Design, Analysis and Optimization of Cache Systems

A Dissertation Presented

by

MOSTAFA DEHGHAN

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

February 2017

College of Information and Computer Sciences

Design, Analysis and Optimization of Cache Systems

A Dissertation Presented

by

MOSTAFA DEHGHAN

Approved as to style and content by:

Don Towsley, Chair

Jim Kurose, Member

Ramesh Sitaraman, Member

Christopher Hollot, Member

James Allan, Chair of the Faculty
College of Information and Computer Sciences

To Mom.

ACKNOWLEDGMENTS

First and foremost, I am truly grateful to have had Professor Donald Towsley as my PhD adviser. He has been an exceptional support and a great source of inspiration. I have learned so many things from him. Each single meeting with Don was worth a semester-long course. I cannot thank him enough.

I would also like to express my deepest gratitude to Professor Dennis Goeckel who advised me for two years before joining the Computer Science department. He is an amazing adviser and a great teacher. I was lucky enough to have the pleasure of working with Professor Jim Kurose before he joined the National Science Foundation. His great personality cannot be put into words. It has been an honor to know him. I would also like to thank my other committee members Professor Ramesh Sitaraman and Professor Kris Hollot. Their comments certainly improved this thesis.

I want to thank my peers and friends in the Computer Science department for making my time at UMass memorable. Misha, Jennie, Anand, Cheng, Bo, Yeon-Sup, James, Fabricio, Bruce, Kun, Keen and Pinar, thanks for being such awesome friends! I am grateful to have known the younger generation of Iranian students in the Computer Science department who called me Dad! So many great memories were created by Hamed, Milad, Shahrzad, Hadi, Pegah, and Fatemeh.

Last, but not the least, I want to thank Leeanne Leclerc for making sure I was in sync with program requirements.

ABSTRACT

Design, Analysis and Optimization of Cache Systems

FEBRUARY 2017

MOSTAFA DEHGHAN

B.Sc., SHARIF UNIVERSITY OF TECHNOLOGY

M.Sc., UNIVERSITY OF CALGARY

M.Sc., UNIVERSITY OF MASSACHUSETTS AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Don Towsley

The increase in data traffic over the past years is predicted to continue more aggressively in the years to come. However, traditional methods such as increasing the amount of spectrum or deploying more base stations are no longer sufficient to cope with the traffic growth. Caching is hence recognized as one of the most effective means to improve the performance of Internet services by bringing content closer to the end-users. Although the benefits of in-network content caching has been demonstrated in various contexts, they introduce new challenges in terms of

modeling and analyzing network performance. Building on analytical results for Time-To-Live caches and the flexibility they provide in modeling caches, this thesis investigates various aspects in which caching affects network design and performance. The complexity of making optimal routing and content placement decisions is studied first. Showing the infeasibility of implementing the optimal strategy, low-complexity techniques are developed to achieve near-optimal performance in terms of the delay observed by end-users. The problem of differentiated cache services is studied next with the question "how can Content Distribution Networks implement caching policies to provide differentiated services to different content providers?". A utility-maximization framework is formulated to design caching policies with fairness considerations and implications on market economy for cache service providers and content publishers. An online algorithm is also developed with the purpose of implementing the utility-based cache policies with no a priori information on the number of contents and file popularities. This thesis also analyzes caches in conjunction with data structures, *e.g.* Pending Interest Table, proposed in the future Internet architecture designs such as Named Data Networking. The analysis provides the means to understand system performance under different circumstances, and develop techniques to achieve optimal performance.

TABLE OF CONTENTS

LIST OF FIGURES

Figure **Page**

CHAPTER 1

INTRODUCTION

The Internet has become a global information depository and content distribution platform, where various types of information or content are stored in the "cloud", hosted by a large array of *content providers*, and delivered or "streamed" on demand. The (nearly) "anytime, anywhere access" of online information or content – especially multimedia content – has precipitated rapid growth in Internet data traffic in recent years, both in wired and wireless (cellular) networks. The increase in data traffic over past years is predicted to continue more aggressively, with global Internet traffic in 2019 estimated to reach 64 times its volume in 2005 [1]. A primary contributor to this rapid growth in data traffic comes from online video streaming services such as Netflix, Hulu, YouTube and Amazon Video, just to name a few. It was reported [61] that Netflix alone consumed nearly a third of the peak downstream traffic in North America in 2012, and it is predicted [1] that nearly 90% of all data traffic will come from video content distributors in the near future.

Massive data traffic generated by large-scale online information access – especially, "over-the-top" video delivery – imposes enormous burden on the Internet and poses many challenging issues. Traditional methods such as increasing the amount of spectrum or deploying more base stations are not sufficient to cope with this

predicted traffic increase [2, 31]. Caching is recognized, in current and future Internet architecture proposals, as one of the most effective means to improve the performance of web applications. By bringing content closer to users, caches greatly reduce network bandwidth usage, server load, and perceived service delays [8].

1.1 Contributions

This thesis considers the problem of network caching by first studying the complexity of caching and routing in heterogeneous cache-enabled networks. Second, a utility-driven caching framework is introduced as a general framework for cache policies enabling Content Distribution Networks (CDNs) to provide differentiated services for different content. Third, the problem of cache management is considered for multiple content providers studying the question whether a cache should be shared among multiple content providers for their content or should the cache be partitioned. Finally, Pending Interest Table (PIT) is studied as an important data structure in future Internet architecture designs. The contributions of this thesis can be summarized as follows:

1.1.1 Complexity Analysis of Caching and Routing

We study the *joint* problem of caching and routing, considering inter-related routing and caching decisions, with the goal of minimizing average content access delay over all user requests. We consider a scenario in which users request content that is permanently stored at a back-end server, and that can be accessed in one of two ways − either directly from the back-end server over an uncached path, or

via one of the caches located within the network. We consider two variants of the problem. In the first case, referred to as the *congestion-insensitive case*, we assume delays are independent of traffic loads on all paths. In the second case, referred to as the *congestion-sensitive case*, we assume delays to the back-end server depends on the traffic load. Our goal is twofold. First, we seek an understanding of the computational complexity of the joint caching and routing problem: Can the general problem be solved optimally in polynomial time? If not, are there problem instances that are tractable and what aspects make the general problem intractable? Second, we seek efficient approximate solutions to the joint caching and routing problem that perform well in practice.

1.1.2 Utility-Driven Caching

Traditional cache management policies such as Least Recently Used (LRU) treat different contents in a strongly coupled manner that makes it difficult for (cache) service providers to implement differentiated services, and for content publishers to account for the valuation of their content delivered through content distribution networks. We propose a utility-driven caching framework, where each content has an associated utility and content is stored and managed in a cache so as to maximize the aggregate utility for all content. Utilities can be chosen to trade off user satisfaction and cost of storing the content in the cache. We draw on analytical results for time-to-live (TTL) caches [26], to design caches with ties to utilities for individual (or classes of) contents. Utility functions also have implicit notions of fairness that dictate the time each content stays in cache. Our framework allows us to develop *online* algorithms for cache management, for which we prove achieve optimal performance.

Our framework has implications for distributed pricing and control mechanisms and hence is well-suited for designing cache market economic models.

1.1.3 Cache Sharing and Partitioning

Content providers (CPs) play a key role in procuring and delivering content in future Information Centric Networks (ICNs). Given that it is part of the ICN data plane substrate, storage or cache must therefore be *multiplexed* or *shared* among multiple CPs, say, multiple (competing) video streaming service providers. This poses a fundamental research question that is pertinent to all ICN designs: *how to share or allocate the cache resource within a single network forwarding element and across various network forwarding elements among multiple content providers so as to maximize the cache resource utilization* or *provide best utilities to content providers?* We show that if all CPs offer *distinct* content objects, *partitioning the cache into slices of appropriate sizes, one slice per CP*, yields the best cache allocation strategy as it maximizes overall cache utility. We argue that partitioning and sharing some partitions achieves the optimal performance for most instances. Going beyond cache utilization efficiency, we further consider a scenario where the utility of each CP is expressed as a monotonically increasing, concave function of the aggregate hit rate of the content objects in its offering, and develop *online* algorithms for solving the multi-CP cache allocation problem that achieve the *optimal performance.*

1.1.4 PIT Modeling and Timeout Optimization

One of the core components of the Named Data Networking (NDN) architecture is the Pending Interest Table (PIT), which performs collapsed forwarding by keeping

track of currently unsatisfied Interest packets. We analyze a cache with a Pending Interest Table, to compute the cache hit probability, response time perceived by the users, and the size of the Pending Interest Table. Unlike previous research that assumes a file is instantaneously downloaded to the cache in case of a cache miss, we assume a non-zero download delay modeled as a random variable.

As the rate of incoming Interest packets increases, the number of PIT entries can significantly increase, demanding a large memory space and degrading PIT performance. To prevent PIT size bloat and ensure efficient I/O operations at line speed, PIT entries are purged after a *timeout* period. An efficient design of the PIT is therefore an important factor in performance of NDN at wire speed, and an accurate assessment of the PIT size is key to achieving this. This has motivated us to study the problem of optimizing the PIT timeout.

1.2 Thesis Outline

The remainder of this thesis is structured as follows. Chapter 2 investigates the problem of optimal request routing and content caching in a heterogeneous network supporting in-network content caching with the goal of minimizing average content access delay. Chapter 3 proposes a utility-driven caching framework, where each content has an associated utility and content is stored and managed in a cache so as to maximize the aggregate utility for all content. Chapter 4 considers the problem of allocating cache resources among multiple content providers, where the cache can be partitioned into slices and each partition can be dedicated to a particular content provider, or shared among a number of them. Chapter 5 analyzes the cache

hit probability, Interest aggregation probability and PIT size as functions of PIT timeout, and develops online algorithms for PIT management. Finally, Chapter 6 concludes the thesis.

CHAPTER 2

ON THE COMPLEXITY OF OPTIMAL ROUTING AND CONTENT CACHING IN HETEROGENEOUS NETWORKS

2.1 Introduction

In-network content caching has received considerable attention in recent years as a means to address the explosive growth in data access seen in today's networks. Its main premise is to store content at the network's edge – close to the end users – to reduce user content access delay and network bandwidth usage. The benefits of in-network content caching have been demonstrated in the context of CDNs [34, 54, 64] as well as hybrid networks comprised of cellular and MANETs or femto-cell networks [3, 57, 63].

In this chapter, we study a *joint* problem of caching and routing, considering the inter-related routing and caching decisions, with the goal of minimizing average content access delay over all user requests. We consider a scenario in which users request content that is permanently stored at a back-end server, and that can be accessed in one of two ways − either directly from the back-end server over an uncached path, or via one of the caches located within the network. These caches can be located either at the network edge as in the case of a CDN, or can be in-network

caches in the case of a hybrid wireless network. In the latter setting, MANET-like routing might be used to route content requests to in-network caches, while a separate (and potentially costly, congested, and/or slower speed) cellular link might be used to directly access the back-end server. If a request is routed to an in-network cache that holds the content, the request is served immediately. Otherwise, the cache must download the content from the back-end server before serving it to the user, incurring an additional delay. Additionally, the cache must decide whether or not to store the downloaded content.

Our goal in this chapter is twofold. First, we seek an understanding of the computational complexity of the joint caching and routing problem: Can the general problem be solved optimally in polynomial time? If not, are there problem instances that are tractable and what aspects make the general problem intractable? Second, we seek efficient approximate solutions to the joint caching and routing problem that perform well in practice.

Our contributions can be summarized as follows:

- We provide a unified optimization formulation for the joint caching and routing problem for the congestion-insensitive and congestion-sensitive models and prove that the problem is NP-complete in both cases.

- For the congestion-insensitive uncached path model, we show that the optimal solution can be found in polynomial time if each content is requested by only one user, or when the number of caches in the network is at most two. Moreover, we identify the root cause of the problem complexity in general cases − cycles with an odd number of users and caches in the bipartite graph representing

connections between users and caches. For the congestion-sensitive uncached path model, however, we show that the problem remains NP-complete even if there is only one cache in the network and each content is requested by only one user.

- We develop a greedy caching and routing algorithm that achieves an average delay within a $(1 - 1/e)$ factor of the optimal solution and a second greedy algorithm of lower complexity.

- We evaluate the performance of the proposed greedy algorithms together with the optimal solution (via brute-force search) and a baseline solution based on LRU through numerical evaluations and trace-driven simulations. Numerical results show that the greedy algorithms perform close to optimal when computing the optimal solution is feasible. Results from trace-driven simulations show that the greedy algorithms yield significant performance improvement over solutions based on traditional LRU caching policy.

2.2 Network Model

Consider a network with N users generating requests for a set of K unique unit size files, $F = \{f_1, f_2, \ldots, f_K\}$. Throughout this chapter, we will use the terms content and file interchangeably. We assume that these files reside permanently at the back-end server. As shown in Figure 2.1, there are M caches in the network that can serve user requests.

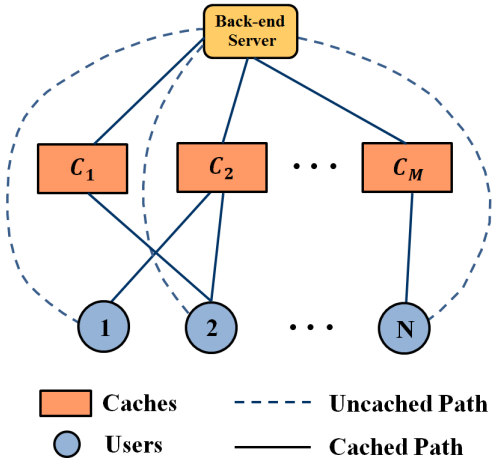


Figure 2.1: Hybrid network with in-network caching

All files are available at the back-end server and users are directly connected to this server via a cellular infrastructure. We refer to the cellular path between the user and the back-end server as the *uncached path*. Each user can also access a subset of the M in-network caches where the content might be cached. We refer to a connection between a user and a cache as a *cached path*.

Let C_m denote the storage capacity of the m-th cache measured by the maximum number of files it can store. If user i requests file j and it is present in the cache, then the request is served immediately. We refer to this event as a cache hit. However, if content j is not present in the cache, the cache then forwards the request to the back-end server, downloads file j from the back-end server and forwards it to the user. We refer to this event as a cache miss, since it becomes necessary to download content from the back-end server in order to satisfy the request. *Note that in case of a cache miss, the cache may or may not store the downloaded content.*

User i generates requests at rate λ_i for the files in F according to a Poisson process. The aggregate request rate of all users is λ. We assume the independent reference model (IRM) and denote by q_{ij} the probability that a request generated by user i is for file j (referred to as the *file popularity*). The popularity of the same file can vary from one user to another.

Let a_{im} denote the existence of a connection between user i and cache m, with $a_{im} = 1$ if user i is connected to cache m, and $a_{im} = 0$, otherwise. We consider two models for the delay over the paths to the back-end server. The first is a congestion-insensitive delay model where delays are independent of traffic loads on links to the back-end server. In this case, the average delay experienced for a request by user i sent over the uncached path is d_i^b. Also, for the user-cache connections, we denote the average delays incurred by user i in the event of a cache hit or miss at cache m by d_{im}^h and d_{im}^m, respectively. We assume that $d_{im}^h < d_i^b < d_{im}^m$ if $a_{im} = 1$. The second model is a congestion-sensitive delay model where delays experienced over the paths to the back-end server depend on the traffic load. In this case, we assume that the requests sent over the uncached paths, and the requests missed from caches experience constant initial delays d_i^b and d_{im}^m as well as load-dependent (queueing) delays captured by convex functions $d_b(\cdot)$ and $d_c(\cdot)$, respectively.

2.3 Problem Formulation

In this work, we consider a joint caching and routing problem with the goal of minimizing average content access delay over all user requests for all files. The solution to this problem requires addressing two closely-related questions 1) How

should contents be managed in the cache - which files should be kept in the caches, and what cache replacement strategy should be used? and 2) How should users route their requests between the cached and uncached paths?

For our routing policy, we introduce a decision variable p_{ijm} that denotes the fraction of user i requests for content j sent to cache m. User i sends the remaining $1 - \sum_m p_{ijm}$ fraction of her requests for content j to the back-end server through the uncached path.

It is shown in [44] that *static caching* minimizes expected delay for a single cache when user demands and routing are fixed. With static caching, a set of files is stored in the cache, and the cache content does not change in the event of a cache hit or miss. The argument in [44] was extended in [18] and [19] to a network of caches to show that static caching achieves minimum expected delay under a fixed routing policy. Hence, we introduce binary variables $x_{jm} \in \{0, 1\}$ to denote the content placement in caches, where $x_{jm} = 1$ indicates file j is stored in cache m and $x_{jm} = 0$ indicates otherwise.

We denote by $D(\mathbf{x}, \mathbf{p})$ the expected delay obtained by a content placement strategy $\mathbf{x} = [x_{jm}]$, and a routing strategy $\mathbf{p} = [p_{ijm}]$. We also use $D_\varnothing$ to denote the expected delay when no content is cached, where $D_\varnothing$ is assumed to be finite. The caching *gain* can then be defined as $G(\mathbf{x}, \mathbf{p}) \triangleq D_\varnothing - D(\mathbf{x}, \mathbf{p})$. The goal of joint caching and routing is to maximize $G(\mathbf{x}, \mathbf{p})$ which can equivalently be obtained by solving the following Mixed-Integer Program (MIP):

$$
\begin{aligned}
\text{minimize } D(\mathbf{x},\mathbf{p}) = \frac{1}{\lambda}\Bigg[& \sum_i \sum_j \lambda_i q_{ij} \Bigg(\sum_m p_{ijm} x_{jm} d_{im}^h \\
& + (1 - \sum_m p_{ijm}) d_i^b + \sum_m (1 - x_{jm}) p_{ijm} d_{im}^m \Bigg) \\
& + \lambda_b(\mathbf{p}) d_b\Big(\lambda_b(\mathbf{p})\Big) + \lambda_c(\mathbf{x},\mathbf{p}) d_c\Big(\lambda_c(\mathbf{x},\mathbf{p})\Big) \Bigg], \\
\text{such that } & \sum_m p_{ijm} \leq 1 \quad \forall i,j \\
& \sum_j x_{jm} \leq C_m \quad \forall m \\
& x_{jm} \in \{0,1\} \quad \forall j,m \\
& 0 \leq p_{ijm} \leq a_{im} \quad \forall i,j,m.
\end{aligned} \tag{2.1}
$$

In the formulation above,

$$\lambda_b(\mathbf{p}) \triangleq \sum_i \sum_j \lambda_i q_{ij} (1 - \sum_m p_{ijm}),$$

and

$$\lambda_c(\mathbf{x},\mathbf{p}) \triangleq \sum_i \sum_j \lambda_i q_{ij} \sum_m (1 - x_{jm}) p_{ijm},$$

denote the request load over the uncached paths, and the load of requests missed from caches, respectively.

In the next two sections, we express the delay function $D(\mathbf{x},\mathbf{p})$ for the cases of *i)* congestion-insensitive and *ii)* congestion-sensitive uncached path delay models, and discuss why the joint caching and routing problem is NP-complete.

2.4 Congestion-Insensitive Uncached Path

First, we consider the case where delays on the uncached path, d_i^b, do not depend on the traffic load on the back-end server. Hence, throughout this section we assume that $d_b(\cdot) = d_c(\cdot) = 0$.

Without loss of generality, we assume that $d_{im}^h < d_i^b < d_{im}^m$ whenever user i is connected to cache m, *i.e.* $a_{im} = 1$. The routing variables p_{ijm} for user i are easily determined when the above assumption does not hold. Note that if $d_i^b \geq d_{im}^m$, user i will never use the uncached path. Also, if $d_i^b \leq d_{im}^h$, user i will never use cache m.

It is easy to see that with the congestion-insensitive model, given a content placement, the average minimum delay is obtained by routing requests for the cached content to caches, and routing the remaining requests to the uncached path. Note that under this routing policy no cache misses occur.

Note that $D(\mathbf{x}, \mathbf{p})$ is a linear function of the routing variables. Also note the additional constraint $p_{ijm} \leq x_{jm} \cdot a_{im}$, which is due to the fact that only requests for cached content are routed to caches. Since $d_{im}^h < d_i^b < d_{im}^m$, users have no incentive to split the traffic for any content between the cached and uncached paths, and hence there will be an optimal solution such that no routing variable has a fractional value, *i.e.* $p_{ijm} \in \{0, 1\}$.

2.4.1 Hardness of General Case

The above formulation of the joint caching and routing problem is a generalization of the Helper Decision Problem (HDP), stated below, proved to be NP-complete in [31].

Problem 1. *(Helper Decision Problem) Let's denote the connectivity graph in Figure 2.1 by bipartite graph* $G = (\mathcal{U}, \mathcal{H}, E)$ *where* $\mathcal{U}$ *and* $\mathcal{H}$ *denote the set of users and caches, respectively, and* E *denotes the set of edges between elements of* $\mathcal{U}$ *and* $\mathcal{H}$*. We ask the following question: given the graph* G *and a library of files* F *and a real number* $Q > 0$*, does there exist any way of placing elements of* F *in the nodes of* $\mathcal{H}$*, such that the total delay observed by the users is greater than or equal to* Q*?*

Our formulation is more general as we consider non-homogeneous delays for the cached and uncached paths. Therefore, we have the following result.

Theorem 1. *The optimal joint caching and routing problem with congestion-insensitive uncached paths is NP-complete.*

Proof. HDP reduces to the optimization problem in (2.1) by setting $d_b(\cdot) = d_c(\cdot) = 0$, $d_i^b = 1$, $d_{im}^h = 0$, and $C_m = C$, where C is the cache size at all caches in HDP. Hence, joint caching and routing problem is NP-hard. Moreover, for any given routing and caching, average delay can be computed in polynomial time. Therefore, the joint caching and routing problem in case of congestion-insensitive uncached paths is NP-complete. □

Although the problem is NP-complete in general, we will show that the joint caching and routing problem can be solved in polynomial time for several special cases. We will also identify what makes the problem "hard" in general. We first consider a restrictive setting where each user is interested in only one file and each file is requested by only one user. Next, we consider a network with two caches (but each user may be interested in an arbitrary number of files). We present polynomial

time solutions for both cases. Finally, we present an example that demonstrates what we conjecture to be *the* source of the complexity of this problem.

2.4.2 Special Case: One File per User

Consider the network illustrated in Figure 2.1, but assume each user is interested in only one file, *i.e.* $q_{ii} = 1$, and $q_{ij} = 0$ for $i \neq j$. In this case, the optimal solution to the joint caching and routing problem can be found in polynomial time based on a solution to the *maximum weighted matching* problem. A similar reduction of a caching problem to the maximum weighted matching problem was also previously presented in [71].

Note that in this case, the number of files equals the number of users, *i.e.* $N = K$. To avoid triviality, we assume that the number of users is larger than the capacity of each cache in the network, *i.e.* $C_m < N, \forall m$.

Theorem 2. *The solution to the joint caching and routing problem with congestion-insensitive uncached paths in the case that each user is interested in one file can be computed in polynomial time.*

Proof. The assumption that each user is interested in only one file allows us to rewrite the objective function in (2.1) as

$$D(\mathbf{x}, \mathbf{p}) = \frac{1}{\lambda}\Big(\sum_{i=1}^{N} \lambda_i d_i^b - \sum_i \sum_m \lambda_i p_{iim} (d_i^b - d_{im}^h) \Big).$$

Since $\sum_{i=1}^{N} \lambda_i d_i^b$ is a constant independent of the decision variables, minimizing the above objective function is equivalent to maximizing $\sum_i \sum_m \lambda_i p_{iim} (d_i^b - d_{im}^h)$. Note

that $\lambda_i(d_i^b - d_{im}^h)$ can be interpreted as the gain obtained by having file i in cache m. This problem can then be naturally seen as matching files to caches with the goal of maximizing the sum of individual gains. In what follows, we map this problem to the maximum weighted matching problem.

For each cache of size C_m, we introduce C_m nodes $\{v_m^1, v_m^2, \ldots, v_m^{C_m}\}$ representing unit size micro-caches that form cache m. Let $V = \{v_1^1, v_1^2, \ldots, v_1^{C_1}, \ldots, v_M^1, \ldots, v_M^{C_M}\}$ denote the set of all such nodes, and let $U = \{u_1, u_2, \ldots, u_N\}$ denote the set of all files. We define the bipartite graph $G(U, V, E)$ with $\lambda_i(d_i^b - d_{im}^h)$ as the weight of the edges connecting node u_i to nodes $v_m^s, \forall s \in \{1, 2, \ldots, C_m\}$. Figure 2.2 demonstrates a bipartite graph with user/file nodes u and the micro-cache nodes v with the edge weights shown for some of the edges. Note that the bipartite graph consists of $|U| + |V| = N + \sum_m C_m$ vertices and $|E| = O(N \sum_m C_m)$ edges.

The optimal solution to the joint content placement and routing problem corresponds to the maximum weighted matching for graph G. Edges selected in the maximum matching determine what content should be placed in which cache. Users then route to caches for cached content, and to the uncached path for the remaining files.

The maximum weighted matching problem for bipartite graphs can be solved in $O(|V|^2|E|)$ using the Hungarian algorithm [78]. In our context, the complexity is $O(M^3N^4)$. Note that $\sum_m C_m = O(MN)$ as we assume $C_m < N, \forall m$. Therefore, we can solve the joint caching and routing problem in polynomial time when users are interested in one file only. □

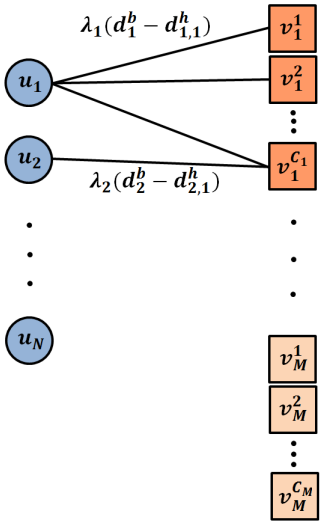


Figure 2.2: Modeling content placement as a maximum weighted matching problem. Each user is interested in only one file and each file is requested by only one user. Problem can be solved by matching users to cache spaces.

2.4.3 Special Case: Network with Two Caches

Next, we show that the optimal solution for the joint caching and routing problem can be found in polynomial time when there are only two caches in the network. Specifically, we prove that the solution to the integer program (2.1) can be found in polynomial time when there are two caches in the network.

By relaxing the integer constraints on content placement variables, x_{jm}, and allowing them to take real values, *i.e.* $0 \leq x_{jm} \leq 1$, we obtain a linear problem (LP) that is generally referred to as the "relaxed" problem. Since the objective function in (2.1) is convex, the solution to the *relaxed problem* can be found in polynomial time for all instances of the problem. Note that the set of constraints in the relaxed version of (2.1), namely, *i)* $\sum_m p_{ijm} \leq 1$, *ii)* $\sum_j x_{jm} \leq C_m$, *iii)* $-x_{jm} \leq 0,\ x_{jm} \leq 1$, and *iv)* $-p_{ijm} \leq 0,\ p_{ijm} - x_{jm} \cdot a_{im} \leq 0$ can be written in the linear form $\mathbf{Az} \leq \mathbf{b}$

where the entries of $\mathbf{A}$ and $\mathbf{b}$ are all integers, and $\mathbf{z}$ consists of the x_{jm} and p_{ijm} entries. We show that solving the relaxed program for a network with two caches produces integral solutions.

Before delving into the proof we introduce some definitions and results from [32]:

Definition 1. *A square integer matrix is called* unimodular *if it has determinant* $+1$ *or* -1.

Definition 2. *An* $m \times n$ *integral matrix* $\mathbf{A}$ *is* totally unimodular *if the determinant of every square submatrix is* 0, 1, *or* -1.

Proposition 1. *If for a linear program* $\{\max \mathbf{c}^T\mathbf{z} : \mathbf{Az} \leq \mathbf{b}\}$, $\mathbf{A}$ *is totally unimodular and* $\mathbf{b}$ *is integral, then all vertex solutions of the linear program are integral.*

From Proposition 1, then, it suffices to show that the matrix $\mathbf{A}$ is totally unimodular for a network with two caches to prove that the optimization problem can be solved in polynomial time. To prove that the matrix $\mathbf{A}$ is totally unimodular we use the following result from [62]:

Proposition 2. *A matrix is totally unimodular if and only if for every subset* R *of rows, there is an assignment* $s : R \to \pm 1$ *of signs to rows so that the signed sum* $\sum_{r \in R} s(r)r$ *(which is a row vector of the same width as the matrix) has all its entries in* $\{0, \pm 1\}$.

Theorem 3. *For a network with two caches, the LP relaxation of* (2.1) *with* $d_b(\cdot) = d_c(\cdot) = 0$ *produces an integral solution in polynomial time.*

Proof. In Appendix A, we give a constructive proof showing that for any subset R of rows of $\mathbf{A}$ we can find an assignment s that satisfies Proposition 2. □

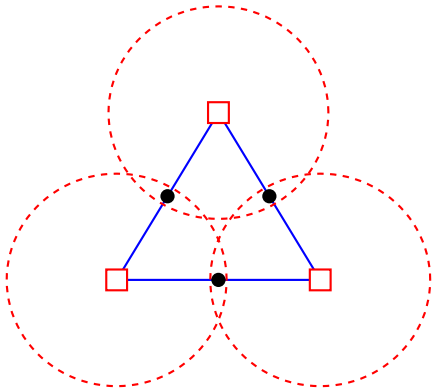


Figure 2.3: A network with three users (solid circles) and three caches (squares). Each user is in the communication range of two of the caches.

2.4.4 Complexity Discussion

Consider a network with three users and three caches as depicted in Figure 2.3. With each user connected to two of the caches, the user-cache connections can be seen to form a cycle as demonstrated in Figure 2.4a. Assume all paths from users to caches have equal hit and miss delays. Also, assume that each cache has the capacity of storing one file, and that all three users are interested in two files, noted here as green and red.

For the above network, the optimal content placement is to replicate one of the files in two of the caches, and have one copy of the other file in the third cache, as shown in Figure 2.4b. The solution to the *relaxed* optimization problem however

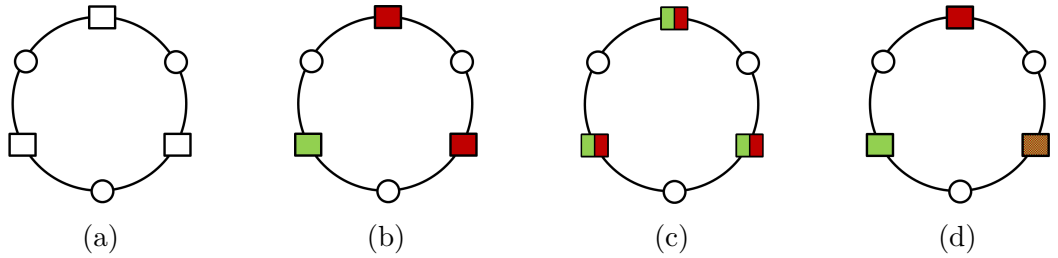


Figure 2.4: (a) A network of three users (circles) connected to three caches (squares) forming a cycle. Users are equally interested in two files, red and green. (b) Optimal content placement according to binary placement decisions, *i.e.* $x_{jm} \in \{0, 1\}$. (c) Optimal content placement assuming fractions of files can be stored in caches, *i.e.* $0 \leq x_{jm} \leq 1$. (d) Optimal content placement with the possibility of content coding. A copy of the two files is stored in two of the caches, and the third cache keeps a coded copy, *e.g.* XOR of the two files.

stores half of each file in each cache, *i.e.* $x_{1m} = x_{2m} = 0.5$, which achieves strictly smaller average delay. This solution is illustrated in Figure 2.4c[1].

The above discussion shows how the solution to the MILP optimization problem differs from its relaxed counterpart for the network shown in Figure 2.3. Such mismatch between the two solutions is also observed for larger networks that contain odd number of users and odd number of caches connected in a way that form a cycle. We conjecture that these cycles are *the* source of complexity in the problem of joint caching and routing, and for networks that do not have any such cycles the solu-

[1]Note that we do not consider the solution of the relaxed problem as a legitimate content placement. Although it looks like all users can access the two files via the caches in Figure 2.4c, when splitting the files in halves, two of the caches will store the same half copy of a file, and the user connected to those caches will only get half of that file from the caches and still needs to use the uncached path for the other half. However, we acknowledge that with the possibility of coding, content placement can be done in such a way that users can get both files from caches, as is shown in Figure 2.4d. We are not considering coded content placement in this work.

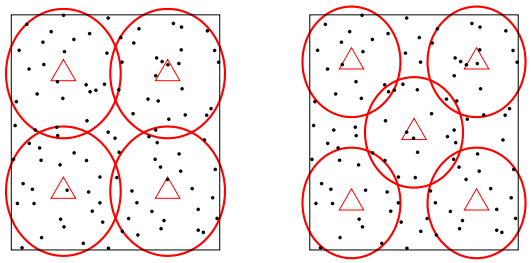


Figure 2.5: Examples of network topologies conforming to conjecture criteria.

tion to the optimization problem (2.1) matches that of the relaxed problem. More specifically we have:

Conjecture 1. *The optimal solution to the problem of joint caching and routing can be found in polynomial time if there are no cycles of length* $4k + 2$ *for any* $k \geq 1$ *in the bipartite graph corresponding to the user-cache connections.*

We have performed numerical simulations with thousands of randomly generated sample problems similar to the ones shown in Figure 2.5, with networks of four and five caches and up to 100 users in the network. We have then solved the LP version of MILP (2.1) to compute the optimal caching and routing. For all these sample problems, we have observed that the optimal solutions are integral. Although not a proof, these results support our conjecture.

2.5 Congestion-Sensitive Uncached Path

Next, we consider the case where delays for requests over the uncached paths and requests missed from caches depend on the load placed by such requests. Namely, we

compute the average delay over the uncached paths and the paths from the caches to the back-end server using convex functions $d_b(\cdot)$ and $d_c(\cdot)$, respectively. The reason we treat requests over the uncached path and missed requests from caches separately is that these paths could use different infrastructures to reach the back-end server. For example, requests from mobile users over the uncached path could use the LTE infrastructure, while missed requests from caches deployed on WiFi access points could use a wireline broadband connection.

2.5.1 Hardness of General Case

Note that we can consider the congestion-insensitive delay model as a special case of the congestion-sensitive model where $d_b(\cdot) = d_c(\cdot) = 0$. Thus, this problem is NP-complete in general. In the remainder of this section, however, we will prove that the problem of joint caching and routing in the case of a congestion-sensitive delay model remains NP-complete even if there is only one cache in the network and each content is of interest to no more than one user.

2.5.2 Hardness of Single-Cache Case

Here, we consider a special case of the problem where delays for requests sent over the uncached paths are modeled as an M/M/1 queue, *i.e.* $d_b(\lambda_b) = 1/(\mu_b - \lambda_b)$, where μ_b denotes the service rate. Also, requests generated by cache misses are assumed to observe a constant delay d_i^c, *i.e.* $d_c(\lambda_c) = 0$. Modifying the delay function $D(\mathbf{x}, \mathbf{p})$ in (2.1) for the case of one cache, *i.e.* $M = 1$, and assuming each user is interested in only one file, *i.e.* $q_{ii} = 1, \forall i$, and $q_{ij} = 0$ for $i \neq j$, we can rewrite the optimization problem as

$$
\begin{aligned}
\text{minimize} \quad & \frac{1}{\lambda}\left[\sum_{i=1}^{N}\lambda_i x_i p_i d_i^h + \sum_{i=1}^{N}\lambda_i(1-x_i)p_i d_i^c \right. \\
& \left. + \sum_{i=1}^{N}\lambda_i(1-p_i)d_i^b + \frac{\sum_{i=1}^{N}\lambda_i(1-p_i)}{\mu_b - \sum_{i=1}^{N}\lambda_i(1-p_i)}\right] \\
\text{such that} \quad & \sum_{i=1}^{N} x_i \leq C \qquad (2.2) \\
& 0 \leq p_i \leq a_i \\
& x_i \in \{0,1\},
\end{aligned}
$$

where $p_i = p_{ii1}$ denotes the fraction of user i requests routed to the cache. Also, a_i denotes whether user i is connected to the cache.

To show that the above optimization problem is NP-complete, we consider the corresponding decision problem, Congestion Sensitive Delay Decision Problem (CSDDP).

