March 2020

# Towards Optimized Traffic Provisioning and Adaptive Cache Management for Content Delivery

Aditya Sundarrajan

# TOWARDS OPTIMIZED TRAFFIC PROVISIONING AND ADAPTIVE CACHE MANAGEMENT FOR CONTENT DELIVERY

A Dissertation Presented

by

ADITYA SUNDARRAJAN

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 2020

College of Information and Computer Sciences

# TOWARDS OPTIMIZED TRAFFIC PROVISIONING AND ADAPTIVE CACHE MANAGEMENT FOR CONTENT DELIVERY

A Dissertation Presented

by

ADITYA SUNDARRAJAN

Approved as to style and content by:

Ramesh K. Sitaraman, Chair

Prashant Shenoy, Member

Donald Towsley, Member

Michael Zink, Member

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

# ACKNOWLEDGMENTS

This dissertation has been a few years in the making. There are several people who have helped me along the way, and I am very grateful for their support. First and foremost, I would like to thank my advisor Ramesh K. Sitaraman. Ramesh has been a great source of inspiration and a fantastic mentor over the years. Ramesh has taught me the importance of conducting meaningful and impactful research that has far-reaching consequences. This has shaped my own views on research that I hope to build upon. I am also grateful to Ramesh for helping me foster collaborations with researchers from other academic institutions and the industry to identify and solve real world challenges.

I am also grateful to my collaborators who I have had the pleasure of working with over the years. I would like to thank the many engineers I have spoken to and worked with at Akamai Technologies on the challenges faced in content delivery and the impacts of addressing these challenges. I would specifically like to thank Mangesh Kasbekar and Mingdong Feng who have been my collaborators on several projects. Their insights on how large globally distributed systems work have helped me develop practical and deployable solutions. I would also like to thank Soumya Basu, Sanjay Shakkottai, Javad Ghaderi, Vadim Kirilin and Sergey Gorinsky for their collaboration on different projects that have contributed to this dissertation.

I want to thank my committee members Prashant Shenoy, Don Towsley and Michael Zink for their valuable insights and feedback on ways to improve this dissertation. I would also like to thank my good friends and lab mates for making my time at UMass and Amherst more memorable. I specifically want to thank Prateek Sharma, Anirudh Sabnis, Kevin Spiteri and Sohaib Ahmad for their support and

friendship. The computer science graduate program at UMass has some of the best administrative staff I have ever worked with and I specifically want to thank Leeanne Leclerc, Eileen Hamel, Karren Sacco and Barbara Sutherland for seamlessly taking care of all the paperwork and administrivia.

Finally, I am very grateful to my parents, sister and grandparents for their whole-hearted support and encouragement.

# ABSTRACT

## TOWARDS OPTIMIZED TRAFFIC PROVISIONING AND ADAPTIVE CACHE MANAGEMENT FOR CONTENT DELIVERY

FEBRUARY 2020

ADITYA SUNDARRAJAN

B.E., ANNA UNIVERSITY

M.S., THE UNIVERSITY OF ARIZONA

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Ramesh K. Sitaraman

Content delivery networks (CDNs) deploy hundreds of thousands of servers around the world to cache and serve trillions of user requests every day for a diverse set of content such as web pages, videos, software downloads and images. In this dissertation, we propose algorithms to provision traffic across cache servers and manage the content they host to achieve performance objectives such as maximizing the cache hit rate, minimizing the bandwidth cost of the network and minimizing the energy consumption of the servers.

Traffic provisioning is the process of determining the set of content domains hosted on the servers. We propose footprint descriptors that effectively capture the popularity characteristics and caching performance of different content classes. We also propose a footprint descriptor calculus that can be used to decide how content should

be mixed or partitioned to efficiently provision traffic. To automate traffic provisioning, we propose optimization models to provision traffic such that the cache miss traffic from the network is minimized without overloading the servers. We find that such optimization models produce significant reductions in the cache miss traffic when compared with traffic provisioning algorithms in use today.

Cache management is the process of deciding how content is cached in the servers of a CDN. We propose TTL-based caching algorithms that provably achieve performance targets specified by a CDN operator. We show that the proposed algorithms converge to the target hit rate and target cache size with low error. Finally, we propose cache management algorithms to make the servers energy-efficient using disk shutdown. We find that disk shutdown is well suited for CDN servers and provides energy savings without significantly impacting cache hit rates.

# TABLE OF CONTENTS

**Page**

**ACKNOWLEDGMENTS** . . . . . **iv**

**ABSTRACT** . . . . . **vi**

**LIST OF TABLES** . . . . . **xiii**

**LIST OF FIGURES** . . . . . **xv**

**CHAPTER**

**1. INTRODUCTION** . . . . . **1**

1.1 The role of caching in CDNs . . . . . 1

1.1.1 Performance and cost metrics of a CDN . . . . . 2

1.2 Traffic provisioning and cache management in CDNs . . . . . 4

1.2.1 Traffic provisioning . . . . . 4
1.2.2 Cache management . . . . . 5

1.3 Challenges in traffic provisioning and cache management and proposed solutions . . . . . 5

1.3.1 Predicting caching characteristics of traffic mixes using footprint descriptors . . . . . 5
1.3.2 Midgress-aware traffic provisioning . . . . . 6
1.3.3 Adaptive cache management using TTL-based caching . . . . . 7
1.3.4 Energy-efficient caching using disk shutdown . . . . . 8

1.4 Dissertation outline . . . . . 9

**2. TRAFFIC MIXING USING FOOTPRINT DESCRIPTORS** . . . . . **10**

2.1 Traffic class characteristics . . . . . 14

2.1.1 Trace collection . . . . . 14
2.1.2 Analysis of traffic classes . . . . . 15

2.2 Theory of footprint descriptors . . . . . 18

2.2.1 Footprint descriptors (FD) . . . . . 18
2.2.2 Estimating cache properties from footprint descriptors . . . . . 19
2.2.3 A calculus of footprint descriptors . . . . . 20

2.2.3.1 Addition . . . . . 21
2.2.3.2 Subtraction . . . . . 24
2.2.3.3 Scaling . . . . . 25

2.2.4 A simpler footprint descriptor (SFD) . . . . . 26

2.3 Validation of footprint calculus . . . . . 27

2.3.1 Experimental evaluation . . . . . 29

2.4 Applying footprint descriptors in a production CDN . . . . . 33

2.4.1 Traffic mix evaluation service . . . . . 33

2.4.1.1 Estimating space requirement of a traffic mix . . . . . 33
2.4.1.2 Predicting the outcome of traffic mixing in a given server . . . . . 35
2.4.1.3 Splitting domains in a traffic class . . . . . 36

2.4.2 Hit rate targets with cache partitioning . . . . . 37
2.4.3 Parallelizing the computation of FDs . . . . . 39

2.5 Related work . . . . . 41
2.6 Conclusion . . . . . 43

**3. MIDGRESS-AWARE TRAFFIC PROVISIONING . . . . . 44**

3.1 Our optimization solution for traffic provisioning . . . . . 48

3.1.1 Modeling cache eviction and midgress . . . . . 49

3.1.1.1 Eviction age equality . . . . . 50

3.1.2 Formulation of our optimization model . . . . . 51
3.1.3 Solving the optimization model OPT . . . . . 56

3.2 Traffic provisioning heuristics . . . . . 56

3.2.1 Midgress-unaware baseline . . . 57
3.2.2 Midgress-aware local search . . . 58

3.3 Experimental evaluation . . . 59

3.3.1 Metro-level traffic provisioning . . . 62

3.3.1.1 Understanding how traffic provisioning can impact midgress . . . 63

3.3.2 Cluster-level traffic provisioning . . . 65
3.3.3 Robustness to variations in cache management policy . . . 66

3.4 Extensions of midgress-aware traffic provisioning . . . 67

3.4.1 Minimum redundancy guarantee . . . 68

3.4.1.1 Experimental evaluation . . . 69

3.4.2 Maximum cache miss rate guarantee . . . 70

3.4.2.1 Experimental evaluation . . . 71

3.4.3 Traffic provisioning in partitioned caches . . . 71

3.4.3.1 Modeling and implementing traffic provisioning for partitioned caches . . . 72
3.4.3.2 Experimental evaluation . . . 73

3.5 Related work . . . 74
3.6 Conclusion . . . 75

**4. ADAPTIVE CACHE MANAGEMENT USING TTL-BASED CACHING . . . 77**

4.1 TTL-based caching algorithms . . . 79

4.1.1 d-TTL . . . 80
4.1.2 f-TTL . . . 80

4.1.2.1 Algorithm description . . . 81
4.1.2.2 Achieving the target cache size . . . 82
4.1.2.3 TTL update rules . . . 82

4.2 Implementation of d-TTL and f-TTL . . . 84
4.3 Experimental evaluation . . . 85

4.3.1 Hit rate performance of d-TTL and f-TTL . . . 86
4.3.2 Convergence of d-TTL and f-TTL . . . 87
4.3.3 Accuracy of d-TTL and f-TTL . . . 89
4.3.4 Sensitivity and robustness of d-TTL and f-TTL . . . 91
4.3.5 Effect of target normalized size on f-TTL . . . 93

4.4 Related Work . . . 94
4.5 Conclusion . . . 95

**5. ENERGY-EFFICIENT CACHING USING DISK SHUTDOWN . . . 96**

5.1 Cache management schemes . . . 101

5.1.1 A typical algorithm without disk shutdown . . . 101
5.1.2 Energy-efficient cache management . . . 102
5.1.3 An ideal energy-efficient variant of LRU . . . 103
5.1.4 Evaluation methodology . . . 104

5.1.4.1 Content request traces . . . 104
5.1.4.2 Cache simulator for disk shutdown . . . 106

5.2 Cache sizing algorithms . . . 107

5.2.1 Experimental evaluation . . . 111

5.3 Disk shutdown algorithms . . . 113

5.3.1 Experimental evaluation . . . 114

5.4 Content placement & eviction algorithms . . . 117

5.4.1 Experimental evaluation . . . 118
5.4.2 Disk power cycles and impact on lifetimes . . . 119

5.5 Understanding cluster hit rates . . . 119

5.5.1 Experimental evaluation . . . 120

5.6 Related work . . . 122
5.7 Conclusion . . . 123

**6. SUMMARY AND FUTURE WORK . . . 125**

6.1 Future work . . . 126

6.1.1 Traffic provisioning of hierarchical caches . . . 126

6.1.2 Modeling server provisioning . . . . . 127
6.1.3 Machine learning based caching . . . . . 127

**APPENDICES**

**A. SOLVING THE OPTIMIZATION MODEL FOR MIDGRESS-AWARE TRAFFIC PROVISIONING . . . . . 129**
**B. DISK POWER MODEL . . . . . 135**

**BIBLIOGRAPHY . . . . . 138**

# LIST OF TABLES


**Table** **Page**

2.1 Characteristics of the chosen traffic class traces. . . . . 15

2.2 Characteristics of the $2^{nd}$ set of traces. . . . . 28

2.3 Characteristics of the $3^{rd}$ set of traces. . . . . 28

2.4 Average prediction error of the baseline algorithm vs. the FD calculus. . . . . 29

2.5 Production validation. . . . . 32

3.1 Input parameters of optimization model. . . . . 53

3.2 Output parameters of optimization model. . . . . 54

3.3 Traffic class characteristics . . . . . 61

4.1 Comparison of target hit rate and average cache size achieved by d-TTL and f-TTL with Fixed-TTL and LRU. . . . . 90

4.2 Impact of exponential changes in constant step size $\eta$ on the performance of d-TTL (robustness analysis of d-TTL). . . . . 91

4.3 Impact of linear changes in constant step size $\eta = 0.01$ on the performance of d-TTL (sensitivity analysis of d-TTL). . . . . 91

4.4 Impact of exponential changes in constant step size $\eta^s$ on the performance of f-TTL (robustness analysis of f-TTL). . . . . 92

4.5 Impact of linear changes in constant step size $\eta^s =$ 1e-9 on the performance of f-TTL (sensitivity analysis of f-TTL). . . . . 92

4.6 Impact of normalized cache size target on the performance of f-TTL. . . . . 93

5.1 Algorithms for energy-efficient cache management. The starred algorithms are simple options that we use as a baseline that we improve upon. . . . . 102

5.2 Characteristics of content request traces from the Akamai cluster. . . . . 104

B.1 Disk power consumption. . . . . 136

# LIST OF FIGURES

**Figure** **Page**


1.1 Content is served from the cache of a CDN server, from one of its peers within the same cluster, or from the content provider's origin. . . . . 2

1.2 The traffic provisioning and cache management systems work together to maximize the cache hit rate of the CDN. . . . . 4

2.1 Popularity distribution of the 4 traffic classes. . . . . 16

2.2 Object size distribution of the 4 traffic classes. . . . . 16

2.3 Hit rate curves of the 4 traffic classes. . . . . 17

2.4 Hit rate curves of the traffic mix in Table 2.2 before and after scaling the download traffic class by a factor of 20. . . . . 31

2.5 Hit rate curves of traffic classes in Table 2.2. . . . . 31

2.6 Addition operation on web and download traffic classes in Table 2.1. . . . . 34

2.7 Pre-addition and post-addition HRCs of web and download traffic classes in Table 2.1. . . . . 36

2.8 Subtracting a subset of domains from the video traffic class in Table 2.1. . . . . 37

2.9 Map-reduce framework to parallelize footprint descriptor computation. . . . . 40

3.1 MRCs of traffic classes $TC_1$, $TC_2$ and $TC_3$. . . . . 46

3.2 MRCs of two traffic classes in a metro area. . . . . 52

3.3 Eviction age functions of two traffic classes in a metro area. The eviction age functions indicate how traffic classes compete for shared cache space. . . . . 53

3.4 MRC of OPT, local search and baseline fit algorithms. . . . . 62

3.5 Average miss rate of each traffic class in a metro area of cache size 100 TB. . . . . 63

3.6 Average number of sites each traffic class is assigned to in a metro area of cache size 100 TB. . . . . 64

3.7 Increasing minimum redundancy increases the overall midgress of the metro area. . . . . 69

3.8 MRC of OPT and baseline fit on shared and partitioned caches. . . . . 73

4.1 Hit rate curve for object hit rates. . . . . 86

4.2 Hit rate curve for byte hit rates. . . . . 86

4.3 Object hit rate convergence over time for d-TTL; target hit rate=60%. . . . . 88

4.4 Object hit rate convergence over time for f-TTL; target hit rate=60%. . . . . 88

4.5 Byte hit rate convergence over time for d-TTL; target hit rate=60%. . . . . 89

4.6 Byte hit rate convergence over time for f-TTL; target hit rate=60%. . . . . 89

5.1 Hit rate decreases significantly when disks are shut down using a simple baseline scheme. . . . . 99

5.2 Popularity of content accessed by users on a CDN server. . . . . 100

5.3 Content traffic served to users from our cluster (in Gbps), averaged hourly. . . . . 105

5.4 A large fraction of the requests are for a small fraction of the objects. . . . . 105

5.5 The architectural components of a content cache that uses disk shutdown. . . . . 106

5.6 The `Hybrid` cache sizing algorithm. . . . . 108

5.7 Hit rate curve (HRC) shows the relation between cache size and hit rate. . . . . 110

5.8 Comparing the hit rate performance of the different cache sizing algorithms with the hit rate of `NOOFF` that does not shut down disks. . . . . 112

5.9 The storage-based cache sizing algorithm occasionally overloads the disks, since it does not factor in disk load. This deficiency can be corrected with a hybrid scheme. . . . . 113

5.10 `Fixed` disk shutdown provides a better energy-performance tradeoff than `Random` disk shutdown. . . . . 116

5.11 `Fixed` replicates content less than `Random` resulting in a more efficient use of the cache space. . . . . 116

5.12 Energy-performance tradeoff of content placement & eviction algorithms. . . . . 119

5.13 Energy-performance tradeoff of disk shutdown within a cluster. . . . . 121

5.14 Cluster hit rate provide a better energy-performance tradeoff because of content replication across servers. . . . . 122

# CHAPTER 1

# INTRODUCTION

Content delivery networks (CDNs) cache and serve trillions of user requests every day for a diverse set of content such as web pages, videos, software downloads, images and applications, among others. Content providers such as web portals, SaaS application providers, e-commerce sites, news outlets, media companies, social networks, and movie distribution services use CDNs to host and deliver their content. CDNs are now ubiquitous and are key to the functioning of the Internet, as most of the content accessed by users are served by such networks. It is estimated that 71% of all online content will be delivered by CDNs in 2021 up from 52% in 2016 [62].

A large CDN has a deployed network of hundreds of thousands of cache servers distributed throughout the world. The servers are deployed in *clusters* that are located in over a thousand data centers around the world. When a user requests content that is hosted on the CDN, the request is mapped to a proximal server in the network (see Figure 1.1). If the server has the requested content in its cache (i.e., a "cache hit"), it is delivered to the user. If the content is not available in the sever, it is fetched from one of its peers within the cluster (i.e., a "cluster hit") and served to the user. If the content is not available in the network (i.e., a "cache miss"), it is fetched from the content provider's origin that has the original copy of the content.

## 1.1 The role of caching in CDNs

CDNs efficiently deliver content by caching the requested objects at the closest edge servers. A cache hit is highly desirable because user requests do not traverse the

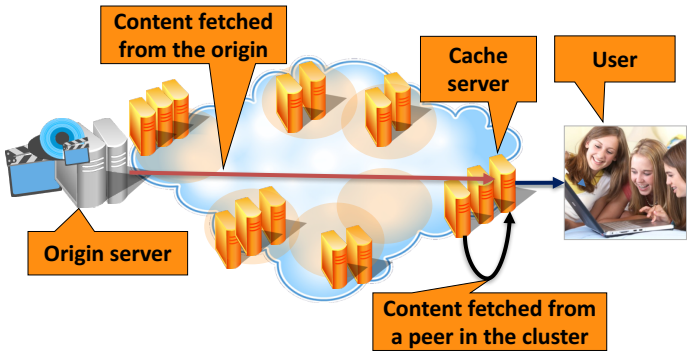


Figure 1.1: Content is served from the cache of a CDN server, from one of its peers within the same cluster, or from the content provider's origin.

high latency WAN to retrieve content from the origin server that might be far away. While a cache hit is the best case scenario, a cluster hit isn't much worse. During a cluster hit, the response time for serving the object is increased due to the additional overhead of fetching content from a peer. This could increase the latency to the order of several hundred microseconds to a few milliseconds but is still much better than a cache miss which could add hundreds of milliseconds of latency to each request. A cluster hit is therefore still desirable, and a cache miss is the least desirable.

### 1.1.1 Performance and cost metrics of a CDN

A CDN is designed to optimize several performance and cost metrics. We discuss three important metrics that are relevant to this dissertation.

*1) End-user latency:* End-user latency is the time elapsed between requesting content from a CDN and receiving a response at the end-user. Users experience a small latency during a cache hit. On the other hand, during a cache miss, the request has to traverse the high latency WAN link to retrieve content from the origin servers.

*2) Bandwidth cost:* Bandwidth cost of a CDN has two parts. The egress bandwidth cost is the cost of serving egress traffic from the edge servers to the users. This is not a cost overhead for the CDN and is paid by the content provider. On the other hand,

the midgress bandwidth cost is the cost due to *midgress traffic* which is the cache miss traffic from the edge servers to the origin servers. This is a cost overhead for the CDN and reducing the midgress bandwidth cost can lead to significant savings in the operating expenses of the CDN.

*3) Energy cost:* Energy cost is the cost associated with powering and running the servers that host the requested content and the networking equipment that route requests within the data center. The energy cost also includes the cost of cooling the servers and networking equipment, to prevent overheating due to continuous use. The energy cost constitutes a significant fraction of the operating expenses of a CDN and any reduction will result in large savings. Moreover, energy reduction also reduces the carbon emissions which is good for the environment.

A fourth metric that relates the three performance and cost metrics described above is the *cache hit rate*. There are two notions of cache hit rate. The *object hit rate (OHR)* is the percentage of requests that are found in cache. The *byte hit rate (BHR)* is the object hit rate weighted by the object size. A larger cache hit rate leads to an increased *origin offload ratio* which is the ratio of the total traffic served to users to the traffic served by the origin servers. An increased origin offload ratio ensures a smaller end-user latency due to fewer cache misses and reduced load at the origin. A larger cache hit rate also reduces the midgress traffic which in turn reduces the midgress bandwidth cost. On the other hand, a large number of active servers are required to achieve high cache hit rates. This could lead to an increase in the energy cost for the CDN. Hence, optimizing the energy cost presents a cost-performance tradeoff that a CDN operator should take into consideration. *From multiple perspectives, cache hits at the edge servers are highly desirable and maximizing the cache hit rate while being cognizant of its impact on different performance and cost metrics is a recurring theme throughout this dissertation.*

## 1.2 Traffic provisioning and cache management in CDNs

When users request content that is hosted on a CDN, the requests are classified into *traffic classes*. A traffic class is a collection of domains that host a specific type of content belonging to one or more content providers with similar requirements. For example, CNN videos and Apple iOS software downloads are each examples of a traffic class. Large CDNs host content that belong to thousands of domains which can in turn be classified into hundreds of traffic classes.

