J. D. Ullman, “Principles of optimal page replacement,” *Journal of the ACM (JACM)*, vol. 18, no. 1, pp. 80–93, 1971.

[22] N. K. Panigrahy, “Resource allocation in distributed service networks,” 2021.

[23] D. J. Daley, *Introduction to the Theory of Point Processes: Elementary Theory and Methods*. Springer, 2014.

[24] W. Ali, S. M. Shamsuddin, and A. S. Ismail, “Intelligent web proxy caching approaches based on machine learning techniques,” *Decision Support Systems*, vol. 53, no. 3, pp. 565–579, 2012.

[25] P. Blasco and D. Gündüz, “Learning-based optimization of cache content in a small cell base station,” in *2014 IEEE international conference on communications (ICC)*, pp. 1897–1903, IEEE, 2014.

[26] C. Zhong, M. C. Gursoy, and S. Velipasalar, “A deep reinforcement learning-based framework for content caching,” in *2018 52nd Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6, IEEE, 2018.

[27] S. Li, J. Xu, M. Van Der Schaar, and W. Li, “Popularity-driven content caching,” in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pp. 1–9, IEEE, 2016.

[28] S. Borst, V. Gupta, and A. Walid, “Distributed caching algorithms for content distribution networks,” in *2010 Proceedings IEEE INFOCOM*, pp. 1–9, IEEE, 2010.

[29] K. Poularakis and L. Tassiulas, “On the complexity of optimal content placement in hierarchical caching networks,” *IEEE Transactions on Communications*, vol. 64, no. 5, pp. 2092–2103, 2016.

[30] S. S. Tanzil, W. Hoiles, and V. Krishnamurthy, “Adaptive scheme for caching youtube content in a cellular network: Machine learning approach,” *Ieee Access*, vol. 5, pp. 5870–5881, 2017.

[31] A. Sadeghi, F. Sheikholeslami, and G. B. Giannakis, “Optimal and scalable caching for 5g using reinforcement learning of space-time popularities,” *IEEE Journal of Selected Topics in Signal Processing*, vol. 12, no. 1, pp. 180–190, 2017.

[32] A. Sadeghi, G. Wang, and G. B. Giannakis, "Deep reinforcement learning for adaptive caching in hierarchical content delivery networks," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 4, pp. 1024–1033, 2019.

[33] X. Wang, C. Wang, X. Li, V. C. Leung, and T. Taleb, "Federated deep reinforcement learning for internet of things with decentralized cooperative edge caching," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9441–9455, 2020.

[34] S. Tuli and G. Casale, "Optimizing the performance of fog computing environments using ai and co-simulation," in *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering*, pp. 25–28, 2022.

[35] G. Garbi, E. Incerto, and M. Tribastone, "Learning queuing networks by recurrent neural networks," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, pp. 56–66, 2020.

[36] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska, "Bao: Learning to steer query optimizers," *arXiv preprint arXiv:2004.03814*, 2020.

[37] V. Nathan, J. Ding, M. Alizadeh, and T. Kraska, "Learning multi-dimensional indexes," in *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pp. 985–1000, 2020.

[38] J. Gao, "Machine learning applications for data center optimization," 2014.

[39] A. Jindal, S. Qiao, R. Sen, and H. Patel, "Microlearner: A fine-grained learning optimizer for big data workloads at microsoft," in *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pp. 2423–2434, IEEE, 2021.

[40] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and J. Wu, "Robust network traffic classification," *IEEE/ACM transactions on networking*, vol. 23, no. 4, pp. 1257–1270, 2014.

[41] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, “Resource management with deep reinforcement learning,” in *Proceedings of the 15th ACM workshop on hot topics in networks*, pp. 50–56, 2016.

[42] H.-G. Muller and J.-L. Wang, “Hazard rate estimation under random censoring with varying kernels and bandwidths,” *Biometrics*, pp. 61–76, 1994.

[43] J.-L. Wang *et al.*, “Smoothing hazard rates,” *Encyclopedia of biostatistics*, vol. 7, pp. 4986–4997, 2005.

[44] V. Kirilin, A. Sundarrajan, S. Gorinsky, and R. K. Sitaraman, “Rl-cache: Learning-based cache admission for content delivery,” in *Proceedings of the 2019 Workshop on Network Meets AI & ML*, pp. 57–63, 2019.

[45] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, “Efficient {MRC} construction with {SHARDS},” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 95–110, 2015.

[46] A. Sabnis and R. K. Sitaraman, “Jedi: model-driven trace generation for cache simulations,” in *Proceedings of the 22nd ACM Internet Measurement Conference*, pp. 679–693, 2022.

[47] A. Sabnis and R. K. Sitaraman, “Tragen: a synthetic trace generator for realistic cache simulations,” in *Proceedings of the 21st ACM Internet Measurement Conference*, pp. 366–379, 2021.

[48] G. Vietri, L. V. Rodriguez, W. A. Martinez, S. Lyons, J. Liu, R. Rangaswami, M. Zhao, and G. Narasimhan, “Driving cache replacement with {ML-based}{LeCaR},” in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[49] R. Costa and J. Pazos, “Mlcache: A multi-armed bandit policy for an operating system page cache,” tech. rep., Technical report, University of British Columbia, 2017.

[50] E. J. O’neil, P. E. O’neil, and G. Weikum, “The lru-k page replacement algorithm for database disk buffering,” *Acm Sigmod Record*, vol. 22, no. 2, pp. 297–306, 1993.