Problem 2. *(Congestion Sensitive Delay Decision Problem) Let* $\mathbf{\Lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_N]$ *denote user request rates, and let* $\mathbf{d}^h = [d_i^h]$, $\mathbf{d}^c = [d_i^c]$ *and* $\mathbf{d}^b = [d_i^b]$ *denote the hit delay, miss delay and initial access delay of the uncached path, respectively. Also, let* μ *be the service rate of the back-end server, and* C *be the cache capacity.*
We ask the following question: given the parameters $(\mu_b, \mathbf{\Lambda}, \mathbf{d}^h, \mathbf{d}^c, \mathbf{d}^b, C)$ *and a real number* d*, is there any assignment of* $\mathbf{x} = [x_i]$ *and* $\mathbf{p} = [p_i]$ *such that* $D(\mathbf{x}, \mathbf{p}) \leq d$*.*

It is clear that for any given content placement $\mathbf{x}$ and routing policy $\mathbf{p}$ the answer to CSDDP can be verified in polynomial time, and hence CSDDP is in class NP. To prove that CSDDP is NP-hard, we use the fact that the Equal Cardinality Partition (ECP) problem define below is NP-hard.

Problem 3. *(Equal Cardinality Partition) Given a set A of n numbers, can A be partitioned into two disjoint subsets A_1 and A_2 such that $A = A_1 \cup A_2$, the sum of the numbers in A_1 equals the sum of the numbers in A_2 and that $|A_1| = |A_2|$?*

Lemma 1. *ECP is NP-hard.*

Proof. See Appendix B. □

By reducing ECP to CSDDP, we have the following result:

Theorem 4. *CSDDP is NP-complete.*

Proof. See Appendix C for a detailed proof. □

Although this problem is NP-complete even in a very restrictive case with one cache and each user requesting one file, in the next section we show that a greedy algorithm can find approximate solutions with guaranteed performance.

Note that problem formulation (2.1) assumes a single queue shared by all caches to the back-end server. An alternative choice is to have distinct communication channels from each cache to the back-end server. In that case, $\lambda_c d_c(\lambda_c)$ in (2.1) should be replaced with a sum of M delay terms, where M is the number of caches. The model with distinct queues also results in an NP-complete problem, since problem (2.1) is NP-complete when there is only one cache in the network.

2.6 Approximation Algorithm

In this section, we show that the problem of joint caching and routing (for both congestion-insensitive and congestion-sensitive delay models) can be formulated as

the maximization of a monotone submodular function subject to matroid constraints. This enables us to devise algorithms with provable approximation guarantees.

We first review the definition and properties of matroids [53], and monotone [5] and submodular [62] functions, and then show our problem can be formulated as the maximization of a monotone submodular function subject to matroid constraints.

Definition 3. *A matroid M is a pair $M = (S, I)$, where S is a finite set and $I \subseteq 2^S$ is a family of subsets of S with the following properties:*

1. *$\emptyset \in I$,*
2. *I is downward closed,* i.e. *if $Y \in I$ and $X \subseteq Y$, then $X \in I$,*
3. *If $X, Y \in I$, and $|X| < |Y|$, then $\exists y \in Y \backslash X$ such that $X \cup \{y\} \in I$.*

Definition 4. *Let S be a finite set. A set function $f : 2^S \to \mathbb{R}$ is submodular if for every $X, Y \subseteq S$ with $X \subseteq Y$ and every $x \in S \backslash Y$ we have*

$$f(X \cup \{x\}) - f(X) \geq f(Y \cup \{x\}) - f(Y).$$

Definition 5. *A set function f is monotone increasing if $X \subseteq Y$ implies that $f(X) \leq f(Y)$.*

Let X_m denote the set of files stored in cache m, and define $X = X_1 \sqcup X_2 \sqcup \ldots \sqcup X_M$ to be the set of files stored in the M caches, where $\sqcup$ denotes disjoint union. X is the set equivalent of the binary content placement $\mathbf{x}$ defined in (2.1). Note that $|X_m| \leq C_m$

Let $S_m = \{s_{1m}, s_{2m}, \ldots, s_{Km}\}$ denote the set of all possible files that could be placed in cache m where s_{jm} denotes the storage of file j in cache m. The set element s_{jm} corresponds to the binary variable x_{jm} defined in the optimization problem (2.1) such that $x_{jm} = 1$ if and only if the element $s_{jm} \in X$. Define the super set $S = S_1 \cup S_2 \cup \ldots \cup S_M$ as the set of all possible content placements in the M caches. We have the following lemma.

Lemma 2. *The constraints in* (2.1) *form a matroid on* S.

Proof. For a given content placement $\mathbf{x}$, the optimal routing policy can be computed in polynomial time since $D_{\mathbf{x}}(\mathbf{p}) = D(\mathbf{p}; \mathbf{x})$ is a convex function. With that in mind, we can write average delay as a function of the content placement $X \subseteq S$. Thus, the constraints on the capacities of the caches can be expressed as $X \subseteq \mathcal{I}$ where

$$\mathcal{I} = \{X \subseteq S : |X \cap S_m| \leq C_m, \forall m = 1, \ldots, M\}.$$

Note that $(S, \mathcal{I})$ defines a matroid. □

Let $d_{ij}(\mathbf{x})$ denote the minimum average delay for user i accessing file j through a cached path, given content placement $\mathbf{x}$, excluding queueing delay for fetching content from the back-end server in the case of a cache miss. We have

$$d_{ij}(\mathbf{x}) = \min_m d_{ijm},$$

where d_{ijm} denotes the average delay of accessing content j from cache m, excluding the queueing delay, defined as (x_{jm} indicates that file j is in cache m)

$$d_{ijm} = d^h_{im} x_{jm} + d^m_{im}(1 - x_{jm}).$$

Similarly, we define

$$y_{ij} = \max_m a_{im} x_{jm},$$

denoting whether user i can access content j from a neighboring cache.

Given the content placement in the caches, let $p_{ij} \triangleq \sum_m p_{ijm}$ denote the fraction of the traffic for which user i uses the cached paths to access content j. Also, let

$$\lambda_c(\mathbf{p}) = \sum_{i,j} \lambda_i q_{ij}(1 - y_{ij}) p_{ij},$$

and

$$\lambda_b(\mathbf{p}) = \sum_{i,j} \lambda_i q_{i,j}(1 - p_{ij}),$$

denote the aggregate request rate for missed requests, and requests sent over the uncached paths, respectively. We rewrite the delay functions for the congestion-insensitive and the congestion-sensitive models as

$$D(\mathbf{p}; \mathbf{x}) = \frac{1}{\lambda} \left(\sum_{i,j} \lambda_i q_{ij} p_{ij} d_{ij}(\mathbf{x}) + \sum_{i,j} \lambda_i q_{ij}(1 - p_{ij}) d^b_i \right),$$

and

$$D(\mathbf{p}; \mathbf{x}) = \frac{1}{\lambda} \left[\sum_{i,j} \lambda_i q_{ij} \Big(p_{ij} d_{ij}(\mathbf{x}) + (1 - p_{ij}) d^b_i \Big) + \lambda_b(\mathbf{p}) d_b\Big(\lambda_b(\mathbf{p})\Big) + \lambda_c(\mathbf{p}) d_c\Big(\lambda_c(\mathbf{p})\Big) \right],$$

respectively. The optimal routing policy for a given content placement $\mathbf{x}$, then, is one that minimizes $D(\mathbf{p};\mathbf{x})$, and can be found by solving the following optimization problem:

$$\begin{aligned} &\text{minimize} \quad D(\mathbf{p};\mathbf{x}) \\ &\text{such that} \quad 0 \leq p_{ij} \leq 1 \quad \forall i,j. \end{aligned}$$

Note that $D(\mathbf{p};\mathbf{x})$ is convex and the above optimization problem can be solved in polynomial time.

Let $\mathbf{x}_X$ be the equivalent binary representation of the content placement set X. It is clear that adding items to the set X can only decrease the value of $D(\mathbf{p};\mathbf{x}_X)$. Moreover, one might expect that adding an item to a set containing a smaller number of files might decrease the delay by a larger amount compared to adding the item to a set containing a larger number of files. We formally prove this statement through the following lemma for both congestion-insensitive and congestion-sensitive delay models bearing in mind that $D_\emptyset$ denotes the expected delay with no files cached:

Lemma 3. *Let $\mathcal{P}$ denote all routing policies. For $X \subseteq S$, the function $G(X) = D_\emptyset - \min_{\mathbf{p}\in\mathcal{P}} D(\mathbf{p};\mathbf{x}_X)$ is a monotone increasing and submodular function.*

Proof. See Appendix D for a detailed proof. □

A direct consequence of Lemma 3 is that the objective of the joint caching and routing problem is to maximize a monotone submodular function. Therefore,

Theorem 5. *The approximate solution obtained by the greedy algorithm in Algorithm 1 is within a $(1-1/e)$ factor of the optimal solution $G(X^*)$.*

Proof. It was shown in [9] that the greedy algorithm for maximizing a monotone submodular set function with matriod constraints yields a $(1 - 1/e)$-approximation. □

Algorithm 1 starts with empty caches and at each step greedily adds a file to the cache that maximizes function G. This process continues until all caches are filled to capacity. Optimal routing is then determined based on the content placement.

Algorithm 1 GreedyWG: A greedy approximation with performance guarantees.

$S \leftarrow \{s_{jm} : 1 \leq j \leq K, 1 \leq m \leq M\}$
$X_m \leftarrow \emptyset, \forall m$
$X \leftarrow \emptyset$
for $c \leftarrow 1$ **to** $\sum_m C_m$ **do**
    $s_{j^*m^*} \leftarrow \arg\max_{s_{jm} \in S} G(X \cup \{s_{jm}\})$
    $X_{m^*} \leftarrow X_{m^*} \cup \{s_{j^*m^*}\}$
    $X \leftarrow X \cup \{s_{j^*m^*}\}$
    if $|X_{m^*}| = C_{m^*}$ **then**
        $S \leftarrow S \backslash s_{jm^*}, \forall j$
    else
        $S \leftarrow S \backslash s_{j^*m^*}$
Content placement is done according to X.
Determine the routing as $\mathbf{p}^* \leftarrow \arg\min_{\mathbf{p}} D(\mathbf{p}; \mathbf{x}_X)$.

Although the greedy algorithm in Algorithm 1 is guaranteed to find solutions within a $(1 - 1/e)$ factor of the optimal solution, its complexity is high, $O(M^2N^2K^2 \log(NK))$. We devise a second, computationally more efficient, greedy algorithm in Algorithm 2 with time complexity $O(M^3NK)$. We do not have accuracy guarantees for Algorithm 2, but in the next section, we will show that it performs very well in practice.

Algorithm 2 is based on the following ideas. It starts with all caches empty and initializes cache access delays for users as the miss delays to their closest caches. Then at each step a file is greedily selected to be placed in a cache that maximizes the change in the user access delays, $\sum_i \lambda_i q_{ij}(d_{ij} - \min\{d_{ij}, d^h_{im}\})$. This process continues until the caches are filled. Finally, similar to Algorithm 1, a routing policy that minimizes $D(\mathbf{p}; \mathbf{x})$ is determined.

Algorithm 2 Greedy: A greedy approximation without known performance guarantees.

$X_m \leftarrow \emptyset, \forall m$
$X \leftarrow \emptyset$
$d_{ij} \leftarrow \min_m\{d^m_{im}\}, \forall i, j$
for $c \leftarrow 1$ **to** $\sum_m C_m$ **do**
    $G_{jm} \leftarrow [0]_{K \times M}$
    for $m \leftarrow 1$ **to** M **do**
        if $|X_m| < C_m$ **then**
            for $j \leftarrow 1$ **to** K **do**
                $G_{jm} \leftarrow \sum_i \lambda_i q_{ij}(d_{ij} - \min\{d_{ij}, d^h_{im}\})$
    $[j^*, m^*] \leftarrow \arg\max_{j,m} G_{jm}$
    $X_{m^*} \leftarrow X_{m^*} \cup \{s_{j^* m^*}\}$
    $X \leftarrow X \cup \{s_{j^* m^*}\}$
    $d_{ij^*} \leftarrow \min\{d_{ij^*}, d^h_{im^*}\}, \forall i$
Content placement is done according to X.
Determine the routing as $\mathbf{p}^* \leftarrow \arg\min_{\mathbf{p}} D(\mathbf{p}; \mathbf{x}_X)$

2.7 Performance Evaluation

In this section, we evaluate the performance of the approximate algorithms through discrete-event simulations. Our goal here is to evaluate 1) how well the solutions for the greedy algorithms compare to the optimal solutions (when com-

puting the optimal solution is feasible), and 2) how well solutions from the greedy algorithms compare to those produced by a baseline.

Here, we consider a congestion-sensitive model where the requests over the uncached paths experience a queuing delay modeled as an M/M/1 queue[2], while requests generated by cache misses experience a constant delay. For our baseline, we compare the approximate algorithms to the following algorithm we refer to as *p-LRU*.

2.7.1 p-LRU

The cache replacement policy at all caches is Least Recently Used (LRU). For the routing policy, we assume that users not connected to any caches forward all their requests to the back-end servers. The remaining users forwards each request to a cache with probability p and with probability $1 - p$ forward the request to the uncached path. If user i decides to use a cached path, she chooses uniformly at random one of the n_i caches she is connected to. The value of p is the same for all users that have access to a cache, and is chosen to minimize the average delay.

First, assuming users equally split their traffic across the caches that they can access, the aggregate popularity for individual files is computed at each cache. Let r_j^m denote the normalized aggregate popularity of file j at cache m. We have

$$r_j^m = \frac{1}{\Lambda} \sum_{i \in \mathcal{I}_m} \lambda_i q_{ij} / n_i,$$

[2]Note that our analysis is valid for a G/G/1 queue, and the M/M/1 queue is only assumed for evaluation purposes since there is no closed form formula for a G/G/1 queue.

where $\mathcal{I}_m$ denotes the set of users connected to cache m, and Λ is the normalizing constant across all files. Note that r_j^m is independent of the parameter p. With the aggregate popularities at hand, hit probabilities are computed at each cache using the *characteristic time* approximation [12]. Let $\mathbb{P}(x_{jm} = 1)$ denote the probability that file j resides in cache m. From [12] we have

$$\mathbb{P}(x_{jm} = 1) = 1 - \exp\left(-r_j^m T_m\right),$$

where T_m is the characteristic time of cache m is the unique solution to the equation

$$C_m = \sum_j 1 - \exp\left(-r_j^m T_m\right).$$

Given the cache hit probabilities, the average delay in accessing content j from caches for user i equals

$$d_{ij}^c = \frac{1}{n_i} \sum_{m \in \mathcal{M}_i} \left[\mathbb{P}(x_{jm} = 1) d_{im}^h + (1 - \mathbb{P}(x_{jm} = 1)) d_{im}^m\right],$$

where $\mathcal{M}_i$ denotes the set of caches that user i is connected to. Note that $|\mathcal{M}_i| = n_i$.

Let $\mathcal{I}$ denote the set of users that are connected to at least one cache, and let $\lambda_{\mathcal{I}}$ denote the aggregate request rate of these users. The average delay to access content from caches equals

$$D_c = \frac{1}{\lambda_{\mathcal{I}}} \sum_{i \in \mathcal{I}} \sum_j \lambda_i q_{ij} d_{ij}^c.$$

Remember that some users may not be connected to any caches. Considering the traffic from all users, we can write the overall average delay as

$$D_{\text{LRU}} = \frac{1}{\lambda}\left[p\lambda_{\mathcal{I}} D_c + (1-p)\sum_{i\in\mathcal{I}} \lambda_i d_i^b + \sum_{i\notin\mathcal{I}} \lambda_i d_i^b + \frac{\mu}{\mu - (1-p)\sum_{i\in\mathcal{I}} \lambda_i - \sum_{i\notin\mathcal{I}} \lambda_i} - 1\right].$$

By differentiating D_{LRU} with respect to p, the optimal value of p is found to be

$$p^* = \max\{0, \min\{1, \left(\sqrt{\frac{\mu\sum_{i\in\mathcal{I}} \lambda_i}{\lambda_{\mathcal{I}} D_c - \sum_{i\in\mathcal{I}} \lambda_i d_i^b}} - \mu + \lambda\right)/\lambda_{\mathcal{I}}\}\}.$$

2.7.2 Network Setup

We consider a network with users uniformly distributed in a 2-D square. We consider two architectures. First, we assume there is only one large cache at the center of the network as in Figure 2.6a. Second, we consider a network with five small caches with equal storage capacities as in Figure 2.6b. Figure 2.6 also shows the communication range of the caches in each case. In the single-cache network, the cache has a larger communication range and five times the capacity of each of the caches in the multi-cache network.

Users that are not in communication range of any caches can only use the uncached path to the back-end server. The hit delay for each user is linearly proportional to the distance from the cache and has the maximum value[3] of 12.5 time units and 5.5 time units for the single and multi-cache systems, respectively. For a cache

[3]Here, delay aggregates all request propagation and download delays as well as the processing and queuing delays. We use normalized delay values instead of using any specific time unit.

miss, an additional delay of 25 time units is added to the hit delay. The initial access delay of the uncached path is set to five time units for each user, and the service rate is proportional to the aggregate request rate, where the scaling factor will be specified later.

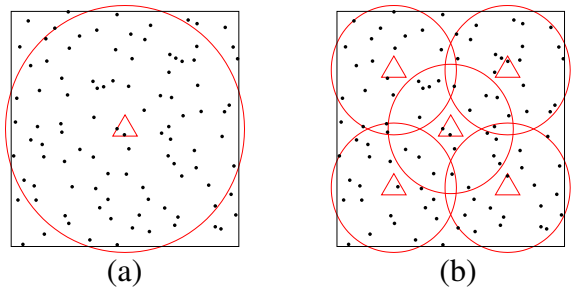


Figure 2.6: A network with (a) one cache, and (b) five caches.

2.7.3 Numerical Evaluation

2.7.3.1 GreedyWG vs. Optimal

First, we compare the solution of GreedyWG the approximate algorithm in Algorithm 1 to the optimal solution. Due to the exponential complexity of finding the optimal solution, we are only able to compute the optimal solution for small problem instances. Here, we consider a network with five users and a single cache. User request rates are arbitrarily set to satisfy $\sum_i \lambda_i = 5$. We assume users are interested in 15 files, and that the aggregate user request popularities follow a Zipf distribution with skewness parameter 0.6. The service rate of the back-end server is $\mu = 1$.

Figure 2.7 shows the average delay and the 95% confidence interval over 100 runs of each algorithm. It is clear that GreedyWG is very close to optimal. In fact, we

observe that GreedyWG differs from the optimal solution less than 20% of the time, and the relative difference is never more than 1%.

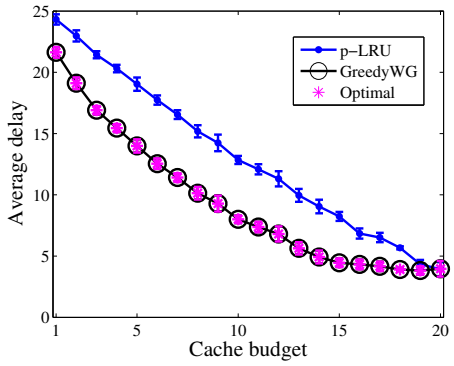


Figure 2.7: Evaluation of GreedyWG against Optimal and p-LRU.

2.7.3.2 GreedyWG vs. Greedy

Next, we compare the solutions of GreedyWG against those of Greedy, the approximation algorithm, Algorithm 2, with lower computational complexity but no performance guarantees. We consider a network with five caches and 100 users uniformly distributed in a 10×10 field.

Figure 2.8 shows the average delay and the 95% confidence interval for different values of available cache budget. Greedy (red curve) is barely distinguishable from GreedyWG (black curve), meaning that Greedy performs very close to GreedyWG.

We also evaluate these algorithms over different values of the service rate at the back-end server. Figure 2.9 shows the average delay for μ between 2 to 7, with the aggregate traffic rate set to $\lambda = 5$. Similar to Figure 2.8, Greedy performs very close to GreedyWG, and is always within 1% of GreedyWG.

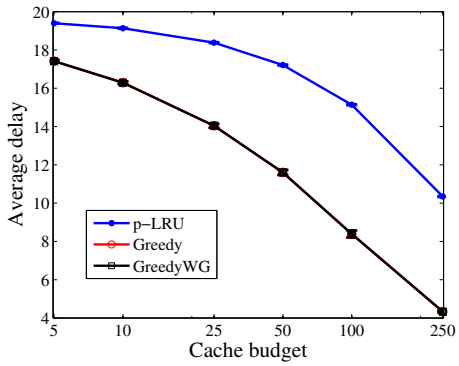


Figure 2.8: Evaluation of the two greedy approximations over different values of the cache budget split equally between five caches. Aggregate user request rate is $\lambda = 5$, and service rate of the back-end server equals 2.5.

2.7.4 Trace-driven Simulation

Here, we present trace-driven evaluation results where we use traces for web accesses collected from a gateway router at IBM research lab [82]. The trace consists of approximately 9 million requests generated for more around 3.3 million distinct files over a period of five hours. We only consider Greedy, the greedy algorithm presented in Algorithm 2, since it performs nearly as well as Algorithm 1, and has lower complexity.

The access delay to each cache equals one-tenth of the distance from the cache in case of a cache hit. For a cache miss, an additional delay of 25ms is added to the hit delay. The initial access delay of the uncached path is set to 5ms for each user, and the service rate is proportional to the aggregate request rate, where the scaling factor will be specified later.

To evaluate the Greedy algorithm using the trace data, we first divide the trace into smaller segments of approximately 120,000 requests. Each segment includes

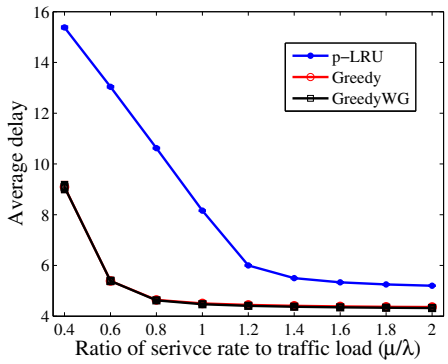


Figure 2.9: Evaluation of the greedy algorithms for different values of the service rate at the back-end server. Aggregate user request rate is $\lambda = 5$, and the service rate varies from 2 to 10. Cache budget is set to 125.

requests for approximately 40,000 distinct files, generated by approximately 2500 users. To simulate requests from the ith segment, we first compute the file popularities using the $(i-1)$st segment, and compute the optimal value of p for the p-LRU algorithm.

Figure 2.10 compares the average delays for different cache budgets for the p-LRU and the Greedy algorithms for the single-cache (S) and multi-cache (M) networks. Reductions in average delay of up to 50% are observed for both single-cache and multi-cache networks when using Greedy over p-LRU. While p-LRU yields similar performance in both single-cache and multi-cache architectures, Greedy shows the advantage of one architecture over the other depending on the cache budget. When the cache budget is small, it is better to have a single cache with larger cache size and coverage so that more users can access popular files from the cache; when the

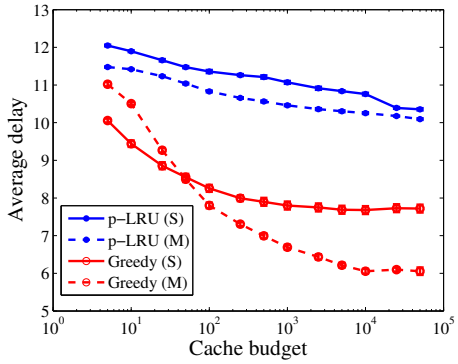


Figure 2.10: Evaluation of the Greedy and p-LRU for the single-cache (S) and multi-cache (M) network setups for different values of the available cache budget. The service rate is set to be 0.8 times the aggregate traffic rate.

cache budget is large, it is better to have multiple caches, each with smaller size and coverage, so that users can access files from nearby caches with smaller hit delays.

We also evaluate the algorithms for different values of the service rate of the uncached path assuming the cache budget is fixed at $10,000$. Figure 2.11 shows the average delay when the ratio of service rate to the total request rate changes from 0.6 to 1.2. Similar to Figure 2.10, the Greedy algorithm significantly reduces the average content access delay. Again, the cache architecture makes little difference for p-LRU but significant affect to the performance of the Greedy algorithm. Moreover, the difference decreases as the service rate on the uncached path increases, as more traffic is offloaded to the uncached path.

2.8 Related Work

Benefits of content caching have been theoretically analyzed [3,30,60,65,66]. [3,30] demonstrate that the asymptotic throughput capacity of a network is significantly

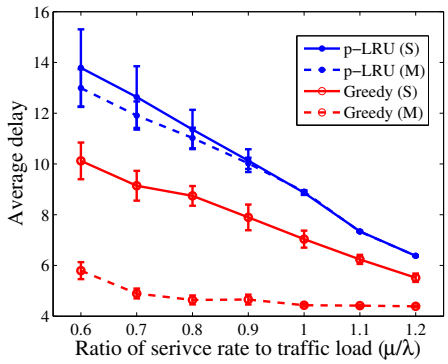


Figure 2.11: Evaluation of the Greedy and p-LRU algorithms for different values of the service rate to aggregate traffic ratio for the single-cache (S) and multi-cache (M) network setups.

increased by adding caching capabilities to the nodes. In this chapter, we have considered the *joint* routing and cache-content management problems. Numerous past research efforts have considered these problems separately. The problem of content placement in caches, has received significant attention in the Internet, in hybrid networks such as those considered in this chapter, and in sensor networks [4, 8, 57, 63, 72]. Baev *et al.* [4] prove that the problem of content placement with the objective of minimizing the access delay is NP-complete, and present approximate algorithms. More recently, Giovanidis *et al.* [29] introduced multi-LRU, a family of decentralized caching policies, that extends the classical LRU policy to cases where objects can be retrieved from more than one cache. The separate problem of efficient routing in cache networks has also been explored in the literature [11, 58]. Cache-aware routing schemes that calculate paths with minimum transportation costs based on given caching policy and request demand have been proposed in [65].

The joint caching and routing problem, with the objective of minimizing content access delay, has recently been studied in [57, 63], where the authors consider a hybrid network consisting of multiple femtocell caches and a cellular infrastructure. Both papers assume that users greedily choose the minimum delay path to access content, *i.e.* requests for cached content are routed to caches (where content is know to reside), whereas remaining requests are routed to the (uncached) cellular network. They assume that the delays are constant and independent of the request rate.

Our work differs from much of the previous research discussed above by considering a joint caching and routing problem, where we determine the optimal routes users should take for accessing content as well as the optimal caching policy. Our research differs from [57, 63] in that we consider heterogeneous delays between users and caches, consider a congestion-insensitive delay model for the uncached path as well as a congestion-sensitive model, investigate the problem's time complexity, and propose bounded approximate solutions for both congestion-insensitive and congestion-sensitive scenarios. We also determine scenarios for which the optimal solution can be found in polynomial time for the congestion-insensitive delay model, and ascertain the root cause of the complexity of the general problem.

Algorithms for joint caching and routing schemes were previously proposed in [37] and [15] based on the primal-dual method. These algorithms are based on the Lagrangian relaxation method and rely on iterative algorithms to reach a solution with certain optimality criteria. As such, there is no efficiency guarantee on the results nor the running time of these algorithms. In contrast, our proposed approximation

algorithms require fixed running time, and are guaranteed to be withing a factor $1 - 1/e$ of the optimal solution.

2.9 Conclusion

In this chapter, we considered the problem of joint content placement and routing in heterogeneous networks that support in-network caching but also provide a separate (uncached) path to a back-end content server; we considered cases in which paths to the back-end server were modeled as congestion-insensitive, constant-delay paths, and congestion-sensitive paths modeled by a convex delay rate function. We provided fundamental complexity results showing that the problem of joint caching and routing is NP-complete in both cases, developed a greedy algorithm with guaranteed performance of $(1-1/e)$ of the optimal solution as well as a lower complexity heuristic that was empirically found to provide average delay performance that was within 1% of optimal (for small instances of the problem) and that significantly reduce the average content access delay over the case of optimized traditional LRU caching. Our investigation of special-case scenarios − the congestion-insensitive multiple-cache single-file-of-interest case (where we demonstrated an optimal polynomial time solution) and the congestion-sensitive single-cache single-file-of-interest case (which we demonstrated remained NP-complete) − helped illuminate what makes the problem "hard" in general. Our future work is aimed at developing distributed algorithms for content placement and routing, and on developing solutions for the case of time-varying content popularity.

CHAPTER 3

A UTILITY OPTIMIZATION APPROACH TO NETWORK CACHE DESIGN

3.1 Introduction

In a caching system, admission and eviction policies determine which contents are added and removed from a cache when a miss occurs. Usually, these policies are devised so as to mitigate staleness and increase hit probability. Nonetheless, the utility of having a high hit probability can vary across contents. This occurs, for instance, when service level agreements must be met, or if certain contents are more difficult to obtain than others. Traditional cache management policies such as LRU treat different contents in a strongly coupled manner that makes it difficult for (cache) service providers to implement differentiated services, and for content publishers to account for the valuation of their content delivered through content distribution networks. In this chapter, we propose a utility-driven caching framework, where each content has an associated utility and content is stored and managed in a cache so as to maximize the aggregate utility for all content. Utilities can be chosen to trade off user satisfaction and cost of storing the content in the cache. We draw on analytical results for time-to-live (TTL) caches [26], to design caches with ties to utilities for individual (or classes of) contents. Utility functions also have implicit

notions of fairness that dictate the time each content stays in cache. Our framework can be used to develop *online* algorithms for cache management. Our framework has implications for distributed pricing and control mechanisms and hence is well-suited for designing cache market economic models.

Our main contribution in this chapter is the formulation of the utility-based optimization framework for maximizing aggregate content publisher utility subject to buffer capacity constraints at the service provider. We also develop online algorithms for managing cache content, and perform simulations to show their efficiency using different utility functions with different notions of fairness.

3.2 Model

Consider a cache of size B that serves a set of N files. We assume that requests for file i are described by a Poisson process with rate λ_i. Furthermore, file i has size s_i. Let h_i denote the hit probability for content i. Associated with each content, $i = 1, \ldots, N$, is a utility function $U_i : [0, 1] \to \mathbb{R}$ that represents the "satisfaction" perceived by observing hit probability h_i. $U_i(\cdot)$ is assumed to be increasing, continuously differentiable, and strictly concave. We further assume that $U_i'(0) = \infty$. This ensures that $h_i > 0, \forall i$. Note that a function with these properties is invertible. We will treat utility functions that do not satisfy these constraints as special cases.

Our analysis in this chapter is based on Time-To-Live (TTL) caches explained in Appendix E. The hit probability of file i for these two classes of non-reset and reset TTL caches can be expressed as

$$h_i = 1 - \frac{1}{1 + \lambda_i t_i}, \tag{3.1}$$

and

$$h_i = 1 - e^{-\lambda_i t_i}, \tag{3.2}$$

respectively, where requests for file i arrive at the cache according to a Poisson process with rate λ_i.

3.3 Cache Utility Maximization

In this section, we formulate cache management as a utility maximization problem. We introduce two formulations, one where the buffer size introduces a hard constraint and a second where it introduces a soft constraint.

3.3.1 Hard Constraint Formulation

We are interested in designing a cache management policy that optimizes the sum of utilities over all files, more precisely,

$$\begin{aligned} \underset{h_i}{\text{maximize}} \quad & \sum_{i=1}^{N} U_i(h_i) \\ \text{such that} \quad & \sum_{i=1}^{N} s_i h_i = B \\ & 0 \leq h_i \leq 1, \quad i = 1, 2, \ldots, N. \end{aligned} \tag{3.3}$$

Note that the feasible solution set is convex and since the objective function is strictly concave and continuous, a unique maximizer, called the optimal solution, exists. Also

note that the buffer constraint is based on the *expected* number of files not exceeding the buffer size and not the total number of files. Towards the end of this section, we show that the buffer space can be managed in a way such that the probability of *violating* the buffer size constraint vanishes as the number of files and cache size grow large.

In the above formulation, utilities are defined as functions of *file* hit probability, *i.e.* $U_i(h_i)$. In Section 4.6, we argue that utilities can alternatively be defined as functions of *byte* hit probabilities, *i.e.* $U_i(s_i h_i)$.

The above formulation does not enforce any special technique for managing the cache content to achieve the desired h_is, and any strategy that can easily adjust the hit probabilities can be employed. We use the TTL cache as our building block because it provides the means through setting timers to control the hit probabilities of different files in order to maximize the sum of utilities.

Using timer based caching techniques for controlling the hit probabilities with $0 < t_i < \infty$ ensures that $0 < h_i < 1$, and hence, disregarding the possibility of $h_i = 0$ or $h_i = 1$, we can write the Lagrangian function as

$$\mathcal{L}(\mathbf{h}, \alpha) = \sum_{i=1}^{N} U_i(h_i) - \alpha \left[\sum_{i=1}^{N} s_i h_i - B \right] = \sum_{i=1}^{N} \Big[U_i(h_i) - \alpha s_i h_i \Big] + \alpha B,$$

where α is the Lagrange multiplier.

In order to achieve the maximum in $\mathcal{L}(\mathbf{h}, \alpha)$, the hit probabilities should satisfy

$$\frac{\partial \mathcal{L}}{\partial h_i} = U_i'(h_i) - \alpha s_i = 0. \tag{3.4}$$

Let $U_i'(\cdot)$ denote the derivative of the the utility function $U_i(\cdot)$, and define $U_i'^{-1}(\cdot)$ as its inverse function. From (3.4) we get

$$U_i'(h_i) = \alpha s_i,$$

or equivalently

$$h_i = U_i'^{-1}(\alpha s_i). \tag{3.5}$$

Applying the cache storage constraint we obtain

$$\sum_i s_i h_i = \sum_i s_i U_i'^{-1}(\alpha s_i) = B, \tag{3.6}$$

and α can be computed by solving the fixed-point equation given above.

As mentioned before, we can implement utility maximizing caches using TTL based policies. Using the expression for the hit probabilities of non-reset and reset TTL caches given in Appendix E, we can compute the timer parameters t_i, once α is determined from (3.6). For non-reset TTL caches we obtain

$$t_i = -\frac{1}{\lambda_i}\Big(1 - \frac{1}{1 - U_i'^{-1}(\alpha s_i)}\Big), \tag{3.7}$$

and for reset TTL caches we get

$$t_i = -\frac{1}{\lambda_i}\log\Big(1 - U_i'^{-1}(\alpha s_i)\Big). \tag{3.8}$$

3.3.2 Soft Constraint Formulation

The formulation in (3.3) assumes a hard constraint on cache capacity. In some circumstances it may be appropriate for the (cache) service provider to increase the available cache storage at some cost to the file provider for the additional resources[1]. In this case the cache capacity constraint can be replaced with a penalty function $C(\cdot)$ denoting the cost for the extra cache storage. Here, $C(\cdot)$ is assumed to be a convex and increasing function. We can now write the utility and cost driven caching formulation as

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} U_i(h_i) - C(\sum_{i=1}^{N} s_i h_i - B) \\ \text{such that} \quad & 0 \leq h_i \leq 1, \quad i = 1, 2, \ldots, N. \end{aligned} \tag{3.9}$$

Note the optimality condition for the above optimization problem states that

$$U_i'(h_i) = s_i C'(\sum_{i=1}^{N} s_i h_i - B).$$

Therefore, for the hit probabilities we obtain

$$h_i = U_i'^{-1}\Big(s_i C'(\sum_{i=1}^{N} s_i h_i - B)\Big).$$

[1]One straightforward way of thinking about this is to turn the cache memory disks on and off based on the demand.

Multiplying the two sides of the above equation by s_i and summing over all i, we can compute the optimal value for the cache storage, $B^* = \sum_i s_i h_i$, using the fixed-point equation

$$B^* = \sum_{i=1}^{N} s_i U_i'^{-1}\Big(s_i C'(B^* - B)\Big). \tag{3.10}$$

3.3.3 Buffer Constraint Violations

Before we leave this section, we address an issue that arises in both formulations, namely how to deal with the fact that there may be more contents with unexpired timers than can be stored in the buffer. This occurs in the formulation of (3.3) because the constraint is on the *average* buffer occupancy and in (3.9) because there is no constraint. Let us focus on the formulation in (3.3) first. Our approach is to provide a buffer of size $B(1+\epsilon)$ with $\epsilon > 0$, where a portion B is used to solve the optimization problem and the additional portion ϵB to handle buffer violations. We will see that as the number of contents, N, increases, we can get by growing B in a sublinear manner, and allow ϵ to shrink to zero, while ensuring that content will not be evicted from the cache before their timers expire with high probability. Let X_i denote whether content i is in the cache or not; $P(X_i = 1) = h_i$. Now, let $\mathbb{E}\big[\sum_{i=1}^{N} s_i X_i\big] = \sum_{i=1}^{N} s_i h_i = B$. We write $B(N)$ as a function of N, and assume that $B(N) = \omega(1)$.