A large CDN is composed of several subsystems that work together to maximize cache hits. Traffic provisioning and cache management are two important systems (Figure 1.2) that perform complementary functions.

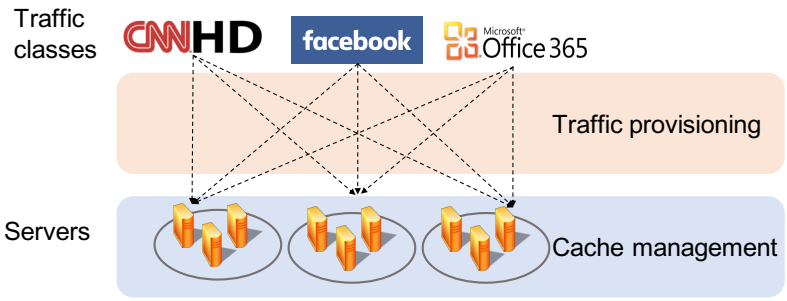


Figure 1.2: The traffic provisioning and cache management systems work together to maximize the cache hit rate of the CDN.

### 1.2.1 Traffic provisioning

The traffic provisioning system operates at a higher level and determines what stream of user requests are directed to each server in the CDN. Traffic provisioning is the process of deciding what mix of traffic classes are to be cached and served by each cluster (resp. server) to maximize the cache hit rate. As shown in Figure 1.2, the traffic provisioning system decides which clusters (resp. servers) serve what fraction of each traffic class such as CNN videos, Facebook images, or Microsoft downloads. The goal of traffic provisioning is optimizing metrics that are important to the CDN, such as maximizing the aggregate hit rate of each cluster (resp. server) and the specific

hit rates of each traffic class that it hosts. Once traffic provisioning is complete, the *mapping system* routes user requests to their appropriate servers in *real time*. The reader is referred to [26, 92] for a more detailed description of the mapping system in the world's largest CDN.

### 1.2.2 Cache management

The cache management system operates at the level of an individual cluster or server. Each server maintains a cache that stores content requested by users. The cache management system is composed of two policies that manage the content cached in the servers. A *cache admission policy* decides if an object should be cached in the server. A *cache eviction policy* decides which existing object(s) should be evicted from cache when space needs to be created for a newly admitted object. Then, given a stream of requests for a mix of traffic classes hosted in the cluster (resp. server), the goal of the cache management system is to determine how to admit and evict objects to maximize the cache hit rate.

## 1.3 Challenges in traffic provisioning and cache management and proposed solutions

The complexity of content distribution and the diversity of performance and cost metrics pose challenges for efficient traffic provisioning and cache management. We list four challenges that we address in this dissertation and briefly describe the proposed solutions.

### 1.3.1 Predicting caching characteristics of traffic mixes using footprint descriptors

*Challenge:* Provisioning and controlling the sharing of the available cache space in a server or cluster of servers among the hundreds of traffic classes hosted on a CDN is an important and challenging operational area. It has direct impact on the

cost-performance tradeoff of the CDN. The wide variability in popularity distributions, object size distributions and caching performance across traffic classes makes it challenging to predict the hit rate of traffic mixes that share the cache space. This in turn affects traffic provisioning across the CDN. Developing better caching models that accurately capture popularity and caching characteristics will enable better traffic provisioning and cache management.

*Proposed solution:* Managing a vast shared caching infrastructure requires careful modeling of user request sequences for each traffic class. We introduce the notion of a footprint descriptor that is a succinct representation of the cache requirements of a request sequence. Leveraging novel connections to Fourier analysis, we develop a footprint descriptor calculus that allows us to predict the cache requirements when different traffic classes are added, subtracted and scaled to within a prediction error of 2.5%. We integrated our footprint calculus in the traffic provisioning operations of a production CDN and show how it is used to solve key challenges in cache sizing, traffic mixing, and cache partitioning. This work has been published in ACM CoNEXT [107].

### 1.3.2 Midgress-aware traffic provisioning

*Challenge:* Traffic provisioning is traditionally performed by an operator who makes ad hoc decisions based on past experiences. This can be sub-optimal due to the wide variability in the popularity and caching characteristics of traffic classes that are hosted on a CDN. The holy grail of traffic provisioning is to optimally map traffic classes to clusters (resp. servers) automatically without human intervention at all. Such automation should achieve different performance objectives such as minimizing the cache miss traffic or minimizing the end-user latency, while also taking into account the resource constraints that are imposed by the clusters (resp. servers) in the network.

*Proposed solution:* We formulate traffic provisioning across the CDN as a mixed integer linear problem. We propose algorithms that provision traffic classes to servers and clusters such that the midgress traffic is minimized and no server is overloaded. Using extensive traces from Akamai's CDN, we show that our midgress-aware traffic provisioning schemes can reduce midgress by nearly 20% in comparison with the midgress-unaware schemes that are currently in use. We also propose an efficient heuristic for traffic provisioning that achieves near-optimal midgress reduction and is suitable for use in production settings. Further, we show how our algorithms can be extended to other settings that require minimum caching performance per traffic class and minimum content duplication for fault tolerance. Finally, our work provides a strong case for implementing midgress-aware traffic provisioning in production CDNs. This work is under submission.

### 1.3.3 Adaptive cache management using TTL-based caching

*Challenge:* CDNs would like to guarantee certain caching performance such as a minimum cache hit rate or maximum cache space usage, for every traffic class. But, it is difficult to do so in the face of increasing variability in user demands and the characteristics of traffic classes served. Moreover, the request distribution in production settings are non-stationary and difficult to model and analyze theoretically. Developing cache management algorithms that adapt to such non-stationarity will enable better use of caching resources and the ability to provide better performance guarantees.

*Proposed solution:* TTL-based caching algorithms enable better control on the cacheability of content. We propose two TTL-based caching algorithms that have provable guarantees for bursty and non-stationary request traffic. The first algorithm called d-TTL adapts a TTL parameter using stochastic approximation to converge to its target hit rate. The second algorithm called f-TTL adapts two TTL parameters

using stochastic approximation to converge to the dual target of hit rate and the expected cache size, provided they are feasible. Using extensive traces from Akamai, we show that both algorithms converge to their hit rate targets with a small error of less than 2.3%. But f-TTL requires significantly smaller cache size since it uses one of the TTL parameters to filter out non-stationary and unpopular content. This work has been published in IEEE/ACM Transactions on Networking [7].

### 1.3.4 Energy-efficient caching using disk shutdown

*Challenge:* So far, we have focused on addressing challenges related to cache hit rates which indirectly affect the performance and cost metrics such as the end-user latency and the bandwidth cost. But CDNs are also concerned about the energy consumption of the networks. Energy minimization has become critical for two reasons. Deployed servers in data centers now account for more than 1.5% of the global power consumption [77], consuming more power than mid-sized countries such as Argentina.With greater awareness of climate change, the CDN industry is increasingly focused on making their systems more sustainable.

A second motivator is the rising cost of energy. The cost of energy has been rising over the past decade [8]. The cost structures at most data centers are such that energy cost presently ranges between 30-50% of the total operating expense, and is expected to only rise further in the coming decades. Hence, CDNs have great incentive to reduce the operating expenses by being energy-efficient.

*Proposed solution:* Each CDN server has multiple spinning disks that are used for caching content. These disks account for 40-55% of the total server energy usage of a CDN. Reducing the energy consumption of a CDN by shutting down some of the disks is the main focus of our work. We propose and evaluate cache management schemes that allow disks to be shut down without significantly impacting cache hit rates and user-perceived performance. We empirically evaluate the energy-performance tradeoff

of our algorithms using extensive request traces from Akamai. We show that it is feasible to obtain a 30% disk energy savings with a 6.5% reduction in the normalized server hit rate and a mere 3% reduction in the normalized cluster hit rate. This work establishes disk shutdown as a key mechanism for energy savings in CDNs. This work has been published in ACM e-Energy [108].

## 1.4 Dissertation outline

We characterize the variability of traffic classes served by a CDN in Chapter 2. We also develop the theory of footprint descriptors and show how they can be used to predict the caching characteristics of traffic mixes. In Chapter 3, we develop optimization models for traffic provisioning to minimize midgress across the CDN. We also develop heuristic algorithms that can be deployed in practice. In Chapter 4, we describe a cache management scheme using adaptive TTLs to guarantee cache performance metrics. In Chapter 5, we describe how disk shutdown can be used to make the cache management system energy-efficient without significantly impacting the cache hit rate. We conclude in Chapter 6 with a summary of the proposed algorithms and some ideas for future work.

# CHAPTER 2

# TRAFFIC MIXING USING FOOTPRINT DESCRIPTORS

A large CDN hosts content from tens of thousands of domains belonging to web sites of thousands of content providers. Further, each content provider may host different types of content, including web, downloads, videos, and images[1]. The content traffic served by CDNs are in turn classified into traffic classes. A global CDN hosts content belonging to hundreds of traffic classes. Requests from users accessing the content provider's web sites are routed by the CDN to an appropriate cache server that can serve the content using a process called mapping [26].

A CDN performs traffic provisioning and request routing at the granularity of a traffic class. Thus, a key decision a CDN must make is which subset of its hundreds of thousands of servers must serve which traffic classes. Different traffic classes may have different caching characteristics and different performance requirements. A traffic class consisting of web content from an e-tailer may require fast response times and high cache hit rates to aid more sales conversions, and the object sizes are smaller. In contrast, a traffic class consisting of background software downloads has large object sizes, but can tolerate lower hit rates and slower response times.

– *Traffic provisioning:* Traffic provisioning is the process of determining which traffic classes are hosted in which cache servers of the CDN, given a vast platform of cache servers with varying amounts of cache space available at each server. Despite the

[1]Content providers often segregate their content by type and place them on different domains for better content management and delivery. For instance, a content provider may have a different domain for each content type, such as {`www,video,image,download`}`.foo.com`.

potentially diverse requirements for each traffic class, it is economically and operationally advantageous for the CDN to use a single shared platform of servers to serve all the traffic classes and to have each cache serve multiple traffic classes. However, such sharing across multiple traffic classes poses significant challenges that are the focus of this work.

Provisioning traffic classes and controlling the sharing of the available cache space between those classes to maximize cache hit rates is an important challenge with direct impact on the cost-performance tradeoff of the CDN. For example, servers hosting an aggressive mix of traffic classes relative to the available cache space may end up providing poor cache hit rates. This may violate the performance requirements for some classes, and raise bandwidth cost due to elevated midgress traffic. Conversely, servers hosting a conservative mix of traffic classes may end up underutilizing their resources, which makes the CDN buy more servers than necessary.

The goal of traffic provisioning is to *model* the caching requirements of traffic classes and to *predict* the best way to assign traffic classes to cache servers, so as to optimize the use of the cache resources and provide an acceptable hit rate at a reasonable cost. To do so, the process takes as input the sizes of the caches available in servers across the CDN, and the characteristics of the request sequences for each traffic class. The process outputs the set of servers that serve each traffic class. Traffic provisioning is an offline planning step but it must be performed regularly, since new traffic classes are added or removed to the system and caching characteristics of existing classes may change. Once traffic provisioning is complete, its output is used by a mapping system to route the requests of each traffic class to one of the provisioned cache servers in real-time.

– *Challenges and contributions:* The main conceptual challenges in traffic provisioning and our contributions in addressing those challenges are below.

**1)** To provide effective traffic provisioning, we need to first understand the diversity of traffic classes hosted on a modern CDN, and how they vary in terms of user request patterns, content popularity, object sizes, and caching requirements. *In Section 2.1, we provide the first detailed characterization of traffic classes on a modern CDN.*

**2)** The user requests for each traffic class must be modeled efficiently from the traces. While traces may contain hundreds of millions of requests, the model must be *concise*, and must be able to *predict* the resource-performance tradeoffs for caches that serve that traffic class. *In Section 2.2.1, we propose the novel notion of a footprint descriptor (FD) that is computed efficiently from user request traces of that traffic class. Using footprint descriptors, we can derive the full tradeoff between cache size and hit rate for each traffic class.*

**3)** A main goal of traffic provisioning is to answer important "what-if" questions through modeling and prediction. Examples of such questions include: what would the hit rate be when multiple traffic classes are mixed together and served by a single shared cache? How should you partition a cache across multiple traffic classes, so that each class receives its target hit rate? How would the hit rates change if the traffic volume of a traffic class is increased? *In Section 2.2.3, we develop a calculus for footprint descriptors that lets us perform addition, subtraction, and scaling operations on request sequences. The calculus lets us model, predict, and answer the key "what-if" questions that arise in the traffic provisioning context. For instance, the calculus allows us to efficiently compute the footprint descriptor of a mix of traffic classes, given the footprint descriptors of each individual class in the mix.*

**4)** Traffic provisioning must be able to process and manipulate traffic class models in an *efficient fashion. In Section 2.2.3, we show an intriguing connection to Fourier analysis that lets us visualize and manipulate footprint descriptors. Specifically, we show how Fast Fourier Transform (FFT) can be used to transform footprint descrip-*

*tors to the "frequency" domain. Analogous to how signal processing can be speeded up by using Fourier Transforms, we show how footprint descriptors can be efficiently manipulated in the frequency domain.*

**5)** The models used in traffic provisioning should provide predictions that are accurate enough to use in production CDN operations. *In Section 2.3, we highlight the need for footprint descriptor calculus through simulations using traces from production servers. We also compare our predictions with hit rates from the production network and show that the prediction error is at most 2.5% in the scenarios considered. In Section 2.4, we show how footprint descriptors are used to solve key challenges in CDN operations.*

– *Footprint descriptor modeling versus cache simulations:* In theory, one could evaluate the hit rates of different traffic class mixes by experimentally simulating cache operations on each request trace mix. But, simulating various combinations of several hundred traffic classes for different cache sizes is unscalable and prohibitively expensive, even for an offline computation, since it must be repeated periodically (say, every few days). With our approach, the footprint descriptor is computed for each traffic class only once (Section 2.4.3 shows how to compute FDs efficiently using a map-reduce paradigm) and traffic mixes are evaluated rapidly using footprint descriptor calculus. *The power of footprint descriptors is that it needs to be computed only once for each traffic class from the voluminous traces. Various operations on traffic classes can then be performed rapidly using the calculus without costly cache simulations of traffic class mixes.*

– *Roadmap:* The rest of this chapter is organized as follows. In Section 2.1, we describe the characteristics of the different traffic classes hosted on the CDN. In Section 2.2, we introduce the notion and develop the theory of footprint descriptors. In Section 2.4, we show how footprint descriptors can be used in a production setting for CDN cache operations. In Section 2.5 we review prior work and conclude in Section 2.6.

## 2.1 Traffic class characteristics

Each domain hosted on the CDN can be thought of as generating a *request sequence* that consists of users requesting content from that domain. A domain also belongs to a traffic class, where each traffic class is a set of domains from a set of similar content providers, usually serving a specific content type. A request sequence for a traffic class is simply a sequence of requests received for some domain within that class. The major traffic classes in a modern CDN have content types that are either web sites, videos, images, or downloads. A large CDN may host tens of thousands of domains from thousands of content providers that form several hundred traffic classes. The main challenge in traffic provisioning is the diversity of access patterns, object sizes, and resource requirements across different traffic classes.

### 2.1.1 Trace collection

To illustrate this diversity of traffic classes in a quantitative fashion, we collected extensive traces from Akamai's production CDN for four representative traffic classes from two production cache servers in Akamai's CDN. The data set contains anonymized logs of content accessed by end-users. Each line in the production trace corresponds to a single request and contains a timestamp, the requested URL (anonymized), and the size of the object.

The four representative traffic classes each represent a major content type: web, downloads, videos and images. The web request trace is for HTML objects and associated objects such as css and javascript files. The downloads request trace contains predominantly large objects consisting of software updates from a content provider. The image trace contains images embedded in web pages. The video trace contains video-on-demand (VOD) objects. Typically the objects belonging to the download and video traffic classes are several GB in size. But, in our traces, these objects are smaller because a CDN fragments such large files to smaller chunks to avoid caching

the entire object. CDNs typically cache only the byte-range that is requested and a few extra bytes, anticipating future requests (spatial locality). Videos are normally served in chunks that correspond to a few seconds of the video. Hence, CDNs typically only cache those chunks that are requested to avoid polluting the cache with content that has not been requested.

We collected the web and download traces from one production server and the image and video traces from the other production server. The characteristics of these traces are described in Table 2.1. We also collected additional web, download, image and video traces from two more servers to evaluate the accuracy of our cache models. These traces are described in Section 2.3.

| Traffic class | Web | Download | Image | Video |
|---|---|---|---|---|
| Length of trace (days) | 2.5 | 2.5 | 3.5 | 3.5 |
| Arrival rate (req/s) | 520 | 77 | 52 | 57 |
| Traffic volume (Mbps) | 333.0 | 216.5 | 8.4 | 361.5 |
| Object count (millions) | 10.3 | 0.7 | 1.3 | 6.1 |
| Average object size (MB) | 0.21 | 2.32 | 0.03 | 1.53 |

Table 2.1: Characteristics of the chosen traffic class traces.

### 2.1.2 Analysis of traffic classes

We compare and contrast the four chosen traffic classes based on their popularity distribution, object size distribution and cache hit rate. Figure 2.1 shows the popularity distribution of each traffic class. For our download traffic class, we see that 92% of all requests are for only 10% of the objects. For our web traffic class, 87% requests are for 10% of the objects. Both of these traffic classes have a long tail of popularity, indicating the presence of a large amount of unpopular content. For our image traffic class, 90% requests are for 30% of the objects. The *footprint* of a set of objects is the total bytes that need to be stored in cache to serve those objects from cache. From the traces we collected, we observe that we can achieve a high cache hit rate with a

small footprint for our image class due to smaller object sizes. Finally, 90% of the requests in our video class are for 65% of the requested objects (i.e. requested video chunks), indicating that a larger footprint needs to be cached to achieve a good hit rate.

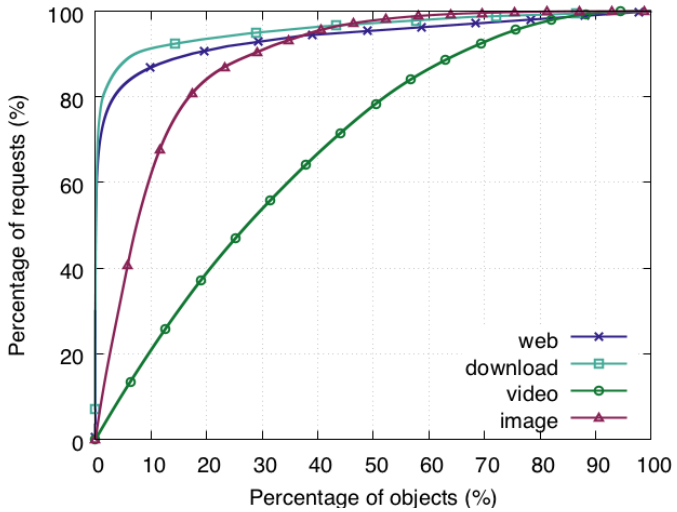


Figure 2.1: Popularity distribution of the 4 traffic classes.

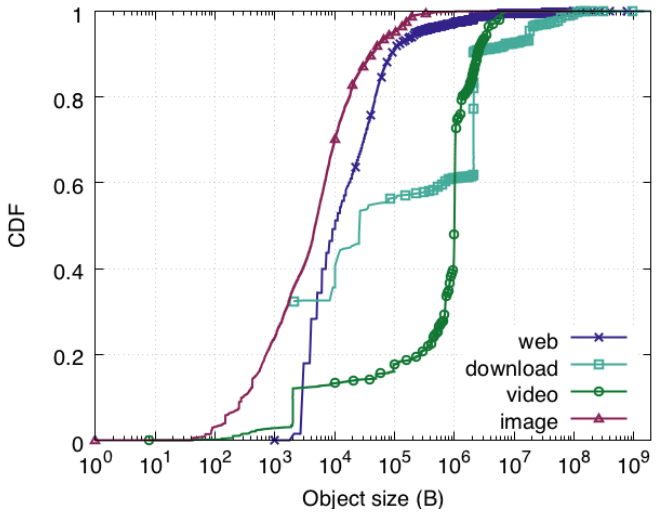


Figure 2.2: Object size distribution of the 4 traffic classes.