[51] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, “Evaluating content management techniques for web proxy caches,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3–11, 2000.

[52] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li, “An analysis of facebook photo caching,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 167–181, 2013.

[53] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, “{AdaptSize}: Orchestrating the hot object memory cache in a content delivery network,” in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pp. 483–498, 2017.

[54] N. Beckmann, H. Chen, and A. Cidon, “{LHD}: Improving cache hit rate by maximizing hit density,” in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pp. 389–403, 2018.

[55] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, “In-edge ai: Intelligentizing mobile edge computing, caching and communication by federated learning,” *Ieee Network*, vol. 33, no. 5, pp. 156–165, 2019.

[56] H. Zhu, J. Xu, S. Liu, and Y. Jin, “Federated learning on non-iid data: A survey,” *Neurocomputing*, vol. 465, pp. 371–390, 2021.

[57] H. Che, Y. Tung, and Z. Wang, “Hierarchical web caching systems: Modeling, design and experimental results,” *IEEE journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[58] X. Li, X. Wang, P.-J. Wan, Z. Han, and V. C. Leung, “Hierarchical edge caching in device-to-device aided mobile networks: Modeling, optimization, and design,” *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 8, pp. 1768–1785, 2018.

[59] J. Carlson, *Redis in action*. Simon and Schuster, 2013.

[60] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, *et al.*, “Memcached design on high performance rdma capable interconnects,” in *2011 International Conference on Parallel Processing*, pp. 743–752, IEEE, 2011.

[61] P.-H. Kamp, “Notes from the architect.” Available at: `https://varnish-cache.org/docs/trunk/phk/notes.html`, 2006. Accessed: 28/11/2023.

[62] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, “Traffic model and performance evaluation of web servers,” *Performance Evaluation*, vol. 46, no. 2-3, pp. 77–100, 2001.

Appendix A

Chapter 5 Supplement

Table A1.1: Performance Comparison on Wikipedia 2018 Trace

	16GB	**32GB**	**64GB**	**128GB**
HR-Cache (Ours)	0.64157	0.5665	0.4810	0.4045
LRB (SOA)	0.6553	0.5923	0.5205	0.4209
LRU	0.7485	0.6631	0.5633	0.4554
LRU-4	0.6946	0.6120	0.5205	0.4284
S4LRU	0.7163	0.6387	0.5500	0.4613
GDSF	0.7496	0.6922	0.6131	0.4950
LFUDA	0.7074	0.6195	0.5241	0.4274
AdaptSize	0.7773	0.7097	0.6234	0.6056
Hyperbolic	0.7750	0.7006	0.6103	0.5071
LHD	0.7596	0.6987	0.6212	0.5239
LeCaR	0.7197	0.6347	0.5398	0.4429
UCB	0.7596	0.6764	0.5766	0.4637

Table A1.2: Performance Comparison on Wikipedia 2019 Trace

	16GB	**32GB**	**64GB**	**128GB**
HR-Cache (Ours)	0.6438	0.5747	0.4924	0.4156
LRB (SOA)	0.6442	0.5777	0.5114	0.4304
LRU	0.7407	0.6613	0.5700	0.4714
LRU-4	0.6852	0.6067	0.5231	0.4399
S4LRU	0.7054	0.6318	0.5498	0.4665
GDSF	0.7478	0.6730	0.5929	0.4999
LFUDA	0.6942	0.6146	0.5281	0.4405
AdaptSize	0.7851	0.7147	0.6228	0.5708
Hyperbolic	0.7697	0.6991	0.6159	0.5230
LHD	0.7671	0.7036	0.6286	0.5372
LeCaR	0.7142	0.6333	0.5446	0.4532
UCB	0.7783	0.6928	0.5912	0.4853

Table A1.3: Performance Comparison on CloudPhysics Trace

	1GB	**4GB**	**8GB**	**16GB**
HR-Cache (Ours)	0.7317	0.6248	0.6201	0.5181
LRB (SOA)	0.7337	0.6425	0.6259	0.5294
LRU	0.7681	0.6738	0.6768	0.6013
LRU-4	0.7510	0.6278	0.6312	0.5664
S4LRU	0.7523	0.6605	0.6277	0.5678
GDSF	0.8154	0.7195	0.7025	0.6540
LFUDA	0.7513	0.6575	0.6531	0.5689
AdaptSize	0.9472	0.9093	0.9078	0.8143
Hyperbolic	0.7826	0.6993	0.6851	0.6365
LHD	0.8362	0.7956	0.7858	0.7262
LeCaR	0.7505	0.6419	0.6510	0.5764
UCB	0.7864	0.6748	0.6411	0.5346

TABLE A1.4: Performance Comparison on EU Trace

	256GB	**512GB**	**1TB**	**2TB**
HR-Cache (Ours)	0.7139	0.6510	0.5791	0.5123
LRB (SOA)	0.7191	0.6814	0.6358	0.5737
LRU	0.7302	0.6778	0.6131	0.5254
LRU-4	0.9063	0.8430	0.7415	0.6087
S4LRU	0.7169	0.6571	0.5883	0.5166
GDSF	0.7516	0.6983	0.6310	0.5531
LFUDA	0.7178	0.6603	0.5960	0.5222
AdaptSize	0.9200	0.8540	0.8091	0.7732
Hyperbolic	0.7440	0.6984	0.6379	0.5663
LHD	0.7903	0.7353	0.6724	0.5823
LeCaR	0.7208	0.6622	0.5914	0.5110
UCB	0.9887	0.9792	0.9660	0.9377