Theorem 6. *Assume that $s_i \leq s_{\max}, \forall i$. For any $\epsilon > 0$,*

$$\mathbb{P}\Big(\sum_{i=1}^{N} s_i X_i \geq B(N)(1+\epsilon)\Big) \leq e^{-\frac{\epsilon^2 B(N)}{(2+\epsilon)s_{\max}}}.$$

The proof follows from the application of a Chernoff bound.

Theorem 6 states that we can size the buffer as $B(1+\epsilon)$ while using a portion B as the constraint in the optimization. The remaining portion, ϵB, is used to protect against buffer constraint violations. It suffices for our purpose that $\frac{\epsilon^2 B(N)}{s_{\max}} = \omega(1)$. This allows us to select $B(N) = o(N)$ while at the same time selecting $\epsilon = o(1)$. As an example, consider Zipf's law with $\lambda_i = \lambda/i^z$, $\lambda > 0$, $0 < z < 1$, $i = 1, \ldots, N$ under the assumption that $\max\{t_i\} = t$ for some $t < \infty$. In this case, we can grow the buffer as $B(N) = O(N^{1-z})$ while ϵ can shrink as $\epsilon = 1/N^{(1-z)/3}$. Analogous expressions can be derived for $z \geq 1$.

Similar choices can be made for the soft constraint formulation.

3.4 Utility Functions and Fairness

Using different utility functions in the optimization formulation (3.3) yields different timer values for the files. In this sense, each utility function defines a notion of fairness in allocating storage resources to different files. In this section, we study a number of utility functions that have important fairness properties associated with them.

Here, we consider the family of β-fair (also known as *isoelastic*) utility functions given by

$$U_i(h_i) = \begin{cases} w_i \frac{h_i^{1-\beta}}{1-\beta} & \beta \geq 0, \beta \neq 1; \\ \\ w_i \log h_i & \beta = 1, \end{cases}$$

where the coefficient $w_i \geq 0$ denotes the weight for file i. This family of utility functions unifies different notions of fairness in resource allocation [67]. In the remainder

of this section, we investigate some of the choices for β that lead to interesting special cases.

3.4.1 $\beta = 0$

With $\beta = 0$, we get $U_i(h_i) = w_i h_i$, and maximizing the sum of the utilities corresponds to

$$\max_{h_i} \sum_i w_i h_i.$$

The above utility function defined does not satisfy the requirements for a utility function mentioned in Section 3.2, as it is not strictly concave. However, it is easy to see that the sum of the utilities is maximized when

$$h_i = 1, i = 1, \ldots, m-1,$$

$$h_m = (B - \sum_{i=1}^{m-1} s_i)/s_m,$$

and

$$h_i = 0, i = m+1, \ldots, N,$$

where

$$m = \arg\min_n \{\sum_{i=1}^{n} s_i > B\}$$

and files are sorted such that $w_1/s_1 \geq \ldots \geq w_N/s_N$.

Note that the policy obtained by implementing this utility function with $w_i = \lambda_i$ maximizes the overall throughput.

3.4.2 $\beta = 1$

Letting $\beta = 1$, we get $U_i(h_i) = w_i \log h_i$, and hence maximizing the sum of the utilities corresponds to

$$\max_{h_i} \sum_i w_i \log h_i.$$

It is easy to see that $U_i'^{-1}(\alpha s_i) = \dfrac{w_i}{\alpha s_i}$, and hence using (3.6) we obtain

$$\sum_i s_i U_i'^{-1}(\alpha s_i) = \sum_i w_i/\alpha = B,$$

which yields

$$\alpha = \sum_i w_i/B.$$

The hit probability of file i then equals

$$h_i = U_i'^{-1}(\alpha s_i) = \frac{w_i}{s_i \sum_j w_j} B.$$

This utility function implements a *proportionally fair* policy [40]. With $w_i = \lambda_i$, the hit probability of file i is proportional to λ_i/s_i.

3.4.3 $\beta = 2$

With $\beta = 2$, we get $U_i(h_i) = -w_i/h_i$, and maximizing the total utility corresponds to

$$\max_{h_i} \sum_i -w_i/h_i.$$

In this case, we get $U_i'^{-1}(\alpha s_i) = \sqrt{w_i}/\sqrt{\alpha s_i}$, therefore

$$\sum_i s_i U_i'^{-1}(\alpha s_i) = \sum_i \sqrt{w_i s_i}/\sqrt{\alpha} = B,$$

and hence

$$\alpha = \Big(\sum_i \sqrt{w_i s_i} \Big)^2 / B^2.$$

The hit probability of file i then equals

$$h_i = U_i'^{-1}(\alpha s_i) = \frac{\sqrt{w_i}}{s_i \sqrt{\alpha}} = \frac{\sqrt{w_i}}{s_i \sum_j \sqrt{w_j}} B.$$

The utility function defined above is known to yield minimum potential delay fairness. It was shown in [40] that the TCP congestion control protocol implements such a utility function.

3.4.4 $\beta \to \infty$

Since $U_i(\cdot)$'s are defined to be concave functions, as $\beta \to \infty$ the solution of the optimization problems (3.3) and (3.9) converges to the *max-min fair* allocation (see [48] for proof). Therefore, we obtain

$$h_i = \min \{1, B / \sum_j s_j\}, \quad \forall i.$$

A brief summary of the utility functions discussed here is given in Table 3.1.

3.5 Reverse Engineering

In this section, we study the widely used replacement-based caching policies, FIFO and LRU, and show that their hit/miss behaviors can be duplicated in our framework through an appropriate choice of utility functions.

Table 3.1: The family of β-fair utility functions.

β	$\max \sum_i U_i(h_i)$	h_i	implication
0	$\max \sum w_i h_i$	$h_i = 1, i < m, h_m = (B - \sum_{i=1}^{m-1} s_i)/s_m, h_i = 0, i > m$	maximizing overall throughput
1	$\max \sum w_i \log h_i$	$h_i = w_i B/(s_i \sum_j w_j)$	proportional fairness
2	$\max -\sum w_i/h_i$	$h_i = \sqrt{w_i} B/(s_i \sum_j \sqrt{w_j})$	minimize potential delay
∞	$\max \min h_i$	$h_i = \min\{1, B/\sum_j s_j\}$	max-min fairness

It was shown in [25] that, with a proper choice of timer values, a TTL cache can approximately generate the same statistical properties, *i.e.* same hit/miss probabilities, as FIFO and LRU caching policies. In implementing these caches, non-reset and reset TTL caches are used for FIFO and LRU, respectively, with $t_i = T, i = 1, \ldots, N$ where T denotes the *characteristic time* [12] of these caches. For FIFO and LRU caches with Poisson arrivals the hit probabilities can be expressed as $h_i = 1 - 1/(1 + \lambda_i T)$ and $h_i = 1 - e^{-\lambda_i T}$, and T is computed such that $\sum_i s_i h_i = B$. For example for the LRU policy T is the unique solution to the fixed-point equation

$$\sum_{i=1}^{N} s_i \left(1 - e^{-\lambda_i T}\right) = B.$$

In our framework, we see from (3.5) that the file hit probabilities depend on the Lagrange multiplier α corresponding to the cache size constraint in (3.3). This suggests a connection between T and α. Further note that the hit probabilities are increasing functions of T. On the other hand, since utility functions are concave and increasing, $h_i = U_i'^{-1}(\alpha s_i)$ is a decreasing function of α. Hence, we can denote T as a decreasing function of α, *i.e.* $T = f(\alpha)$.

Different choices of function $f(\cdot)$ would result in different utility functions for FIFO and LRU policies. However, if we impose the functional dependence $U_i(h_i) = \lambda_i U_0(h_i)$, then the equation $h_i = U_i'^{-1}(\alpha s_i)$ yields

$$h_i = U_0'^{-1}(\alpha s_i/\lambda_i).$$

From the expressions for the hit probabilities of the FIFO and LRU policies, we obtain $T = 1/\alpha$. In the remainder of the section, we use this to derive utility functions for the FIFO and LRU policies.

3.5.1 FIFO

The hit probability of file i with request rate λ_i in a FIFO cache with characteristic time T is

$$h_i = 1 - \frac{1}{1+\lambda_i T}.$$

Substituting this into (3.5) and letting $T = 1/\alpha$ yields

$$U_i'^{-1}(\alpha s_i) = 1 - \frac{1}{1+\lambda_i/\alpha} = 1 - \frac{1}{1+\lambda_i s_i/(\alpha s_i)}.$$

Computing the inverse of $U_i'^{-1}(\cdot)$ yields

$$U_i'(x) = \lambda_i s_i\Big(\frac{1}{x} - 1\Big),$$

and integration of the two sides of the above equation yields the utility function for the FIFO cache

$$U_i(h_i) = \lambda_i s_i\Big(\log h_i - h_i\Big).$$

3.5.2 LRU

Taking $h_i = 1 - e^{-\lambda_i T}$ for the LRU policy and letting $T = 1/\alpha$ yields

$$U_i'^{-1}(\alpha s_i) = 1 - e^{-\lambda_i/\alpha} = 1 - e^{-\lambda_i s_i/(\alpha s_i)},$$

which yields

$$U_i'(x) = \frac{-\lambda_i s_i}{\ln(1-x)}.$$

Integration of the two sides of the above equation yields the utility function for the LRU caching policy

$$U_i(h_i) = \lambda_i s_i \mathrm{li}(1 - h_i),$$

where $\mathrm{li}(\cdot)$ is the logarithmic integral function

$$\mathrm{li}(x) = \int_0^x \frac{\mathrm{d}t}{\ln t}.$$

It is easy to verify, using the approach explained in Section 3.3, that the utility functions computed above indeed yield the correct expressions for the hit probabilities of the FIFO and LRU caches. We believe these utility functions are unique if restricted to be multiplicative in[2] λ_i.

Figure 3.1 depicts the utility functions for the hit probability of a file with $s_i = 1$ and $\lambda_i = 1$ for LRU and FIFO caches.

---

[2]We note that utility functions, defined in this context, are subject to affine transformations, *i.e.* $aU + b$ yields the same hit probabilities as U for any constant $a > 0$ and b.

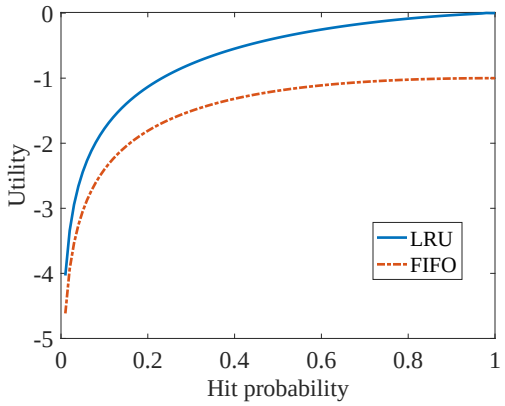


Figure 3.1: Utility functions associated with LRU and FIFO caching policies.

3.6 Online Algorithms

In Section 3.3, we formulated utility-driven caching as a convex optimization problem either with a fixed or an elastic cache size. However, it is not feasible to solve the optimization problem offline and then implement the optimal strategy. Moreover, the system parameters can change over time. Therefore, we need algorithms that can be used to implement the optimal strategy and adapt to changes in the system by collecting limited information. In this section, we develop such algorithms.

3.6.1 Dual Solution

The utility-driven caching formulated in (3.3) is a convex optimization problem, and hence the duality gap is zero, *i.e.*, solving the dual problem finds the optimal solution. The Lagrange dual of problem (3.3) is obtained by incorporating the constraints into the maximization by means of Lagrange multipliers

$$\underset{\alpha,\boldsymbol{\nu},\boldsymbol{\eta}}{\text{minimize}} \quad D(\alpha,\boldsymbol{\nu},\boldsymbol{\eta}) = \max_{h_i} \left\{ \sum_{i=1}^{N} U_i(h_i) - \alpha \left[\sum_{i=1}^{N} s_i h_i - B \right] - \sum_{i=1}^{N} \nu_i (h_i - 1) + \sum_{i=1}^{N} \eta_i h_i \right\}$$

such that $\alpha \geq 0, \quad \boldsymbol{\nu}, \boldsymbol{\eta} \geq \mathbf{0}$.

Using timer based caching techniques for controlling the hit probabilities with $0 < t_i < \infty$ ensures that $0 < h_i < 1$, and hence we have $\nu_i = 0$ and $\eta_i = 0$.

Here, we consider an algorithm based on the dual solution to the utility maximization problem (3.3). Note that with $\nu_i = 0$ and $\eta_i = 0$, we can write the Lagrange dual of the utility maximization problem as

$$D(\alpha) = \max_{h_i} \left\{ \sum_{i=1}^{N} U_i(h_i) - \alpha \left[\sum_{i=1}^{N} s_i h_i - B \right] \right\},$$

and the dual problem can be written as

$$\min_{\alpha \geq 0} D(\alpha).$$

A natural decentralized approach to consider for minimizing $D(\alpha)$ is to gradually move the decision variables towards the optimal point using the gradient descent algorithm. The gradient can be easily computed as

$$\frac{\partial D(\alpha)}{\partial \alpha} = -\Big(\sum_{i} s_i h_i - B \Big),$$

and since we are doing a gradient *descent*, α should be updated according to the *negative* of the gradient as

$$\alpha \leftarrow \max \Big\{ 0, \alpha + \gamma \Big(\sum_{i} s_i h_i - B \Big) \Big\},$$

where $\gamma > 0$ controls the step size at each iteration. Note that the KKT conditions require that $\alpha \geq 0$.

Based on the discussion in Section 3.3, to satisfy the optimality condition we must have

$$U_i'(h_i) = \alpha s_i,$$

or equivalently

$$h_i = U_i'^{-1}(\alpha s_i).$$

The hit probabilities are then controlled based on the timer parameters t_i which can be set according to (3.7) and (3.8) for non-reset and reset TTL caches.

Considering the hit probabilities as indicators of files residing in the cache, the expression $\sum_i s_i h_i$ can be interpreted as the number of items currently in the cache, denoted here as B_c. We can thus summarize the control algorithm for a reset TTL algorithm as

$$\begin{aligned} t_i &= -\frac{1}{\lambda_i} \log\Big(1 - U_i'^{-1}(\alpha s_i)\Big), \\ \alpha &\leftarrow \max\{0, \alpha + \gamma(B_c - B)\}. \end{aligned} \tag{3.11}$$

We obtain an algorithm for a non-reset TTL cache by using the correct expression for t_i in (3.7).

Let α^* denote the optimal value for α. We show in Appendix F that $D(\alpha) - D(\alpha^*)$ is a Lyapunov function and the above algorithm converges to the optimal solution.

3.6.2 Primal Solution

We now consider an algorithm based on the optimization problem in (3.9) known as the *primal* formulation.

Let $W(\mathbf{h})$ denote the objective function in (3.9) defined as

$$W(\mathbf{h}) = \sum_{i=1}^{N} U_i(h_i) - C(\sum_{i=1}^{N} s_i h_i - B).$$

A natural approach to obtain the maximum value for $W(\mathbf{h})$ is to use the gradient ascent algorithm. The basic idea behind the gradient ascent algorithm is to move the variables h_i in the direction of the gradient

$$\frac{\partial W(\mathbf{h})}{\partial h_i} = U_i'(h_i) - s_i C'(\sum_{i=1}^{N} s_i h_i - B).$$

Since the hit probabilities are controlled by the TTL timers, we move h_i towards the optimal point by updating t_i. Let $\dot{h_i}$ denote the derivative of the hit probability h_i with respect to time. Similarly, define $\dot{t_i}$ as the derivative of the timer parameter t_i with respect to time. We have

$$\dot{h_i} = \frac{\partial h_i}{\partial t_i} \dot{t_i}.$$

From the expressions for h_i, it is easy to confirm that $\partial h_i / \partial t_i > 0$ for non-reset and reset TTL caches. Therefore, moving t_i in the direction of the gradient, also moves h_i in that direction.

By gradient ascent, the timer parameters should be updated according to

$$t_i \leftarrow \max\left\{0, t_i + k_i\Big[U_i'(h_i) - s_i C'(B_c - B)\Big]\right\},$$

where $k_i > 0$ is the step-size parameter, and $\sum_{i=1}^{N} s_i h_i$ has been replaced with B_c based on the same argument as in the dual solution.

Let $\mathbf{h}^*$ denote the optimal solution to (3.9). We show in Appendix G that $W(\mathbf{h}^*) - W(\mathbf{h})$ is a Lyapunov function, and the above algorithm converges to the optimal solution.

3.6.3 Primal-Dual Solution

Here, we consider a third algorithm that combines elements of the previous two algorithms. Consider the control algorithm

$$t_i \leftarrow \max\{0, t_i + k_i[U_i'(h_i) - \alpha s_i]\},$$
$$\alpha \leftarrow \max\{0, \alpha + \gamma(B_c - B)\}.$$

Using Lyapunov techniques we show in Appendix H that the above algorithm converges to the optimal solution.

Now, rather than updating the timer parameters according to the above rule explicitly based on the utility function, we can have update rules based on a cache hit or miss. Consider the following differential equation

$$\dot{t}_i = \delta_m(t_i, \alpha)(1 - h_i)\lambda_i - \delta_h(t_i, \alpha)h_i\lambda_i, \tag{3.12}$$

where $\delta_m(t_i, \alpha)$ and $-\delta_h(t_i, \alpha)$ denote the change in t_i upon a cache miss or hit for file i, respectively. More specifically, the timer for file i is increased by $\delta_m(t_i, \alpha)$ upon a cache miss, and decreased by $\delta_h(t_i, \alpha)$ on a cache hit.

The equilibrium for (3.12) happens when $\dot{t_i} = 0$, which solving for h_i yields

$$h_i = \frac{\delta_m(t_i, \alpha)}{\delta_m(t_i, \alpha) + \delta_h(t_i, \alpha)}. \tag{3.13}$$

Comparing the above expression with $h_i = U_i'^{-1}(\alpha s_i)$ suggests that setting

$$\delta_m(t_i, \alpha) = U_i'^{-1}(\alpha s_i) \quad \text{and} \quad \delta_h(t_i, \alpha) = 1 - U_i'^{-1}(\alpha s_i),$$

can achieve desired hit probabilities and caching policies.

Note that depending on the utility function, multiplying $\delta_m(t_i, \alpha)$ and $\delta_h(t_i, \alpha)$ by a positive entity might yield simpler expressions. For example, to implement proportional fairness, we use

$$\delta_m(t_i, \alpha) = \lambda_i, \text{ and } \delta_h(t_i, \alpha) = \alpha s_i - \lambda_i. \tag{3.14}$$

Note that here we have multiplied the original expressions for $\delta_m(t_i, \alpha)$ and $\delta_h(t_i, \alpha)$ by αs_i.

In the case of max-min fairness, we want to equalize the hit probabilities at $h_i = B/\sum_j s_j$. This can be done by letting t_i evolve such that it increases when h_i is below a value shared across all files, and decreases when it is above that value. Such a dynamic can be implemented by increasing the timer t_i by one unit upon each

cache miss for file i, and decreasing it by $\alpha - 1$ upon each request for file i leading to a cache hit, *i.e.*

$$\delta_m(t_i, \alpha) = 1, \text{ and } \delta_h(t_i, \alpha) = \alpha - 1.$$

It is clear from (3.13) that at equilibrium, all files have the same hit probability $h_i = 1/\alpha$ irrespective of the object sizes s_i. Moreover, α will converge to $\sum_j s_j/B$ to maintain cache occupancy at B, yielding $h_i = B/\sum_j s_j$.

Note that with the above choices for $\delta_m(t_i, \alpha)$ and $\delta_h(t_i, \alpha)$ functions, max-min fairness can be implemented without requiring knowledge of request arrival rates λ_i.

3.6.4 Estimation of λ_i

Computing the timer parameter t_i in the algorithms discussed in this section requires knowing the request arrival rates for most of the policies. Estimation techniques can be used to approximate the request rates in case such knowledge is not available at the (cache) service provider.

Let r_i denote the remaining TTL time for file i. Note that r_i can be computed based on t_i and a time-stamp for the last time file i was requested. Let X_i denote the random variable corresponding to the inter-arrival times for the requests for file i, and $\bar{X}_i$ be its mean. We can approximate the mean inter-arrival time as $\hat{\bar{X}}_i = t_i - r_i$. Note that $\hat{\bar{X}}_i$ defined in this way is a one-sample unbiased estimator of $\bar{X}_i$. Therefore, $\hat{\bar{X}}_i$ is an unbiased estimator of $1/\lambda_i$. In the simulation section, we will use this estimator when computing the timer parameters for evaluating our algorithms.

3.7 Simulations

In this section, we perform experiments to understand the implications of the framework developed in Section 3.3, and evaluate the efficiency of the online algorithms developed in Section 3.6. Throughout this section, we assume that files are unit sized, *i.e.* $s_i = 1, \forall i$.

3.7.1 Cache Size Violations

In our utility-driven formulation in (3.3), the constraint is on the average cache occupancy, and hence it is possible to have more than B items in the cache. In Section 3.3.3, we discussed allocating a buffer space to prevent cache size violations. A buffer of size ϵB is added to the cache of size B, and a forced content eviction occurs only if number of items in the cache exceeds $(1+\epsilon)B$.

Here, we simulate an LRU cache implemented as a TTL cache. Recall from Section 3.3.3 that the probability of buffer constraint violation vanishes when $B(N) = o(N)$ and $\epsilon^2 B(N) = \omega(1)$. Note that here $s_{\max} = 1$. To serve a set of N contents, we set the average cache size to $B = \sqrt{N}$, and allocate the extra buffer by setting $\epsilon = \frac{1}{\sqrt[5]{N}}$. Figure 3.2(a) shows how the probability of forced content eviction decreases as the system parameters scale up. Figure 3.2(b) shows that the percentage of the extra buffer space allocated to prevent forced evictions decreases as cache size and number of files increase. Note that these results are in harmony with Theorem 6 from Section 3.3.3.

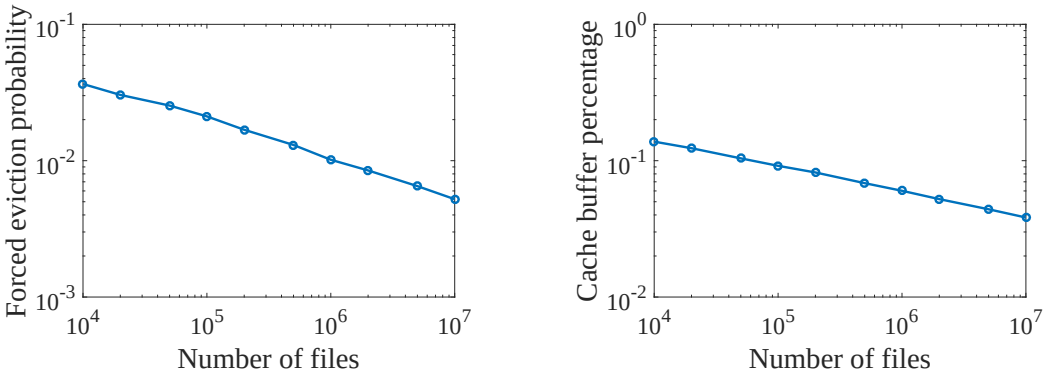


Figure 3.2: (a) Probability of cache size violation and (b) percentage of extra buffer space decrease as system scales as $B = \sqrt{N}$ and $\epsilon = \frac{1}{\sqrt[5]{N}}$.

3.7.2 Elastic Cache Size

In Section 3.3.2, we formulated the utility-driven caching problem with no strict constraint on cache size. In this case, cache size can grow arbitrarily large incurring a cost of storage. Optimal cache size is then computed by (3.10) to obtain maximum utility minus cost.

Here, we use the utility function computed in Section 3.5 for LRU to determine the optimal cache size. Figure 3.3 shows the optimal cache size as a function of the aggregate load on a cache implementing LRU policy, where the penalty function can be linear, quadratic, cubic or exponential:

$$C(x) = \begin{cases} 0.01\ x \\ 0.01\ x^2 \\ 0.01\ x^3 \\ 0.01\ e^x \end{cases} .$$

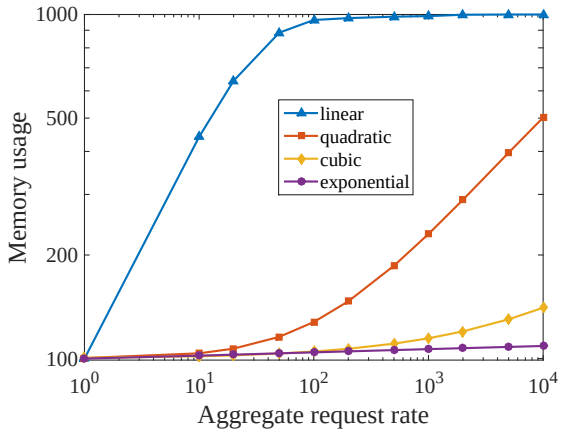


Figure 3.3: Adjusting the cache size to maximize utility-cost trade off for an LRU cache. Four different cost function are evaluated.

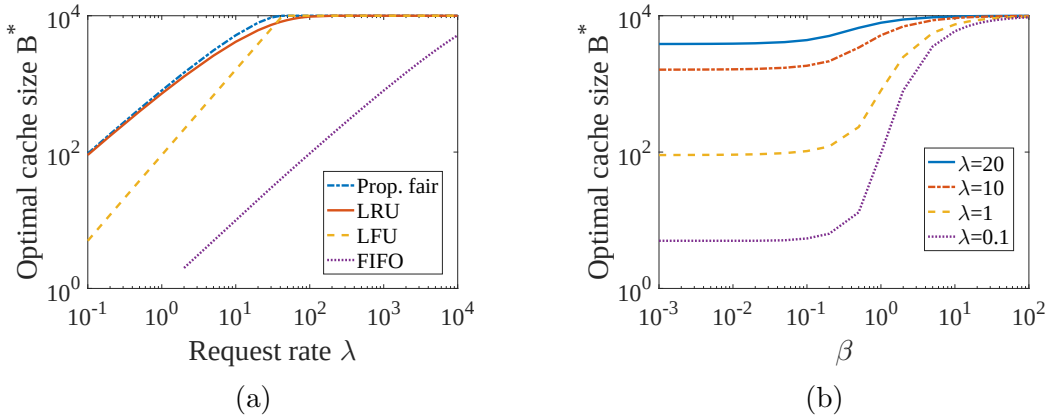


Figure 3.4: Optimal cache size as a function of (a) request arrival rate, and (b) fairness parameter β from β-fair utility functions.

Initially the cache size is set to $B = 100$, where requests are generated for $N = 1000$ files with popularities following a Zipf distribution with parameter 0.8.

We continue with the linear cost function, and compare the optimal cache size under four caching policies: proportionally fair, LRU, LFU, and FIFO. The number

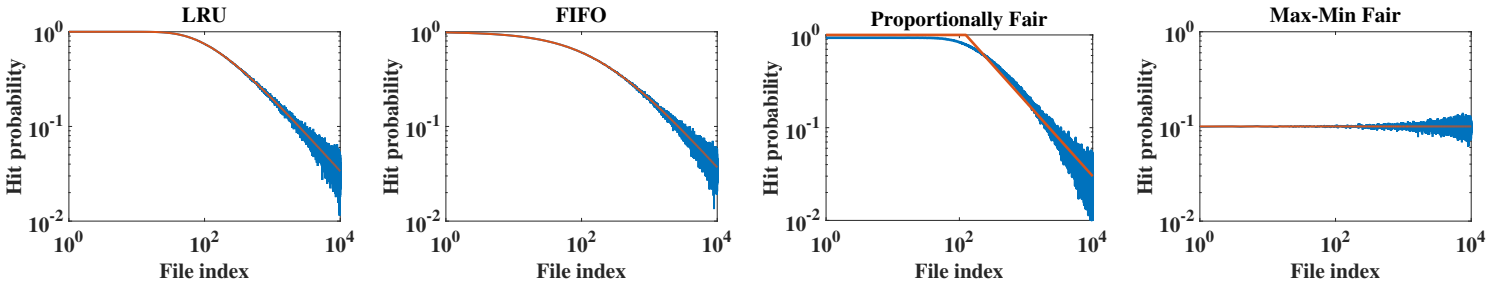


Figure 3.5: Hit probabilities from implementing the online dual algorithm using utility functions for LRU, FIFO, proportionally fair and max-min fair policies using estimated λ_i.

of files is assumed to be $N = 10^4$. Figure 3.4(a) shows how optimal cache size changes with arrival request rate for the mentioned policies. We can see that optimal cache size is an increasing function of the request rate. This is expected since as request rate increases, a larger utility is obtained from the cache.

In order to see how the choice of a caching policy affects optimal cache size, we take the β-fair utility functions and compute optimal cache size as a function of β. Figure 3.4(b) shows the results under four request arrival rates. We observe that optimal cache size increases as β increases.

3.7.3 Online Algorithms

Per our discussion in Section 3.5, non-reset and reset TTL caches can be used with $t_i = T, i = 1, \ldots, N$ to implement caches with the same statistical properties as FIFO and LRU caches. However, previous approaches require precomputing the cache characteristic time T. By using the online dual algorithm developed in Section 3.6.1 we are able to implement these policies with no a priori information of T. We do so by implementing non-reset and reset TTL caches, with the timer parameters for

all files set as $t_i = 1/\alpha$, where α denotes the dual variable and is updated according to (3.11).

For the proportionally fair policy, timer parameters are set to

$$t_i = \frac{-1}{\lambda_i} \log\left(1 - \frac{\lambda_i}{\alpha}\right),$$

and for the max-min fair policy we set the timers as

$$t_i = \frac{-1}{\lambda_i} \log\left(1 - \frac{1}{\alpha}\right).$$

We implement the proportionally fair and max-min fair policies as reset TTL caches.

In the experiments to follow, we consider a cache with the expected number of files in the cache set to $B = 1000$. Requests arrive for $N = 10^4$ files according to a Poisson process with aggregate rate $\sum_i \lambda_i = 1$. File popularities follow a Zipf distribution with parameter $z = 0.8$, *i.e.* $\lambda_i \propto 1/i^z$. In computing the timer parameters we use estimated values for the file request rates as explained in Section 3.6.4.

Figure 3.5 compares the hit probabilities achieved by our online dual algorithm with those computed numerically for the four policies explained above. It is clear that the online algorithms yield the exact hit probabilities for the FIFO, LRU and max-min fair policies. For the proportionally fair policy however, the simulated hit probabilities do not exactly match numerically computed values. This is due to an error in estimating $\lambda_i, i = 1, \ldots, N$. Note that we use a simple estimator here that produces an estimate that is unbiased for $1/\lambda_i$. However, the reciprocal of the estimate is not an unbiased estimate of λ_i. It is clear from the above equations that

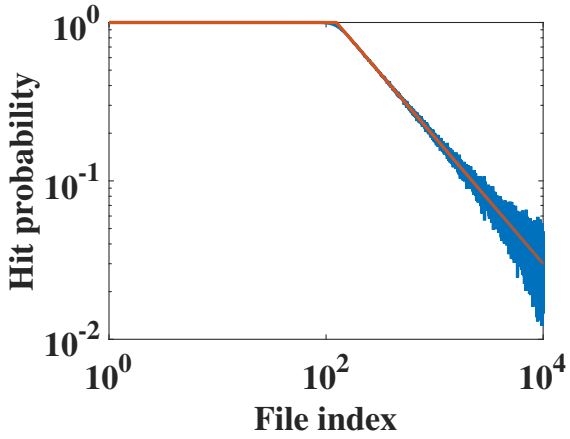


Figure 3.6: Proportionally fair policy implemented using the dual algorithm with exact knowledge of λ_is.

computing timer parameters for the max-min fair policy only require estimates of $1/\lambda_i$ and hence the results are good. Proportionally fair policy on the other hand requires estimating λ_i as well, hence using a biased estimate of λ_i introduces some error.

To confirm the above reasoning, we also simulate the proportionally fair policy assuming perfect knowledge of the request rates. Figure 3.6 shows that in this case simulation results exactly match the numerical values.

We can also use the primal-dual algorithm to implement the proportionally fair policy. Here, we implement this policy using the update rules in (3.14), and estimated values for the request rates. Figure 3.7 shows that, unlike the dual approach, the simulation results match the numerical values. This example demonstrates how one algorithm may be more desirable than others in implementing a specific policy.

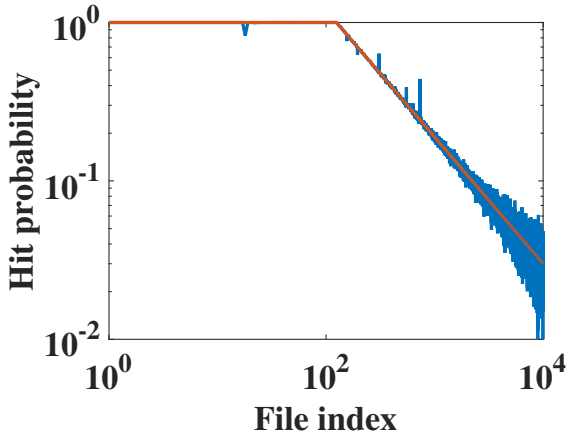
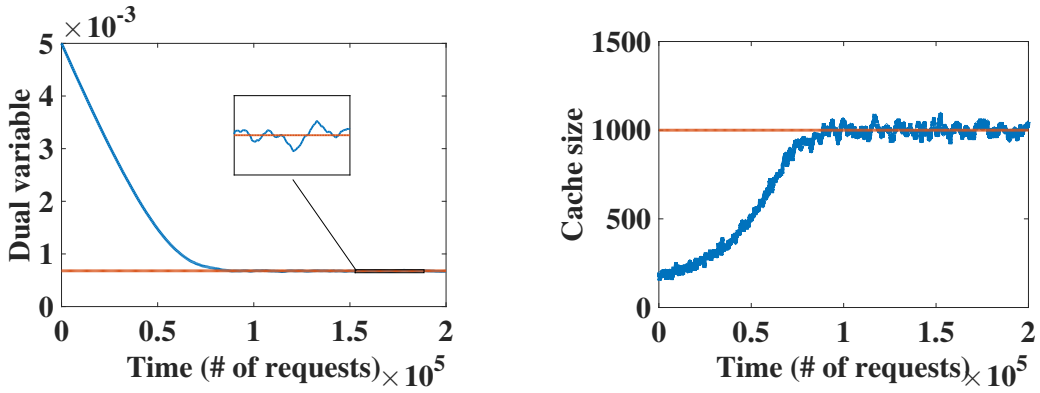


Figure 3.7: Proportionally fair policy implemented using the primal-dual algorithm with $\delta_m(t_i, \alpha) = \lambda_i$ and $\delta_h(t_i, \alpha) = \alpha - \lambda_i$, with approximate λ_i values.

The algorithms explained in Section 3.6 are proven to be globally and asymptotically stable, and converge to the optimal solution. Figure 3.8(a) shows the convergence of the dual variable for the LRU policy. The red line in this figure shows $1/T = 6.8 \times 10^{-4}$ where T is the characteristic time of the LRU cache computed according to the discussion in Section 3.5. Also, Figure 3.8(b) shows how the number of contents in the cache is centered around the capacity B. The probability density and complementary cumulative distribution function (CCDF) for the number of files in cache are shown in Figure 3.9. The probability of violating the capacity B by more than 10% is less than 2.5×10^{-4}. For larger systems, *i.e.* for large B and N, the probability of violating the target cache capacity becomes infinitesimally small; see the discussion in Section 3.3.3. This is what we also observe in our simulations. Similar behavior in the convergence of the dual variable and cache size is observed in implementing the other policies as well.



(a) (b)

Figure 3.8: Convergence and stability of dual algorithm for the utility function representing LRU policy.

To see how the algorithms compare in terms of converging to optimal solution, we compute the differences in hit probabilities between the optimal solution computed offline, h_i, and the hit probability resulted from the algorithm, $\hat{h}_i$, at different iterations of the algorithm. To compute the distance between the two probability distributions, we use the Kullback–Leibler (KL) divergence measure defined as follows

$$D_{KL} = \sum_{i=1}^{N} h_i \log \frac{h_i}{\hat{h}_i}.$$

Figure 3.10 shows divergence from optimal over time for proportionally fair and max-min fair policies implemented by dual and primal-dual algorithms assuming exact knowledge of request rates. The two algorithms are observed to perform similarly for both policies.