Figure 2.2 shows the CDF of the object size distribution for each traffic class. The x-axis is shown in log scale for clarity. In general, we see that the image and web traffic

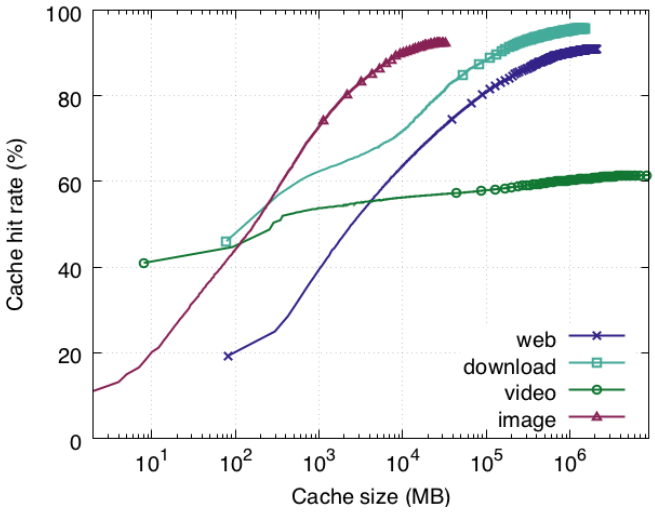


Figure 2.3: Hit rate curves of the 4 traffic classes.

classes have predominantly small objects and both the download and video traffic classes have predominantly large objects. The extreme variability in object sizes across traffic classes makes it challenging to manage cache resources across the CDN because different traffic classes require different amounts of cache space to achieve the same hit rate.

We compare the cache hit rate of the different traffic classes in Figure 2.3 by plotting their *hit rate curves (HRCs)* which gives the hit rate as a function of cache size. These hit rate curves were derived using footprint descriptors as shown in Section 2.2.2. The x-axis is shown in log scale for clarity. Note that we need a relatively smaller cache to achieve a large cache hit rate for the image traffic class, when compared to the video traffic class which needs a much larger cache for the same hit rate. For example, to achieve a hit rate of 60%, the image class requires a cache space of about 0.4 GB, whereas the video class requires a cache space of about 1 TB.

The extreme variability in popularity, object size and caching performance highlights the importance of efficient traffic provisioning when caching content belonging to different classes in a shared server. Note that two traffic classes of the same content

type but different content providers may have different access characteristics and performance requirements. Thus, traffic provisioning is done on a per-traffic-class basis, rather than on content types.

## 2.2 Theory of footprint descriptors

We now describe a concise space-time representation of a traffic class called footprint descriptor (FD) and derive a calculus for evaluating traffic mixes. Let $\rho$ be a *request sequence* $\langle r_1, r_2, \cdots r_n \rangle$ corresponding to a traffic class $\tau$, where each request $r_i$ represents a user requesting an object belonging to that traffic class. Each request $r_i$ has associated with it the timestamp $t_i$ when the request was made, a unique identifier $id_i$ of the object (such as its URL), and the size of the object $s_i$. We denote a subsequence $\rho'$ of $\rho$ to be the sequence of consecutive requests $\langle r_i, r_{i+1}, \cdots r_j \rangle$ of $\rho$, for some $i \leq j$. We call $\rho'$ a *reuse subsequence* if the same object is accessed in the first and last request of the subsequence, but is not accessed elsewhere in $\rho'$. It is known that reuse subsequences have great significance in evaluating caching properties in other contexts [85]. They play an important role in FDs and their calculus as well.

### 2.2.1 Footprint descriptors (FD)

A typical request stream $\rho$ may have tens of millions of requests. We would like to efficiently summarize the attributes of $\rho$ using the notion of a footprint descriptor, so that we may answer questions regarding the cacheability of $\rho$ using FD. To that end, we define the FD of $\rho$ as the tuple $\langle \lambda, P^r(s,t), P^a(s,t) \rangle$, where $\lambda$ is the traffic volume (in bits requested per second), $P^r(s,t)$ is the reuse-sequence descriptor function, and $P^a(s,t)$ is the all-sequence descriptor function. We describe each component of FD below.

1) The traffic volume $\lambda$ is the average number of bits requested per second in the request sequence $\rho$. For a sequence $\rho$ with $n$ requests, the traffic volume $\lambda =$

$(\sum_i b_i)/(t_n - t_1)$, where $b_i$ denotes the number of bits requested by the $i^{th}$ request and $t_i$ denotes the time of the $i^{th}$ request.

2) The reuse-sequence descriptor $P^r(s,t)$ is a *"space-time"* description of the reuse-subsequences $\rho'$ of $\rho$. In particular, it provides the joint probability distribution of the unique bytes $s$ and the duration $t$ for reuse subsequences $\rho'$ of $\rho$. The unique bytes $s$ accessed in $\rho'$ is simply the sum of the sizes of all the unique objects requested in $\rho'$. The duration of $t$ of $\rho'$ is the difference in timestamps of the first and the last request in $\rho'$. Then, $P^r(s,t)$ is the probability that $s$ unique bytes of content is requested in some reuse sequence $\rho'$ of duration $t$. Given a request sequence $\rho$, $P^r(s,t)$ can be estimated by enumerating all its reuse sequences $\rho'$ and tallying its unique bytes $s$ and duration $t$. Note that the unique bytes and duration on the first access of any object is infinity. This accounts for the cold cache miss rate.

3) The all-sequence footprint descriptor $P^a(s,t)$ computes a similar statistic, but using any subsequence $\rho'$ of $\rho$, i.e., $\rho'$ is not necessarily a reuse subsequence and the first and last request of $\rho'$ can be arbitrary. $P^a(s,t)$ is the probability that $s$ unique bytes of content is requested in some subsequence $\rho'$ of duration $t$. Given a request sequence $\rho$, $P^a(s,t)$ can be estimated by enumerating all its subsequences $\rho'$ and tallying the unique bytes $s$ and duration $t$.

### 2.2.2 Estimating cache properties from footprint descriptors

A footprint descriptor is a succinct representation of a request sequence that allows us to predict the hit rate performance that can be achieved for that sequence. We now show that the hit rate curve (HRC) of a request sequence can be derived from its FD in the context of the commonly-implemented Least-Recently-Used (LRU) caching algorithm. Most production CDNs use extensions of LRU, including Akamai [81], Varnish [67] and NGINX [101].

**Theorem 2.2.1.** *The hit rate curve $HRC(s)$ for a request sequence $\rho$ is a function that provides the hit rate achieved for $\rho$ by an LRU cache of size $s$. The function $HRC(s)$ can be computed from the reuse-sequence descriptor $P^r(s,t)$ as follows.*

$$HRC(s) = \sum_{s' \leq s} \sum_{t} P^r(s', t).$$

*Proof.* Let $\rho' = \langle r_i, r_{i+1}, \cdots r_j \rangle$ be a reuse sequence of the request sequence $\rho$. That is, $r_i$ and $r_j$ are consecutive requests for the same object. For any cache of size $s$ that uses LRU, the request $r_j$ experiences a cache hit if and only if the unique bytes requested in $\rho'$ is at most $s$, i.e., if the unique bytes is more than $s$ the object requested by $r_i$ that enters the cache will get evicted by the time the next request for the same object arrives at $r_j$. Thus, the hit rate $HRC(s)$ is simply the probability that a reuse sequence has unique bytes that is at most $s$, which in turn equals $\sum_{s' \leq s} \sum_{t} P^r(s', t)$. □

Besides LRU, the above theorem can be extended to other stack algorithms using the well-known relationship between unique bytes in a reuse sequence (called the stack distance) and hit rate [85].

### 2.2.3 A calculus of footprint descriptors

The power of footprint descriptors is that it can support operations on request sequences that are important for traffic provisioning. We present three key operations, addition, subtraction, and scaling. In Section 2.4, we show key applications of these operations in traffic provisioning in the production network.

#### 2.2.3.1 Addition

Let $\rho_1$ and $\rho_2$ be two request sequences that are independent and share no common objects[2]. The addition operator $\oplus$ can be applied to the two sequences to obtain a new sequence $\rho$ which we represent as $\rho = \rho_1 \oplus \rho_2$. Request sequence $\rho$ is obtained by interleaving $\rho_1$ and $\rho_2$ in accordance with the time stamp for the requests. We now show how the footprint descriptor $FD = \langle \lambda, P^r, P^a \rangle$ for $\rho$ can be derived from the footprint descriptor $FD_1 = \langle \lambda_1, P_1^r, P_1^a \rangle$ for $\rho_1$ and $FD_2$ for $\rho_2 = \langle \lambda_2, P_2^r, P_2^a \rangle$.

The traffic volume $\lambda$ of $\rho$ is simply the sum of the traffic volumes of $\rho_1$ and $\rho_2$, i.e.,

$$\lambda = \lambda_1 + \lambda_2 \tag{2.1}$$

To compute the descriptor functions, we introduce some notation. Given a descriptor function $P(s,t)$, let $P(s \mid t)$ denote the conditional probability of unique bytes $s$ given time duration $t$ and let $P(t)$ denote the marginal distribution, i.e., $P(t) = \sum_s P(s,t)$. Thus,

$$P(s,t) = P(s \mid t)P(t). \tag{2.2}$$

The key observation of our calculus is that when two request sequences are combined, i.e., $\rho = \rho_1 \oplus \rho_2$, and we examine a subsequence $\rho'$ of $\rho$ of duration $t$ with $s$ unique bytes, the unique bytes $s$ in $\rho'$ either come from $\rho_1$ or $\rho_2$. Since $\rho_1$ and $\rho_2$ have non-overlapping sets of objects, some $s_1$ must come from $\rho_1$ and the remaining $s - s_1$ must come from $\rho_2$. Thus, to compute a descriptor function $P(s \mid t)$ for $\rho$ from the descriptor functions $P_1(s \mid t)$ and $P_2(s \mid t)$ for $\rho_1$ and $\rho_2$ respectively, we can use the convolution operator to enumerate and add up the probabilities of all possible ways

[2]The assumption that two traffic classes share no common objects is reasonable in practice, since the objects belong to different domains from possibly different content providers. Such objects are treated as being different by the caching system.

of obtaining $s_1$ unique bytes from $\rho_1$ and the remaining $s - s_1$ unique bytes from $\rho_2$. That is,

$$\begin{aligned} P(s \mid t) &= P_1(s \mid t) * P_2(s \mid t) \\ &= \sum_{s_1=0}^{s} P_1(s_1 \mid t) P_2(s - s_1 \mid t). \end{aligned}$$

Using this observation, we now compute $P^r(s \mid t)$ of $\rho$ from $\langle P_1^r(s \mid t),\, P_1^a(s \mid t)\rangle$ of $\rho_1$ and $\langle P_2^r(s \mid t),\, P_2^a(s \mid t)\rangle$ of $\rho_2$ as follows. Let $\rho'$ be a reuse sequence of $\rho$, i.e., the first and the last request of $\rho'$ is for the same object. Let $\rho'$ have $s$ unique bytes and duration $t$. $\rho'$ can be broken up into two subsequences $\rho'_1$ of $\rho_1$ and $\rho'_2$ of $\rho_2$. With probability $\frac{\lambda_1}{\lambda_1+\lambda_2}$ the first (and, last) request of $\rho'$ is derived from $\rho_1$. That is, $\rho'$ is composed of a reuse sequence $\rho'_1$ and an arbitrary sequence $\rho'_2$. Similarly, with probability $\frac{\lambda_2}{\lambda_1+\lambda_2}$, $\rho'$ is composed of a reuse sequence $\rho'_2$ and an arbitrary sequence $\rho'_1$. Thus,

$$\begin{aligned} P^r(s \mid t) &= \frac{\lambda_1}{\lambda_1+\lambda_2}\left(P_1^r(s \mid t) * P_2^a(s \mid t)\right) \\ &\quad + \frac{\lambda_2}{\lambda_1+\lambda_2}\left(P_1^a(s \mid t) * P_2^r(s \mid t)\right), \end{aligned} \tag{2.3}$$

where $*$ denotes the convolution operator.

We can also compute $P^a(s \mid t)$ from $P_1^a(s \mid t)$ and $P_2^a(s \mid t)$. The computation is analogous to the above, except that $\rho'$ can be an arbitrary sequence of $\rho$, not necessarily a reuse sequence. Since $\rho'$ is composed of two arbitrary subsequences of $\rho_1$ and $\rho_2$, our computation involves only one convolution below.

$$P^a(s \mid t) = P_1^a(s \mid t) * P_2^a(s \mid t). \tag{2.4}$$

Note that the convolution operator $*$ arises naturally in our calculus, allowing us to leverage the powerful tools of Fourier analysis for the efficient computation of

footprint descriptors. Putting together Equations 2.1, 2.2, 2.3, and 2.4 above, we can compute the FD of $\rho$ from the FDs of $\rho_1$ and $\rho_2$ as we show in more detail in Algorithm 1.

**Algorithm 1** Addition algorithm

**Input:** $FD_1 = \langle \lambda_1, P_1^r, P_1^a \rangle$, $FD_2 = \langle \lambda_2, P_2^r, P_2^a \rangle$, $S$ and $T$ be the buckets for $s$ and $t$ respectively

**Output:** $FD = \langle \lambda, P^r, P^a \rangle$

1: $\lambda = \lambda_1 + \lambda_2$
2: **for all** t $\in T$ **do**
3: $\quad P^r(t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^r(t) + \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^r(t)$
4: $\quad P^a(t) = \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^a(t) + \frac{\lambda_2}{\lambda_1+\lambda_2} P_2^a(t)$
5: $\quad$ **for all** s $\in S$ **do**
6: $\quad\quad P^r(s \mid t) = \frac{\lambda_1}{\lambda_1+\lambda_2} (P_1^r(s \mid t) * P_2^a(s \mid t)) + \frac{\lambda_2}{\lambda_1+\lambda_2} (P_1^a(s \mid t) * P_2^r(s \mid t))$
7: $\quad\quad P^r(s,t) = P^r(s \mid t) P^r(t)$
8: $\quad\quad P^a(s \mid t) = P_1^a(s \mid t) * P_2^a(s \mid t)$
9: $\quad\quad P^a(s,t) = P^a(s \mid t) P^a(t)$

*Time complexity:* We can use Fourier analysis to speed up the computation of the addition operation. Let $S$ and $T$ be the maximum value buckets for $s$ and $t$ respectively. The addition operation can be performed in $O(TS \log S)$ time, since we need to perform 3 convolution operations in total for Equations 2.3 and 2.4 for every value of $t$, where each convolution takes $O(S \log S)$ time using Fast Fourier Transform algorithm (FFT) and $t$ takes on $T$ values.

*Inferring the individual hit rates of $\rho_1$ and $\rho_2$ after addition:* Let the hit rate curves $HRC_1'(s)$ and $HRC_2'(s)$ represent the post-addition individual hit rate curves of $\rho_1$ and $\rho_2$ within $\rho_1 \oplus \rho_2$, i.e. $HRC_i'(s)$ is the post-addition hit rate of $\rho_i$ when the traffic mix occupies cache capacity $s$. Then, $HRC_1'(s)$ and $HRC_2'(s)$ can be computed as follows.

$$HRC_1'(s) = \sum_{s' \le s} \sum_t P^r(s' \mid t) P_1^r(t).$$
$$HRC_2'(s) = \sum_{s' \le s} \sum_t P^r(s' \mid t) P_2^r(t). \tag{2.5}$$

#### 2.2.3.2 Subtraction

The subtraction operation models the traffic provisioning operation of removing some traffic classes from the list of traffic classes served by a cache server. The result of that operation is that the request stream corresponding to those traffic classes are subtracted out. Given a request sequence $\rho_1$ that is a subsequence of $\rho$, we define $\rho_2 = \rho \ominus \rho_1$ to be the sequence obtained when the requests of $\rho_1$ are removed from $\rho$. We show how the FD of the resultant sequence $\rho_2$ can be obtained from the FDs for $\rho$ and $\rho_1$. Note that we relate the request streams with the addition operator, i.e., $\rho = \rho_1 \oplus \rho_2$. Thus, the FD of $\rho_2$ can be computed by simply "inverting" Equations 2.1, 2.3, and 2.4 that we derived earlier for addition. By inverting Equation 2.1, we get

$$\lambda_2 = \lambda - \lambda_1 \tag{2.6}$$

The key idea for finding the descriptor functions is that the convolution $A = B * C$ can be inverted by using the frequency domain, i.e., $C = \mathcal{F}^{-1}(\mathcal{F}(A)/\mathcal{F}(B))$, where $\mathcal{F}$ and $\mathcal{F}^{-1}$ are Fourier transform and its inverse respectively. Thus, inverting Equation 2.4 we get

$$P_2^a(s \mid t) = \mathcal{F}^{-1}(\mathcal{F}(P^a(s \mid t))/\mathcal{F}(P_1^a(s \mid t))). \tag{2.7}$$

To find $P_2^r(s \mid t)$, we use Equation 2.3 to first compute $P_2^r(s \mid t) * P_1^a(s \mid t)$. Then, since we know $P_1^a(s \mid t)$, we can compute $P_2^r(s \mid t)$ by using the Fourier transform and its inverse as above. We provide the details of the subtraction algorithm in Algorithm 2. $P_2^r(s \mid t)$ on line 8 in Algorithm 2 is computed from Equations 2.3 and 2.7.

*Time complexity:* Let $S$ and $T$ be the maximum value buckets for $s$ and $t$ respectively. The subtraction operation can be performed in $O(TS \log S)$ time, since $t$ takes on $T$ values, we need to perform $O(1)$ Fourier (or, inverse Fourier) transforms

**Algorithm 2** Subtraction algorithm

**Input:** $FD = \langle \lambda, P^r, P^a \rangle$, $FD_1 = \langle \lambda_1, P_1^r, P_1^a \rangle$, $S$ and $T$ be the buckets for $s$ and $t$ respectively

**Output:** $FD_2 = \langle \lambda_2, P_2^r, P_2^a \rangle$

1: $\lambda_2 = \lambda - \lambda_1$
2: **for all** $t \in T$ **do**
3: $\quad P_2^r(t) = \left( P^r(t) - \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^r(t) \right) \frac{\lambda_1+\lambda_2}{\lambda_2}$
4: $\quad P_2^a(t) = \left( P^a(t) - \frac{\lambda_1}{\lambda_1+\lambda_2} P_1^a(t) \right) \frac{\lambda_1+\lambda_2}{\lambda_2}$
5: $\quad$ **for all** $s \in S$ **do**
6: $\quad\quad P_2^a(s \mid t) = \mathcal{F}^{-1}(\mathcal{F}(P^a(s \mid t))/\mathcal{F}(P_1^a(s \mid t)))$
7: $\quad\quad P_2^a(s,t) = P_2^a(s \mid t) P_2^a(t)$
8: $\quad\quad P_2^r(s,t) = P_2^r(s \mid t) P_2^r(t)$

for each value of $t$, and each Fourier (or, inverse Fourier) transform takes $O(S \log S)$ time using FFT.

#### 2.2.3.3 Scaling

Suppose we wish to increase or decrease the traffic volume for a request sequence $\rho_1$. This operation is called scaling and we express the new request sequence $\rho = \rho_1 \otimes \tau$, where $\tau$ is the factor by which the volume is increased (resp. decreased). We model the volume increase (resp. decrease) as scaling the time variable, i.e., the time stamp of each request in $\rho_1$ is divided by the factor $\tau$. This has the effect of decreasing (resp. increasing) the inter-arrival times for the requests by $\tau$ when $\tau > 1$ ($\tau < 1$). We compute the FD of $\rho$ from the FD of $\rho_1$ as follows.

$$\lambda = \lambda_1 \tau; P^r(s, t/\tau) = P_1^r(s,t); P^a(s, t/\tau) = P_1^a(s,t). \tag{2.8}$$

It is worth noting that scaling does not change the hit rate curve of $P^r(s, t/\tau)$ since the marginal distribution of $P^r(s) = P_1^r(s)$.

*Time complexity:* Let $S$ and $T$ be the maximum value buckets for $s$ and $t$ respectively. The computations in Equation 2.8 can be performed in $O(ST)$ time, faster than addition or subtraction.

*Note:* It should be noted that the footprint descriptor calculus described in this section predicts the byte hit rate of a request sequence, which is the metric considered in this work. The calculus works just the same to predict the object hit rate, with the slight modification that $\lambda$ for a request sequence $\rho$ is the arrival rate in requests per second rather than in bits per second. Also, note that the calculus allows us to compute the FD of complex traffic mixing operations by composing the three supported operators. For example, to subtract a subclass $\tau_1'$ from a class $\tau_1$ and add the resultant to half the volume of class $\tau_2$, the FD of the final mix $\tau = ((\tau_1 \ominus \tau_1') \oplus (\tau_2 \otimes 1/2))$ can be computed efficiently using the calculus and FFT from the FD's of $\tau_1, \tau_1'$ and $\tau_2$.