We then compare the algorithms when request arrival rates are not known but are estimated. Figure 3.11 shows divergence over time. In this case, the primal-dual

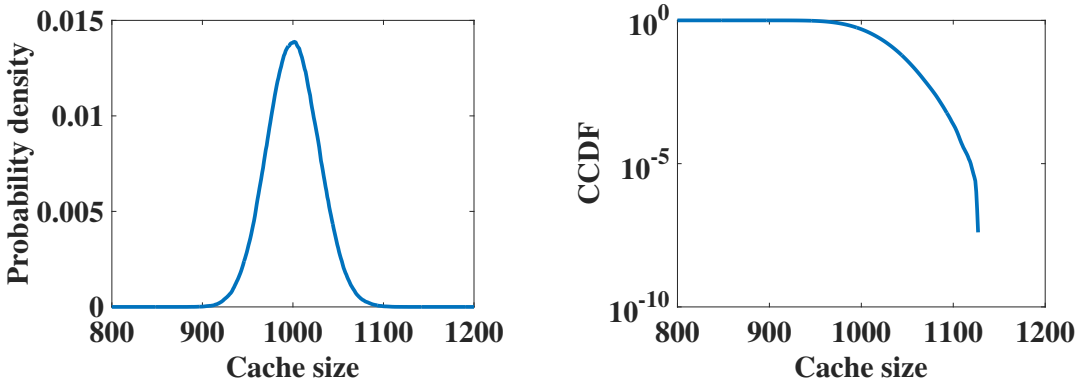


Figure 3.9: Cache size distribution and CCDF from dual algorithm with the utility function representing LRU policy.

algorithm performs much better than dual in implementing the proportionally fair policy. The two algorithms show a similar performance for max-min fair policy.

3.7.4 Non-reset vs. Reset TTL

The online algorithms described in Section 3.6 can be implemented as non-reset or reset TTL caches. Here, we perform experiments to understand how the choice of TTL type affects the performance of these algorithms. We take the proportionally fair and max-min fair policies and implement them as reset and non-reset TTL caches using the primal-dual algorithm. Using the KL-divergence measure, we compute the differences in hit probabilities between the optimal solution computed offline, h_i, and the hit probability resulted from the algorithm, $\hat{h}_i$, at different iterations of the algorithm. Figure 3.12 shows how divergence decreases over time for both non-reset and reset TTL implementations. We observe that the two implementations converge to the optimal solution at almost the same rate.

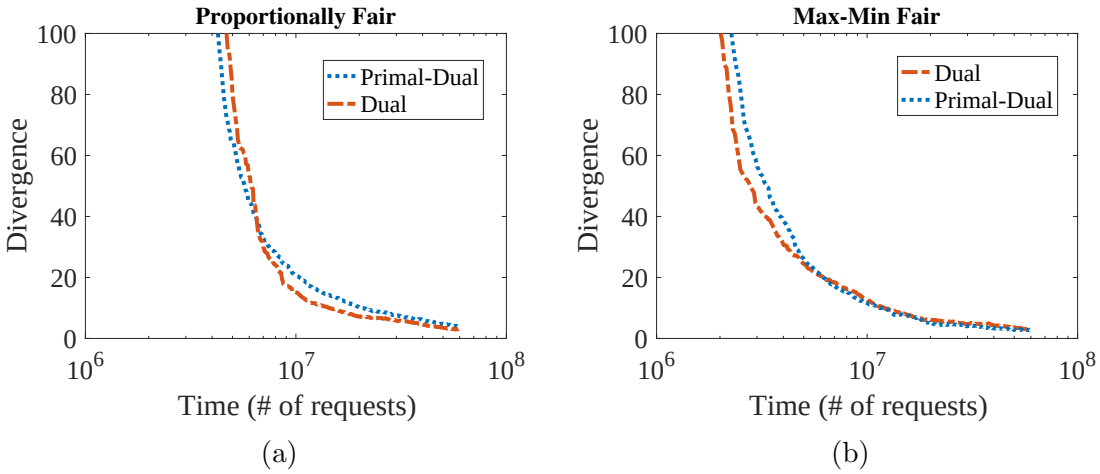


Figure 3.10: Divergence from optimal at algorithm iterations for (a) proportionally fair, and (b) max-min fair policies implemented by dual and primal-dual algorithms assuming exact knowledge of request rates.

3.7.5 Trace-driven Simulation

Earlier in this section, we showed that the LRU policy can be implemented as a TTL cache using the dual algorithm. To show that the dual algorithm can be used in realistic settings, we use requests from a trace for web accesses collected from a gateway router at IBM research lab [82]. We use the trace to compute cache hits for the replacement-based implementation of LRU, h_R, and the implementation based on the dual algorithm, h_D. We count the number of hits from each implementation over windows of $W = 3000$ requests and compute the relative error as

$$\text{relative error} = \frac{h_c - h_D}{W}.$$

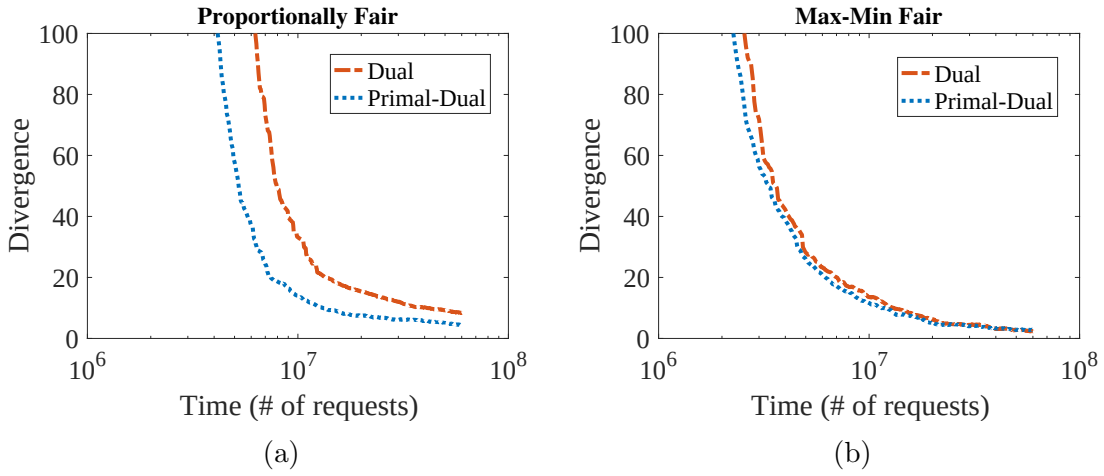


Figure 3.11: Divergence from optimal at algorithm iterations for (a) proportionally fair, and (b) max-min fair policies implemented by dual and primal-dual algorithms with estimated request rates.

Figure 3.13 shows the relative error over time. It is clear that the relative error is small meaning that the LRU implementation based on the dual algorithm performs close to its replacement-based implementation.

3.8 Related Work

Utility functions have been widely used in the modeling and control of computer networks, from stability analysis of queues to the study of fairness and service differentiation in network resource allocation; see [51,67] and references therein. Kelly [39] was the first to formulate the problem of rate allocation as one of achieving maximum aggregate utility for users, and describe how network-wide optimal rate allocation can be achieved by having individual users control their transmission rates. The work of Kelly *et al.* [40] presents the first mathematical model and analysis of

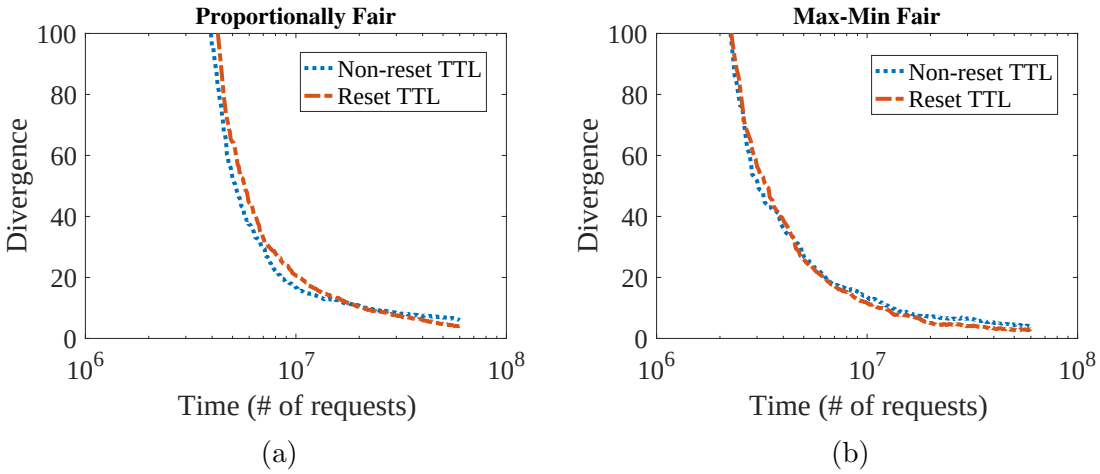


Figure 3.12: Divergence from optimal at algorithm iterations for (a) proportionally fair, and (b) max-min fair policies implemented by priml-dual algorithm as non-reset and reset TTL caches.

the behavior of congestion control algorithms for general topology networks. Since then, there has been extensive research in generalizing and applying Kelly's *Network Utility Maximization* framework to model and analyze various network protocols and architectures. This framework has been used to study problems such as network routing [73], throughput maximization [20], dynamic power allocation [52] and scheduling in energy harvesting networks [35], among many others.

The issue of service differentiation in the context of web cache management has also been extensively studied (*e.g.* see [23] and references therein). The majority of the work on this topic, however, uses cache partitioning as the means to provide service differentiation [42, 59, 69]. Ma and Towsley [46] have recently proposed using utility functions for the purpose of designing contracts that allow service providers to monetize caching. We build upon our previous work [17] which introduced a utility-

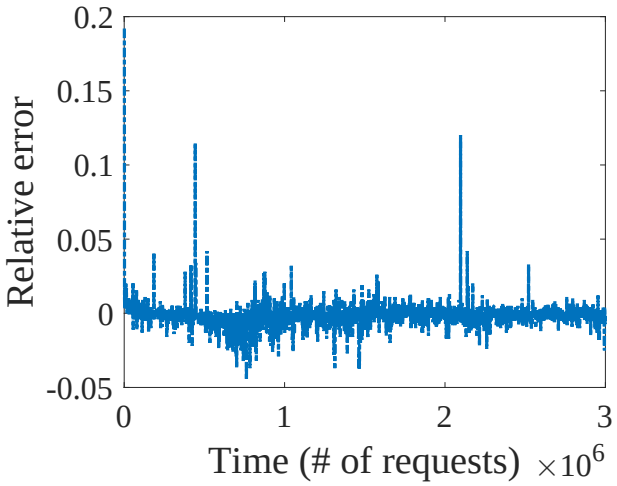


Figure 3.13: Relative error in hit counts from replacement-based implementation of LRU and the implementation based on the dual algorithm.

driven approach to managing cached content based on the utility associated with each content.

3.9 Discussion

In this section, we explore the implications of utility-driven caching on monetizing the caching service and discuss some future research directions.

3.9.1 Unequal File Sizes

The utility maximization framework in Section 3.3 assumes that utilities are defined as functions of *file* hit probability h_i. However, it is possible to define utilities as functions of *byte* hit probabilities, $U_i(s_i h_i)$. In this case, the following formulations can be used to implement a utility-driven caching policy:

$$\begin{aligned} \underset{h_i}{\text{maximize}} \quad & \sum_{i=1}^{N} U_i(s_i h_i) \\ \text{such that} \quad & \sum_{i=1}^{N} s_i h_i = B \\ & 0 \leq h_i \leq 1, \quad i = 1, 2, \ldots, N, \end{aligned}$$

Implications of the above formulation and its difference with (3.3) requires further investigation.

3.9.2 Decomposition

The formulation of the problem in Section 3.3 assumes that the utility functions $U_i(\cdot)$ are known to the system. In reality content providers might decide not to share their utility functions with the service provider. To handle this case, we decompose the optimization problem (3.3) into two simpler problems.

Suppose that cache storage is offered as a service and the service provider charges content providers at a constant rate r for storage space. Hence, a content provider needs to pay an amount of $w_i = rh_i$ to receive hit probability h_i for file i. The utility maximization problem for the content provider of file i can then be written as

$$\begin{aligned} \text{maximize} \quad & U_i(w_i/r) - w_i \\ \text{such that} \quad & w_i \geq 0 \end{aligned} \tag{3.15}$$

Now, assuming that the service provider knows the vector $\mathbf{w}$, for a proportionally fair resource allocation, the hit probabilities should be set according to

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{N} w_i \log h_i \\ \text{such that} \quad & \sum_{i=1}^{N} s_i h_i = B \end{aligned} \tag{3.16}$$

It was shown in [39] that there always exist vectors $\mathbf{w}$ and $\mathbf{h}$, such that $\mathbf{w}$ solves (3.15) and $\mathbf{h}$ solves (3.16); further, the vector $\mathbf{h}$ is the unique solution to (3.3).

3.9.3 Cost and Utility Functions

In Section 3.3.2, we defined a penalty function denoting the cost of using additional storage space. One might also define cost functions based on network bandwidth consumed in retrieving uncached content. This is especially interesting when modeling in-network caches with network links that are likely to be congested.

Optimization problem (3.3) uses utility functions defined as functions of the hit probabilities. It is reasonable to define utility as a function of the hit *rate*. How this affects the problem, *e.g.* the notion of fairness, is a question that requires further investigation. One argument in support of utilities as functions of hit rates is that a service provider might prefer pricing based on request rate rather than cache occupancy. Moreover, in designing hierarchical caches a service provider's objective could be to minimize the internal bandwidth cost. This can be achieved by defining the utility functions as $U_i = -C_i(m_i)$ where $C_i(m_i)$ denotes the cost associated with miss rate m_i for file i.

3.9.4 Online Algorithms

In Section 3.6, we developed three online algorithms that can be used to implement utility-driven caching. Although these algorithms are proven to be stable and converge to the optimal solution, they have distinct features that can make one algorithm more effective in implementing a policy. For example, implementing the max-min fair policy based on the dual solution requires knowing/estimating the file request rates, while it is easily implemented using the modified primal-dual solution without such knowledge. Moreover, the convergence rate of these algorithms may differ for different policies. The choice of non-reset or reset TTL caches also has implications on the design and performance of these algorithms. These are subjects that require further study.

3.9.5 Non-reset vs. Reset TTL

A utility-maximizing caching policy can be implemented as a reset or non-reset TTL cache. The choice of the TTL type might affect ease of implementation, sensitivity to arrival rate estimation or convergence time of algorithms. For example, while a TTL reset is a better choice for implementing LRU, a non-reset TTL is a better choice for implementing FIFO. Figure 3.12 in Section 3.7 shows that the same accuracy can be obtained by implementing proportionally fair and max-min fair policies as reset or non-reset TTL caches. The right choice of TTL type can simplify algorithm implementation as is the case for LRU and FIFO policies.

3.10 Conclusion

In this chapter, we proposed the concept of utility-driven caching, and formulated it as an optimization problem with rigid and elastic cache storage size constraints. Utility-driven caching provides a general framework for defining caching policies with considerations of fairness among various groups of files, and implications on market economy for (cache) service providers and content publishers. This framework has the capability to model existing caching policies such as FIFO and LRU, as utility-driven caching policies.

We developed three decentralized algorithms that implement utility-driven caching policies in an online fashion and that can adapt to changes in file request rates over time. We prove that these algorithms are globally stable and converge to the optimal solution. Through simulations we illustrated the efficiency of these algorithms and the flexibility of our approach.

CHAPTER 4

SHARING CACHE RESOURCES AMONG CONTENT PROVIDERS: A UTILITY-BASED APPROACH

4.1 Introduction

While there has been a flurry of recent research on the design of caching mechanisms for Information Centric Networks (ICNs) [8, 10, 47], relatively little attention has been paid to the problem of storage or *cache resource allocation among multiple content providers*. In this chapter, we consider content providers (CPs) as key players in procuring and delivering content in future ICNs. Given that it is part of the ICN data plane substrate, storage or cache must therefore be *multiplexed* or *shared* among multiple CPs, say, multiple (competing) video streaming service providers. This poses a fundamental research question that is pertinent to all ICN designs: *how to share or allocate the cache resource within a single network forwarding element and across various network forwarding elements among multiple content providers so as to maximize the cache resource utilization* or *provide best utilities to content providers?*

In this chapter, we develop a general utility maximization framework to address the aforementioned fundamental problem. We study the *multi-CP* cache allocation problem *in a single network cache element*. Our results illustrate the importance of

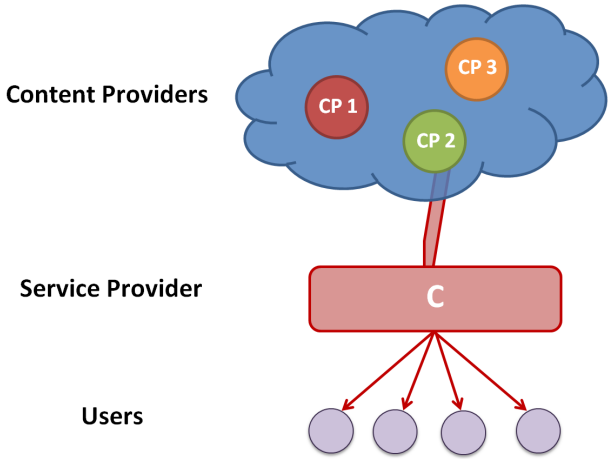


Figure 4.1: Network Model.

considering the cache allocation problem among multiple CPs in an ICN and has implications on ICN architectural design: from the perspective of cache resource efficiency or utility maximization of CPs, cache partitioning (among CPs) should be a basic principle for cache resource allocation in ICNs.

The main contribution of this chapter is the formulation of a utility–based optimization framework for maximizing content provider utilities subject to cache capacity constraints at the service provider. We formulate the problem for two scenarios: 1) where content providers serve distinct sets of contents, 2) where some content is served by multiple content providers. We also develop online algorithms for managing cache partitions, and prove the convergence of these algorithms to the optimal solution using Lyapunov functions.

4.2 Model and Problem Setting

Consider a network as shown in Figure 4.1, where users access content, *e.g.* videos, from K content providers (CPs). CP k ($k = 1, \ldots, K$) serves a set S_k of n_k unit size files where $n_k = |S_k|$; we will usually label these files $i = 1, \ldots, n_k$. All CPs share a content cache, supplied by a third-party network provider, referred to as a service provider hereafter. Content providers have business relations with the service provider and pay for cache resources. There are two possible scenarios: i) the content objects offered by the CPs are all *distinct*; and ii) some *common* objects are provided by different CPs. Due to disparate user bases, the access patterns of these content objects may vary across the CPs.

We assume that requests are described by a Poisson process with request rate for file i of CP k being $\lambda_{k,i} = \lambda_k p_{k,i}, i \in S_k, k = 1, \ldots, K$, where λ_k denotes the aggregate request rate for contents from CP k, and $p_{k,i}$ is the probability that a request to CP k is for content i. Associated with each CP is a utility $U_k(x)$ that is an *increasing* and *concave* function of parameter x. We focus on the problem where service provider wishes to maximize the sum of the utilities over all content providers, $\sum_k U_k(h_k)$, through a proper allocation of cache space to the CPs. Here, h_k is the hit rate of CP k over all its files. In the simple case where $U_k(h_k) = h_k$, the objective becomes that of maximizing the overall cache hit rate, which provides a measure of overall cache utilization.

Cache Partitioning: When the cache is shared among the CPs, content objects offered by all CPs compete for the storage space based on their access patterns. To restrict cache contention to smaller sets of content, the service provider can form

content groups from files served by a CP or multiple CPs, and partition the cache into slices and dedicate a partition to each content group. Let P denote the number of content groups/cache partitions. Also, let V_p and $C_p, p = 1, \ldots, P$ denote the content groups and partition sizes, respectively. Note that $P = 1$ implies that the cache is shared as a whole, while $P > 1$ means it is partitioned.

The first question to ask is: *what is the optimal number of partitions and how should files be grouped?* To determine the number of slices and that what files should be requested from which partition, the service provider proceeds as follows. Files are first grouped into disjoint sets according to which content providers serve them. The service provider then decides how many partitions to create, and whether to dedicate a separate partition for each set of files, or have multiple sets of files share a partition. In the next section, we explain what changes if the cache manager made partitioning decisions on a per file basis rather than sets of files.

Assuming the answer to the first question, the second question is: *how should the partitions be sized?* Let $\mathbf{C} = (C_1, C_2, \ldots, C_P)$ denote the vector of partition sizes. For each content provider k, hit rate is a function of the partition sizes $h_k(\mathbf{C})$. For a cache of size C, we formulate this question as the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \sum_{k=1}^{K} U_k\Big(h_k(\mathbf{C})\Big) \\ \text{such that} \quad & \sum_{p=1}^{P} C_p \leq C \\ & C_p = 0, 1, 2, \ldots; \quad p = 1, 2, \ldots, P. \end{aligned}$$

Note that the above formulation is an integer programming problem that is typically hard to solve. However, in practice caches are large and therefore we assume C_p can take any real value, as the rounding error will be negligible.

Cache Characteristic Time: Assume a cache of size C serving n contents with popularity distribution $p_i,\ i = 1, \ldots, n$. Under the independent reference model (requests are i.i.d.), Fagin [21] introduced the notion of a *window size* T that satisfies

$$C = \sum_{i=1}^{n} (1 - (1 - p_i)^T).$$

The miss probability associated with a window of size T is defined as

$$m(T) = \sum_{i=1}^{n} p_i (1 - p_i)^T.$$

Fagin introduced a cumulative probability distribution, F, that is continuously differentiable in $(0, 1)$ with $F(0) = 0$ and $F(1) = 1$. The right-derivative of F at 0, denoted by $F'(0)$, may be infinite. This will allow us to account for Zipf-like distributions. Define

$$p_i^{(n)} = F(i/n) - F((i-1)/n), \quad i = 1, \ldots, n$$

the probability that page i is requested. Hereafter, we will refer to F as the popularity distribution. If $C/n = \beta$ then $T/n \to \tau_0$ where (see [21])

$$\beta = \int_0^1 (1 - e^{-F'(x)\tau_0}) dx \tag{4.1}$$

and $m(T) \to \mu$ where

$$\mu = \int_0^1 F'(x) e^{-F'(x)\tau_0} dx. \tag{4.2}$$

Moreover, μ is the limiting miss probability under LRU when $n \to \infty$.

Suppose requests arrive according to a Poisson process with rate λ. Express β as

$$\beta = \int_0^1 P(X(x) < \tau_0/\lambda) dx$$

where $X(x)$ is an exponential random variable with intensity $\lambda F'(x)$. $X(x)$ is the inter-arrival time of two requests for content of type x. If this time is less than τ_0/λ, then the request is served from the cache, otherwise it is not. In practice, as n is finite, this is approximated by

$$C = \beta n = \sum_{i=1}^{n} (1 - e^{-\lambda p_i^{(n)} T_c}) \tag{4.3}$$

where T_c is the Characteristic Time (CT) for the finite content cache [12]. The aggregate miss probability is approximated by

$$m(T_c) = 1 - \sum_{i=1}^{n} p_i^{(n)} \big(1 - e^{-\lambda p_i^{(n)} T_c}\big). \tag{4.4}$$

Fagin's results suffice to show that as $n \to \infty$, the r.h.s. of (4.4) converges to the LRU miss probability.

In the context of K providers, let $n_k = b_k n$, $n, b_k \in \mathbb{N}$, $k = 1, \ldots, K$. Denote $B_k := \sum_{j=1}^{k} b_j$ with $B_0 = 0$ by convention. It helps also to denote B_K by B. Let

$F_1, F_2, \ldots, F_K$ be continuous uniformly differentiable CDFs in $(0,1)$. $F_k'(0)$ may be infinite for $k = 1, \ldots K$. If each provider has a cache that can store a fraction β_k of its contents, then the earlier described CT approximation, (4.3), (4.4), applies with

$$p_{k,i}^{(n)} = F_k\Big(\frac{i}{b_k n}\Big) - F_k\Big(\frac{i-1}{b_k n}\Big), \quad i = 1, \ldots, b_k n. \tag{4.5}$$

We denote the asymptotic miss probabilities for the K caches, each using LRU, by

$$\mu_k^{(p)} = \int_0^1 F_k'(x) e^{-F_k'(x)\tau_k} dx, \quad k = 1, \ldots, K \tag{4.6}$$

where τ_k is the solution of (4.1) with β replaced by β_k.

Assume that the providers share a cache of size C. Define $\beta^{(s)} = \sum_{k=1}^K \beta_k$. We introduce $\mu^{(s)}$ and τ_0 through the following two equations,

$$\mu^{(s)} = \sum_{k=1}^K a_k \int_0^1 F_k'(x) e^{-a_k F_k'(x)\tau_0 B/b_k} dx, \tag{4.7}$$

$$\beta^{(s)} = 1 - \sum_{k=1}^K \frac{b_k}{B} \int_0^1 e^{-a_k F_k'(x)\tau_0 B/b_k} dx \tag{4.8}$$

where $a_k := \lambda_k/\lambda$, $k = 1, \ldots, K$.

Theorem 7. *Assume that we have K providers with popularity distributions $F_1, \ldots, F_K$ as defined above, with numbers of contents given by $b_k n, b_k \in \mathbb{N}, n = 1, 2, \ldots$ and request rates λ_k, $k = 1, \ldots, K$. Construct the sequence of popularity probabilities $\{p_{k,i}^{(n)}\}$, $n = 1, \ldots$ defined in (4.5) and cache sizes $C^{(n)}$ such that $C^{(n)}/n = \beta$. Then, the aggregate miss probability under LRU converges to $\mu^{(s)}$ given in (4.7), where τ_0 is the unique solution of (4.8).*

Proof. See Appendix J. □

Remark. This extends Fagin's results to include any asymptotic popularity CDF F that is continuously differentiable in $(0, 1)$ except at a countable number of points.

4.3 Cache Resource Allocation among Content Providers

In this section, we formulate cache management as a utility maximization problem. We introduce two formulations, one for the case where content providers serve distinct contents, and another one for the case where some contents are served by multiple providers.

4.3.1 Content Providers with Distinct Objects

Consider the case of K providers with $n_k = b_k n$ contents each where $b_k, n \in \mathbb{N}$ and $k = 1, \ldots, K$. Also, let $B = \sum_{k=1}^{K} b_k$ Assume that requests to CP k is characterized by a Poisson process with rate λ_k.

We ask the question whether the cache should be shared or partitioned between CPs under the LRU policy. It is easy to construct cases where sharing the cache is beneficial. However these arise when the cache size and the number of contents per CP are small. Evidence suggests that partitioning provides a larger aggregate utility than sharing as cache size and number of contents grow. In fact, the following theorem shows that asymptotically, under the assumptions of Theorem 1, in the limit as $n \to \infty$, the sum of utilities under LRU when the cache is partitioned, is at least as large as it is under LRU when the CPs share the cache. To do this, we

formulate the following optimization problem: namely to partition the cache among the providers so as to maximize the sum of utilities:

$$\begin{aligned} \max_{\beta_k} U^{(p)} &:= \sum_{k=1}^{K} U_k(\lambda_k(1-\mu_k(\beta_k))) \\ \text{s.t. } \beta &= \sum_{k=1}^{K} \frac{b_k}{B}\beta_k, \\ \beta_k &\geq 0, \quad k = 1, 2, \ldots, K. \end{aligned} \tag{4.9}$$

Observe that $\lambda_k(1-\mu_k(\beta_k))$ in (4.9) is the hit rate of documents of CP k, where $\mu_k(\beta_k)$ is given by (4.6).

Here, μ_k is the asymptotic miss probability for content served by CP k, β is the cache size constraint expressed in terms of the fraction of the aggregate content that can be cached, b_k/B is the fraction of content belonging to provider k, and β_k is the fraction of CP k content that is permitted in CP k's partition. The direct dependence of β_k on μ_k is difficult to capture. Hence, we use (4.1) and (4.2), to transform the above problem into:

$$\begin{aligned} \max_{\tau_k} U^{(p)} &:= \sum_{k=1}^{K} U_k\Big(\lambda_k\big(1-\int_0^1 e^{-F'(x)\tau_k}dx\big)\Big) \\ \text{s.t. } \beta &= 1 - \sum_{k=1}^{K} \frac{b_k}{B}\int_0^1 e^{-F'_k(x)\tau_k}dx, \\ \tau_k &\geq 0, \quad k = 1, 2, \ldots, K. \end{aligned} \tag{4.10}$$

Theorem 8. *Assume K providers with popularity distributions constructed from distributions $F_1, \ldots, F_K$ using (4.5) with number of contents $b_k n$ and request rates*

λ_k, $k = 1, \ldots, K$ *sharing a cache of size* $C^{(n)}$ *such that* $C^{(n)}/n = \beta$. *Then, as* $n \to \infty$ *the sum of utilities under partitioning is at least as large as that under sharing.*

Proof. The sum of utilites for the shared case is

$$U^{(s)} = \sum_{k=1}^{K} U_k\Big(\lambda_k\big(1 - \int_0^1 e^{-F'(x)\tau_0} dx\big)\Big)$$

where τ_0 is the unique solution to

$$\beta = 1 - \sum_{k=1}^{K} \frac{b_k}{B} \int_0^1 e^{-F'_k(x)\tau_0} dx.$$

When we set $\tau_k = a_k B \tau_0 / b_k$ in $U^{(p)}$ then $U^{(p)} = U^{(s)}$, proving the theorem. □

Based on the above theorem, we focus solely on partitioned caches and use the CT approximation to formulate the utility–maximizing resource allocation problem for content providers with distinct files as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{k=1}^{K} U_k\Big(h_k(C_k)\Big) \\ \text{such that} \quad & \sum_{k=1}^{K} C_k \leq C, \\ & C_k \geq 0, \quad k = 1, 2, \ldots, K. \end{aligned} \tag{4.11}$$

In our formulation, we assume that each partition employs LRU for managing the cache content. Therefore, we can compute the hit rate for content provider k as

$$h_k(C_k) = \lambda_k \sum_{i=1}^{n_k} p_{k,i}\big(1 - e^{-\lambda_k p_{k,i} T_k(C_k)}\big), \tag{4.12}$$

where $T_k(C_k)$ denotes the characteristic time of the partition with size C_k dedicated to content provider k. $T_k(C_k)$ is the unique solution to the equation

$$C_k = \sum_{i=1}^{n_k} (1 - e^{-\lambda_k p_{k,i} T_k}). \tag{4.13}$$

The following theorem establishes that resource allocation problem (4.11) has a unique optimal solution:

Theorem 9. *Given strictly concave utility functions, the resource allocation problem* (4.11) *has a unique optimal solution.*

Proof. In Appendix I, we show that $h_k(C_k)$ in (4.12) is an increasing concave function of C_k. Since U_k is assumed to be an increasing and strictly concave function of the cache hit rate h_k, it follows that U_k is an increasing and strictly concave function of C_k. The objective function in (4.11) is a linear combination of strictly concave functions, and hence is concave. Since the feasible solution set is convex, a unique maximizer called the optimal solution exists. □

Last, it is straightforward to show that partitioning is at least as good as sharing, for finite size systems using the CT approximation.

4.3.2 Content Providers with Common Objects

Here, we first assume there are only two content providers in the network and then consider the general case. There are three sets of content, S_0 of size n_0 served by both providers, and S_1 and S_2, sizes n_1 and n_2 served separately by each of the providers. Requests are made to S_0 at rate $\lambda_{0,k}$ from provider k and to S_k at rate λ_k.

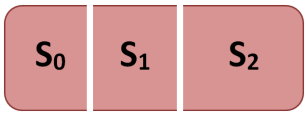


Figure 4.2: Partitioning cache into three slices. One partition for the set of common files, S_0, and two other partitions, one for the remaining files from each content provider, S_k.

Given a request is made to S_0 from provider k, it is for content i with probability $p_{0,k,i}$. Similarly, if the request is for content in S_k, $k = 1, 2$, it is for content i with probability $p_{k,i}$.

We have two conceivable cases for the files in S_0: 1) each content provider needs to maintain its own copy of the content, *e.g.* due to security reasons, or 2) one copy can be kept in cache to serve requests to either of the content providers. The first case can be treated as if there is no common content between the two content providers, and hence can be cast as problem (4.11). For the second case, we consider three strategies for managing the cache:

- **Strategy 1 (S1):** sharing the whole cache as one large partition,
- **Strategy 2 (S2):** partitioning into two dedicated slices, one for each CP,
- **Strategy 3 (S3):** partitioning into three slices, one shared partition for the set of common contents, and two other partitions for the remaining files of each CP, as shown in Figure 4.2.

The following theorem states that **S3** performs at least as well as **S1** in an asymptotic sense.

Theorem 10. *Assume that we have two providers with a set of shared files S_0, and sets of non-shared files, S_1, S_2 with numbers of files $n_k = b_k n$, $b_k, n \in \mathbb{N}$, and*

$k = 0, 1, 2$. *Assume that requests to these sets occur with rates* $\lambda_{0,k}$ *and* λ_k *and content popularities are described by asymptotic popularity distributions* $F_{0,k}$, *and* F_k. *Construct the sequence of popularity probabilities* $\{p_{k,i}^{(n)}\}$, $n = 1, \ldots$ *similar to* (4.5) *and cache sizes* $C^{(n)}$ *such that* $C^{(n)}/n = \beta$. *Then, the asymptotic aggregate LRU miss probability is at least as small under* **S3** *as under* **S1**.

The proof is similar to the proof of Theorem 8.

Neither **S2** nor **S3** outperforms the other for all problem instances, even asymptotically. However, we present a class of workloads for which asymptotically **S3** outperforms **S2** and then follow it with an example where **S2** outperforms **S3**.

Consider the following workload where the asymptotic popularity distributions of requests to the two providers for the shared content are identical, $F_{0,1} = F_{0,2}$.

Theorem 11. *Assume that we have two providers with a set of shared files* S_0 *and sets of non-shared files,* S_1, S_2 *with numbers of files* $n_k = b_k n$, $b_k \in \mathbb{N}$, $n = 1, \ldots,$ *and* $k = 1, 2, 3$. *Assume that requests are described by Poisson processes with rates* $\lambda_{0,k}$ *and* λ_k, $k = 1, 2$, *and content popularities are described by asymptotic popularity distributions* $F_{0,1} = F_{0,2}$, *and* F_1, F_2. *Construct the sequence of popularity probabilities* $\{p_{k,i}^{(n)}\}$, $n = 1, \ldots$ *similar to* (4.5) *and cache sizes* $C^{(n)}$ *such that* $C^{(n)}/n = \beta$. *Then the asymptotic aggregate hit probability under LRU is at least as large under* **S3** *as under* **S2**.

The proof is found in Appendix K

Below is an example where **S2** outperforms **S3**. The asymptotic popularity distributions for the shared content are given by

$$F_{0,1}(x) = \begin{cases} 2x/11 & 0 < x \leq 1/2 \\ (20x-9)/11 & 1/2 < x < 1 \end{cases}$$

and

$$F_{0,2}(x) = \begin{cases} 300x/151 & 0 < x \leq 1/2 \\ (2x+149)/151 & 1/2 < x < 1 \end{cases}$$

with request rates $\lambda_{0,1} = 1.1$ and $\lambda_{0,2} = 15.1$. The asymptotic popularities of the non-shared contents are $F_1(x) = F_2(x) = x$ with request rates $\lambda_1 = 20$ and $\lambda_2 = 30$. Last, there are equal numbers of content in each of these sets, $n_0 = n_1 = n_2$. If we set $\beta = 2/3$, then the aggregate hit probability under **S3** with optimal partitioning is 0.804, which is slightly lower than the aggregate hit probability, 0.816, under **S2** with optimal partitioning.

The above examples show that the workloads of the content providers can affect which strategy is optimal. However, we argue that partitioning into three slices should provide the best performance in most practical situations, where content providers have similar popularity patterns for the contents they commonly serve. This is unlike the second example where the two content providers have disparate rates for the common contents they serve. In Section 5.6, we will show that even if two content providers have dissimilar request rates for their common contents, partitioning into three slices does better.

Based on the above argument for the performance of partitioning into three slices in the case of two content providers, for K content providers with common files, one should create a partition for each set of files that are served by a number of content

Algorithm 3 Partitioning a Cache serving K content providers with possibility of common files among some content providers.

$S \leftarrow S_1 \cup S_2 \cup \ldots \cup S_K$.
$\mathcal{P} \leftarrow \emptyset$.
for $f \in S$ **do**
    $M_f \leftarrow \{k :$ Content provider k serves files $f\}$.
    if Exists $(V, M) \in P$ such that $M = M_f$ **then**
        $V \leftarrow V \cup \{f\}$.
    else
        $\mathcal{P} \leftarrow \mathcal{P} \cup \{(\{f\}, M_f)\}$.

providers. A procedure for creating the optimal set of partitions $\mathcal{P}$ with the files routed to each partition is given in Algorithm 3. Algorithm 3 runs in $O(|S|^2)$ where S denotes the set of all files served by all content providers. Note that the number of partitions can grow exponentially with the number of content providers.

Once the set of partitions $\mathcal{P}$ and the set of files corresponding to each partition is determined, the optimal partition sizes can be computed through the following optimization problem:

$$\begin{aligned} \text{maximize} \quad & \sum_k U_k(h_k) \\ \text{such that} \quad & \sum_{p=1}^{|\mathcal{P}|} C_p \leq C \\ & C_p \geq 0, \quad p = 1, 2, \ldots, |\mathcal{P}|, \end{aligned} \tag{4.14}$$

where hit rate for CP k is computed as

$$h_k = \sum_{p=1}^{|\mathcal{P}|} \lambda_k \sum_{i \in V_p} p_{k,i} \big(1 - e^{-\lambda_i p_{k,i} T_p(C_p)}\big)$$

where V_p denotes the set of files requested from partition p, and $\lambda_i \triangleq \sum_k \lambda_{k,i}$ denotes the aggregate request rate for content i through all content providers, and T_p denotes the characteristic time of partition p.

Theorem 12. *Given strictly concave utility functions, resource allocation problem* (4.14) *has a unique optimal solution.*

Proof. In Appendix L, we show that the optimization problem (4.14) has a concave objective function. Since the feasible solution set is convex, a unique maximizer exists. □

4.3.3 Implications

Cache Partitioning: Number of Slices, Management Complexity and Static Caching. In our utility maximization formulations (4.11) and (4.14) and their solution, the cache is only partitioned and allocated per CP for a set of *distinct* content objects owned by the CP; a cache slice is allocated and shared among several CPs only for a set of *common* content objects belonging to these CPs. This is justified by cache management complexity considerations, as further partitioning of a slice allocated to a CP to be exclusively utilized by the same CP simply incurs additional management complexity. In addition, we show in Appendix M that partitioning a cache slice into smaller slices and *probabilistically routing* requests to content objects of a CP is sub-optimal.

As an alternative to CP-oriented cache allocation and partitioning approach, one could adopt a per-object cache allocation and partitioning approach (regardless of the CP or CPs which own the objects). Under such an approach, it is not hard

to show that the optimal *per-object* cache allocation strategy that maximizes the overall cache hit rate is equivalent to the *static caching* policy [44]: the cache is only allocated to the C most popular objects among all content providers. Alternatively, such a solution can also be obtained using the same CP-oriented, utility maximization cache allocation framework where only the most popular content from each provider is cached.

Utility Functions and Fairness. Different utility functions in problems (4.11) and (4.14) yield different partition sizes for content providers. In this sense, each utility function defines a notion of fairness in allocating storage resources to different content providers. The family of α-fair utility functions expressed as

$$U(x) = \begin{cases} \frac{x^{1-\alpha}-1}{1-\alpha} & \alpha \geq 0, \alpha \neq 1; \\ \\ \log x & \alpha = 1, \end{cases}$$

unifies different notions of fairness in resource allocation [67]. Some choices of α lead to especially interesting utility functions. Table 4.1 gives a brief summary of these functions. We will use these utilities in Section 5.6 to understand the effect of particular choices for utility functions, and in evaluating our proposed algorithms.

4.4 Online Algorithms

In the previous section, we formulated cache partitioning as a convex optimization problem. However, it is not feasible to solve the optimization problem offline and then implement the optimal strategy. Moreover, system parameters can change over

Table 4.1: α-fair utility functions

α	$U_k(h_k)$	implication
0	h_k	hit rate
1	$\log h_k$	proportional fairness
2	$-1/h_k$	potential delay
∞	$\lim_{\alpha\to\infty} \frac{h_k^{1-\alpha}-1}{1-\alpha}$	max-min fairness

time. Therefore, we need algorithms that can implement the optimal strategy and adapt to changes in the system by collecting limited information. In this section, we develop such algorithms.

4.4.1 Content Providers with Distinct Contents

The formulation in (4.11) assumes a hard constraint on the cache capacity. In some circumstances it may be appropriate for the cache manager to increase the available storage at some cost to provide additional resources for the content providers. One way of doing this is to turn cache storage disks on and off based on demand [70]. In this case, the cache capacity constraint can be replaced with a penalty function $P(\cdot)$ denoting the cost for the extra cache storage. Here, $P(\cdot)$ is assumed to be convex and increasing. We can now write the utility and cost driven caching formulation as

$$\begin{aligned} \text{maximize} \quad & \sum_k U_k(h_k(C_k)) - P(\sum_k C_k - C) \\ \text{such that} \quad & C_k \geq 0, \quad k = 1, \ldots, K. \end{aligned} \tag{4.15}$$

Let $W(\mathbf{C})$ denote the objective function in (4.15) defined as

$$W(\mathbf{C}) = \sum_k U_k(h_k(C_k)) - P(\sum_k C_k - C).$$

A natural approach to obtaining the maximum value for $W(\mathbf{C})$ is to use a gradient ascent algorithm. The basic idea behind a gradient ascent algorithm is to move the variables C_k in the direction of the gradient,

$$\begin{aligned}\frac{\partial W}{\partial C_k} &= \frac{\partial U_k}{\partial C_k} - P'(\sum_k C_k - C), \\ &= U_k'(h_k)\frac{\partial h_k}{\partial C_k} - P'(\sum_k C_k - C).\end{aligned}$$

Note that since h_k is an increasing function of C_k, moving C_k in the direction of the gradient also moves h_k in that direction.

By gradient ascent, partition sizes should be updated according to

$$C_k \leftarrow \max\Big\{0, C_k + \gamma_k\Big[U_k'(h_k)\frac{\partial h_k}{\partial C_k} - P'(\sum_k C_k - C)\Big]\Big\},$$

where γ_k is a step-size parameter.

Theorem 13. *The above gradient ascent algorithm converges to the optimal solution.*

Proof. Let $\mathbf{C}^*$ denote the optimal solution to (4). We show in Appendix N that $W(\mathbf{C}^*) - W(\mathbf{C})$ is a Lyapunov function, and the above algorithm converges to the optimal solution. □

4.4.1.1 Algorithm Implementation

In implementing the gradient ascent algorithm, we restrict ourselves to the case where the total cache size is C. Defining $\eta \triangleq P'(0)$, we can re-write the gradient ascent algorithm as

$$C_k \leftarrow \max\left\{0, C_k + \gamma_k\left[U_k'(h_k)\frac{\partial h_k}{\partial C_k} - \eta\right]\right\}.$$

In order to update C_k then, the cache manager needs to estimate $\frac{\partial U_k}{\partial C_k} = U_k'(h_k)\frac{\partial h_k}{\partial C_k}$ by gathering hit rate information for each content provider. Instead of computing $U_k'(h_k)$ and $\partial h_k/\partial C_k$ separately, however, we suggest using

$$\frac{\partial U_k}{\partial C_k} \approx \frac{\Delta U_k}{\Delta C_k} = \frac{U_k(h_k^t) - U_k(h_k^{t-1})}{C_k^t - C_k^{t-1}},$$

where the superscripts t and $t-1$ denote the iteration steps. We then use $\frac{\Delta U_k}{\Delta C_k}$ as an estimate of $U_k'(h_k)\frac{\partial h_k}{\partial C_k}$ to determine the value of C_k at the next iteration.

Moreover, since we impose the constraint that $\sum_k C_k = C$, we let η take the mean of the $\frac{\Delta U_k}{\Delta C_k}$ values. The algorithm reaches a stable point once the $\frac{\Delta U_k}{\Delta C_k}$s are equal or very close to each other. Algorithm 6 shows the rules for updating the partition sizes.

4.4.2 Content Providers with Common Content

We now focus on the case where some contents can be served by multiple content providers. Algorithm 3 computes the optimal number of partitions for this case. Let

Algorithm 4 Online algorithm for updating the partition sizes.

1: Start with an initial partitioning $\mathbf{C}^0 \leftarrow (C_1 \cup C_2 \cup \ldots \cup C_K)$.
2: Estimate hit rates for each partition by counting the number of hit requests $\mathbf{h}^0 \leftarrow (h_1, \ldots, h_K)$.
3: Make arbitrary changes to the partition sizes $\mathbf{\Delta}^0$, such that $\sum_k \mathbf{\Delta}_k^0 = 0$, $\mathbf{C}^1 \leftarrow \mathbf{C}^0 + \mathbf{\Delta}^0$.
4: Estimate the hit rates $\mathbf{h}^1$ for the new partition sizes.
5: $t \leftarrow 1$.
6: $\delta_k^t \leftarrow \Big(U_k(h_k^t) - U_k(h_k^{t-1})\Big) / \Big(C_k^t - C_k^{t-1}\Big)$.
7: $\eta^t \leftarrow \Big(\sum_k \delta_k^t\Big)/K$.
8: **if** $\max_k \{\delta_k^t - \eta^t\} > \epsilon$ **then**
9: $\quad \mathbf{\Delta}_k^t = \gamma(\delta_k^t - \eta^t)$.
10: $\quad \mathbf{C}^{t+1} \leftarrow \mathbf{C}^t + \mathbf{\Delta}^t$.
11: $\quad$ Estimate hit rates $\mathbf{h}^{t+1}$.
12: $\quad t \leftarrow t + 1$.
13: $\quad$ **goto** 6.

$\mathcal{P}$ and $\mathbf{C} = (C_1, \ldots, C_{|\mathcal{P}|})$ denote the set of partitions and the vector of partition sizes, respectively. The hit rate for content provider k can be written as

$$h_k(\mathbf{C}) = \sum_{p=1}^{|\mathcal{P}|} \sum_{i \in V_p} \lambda_{ik}(1 - e^{-\lambda_i T_p}),$$

where V_p denotes the set of files requested from partition p, and λ_i denotes the aggregate request rate at partition p for file i.

Similar to (4.15), we consider a penalty function for violating the cache size constraint and rewrite the optimization problem in (4.14) as

$$\begin{aligned} \text{maximize} \quad & \sum_k U_k(h_k(\mathbf{C})) - P(\sum_p C_p - C) \\ \text{such that} \quad & C_p \geq 0, \quad p = 1, \ldots, |\mathcal{P}|. \end{aligned} \tag{4.16}$$

Let $W(\mathbf{C})$ denote the objective function in the above problem. Taking the derivative of W with respect to C_p yields

$$\frac{\partial W}{\partial C_p} = \sum_k U_k'(h_k)\frac{\partial h_k}{\partial C_p} - P'(\sum_p C_p - C).$$

Following a similar argument as in the previous section, we can show that a gradient ascent algorithm converges to the optimal solution.

4.4.2.1 Algorithm Implementation

The implementation of the gradient ascent algorithm in this case is similar to the one in Section 4.4.1.1. However, we need to keep track of hit rates for content provider k from all partitions that store its files. This can be done by counting the number of hit requests for each content provider and each partition through a $K \times |\mathcal{P}|$ matrix, as shown in Algorithm 5. Also, we propose estimating $\partial W/\partial C_p$ as

$$\frac{\partial W}{\partial C_p} \approx \sum_k U_k'(h_k)\frac{\Delta h_{kp}}{\Delta C_p},$$

where Δh_{kp} denotes the change in aggregate hit rate for content provider k from partition p resulted from changing the size of partition p by ΔC_p.

4.5 Evaluation

In this section, we perform numerical simulations, first to understand the efficacy of cache partitioning on the utility observed by content providers, and second to evaluate the performance of our proposed online algorithms.

Algorithm 5 Online algorithm for updating the partition sizes.

1: Compute the number of partitions P using Algorithm 3.
2: Start with an initial partitioning $\mathbf{C}^0 \leftarrow (C_1 \cup C_2 \cup \ldots \cup C_P)$.
3: Estimate hit rates for each provider/partition pair $\mathbf{H}^0 \leftarrow \begin{pmatrix} h_{11} & \ldots & h_{1P} \\ \vdots & \ddots & \vdots \\ h_{K1} & \ldots & h_{KP} \end{pmatrix}$.
4: Make arbitrary changes to the partition sizes $\mathbf{\Delta}^0$, such that $\sum_p \mathbf{\Delta}_p^0 = 0$, $\mathbf{C}^1 \leftarrow \mathbf{C}^0 + \mathbf{\Delta}^0$.
5: Estimate the hit rates $\mathbf{H}^1$ for the new partition sizes.
6: $t \leftarrow 1$.
7: $\delta_p^t \leftarrow \sum_k U_k'(h_k^{t-1})(h_{kp}^t - h_{kp}^{t-1})/(C_p^t - C_p^{t-1})$.
8: $\eta^t \leftarrow \left(\sum_p \delta_p^t\right)/P$.
9: **if** $\max_p \{\delta_p^t - \eta^t\} > \epsilon$ **then**
10: $\quad \mathbf{\Delta}_p^t = \gamma(\delta_p^t - \eta^t)$.
11: $\quad \mathbf{C}^{t+1} \leftarrow \mathbf{C}^t + \mathbf{\Delta}^t$.
12: $\quad$ Estimate hit rates $\mathbf{H}^{t+1}$.
13: $\quad t \leftarrow t + 1$.
14: $\quad$ **goto** 6.

For our base case, we consider a cache with capacity $C = 10^4$. Each partition uses LRU as the cache management policy. We consider two content providers that serve $n_1 = 10^4$ and $n_2 = 2 \times 10^4$ contents. Content popularities for the two providers follow Zipf distributions, *i.e.* $p_i \propto 1/i^z$, with parameters $z_1 = 0.6$ and $z_2 = 0.8$, respectively. Requests for the files from the two content providers arrive as Poisson processes with aggregate rates $\lambda_1 = 15$ and $\lambda_2 = 10$. The utilities of the two content providers are $U_1(h_1) = w_1 \log h_1$ and $U_2(h_2) = h_2$. Unless otherwise specified, we let $w_1 = 1$ so that the two content providers are equally important to the service provider.

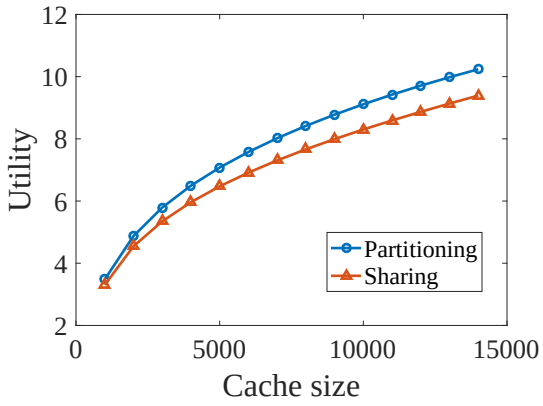


Figure 4.3: Efficacy of cache partitioning when content providers serve distinct files.

We consider two scenarios here. In the first scenario, the two content providers serve completely separate files. In the second scenario, files $S_0 = \{1, 4, 7, \ldots, 10^4\}$, are served by both providers. For each scenario the appropriate optimization formulation is chosen.

4.5.1 Cache Partitioning

To understand the efficacy of cache partitioning, we first look at solutions of optimization problems (4.11) and (4.14). Here, we measure the gain in total utility through partitioning the cache by computing the utility obtained by sharing the cache between the content providers and the utility obtained by partitioning the cache. Figure 4.3 shows the utility gain when content providers serve distinct files. In this example, the aggregate utility increases by 10% from partitioning the cache.

Figure 4.4 shows the utilities for the case when files in $S_0 = \{1, 4, 7, \ldots, 10^4\}$ are served by both content providers. Two cases are considered here: a) request rates for the common content are similar for two content providers. This is done by letting

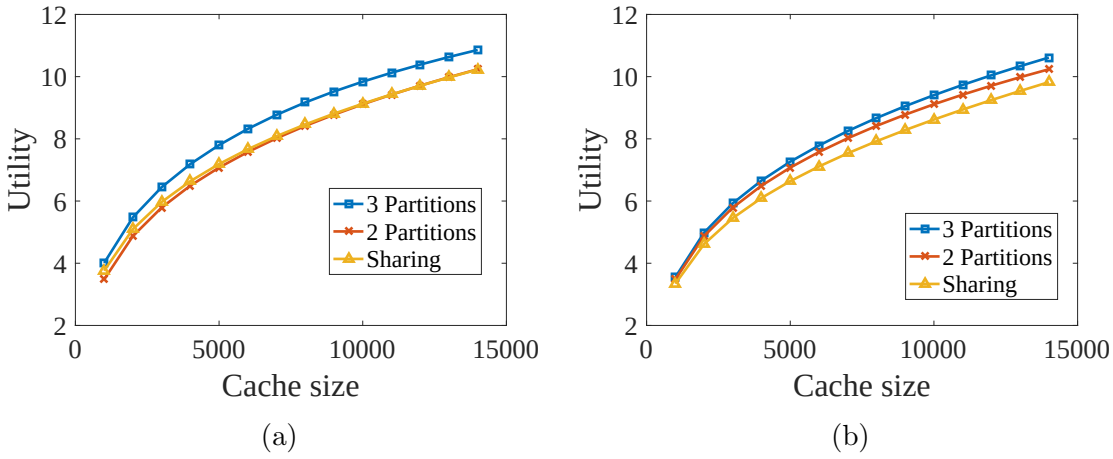


Figure 4.4: Efficacy of cache partitioning when some content is served by both content providers. Request rates for the common contents from the two content providers are set to be (a) similar, and (b) dissimilar.

$p_{k,1} > p_{k,2} > \ldots > p_{k,n_0}$ for both providers. b) Requests rates from the two content providers for the common files are set to be dissimilar. This is done by setting the file popularities for the second CP as $p_{2,1} < p_{2,2} < \ldots < p_{2,n_0}$. In both cases partitioning the cache into three slices shows the best performance.

We next look at the effect of various parameters on cache partitioning, when CPs serve distinct contents and when they serve some common content with similar popularities. We fix the parameters of the second content provider, and study the effect of changing weight parameter w_1 and aggregate request rate λ_1 of the first content provider. We also change the Zipfian file popularity distribution parameter z_1. To study the effect of the utility function, we take it to be the α–fair utility function and vary α for the first content provider, α_1.

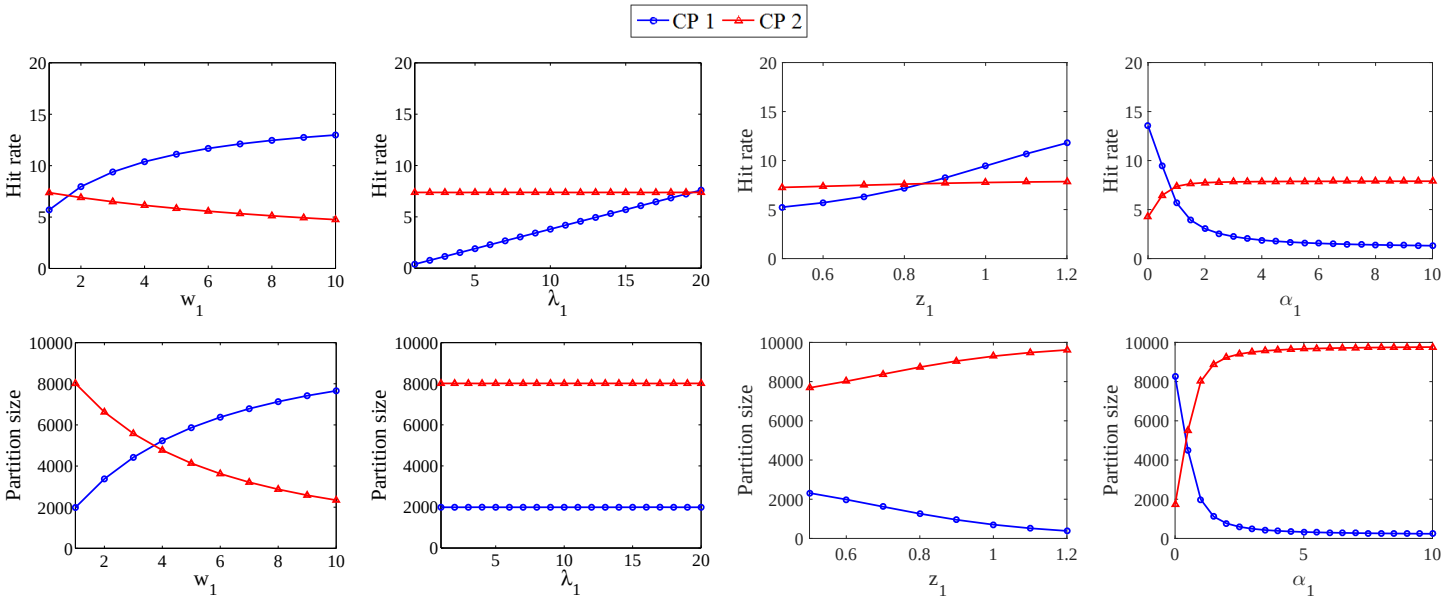


Figure 4.5: Effect of the parameters on hit rates and partition sizes when content providers serve distinct files.

Figure 4.5 shows how hit rates and partition sizes of the two content providers vary as functions of w_1, λ_1, z_1 and α_1. As expected, by increasing the weight w_1, content provider one gets a larger share of the cache, and hence a higher hit rate. Increasing λ_1 has no effect on the partition sizes. This is because the first content provider uses the log utility function, and it is easy to see that the derivative $U_1'(h_1)\partial h_1/\partial C_p$ does not depend on the aggregate rate. In our example, changing the aggregate request rate for the second content provider with $U_2(h_2) = h_2$ results in different partition sizes. As the popularity distribution for contents from the first content provider becomes more skewed, *i.e.* as z_1 increases, the set of popular files decreases in size. Consequently, the dedicated partition size for content provider one decreases as z_1 increases. Increasing α_1 changes the notion of fairness between

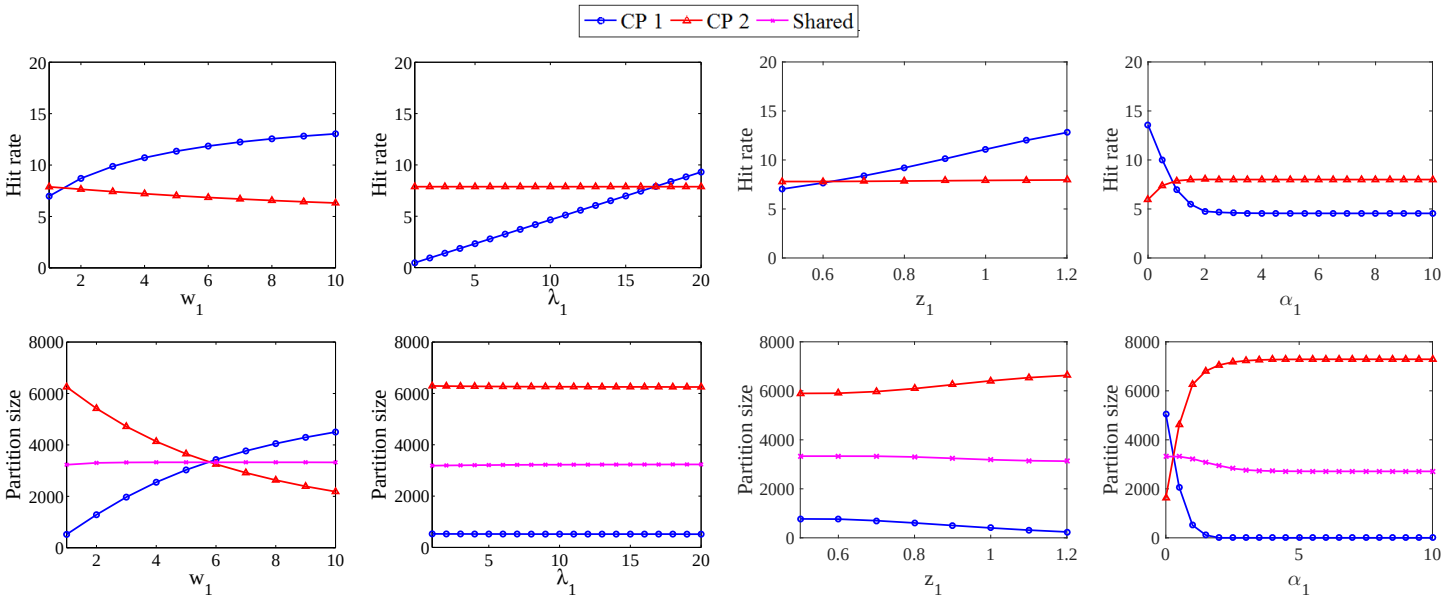


Figure 4.6: Effect of the parameters on hit rates and partition sizes when some content is served by both content providers.

the two content providers in favor of the second content provider, and the size of the partition allocated to the first content provider and its hit rate decreases as α_1 increases.

Figure 4.6 repeats the same experiment for the case when some common content is served by both content providers. The cache is partitioned into three slices in this case, one of them storing common content. Very similar behavior as in Figure 4.5 is observed here.

To understand the fairness notion of the α-fair utility functions, we next use the same utility function for both of the content providers, and vary the value of α to see how the hit rates and partition sizes change. Figure 4.7 shows the effect of α on hit rates and partition sizes for the case when content providers serve distinct

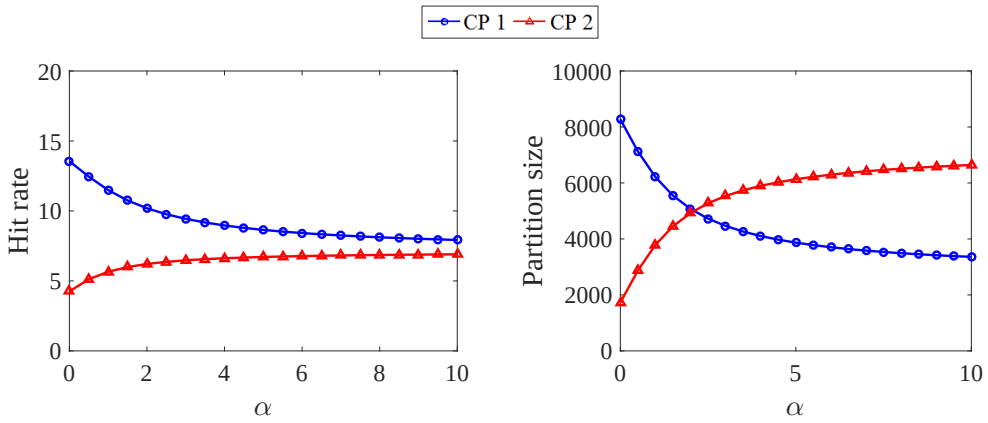


Figure 4.7: α-fair resource allocation for content providers serving distinct content. $U_k(h_k) = h_k^{1-\alpha}/(1-\alpha)$.

files. As α increases, partition sizes change so that hit rates become closer to each other. This is expected since the α-fair utility function realizes the max-min notion of fairness as $\alpha \to \infty$.

Figure 4.8 shows the changes in resource allocation based on the α-fair notion of fairness when common content is served by the content providers.

4.5.2 Online Algorithms

Here, we evaluate the online algorithms presented in Section 4.4 through numerical simulations. Requests are generated according to the parameters presented in the beginning of the section, and the service provider adjusts partition sizes based on the number of hits between iterations. The service provider is assumed to know the utility functions of the content providers. The utility function of the first content provider is fixed to be $U_1(h_1) = \log h_1$. We consider three utility functions for the second content provider, namely $U_2(h_2) = h_2$, $U_2(h_2) = \log h_2$ and $U_2(h_2) = -1/h_2$.

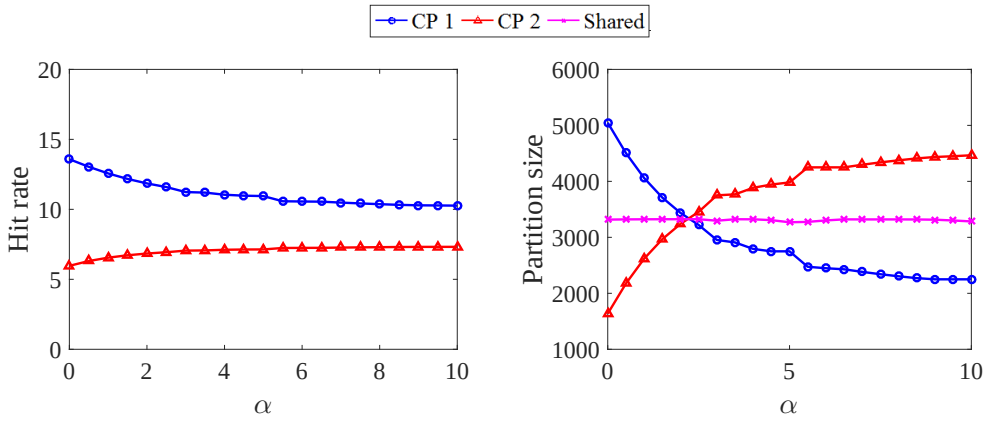
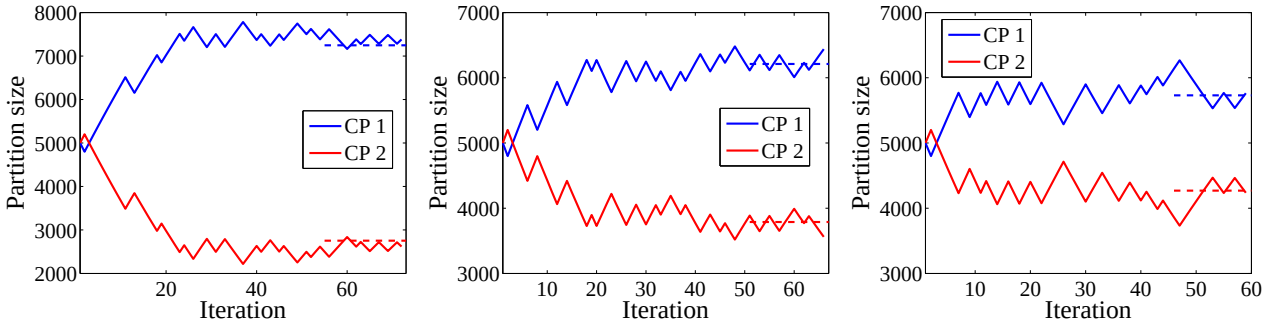


Figure 4.8: α-fair resource allocation when some content is served by both content providers. $U_k(h_k) = h_k^{1-\alpha}/(1-\alpha)$.

We first consider the case where content providers serve distinct files. We initially partition the cache into two equal size slices $C_1 = C_2 = 5000$ and use Algorithm 6 to obtain the optimal partition sizes. Figure 4.9 shows how the partition sizes for the two content providers change at each iteration of the algorithm and that they converge to the optimal values computed from (4.11), marked with dashed lines.

Next, we consider the case where some content is served by both content providers. We first partition the cache into three slices of sizes $C_1 = C_2 = 4000$ and $C_3 = 2000$, where slice 3 serves the common content, and use Algorithm 5 to obtain the optimal partitioning. Figure 4.10 shows the changes in the three partitions as the algorithm converges to a stable point. For each partition the optimal size computed by (4.14) is shown by dashed lines.



(a) $U_2(h_2) = h_2$. (b) $U_2(h_2) = \log h_2$. (c) $U_2(h_2) = -1/h_2$.

Figure 4.9: Convergence of the online algorithm when content providers serve distinct files. $U_1(h_1) = \log h_1$.

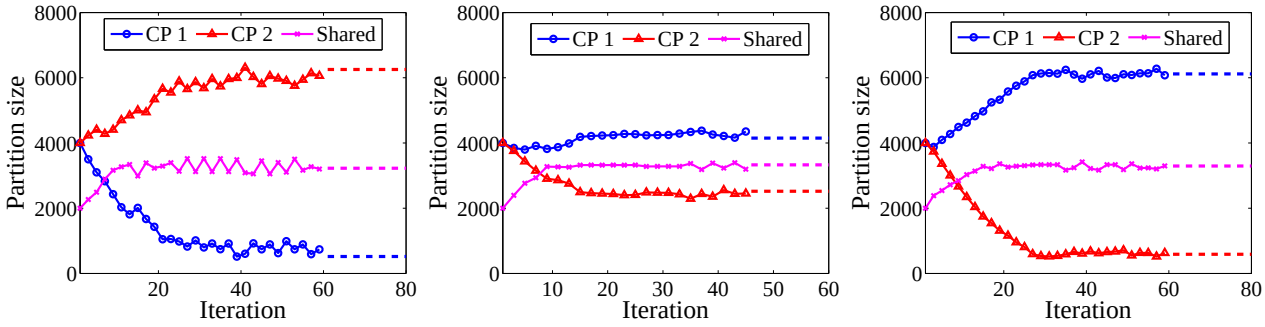
4.6 Discussion

In this section, we explore the implications of utility-driven cache partitioning on monetizing caching service and present some future research directions. We end with a brief discussion of the related work.

Decomposition. The formulation of the problem in Section 4.3 assumes that the utility functions $U_k(\cdot)$ are known to the system. In reality the content providers may not want to reveal their utility functions to the service provider. To handle this case, we decompose optimization problem (4.11) into two simpler problems.

Suppose that cache storage is offered as a service and the service provider charges content providers at a constant rate r for storage space. Hence, a content provider needs to pay an amount of $w_k = rh_k$ to obtain hit rate h_k. The utility maximization problem for content provider k can be written as

$$\text{maximize} \quad U_k(\frac{w_k}{r}) - w_k \tag{4.17}$$



(a) $U_2(h_2) = h_2$. (b) $U_2(h_2) = \log h_2$. (c) $U_2(h_2) = -1/h_2$.

Figure 4.10: Convergence of the online algorithm when some content is served by both content providers. $U_1(h_1) = \log h_1$.

$$\text{such that} \quad w_k \geq 0$$

Now, assuming that the service provider knows the vector $\mathbf{w}$, for a proportionally fair resource allocation, the hit rates should be set according to

$$\begin{aligned} \text{maximize} \quad & \sum_{k=1}^{K} w_k \log\left(h_k\right) \\ \text{such that} \quad & \sum_{p} C_p = C. \end{aligned} \tag{4.18}$$

It was shown in [39] that there always exist vectors $\mathbf{w}$ and $\mathbf{h}$, such that $\mathbf{w}$ solves (4.17) and $\mathbf{h}$ solves (4.18); furthermore, the vector $\mathbf{h}$ is the unique optimal solution.

Cost and Utility Functions. In Section 4.4, we defined a penalty function denoting the cost of using additional storage space. One might also define cost functions based

on the consumed network bandwidth. This is especially interesting in modeling in-network caches with network links that are likely to be congested.

Optimization problems (4.11) and (4.14) use utility functions defined as functions of the hit rate. It is reasonable to define utility as a function of the hit *probability*. Whether this significantly changes the problem, *e.g.* in the notion of fairness, is a question that requires further investigation. One argument in support of utilities as functions of hit rates is that a service provider might prefer pricing based on request rate rather than the cache occupancy. Moreover, in designing hierarchical caches a service provider's objective could be to minimize the internal bandwidth cost. This can be achieved by defining the utility functions as $U_k = -P_k(m_k)$ where $P_k(m_k)$ denotes the cost associated with miss rate m_k for content provider k.

4.7 Related Work

Internet cache management issues have been extensively studied in the context of web caching (e.g., see [12, 23] and references therein). In this context, biased replacement policies for different kinds of content classes [41] and differentiated caching services via cache partitioning [43, 45] haven been proposed and studied. None of these studies explicitly deal with the cache allocation problem among multiple content providers. The emergence of content-oriented networking has renewed research interests in cache management issues for content delivery, especially in the design of cache replacement policies for the content-oriented architecture [8, 10, 47, 83]. The cache allocation problem among content providers has attracted relatively little attention. Perhaps most closely related to our work is the study in [33] where a

game-theoretic cache allocation approach is developed. This approach requires the content providers to report the true demands from their content access. In contrast, we develop a general utility maximization framework for studying the cache allocation problem. Since its first proposal by Kelly *et al.* [39], the network utility maximization framework has been applied to a variety of networking problems from stability analysis of queues [20] to the study of fairness in network resource allocation [51]. A utility maximization framework for caching policies was developed in [17] to provide differentiated services to content. This framework was adopted by [13] to study the cache resource allocation problem in an informal and heuristic manner. We make precise statements to support the observations in [13]. In this respect, our contribution lies in establishing the key properties of CP utilities as a function of cache sizes and in postulating cache partitioning as a basic principle for cache sharing among content providers. Furthermore, we develop decentralized algorithms to implement utility-driven cache partitioning, and prove that they converge to the optimal solution.