### 2.2.4 A simpler footprint descriptor (SFD)

In this section, we outline a simplification of footprint descriptors that makes implementations faster, at the cost of some theoretical rigor. Empirically, we observed that on production traces the descriptor functions $P^a(s,t)$ and $P^r(s,t)$ were statistically similar, i.e., the reuse sequences that start and end in a request for the same object, and arbitrary sequences that do not have the reuse property were statistically similar. The reason is that request sequences have requests for millions of different objects, and conditioning on starting and ending on a request for the same object does not alter the statistical behavior of the rest of the sequence very much. Therefore, a simpler footprint descriptor (SFD) is a tuple $\langle \lambda, P^r(s,t) \rangle$, i.e., the all-sequence descriptor $P^a(s,t)$ is dropped since it is similar to reuse-sequence descriptor $P^r(s,t)$. Having just one descriptor function makes computing SFD much simpler and faster for the addition and subtraction operations. For instance, if $P^a(s,t)$ is assumed identical to $P^r(s,t)$, Equation 2.4 simplifies to the following equation that requires just one convolution instead of two.

$$P^r(s \mid t) = P_1^r(s \mid t) * P_2^r(s \mid t). \tag{2.9}$$

Note that SFDs can be used to derive the cache hit rate curve HRC as described in Section 2.2.2, since it depends only on $P^r(s,t)$. For these reasons, we often use SFDs in practice, in lieu of FDs.

## 2.3 Validation of footprint calculus

In this section, we validate the addition operation described in Section 2.2.3 by computing the hit rate curves using the footprint calculus on SFDs described in Section 2.2.4. We then compare the calculus predictions with the hit rates obtained via cache simulations using the production traces, a simple baseline algorithm, as well as hit rates obtained directly from the production server. Further validation of addition and subtraction also appears as part of the case studies in Sections 2.4.1.1 and 2.4.1.3 respectively. We do not validate the scaling operator since the hit rate curve after scaling remains unchanged.

*Additional traces for validating scalability of addition:* Our initial set of traces described in Table 2.1 were from servers that each served two major classes i.e., the top two traffic classes accounted for most of the traffic from the server. To validate the addition of more traffic classes, we chose two additional production servers one that served four traffic classes across the four content types of web, image, video and download and another server that served nine traffic classes across three content types, namely web, video and download. These new traces let us evaluate the accuracy of the calculus when a larger number of traffic classes are mixed. The details of the additional traces are described in Tables 2.2 and 2.3 respectively. In Table 2.3 we show nine different traffic classes that have web, video, and download content from different content providers.

*Baseline traffic mixing algorithm:* We describe a baseline traffic mixing algorithm commonly used in operations that predicts the cache hit rate of a traffic mix using only the hit rate curves of all traffic classes. For every value of the cache hit rate,

| Traffic class | Web | Download | Image | Video |
|---|---|---|---|---|
| Length of trace (days) | 1 | 1 | 1 | 1 |
| Arrival rate (req/s) | 223.53 | 51.04 | 216.42 | 180.01 |
| Traffic volume (Mbps) | 411.95 | 101.40 | 41.23 | 181.11 |
| Object count (millions) | 2.89 | 0.23 | 8.78 | 2.54 |
| Average object size (MB) | 0.3 | 0.4 | 0.02 | 0.24 |

Table 2.2: Characteristics of the $2^{nd}$ set of traces.

| Traffic class | Web-1 | Web-2 | Web-3 | Web-4 | Video-1 | Video-2 | Video-3 | Video-4 | Download |
|---|---|---|---|---|---|---|---|---|---|
| Length of trace (days) | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| Arrival rate (req/s) | 168 | 16 | 6 | 3 | 53 | 21 | 4 | 3 | 22 |
| Traffic volume (Mbps) | 1105.4 | 114.7 | 1.8 | 0.002 | 292.6 | 112.4 | 21.6 | 14.9 | 234.5 |
| Object count (millions) | 15 | 1.6 | 0.05 | 0.07 | 11.7 | 2.5 | 1.9 | 0.6 | 1.9 |
| Average object size (MB) | 1.6 | 1.9 | 0.05 | 1.7 | 0.7 | 0.8 | 0.7 | 0.9 | 2.0 |

Table 2.3: Characteristics of the $3^{rd}$ set of traces.

the baseline algorithm determines the cache capacities for each traffic class from their respective hit rate curves and adds them up. The hit rate curve thus produced is the predicted curve for the traffic mix. For example, consider two traffic classes with hit rate curves $HRC_1(s)$ and $HRC_2(s)$ respectively. Then, the cache capacity required by the traffic mix to achieve hit rate $h$ is predicted as $HRC_1^{-1}(h) + HRC_2^{-1}(h)$. This is repeated for all values of $h$ to produce the hit rate curve of the mix.

The baseline algorithm described above is extremely simple and fast with time complexity $O(S)$, for hit rate curves having $S$ cache size buckets. While simple,

the baseline scheme does not account for the inter-arrival time distributions of the request sequences, and hence is an unreliable predictor of cache hit rates. We discuss the shortcomings in the following section.

### 2.3.1 Experimental evaluation

We show via simulations using production traces that the calculus is more accurate at predicting hit rates of traffic mixes and is necessary for traffic provisioning. For our simulation-based validation, we combine production traces corresponding to each traffic mix and perform a cache simulation on these merged traces for different cache sizes to obtain a hit rate curve. We call this the "simulated" hit rate curve. We also compute the hit rate curve of the traffic mix using the calculus. We call this the "calculated" hit rate curve. Finally, we compute the hit rate curve using the baseline algorithm and we call this the "baseline" hit rate curve.

| Traffic mixes | Average error of baseline, % | Average error of calculus, % |
|---|---|---|
| web+download (Table 2.1) | 0.24 | 0.13 |
| video+image (Table 2.1) | 0.63 | 0.10 |
| Traffic classes in Table 2.2 | 10.2 | 0.28 |
| Traffic classes in Table 2.3 | 11.8 | 0.34 |

Table 2.4: Average prediction error of the baseline algorithm vs. the FD calculus.

In Table 2.4, we present the average error of the baseline and the calculated hit rates with respect to the simulated values. We present the error for performing the addition operation for the web and download classes, and video and image classes in Table 2.1 and the addition of all classes in Tables 2.2 and 2.3. While the baseline algorithm has a small error for web+download and video+image from Table 2.1, the large difference and variability in error between the baseline algorithm and the calculus in general, and the consistently small average error of the calculus, highlight the need for the more accurate calculus in predicting the effects of traffic mixing.

We now discuss another scenario where the calculus is superior to the baseline algorithm. Very often, CDN operators need to predict the effects of traffic scaling on traffic mixing, to better provision caches under varying traffic conditions. For instance, CDN operators would like to know the hit rate of a traffic mix when the traffic volumes of one or more traffic classes are varied. We show that under such circumstances, the calculus (using the scaling operation in conjunction with addition) provides more reliable outputs than the baseline algorithm which responds erratically to traffic scaling.

To illustrate this scenario, we consider the traffic mix of the traffic classes in Table 2.2. We consider two scenarios, 1) the traffic classes are mixed at their current traffic volumes (unscaled versions) and 2) we scale the traffic volume of the download traffic class up by 20 times, to 2028 Mbps, and predict the cache hit rate under traffic mixing in this new scenario (scaled versions). In Figure 2.4, we plot the "simulated", "calculated" and "baseline" hit rate curves without scaling. We also plot the hit rates curves of the traffic mix (after scaling the download traffic class) predicted by the calculus ("calculated-scale") and the baseline algorithm ("baseline-scale"). We refer to traffic mix web+image+video+download from Table 2.2 as "wivd". We also zoom in on the x-axis for clarity.

From Figure 2.4, we see that the hit rate curve predicted by the calculus, wivd(calculated), closely matches the simulated hit rate curve, wivd(simulated), with an average error of 0.28%. However, the error between the simulated curve, wivd(simulated), and the baseline scheme, wivd(baseline), without scaling is much higher at 10.2% on average, as previously discussed. After scaling the download traffic class up by a factor of 20, we see that the hit rate of the traffic mix, wivd(calculated-scale), increases as expected because the download traffic class dominates the mix (see the hit rate curve of the download traffic class labelled pre-addition in Figure 2.5). On the contrary, there is little change in the estimate of the baseline algorithm after scaling,

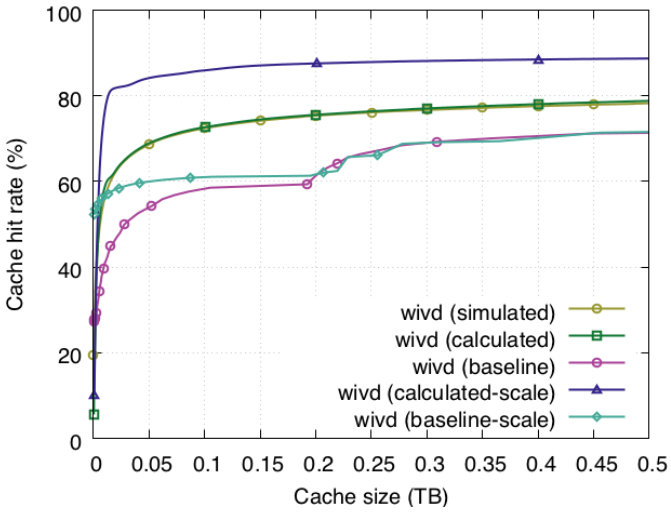


Figure 2.4: Hit rate curves of the traffic mix in Table 2.2 before and after scaling the download traffic class by a factor of 20.

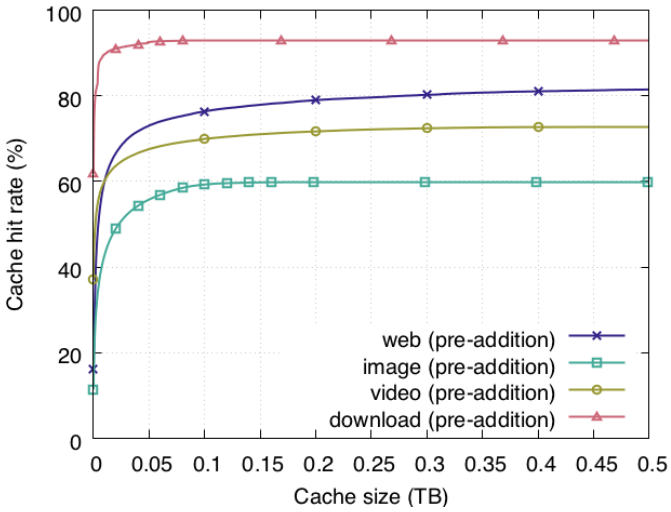


Figure 2.5: Hit rate curves of traffic classes in Table 2.2.

wivd(baseline-scale), and the estimated hit rate is less than that of the calculus by 17.8% on average. This is because the baseline algorithm is unaware of the changes in the inter-arrival times of requests after scaling, which the calculus takes into account.

We do similar comparisons for the other sets of traces. We scale up 20 times the image traffic class in the video+image mix in Table 2.1. We also scale up 20 times the download traffic class in both the web+download mix in Table 2.1 and the

mix of traffic classes in Table 2.3. We find that "baseline-scale" is 8.6% less than "calculated-scale" on average in the case of video+image in Table 2.1, 4.2% less than "calculated-scale" on average in the case of web+download in Table 2.1 and 23.1% less than "calculated-scale" on average for the traffic mix in Table 2.3. This evaluation further emphasizes the need for the more accurate calculus that accounts for both the spatial and temporal interactions between traffic classes during traffic mixing and predicts hit rates accurately.

*Production evaluation:* We perform a production validation by comparing the hit rates produced by our calculus with the hit rates measured directly from the production server serving the required mix of traffic over the same time period. Measuring it directly from the production server measures the actual production caching software implemented on the deployed hardware. This validation method validates only one point on the HRC, the point that corresponds to the actual cache size of the production server. We measured the average hit rate reported by the production servers corresponding to the traffic mixes in Tables 2.1, 2.2 and 2.3. The results are in Table 2.5.

| Traffic mix | Cache size (TB) | Hit rate from calculus, % | Hit rate from server, % |
|---|---|---|---|
| web+download (Table 2.1) | 3.0 | 86.6 | 84.1 |
| video+image (Table 2.1) | 3.7 | 37.7 | 39.3 |
| Traffic classes in Table 2.2 | 2.0 | 79.9 | 77.4 |
| Traffic classes in Table 2.3 | 3.7 | 68.6 | 67.5 |

Table 2.5: Production validation.

From Table 2.5, we see that that calculus predicts the cache hit rate of the traffic mixes considered with a prediction error of at most 2.5% in all cases. The difference

in hit rates is in part due to the fact that the production servers were intermittently used to serve small amounts of other traffic classes by the mapping system. Moreover, the footprint calculus models a pure LRU algorithm while the production system has extra optimizations that we do not currently model. The low prediction errors from our production validation further strengthens the case for using footprint descriptor calculus to provision traffic in CDNs.

## 2.4 Applying footprint descriptors in a production CDN

In this section, we provide case studies to show how footprint descriptors play a key role in traffic provisioning in CDNs.

### 2.4.1 Traffic mix evaluation service

Several what-if questions arise in the process of determining the optimal traffic mix of traffic classes that is served by a cache server. A traffic mix evaluation service can provide detailed information about cache occupancy and hit rates when various traffic classes are combined together. The output of the service prevents poor mixing choices from going into effect in any cache server. We used footprint descriptors to implement the service that is currently in use by the operations staff at Akamai in a limited beta setting. The service computes footprint descriptors for each traffic class hosted on the CDN using the techniques described in Section 2.2. The service keeps a database of all the cache servers and their properties, including the cache space available. We show how FDs are used to answer key questions that arise in the context of traffic mixing.

#### 2.4.1.1 Estimating space requirement of a traffic mix

Given a set of traffic classes with their respective traffic volumes, a CDN operator might be interested in the cache capacity required by the traffic mix to provision

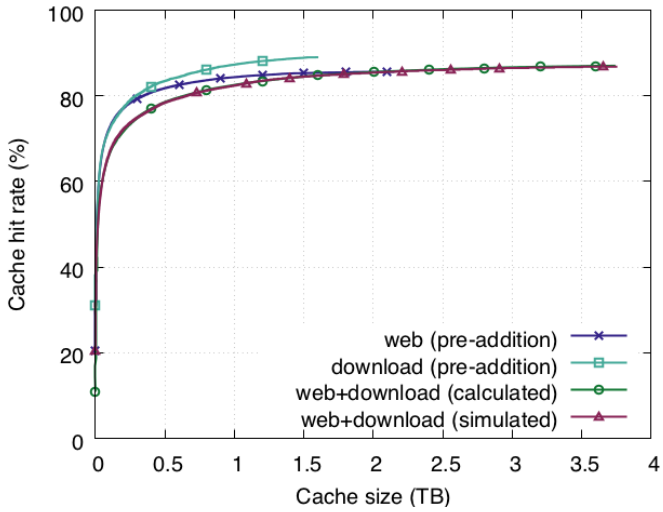


Figure 2.6: Addition operation on web and download traffic classes in Table 2.1.

servers to achieve a target hit rate. The traffic mix evaluation service computes the output using the following steps.

1) The footprint descriptor of all the traffic classes of interest are computed efficiently using the techniques outlined later in Section 2.4.3. Each traffic class is then scaled to the required traffic volume using the $\otimes$ operator described in Section 2.2.3.3, and added together using the $\oplus$ operator described in Section 2.2.3.1, which gives the footprint descriptor of the traffic mix.

2) Using Theorem 2.2.1, the HRC of the traffic mix is computed from its footprint descriptor.

3) Given the HRC and the target hit rate $h$, the required cache size $s$ is determined such that the hit rate $HRC(s) \geq h$.

*Validation of functionality:* We consider one example, the traffic mix of the web and download traffic classes in Table 2.1 to illustrate the use of footprint descriptor calculus. Figure 2.6 shows the HRCs of the web and download traffic classes labeled "pre-addition", as well as the HRC of the mix as computed by the calculus labeled "calculated". To validate the correctness of the HRC of the mix, we merge the request traces for both the classes by interleaving the requests in the ascending order

of their time stamp. The HRC derived from the merged trace using cache simulations is shown in Figure 2.6, labeled "simulated". As can be seen, the calculated and simulated HRCs match closely with an average error of 0.13%, thus validating that the addition algorithm works on production traces.

#### 2.4.1.2 Predicting the outcome of traffic mixing in a given server

For this use case, the operator provides the cache size of the server and the traffic classes with their respective traffic volumes to be mixed in that server. With this input, the service does the following:

1) The footprint descriptors of all the traffic classes of interest are computed, scaled to the required traffic volume using the $\otimes$ operator described in Section 2.2.3.3, and added together using the $\oplus$ operator described in Section 2.2.3.1, to give the footprint descriptor of the traffic mix.

2) Using Theorem 2.2.1, the HRC of the traffic mix is computed. Using Equation 2.5, we also compute the (post-addition) HRCs of each traffic class in the mix.

3) Given the total cache size and the HRC of the mix, we obtain the hit rate of the mix. Using the total cache size and the (post-addition) HRC of the individual traffic classes, we obtain the hit rates of each traffic class in the mix.

4) Using the post-addition hit rates of each traffic class and their (pre-addition) HRCs, we can obtain the cache space occupied by each traffic class after the addition.

Thus, our service predicts the overall cache hit rate for a given traffic mix, individual cache hit rates for each traffic class after mixing, and the cache space occupied by each traffic class in the cache after mixing. This information is valuable to differentiate between good and poor mixes.

*Validation of functionality:* We continue with the above example of mixing web and download traffic classes described in Table 2.1. We assume here that the cache server has a total cache size of 1 TB. As shown in Figure 2.6, the HRC labeled "web+

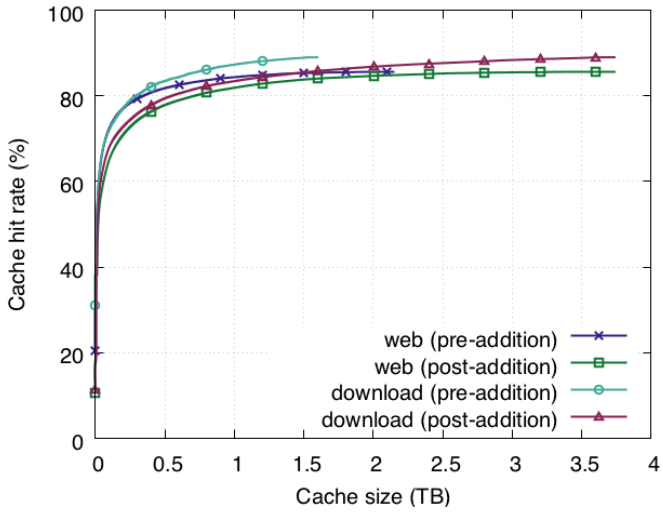


Figure 2.7: Pre-addition and post-addition HRCs of web and download traffic classes in Table 2.1.

download(calculated)" says that the cache will get 82% overall hit rate. The HRC labeled "web+download(simulated)" confirms the correctness of the value. Figure 2.7 shows the pre-addition and post-addition HRCs for both the web and download traffic classes. Plugging in 1 TB as the cache space, we see that within the mix, the web traffic class gets a hit rate of 80.5%, while the download traffic class gets a hit rate of 84%. Now, performing a reverse lookup for these hit rates in the original HRCs for these two traffic classes in Figure 2.7, we see that the web traffic class occupies 300 GB in cache, while the download traffic class occupies the remaining 700 GB.

#### 2.4.1.3 Splitting domains in a traffic class

A traffic class consists of user requests for content hosted on a specific set of domains. Often, domains need to be removed from traffic classes to achieve better load balancing. In such situations, it is important to predict how the caching characteristics of the traffic class would change if some domains are removed. This prediction can be performed using the subtraction operator $\ominus$ described in Section 2.2.3.2. We first collect the traces for the domains to be removed from the original traffic class,

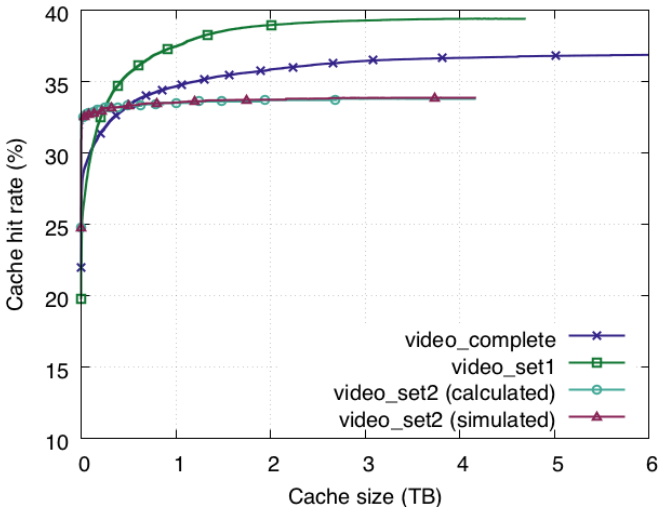


Figure 2.8: Subtracting a subset of domains from the video traffic class in Table 2.1.

and we compute the footprint descriptor for the resulting traffic class using the subtraction operation. The resultant footprint descriptor can be used to compute the hit rates after the split.