4.8 Conclusion

We proposed utility-based partitioning of a cache among content providers, and formulated it as an optimization problem with constraints on the service providers cache storage size. Utility-driven cache partitioning provides a general framework for managing a cache with considerations of fairness among different content providers, and has implications on market economy for service providers and content distributors. We considered two scenarios where 1) content providers served disjoint sets

of files, or 2) some content was served by multiple content providers. We developed decentralized algorithms for each scenario to implement utility-driven cache partitioning in an online fashion. These algorithms adapt to changes in request rates of content providers by dynamically adjusting the partition sizes. We theoretically proved that these algorithms are globally stable and converge to the optimal solution, and through numerical evaluations illustrated their efficiency.

CHAPTER 5

MODELING AND OPTIMIZATION OF PENDING INTEREST TABLE TIMEOUT

5.1 Introdcution

Pending Interest Table (PIT) is one of the core components of the Named Data Networking (NDN) architecture. It performs Interest aggregation by keeping track of currently unsatisfied Interest packets. PITs provide advantages such as enabling communicating without the knowledge of source and destination, reducing bandwidth usage, and better security, to name a few. To prevent PIT size bloat and ensure efficient I/O operations at line speed, PIT entries are purged after a *timeout* period. A timeout also provides protection against flooding attacks [79].

An efficient design of the PIT is therefore an important factor in the performance of NDN at wire speed, and an accurate assessment of the PIT size is key to achieving this. Despite many experimental and numerical evaluations [15, 75, 77, 81], not much attention has been paid to analytic modeling of Pending Interest Tables. In this chapter, we analyze an LRU cache with a Pending Interest Table, to compute the cache hit probability, Interest aggregation probability, and the size of the Pending Interest Table as functions of the PIT timeout. Exploiting our analysis, we then study the problem of optimizing the PIT timeout.

In this chapter, we analyze an LRU cache with a Pending Interest Table. Our analysis is generic and can be easily modified to model other policies such as FIFO and Random. We also formulate an optimization problem for computing the optimal PIT timeout.

5.2 Model Description

In this section, we introduce our model for a cache with requests arriving for a set of K unique files of unit size. Throughout this chapter, we will use the terms content and file interchangeably. We assume that each file resides permanently at a content custodian that the cache can access.

Once a request arrives for a file that is in the cache, the request is served instantly. However, if the content is not found in the cache, the request is forwarded to the content custodian. We consider the case where downloading content k from the custodian incurs a non-zero delay denoted by random variable D_k. With a misuse of notation, we will use $D_k(\cdot)$ to denote the CDF of D_k. We allow for different download delay distributions for different files.

It is assumed that the cache employs a Pending Interest Table (PIT) to aggregate similar requests arriving while the content is being downloaded to the cache. With the arrival of the first request to a file that is not in cache, an entry is created in the PIT. All successive requests for the same content during time D_k are aggregated at the PIT and not forwarded to the custodian. A PIT entry is deleted after a *timeout* period τ, or once the content is downloaded into the cache.

Requests for file k arrive according to a renewal process with inter-arrival distribution $F_k(\cdot)$, and the cache employs the Least Recently Used (LRU) policy for content evictions. We rely on the *characteristic time* approximation to model the LRU policy as a Time-To-Live (TTL) cache in which content eviction occurs upon the expiration of a timer. By relying on the TTL notion, our analysis can be simply adapted to policies other than LRU, as TTL caches admit a general approach to their analysis and are known to provide accurate tools for modeling replacement-based caches such as LRU, FIFO, *etc.*

5.2.1 Renewal Arrivals

Let X_i be the time interval between the $(i-1)$st and the i-th requests for a given file. Inter-request times are assumed to be i.i.d and have distribution function $F(x) = \mathbb{P}(X_i \leq x)$. Let $\lambda = 1/\mathbb{E}[X_i]$ denote the arrival rate for the given file. Without loss of generality, we assume that a request for a file that is not in the cache occurs at $t = 0$, *i.e.* $X_0 = 0$. Let M_t denote the number of requests for the given file in the interval $(0, t]$. M_t is called the *renewal (counting) process*. Note that the request at $t = 0$ is excluded, *i.e.* $M_0 = 0$. Also, let

$$\tau_n = X_1 + X_2 + \ldots + X_n, \quad n \geq 1 \quad (\tau_0 = 0),$$

denote the time until the arrival of the nth request. We have

$$\mathbb{P}(\tau_n \leq t) = \mathbb{P}(X_1 + X_2 + \ldots + X_n \leq t) = F^{(n)}(t),$$

where $F^{(n)}(t)$ denotes the n-fold convolution of the distribution function $F(x)$ with itself.

The expected number of renewals in time interval $(0, t]$ is the *renewal function* $m(t) = \mathbb{E}[M_t]$, and can be expressed as

$$m(t) = \sum_{n=1}^{\infty} F^{(n)}(t), \quad t \geq 0.$$

Poisson Arrivals

A Poisson process is a renewal process with parameter λ whose inter-arrival times have the exponential distribution $F(x) = 1 - e^{-\lambda x}$. For a Poisson process, the renewal function simplifies to $m(t) = \lambda t$. We will use Poisson arrivals in studying the LRU cache in order to obtain closed form expressions for our metrics of interest.

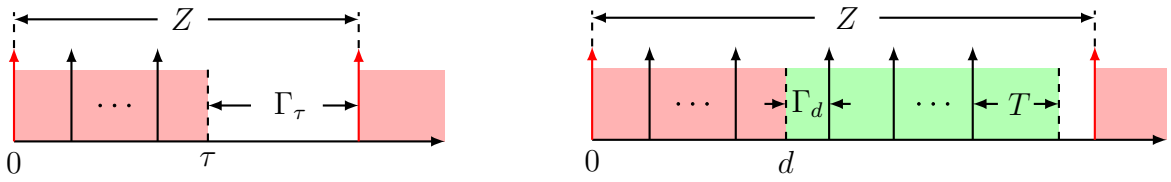
5.2.2 Cache Characteristic Time

Our analysis in this chapter is based on the notion of cache characteristic time explained in Appendix E. According to this notion, for an LRU cache of size C, the probability that a request for file k results in a hit can be approximated by

$$h_k = 1 - e^{-\lambda_k T},$$

where λ_k is the request rate for file k, and T is a constant denoting the characteristic time of the cache and is computed as the unique solution to the equation

$$\sum_{k=1}^{K} (1 - e^{-\lambda_k T}) = C,$$



(a) $d \geq \tau$ (b) $d < \tau$

Figure 5.1: An entry is created in PIT for the file at time $t = 0$. Content enters the cache at time $t = D$, and PIT entry is deleted. (a) The content will be evicted from cache at time $t_E = D + T_0$ if $\Gamma_D > T_0$. (b) The TTL will be reset if $\Gamma_D \leq T_0$, and the content will continue to stay in the cache as long as $Y_n \leq T_n$. Here, $t_E = \tau_{N+M_D} + T_N$ denotes the time that the content is evicted from cache.

where C is the capacity of the LRU cache.

5.2.3 Metrics of Interest

In our analysis, we are mainly interested in computing the cache hit probability and request aggregation probability at PIT. We compute the cache hit probability for each file as well as the probability that an entry exists in the PIT for a file, and capture the distribution of the PIT size as a function of the timeout value τ. We also compute the rate at which requests for different files are forwarded to the content custodian.

5.3 LRU with Poisson Arrivals

In this section, we model an LRU cache of capacity C with requests arriving to the cache according to renewal processes. Figure 5.1 illustrates the cache dynamics with requests arriving for a given content. With renewal processes, inter-request times are i.i.d and follow distribution F_k. In Figure 5.1, Γ_t denotes the period between time

t and the arrival of the next request, and is known as the *excess life* of the renewal process at time t.

With the arrival of the first missed request an entry is created for the file in the PIT. The PIT timeout value is assumed to be τ. After the PIT entry is created, consecutive requests for the file are aggregated until either the content is downloaded to the cache at time d, *i.e.* when $d < \tau$, or the PIT entry expires, *i.e.* when $\tau \leq d$. Content is stored at the cache only if $d < \tau$.

We model the LRU policy as a TTL cache, and hence when a content enters the cache, its TTL is set to T. If the next request for the file arrives after the TTL expires, *i.e.* $\Gamma_d > T$, the content will be evicted from cache at time $t = d + T$. However, if a request arrives before the TTL expires, the TTL will be reset to T. The file will remain in the cache as long as successive requests arrive no later than the TTL expires.

The process explained above can be considered as a renewal cycle of length Z that begins with a request for a content that is neither in cache nor PIT, and ends with the next request for which the content cannot be found in cache or PIT. These requests are marked red in Figure 5.1, and are the requests that are forwarded to content custodian. Requests that arrive while there is an entry for the content in the PIT are aggregated. Once the content is cached, arriving requests generate cache hits. Note that the cycles explained above are statistically the same, and without loss of generality, we consider a cycle starting at $t = 0$.

Let A_k denote the number of aggregated requests in a cycle. With renewal processes, the expected number of arrivals within a period Δt equals $m_k(\Delta t)$. Therefore,

we have

$$\mathbb{E}[A_k \mid D_k = d] = m_k(\min(d, \tau)) = m_k(\tau)\mathbb{1}_{\{d \geq \tau\}} + m_k(d)\mathbb{1}_{\{d < \tau\}}.$$

The expected number of aggregated requests then is

$$\mathbb{E}[A_k] = \int_0^\infty A_k \mathrm{d}D_k(d) = m_k(\tau)\Big(1 - D_k(\tau)\Big) + \int_0^\tau m_k(d)\mathrm{d}D_k(d). \tag{5.1}$$

From Figure 5.1 we see that that if $d \geq \tau$, the cycle Z ends at $t = \tau + \Gamma_\tau$. Also, if $\Gamma_d > T$, the cycle Z ends with no cache hits. With $\Gamma_d \leq T$ we observe the first cache hit. Therefore, assuming $d < \tau$, the first cache hit occurs with probability $\mathbb{P}(\Gamma_d \leq T)$. Recall that Γ_t is known as the *excess life* of a renewal process at time t, for which the distribution function is expressed as (see Eq. (6.1) of Chapter 5 in [38])

$$\mathbb{P}(\Gamma_t \leq \gamma) = F_k(t + \gamma) - \int_0^t \Big(1 - F_k(t + \gamma - x)\Big)\mathrm{d}m_k(x).$$

The second hit occurs if the next request inter-arrival time is no more than T. Assuming that exactly N_k requests result in cache hit, we must have $N_k - 1$ inter-arrival times that are smaller than or equal to T, followed by an inter-arrival time that is greater than T. Therefore, given that $d < \tau$, the number of cache hits is characterized by a geometric distribution. We have

$$\mathbb{P}(N_k = n \mid d < \tau) = \begin{cases} P_\Gamma & n = 0, \\ (1 - P_\Gamma)(1 - P_T)^{n-1} P_T & n \geq 1, \end{cases}$$

where

$$P_\Gamma = 1 - \mathbb{P}(\Gamma_d \leq T),$$

is the probability of no cache hits, and

$$P_T = 1 - F_k(T),$$

denotes the probability that the request inter-arrival time is larger than T. This yields

$$\mathbb{E}[N_k \mid d < \tau] = (1 - P_\Gamma)/P_T.$$

The expected number of cache hits per cycle then is

$$\mathbb{E}[N_k] = \frac{\int_0^\tau \mathbb{P}(\Gamma_d \leq T)\mathrm{d}D_k(d)}{1 - F_k(T)}. \tag{5.2}$$

5.3.1 Poisson Arrivals and Exponentially Distributed Delays

With a Poisson process, the renewal function for file k is $m_k(t) = \lambda_k t$; using (5.1) allows us to write the expected number of aggregated requests in a cycle as

$$E[A_k] = \lambda_k \tau (1 - D_k(\tau)) + \lambda_k \int_0^\tau d\mathrm{d}D_k(d).$$

Assuming exponentially distributed download delays, we obtain

$$\begin{aligned}\mathbb{E}[A_k] &= \lambda_k \tau e^{-\tau/\mathbb{E}[D_k]} + \lambda_k \mathbb{E}[D_k]\Big(1 - (1 + \frac{\tau}{\mathbb{E}[D_k]})e^{-\tau/\mathbb{E}[D_k]}\Big) \\ &= \Big(1 - e^{-\tau/\mathbb{E}[D_k]}\Big)\lambda_k \mathbb{E}[D_k].\end{aligned}$$

With Poisson arrivals, request inter-arrival times have the exponential distribution $F_k(t) = 1 - e^{\lambda_k t}$. Moreover, due to the memoryless property of exponential distribution, Γ_t also follows an exponential distribution with rate λ_k, *i.e.*,

$$\mathbb{P}(\Gamma_t \leq \gamma) = 1 - e^{-\lambda_k \gamma}.$$

Based on (5.2), the expected number of cache hits per cycle, hence, equals

$$\mathbb{E}[N_k] = D_k(\tau)\Big(e^{\lambda_k T} - 1\Big).$$

Assuming that the download delays follow the exponential distribution, we obtain

$$\mathbb{E}[N_k] = \Big(1 - e^{-\tau/\mathbb{E}[D_k]}\Big)\Big(e^{\lambda_k T} - 1\Big).$$

Cache Hit Probability: We can now write the cache hit probability for content k as the expected number of hit requests divided by the expected total number of requests,

$$h_k = \frac{\mathbb{E}[N_k]}{1 + \mathbb{E}[A_k] + \mathbb{E}[N_k]} = \frac{(1 - e^{-\tau/\mathbb{E}[D_k]})(e^{\lambda_k T} - 1)}{1 + (1 - e^{-\tau/\mathbb{E}[D_k]})(\lambda_k \mathbb{E}[D_k] + e^{\lambda_k T} - 1)}. \tag{5.3}$$

Note that assuming download delay is zero, *i.e.* $\mathbb{E}[D_k] = 0$, we obtain $h_k = 1 - e^{-\lambda_k T}$ which is the expression given by Che *et al.* [12] for the hit probability of an LRU cache with Poisson arrivals. Also, as $\tau \to \infty$ we get

$$h_k = \frac{e^{\lambda_k T} - 1}{\lambda_k \mathbb{E}[D_k] + e^{\lambda_k T}},$$

which is the expression obtained in [16] for cache hit probability under the assumption that PIT entries do not timeout.

Note that the value of the characteristic time T is obtained by solving the fixed-point equation

$$\sum_{k=1}^{K} h_k = C.$$

PIT Aggregation Probability: We can write the probability of request aggregation in PIT for file k as

$$a_k = \frac{\mathbb{E}[A_k]}{1 + \mathbb{E}[A_k] + \mathbb{E}[N_k]} = \frac{\lambda_k \mathbb{E}[D_k](1 - e^{-\tau/\mathbb{E}[D_k]})}{1 + (1 - e^{-\tau/\mathbb{E}[D_k]})(\lambda_k \mathbb{E}[D_k] + e^{\lambda_k T} - 1)}. \tag{5.4}$$

Note that a_k also denotes the probability of having an entry in the PIT for content k.

To compute statistics regarding the PIT size we proceed as follows. We define Bernoulli random variables $\mathcal{A}_k, \forall k$ to indicate whether file k resides in PIT, *i.e.* $\mathbb{P}(\mathcal{A}_k = 1) = 1 - \mathbb{P}(\mathcal{A}_k = 0) = a_k$. We then define $\mathcal{S} = \sum_k \mathcal{A}_k$ to denote the number of entries in the PIT, *i.e.* the PIT size. The random variable $\mathcal{S}$ follows a Poisson Binomial distribution with mean $\mu_{\mathcal{S}} = \sum_k a_k$, and variance $\sigma^2_{\mathcal{S}} = \sum_k a_k(1 - a_k)$. It is well known that a Poisson Binomial distribution can be approximated by a Gaussian distribution [50]. Therefore, we approximate the PIT size distribution with a Gaussian distribution with mean $\mu_{\mathcal{S}}$, and variance $\sigma^2_{\mathcal{S}}$. In Section 5.6, we show that a Gaussian distribution accurately approximates the PIT size distribution.

Request Forwarding Probability: In each cycle that was defined earlier in the section, one request is forwarded to the custodian. Therefore, the probability that a request gets forwarded to the custodian is

$$f_k = \frac{1}{1 + \mathbb{E}[A_k] + \mathbb{E}[N_k]} = \frac{1}{1 + (1 - e^{-\tau/\mathbb{E}[D_k]})(\lambda_k \mathbb{E}[D_k] + e^{\lambda_k T} - 1)}. \tag{5.5}$$

The average rate at which requests get forwarded to the custodian then can be obtained by

$$f = \sum_k \lambda_k f_k.$$

In the following two sections, we exploit the results developed here to *i)* optimize the PIT timeout τ when PIT size is fixed a priori, and *ii)* formulate the memory-bandwidth tradeoff as a convex optimization problem.

5.4 Constrained PIT Size

In this section, we focus on the problem of setting τ so that the probability of overflowing a PIT of size Θ is less than or equal to δ. This corresponds to the following PIT timeout optimization problem:

$$\begin{aligned} &\text{maximize} \quad \tau \\ &\text{such that} \quad \mathbb{P}(\mathcal{S}(\tau) \geq \Theta) \leq \delta, \end{aligned} \tag{5.6}$$

where $\mathcal{S}(\tau)$ is a random variable denoting the PIT size as a function of the timeout value τ. As discussed in the previous section, the PIT size can be approximately modeled as a Gaussian distribution. Our approach is to set the timeout value such that the expected PIT size $\mu(\tau)$ equals $(1-\epsilon)\Theta$ where $\epsilon > 0$ is chosen in a way that the probability of PIT size being larger that Θ is no more than δ, *i.e.* $\mathbb{P}(\mathcal{S}(\tau) \geq \Theta) \leq \delta$.

With the assumption of a Gaussian distribution with mean $\mu(\tau) = (1-\epsilon)\Theta$ and variance σ^2, we have

$$\mathbb{P}(\mathcal{S}(\tau) \geq \Theta) = {}^1\!/_2 - {}^1\!/_2 \operatorname{erf}\left(\frac{\Theta - (1-\epsilon)\Theta}{\sigma\sqrt{2}}\right) = {}^1\!/_2 - {}^1\!/_2 \operatorname{erf}\left(\frac{\epsilon\Theta}{\sigma\sqrt{2}}\right),$$

where $\operatorname{erf}(\cdot)$ denotes the error function defined as

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} \mathrm{d}t.$$

In order to guarantee $\mathbb{P}(\mathcal{S}(\tau) \geq \Theta) \leq \delta$, we should have

$$\epsilon \geq \frac{\sigma\sqrt{2}}{\Theta} \operatorname{erf}^{-1}(1-2\delta). \tag{5.7}$$

With ϵ satisfying the above inequality, the timeout value can be computed according to the equation

$$\mu(\tau) = \sum_k a_k(\tau),$$

where $a_k(\tau)$ denote request aggregation probability for file k, and can be computed based on the expression for a_k in (5.4). Therefore, we can re-write the optimization problem (5.6) as follows

$$\begin{aligned} &\text{maximize} \quad \tau \\ &\text{such that} \quad \mu(\tau) \leq (1-\epsilon)\Theta. \end{aligned} \tag{5.8}$$

Note that ϵ and σ^2 are also functions of the PIT timeout.

The constraint in the above optimization problem does not define a convex region. However, we observe that PIT size $\mu(\tau)$ is an increasing function of τ, and rely on our intuition that the smallest τ violating the constraint $\mu(\tau) < (1-\epsilon)\Theta$ must be the optimal solution. Therefore, we assume that problem (5.8) has a unique solution that can be found through a linear search.

5.4.1 Online Algorithm

Assuming the optimization problem (5.8) can be solved to find the optimal solution, due to the system dynamics, *e.g.* changes in request arrival rate, it is still not practical to select a timeout value that guarantees performance requirements. Therefore, we need algorithms to adapt τ to changes in the system.

Here, we rely on our intuition to assume that $\mu(\tau) - (1-\epsilon)\Theta$ is an increasing function of τ, and hence the solution of optimization problem (5.8) is the root of the equation $\mu(\tau) - (1-\epsilon)\Theta = 0$. Therefore, we can use Newton's method to find the optimal value of τ. Newton's method defines an iterative algorithm starting with an initial guess and updating the parameter at every step. Here, we use a modified Newton's method so that we do not have to compute a derivative. Since we assume $\mu(\tau) - (1-\epsilon)\Theta$ is an increasing function of τ, *i.e.* its derivative with respect to τ is non-negative, τ should be updated as

$$\tau_{t+1} \leftarrow \tau_t - \gamma\Big(\mu(\tau_t) - (1-\epsilon_t)\Theta\Big)$$

Here, $\gamma > 0$ controls the step size at each iteration.

In order to compute ϵ_t at each iteration using (5.7), we need to compute the variance of the PIT size. To do so, we propose keeping records of the last n PIT size observations $S_i, i = 1, \ldots, n$. An unbiased estimate of the variance can then be obtained by

$$\tilde{\sigma}_t^2 = \frac{1}{n-1} \sum_{i=1}^{n} (S_i - \bar{S}_t)^2,$$

where $\bar{S}_t = \frac{1}{n} \sum S_i$ denotes the sample mean at time t. We can further use the sample mean $\bar{S}_t$ as an estimate of $\mu(\tau_t)$. Hence, we can use the following iterative algorithm for updating the timeout at each step

$$\tau_{t+1} \leftarrow \tau_t - \gamma\Big(\bar{S}_t - (1 - \epsilon_t)\Theta\Big). \tag{5.9}$$

We show in Section 5.6 that the above algorithm is effective in finding the optimal timeout. However, it requires a lot of book-keeping for estimating the mean and variance of the PIT size. Therefore, we propose an update rule for τ that does not require such an estimation. With an incoming message, let $\mathbb{1}_{\{\mathcal{S} \geq \Theta\}}$ denote whether the current PIT size is larger than the threshold Θ. Now, let λ denote the overall request rate, and consider the following differential equation

$$\dot{\tau} = \gamma\delta(1 - \mathbb{1}_{\{\mathcal{S} \geq \Theta\}})\lambda - \gamma(1 - \delta)\mathbb{1}_{\{\mathcal{S} \geq \Theta\}}\lambda. \tag{5.10}$$

Note that τ increases when the current PIT size is smaller than the threshold Θ and decreases otherwise. Taking the expectation of both sides of the above differential equation, and noting that $\mathbb{E}\Big[\mathbb{1}_{\{\mathcal{S} \geq \theta\}}\Big] = \mathbb{P}(\mathcal{S} \geq \Theta)$, we obtain

$$\mathbb{E}[\dot{\tau}] = \gamma\delta(1 - \mathbb{P}(\mathcal{S} \geq \Theta))\lambda - \gamma(1 - \delta)\mathbb{P}(\mathcal{S} \geq \Theta)\lambda.$$

At steady state, $\mathbb{E}[\dot{\tau}] = 0$, and we obtain $\mathbb{P}(\mathcal{S} \geq \Theta) = \delta$. In Section 5.6, we show that this simple algorithm is indeed effective in finding the optimal solution for PIT timeout.

5.5 Memory-Bandwidth Tradeoff

A major benefit of caching to ISPs is that it reduces inter-domain traffic and hence reduces bandwidth costs. However, there is a cost associated with the memory consumed by caches. In an ICN, request aggregation by PIT reduces outgoing traffic even further while incurring additional costs for PIT management. In this section, we formulate the memory-bandwidth tradeoff as an optimization problem with the objective of minimizing the aggregate bandwidth usage cost and cache and PIT storage costs.

Here, we implement an LRU cache as a Time-To-Live (TTL) cache based on the characteristic time approximation discussed in Section 5.2. By doing so, cache size can be controlled by the characteristic time T. As T increases, cache size grows larger, producing more cache hits and hence reducing the traffic forwarded upstream. Similarly, PIT size can be controlled using the timeout value τ. PIT size grows as τ is increased, which results in more request aggregation, and less request forwarding.

Although cache size is mainly a function of the characteristic time T, it is clear from (5.3) that it is also affected by PIT timeout value τ. Similarly, PIT size is affected by the value of characteristic time T. In the remainder, we will use $C(\tau, T)$

and $S(\tau, T)$ to denote the expected cache and PIT sizes, respectively. Also, we define bandwidth consumption in terms of the expected request forwarding rate, *i.e.*

$$B(\tau, T) = \sum_k \lambda_k f_k(\tau, T),$$

where $f_k(\tau, T)$ is the request forwarding probability for file k with the expression given in (5.5).

We quantify the cost of bandwidth by $L_b(B)$ where B is the request forwarding rate. Similarly, we quantify cache and PIT storage costs by convex functions $L_c(C)$ and $L_s(S)$ where C and S denote the size of cache and PIT, respectively. Therefore, we solve for the trade-off between memory and bandwidth costs through the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & L_b\Big(B(\tau, T)\Big) + L_c\Big(C(\tau, T)\Big) + L_s\Big(S(\tau, T)\Big) \\ \text{such that} \quad & \tau, T \geq 0. \end{aligned} \tag{5.11}$$

Remark 1: Problem (5.11) is *not* a convex optimization problem, and finding the optimal solution is *hard* in general. It is easy to see that two important variants of the problem with *i)* fixed cache size, and *ii)* fixed PIT size are also non-convex.

Remark 2: For a triplet (B^*, C^*, S^*) that minimizes the total cost in (5.11), in general there are multiple (τ, T) pairs that achieve the same objective. Our many numerical experiments also suggest that finding a (τ, T) pair that is a local minima results in considerable cost savings, and is comparable in performance to a global minimum that is computed via brute-force. Based on this observation, we develop

Algorithm 6 Online algorithm for minimizing total cost.

1: Start with an initial guess $\mathbf{x}_0 \leftarrow (\tau_0, T_0)$.
2: Use $\mathbf{x}_0$ and measure system cost $W(\mathbf{x}_0)$.
3: **for** $t = 1, 2, \ldots$ **do**
4: Select a unit vector $\mathbf{u}$ uniformly at random.
5: Let $\mathbf{y}_t \leftarrow \mathbf{x}_t + \delta\mathbf{u}$.
6: Use $\mathbf{y}_t$ and measure system cost $W(\mathbf{y}_t)$.
7: Estimate gradient as $g(\mathbf{x}_t) = \frac{W(\mathbf{y}_t) - W(\mathbf{x}_t)}{\delta}\mathbf{u}$.
8: $\mathbf{x}_{t+1} \leftarrow \mathbf{x}_t - \gamma_t g(\mathbf{x}_t)$.

an algorithm described in the remainder of the section to find a *local* minimizer to (5.11).

5.5.1 Online Algorithm

Although optimization problem (5.11) can not be solved in polynomial time to find the optimal timeout and characteristic time values, a local minimum can be computed. For a system operating with some values set for parameters (τ, T), finding a local minimum around (τ, T) can considerably reduce system costs. In this section, we develop an algorithm to achieve this.

Let $W(\tau, T)$ denote the objective function in (5.11). A natural decentralized approach in finding a local minimum for $W(\cdot)$ is to use gradient descent. The basic idea behind gradient descent is to move the variables τ and T in the opposite direction of the gradient

$$\nabla W = (\frac{\partial W}{\partial \tau}, \frac{\partial W}{\partial T}).$$

By gradient descent, the parameters should be updated according to

$$\tau \leftarrow \max\left\{0, \tau - \gamma\frac{\partial W}{\partial \tau}\right\},$$

and

$$T \leftarrow \max\left\{0, T - \gamma \frac{\partial W}{\partial T}\right\},$$

where $\gamma > 0$ is the step-size parameter.

In order to implement gradient descent based on the above update rules, one needs to compute the partial derivatives $\partial W/\partial \tau$ and $\partial W/\partial T$. The expressions for these partial derivatives are unwieldy and cumbersome to derive. Instead, we propose an iterative algorithm that uses an estimated gradient at each iteration. We use

$$\frac{\partial W}{\partial \tau} \approx \frac{\Delta W}{\Delta \tau} \qquad \text{and} \qquad \frac{\partial W}{\partial T} \approx \frac{\Delta W}{\Delta T},$$

where ΔW denotes the changes in cost value, while $\Delta \tau$ and ΔT denote the changes in PIT timeout and cache characteristic time. Our proposed algorithm is based on the randomized gradient descent algorithm described in [56], and is summarized in Algorithm 6. The basic idea is to select a point uniformly at random around the current position and move there if cost is reduced. Note that for an n-dimensional problem, here $n = 2$, this method is only $O(n)$ slower than the standard gradient method in converging to a local optima [68]. In the next section, we will show that Algorithm 6 is effective in practice.

Remark: Algorithm 6 is most suitable when there is an estimate for optimal values of τ and T for example from an approximate solution to (5.11). Starting with a (τ, T) pair that is close to the optimal solution, one can find the optimal solution through Algorithm 6.

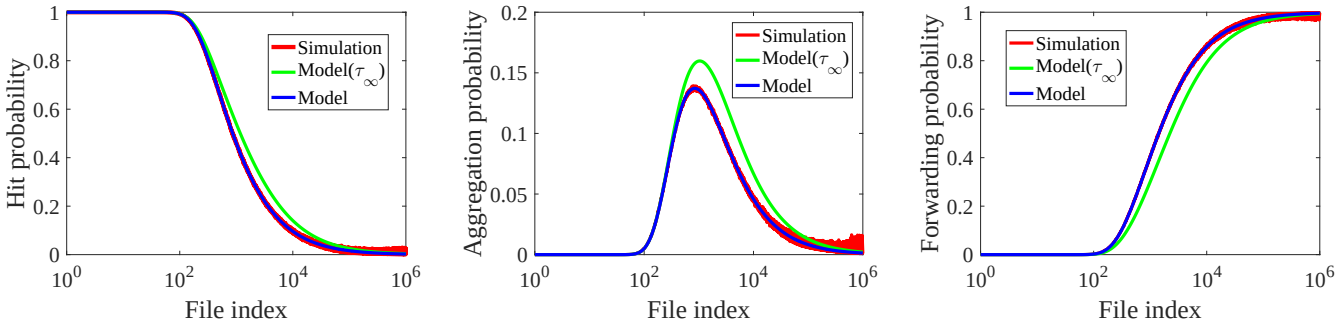


Figure 5.2: Per file probability of (a) cache hit, h_k, (b) PIT aggregation, a_k, and (c) request forwarding, f_k. The curve with label 'Model(τ_∞)' shows the probability values when $\tau = \infty$, *i.e.* ignoring the effect of PIT timeout.

5.6 Performance Evaluation

In this section, we first evaluate the accuracy of the model developed in Section 5.3, and then evaluate the performance of the online algorithms described in Sections 5.4.1 and 5.5.1.

In the following evaluations, we simulate an LRU cache of size $C = 10,000$, where requests arrive for $K = 10^6$ files. File popularities follow a Zipf distribution with parameter $\alpha = 0.8$, *i.e.* $\lambda_k \propto 1/k^\alpha$. Requests for files arrive according to Poisson processes, and the aggregate request rate is assumed to be $\lambda = 10^5$ requests per second. We also assume that file download delays follow an exponential distribution with mean 100ms for all files. Also, PIT timeout is set to $\tau = 100$ms.

5.6.1 Model Evaluation

Figures 5.2(a)-(c) show the cache hit probability h_k, PIT aggregation probability a_k, and request forwarding probability f_k for individual files. It is clear that our

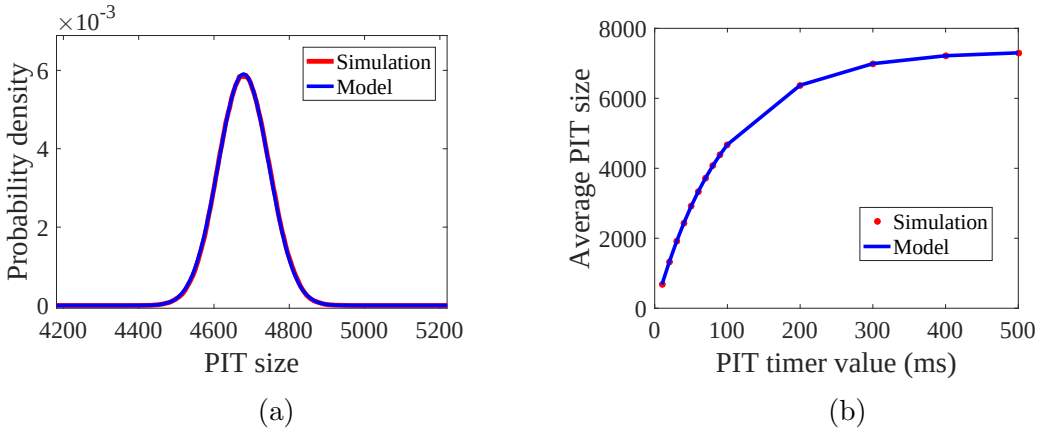


Figure 5.3: (a) Approximating PIT size distribution with a Gaussian. (b) Average PIT size increases with PIT timeout.

model accurately estimates these probabilities and agrees with simulation results. The green curves in Figures 5.2(a)-(c) correspond to the model in [16] which ignores the effect of PIT timeout in the analysis. The effect of PIT timeout is especially clear on request aggregation and forwarding probabilities.

In Section 5.3, we advocated using the Gaussian distribution to approximate the size of the Pending Interest Table. Figure 5.3(a) shows the PIT size distribution computed from simulations, as well as a Gaussian distributions with moments computed based on the discussion in Section 5.3. It is clear that the Gaussian distribution accurately represents the distribution of the PIT size. Also, Figure 5.3(b) shows the effect of timeout value τ on the average PIT size. It is clear that our model accurately captures the average PIT size.

5.6.2 Constrained PIT Size

In this section, we assume that PIT is constrained to have no more than $\Theta = 2500$ entries, and that incoming Interests will be dropped if the PIT contains 2500 entries. Here, we evaluate the performance of the algorithm described in Section 5.4.1

Figures 5.4(a) and (b) show the changes in PIT timeout value based on the update rule (5.9) when Interest drop probability is bounded by $\delta \in \{10^{-3}, 10^{-6}\}$. Figures 5.4(c) and (d) show how the PIT size changes over time. The red lines in Figures 5.4(a)-(d) denote the optimal values for τ and $\mu(\tau)$ obtained by solving (5.8). Note how a smaller δ results in smaller PIT timeout and size.

We also use the update rule (5.10) to find the optimal timeout value for the above setting. Figure 5.5 shows the convergence of the PIT timeout to the optimal values. The convergence of PIT timeout is not smooth compared to Figures 5.4(a) and (b), but updates in this case require no estimation of PIT size mean and variance.