*Validation of functionality:* We plot the results of this operation in Figure 2.8. Note that the x-axis has been truncated for clarity of presentation. In this instance, we want to remove a certain set of domains from the video traffic class in Table 2.1. In Figure 2.8, video_complete indicates the hit rate curve for the entire traffic class, video_set2(calculated) is the hit rate curve of the remaining domains when video_set1 is removed. video_set2(simulated) is the hit rate curve of video_set2 computed via cache simulations. We see that video_set2(calculated) compares very well with video_set2(simulated), with an average error of 0.05%, confirming that the $\ominus$ operator works well on production traces as well.

### 2.4.2 Hit rate targets with cache partitioning

In many situations, it is necessary to guarantee a certain hit rate performance for a subset of traffic classes in a traffic mix, while ensuring that the traffic mix does well overall. The traffic mix evaluation service uses cache partitioning to ensure that

each traffic class meets its target hit rate while making the best use of the remaining cache space to maximize the overall cache hit rate. The aims of cache partitioning are twofold:

1) Ensure that each traffic class gets at least its requisite cache space, and

2) Any leftover space is assigned appropriately to the traffic classes, so that the hit rate of the cache as a whole is maximized.

The first aim is achieved using our calculus as follows:

1) Using Theorem 2.2.1, HRCs are computed from the footprint descriptor for all traffic classes, and

2) A reverse lookup is performed on the HRCs, to get the cache size needed for the given target hit rate. This is the requisite partition size for each traffic class.

Towards the second aim, the leftover cache space is divided into a number of smaller blocks. Each block is added incrementally to the traffic classes, using the following method:

1) Compute the traffic-weighted first derivative of the HRC of each traffic class at the point where the size of the cache is equal to its current partition size,

2) Identify the traffic class with the highest value of this first derivative. This class can provide the most benefit in cache hit rates if it is given the block being considered, and

3) Assign the block to the class with the highest derivative, and increase its partition size.

All the remaining cache space is assigned to traffic classes in this way. The resulting partition of cache space maximizes the server's cache hit rate, while meeting the hit rate targets of the individual traffic classes. Once the partition sizes are determined, the partitions are implemented as separate virtual LRU caches within the given server. The method of allocating leftover cache space is similar to the utility

maximization approach described in [100] where the first derivative of the HRC is the utility function.

*Validation of functionality:* We continue with our example of mixing web and download traffic classes from Table 2.1. We assume here that the cache server has a total cache size of 1 TB. First we consider targets of 85% hit rates for both the traffic classes. The (post-addition) HRCs for the individual traffic classes suggest that neither class can achieve this hit rate, since the cache space available is 1 TB. Thus, partitioning to achieve 85% hit rate is infeasible. Next, we consider a hit rate target of 83% for the web class and 75% for the download class. These targets are chosen to illustrate how footprint descriptors can be used for cache partitioning. A reverse lookup of hit rate in Figure 2.6 shows that the requisite cache spaces for the web and download classes are 625 GB and 125 GB, respectively. This leaves 250 GB of available cache space unassigned to either class. The first derivative method [100] is used to determine the assignment of the leftover 250 GB, and it assigns all of it to the download class. Thus, the 1 TB cache is partitioned into 625 GB and 375 GB partitions. The hit rate of the web class meets its target of 83%, while the download class achieves a better than target hit rate of 82.5%. As observed in [100], cache partitioning can be used to improve the overall cache hit rate when multiple traffic classes share the cache space. Indeed the cache hit rate after mixing the web and download traffic classes increases from 82% without explicit partitioning to 83% with partitioning.

### 2.4.3 Parallelizing the computation of FDs

Before footprint descriptors can be used in operational decision making as illustrated in previous sections, they need to be computed from request sequences. Typically, request sequences over observation periods of a couple of days to a few weeks are processed to compute footprint descriptors. These sequences may contain

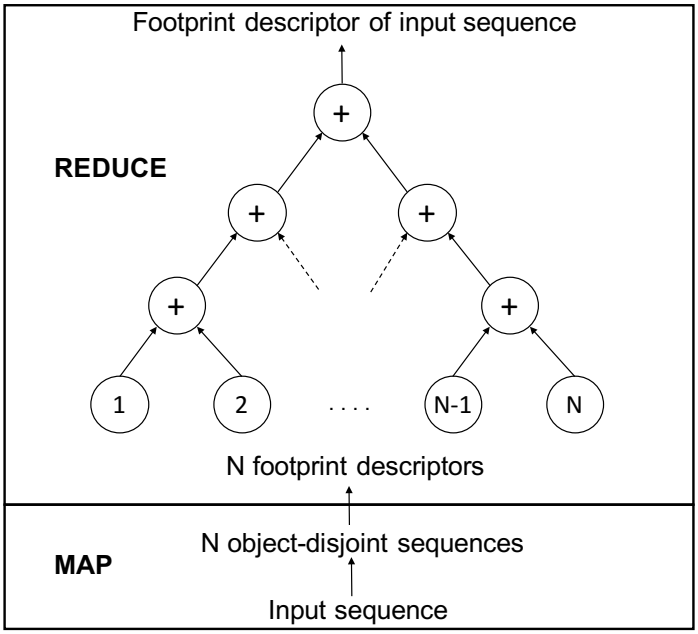


Figure 2.9: Map-reduce framework to parallelize footprint descriptor computation.

over a billion distinct URLs. Such industrial-strength computation is a heavyweight proposition both in terms of memory and CPU cycles. We develop a novel map-reduce-based framework that uses the $\oplus$ operation of our calculus to parallelize the computation of footprint descriptors for large request sequences. The procedure is below.

*Map phase:* Split the input request sequence into N smaller request sequences that share no objects between them. We accomplish this by hashing the URL of each request into N buckets, each bucket representing a smaller request sequence.

*Reduce phase:* Compute the footprint descriptors of the N smaller sequences in parallel. Using the $\oplus$ operator, add the footprint descriptors for the smaller sequences in parallel, until we are left with one footprint descriptor. This is the footprint descriptor of the input stream.

The complete framework is shown in Figure 2.9 where, $\oplus$ is the addition algorithm described in Section 2.2.3.1. The reduce phase begins by computing the footprint descriptors for the N object-disjoint smaller sequences (the N leaves) and adding

them in parallel, bottom up, until we obtain the footprint descriptor of the original input sequence which is the root of the tree. This framework parallelizes a seemingly serial process and can speed up computation that increases nearly linearly in the number of compute nodes.

*Validation:* We implement this parallel footprint descriptor computation algorithm in Amazon EMR [1]. We run map-reduce jobs in a cluster with up to 32 nodes. We use m3.xlarge machines in all experiments. The reported time is the elapsed time of the map-reduce job recorded by the cluster.

To evaluate the speedup due to parallelization, we use a much larger set of production traces corresponding to a video traffic class collected from 29 servers over a period of 8 days. The request rate is 1,234 req/s and the traffic volume is 6.76 Gbps. The trace contains 227.3 million objects with an average object size of 0.7 MB.

We observe that it takes 420 minutes to compute the footprint descriptor of the video traffic class without parallelization and 28 minutes with a 16-way parallelization and 16 minutes with a 32 way parallelization, that is a speed up of 15 and 26.2 respectively with no impact on the accuracy of the output.

## 2.5 Related work

Caching has been active area of research for the past few decades. We only review the work on caching that is most closely related to our work. Much of the closely-related prior work fall into the realm of cache modeling and cache composition.

*Cache modeling:* We subdivide the relevant work on cache modeling into empirical modeling based on stack distance, which is the number of unique bytes in a reuse request subsequence, and theoretical modeling that assumes certain statistical properties of the request sequences. Stack distance-based caching models were first proposed in [85]. Stack distance is useful to compute hit rate curves that plot the cache hit rate as a function of cache size. The simple algorithm proposed in [85] has

high space and time overheads and is infeasible for large input sequences such as those in the CDN context. Subsequently, several time and space-efficient algorithms have been proposed in the literature, such as those in [4, 91, 113, 118]. *However, none of the stack distance algorithms in the literature provide a calculus that allows operations such as addition, subtraction, and scaling, a key necessary ingredient that the footprint descriptor calculus provides for CDN traffic provisioning.* Several theoretical models have been proposed to predict cache hit rates as early as the 1970's [55, 74]. Of particular interest is the work in [24, 42] that relates the cache size with the cache hit rate and cache eviction age for IRM traffic. More recent work [36, 52] extend such ideas to caching policies beyond LRU. *In contrast, our work is focused on modeling and predicting properties of arbitrary production workloads of a CDN that are hard to capture with IRM-like models.*

*Cache composition:* Cache composition has been studied in the context of CPU caches, where applications running in multi-processor machines share the CPU cache. More recently, cache composition has been studied in the context of memory caches and storage systems. Analytical models have been developed to predict cache hit rates of time-shared systems, such as those in [3,105,109]. These models compute the hit rate of a shared cache in the presence of context switching. The authors in [22] propose three cache composition models with varying degrees of accuracy that predict the impact on cache hit rates when two non-overlapping applications run together in the shared L2 cache. The model presented in [38] goes one step further to characterize overlapping data in multi-threaded programs by predicting the overlapping footprint based on how threads interleave when running concurrently. Some other related work [114,119,120] develop models that more accurately characterize memory footprint of processes using novel sampling techniques and derive hit rates from those footprints. A more recent work [60] develops a kinetic model of LRU cache, based on the average eviction time (AET). *In contrast to prior work in cache composition, we*

*support a wider range of composition operations on traffic classes, including addition, subtraction and scaling. Unlike prior work, our work is based on a theoretical sound foundation of footprint descriptor calculus. Further, our empirical approach is focused to the specific challenges in CDN traffic provisioning.*

## 2.6 Conclusion

Traffic provisioning in CDNs is challenging because of the diverse requirements imposed by diverse traffic classes. It is also challenging due to the immense scale of the operations, both in terms of traffic volumes and the large network of cache servers. Footprint descriptors provide a simple and elegant way of capturing the caching properties of a traffic class. The theory of footprint descriptor calculus allows us to add, subtract and scale traffic classes to answer important "what-if" questions that arise in CDN operations. The connections to Fourier analysis that allow footprint descriptors to be manipulated in the "frequency domain" is also of interest. Footprint descriptors are well-suited for use in production network operations, since as we show the prediction error is under 2.5% for key use cases on the servers considered. Currently, footprint descriptors are being used in Akamai's production network to decide what traffic classes should be assigned to servers or clusters of servers to maximize cache hit rates.

# CHAPTER 3

# MIDGRESS-AWARE TRAFFIC PROVISIONING

The bandwidth cost of midgress caused by cache misses is a major expense for the CDN and could cost tens of millions of dollars a year[1]. Thus, even a small factor reduction in midgress is an attractive proposition. Much of the work in academia and industry has focused on better cache management for reducing the cache miss rates of individual caches. The past decades have seen research on numerous cache admission and evictions policies, such as Adapt-Size [12], Cliffhanger [30], SLRU [68], TinyLFU [40], S4LRU [61], CFLRU [96], ARC [87], LRU-S [104], LRU-K [94], and GDS [17]. However, the *complementary* problem of optimizing the traffic provisioning process to minimize midgress has not received much attention. The traffic provisioning process determines the mix of traffic classes that each cache serves (Figure 1.2). *The main thesis of this work is that by explicitly incorporating midgress considerations, it is possible to devise traffic provisioning schemes that minimize midgress traffic by nearly 20%, potentially resulting in millions of dollars of bandwidth cost savings.* Further, the midgress reduction due to better traffic provisioning is *complementary* to any potential improvements in the cache management system.

In the current state-of-the-art, production CDNs assign traffic classes to servers with the goal of not overloading the servers, without explicitly focusing on the cache miss rates and midgress reduction of the network. CDN operators have traditionally

[1] As a back-of-the-envelope calculation, a large CDN serving 50 Tbps of egress traffic at a 20% miss rate at the edge has a midgress traffic of 10 Tbps. The price of network bandwidth varies greatly throughout the world. Though hard to estimate accurately, assuming a blended price of 50 cents per Mbps per month, midgress bandwidth costs 60 million dollars per year.

focused on midgress reduction by simply tweaking the caching management policies that operate at the level of individual servers. Our work shows that traffic provisioning in a midgress-aware manner can provide additional benefits to what can accrue from better cache management alone. Our traffic provisioning approach incorporates *both* traditional load balancing and the newer midgress considerations to minimize midgress traffic. Hence, it can be viewed as a drop-in replacement for an existing (midgress-unaware) traffic provisioning system.

– *Midgress-aware traffic provisioning:* "Midgress-aware" traffic provisioning algorithms explicitly incorporate cache miss traffic in addition to "balancing" the load. We illustrate the need for midgress awareness through a simple example. Consider two servers and three traffic classes. Each server has a cache size of 4 TB and sufficient capacity to serve all traffic classes. The three traffic classes have equal load of $\lambda$ that need to be assigned to the two servers. The miss rate curves (MRCs) for the three traffic classes are as shown in Figure 3.1. The MRCs of traffic classes $TC_1$ and $TC_3$ flatten out quickly (high gradient in the beginning and zero-gradient there after). This means that they require very little cache space to achieve the best possible performance. On the other hand, traffic class $TC_2$ has a slowly decreasing gradient. Thus, the miss rate of $TC_2$ keeps decreasing as more cache space is allocated to it.

Current traffic provisioning algorithms are *midgress-unaware* in that they only ensure that no server is overloaded. Such an algorithm could choose *any* assignment of traffic classes to servers, since any server has sufficient capacity to serve all classes, e.g., assigning $TC_1$ and $TC_2$ to server 1 and $TC_3$ to server 2 is one possible solution. More generally, any assignment with $(x + y + z) \times \lambda$ traffic to server 1 and $((1 - x) + (1 - y) + (1 - x)) \times \lambda$ traffic to server 2 is feasible, where $x, y$ and $z \in [0, 1]$, are the traffic fractions of $TC_1$, $TC_2$ and $TC_3$ respectively. However, a midgress-aware algorithm would choose an assignment that minimizes the overall cache miss traffic from the two servers, while also ensuring that no server is overloaded. In the above

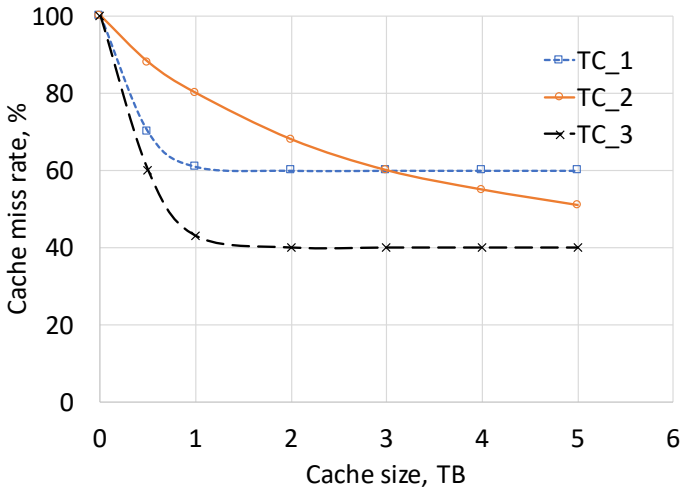


Figure 3.1: MRCs of traffic classes $TC_1$, $TC_2$ and $TC_3$.

example, assigning all of $TC_1$ and $TC_3$ to server 1 and all of $TC_2$ to server 2 would result in the least amount of cache miss traffic from the two servers. This is because $TC_2$ gets the largest cache space possible for its entire load and $TC_1$ and $TC_3$ get enough space to achieve the smallest cache miss rates.

– *Contributions:* We make the following specific contributions.

**1)** We develop an optimization model for midgress-aware traffic provisioning that assigns traffic classes to servers in a manner that minimizes midgress traffic. The model is a non-convex mixed-integer linear program (MILP) that we solve using CPLEX (a standard constraint solver). Our work is the first to explicitly model and minimize midgress in the traffic provisioning process, a significant advance over the current state-of-the-art in traffic provisioning algorithms in CDNs. *Since a large CDN could incur a midgress of 10+ Tbps at a cost of $60+ million/year, even a small midgress reduction translates into large cost savings for the CDN.*

**2)** We apply our optimization solution to *metro-level traffic provisioning* where the traffic classes provisioned to a set of server clusters within a metro area (e.g., NY city) are re-provisioned to reduce the total midgress traffic. Metro-level traffic re-provisioning is a common operation within a CDN, since the latency impact of

moving a traffic class from one cluster to another within the same metro is likely minimal. Using extensive production traces from Akamai's CDN, we show that our midgress-aware traffic provisioning can reduce the midgress of an entire metro-area by 18.37% on average compared to the prior midgress-unaware traffic provisioning.

**3)** We also use our optimization solution for *cluster-level provisioning* where we take traffic classes assigned to a cluster and re-provision them to reduce the total midgress traffic. Cluster-level traffic (re-)provisioning is also a common operation within a CDN since moving a traffic class from one server to another within the same cluster will likely not impact end-user latencies. Using production traces from Akamai's CDN, we show that cluster-level provisioning in conjunction with metro-level provisioning can reduce the midgress of a traffic class by 41.07% on average compared to the prior midgress-unaware traffic provisioning.

**4)** To be useful in practice, midgress-aware traffic provisioning has to be computationally efficient, since such provisioning is performed periodically across hundreds of thousands of servers, deployed in 1000+ clusters around the world, that serve hundreds of traffic classes. We propose a footprint-aware heuristic called `local search` that is fast and near-optimal. The midgress achieved by `local search` is within 1.1% of optimal for both metro-level and cluster-level traffic provisioning. Further, in our experiments, `local search` completed in only 2 minutes, while finding the optimal took several hours.

**5)** Using production traces from Akamai, we show that our traffic provisioning algorithms are robust across different cache management policies and provide a midgress reduction in the range 7.76% - 13.3%.

**6)** CDN operators often have to deal with additional constraints such as maintaining a certain level of traffic class redundancy for fault tolerance or guaranteeing a minimum level of caching performance for certain traffic classes. We show how the op-

timization model for midgress-aware traffic provisioning and the heuristic algorithm, `local search`, can be extended to accommodate such constraints.

**7)** While the above results are for "shared" caches where a single unpartitioned cache is used to store objects from all traffic classes, we show that our traffic provisioning approach can be modified to also work with "partitioned" caches where each traffic class is assigned a separate partition of the cache. We show that the midgress of partitioned caches can be reduced by more than 14% using our midgress-aware traffic provisioning approach, when compared to a baseline midgress-unaware approach.

– *Roadmap:* The rest of this chapter is organized as follows. In Section 3.1, we model traffic provisioning to minimize midgress as a non-convex mixed-integer optimization problem that can be solved by a constraint solver such as CPLEX. In Section 3.2, we propose a faster heuristic for midgress-aware traffic provisioning called `local search`, as well as a midgress-unaware baseline heuristic called `baseline fit`. In Section 3.3, we evaluate our optimization model and heuristics using extensive traces from Akamai's production CDN to empirically understand the midgress reduction achieved by our algorithms. In Section 3.4, we extend and evaluate our midgress-aware traffic provisioning algorithms to include other cache performance guarantees such as minimum redundancy and maximum cache miss rate per traffic class. Further, we extend our work to partitioned caches. In Section 3.5 we discuss some related work before concluding in Section 3.6.

## 3.1 Our optimization solution for traffic provisioning

We model traffic provisioning in a CDN as follows. We are given a set of $N$ traffic classes. For each traffic class $j$, we are given the (predicted) amount of load of $\lambda_j$ Gbps, $\forall j \in 1 \ldots N$. The predicted load for traffic provisioning is derived from historical load values for these classes by the CDN. Further, we are given $M$ sites where the $i^{th}$ site has a cache of size $C_i$ TB and a capacity of $T_i$ Gbps, $\forall i \in 1 \ldots M$.

In cluster-level traffic provisioning, each site models a single CDN server within a cluster of $M$ servers. In the more complex setting of metro-level traffic provisioning, we model an *entire* cluster as a single site within a metro area with $M$ clusters. While not strictly accurate, we show that viewing the entire cluster as a single site in the metro-area setting is useful in practice. The capacity (resp. cache size) of each site is calculated as either the capacity (resp. cache size) of a single server in the former setting or as the aggregate capacity (resp. cache size) of the entire cluster in the latter setting. Henceforth, a site refers to a server in the cluster-level setting and a cluster in the metro-level setting.

The goal of traffic provisioning is to produce an assignment of traffic classes to sites, such that the total midgress across all the sites is minimized within the constraint that no site is assigned more load than its capacity. Note that a traffic class may be fractionally assigned across multiple sites, e.g., a traffic class with 10 Gbps of load can be assigned across two sites to host 7 Gpbs and 3 Gbps each of that class[2].

### 3.1.1 Modeling cache eviction and midgress

Given a site with an assignment of traffic classes, we need to model the miss traffic (i.e., midgress) that will result from serving those classes. The miss traffic is dependent on the cache management policies used by the sites. Nearly all production CDN caches use LRU (least-recently-used) variants as their eviction policy, since it is very efficient and achieves a comparable (byte) miss rate for typical CDN content traffic in comparison with other more complex eviction policies. For example, Akamai servers evict content using LRU, while admitting objects on second hit [81]. Production installations of the popular content caches Varnish [67] and NGINX [101] also use LRU variants, as do recent academic work on content caching such as AdaptSize [12].

[2]A CDN can implement such a fractionally-provisioned traffic class via a DNS mechanism that returns the ip address of the first site 70% and the ip address of the second site 30% of the time.

Production CDN servers also typically use a shared cache architecture where each server uses a single unpartitioned cache to serve all its traffic classes [107]. It is known that a partitioned cache that is sized in an optimal fashion can yield a greater reduction in midgress over a shared unpartitioned cache under IRM traffic assumptions [37]. However, in a production CDN, each server hosts a large number of traffic classes. Further, both the set of traffic classes hosted by a given server and the volume of traffic served per class by that server varies throughout the day. Thus, there is *significant* overhead involved in maintaining multiple cache partitions whose sizes must be dynamically varied throughout the day. The constant resizing of cache partitions could itself also lead to an increase in the midgress [99]. For these reasons, a shared unpartitioned cache is typically used by CDNs in practice.

*In light of the above discussion, since our goal is to devise traffic provisioning algorithms to reduce midgress in production CDN settings, we develop a model for sites that use an LRU cache eviction policy with a shared cache architecture. But, later, we show empirically that our optimization model and algorithms produce a significant reduction in midgress, even if the CDN were to use other eviction policies (Section 3.3.3). Further, we show that our approach can also be easily extended to provide midgress reduction in a partitioned cache architecture (Section 3.4.3).*

#### 3.1.1.1 Eviction age equality

The eviction age of an object in cache is the difference between the time the object is evicted and the time that it was last accessed. In an LRU cache, at the time of access, the object goes to the head of the LRU list. Then, the eviction age of the object is the time for that object to move from the head to the tail of the LRU list and then get evicted. Thus, this time is about the same for all objects, when the size of an object is small with respect to the size of the cache. *We make the modeling assumption that the eviction age of all objects in cache are equal.* This assumption

is also borne out in production caches and the common eviction age of the objects is logged as the eviction age of the cache.

The notion of eviction age can be extended to a traffic class by averaging the eviction age of all the requested objects from that traffic class. Since we model each object as having the same eviction age, all traffic classes assigned to a site share the same cache, and so they must have the same eviction age, which we also denote to be the eviction age of the cache. The eviction age of a cache has a direct relationship with the cache hit rate. Requests that have inter-arrival times less than or equal to the eviction age experience a cache hit and the rest experience a cache miss. So, for a given mix of traffic classes, as the cache size increases, the eviction age increases and so does the cache hit rate. Eviction age of a cache is similar to the concept of window size in [42]. Eviction age equality is crucial in our modeling of the midgress of traffic classes that share a single LRU cache.

### 3.1.2 Formulation of our optimization model

We now formulate our optimization model (referred to as OPT henceforth) for midgress-aware traffic provisioning.

– *Inputs of OPT:* The input parameters used in the model are summarized in Table 3.1. We are given $N$ traffic classes and $M$ sites. The load $\lambda_j$ of the $j^{th}$ traffic class is given, for all $1 \leq j \leq N$. The cache size $C_i$ and the capacity $T_i$ of the $i^{th}$ site is also given, for all $1 \leq i \leq M$. Further, for each traffic class, we are given the miss rate curve (MRC) and eviction age function as described below.

*1) Miss rate curve (MRC), $\mathcal{M}_j(c)$.* The MRC of a traffic class plots the cache miss rate as a function of cache size $c$. In this work, we assume that this function is convex (decreasing) which is generally true for CDN traffic classes [107]. As examples, MRC of two traffic classes, traffic class 2 and 14 (see Table 3.3) are shown in Figure 3.2.

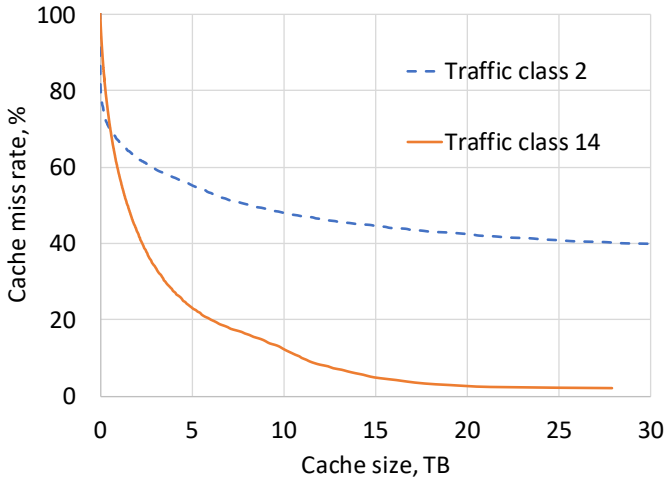


Figure 3.2: MRCs of two traffic classes in a metro area.

From Figure 3.2, we can see that the MRCs are both convex. However, their gradients vary at different rates. For instance, traffic class 2 has higher gradient at very small cache sizes but gradually flattens out as it reaches a cache space of 30 TB. Traffic class 14 on the other hand has a relatively high gradient until about 15 TB after which the MRC flattens out.

*2) Eviction age function,* $\mathcal{T}_j(c, \lambda)$. The eviction age function of a traffic class plots the eviction age at load $\lambda$ as a function of the cache size $c$. The eviction age function also gives us information about *footprint pressure* of a traffic class, which is a relative measure of the amount of unique bytes accessed over a time period. A traffic class has high footprint pressure if a large number of unique bytes are accessed over a short time period. In this work, we assume that the eviction age function is convex (increasing) based on observations from production traces. The convexity makes intuitive sense based on the observation that the popularity distribution of CDN content tends to be close to a Zipf distribution [107]. As examples, eviction age functions of traffic classes 2 and 14 (see Table 3.3) are shown in Figure 3.3.

From Figure 3.3, we can see that the eviction age functions are convex. As expected, at the given load, the eviction age increases with increase in cache size.

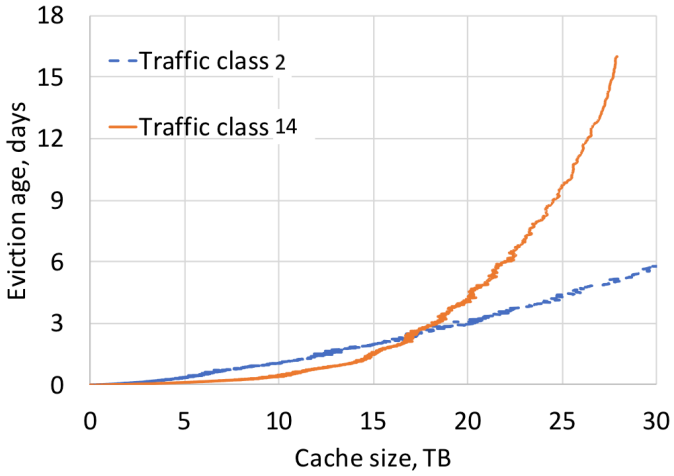


Figure 3.3: Eviction age functions of two traffic classes in a metro area. The eviction age functions indicate how traffic classes compete for shared cache space.

Note that until about an eviction age of 2.1 days, traffic class 14 has higher footprint pressure when compared to traffic class 2, after which this behavior is flipped. Hence, if traffic classes 2 and 14 are assigned to the same site, traffic class 14 gets more cache space at smaller eviction ages ($\leq$ 2.1 days) due to higher footprint pressure and lesser cache space at larger eviction ages ($>$ 2.1 days) due to smaller footprint pressure.

– *Outputs of OPT:* The output parameters of OPT are presented in Table 3.2. The primary output is $x_{ij}$ that represents the fraction of traffic class $j$ assigned to site $i$.

| Notation | Description |
|---|---|
| $N$ | Number of traffic classes |
| $M$ | Number of sites |
| $\lambda_j$ | Load of traffic class $j$ |
| $\mathcal{M}_j(c_{ij})$ | Miss rate of traffic class $j$ at cache capacity $c_{ij}$ in site $i$ |
| $\mathcal{T}_j(c_{ij}, \lambda_j)$ | Eviction age of traffic class $j$ at cache capacity $c_{ij}$ and load $\lambda_j$ in site $i$ |
| $C_i$ | Cache size of site $i$ |
| $T_i$ | Capacity of site $i$ |

Table 3.1: Input parameters of optimization model.

| Notation | Description |
|---|---|
| $c_{ij}$ | Cache size occupied by traffic class $j$ on site $i$ |
| $\rho_i$ | Eviction age of site $i$ and of traffic classes assigned to site $i$ |
| $x_{ij}$ | Fraction of $\lambda_j \in [0,1]$ assigned to site $i$ |

Table 3.2: Output parameters of optimization model.

– *Objective function:* The objective of midgress-aware traffic provisioning is to assign the $N$ traffic classes to the $M$ sites such that the midgress traffic from all the sites is minimized as follows.

$$\text{min.} \quad \sum_{i=1}^{M}\sum_{j=1}^{N} x_{ij}\lambda_j \mathcal{M}_j(c_{ij}) \tag{3.1}$$

– *Resource constraints:* The first set of constraints are the cache size and the capacity constraints of each site.

$$\sum_{j=1}^{N} c_{ij} \leq C_i \quad \forall i = 1 \ldots M \tag{3.2}$$

$$\sum_{j=1}^{N} x_{ij}\lambda_j \leq T_i \quad \forall i = 1 \ldots M \tag{3.3}$$

The cache size constraint (Equation 3.2) states that the cache size occupied by all traffic classes assigned to all sites must not exceed the cache size of the site. The capacity constraint (Equation 3.3) states that the load of all traffic classes assigned to all sites should not exceed the capacity of the site.

– *Eviction age equality constraint:* The eviction age function, $\mathcal{T}_j(c_{ij}, \lambda_j)$ is defined at load $\lambda_j$ for traffic class $j$. When traffic class $j$ is assigned to site $i$, its load can be less than or equal to $\lambda_j$ due to fractional assignments. Let the load of traffic class $j$ assigned to site $i$ be $\lambda'_j \leq \lambda_j$. Then, the eviction age of traffic class $j$ in site $i$ is.

$$\mathcal{T}_j(c_{ij}, \lambda'_j) = \frac{\mathcal{T}_j(c_{ij}, \lambda_j)}{\lambda'_j/\lambda_j} = \frac{\mathcal{T}_j(c_{ij}, \lambda_j)}{x_{ij}} = \rho_i$$

The first equality is due to the fact that decreasing the load of a traffic class by a factor increases the eviction age of that class by the same factor, since the rate of evictions decreases by that factor. In the last equality, $\rho_i$ is the eviction age of site $i$ which is also the eviction age of all traffic classes that are assigned to site $i$. The eviction age equality constraint for all traffic classes at all sites is then given by

$$\mathcal{T}_j(c_{ij}, \lambda_j) = \rho_i x_{ij} \quad \forall j(i) = 1 \ldots N(M). \tag{3.4}$$

As previously discussed, the eviction age equality constraint in Equation 3.4 establishes the condition under which traffic classes assigned to site $i$ share the cache.

– *Load assignment constraint:* The load of a given traffic class can be fractionally assigned across sites. This means that for some traffic class $j$, 50% of the load $\lambda_j$ could be assigned to site 1, 30% to site 2 and the remaining 20% to site 3, and so on. The load assignment constraint ensures that all the load of each traffic class is assigned to one or more sites.

$$\sum_{i=1}^{M} x_{ij} = 1 \quad \forall j = 1 \ldots N \tag{3.5}$$

– *Non-negativity constraints:* The output parameters $\rho_i$, $c_{ij}$ and $x_{ij}$ should be non-negative.

$$\rho_i > 0 \quad \forall i = 1 \ldots M \tag{3.6}$$

$$c_{ij} \geq 0 \quad \forall j = 1 \ldots N \tag{3.7}$$

$$x_{ij} \in [0, 1] \quad \forall j(i) = 1 \ldots N(M) \tag{3.8}$$

Together, Equations 3.1-3.8 constitute the optimization model for midgress-aware traffic provisioning OPT.

### 3.1.3 Solving the optimization model OPT

We are given as inputs the cache size and capacity of each site. Further, we are given the load, the miss rate curve (MRC) and the eviction age function for each traffic class[3]. The complexity of solving the optimization model OPT proposed in Section 3.1.2 is evaluated as follows. The objective function (Equation 3.1) is biconvex since the load fraction $x_{ij}$ is linear and the MRC $\mathcal{M}_j(c_{ij})$ is convex. Equations 3.2-3.3, 3.5-3.8 are affine constraints. The eviction age function $\mathcal{T}_j(c_{ij})$ is convex and the product term $\rho_i x_{ij}$ is bilinear. Equation 3.4 is a non-convex constraint because the feasible set defined by this constraint is non-convex. Overall, the optimization problem is non-convex and in general an NP-hard problem. We make a number of transformations to convert the optimization problem to a mixed integer linear program (MILP) as described in Appendix A, which in turn can be solved efficiently using CPLEX.

## 3.2 Traffic provisioning heuristics

The optimization model OPT proposed in Section 3.1.2 is an NP-hard problem and it can take several hours for a solver to obtain the exact optimal solution. A faster but approximate solution is valuable for a large production CDN such as Akamai that has hundreds of traffic classes, 1000+ clusters, with deployments in every major metro region of the world. To that end, we propose a traffic provisioning heuristic called `local search` that is fast and sufficiently accurate to be used in production. Intuitively, our traffic provisioning heuristic is a "hill climbing" solution for our optimization model in Section 3.1.2. We also consider a midgress-unaware traffic provisioning algorithm called `baseline fit` that we use as a baseline to evaluate

[3]To efficiently compute the MRC and eviction age function for every traffic class, we use a succinct space-time representation of the cacheability properties of a traffic class known as footprint descriptors [107]. The projections in space and time of the footprint descriptor of a traffic class yields the MRC and eviction age function respectively.

the benefits of being midgress-aware. The `baseline fit` algorithm is similar to the midgress-unaware algorithms currently used in production settings.

### 3.2.1 Midgress-unaware baseline

The midgress-unaware traffic provisioning algorithm called `baseline fit` (see Algorithm 3) is based on consistent hashing, similar to the algorithms used in production settings [81]. The algorithm takes as input the set of $N$ traffic classes and the set of $M$ sites that are both hashed to points on a unit circle. The traffic classes are picked in a random order and assigned to sites as follows. Each traffic class $j$ is assigned to the nearest site $i$ on the unit circle in the clockwise direction. If the chosen site $i$ does not have enough capacity to host the entire load $\lambda_j$, then a first fit algorithm is used, starting from the chosen site $i$, and continuing to subsequent sites on the unit circle in the clockwise direction, until all traffic is assigned. The key point to note is that `baseline fit` does not explicitly minimize the miss traffic, but rather it only ensures that no site gets more load than its capacity. That is, it produces a *feasible* solution for our model OPT by obeying Equations 3.2 - 3.8 but does not minimize midgress.

**Algorithm 3** `Baseline fit` algorithm

**Input:** $N, M, \lambda_j, C_i, T_i$
**Output:** Fraction of traffic class $j$ assigned to site $i$, $x_{ij}, \forall j(i) = 1 \dots N(M)$

```
1: x_ij = 0
2: TC_set = set of all traffic classes arranged in random order
3: S_set = set of all sites hashed to a unit circle
4: for all j ∈ TC_set do
5:     i = Site chosen by consistent hashing
6:     if site i has remaining traffic capacity ≥ λ_j then
7:         x_ij += 1
8:     else
9:         Assign traffic fraction λ_j using first fit starting from site i on the unit circle
```

### 3.2.2 Midgress-aware local search

We propose a midgress-aware traffic provisioning algorithm called `local search` (see Algorithm 4) that uses a hill climbing approach to solve the optimization model OPT. It is designed to be fast but may not always produce the optimal solution. The algorithm `local search` begins with a feasible assignment as determined by the `baseline fit` traffic provisioning algorithm. The algorithm operates in rounds where every traffic class is picked one at a time in each round. The traffic class that is picked is reassigned in small increments of a fraction $\delta$ ($0 < \delta < 1$) of its load to the server that minimizes the midgress objective while maintaining feasibility. If a round does not decrease the midgress traffic objective by at least a specified $\epsilon << 1$, the algorithm stops and outputs the final assignment.

*Computing the midgress of a traffic assignment:* The `local search` algorithm requires an efficient way to compute the midgress traffic of each site, given a traffic class assignment. A known technique for computing miss traffic of a site is footprint descriptor calculus described in [107]. A footprint descriptor is an easily-computable space-time description of a traffic class. Knowing the footprint descriptor of each traffic class[4] that is assigned to a site, we use the calculus to efficiently derive the footprint descriptor for the traffic class mix, that in turn provides the MRC of the traffic class mix, from which we derive the midgress of the traffic mix. Note that the characteristics of a traffic class could change slowly over time, requiring the footprint descriptor to be recomputed periodically.

[4]The footprint descriptor of each traffic class is recomputed periodically to account for changes in the content request characteristics for that class, e.g., changes in the popularity and object size distributions of Facebook images over time.

**Algorithm 4** `Local search` algorithm

**Input:** $N, M, \lambda_j, C_i, T_i$
**Output:** Fraction of traffic class $j$ assigned to site $i$, $x_{ij}, \forall j(i) = 1 \ldots N(M)$

Get feasible assignment using `baseline fit` algorithm
$TC_{set}$ = set of all traffic classes arranged in *random* order
$S_{set}$ = set of all sites
**while** True **do**
    $mg_{curr}$ = midgress of current assignment
    **for all** $j \in TC_{set}$ **do**
        $x_{ij} = 0 \quad \forall i = 1 \ldots M$
        $\lambda' = \lambda_j$
        **while** $\lambda' > 0$ **do**
            $S^j_{set} \subseteq S_{set}$ = set of all sites with remaining traffic capacity $\geq \delta\lambda_j$
            **if** $S^j_{set} \neq \emptyset$ **then**
                $i$ = site in $S^j_{set}$ that gives the lowest overall midgress after assigning TC $j$
                $x_{ij}$ += $\delta$
            **else**
                Assign load $\delta\lambda_j$ using fractional first fit starting from a random site
            $\lambda'$ -= $\delta$
    $mg_{new}$ = midgress of new assignment
    **if** $mg_{curr} - mg_{new} < \epsilon$ **then**
        break

## 3.3 Experimental evaluation

A goal of our evaluation is to determine the potential midgress reduction that can be obtained using our midgress-aware traffic provisioning methods. Using production traces collected from a metro area of Akamai's CDN, we compare the midgress of OPT with that of `baseline fit` and `local search` in both metro-level traffic provisioning and cluster-level traffic provisioning scenarios. We perform the evaluation in two steps:

1) We evaluate metro-level traffic provisioning by viewing each cluster as a site. The site is assumed to have cache size and capacity equal to the sum of the cache sizes and capacities of all servers in that cluster. The output of metro-level traffic provisioning is an assignment of traffic classes to clusters that minimizes the midgress of the metro area.

2) The output of metro-level traffic provisioning is the input to cluster-level traffic provisioning. We evaluate cluster-level traffic provisioning by assigning traffic classes to servers within a cluster and further optimizing the midgress performance at the cluster-level.

*Production traces:* To perform our evaluation, we collect production traces from all Akamai CDN servers from a metro area serving traffic for 25 traffic classes over a period of 16 days. The characteristics of the traffic classes are listed in Table 3.3. From Table 3.3, we see that, in this metro area, 9 traffic classes serve web content, 11 traffic classes serve media content and the remaining 5 traffic classes serve software downloads. The traffic classes exhibit a wide variation in load (Gbps), arrival rate (requests/sec), content footprint (in unique bytes), and number of objects. The majority of the load is for media content at 47.3% followed by software downloads at 41.5% and web content at 11.2%. In terms of the unique bytes that are cached in the metro area, the majority is again for media content at 60.9%, followed by 25.6% for web content and 13.5% for software downloads.