5.6.3 Memory-Bandwidth Tradeoff

Here, we use Algorithm 6 to find a solution for PIT timeout value and cache characteristic time so as to reduce the total system cost. For the bandwidth cost we take the following cost function

$$L_b(B) = \beta e^{\eta(B-R)},$$

with β, η and R as costs parameters. This cost function was proposed in [49] based on the motivation that the price charged by a server is low when the server is underloaded, *i.e.* $B < R$, and increases exponentially when the server operates in the

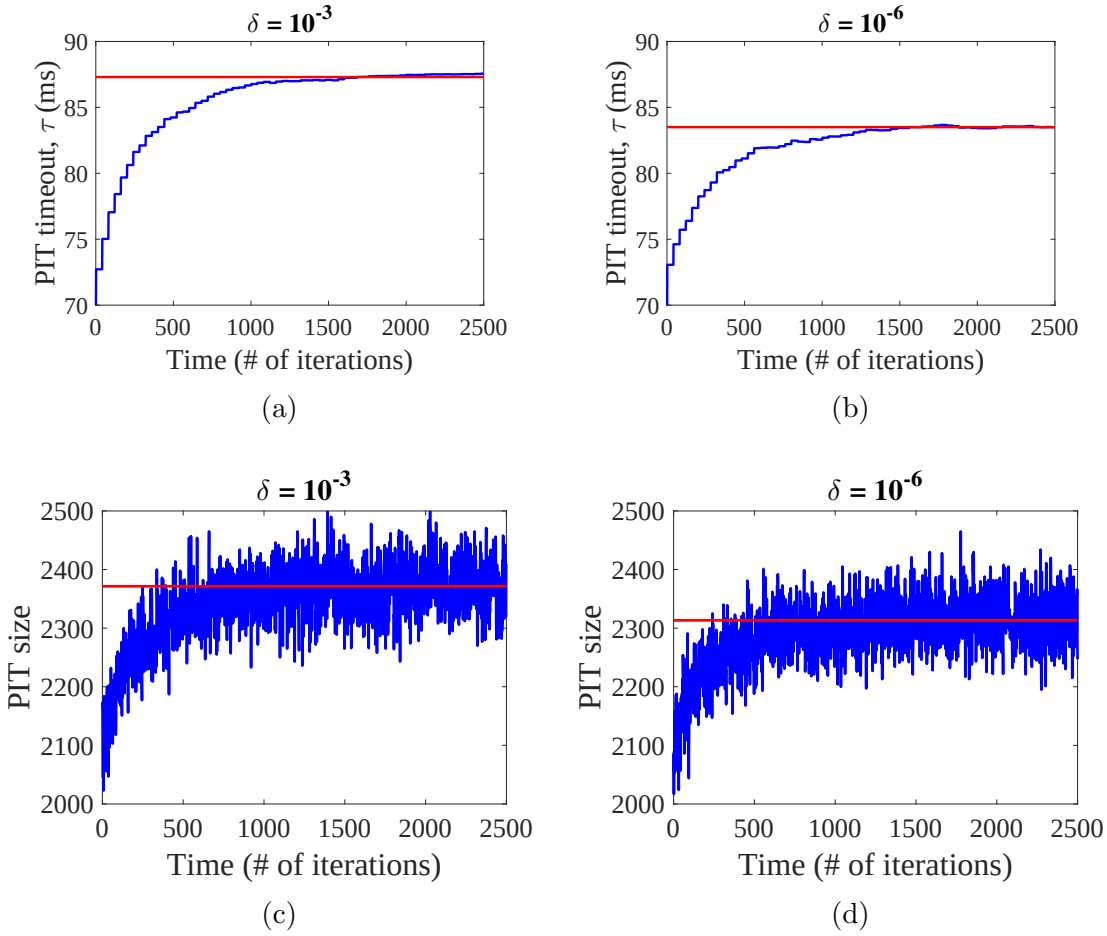


Figure 5.4: Convergence of PIT timeout and PIT size to their optimal values with update rule (5.9) with request dropping probability set to (a) $\delta = 10^{-3}$, and (b) $\delta = 10^{-6}$.

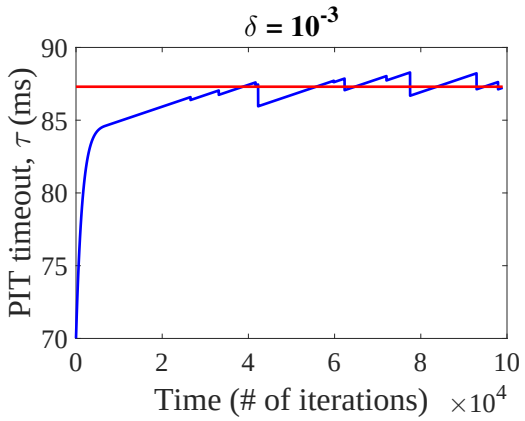


Figure 5.5: Convergence of PIT timeout to optimal value using update rule (5.10).

overloaded regime, *i.e.* $B > R$. Here, we take $\beta = \eta = 0.01$ and $R = 2 \times 10^4$. For cache and PIT storage costs we also usage

$$L_c(C) = e^{0.005(C-2000)} \quad \text{and} \quad L_s(S) = e^{0.005(S-1000)},$$

respectively.

Using brute-force to solve the optimization problem (5.11) with the above cost functions, we obtain the optimal solution to be at $\tau^* = 90$ms and $T^* = 200$ms, producing a cache of size $C^* = 3567$ and a PIT with $S^* = 1478$ entries. Figure 5.6 shows the convergence of the PIT timeout and cache characteristic time to their optimal values producing the optimal cache and PIT sizes. We observe that Algorithm 6 finds the global optimum in this example. In fact, we observe that, in most of our experiments, our approach either finds the optimal solution or gets very close it.

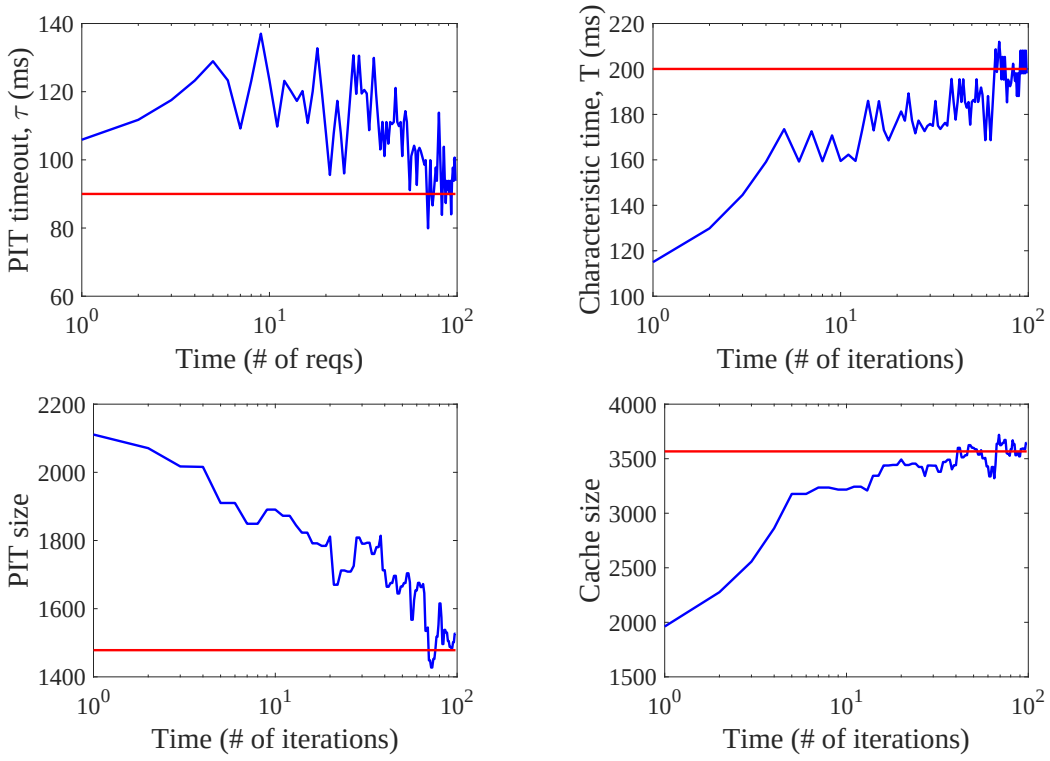


Figure 5.6: Convergence of PIT timeout τ and cache characteristic time T to the optimal values, producing the optimal PIT and cache sizes. Optimal parameter values obtained by solving (5.11) are shown through red lines.

5.7 Related Work

Prompted by the PIT size explosion, researchers have made a great effort in design and implementation of fast and scalable Pending Interest Tables, and reducing the required table size [15, 55, 75, 77, 81]. Dai *et al.* [15] were among the first to study PIT feasibility and proposed a tree-like structure for PIT to ensure fast look-up and update operations. You *et al.* [80] rely on Bloom Filters in order to reduce the memory space for implementing PITs. Authors in [74] propose a semi-stateless forwarding scheme in which requests are tracked every d hops instead of tracking each request at every on-path router. Virgilio *et al.* [76] provide a performance evaluation of the existing PIT architectures in terms of resilience to overload conditions. Their results reveal differentiated weaknesses in each architecture, emphasizing the need for better PIT management strategies. Their hint for a better mechanism is to manage a dynamic PIT timeout that adapts to the changes in network load. This is the approach we took in this chapter.

5.8 Conclusion

In this chapter, we first consider the problem of modeling an LRU cache with a Pending Interest Table. It is assumed that in case of a cache miss, it would take some time to download the content to the cache, where the download delay is modeled as a random variable. While the content is being downloaded to the cache, requests arriving for the content are aggregated at the Pending Interest Table, and not forwarded to the content custodian in order to reduce the load on the server. To prevent the PIT size from growing excessively large, PIT entries are removed after a

certain timeout period. We derive expressions for the cache hit probability, Interest aggregation probability, and the size of the Pending Interest Table as functions of the PIT timeout. Drawing on our analytical model, we then consider the problem of PIT timeout optimization with a strict constraint on the PIT size. Finally, we formulate the memory-bandwidth tradeoff as an optimization problem where the objective is to minimize the total cost of bandwidth usage and storage cost of cache and PIT. We develop online algorithms for the two optimization problems. We perform numerical simulations that demonstrated the accuracy of our approach in modeling LRU caches with Pending Interest Tables, and showed the efficacy of our online algorithms in finding the optimal timeout value and reducing system costs.

CHAPTER 6

CONCLUSION

This thesis is an effort in modeling and analyzing cache-enabled networks. Multiple aspects are considered and analytically studied with the purpose of understanding network performance and developing techniques to optimize them.

The problem of developing optimal joint routing and caching policies in a network supporting in-network caching with the goal of minimizing expected content-access delay is studied first. The problem is proven to be NP-complete, and low-complexity approximation algorithms are designed and shown to achieve near optimal performance.

Utility-driven caching is studied next where utilities are associated with each content expressed as functions of the corresponding content hit probability. Utility-driven caching provides a general framework for defining caching policies with considerations of fairness among various groups of files, and implications on market economy for cache service providers and content publishers. This framework has the capability to model existing caching policies such as FIFO and LRU, as utility-driven caching policies.

The problem of cache resource allocation among content providers is consider next, and a utility-optimizing framework is developed to determine a sharing/partitioning strategy to maximize the utility of content providers.

Finally, the performance of caching policies in accordance with the data structures proposed in future Internet architecture designs is considered. An analytical framework is provided that provides insights into the system behavior and enables developing algorithms for optimizing the performance.

APPENDICES

A Complexity of Network with Two Caches

Proof. Consider the highlighted elements of the matrix in Figure 1, and let r_1 denote the first row of the matrix. Also, let r_2 and r_3 denote the first two rows below the second horizontal line. It is easy to see that if these three rows are selected to be in R, any assignment satisfying Proposition 2 should have $-s(r_1) = s(r_2) = s(r_3)$. Otherwise, the signed sum of the rows will have entries other than $\{0, \pm 1\}$. This observation can be easily extended to see that rows below the second horizontal line can be considered in groups of two such that if the two rows are selected to be in R they will be assigned the same sign.

We sign the rows in R starting from the rows below the second horizontal line. Considering the groups of two rows, we make assignments such that the elements to the left of the vertical line of the signed sum of the rows are in $\{0, -1\}$ only. To see why this is possible, note that the non-zero elements of the matrix to the left of the vertical line can be seen as small blocks of 2×2 matrices. It is easy to see that the signed sum of any subset of these blocks can be made to have elements only in $\{0, -1\}$, with rows in the same group getting the same assignment. The rows between the two horizontal lines are always signed $+1$. The sign of the rows above the first horizontal line follows the assignment of the lines below the second horizontal line based on the previous discussion.

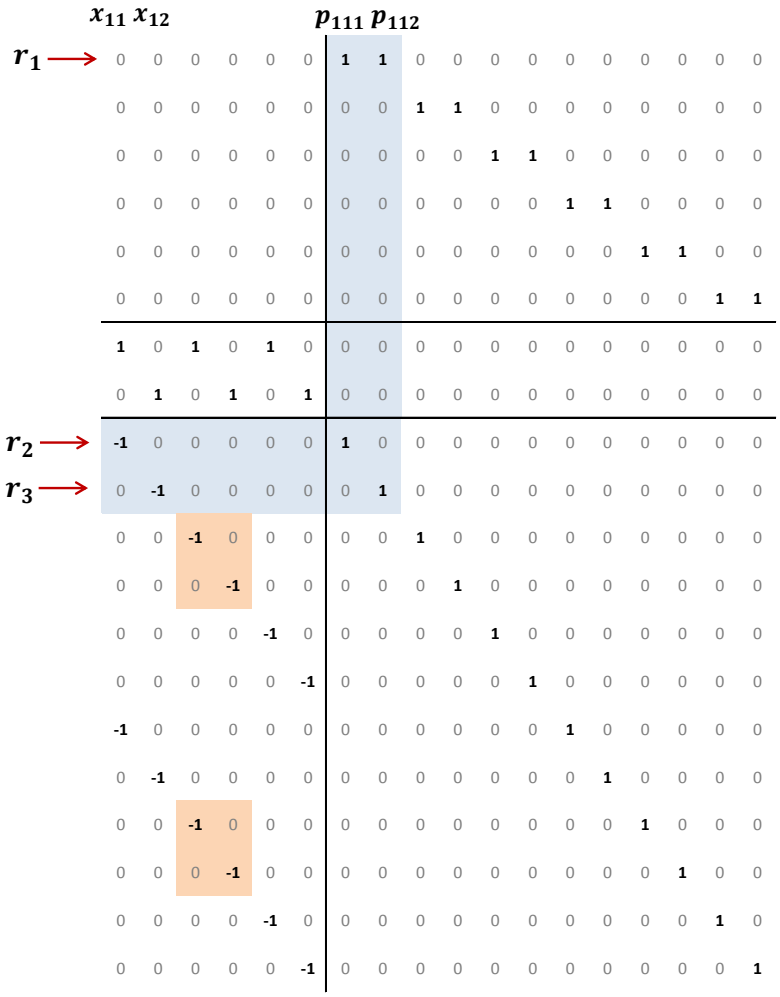


Figure 1: An example of the constraints matrix A for a network with two caches, two users and three files

With the above procedure, the sum of the signed vectors will have entries in $\{0, \pm 1\}$ for any set of rows R, and from Proposition 2 it follows that the matrix A is totally unimodular, and hence the solution for the optimization problem in (2.1) for a network with two caches can be found in polynomial time. □

B Equal Cardinality Partitioning is NP-hard

Proof. A proof of NP-hardness of a more general form of ECP is given in [14]. Here, we give a simpler proof by a reduction from the Partition problem.

Problem 4. *(Partition) Given a set A of n positive integers, can A be partitioned into two disjoint subsets A_1 and A_2 such that $A = A_1 \cup A_2$ and the sum of the numbers in A_1 equals the sum of the numbers in A_2?*

For each instance of Partition with input $A = \{a_1, \ldots, a_n\}$ create an instance $A' = \{a_1, \ldots, a_n, 0, \ldots, 0\}$ by adding n zeros to A. It is easy to see that A' can be partitioned into two subsets with equal cardinality if and only if A can be partitioned. Therefore, Partition $\leq_P$ ECP, and ECP is NP-hard. □

C Congestion Sensitive Delay Decision Problem is NP-complete

Proof. It is easy to see that given some $\mathbf{x}, \mathbf{p}$ the expected delay $D(\mathbf{x}, \mathbf{p})$ can be computed in polynomial time, and hence CSDDP is in NP. To show it is NP-hard, we reduce the problem of Equal Cardinality Partition (ECP) to our problem. For an in-

stance of the ECP(A) problem we create the instance CSDDP($S, A, [\frac{4}{S}], [+\infty], [\frac{4}{a_i}], \frac{n}{2}$) where $S = \sum_{a_i \in A} a_i$.

Now, the set A can be partitioned into subsets A_1 and A_2 with $|A_1| = |A_2|$ if and only if CSDDP achieves delay $(2n + 3)/S$.

To see more clearly why the reduction works, first note that with the delay values being set to $d_i^h = 4/S$ and $d_i^b = 4/a_i$, since $d_i^h < d_i^b$ if a file exists in the cache all the requests for that file will be directed to the cache. Also, since $d_i^c = +\infty$, if a file is not in the cache all the requests for that file will be requested from the back-end server. Therefore, we have $p_i = x_i, \forall i$. Now, with the service rate set to $\mu = S$, we can re-write the optimization problem in (2.2) as follows

$$\begin{aligned} \text{minimize } & \frac{1}{S}\left[\frac{4}{S}\sum_{i=1}^{n} a_i x_i + 4\sum_{i=1}^{n}(1 - x_i) + \frac{S}{\sum_{i=1}^{n} a_i x_i} - 1\right] \\ \text{such that } & \sum_{i=1}^{n} x_i \leq \frac{n}{2} \\ & x_i \in \{0, 1\} \end{aligned}$$

Now, looking at the objective function in the above problem, we can see that

$$Z_1 = 4\sum_{i=1}^{n}(1 - x_i) \geq 2n$$

since we should have $\sum_{i=1}^{n} x_i \leq n/2$. Moreover, $Z_1 = 2n$ if $\sum_{i=1}^{n} x_i = n/2$ meaning that exactly half of the files are in the cache. We also have that

$$Z_2 = \frac{4}{S}\sum_{i=1}^{n} a_i x_i + \frac{S}{\sum_{i=1}^{n} a_i x_i} \geq 4,$$

and $Z_2 = 4$ only if $\sum_{i=1}^{n} a_i x_i = S/2$.

Hence, $Z_1 + Z_2 - 1 = 2n + 3$ if and only if $\sum_{i=1}^{n} a_i x_i = S/2$ and $\sum_{i=1}^{n} x_i = n/2$.

Therefore, if CSDDP$(S, A, [\frac{4}{S}], [+\infty], [\frac{4}{a_i}], \frac{n}{2})$ achieves minimum delay $(2n+3)/S$ then A can be partitioned into equal cardinality subsets.

It is easy to see that if A can be partitioned into two subsets of equal cardinality, then CSDDP$(S, A, [\frac{4}{S}], [+\infty], [\frac{4}{a_i}], \frac{n}{2})$ has minimum delay of $(2n+3)/S$. □

D Gain Function from Caching and Routing is Monotone Increasing and Submodular

Here, we will first prove the lemma for the more general case of convex delay rate function, and as an example consider G/G/1 queues. Given a cache configuration X, let

$$(\mathbf{d}_X^h)_{ij} = \inf\left\{d_{im}^h : X_{jm} = 1\right\},$$

$$(\mathbf{d}_X^c)_{ij} = \inf\left\{d_{im}^c : X_{jm} = 0\right\},$$

denote the minimum cache hit and miss delays for user i accessing file j, with the convention $\inf \emptyset = +\infty$. We assume that $d_{im}^h \leq d_{im}^c$ and hence $0 \leq \mathbf{d}_X^h \leq \mathbf{d}_X^c \leq +\infty$.

Define $(\mathbb{R})_{ij} = \lambda_i q_{ij}$, and let $(\boldsymbol{\lambda}_X^h)_{ij}$, $(\boldsymbol{\lambda}_X^c)_{ij}$, $(\boldsymbol{\lambda}_X^b)_{ij}$ denote the rate of requests for file j sent by user i that are routed through caches containing file j, routed through caches without file j, and directly routed to the back-end server, respectively. We suppress the subscript X when no confusion arises.

The average delay can be written as

$$D(\boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^b) = \boldsymbol{\lambda}^h \cdot \mathbf{d}^h + \boldsymbol{\lambda}^c \cdot \mathbf{d}^c + f(\lambda^c) + g(\lambda^b),$$

where[1] $\lambda^c = \boldsymbol{\lambda}^c \cdot \mathbf{1}$ and $\lambda^b = \boldsymbol{\lambda}^b \cdot \mathbf{1}$, and $f(\cdot)$ and $g(\cdot)$ denote the *total expected delay cost rate* for the G/G/1 queues representing the paths from caches to the back-end server, and direct paths from users to back-end servers, respectively. The total expected delay cost rate function for G/G/1 queues is proved to be convex in [28].

Now we have the following optimization problem,

$$\begin{aligned} \text{minimize} \quad & D(\boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^b) \\ \text{subject to} \quad & \boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^b \geq \mathbf{0}, \\ & \boldsymbol{\lambda}^h + \boldsymbol{\lambda}^c + \boldsymbol{\lambda}^b = \mathbb{R}. \end{aligned} \tag{1}$$

Since Slater's condition holds, the optimal delay $D^\star$ can be found by solving the dual optimization problem. For notational simplicity, we first assume that $\mathbf{d}^h$ and $\mathbf{d}^c$ are finite. This assumption can be easily removed by setting to zero the components of $\boldsymbol{\lambda}^h$ and $\boldsymbol{\lambda}^c$ corresponding to the infinite components of $\mathbf{d}^h$ and $\mathbf{d}^c$, which simply reduces the number of decision variables.

The Lagrangian for (1)

$$L(\boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^b, \boldsymbol{\nu}, \boldsymbol{\xi}^h, \boldsymbol{\xi}^c, \boldsymbol{\xi}^b) = D + \boldsymbol{\nu} \cdot (\mathbb{R} - \boldsymbol{\lambda}^h - \boldsymbol{\lambda}^c - \boldsymbol{\lambda}^b) - \boldsymbol{\xi}^h \cdot \boldsymbol{\lambda}^h - \boldsymbol{\xi}^c \cdot \boldsymbol{\lambda}^c - \boldsymbol{\xi}^b \cdot \boldsymbol{\lambda}^b$$

[1]The dot product is defined by $\mathbf{a} \cdot \mathbf{b} = \sum_{ij} a_{ij} b_{ij}$.

$$= \boldsymbol{\nu} \cdot \mathbb{R} - \boldsymbol{\eta}^c \cdot \boldsymbol{\lambda}^h - \boldsymbol{\eta}^u \cdot \boldsymbol{\lambda}^c - \boldsymbol{\eta}^s \cdot \boldsymbol{\lambda}^b + f(\lambda^c) + g(\lambda^b),$$

where $\boldsymbol{\eta}^h = \boldsymbol{\nu} + \boldsymbol{\xi}^h - \mathbf{d}^h$, $\boldsymbol{\eta}^c = \boldsymbol{\nu} + \boldsymbol{\xi}^c - \mathbf{d}^c$ and $\boldsymbol{\eta}^b = \boldsymbol{\nu} + \boldsymbol{\xi}^b$. The dual function is

$$\begin{aligned}\hat{D}(\boldsymbol{\nu}, \boldsymbol{\xi}^h, \boldsymbol{\xi}^c, \boldsymbol{\xi}^b) &= \inf_{\boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^h} L(\boldsymbol{\lambda}^h, \boldsymbol{\lambda}^c, \boldsymbol{\lambda}^b, \boldsymbol{\nu}, \boldsymbol{\xi}^h, \boldsymbol{\xi}^c, \boldsymbol{\xi}^b) \\ &= \inf_{\boldsymbol{\lambda}^h}(\boldsymbol{\nu} \cdot \mathbb{R} - \boldsymbol{\eta}^h \cdot \boldsymbol{\lambda}^h) + \inf_{\boldsymbol{\lambda}^c}[f(\lambda^c) - \boldsymbol{\eta}^c \cdot \boldsymbol{\lambda}^c] + \inf_{\boldsymbol{\lambda}^b}[g(\lambda^b) - \boldsymbol{\eta}^b \cdot \boldsymbol{\lambda}^b].\end{aligned}$$

For $\hat{D} > -\infty$, we need $\boldsymbol{\eta}^h = \mathbf{0}$, $\boldsymbol{\eta}^c \geq \mathbf{0}$, $\boldsymbol{\eta}^b \geq \mathbf{0}$. When this condition is met,

$$\hat{D}(\boldsymbol{\nu}, \boldsymbol{\xi}^h, \boldsymbol{\xi}^c, \boldsymbol{\xi}^b) = \boldsymbol{\nu} \cdot \mathbb{R} - f^*(\max(\boldsymbol{\eta}^c)) - g^*(\max(\boldsymbol{\eta}^b)),$$

where f^* and g^* are the conjugates of f and g respectively, defined by

$$f^*(y) = \sup_x [xy - f(x)], \quad g^*(y) = \sup_x [xy - g(x)].$$

Thus the dual problem becomes

$$\begin{aligned} \text{maximize} \quad & \boldsymbol{\nu} \cdot \mathbb{R} - f^*(\max(\boldsymbol{\eta}^c)) - g^*(\max(\boldsymbol{\eta}^b)) \\ \text{subject to} \quad & \boldsymbol{\eta}^c, \boldsymbol{\eta}^b \geq \mathbf{0}, \\ & \mathbf{d}^h - \boldsymbol{\nu} \geq \mathbf{0}, \\ & \boldsymbol{\eta}^c + \mathbf{d}^c - \boldsymbol{\nu} \geq \mathbf{0}, \\ & \boldsymbol{\eta}^b - \boldsymbol{\nu} \geq \mathbf{0}. \end{aligned}$$

or

$$\begin{aligned} &\text{maximize} \quad \tilde{D}(s,u) = \mathbf{m}(s,u) \cdot \mathbb{R} - f^*(u) - g^*(s) \\ &\text{subject to} \quad s, u \geq 0, \end{aligned} \tag{2}$$

where $\mathbf{m}(s,u) = \mathbf{d}^h \wedge (s\mathbf{1}) \wedge (\mathbf{d}^c + u\mathbf{1})$ with $\wedge$ denoting component-wise minimum.

Now we make explicit the dependence on cache configuration. Given the cache configuration X, let $s^\star_X$ and $u^\star_X$ be an optimal solution of the dual problem (2) and $D^\star_X$ the optimal delay.

Let X, Y be two cache configurations. We want to show

$$D^\star_X + D^\star_Y \leq D^\star_{X\cup Y} + D^\star_{X\cap Y}. \tag{3}$$

Assume that $X \setminus Y \neq \emptyset$ and $Y \setminus X \neq \emptyset$; otherwise, (3) holds trivially. Without loss of generality, assume $u^\star_X \leq u^\star_Y$. Then

$$\begin{aligned} D^\star_X + D^\star_Y - D^\star_{X\cup Y} - D^\star_{X\cap Y} =& \tilde{D}_X(s^\star_X, u^\star_X) + \tilde{D}_Y(s^\star_Y, u^\star_Y) \\ &- \tilde{D}_{X\cup Y}(s^\star_{X\cup Y}, u^\star_{X\cup Y}) - \tilde{D}_{X\cap Y}(s^\star_{X\cap Y}, u^\star_{X\cap Y}) \\ \leq& \tilde{D}_X(s^\star_X, u^\star_X) + \tilde{D}_Y(s^\star_Y, u^\star_Y) \\ &- \tilde{D}_{X\cup Y}(s^\star_X \wedge s^\star_Y, u^\star_X) - \tilde{D}_{X\cap Y}(s^\star_X \vee s^\star_Y, u^\star_Y) \\ =& \mathbb{R} \cdot [\mathbf{m}_X(s^\star_X, u^\star_X) + \mathbf{m}_Y(s^\star_Y, u^\star_Y) \\ &- \mathbf{m}_{X\cup Y}(s^\star_X \wedge s^\star_Y, u^\star_X) - \mathbf{m}_{X\cap Y}(s^\star_X \vee s^\star_Y, u^\star_Y)] \\ =& \mathbb{R} \cdot [\boldsymbol{\Delta}_1 - \boldsymbol{\Delta}_2], \end{aligned}$$

where

$$\boldsymbol{\Delta}_1 = \mathbf{m}_X(s_X^\star, u_X^\star) - \mathbf{m}_{X\cup Y}(s_X^\star \wedge s_Y^\star, u_X^\star),$$
$$\boldsymbol{\Delta}_2 = \mathbf{m}_{X\cap Y}(s_X^\star \vee s_Y^\star, u_Y^\star) - \mathbf{m}_Y(s_Y^\star, u_Y^\star).$$

Define $h(x, y, z) = x \wedge y - x \wedge z$, which is nonnegative and increasing in x for $y \geq z$. Now consider two cases,

1. $s_X^\star \geq s_Y^\star$.

$$\begin{aligned}\boldsymbol{\Delta}_1 &= \mathbf{d}_X^h \wedge (\mathbf{d}_{\bar{X}}^c + u_X^\star \mathbf{1}) \wedge (s_X^\star \mathbf{1}) - \mathbf{d}_{X\cup Y}^h \wedge (\mathbf{d}_{\overline{X\cup Y}}^c + u_X^\star \mathbf{1}) \wedge (s_Y^\star \mathbf{1}) \\ &= h\Big(\mathbf{d}_X^h \wedge (\mathbf{d}_{\overline{X\cup Y}}^c + u_X^\star \mathbf{1}), (\mathbf{d}_{Y\setminus X}^c + u_X^\star \mathbf{1}) \wedge (s_X^\star \mathbf{1}), \mathbf{d}_{Y\setminus X}^h \wedge (s_Y^\star \mathbf{1})\Big) \\ &\leq h\Big(\mathbf{d}_X^h \wedge (\mathbf{d}_{\overline{X\cup Y}}^c + u_Y^\star \mathbf{1}), (\mathbf{d}_{Y\setminus X}^c + u_Y^\star \mathbf{1}) \wedge (s_X^\star \mathbf{1}), \mathbf{d}_{Y\setminus X}^h \wedge (s_Y^\star \mathbf{1})\Big),\end{aligned}$$

and

$$\begin{aligned}\boldsymbol{\Delta}_2 &= \mathbf{d}_{X\cap Y}^h \wedge (\mathbf{d}_{\overline{X\cap Y}}^c + u_Y^\star \mathbf{1}) \wedge (s_X^\star \mathbf{1}) - \mathbf{d}_Y^h \wedge (\mathbf{d}_{\bar{Y}}^c + u_Y^\star \mathbf{1}) \wedge (s_Y^\star \mathbf{1}) \\ &= h\Big(\mathbf{d}_{X\cap Y}^h \wedge (\mathbf{d}_{\bar{Y}}^c + u_Y^\star \mathbf{1}), (\mathbf{d}_{Y\setminus X}^c + u_Y^\star \mathbf{1}) \wedge (s_X^\star \mathbf{1}), \mathbf{d}_{Y\setminus X}^h \wedge (s_Y^\star \mathbf{1})\Big).\end{aligned}$$

Since

$$\begin{aligned}\mathbf{d}_{X\cap Y}^h \wedge (\mathbf{d}_{\bar{Y}}^c + u_Y^\star \mathbf{1}) - \mathbf{d}_X^h \wedge (\mathbf{d}_{\overline{X\cup Y}}^c + u_Y^\star \mathbf{1}) &= h\Big(\mathbf{d}_{X\cap Y}^h \wedge (\mathbf{d}_{\overline{X\cup Y}}^c + u_Y^\star \mathbf{1}), \mathbf{d}_{X\setminus Y}^c + u_Y^\star \mathbf{1}, \mathbf{d}_{X\setminus Y}^h\Big) \\ &\geq \mathbf{0},\end{aligned}$$

it follows that $\boldsymbol{\Delta}_1 \leq \boldsymbol{\Delta}_2$.

2. $s^\star_X \leq s^\star_Y$.

$$\begin{aligned}\boldsymbol{\Delta}_1 &= \mathbf{d}^h_X \wedge (\mathbf{d}^c_{\bar{X}} + u^\star_X \mathbf{1}) \wedge (s^\star_X \mathbf{1}) - \mathbf{d}^h_{X\cup Y} \wedge (\mathbf{d}^c_{\overline{X\cup Y}} + u^\star_X \mathbf{1}) \wedge (s^\star_X \mathbf{1}),\\ &= h\Big(\mathbf{d}^h_X \wedge (\mathbf{d}^c_{\overline{X\cup Y}} + u^\star_X \mathbf{1}) \wedge (s^\star_X \mathbf{1}), \mathbf{d}^c_{Y\setminus X} + u^\star_Y \mathbf{1}, \mathbf{d}^h_{Y\setminus X}\Big)\\ &\leq h\Big(\mathbf{d}^h_X \wedge (\mathbf{d}^c_{\overline{X\cup Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1}), \mathbf{d}^c_{Y\setminus X} + u^\star_Y \mathbf{1}, \mathbf{d}^h_{Y\setminus X}\Big),\end{aligned}$$

and

$$\begin{aligned}\boldsymbol{\Delta}_2 &= \mathbf{d}^h_{X\cap Y} \wedge (\mathbf{d}^c_{\overline{X\cap Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1}) - \mathbf{d}^h_Y \wedge (\mathbf{d}^c_{\bar{Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1})\\ &= h\Big(\mathbf{d}^h_{X\cap Y} \wedge (\mathbf{d}^c_{\bar{Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1}), \mathbf{d}^c_{Y\setminus X} + u^\star_Y \mathbf{1}, \mathbf{d}^h_{Y\setminus X}\Big).\end{aligned}$$

Since

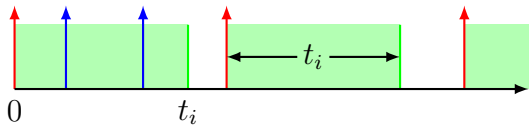
$$\mathbf{d}^h_X \wedge (\mathbf{d}^c_{\overline{X\cup Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1}) \leq \mathbf{d}^h_{X\cap Y} \wedge (\mathbf{d}^c_{\bar{Y}} + u^\star_Y \mathbf{1}) \wedge (s^\star_Y \mathbf{1}),$$

it follows that $\boldsymbol{\Delta}_1 \leq \boldsymbol{\Delta}_2$.

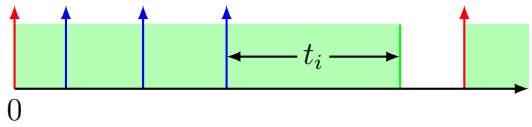
In both cases, (3) holds and hence $D^\star$ is supermodular, and $D_\emptyset - D^\star$ is submodular.

E Time-To-Live Caches and Characteristic Time

Figure 2 shows cache dynamics with requests for a file with (a) a non-reset, and (b) a reset TTL cache assuming the timer for file i equals t_i. We model the dynamics of a single file only, as we are considering a TTL cache with no capacity constraints. The value of t_i for each policy will be determined based on the cache capacity.



(a) A non-reset TTL cache with timer t_i.



(b) A reset TTL cache with timer t_i.

Figure 2: Cache hits and misses for requests for a given file with non-reset and reset TTL caches.

Looking at Figure 2, we can see that the cache occupancy process for a file can be divided into cycles that are separated by cache misses. Note that these cycles are statically the same. Assuming the random variable N_i denotes the number of cache hits for file i in a cycle, the hit probability can be expressed as

$$h_i = \frac{\mathbb{E}[N_i]}{1 + \mathbb{E}[N_i]}.$$

- **Non-reset TTL:** For a non-reset TTL cache, N_i denotes the number of requests within time t_i, and hence for Poisson arrivals with rate λ_i, we obtain

$$\mathbb{E}[N_i^{\text{non-reset}}] = \lambda_i t_i,$$

and hence

$$h_i^{\text{non-reset}} = \frac{\lambda_i t_i}{1 + \lambda_i t_i}.$$

- **Reset TTL:** With a reset TTL cache, a file continues to stay in the cache as long as the consecutive request inter-arrival times are not larger than t_i. Hence, N_i^{reset} defines a geometric distribution with "success" probability $p_i = e^{-\lambda_i t_i}$ for Poisson arrivals. Therefore, we obtain

$$\mathbb{E}[N_i^{\text{reset}}] = \frac{1 - p_i}{p_i} = e^{\lambda_i t_i} - 1,$$

and

$$h_i^{\text{reset}} = 1 - e^{-\lambda_i t_i}.$$

Cache Characteristic Time

Che *et al.* [12] introduced the notion of *cache characteristic time* and used it to approximate the hit probability of an LRU cache with Poisson arrivals. According to Che *et al.* [12] the hit probability of a file with request rate λ_i in an LRU cache can be approximated by

$$h_i = 1 - e^{-\lambda_i T},$$

where T is a constant referred to as the cache characteristic time. The characteristic time approximation was later theoretically justified and extended to other caching policies such as FIFO and RANDOM [27].

Cache characteristic time maps replacement-based caching policies to TTL-based caching policies. The characteristic time for a cache denotes the time since a file is placed at the head of the cache until it gets evicted. The assumption in the approximation is to assume the characteristic time to be a single constant for all files.

An LRU cache is modeled as a reset TTL cache since with every request for a file it is moved to head of the cache. A FIFO cache, however, is modeled as a non-reset TTL cache. Figure 2 shows cache dynamics with requests for a file with (a) a FIFO cache modeled as a non-reset TTL cache, and (b) an LRU cache modeled as a reset TTL cache assuming characteristic time is T. We model the dynamics of a single file only, as we are considering a TTL cache with no capacity constraints. The value of T for each policy will be determined based on the cache capacity.