Footprint descriptors described in [107] are periodically computed for all traffic classes on the production CDN. We use these footprint descriptors to compute the MRCs and the eviction age functions for the 25 traffic classes in Table 3.3, to be used as inputs to our traffic provisioning algorithms.

*Evaluation setup:* To evaluate the traffic provisioning algorithms, we simulate a small metro region with 10 clusters, each containing 10 servers[5]. The capacity of the metro region is set so that the average load is 70% of capacity to reflect the load-to-capacity ratio in a typical CDN. We evaluate the traffic provisioning algorithms at different cache sizes per cluster of 1 TB, 5 TB, 10 TB, 20 TB, 40 TB and 50 TB. For simplicity, we assume that every cluster in the metro area has equal capacity and

[5]While a metro region in a large CDN typically has much larger server deployments, we simulate a scaled-down version to keep our experiments computationally tractable.

| Traffic class id | Content type | Load (Gbps) | Arrival rate (req/s) | Unique bytes (TB) | Unique objects (million) |
|---|---|---|---|---|---|
| 1 | web | 0.39 | 438.41 | 1.83 | 16.36 |
| 2 | web | 1.12 | 232.48 | 70.74 | 38.38 |
| 3 | media | 3.75 | 345.94 | 198.85 | 176.68 |
| 4 | web | 0.24 | 143.67 | 0.008 | 0.03 |
| 5 | web | 0.17 | 145.13 | 0.03 | 0.08 |
| 6 | download | 4.74 | 1338.91 | 28.16 | 19.55 |
| 7 | web | 0.30 | 851.73 | 6.21 | 70.23 |
| 8 | web | 0.58 | 1213.87 | 6.38 | 137.60 |
| 9 | web | 1.59 | 714.42 | 22.58 | 52.91 |
| 10 | download | 0.39 | 307.92 | 1.68 | 0.82 |
| 11 | download | 10.66 | 809.29 | 22.74 | 10.75 |
| 12 | media | 0.43 | 110.22 | 14.13 | 24.41 |
| 13 | web | 0.0013 | 136.32 | 0.04 | 3.58 |
| 14 | media | 7.54 | 93.01 | 30.55 | 2.90 |
| 15 | media | 7.22 | 89.28 | 30.14 | 2.86 |
| 16 | media | 6.04 | 75.14 | 30.38 | 2.89 |
| 17 | media | 0.37 | 139.23 | 12.41 | 26.59 |
| 18 | web | 2.12 | 935.76 | 83.42 | 93.54 |
| 19 | media | 0.35 | 134.87 | 24.48 | 25.12 |
| 20 | download | 1.36 | 276.63 | 3.12 | 2.07 |
| 21 | media | 0.08 | 9.94 | 7.31 | 6.43 |
| 22 | media | 0.90 | 214.53 | 43.48 | 77.90 |
| 23 | media | 0.44 | 48.28 | 28.53 | 26.83 |
| 24 | media | 0.38 | 78.09 | 35.25 | 55.06 |
| 25 | download | 6.99 | 1879.65 | 44.94 | 21.02 |

Table 3.3: Traffic class characteristics

cache size. Every server within a cluster is also assumed to have equal capacity and cache size.

OPT is solved using CPLEX as part of the GAMS modeling language. We use a macOS machine with a 3 GHz Intel Xeon processor with 10 cores and 128 GB RAM for all our experiments. The GAMS program is set to run in parallel mode using 20 threads with a relative optimality gap of 1e-9 and a maximum run time of 40,000 s. Given the complexity of the optimization model, the GAMS program almost always runs for 40,000 s. At that point, the solver has converged to a solution that seldom changes and achieves a relative gap of under 5% at smaller cluster cache sizes less

than or equal to 10TB and a relative gap of under 10% at larger cache sizes. A single run of `baseline fit` takes about 1 s and a single run of `local search` takes about 120 s.

### 3.3.1 Metro-level traffic provisioning

We evaluate OPT, `baseline fit`, and `local search` by computing the cache miss rate of the entire 10-cluster metro area for different cache sizes. These three algorithms each assign the set of 25 input traffic classes to the clusters in the metro area. In the case of `baseline fit` and `local search`, we report the average cache miss rate of 100 runs, where each run considers the traffic classes in a random order. The 95% confidence intervals of the expected cache miss rates have a margin of error of less than 0.4%.

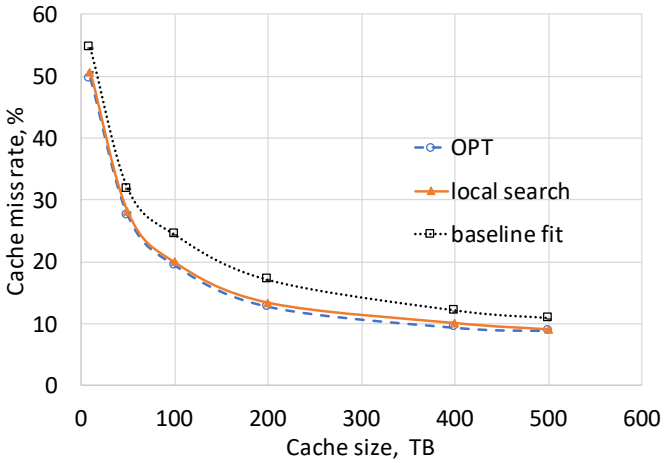


Figure 3.4: MRC of OPT, local search and baseline fit algorithms.

From Figure 3.4 we can see that OPT gives a 18.37% reduction in midgress on average compared to the midgress-unaware `baseline fit` algorithm. This is because OPT takes into account the impact on midgress while assigning traffic classes to clusters in the metro area. This significant improvement in midgress makes the case for implementing midgress-aware traffic provisioning algorithms in CDNs.

From Figure 3.4, we also see that the midgress-aware heuristic, `local search`, performs quite well and gives a 15.44% reduction in midgress on average compared to `baseline fit`. The algorithm `local search` also performs fairly well compared to OPT, with a modest 3.69% increase in midgress compared to OPT on average.

#### 3.3.1.1 Understanding how traffic provisioning can impact midgress

In Figure 3.5, we plot the cache miss rate of the 25 traffic classes when they are provisioned using OPT versus the midgress-unaware `baseline fit`, when the cumulative cache size of clusters in the metro area is 100TB. In addition, we also plot the average number of sites (over 100 different runs) that each traffic class is assigned to in Figure 3.6. From these figures, we see that OPT reduces the cache miss rate of 21 traffic classes when compared to `baseline fit`. In the case of traffic class 11, OPT results in almost 97% reduction in miss rate when compared to `baseline fit`. On the other hand, OPT increases the cache miss rate of four traffic classes, namely traffic classes 4, 5, 13 and 19. By trading off the cache miss rates for these four traffic classes, OPT is able to reduce the overall midgress. But why does OPT choose this trade-off? There are three key insights that midgress-aware traffic provisioning takes into account to optimize midgress that `baseline fit` does not.

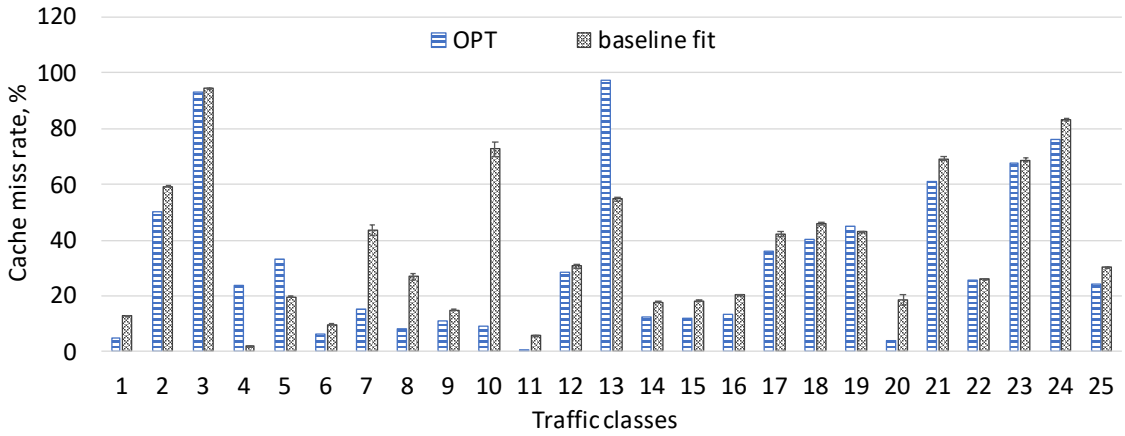


Figure 3.5: Average miss rate of each traffic class in a metro area of cache size 100 TB.

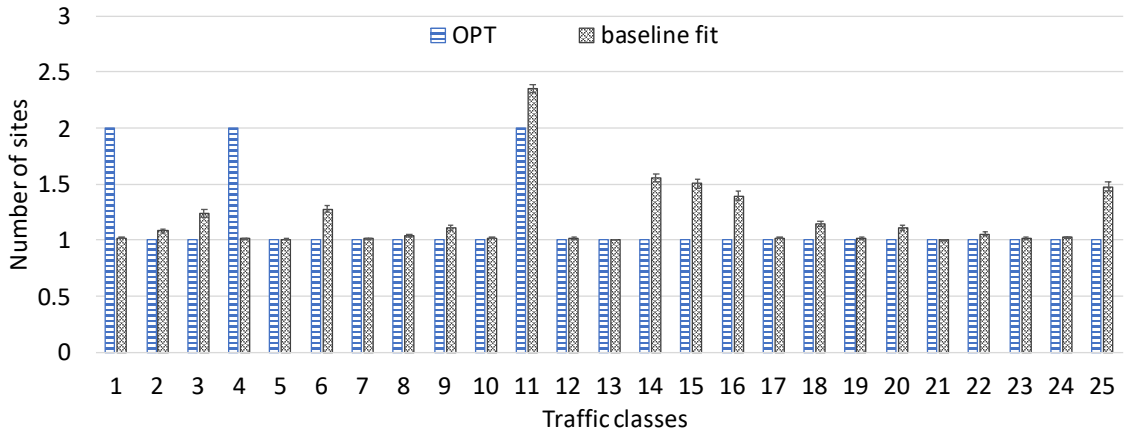


Figure 3.6: Average number of sites each traffic class is assigned to in a metro area of cache size 100 TB.

1) *In OPT's solution, traffic classes that have higher load, higher footprint pressure and greater MRC gradients get to occupy larger portions of the available cache space.* A traffic class has high footprint pressure if a large amount of unique content bytes is requested in a short period of time. This is true for traffic classes 11, 14, 15, 25 and 16 that account for 66.04% of the total load. OPT assigns traffic class 11 to two clusters because its load is greater than the capacity of a single cluster, resulting in that traffic class occupying 6 TB in one cluster with a miss rate of 0.5% and 7 TB in another cluster with a miss rate of nearly 0%. OPT also assigns an entire cluster each to traffic classes 14, 15, 25 and 16.

2) *OPT may split a traffic class and may assign it to multiple clusters if it has a relatively flat MRC.* This is true of traffic class 1 which has a relatively flat MRC and is assigned to two clusters. By reducing its footprint pressure in each of its assigned clusters, traffic class 1 is able to cede cache space to other traffic classes that are in more need.

3) *In OPT's solution, traffic classes that have lower footprint pressure occupy smaller portions of the available cache space.* This is true for traffic classes 4, 5 and 13. It also happens to be the case that these three traffic classes have very low load

among the traffic classes considered. Both these factors render a higher cache miss rate relative to `baseline fit` that is footprint unaware. Note that low load alone does not indicate that it will occupy a smaller portion of the cache. For instance, traffic class 24 has moderate load but it has high footprint pressure and a greater MRC gradient, and ends up occupying 4.2 TB in one cluster.

### 3.3.2 Cluster-level traffic provisioning

The goal of cluster-level traffic provisioning is to assign traffic classes to servers such that the midgress of the cluster is minimized. In our evaluation, we take the output of metro-level traffic provisioning from Section 3.3.1 that assigns traffic classes to each cluster and treat them as the inputs to cluster-level traffic provisioning. In this manner, we are able to understand the additional midgress reduction that is achievable by performing optimization at the cluster level, given that the metro level has already been optimized.

For cluster-level traffic provisioning, each traffic class defined at the metro-level is typically split into multiple subclasses. The subclasses allow better finer-grained allocation of traffic classes within a cluster. Traffic class 14 has very high load and hence was assigned to a cluster all by itself (Figure 3.6) by OPT at the metro-level. We considered that cluster for our evaluation of cluster-level traffic provisioning. Traffic class 14 consisted of 66 traffic subclasses that must be assigned to the 10 servers within a cluster, each server with a 1 TB cache.

OPT reduced the midgress for traffic class 14 by 31.26% after the metro-level optimization, when compared to the midgress achieved by `baseline fit`. After using OPT for cluster-level provisioning, the midgress for traffic class 14 reduced further by 14.26%. In aggregate, the overall reduction of the midgress due to both provisioning steps of OPT is 41.07%, when compared to the baseline. Algorithm `local search` provided nearly as much reduction as OPT. For instance, `local search` provided

a midgress reduction of 35.49%, compared to the baseline. However, `local search` was much faster and completed within 2 minutes, as opposed to the nearly 40,000 s ( 11 hours) taken by OPT.

### 3.3.3 Robustness to variations in cache management policy

So far, we have developed traffic provisioning algorithms that model an LRU cache and evaluated the midgress reduction resulting from midgress-aware traffic provisioning when the sites also use LRU. The past decades have seen much academic research on numerous cache management algorithms that admit and evict objects using some combination of *recency* of access, *frequency* of access and object *size* to decrease cache miss rates of individual caches (see Table 2 of [12]). We show that midgress-aware traffic provisioning algorithms proposed in this work, that model an LRU cache, achieve significant midgress reduction even when a CDN *does not* actually implement LRU at its sites, although such reductions are smaller in magnitude than when a CDN implements LRU.

We choose three typical algorithms from the literature for our evaluation. The first is an LRU variant called second-hit-LRU (or, SH-LRU) where the object is admitted to an LRU cache on second hit as means of filtering out "one-hit-wonder" objects that are accessed once and never again. The second is segmented LRU (SLRU) [68] that is a canonical representative of the family of algorithms that use both recency and frequency in cache management decisions. Finally, we implement the Greed-Dual-Size-Frequency (GDSF) [27] that is a representative of algorithms that use all three of recency, frequency and size. Our evaluation uses the same cluster-level scenario as described in Section 3.3.2, where the goal is to assign the 66 traffic subclasses of traffic class 14 across 10 servers of size 1 TB each. First, we solve OPT that models LRU to get the optimal traffic class assignment across all servers within the cluster. The midgress of OPT's assignment is then computed by simulating the different cache

management algorithms using the request traces of the subclasses. For comparison, we use the midgress-unaware `baseline fit` for traffic provisioning followed by a trace-based simulation of the different cache management algorithms to provide a midgress-unaware baseline. When LRU cache management is used, OPT reduces the midgress by 13.3% when compared to `baseline fit`. In comparison, OPT reduces the midgress by 7.78%, 8.45% and 7.76% for SH-LRU, SLRU and GDSF respectively. The midgress reduction for other non-LRU algorithms is not as much as that for LRU. However, the midgress reductions for other algorithms are still quite robust and significant. The reason is that even when other factors are used for cache management decisions, most reasonable algorithms *still* use recency of access in a very significant way, and recency is well-captured in our OPT model. It is plausible that our OPT model can be reformulated to capture other cache management policies besides LRU, but it is not clear how to model the behavior of other algorithms in a constraint satisfaction framework. Such an extension is a topic for future work.

## 3.4 Extensions of midgress-aware traffic provisioning

While OPT in Section 3.1.2 minimizes the midgress across a metro area, OPT may assign certain traffic classes to only a single site. For example, from Figure 3.6, we see that OPT assigns 22 out of 25 traffic classes only to one cluster. In that case, a failure of that site could lead to severe degradation in performance for users in that metro accessing that traffic class. In the case of site failure, those users would need to be served from a cluster not located in their metro, leading to higher latencies. Another issue is that even though overall miss traffic is reduced by OPT, certain traffic classes may experience unacceptably high miss rates. As seen in Figure 3.5, traffic classes 3, 13 and 24 have miss rates greater than 70%.

We propose two extensions of midgress-aware traffic provisioning that address the above two issues. The first extension enforces a minimum number of sites that a

traffic class must be assigned to and the second extension enforces a maximum cache miss rate per traffic class.

Finally, all results presented until now are for shared caches. While partitioned caches are not commonly used in production settings due to the overheads involved in dynamically resizing those partitions, there is increasing interest in academia to implement and evaluate the performance of partitioned caches [5, 14, 19, 28, 31, 36, 44, 63, 71, 73, 76, 100, 110, 121]. We propose a third extension to show that our traffic provisioning approach can significantly reduce midgress even if the CDN were to use partitioned caches.

### 3.4.1 Minimum redundancy guarantee

Enforcing a minimum redundancy in the optimization model is fairly straightforward. Let $M_j$ be the minimum number of sites that traffic class $j$ should be assigned to. $M_j$ is an integer $\in [1, M]$, where $M$ is the total number of sites. Let $y_{ij}$ be an indicator variable that is set to 1 when $x_{ij} > 0$ and 0 otherwise. Then, the load assignment constraint in Section 3.1.2 is appended to include the following minimum redundancy constraints.

$$y_{ij} = \lceil x_{ij} \rceil \quad \forall\, j = 1 \ldots N \tag{3.9}$$

$$\sum_{i=1}^{M} y_{ij} >= M_j \quad \forall\, j = 1 \ldots N \tag{3.10}$$

$$y_{ij} \in \{0, 1\} \tag{3.11}$$

Equations 3.9 and 3.10 ensure that traffic class $j$ is assigned to at least $M_j$ sites. We call the modified optimization model OPT-M. The additional constraints are affine and they do not increase the complexity of the optimization problem. OPT-M can also be solved using CPLEX. Likewise, both `local search` and `baseline fit`

can be modified to incorporate the redundancy constraint by simply ensuring that each traffic class $j$ is assigned to at least $M_j$ sites in each (re-)provisioning step.

#### 3.4.1.1 Experimental evaluation

We measure the reduction in midgress by OPT-M and the modified `local search` when compared to the modified `baseline fit`, all of which include the minimum redundancy constraint. We use the same evaluation parameters as Section 3.3.1 where the cache size of each cluster in the metro area is 10 TB. Figure 3.7 plots the trade-off between minimum redundancy and cache miss rates of OPT-M, and the modified versions of `local search` and `baseline fit`. We vary the minimum required redundancy from 1 to 3. From Figure 3.7, we see that cache miss rates increase for all three algorithms, resulting in an increased midgress and increased bandwidth cost for the CDN. We also see that the cache miss rate of `baseline fit` with minimum redundancy = 1 (resp. 2) is similar to the cache miss rate of `local search` and OPT-M with minimum redundancy 2 (resp. 3). This shows that midgress-aware traffic provisioning can provide the same midgress as `baseline fit` with added redundancy.

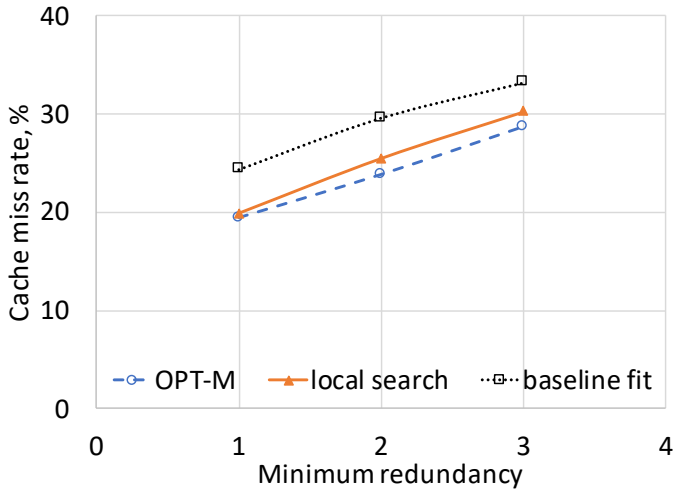


Figure 3.7: Increasing minimum redundancy increases the overall midgress of the metro area.