Looking at Figure 2, we can see that the cache occupancy process for a file can be divided into cycles that are separated by cache misses. Note that these cycles are statically the same. Assuming the random variable N_i denotes the number of cache hits for file i in a cycle, the hit probability can be expressed as

$$h_i = \frac{\mathbb{E}[N_i]}{1 + \mathbb{E}[N_i]}.$$

- **FIFO:** With the FIFO policy, N_i denotes the number of requests withing time T, and hence for Poisson arrivals with rate λ_i, we obtain

  $$\mathbb{E}[N_i^{\text{FIFO}}] = \lambda_i T,$$

  and hence

  $$h_i^{\text{FIFO}} = \frac{\lambda_i T}{1 + \lambda_i T}.$$

- **LRU:** With the LRU policy, a file continues to stay in the cache as long as the consecutive requests inter-arrival times are no larger than T. Hence,

N_i^{LRU} defines a geometric distribution with "success" probability $p_i = e^{-\lambda_i T}$ for Poisson arrivals. Therefore, we obtain

$$\mathbb{E}[N_i^{\text{LRU}}] = \frac{1-p_i}{p_i} = e^{\lambda_i T} - 1,$$

and

$$h_i^{\text{LRU}} = 1 - e^{-\lambda_i T}.$$

For the two policies, the value of the timer T is determined according to the cache capacity constraint

$$\sum_i h_i = C.$$

F Stability of Dual Algorithm

We first note that $D(\alpha)$ is the dual of a convex function and has a unique minimizer α^*. The function $V(\alpha) = D(\alpha) - D(\alpha^*)$, hence, is a non-negative function and equals zero only at $\alpha = \alpha^*$. Differentiating $V(\alpha)$ with respect to time we get

$$\dot{V}(\alpha) = \frac{\partial V}{\partial \alpha}\dot{\alpha} = -\gamma\Big(\sum_i s_i h_i - B\Big)^2 < 0.$$

Therefore, $V(\cdot)$ is a Lyapunov function[2], and the system state will converge to optimum starting from any initial condition.

[2]A description of Lyapunov functions and their applications can be found in [67].

G Stability of Primal Algorithm

We first note that since $W(\mathbf{h})$ is a strictly concave function, it has a unique maximizer $\mathbf{h}^*$. Moreover $V(\mathbf{h}) = W(\mathbf{h}^*) - W(\mathbf{h})$ is a non-negative function and equals zero only at $\mathbf{h} = \mathbf{h}^*$. Differentiating $V(\cdot)$ with respect to time we obtain

$$\dot{V}(\mathbf{h}) = \sum_i \frac{\partial V}{\partial h_i}\dot{h_i} = -\sum_i \left(U_i'(h_i) - C'(\sum_i s_i h_i - B) \right) \dot{h_i}.$$

For $\dot{h_i}$ we have

$$\dot{h_i} = \frac{\partial h_i}{\partial t_i}\dot{t_i}.$$

For non-reset and reset TTL caches we have

$$\frac{\partial h_i}{\partial t_i} = \frac{\lambda_i}{(1+\lambda_i t_i)^2} \qquad \text{and} \qquad \frac{\partial h_i}{\partial t_i} = \lambda_i e^{-\lambda_i t_i},$$

respectively, and hence $\partial h_i / \partial t_i > 0$.

From the controller for t_i we have

$$t_i = k_i \left(U_i'(h_i) - C'(\sum_i s_i h_i - B) \right).$$

Hence, we get

$$\dot{V}(\mathbf{h}) = -\sum_i k_i \frac{\partial h_i}{\partial t_i} \left(U_i'(h_i) - C'(\sum_i s_i h_i - B) \right)^2 < 0.$$

Therefore, $V(\cdot)$ is a Lyapunov function, and the system state will converge to $\mathbf{h}^*$ starting from any initial condition.

H Stability of Primal-Dual Algorithm

As discussed in Section 3.3, the Lagrangian function for the optimization problem (4.11) is expressed as

$$\mathcal{L}(\mathbf{h}, \alpha) = \sum_i U_i(h_i) - \alpha(\sum_i s_i h_i - B).$$

Note that $\mathcal{L}(\mathbf{h}, \alpha)$ is concave in $\mathbf{h}$ and convex in α, and hence first order condition for optimality of $\mathbf{h}^*$ and α^* implies (see [22])

$$\mathcal{L}(\mathbf{h}^*, \alpha) \leq \mathcal{L}(\mathbf{h}, \alpha) + \sum_i \frac{\partial \mathcal{L}}{\partial h_i}(h_i^* - h_i),$$
$$\mathcal{L}(\mathbf{h}, \alpha^*) \geq \mathcal{L}(\mathbf{h}, \alpha) + \frac{\partial \mathcal{L}}{\partial \alpha}(\alpha^* - \alpha).$$

Assume that the hit probability of a file can be expressed by $f(\cdot)$ as a function of the corresponding timer value t_i, *i.e.* $h_i = f(t_i)$. The temporal derivative of the hit probability can therefore be expressed as

$$\dot{h_i} = f'(t_i)\dot{t_i},$$

or equivalently

$$\dot{h_i} = f'(f^{-1}(h_i))\dot{t_i},$$

where $f^{-1}(\cdot)$ denotes the inverse of function $f(\cdot)$. For notation brevity we define $g(h_i) = f'(f^{-1}(h_i))$. Note that as discussed in Appendix G, $f(\cdot)$ is an increasing function, and hence $g(h_i) \geq 0$.

In the remaining, we show that $V(\mathbf{h}, \alpha)$ defined below is a Lyapunov function for the primal-dual algorithm:

$$V(\mathbf{h}, \alpha) = \sum_i \int_{h_i^*}^{h_i} \frac{x - h_i^*}{k_i g(x)} \mathrm{d}x + \frac{1}{2\gamma}(\alpha - \alpha^*)^2.$$

Differentiating the above function with respect to time we obtain

$$\dot{V}(\mathbf{h}, \alpha) = \sum_i \frac{h_i - h_i^*}{k_i g(h_i)} \dot{h_i} + \frac{\alpha - \alpha^*}{\gamma} \dot{\alpha}.$$

Based on the controllers defined for t_i and α we have

$$\dot{h_i} = g(h_i)\dot{t_i} = k_i g(h_i) \frac{\partial \mathcal{L}}{\partial h_i},$$

and

$$\dot{\alpha} = -\gamma \frac{\partial \mathcal{L}}{\partial \alpha}.$$

Replacing for $\dot{h_i}$ and $\dot{\alpha}$ in $\dot{V}(\mathbf{h}, \alpha)$, we obtain

$$\begin{aligned}
\dot{V}(\mathbf{h}, \alpha) &= \sum_i (h_i - h_i^*) \frac{\partial \mathcal{L}}{\partial h_i} - (\alpha - \alpha^*) \frac{\partial \mathcal{L}}{\partial \alpha} \\
&\leq \mathcal{L}(\mathbf{h}, \alpha) - \mathcal{L}(\mathbf{h}^*, \alpha) + \mathcal{L}(\mathbf{h}, \alpha^*) - \mathcal{L}(\mathbf{h}, \alpha) \\
&= \Big(\mathcal{L}(\mathbf{h}^*, \alpha^*) - \mathcal{L}(\mathbf{h}^*, \alpha)\Big) + \Big(\mathcal{L}(\mathbf{h}, \alpha^*) - \mathcal{L}(\mathbf{h}^*, \alpha^*)\Big) \\
&\leq 0,
\end{aligned}$$

where the last inequality follows from

$$\mathcal{L}(\mathbf{h}, \alpha^*) \leq \mathcal{L}(\mathbf{h}^*, \alpha^*) \leq \mathcal{L}(\mathbf{h}^*, \alpha),$$

for any $\mathbf{h}$ and α.

Moreover, $V(\mathbf{h}, \alpha)$ is non-negative and equals zero only at $(\mathbf{h}^*, \alpha^*)$. Therefore, $V(\mathbf{h}, \alpha)$ is a Lyapunov function, and the system state will converge to optimum starting from any initial condition.

I Hit rate is a concave and increasing function of cache size.

Lemma 4. *The hit rate h_k is a concave and strictly increasing function of C_k.*

Proof. Differentiating (4.12) and (4.13) w.r.t. C_k gives

$$\begin{aligned} \frac{dh_k(C_k)}{dC_k} &= \lambda_k^2 \sum_{i=1}^{n_k} p_{k,i}^2 e^{-\lambda_k p_{k,i} T_k(C_k)} \frac{dT_k(C_k)}{dC_k} \\ 1 &= \lambda_k \sum_{i=1}^{n_k} p_{k,i} e^{-\lambda_k p_{k,i} T_k(C_k)} \frac{dT_k(C_k)}{dC_k}. \end{aligned}$$

The latter equation implies that $dT_k(C_k)/dC_k > 0$, which in turn implies from the former that $dh_k(C_k)/dC_k > 0$. This proves that $h_k(C_k)$ is strictly increasing in C_k. Differentiating now the above equations w.r.t. C_k yields

$$\frac{1}{\lambda_k^2} \frac{d^2 h_k(C_k)}{dC_k^2} = \sum_{i=1}^{n_k} p_{k,i}^2 e^{-\lambda_k p_{k,i} T_k(C_k)} g_{k,i}$$

$$0 \;=\; \sum_{i=1}^{n_k} p_{k,i} e^{-\lambda_k p_{k,i} T_k(C_k)} g_{k,i} \tag{4}$$

with $g_{k,i} := d^2T_k(C_k)/dC_k^2 - \lambda_k p_{k,i}(dT_k(C_k)/dC_k)^2$. Assume without loss of generality that $0 \le p_{k,1} \le \cdots \le p_{k,n_k} \le 1$. (4) implies that there exists $1 \le l \le n_k$ such that $g_{k,i} \ge 0$ for $i = 1, \ldots, l$ and $g_{k,i} \le 0$ for $i = l+1, \ldots, n_k$. Hence,

$$\begin{aligned}
\frac{1}{\lambda_k^2}\frac{d^2h_k(C_k)}{dC_k^2} &\le \sum_{i=1}^{l} p_{k,i} e^{-\lambda_k p_{k,i} T_k(C_k)} g_{k,i} + \sum_{i=l+1}^{n_k} p_{k,i}^2 e^{-\lambda_k p_{k,i} T_k(C_k)} g_{k,i} \\
&= \sum_{i=l}^{n_k} p_{k,i} e^{-\lambda_k p_{k,i} T_k(C_k)} g_{k,i}(1 - p_{k,i}) \le 0.
\end{aligned}$$

This proves that $h_k(C_k)$ is concave in C_k. □

J LRU Asymptotic Miss Rate

Proof. We first construct a CDF F from the CP specific CDFs, $\{F_k\}$. When the providers share a single cache, documents are labelled $1, \ldots, Bn$, so that documents $B_{k-1}+1, \ldots, B_k$ are the b_k documents with service provider k. Denote $A_k := \sum_{j=1}^{k} a_j$ with $A_0 = 0$ in what follows. Define

$$F(x) = \sum_{j=1}^{K} \Big(A_{k-1} + a_k F_k\Big(\frac{B}{b_k}x - \frac{B_{k-1}}{b_k}\Big)\Big) \times \mathbf{1}\left\{ \frac{B_{k-1}}{B} \le x \le \frac{B_k}{B} \right\}. \tag{5}$$

Let

$$p_i^{(n)} := F\left(\frac{i}{Bn}\right) - F\left(\frac{i-1}{Bn}\right), \quad i = 1, \ldots, nB_K$$

It is easy to see that

$$p^{(n)}_{B_{k-1}+i} = a_k\, p^{(n)}_{k,i}, \quad i = 1, \ldots, b_k;\ k = 1, \ldots, K.$$

Note that F may not be differentiable at $x \in \{B_1/B, B_2/B, \ldots B_{K-1}/B\}$ and, hence, we cannot apply the result of [21, Theorem 1] directly to our problem.

Let

$$\beta^{(s)}(n, \tau_0) = 1 - \frac{1}{Bn} \sum_{i=1}^{Bn} \left(1 - p_i^{(n)}\right)^{n\tau_0}$$

be the fraction of documents in the cache. Here, $n\tau_0$ corresponds to the window size in [21].

We have

$$\begin{aligned} \beta^{(s)}(n, \tau_0) &= 1 - \frac{1}{Bn} \sum_{k=1}^{K} \sum_{i=1}^{b_k n} \big(1 - p^{(n)}_{B_{k-1}+i}\big)^{n\tau_0} \\ &= 1 - \frac{1}{Bn} \sum_{k=1}^{K} \sum_{i=1}^{b_k n} \big(1 - a_k p^{(n)}_{k,i}\big)^{n\tau_0}. \end{aligned} \tag{6}$$

We are interested in $\beta^{(s)} = \lim_{n\to\infty} \beta^{(s)}(n, \tau_0)$.

By applying the result in [21, Theorem 1], we find that

$$\lim_{n\to\infty} \frac{1}{Bn} \sum_{i=1}^{b_k n} \big(1 - a_k p^{(n)}_{k,i}\big)^{n\tau_0} = \frac{b_k}{B} \int_0^1 e^{-\tau_0 a_k B F_k'(x)/b_k} dx \tag{7}$$

for $k = 1, \ldots, K$, so that

$$\beta^{(s)} = 1 - \sum_{k=1}^{K} \frac{b_k}{B} \int_0^1 e^{-a_k F_k'(x) \tau_0 B/b_k} dx.$$

Equation (4.7) is derived in the same way.

Last, it follows from Theorems 2 and 4 in [21] that $\mu^{(s)}$ is the limiting aggregate miss probability under LRU as $n \to \infty$. □

K Strategy 3 outperforms strategy 2 when content providers have the same popularity distribution.

Proof. The optimization problems under strategies 2 and 3 are

$$\begin{aligned} \min_{\tau_1,\tau_2} \quad & \sum_{k=1}^{2}(a_{0,k}\mu_{0,k}^{(s2)}(\tau_k) + a_k\mu_k^{(s2)}(\tau_k)) \\ \text{s.t.} \quad & \sum_{k=1}^{2}(\beta_{0,k}^{(s2)}(\tau_k) + \beta_k^{(s2)}(\tau_k)) \le \beta, \\ & \beta_{0.k}^{(s2)},\, \beta_k^{(s2)} \ge 0, \quad k = 1, 2. \end{aligned}$$

and

$$\begin{aligned} \min_{\tau_k} \quad & \sum_{k=1}^{3} a_k\mu_k^{(s3)}(\tau_k) \\ \text{s.t.} \quad & \sum_{k=1}^{3} \beta_k^{(s3)}(\tau_k)\beta, \\ & \beta_k^{(s3)} \ge 0, \quad k = 1, 2, 3. \end{aligned} \tag{8}$$

where $\lambda_0 = \lambda_{0,1} + \lambda_{0.2}$, $a_{0,k} = \lambda_{0,k}/\sum_{k=1}^{3}\lambda_k$, and $a_k = \lambda_k/\sum_{k=1}^{3}\lambda_k$,

$$\mu_{0,k}^{(s2)}(\tau) = \int_0^1 F_0'(x)e^{-\frac{a_{0,k}(b_0+b_k)}{(a_{0,k}+a_k)b_0}F_0'(x)\tau}dx$$

$$\mu_k^{(s2)}(\tau) = \int_0^1 F_k'(x) e^{-\frac{a_{0,k}(b_0+b_k)}{(a_{0,k}+a_k)b_k} F_k'(x)\tau} dx$$
$$\beta_{0,k}^{(s2)}(\tau) = \frac{b_0}{2b_0+b_1+b_2}\big(1 - \int_0^1 e^{-\frac{a_{0,k}(b_0+b_k)}{(a_{0,k}+a_k)b_0} F_0'(x)\tau} dx\big)$$
$$\beta_k^{(s2)}(\tau) = \frac{b_k}{2b_0+b_1+b_2}\big(1 - \int_0^1 e^{-\frac{a_k(b_0+b_k)}{(a_{0,k}+a_k)b_k} F_0'(x)\tau} dx\big)$$

and

$$\mu_k^{(s3)}(\tau) = \int_0^1 F_k'(x) e^{-F_k'(x)\tau} dx$$
$$\beta_k^{(s3)}(\tau) = \frac{b_k}{b_0+b_1+b_2}\big(1 - \int_0^1 e^{-F_k'(x)\tau} dx\big)$$

We make two observations:

- $\mu_{0,k}^{(s2)}(\tau) = \mu_0^{(s3)}(\frac{a_{0,k}(2b_0+b_1+b_2)}{(a_{0,k}+a_k)b_0})\tau$, $k = 1, 2$,
- $\mu_0^{(s3)}(\tau)$ is a decreasing function of τ.

Let $\mu^{(s2)*}$ denote the minimum miss probability under strategy 2, which is achieved with τ_1^* and τ_2^*. Let $\mu^{(s3)}(\tau_1, \tau_2, \tau_3)$ denote the miss probability under strategy 3 where τ_k, $k = 1, 2, 3$ satisfy (8). Set $\tau_k = \frac{a_{0,k}(2b_0+b_1+b_2)}{(a_{0,k}+a_k)b_0}\tau_k^*$, $k = 1, 2$ for strategy 3 and allocate provider k a cache of size $\beta_k^{(s2)}$ under strategy 3 for its non-shared content. The aggregate miss probability for non-shared content is then the same under the two strategies and given by $\mu_1^{(s2)}(\tau_1^*) + \mu_2^{(s2)}(\tau_2^*)$.

Under strategy 2, the amount of shared content stored in the cache is $\beta_- = \beta_{0,1}^{(s2)*} + \beta_{0,2}^{(s2)*}$. We allocate a cache of that size to the shared content under strategy

3 and wlog assume that $\tau_1^* \geq \tau_2^*$ Themiss probability over all shared content under strategy 2 is

$$\sum_{k=1}^{2} \frac{a_{0,k}}{a_{0,1} + a_{0.2}} \mu_{0,k}^{(s2)}(\tau_k^*) \geq \mu_{0,k}^{(s2)}(\tau_1^*)$$
$$= \mu_0^{(s3)}(\tau_1)$$

Note that strategy 3 requires only a cache of size $\beta_{0,k}^{(s2)} < \beta_-$ to achieve a smaller miss probability for the shared content than strategy 2 can realize. Adding additional storage to the shared partition can only decrease the hit probability further, thus proving the theorem. □

L Cache Partitioning Problem Has a Unique Optimal Solution.

Proof. Let P and $\mathbf{C} = (C_1, \ldots, C_P)$ denote the number of partitions and the vector of partition sizes, respectively. The hit rate for content provider k in this case can be written as

$$h_k(\mathbf{C}) = \sum_{p=1}^{P} \sum_{i \in V_p} \lambda_{ik}(1 - e^{-\lambda_i T_p}),$$

where V_p denotes the set of files requested from partition p, and $\lambda_i = \sum_k \lambda_{ik}$ denotes the aggregate request rate for file i.

We can re-write the expression for h_k as the sum of the hit rates from each partition, since distinct files are requested from different partitions. We have

$$h_k(\mathbf{C}) = \sum_{p=1}^{P} h_{kp}(C_p),$$

where $h_{kp}(C_p)$ denotes the hit rate for files requested from partition p from content provider k. Since h_{kp} is assumed to be a concave increasing function of C_p, h_k is sum of concave functions and hence is also concave. □

M Partitioning and Probabilistic Routing is Sub-optimal.

Lemma 5. *Partitioning a cache, and probabilistically routing content requests to different partitions is sub-optimal.*

Proof. Assume we partition the cache into two slices of size C_1 and C_2, and route requests to partition one with probability p, and with probability $1-p$ route to partition two. Let h_P denote the hit rate obtained by partitioning the cache. From concavity of $h(C)$ we have

$$\begin{aligned} h_P &= ph(C_1) + (1-p)h(C_2) \\ &\leq ph(C) + (1-p)h(C) = h(C). \end{aligned}$$

□

N Gradient Ascent Algorithm for Cache Partitioning Converges to Optimal Solution.

Proof. We first note that since $W(\mathbf{C})$ is a strictly concave function, it has a unique maximizer $\mathbf{C}^*$. Moreover $V(\mathbf{C}) = W(\mathbf{C}^*) - W(\mathbf{C})$ is a non-negative function and equals zero only at $\mathbf{C} = \mathbf{C}^*$. Differentiating $V(\cdot)$ with respect to time we obtain

$$\dot{V}(\mathbf{C}) = \sum_k \frac{\partial V}{\partial h_k}\dot{h_k} = -\sum_k \left(U_k'(h_k) - P'(\sum_k C_k - C)\frac{\partial C_k}{\partial h_k} \right) \dot{h_k}.$$

For $\dot{h_k}$ we have

$$\dot{h_k} = \frac{\partial h_k}{\partial C_k}\dot{C_k}.$$

From the controller for C_k we have

$$\dot{C_k} = \gamma_k \left(U_k'(h_k)\frac{\partial h_k}{\partial C_k} - P'(\sum_k C_k - C) \right).$$

Since $\frac{\partial h_k}{\partial C_k} \geq 0$, we get

$$\dot{V}(\mathbf{C}) = -\sum_k \gamma_k \frac{\partial h_k}{\partial C_k} \left(U_k'(h_k) - P'(\sum_k C_k - C)\frac{\partial C_k}{\partial h_k} \right)^2 \leq 0.$$

Therefore, $V(\cdot)$ is a Lyapunov function, and the system state will converge to $\mathbf{C}^*$ starting from any initial condition. A description of Lyapunov functions and their applications can be found in [67]. □

BIBLIOGRAPHY

[1] Cisco visual networking index: Forecast and methodology, 2014-2019. White paper, May 2015.

[2] Andrews, J.G., Claussen, H., Dohler, M., Rangan, S., and Reed, M.C. Femtocells: Past, present, and future. *IEEE Journal on Selected Areas in Communications 30*, 3 (April 2012), 497–508.

[3] Azimdoost, B., Westphal, C., and Sadjadpour, H.R. On the throughput capacity of information-centric networks. In *International Teletraffic Congress (ITC)* (September 2013), pp. 1–9.

[4] Baev, Ivan, Rajaraman, Rajmohan, and Swamy, Chaitanya. Approximation algorithms for data placement problems. *SIAM Journal on Computing 38*, 4 (2008), 1411–1429.

[5] Bartle, Robert G. *The elements of real analysis (2nd ed.)*. Wiley New York, 1976.

[6] Berger, Daniel S, Gland, Philipp, Singla, Sahil, and Ciucu, Florin. Exact analysis of ttl cache networks. *Performance Evaluation 79* (2014), 2–23.

[7] Bianchi, Giuseppe, Detti, Andrea, Caponi, Alberto, and Blefari Melazzi, Nicola. Check before storing: What is the performance price of content integrity verification in lru caching? *SIGCOMM Computer Communication Review 43* (July 2013), 59–67.

[8] Borst, S., Gupta, V., and Walid, A. Distributed caching algorithms for content distribution networks. In *INFOCOM* (March 2010), pp. 1–9.

[9] Calinescu, Gruia, Chekuri, Chandra, Pál, Martin, and Vondrák, Jan. Maximizing a submodular set function subject to a matroid constraint (extended abstract). In *IPCO* (2007), pp. 182–196.

[10] Carofiglio, G., Gehlen, V., and Perino, D. Experimental evaluation of memory management in content-centric networking. In *ICC* (June 2011), pp. 1–6.

[11] Chai, W.K, He, D, Psaras, I, and Pavlou, G. Cache less for more in information-centric networks. In *IFIP Networking* (2012).

[12] Che, H, Wang, Z, and Tung, Y. Analysis and design of hierarchical web caching systems. In *INFOCOM* (2001).

[13] Chu, Weibo, Dehghan, Mostafa, Towsley, Don, and Zhang, Zhi-Li. On allocating cache resources to content providers. In *ACM ICN* (2016), pp. 154–159.

[14] Cieliebak, Mark, Stephan Eidenbenz Aris Pagourtzis, and Schlude, Konrad. On the complexity of variations of equal sum subsets. *Nordic Journal of Computing 14*, 3 (2008), 151–172.

[15] Dai, Huichen, Liu, Bin, Chen, Yan, and Wang, Yi. On pending interest table in named data networking. In *ANCS* (2012), pp. 211–222.

[16] Dehghan, Mostafa, Jiang, Bo, Dabirmoghaddam, Ali, and Towsley, Don. On the analysis of caches with pending interest tables. In *ICN* (2015), pp. 69–78.

[17] Dehghan, Mostafa, Massoulie, Laurent, Towsley, Don, Menasche, Daniel, and Tay, Y. C. A utility optimization approach to network cache design. In *INFOCOM* (2016).

[18] Dehghan, Mostafa, Seetharam, Anand, He, Ting, Salonidis, Theodoros, Kurose, Jim, and Towsley, Don. Optimal caching and routing in hybrid networks. In *IEEE MILCOM 2014* (October 2014), pp. 1072–1078.

[19] Dehghan, Mostafa, Seetharam, Anand, Jiang, Bo, He, Ting, Salonidis, Theodoros, Kurose, Jim, Towsley, Don, and Sitaraman, Ramesh. On the complexity of optimal routing and content caching in heterogeneous networks. In *INFOCOM* (2015).

[20] Eryilmaz, Atilla, and Srikant, R. Fair resource allocation in wireless networks using queue-length-based scheduling and congestion control. *IEEE/ACM Transaction on Networking 15*, 6 (December 2007), 1333–1344.

[21] Fagin, Ronald. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences 14*, 2 (1977), 222–250.

[22] Feijer, Diego, and Paganini, Fernando. Stability of primal–dual gradient dynamics and applications to network optimization. *Automatica 46*, 12 (2010), 1974–1981.

[23] Feldman, Michal, and Chuang, John. Service differentiation in web caching and content distribution. In *IASTED Int. Conf. on Communications and Computer Networks* (2002).

[24] Ferragut, Andrés, Rodriguez, Ismael, and Paganini, Fernando. Optimizing ttl caches under heavy-tailed demands. In *ACM SIGMETRICS* (2016), pp. 101–112.

[25] Fofack, N. C., Dehghan, M., Towsley, D., Badov, M., and Goeckel, D. L. On the performance of general cache networks. In *ValueTools* (December 2014).

[26] Fofack, Nicaise Choungmo, Nain, Philippe, Neglia, Giovanni, and Towsley, Don. Performance evaluation of hierarchical ttl-based cache networks. *Computer Networks 65* (2014), 212–231.

[27] Fricker, Christine, Robert, Philippe, and Roberts, James. A versatile and accurate approximation for lru cache performance. In *ITC* (2012).

[28] Fridgeirsdottir, Kristin, and Chiu, Sam. A note on convexity of the expected delay cost in single-server queues. *Operations research 53*, 3 (2005), 568–570.

[29] Giovanidis, Anastasios, and Avranas, Apostolos. Spatial multi-lru caching for wireless networks with coverage overlaps. In *SIGMETRICS* (2016), ACM, pp. 403–405.

[30] Gitzenis, S., Paschos, G.S., and Tassiulas, L. Asymptotic laws for joint content replication and delivery in wireless networks. *IEEE Transactions on Information Theory 59*, 5 (May 2013), 2760–2776.

[31] Golrezaei, N, Shanmugam, K, Dimakis, A, Molisch, A, and Caire, G. Femtocaching: Wireless video content delivery through distributed caching helpers. In *INFOCOM* (2012).

[32] Hoffman, Alan J, and Kruskal, Joseph B. Integral boundary points of convex polyhedra. In *50 Years of Integer Programming 1958-2008*. Springer, 2010, pp. 49–76.

[33] Hoteit, S., El Chamie, M., Saucez, D., and S., Secci. On fair network cache allocation to content providers. *INRIA Technical Report* (2015).

[34] Huang, Cheng, Wang, Angela, Li, Jin, and Ross, Keith W. Understanding hybrid cdn-p2p: Why limelight needs its own red swoosh. In *NOSSDAV* (May 2008), pp. 75–80.

[35] Huang, L., and Neely, M. J. Utility optimal scheduling in energy-harvesting networks. *IEEE/ACM Transactions on Networking 21*, 4 (Aug 2013), 1117–1130.

[36] Jaeyeon, J., Berger, A. W., and Balakrishnan, H. Modeling ttl-based internet caches. In *INFOCOM* (March 2003), pp. 417–426.

[37] Jiang, Wenjie, Ioannidis, Stratis, Massoulié, Laurent, and Picconi, Fabio. Orchestrating massively distributed cdns. In *CoNEXT* (2012), pp. 133–144.

[38] Karlin, and Samuel. *A first course in stochastic processes*, 2 ed. Academic press, 1975.

[39] Kelly, Frank. Charging and rate control for elastic traffic. *European Transactions on Telecommunications 8* (1997), 33–37.

[40] Kelly, Frank P, Maulloo, Aman K, and Tan, David KH. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society* (1998), 237–252.

[41] Kelly, Terence, Chan, Yee Man, Jamin, Sugih, and MacKie-Mason, Jeffrey K. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value. In *Web Caching Workshop* (1999).

[42] Kim, Seongbeom, Chandra, Dhruba, and Solihin, Yan. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *IEEE PACT* (2004), pp. 111–122.

[43] Ko, Bong-Jun, Lee, Kang-Won, Amiri, K., and Calo, S. Scalable service differentiation in a shared storage cache. In *ICDCS* (May 2003), pp. 184–193.

[44] Liu, Z, Nain, P Niclausse, N, and Towsley, D. Static caching of web servers. In *Multimedia Computing and Networking Conference* (1998).

[45] Lu, Ying, Abdelzaher, T.F., and Saxena, Avneesh. Design, implementation, and evaluation of differentiated caching services. *IEEE Transactions on Parallel and Distributed Systems 15*, 5 (May 2004), 440–452.

[46] Ma, Richard TB, and Towsley, Don. Cashing in on caching: On-demand contract design with linear pricing. In *CoNext* (December 2015).

[47] Martina, V., Garetto, M., and Leonardi, E. A unified approach to the performance analysis of caching systems. In *INFOCOM* (April 2014), pp. 2040–2048.

[48] Mo, Jeonghoon, and Walrand, Jean. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking 8*, 5 (2000), 556–567.

[49] Naveen, K.P., Massoulie, Laurent, Baccelli, Emmanuel, Carneiro Viana, Aline, and Towsley, Don. On the interaction between content caching and request assignment in cellular cache networks. In *Workshop on All Things Cellular: Operations, Applications and Challenges* (2015), pp. 37–42.

[50] Neammanee, Kritsana. A refinement of normal approximation to poisson binomial. *International Journal of Mathematics and Mathematical Sciences 2005*, 5 (2005), 717–728.

[51] Neely, Michael J. *Stochastic Network Optimization with Application to Communication and Queueing Systems*. Morgan and Claypool Publishers, 2010.

[52] Neely, M.J., Modiano, E., and Rohrs, C.E. Dynamic power allocation and routing for time varying wireless networks. In *INFOCOM* (March 2003), pp. 745–755.

[53] Nemhauser, George L, and Wolsey, Laurence A. *Integer and combinatorial optimization*, vol. 18. Wiley New York, 1988.

[54] Nygren, E, Sitaraman, R, and Sun, J. The akamai network: a platform for high-performance internet application. *ACM SIGOPS Operating Systems Review 44*, 3 (2010).

[55] Perino, Diego, and Varvello, Matteo. A reality check for content centric networking. In *SIGCOMM Workshop on Information-centric Networking* (2011), pp. 44–49.

[56] Polyak, Boris T. *Introduction to optimization*. 1987.

[57] Poularakis, Konstantinos, Iosifidis, George, and Tassiulas, Leandros. Approximation caching and routing algorithms for massive mobile data delivery. In *Globecom* (2013).

[58] Psaras, I, Chai, W.K, and Pavlou, G. Coordinating in-network caching in content-centric networks: Model and analysis. In *In ACM SIGCOMM Workshop on Information-Centric Networking* (2012).

[59] Qureshi, Moinuddin K, and Patt, Yale N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM International Symposium on Microarchitecture* (2006), pp. 423–432.

[60] Rosensweig, E, Kurose, J, and Towsley, D. Approximate models for general cache networks. In *IEEE INFOCOM* (2010).

[61] Sandvine. Global Internet Phenomena Report - 2H2014.

[62] Schrijver, Alexander. *Combinatorial optimization: polyhedra and efficiency*, vol. 24. Springer, 2003.

[63] Shanmugam, K., Golrezaei, N., Dimakis, A.G., Molisch, A.F., and Caire, G. Femtocaching: Wireless content delivery through distributed caching helpers. *IEEE Transactions on Information Theory 59*, 12 (December 2013), 8402–8413.

[64] Sharma, Abhigyan, Venkataramani, Arun, and Sitaraman, Ramesh K. Distributing content simplifies isp traffic engineering. In *SIGMETRICS* (June 2013), pp. 229–242.

[65] Sourlas, Vasilis, Flegkas, Paris, and Tassiulas, Leandros. Cache-aware routing in information-centric networks. In *IFIP/IEEE International Symposium on Integrated Network Management* (2013).

[66] Sourlas, Vasilis, Gkatzikis, Lazaros, Flegkas, Paris, and Tassiulas, Leandros. Distributed cache management in information-centric networks. *IEEE Transactions on Network and Service Management 10*, 3 (2013).

[67] Srikant, Rayadurgam, and Ying, Lei. *Communication Networks: An Optimization, Control, and Stochastic Networks Perspective*. Cambridge University Press, 2013.

[68] Stich, Sebastian U, Muller, CL, and Gartner, B. Optimization of convex functions with random pursuit. *SIAM Journal on Optimization 23*, 2 (2013), 1284–1309.

[69] Suh, G Edward, Rudolph, Larry, and Devadas, Srinivas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing 28*, 1 (2004), 7–26.

[70] Sundarrajan, A., Kasbekar, M., and Sitaraman, R. Energy-efficient disk caching for content delivery. In *ACM e-Energy* (2016).

[71] Taghizadeh, Mahmoud, Micinski, Kristopher, Ofria, Charles, Torng, Eric, and Biswas, Santosh. Distributed cooperative caching in social wireless networks. *IEEE Transactions on Mobile Computing 12*, 6 (2013), 1037–1053.

[72] Tang, Bin, and Gupta, Himanshu. Cache placement in sensor networks under an update cost constraint. *Journal of Discrete Algorithms 5*, 3 (2007), 422–435.

[73] Tassiulas, L., and Ephremides, Anthony. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. *IEEE Transactions on Automatic Control 37*, 12 (Dec 1992), 1936–1948.

[74] Tsilopoulos, C., Xylomenos, G., and Thomas, Y. Reducing forwarding state in content-centric networks with semi-stateless forwarding. In *INFOCOM, 2014 Proceedings IEEE* (April 2014), pp. 2067–2075.

[75] Varvello, M., Perino, D., and Linguaglossa, L. On the design and implementation of a wire-speed pending interest table. In *INFOCOM workshops* (April 2013), pp. 369–374.

[76] Virgilio, Matteo, Marchetto, Guido, and Sisto, Riccardo. Pit overload analysis in content centric networks. In *SIGCOMM Workshop on Information-centric Networking* (2013), pp. 67–72.

[77] Wang, Yi, He, Keqiang, Dai, Huichen, Meng, Wei, Jiang, Junchen, Liu, Bin, and Chen, Yan. Scalable name lookup in ndn using effective name component encoding. In *ICDCS* (2012), pp. 688–697.

[78] West, Douglas Brent. *Introduction to graph theory (2nd ed.)*. Chapter 3, Prentice Hall, 2001.

[79] Xie, Mengjun, Widjaja, I., and Wang, Haining. Enhancing cache robustness for content-centric networking. In *INFOCOM* (March 2012), pp. 2426–2434.

[80] You, Wei, Mathieu, B., Truong, P., Peltier, J., and Simon, G. Dipit: A distributed bloom-filter based pit table for ccn nodes. In *ICCCN* (July 2012), pp. 1–7.

[81] Yuan, Haowei, and Crowley, P. Scalable pending interest table design: From principles to practice. In *INFOCOM* (April 2014), pp. 2049–2057.

[82] Zerfos, P., Srivatsa, M., Yu, H., Dennerline, D., Franke, H., and Agrawal, D. Platform and applications for massive-scale streaming network analytics. *IBM Journal for Research and Development: Special Edition on Massive Scale Analytics 57* (May 2013), 1–11.

[83] Zhang, Guoqiang, Li, Yang, and Lin, Tao. Caching in information centric networking: A survey. *Computer Networks 57*, 16 (2013), 3128–3141.