### 3.4.2 Maximum cache miss rate guarantee

In practice, CDN operators need to guarantee a maximum cache miss rate for each traffic class. Let $MR_j$ be the maximum cache miss rate for traffic $j$. Then, the optimization model in Section 3.1.2 can be easily extended to incorporate the maximum cache miss rate guarantee.

$$\sum_{i=1}^{M} x_{ij} m_j(c_{ij}) \leq MR_j \quad \forall j = 1 \ldots N \tag{3.12}$$

Equation 3.12 states that the average miss rate of traffic class $j$ across all $M$ sites should be at most $MR_j$. This enforces the maximum cache miss rate guarantee for traffic class $j$. We call the modified optimization model, OPT-MR. Setting $MR_j = 100\%$ for all traffic classes $j$ is equivalent to OPT in Section 3.1.2. Equation 3.12 is a biconvex constraint and does not increase the complexity of the optimization model.

We modify the `local search` algorithm (Algorithm 4) to take into account the maximum cache miss rate constraint per traffic class. We make two modifications. First, the `baseline fit` algorithm in the first step does not always provide a feasible solution when $MR_j < 100\%$. This is because `baseline fit` is midgress unaware. Hence, we start with all traffic classes being unassigned. Second, the re-provisioning step as part of the local search assigns a traffic class to a site only when the miss rate guarantees of all traffic classes assigned to that site are not violated.

*Infeasible solutions:* OPT-MR can be infeasible in cases where certain traffic classes fail to meet the maximum cache miss rate guarantee, $MR_j$. For example, consider the evaluation set up in Section 3.3.1 where the cache size of each cluster is 10 TB. From Figure 3.5, we see that traffic class 3 has a miss rate of 92.88%. The lowest miss rate that traffic class 3 can possibly achieve at 10 TB is 91%. Hence, any maximum cache miss rate target less than 91% cannot be achieved, rendering the problem infeasible.

#### 3.4.2.1 Experimental evaluation

We choose three traffic classes 13, 23 and 24 that have high miss rates in OPT and set their maximum cache miss rates to 70%. We evaluate the performance of metro-level traffic provisioning with the maximum cache miss rate constraint under the same conditions as in Section 3.3.1 where the cache size of each cluster is 10 TB.

OPT-MR returns a feasible solution. The overall miss rate of the metro area with these constraints is 20.04%, a modest 3.24% increase in midgress compared to OPT that has no maximum miss rate constraint. In the process, three traffic classes experience a significant increase in their respective miss rates relative to OPT. The miss rate of traffic class 2 increases from 50.12% to 65.85%, of traffic class 17 from 36.11% to 54.2% and of traffic class 21 from 61.22% to 68.01%. This is due to the fact that both traffic classes 13 and 24 occupy more cache space with OPT-MR than they do with OPT, so they meet their miss rate guarantee, despite traffic class 13 having the lowest load and low footprint pressure, and traffic class 24 having low load.

We run `local search` with the maximum cache miss rate constraint 100 times with different random orderings of the input traffic classes. `Local search` returns a feasible solution 67% of the time indicating that the feasibility of `local search` depends on the ordering of the traffic class inputs. For feasible assignments, `local search` has an average miss rate of 21.69% which is about 8.23% more than that of OPT-MR. `Baseline fit` is footprint-unaware and cannot guarantee a cache miss rate performance.

### 3.4.3 Traffic provisioning in partitioned caches

In a partitioned cache, each traffic class is assigned to its own separate cache partition and each partition performs evictions independently, i.e., each partition has its own LRU list for eviction. As noted earlier, production CDNs do not typically implement partitioned caches due to the significant overheads involved in implementing

and dynamically maintaining the partitions. However, we show that our optimization model for midgress-aware traffic provisioning can be extended to work with partitioned caches. More specifically, we answer the following questions in this section: 1) Is it possible to model midgress-aware traffic provisioning for partitioned caches? 2) Does our traffic provisioning approach for partitioned caches reduce the cache miss rate when compared to a midgress-unaware baseline?, and 3) How does the cache miss rate of midgress-aware traffic provisioning in partitioned caches compare with that of midgress-aware traffic provisioning in shared caches?

#### 3.4.3.1 Modeling and implementing traffic provisioning for partitioned caches

With cache partitioning, every traffic class occupies a separate cache partition with its own LRU list, assuming that the LRU eviction policy is used. Thus, the eviction age of each traffic class can be different. This is unlike shared caches where all traffic classes use an unpartitioned cache and hence have the same eviction age. Therefore, the optimization model for midgress-aware traffic provisioning for partitioned caches is the same as that of OPT (Section 3.1.2), except that we can remove the eviction age equality constraint. Hence, Equations 3.1-3.3 and 3.5-3.8 accurately model the constraints for optimizing midgress for partitioned caches. We call this modified optimization algorithm as OPT-part.

Likewise, we can implement a baseline midgress-unaware algorithm for partitioned caches that is similar to `baseline fit` (Algorithm 3). We call this algorithm `baseline fit-part` and is based on consistent hashing. The algorithm takes as input the set of $N$ traffic classes and the set of $M$ sites that are both hashed to points on a unit circle. Each traffic class $j$ is assigned to the nearest site $i$ on the unit circle in the clockwise direction. If the chosen site $i$ does not have enough capacity to host the entire load $\lambda_j$, then a first fit algorithm is used, starting from the chosen site $i$,

and continuing to subsequent sites on the unit circle in the clockwise direction, until all traffic is assigned. After all traffic class assignments are made, for each site we determine the sizes of the partitions that host each traffic class assigned to it. To compute the partition sizes, we use a known gradient descent algorithm [100] that minimizes the total midgress of that site. The total midgress achieved by `baseline fit-part` is obtained by adding the midgress across all $M$ sites.

#### 3.4.3.2 Experimental evaluation

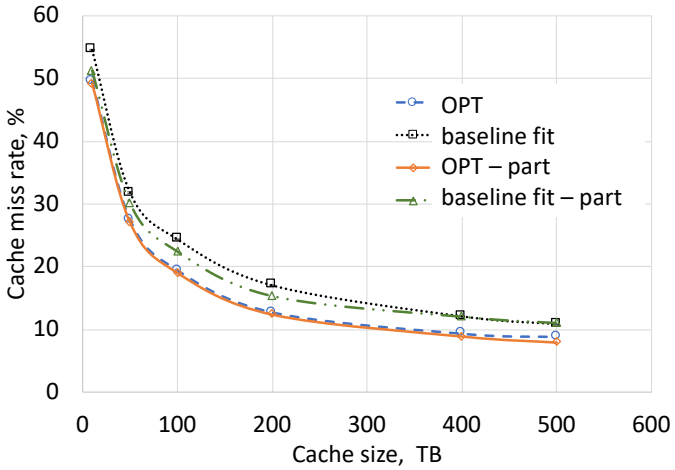


Figure 3.8: MRC of OPT and baseline fit on shared and partitioned caches.

We use production traces collected from a metro area in Akamai's CDN to evaluate the cache miss rates of partitioned caches at the metro-level for different cache sizes. The traces are described in Section 3.3. We evaluate both `baseline fit-part` and OPT-part on these traces and report the average cache miss rate of 100 runs. The 95% confidence intervals of the expected cache miss rates have a margin of error of less than 0.4%.

As shown in Figure 3.8, we see that OPT-part reduces midgress when compared to `baseline fit-part` by more than 14%, on average, across the different cache

sizes. Thus midgress-aware traffic provisioning can significantly reduce the midgress for partitioned caches in comparison with a midgress-unaware algorithm.

As comparison, in Figure 3.8, we also plot OPT and `baseline fit` that we described for shared caches in Sections 3.1 and 3.2. The cache miss rate of OPT-part is only 0.49% less than that of OPT, on average across the different cache sizes. Hence, while OPT-part has the lowest cache miss rate, OPT has nearly the same miss rate without the additional overhead of cache partitioning.

## 3.5 Related work

Traffic provisioning in CDNs has been studied in the context of distributing load to servers in a CDN. However, the load balancing algorithms studied in this context focus on ensuring that servers are not overloaded in terms of CPU, disk, and other server resources and do not explicitly optimize for midgress when performing traffic provisioning. Likewise, minimizing cache misses through better cache management policies has a rich literature stretching several decades. However, we view cache management as complementary to midgress-aware traffic provisioning. We review relevant existing literature in these and other areas below.

*Load balancing:* Several load balancing algorithms have been proposed to improve the end-user performance in web caching and content delivery systems. Request redirection schemes at the network layer, based on DNS [18,56], and at the application layer, based on URL rewriting or HTTP redirection [82], have been proposed to better load balance traffic across multiple servers. Dynamic load balancing algorithms [21, 25, 124] continuously measure the load on different servers and load balance end-user requests to improve performance. Consistent-hashing and randomized load balancing algorithms [69,70,88,89] have also been proposed to load balance end-user requests in content delivery systems. As an extension to traditional load balancing, the authors in [84] design load balancing algorithms that minimize the energy consumption of

CDNs without affecting the end-user performance. Much of the above work are in the context of routing user requests in real-time to servers that can serve the requested traffic classes. But they can be adapted to our context of performing (offline) traffic provisioning, a step that precedes request routing in a production CDN. *However, there is no prior work on explicitly optimizing midgress.*

*Cache management:* There has been a significant amount of research on cache management policies to minimize the miss rate of a cache [10, 11, 15, 33–35, 39, 45, 46, 48, 50, 53, 55, 59, 64, 65, 74, 83, 86, 94, 95, 98, 102, 112, 122]. Some proposed caching policies include Adapt-Size [12], Cliffhanger [30], SLRU [68], TinyLFU [40], S4LRU [61], CFLRU [96], ARC [87], LRU-S [104], LRU-K [94], and GDS [17]. Dynamically partitioning the cache to reduce miss rates has also been explored [5, 14, 19, 28, 31, 36, 44, 63, 71, 73, 76, 100, 110, 121]. However, production CDNs do not employ dynamically-partitioned caches, since it introduces significant performance and operational overhead to maintain partitions for hundreds of traffic classes. We view work on cache management as a complementary technique to traffic provisioning, both with the goal of midgress reduction.

Recent work on footprint descriptors [107] is focused on efficient techniques for evaluating the miss traffic of an assignment of traffic classes to a server. We use footprint descriptors for quickly assessing the midgress caused by a specific traffic assignment, as well as for efficiently computing the MRC and eviction age function of a large number of traffic classes. However, the work in [107] does not propose any mechanisms for traffic provisioning to minimize midgress.

## 3.6 Conclusion

In this work, we propose midgress-aware traffic provisioning that explicitly minimizes the midgress traffic of a CDN, while still ensuring that no server or cluster is overloaded. Using extensive traces for 25 traffic classes from a metro area of Aka-

mai's CDN, we show that the midgress of a metro area can be reduced by 18.37% when compared to a midgress-unaware baseline that is currently used in practice. We propose a midgress-aware heuristic algorithm called `local search` that provisions traffic classes to achieve a midgress reduction that is within 1.1% of the optimum, and is very fast and well suited for production settings. We also show that using our traffic provisioning algorithms at the cluster level results in significant reductions in midgress. Given that a large CDN can have midgress of over 10Tbps, even a small reduction in midgress can result in millions of dollars of savings per year. Our work provides a strong case for implementing midgress-aware traffic provisioning in CDNs. As part of our future work, we intend to deploy our traffic provisioning algorithms in a production setting to evaluate its benefits.

# CHAPTER 4

# ADAPTIVE CACHE MANAGEMENT USING TTL-BASED CACHING

By caching and delivering content to millions of users around the world, CDNs are an integral part of the Internet infrastructure. The major technical challenge in designing caching algorithms for a modern CDN is *adapting* to heterogeneous, bursty (correlations over time) and non-stationary/transient request statistics of the different traffic classes served by a CDN. The traffic classes differ widely in terms of the object size distributions and content access patterns. The popularity of the content also varies by several orders of magnitude with some objects accessed millions of times (e.g., an Apple iOS download), and other objects accessed once or twice (e.g., a photo in a Facebook gallery). This non-uniformity makes it difficult to guarantee performance metrics such as a target hit rate that is desired by the CDN operator and the content providers who use the CDN.

Request statistics clearly play a key role in determining the hit rate of a CDN server. However, when request patterns vary rapidly across servers and time, a one-size-fits-all approach provides inferior hit rate performance in a production CDN setting. Further, manually tuning the caching algorithms for each individual server to account for the varying request statistics is prohibitively expensive. Thus, our goal is to devise *self-tuning* caching algorithms that can automatically learn and adapt to the request traffic and *provably* achieve any feasible hit rate and cache size, even when the request traffic is bursty and non-stationary.

A class of caching algorithms that has received much attention in the last decade are TTL-based caching algorithms [10, 11, 36, 47, 52, 66] that use a time-to-live (TTL)

parameter to determine how long an object may remain in cache. Much is analytically known about TTL caches when the traffic is assumed to follow the independent reference model (IRM) [10, 47]. The cornerstone of such analyses is Fagin's work [42] and follow up work by Che [24] that provide a closed-form characteristic time approximation that relates the TTL value to the achieved hit rate and average cache size. The characteristic time approximation is known to be accurate in cache simulations that use synthetic IRM traffic and is commonly used in the design of caching algorithms for that reason [13, 48, 49, 52, 57].

However, we show that the characteristic time approximation does not provide performance guarantees for actual production CDN traffic that is neither stationary nor respects IRM, similar to the observations made in [54, 79], among others. We used an extensive 9-day request trace from a production server in Akamai's CDN and derived TTL values for multiple hit rate and cache size targets using Che's approximation. We then simulated a cache with those TTL values on the production traces to derive the *actual* hit rate that was achieved. For a target hit rate of 60%, we observed that a fixed-TTL algorithm that uses the TTL computed from Che's approximation achieved a hit rate of 68.23% whereas the dynamic TTL algorithms proposed in this work achieve a hit rate of 59.36%. This difference between the target hit rate and that achieved by fixed-TTL highlights the inaccuracy of the characteristic time approximation in achieving performance targets on production traffic.

– *Contributions:* We make the following contributions in this work.

**1)** We propose two TTL-based algorithms: d-TTL (for "dynamic TTL") and f-TTL (for "filtering TTL"). d-TTL uses a single TTL parameter to achieve a target hit rate. f-TTL uses two TTL parameters to achieve a target hit rate and a target expected cache size, provided they are feasible. Rather than statically deriving the required TTL values from the request statistics, our algorithms *incrementally adapt* the TTL values after each request, using stochastic approximation.

**2)** We implement both d-TTL and f-TTL algorithms and evaluate them using an extensive 9-day trace consisting of more than 500 million requests from a production Akamai CDN server. For a range of object hit rate targets, both d-TTL and f-TTL converge to that target with an error of about 1.3%. For a range of byte hit rate targets, both d-TTL and f-TTL converge to that target with an error that ranges from 0.3% to 2.3%. In particular, f-TTL requires a cache that is 49%(resp. 39%) smaller than d-TTL to achieve the same object hit rate(resp. byte hit rate).

**3)** Finally, from a practitioner's perspective, this work has the potential to enable new CDN pricing models. CDNs would like to charge content providers based on guaranteed hit rate performance and the guaranteed caching resources that are used. While desirable, such pricing models do not commonly exist, in part, because current caching algorithms cannot provide such guarantees with low overhead. The proposed algorithms are the first to provide a theoretical guarantee on hit rate for each content provider, while controlling the cache space that they can use.

– *Roadmap:* The rest of this chapter is organized as follows. In Section 4.1, we discuss two TTL-based algorithms namely d-TTL and f-TTL that achieve a target hit rate and a target expected cache size. In Section 4.2, we describe how d-TTL and f-TTL are implemented in practice. We evaluate the performance of the proposed TTL-based algorithms in Section 4.3. Finally, we discuss some related work in Section 4.4 before concluding in Section 4.5.

## 4.1 TTL-based caching algorithms

A TTL-based caching algorithm works as follows. When a new object is requested, it is placed in cache and associated with a time-to-live (TTL) value. If no new request is received for that object, the TTL value is decremented in real-time and the object is evicted when the TTL becomes zero. If a cached object is requested, a *cache hit*

occurs and the TTL is reset[1] to its original value. When the requested object is not found in cache, a *cache miss* occurs. Consider $T$ different types of content. The *objective* of this work is to (asymptotically) achieve a *target hit rate* and a (feasible) *target cache size* for each type $t \in [T]$. To accurately model non-stationary traffic, we allow the request traffic to be non-independent and non-stationary; the request traffic can have Markovian dependence over time. The traffic comprises a mix of stationary demands (statistics invariant over the timescale of interest), and non-stationary demands (finitely many requests, or in general requests with an asymptotically vanishing request rate). The complete model is described in [7].

### 4.1.1 d-TTL

The goal of the d-TTL algorithm is to adapt its TTL value to achieve a target hit rate[2] $h_t^* \in (0,1)\ \forall t \in [T]$. d-TTL uses stochastic approximation to dynamically increase the TTL when the current hit rate is below the target, and decrease the TTL when the current hit rate is above the target. Let $\theta_t(l)$ be the TTL value after the $l$-th request arrival for content type $t$. Then, if the object experiences a cache miss, d-TTL increases the TTL to, $\theta_t(l+1) = \theta_t(l) + \eta(l)(h_t^*)$, where $\eta(l)$ is a decaying step size of the form $1/l^\alpha$ and $\alpha$ is a constant. On the contrary, in the event of a cache hit, the TTL decreases to $\theta_t(l+1) = \theta_t(l) - \eta(l)(1 - h_t^*)$.

### 4.1.2 f-TTL

d-TTL achieves the target hit rate at the expense of caching rare and unpopular recurring content for an extended period of time. This leads to an increase in cache size without any significant contribution towards the cache hit rate. We present a

---

[1]There are other classes of TTL algorithms that do not reset the TTLs of objects on each request. We only consider reset TTLs in this work.

[2]We consider object hit rates to illustrate the TTL algorithms. d-TTL and f-TTL easily extend to byte hit rates as discussed in [7]

second adaptive TTL algorithm called filtering TTL (f-TTL) that filters out rare and unpopular content to achieve both a target expected cache size and a target hit rate. Thus, f-TTL can achieve the same hit rate as d-TTL but at a smaller cache size.

#### 4.1.2.1 Algorithm description

f-TTL implements a two-level caching algorithm. Let's call the lower level cache $\mathcal{C}_s$ and the higher level cache $\mathcal{C}$. Both these caches store the entire content, i.e. the content metadata such as the object id, object size, etc., and the actual content data. In addition to these two caches, f-TTL maintains a shadow cache that only stores the metadata of the content that is requested, for some period of time. We describe the f-TTL algorithm by discussing its different caching states.

*1) Cache miss.* An incoming request experiences a cache miss when it is unavailable in both cache levels, $\mathcal{C}_s \cup \mathcal{C}$ and its metadata is unavailable in the shadow cache. In this case, the request is cached with a small TTL value in $\mathcal{C}_s$ and its metadata is cached with a larger TTL value in the shadow cache.

*2) Cache virtual hit.* An incoming request experiences a cache virtual hit when it is unavailable in both cache levels, $\mathcal{C}_s \cup \mathcal{C}$ but its metadata is available in the shadow cache. This is an indication that f-TTL prematurely evicted the object from $\mathcal{C}_s$ affecting the cache hit rate. The metadata is removed from the shadow cache and the request is cached with a larger TTL value in $\mathcal{C}$.

*3) Cache hit.* An incoming request experiences a cache hit when it is available in either cache level, $\mathcal{C}_s$ or $\mathcal{C}$. In this case, the request is cached with an updated TTL value in $\mathcal{C}$. Any existing copy in $\mathcal{C}_s$ or its metadata in the shadow cache is removed.

From the above caching states, it can be seen that f-TTL attempts to cache popular content in the higher level cache for a longer duration and filters out rare and infrequently accessed content from the lower level cache quickly. The shadow cache helps promote semi-popular content to the higher level cache.

#### 4.1.2.2 Achieving the target cache size

Before we describe the TTL update rules, we introduce a new performance metric called *normalized cache size* that is used by f-TTL to achieve the target expected cache size. Given a cache size $C$ and arrival rate $\lambda_t$ for some content of type $t$, the normalized cache size, $s_t$, is the cache size normalized by the arrival rate, i.e., $s_t = \frac{C}{\lambda_t}$. The normalized cache size can be thought of as the time it takes to fill up a cache of size $C$ when requests arrive at arrival rate $\lambda_t$. Hence, smaller (larger) the arrival rate, larger (smaller) the normalized cache size.

The normalized cache size can also be thought of as the average time to eviction of requests that reside in a cache of size $C$. To see why this is the case, let us consider a request sequence where (adaptive) TTL value $\theta_t(l)$ is assigned to each request $l$ for content of type $t$. For simplicity, we assume that objects have unit size but the argument easily extends to objects with variable sizes. Then, the instantaneous cache size is the number of objects that have non-zero TTLs at that time instant. Given a total time duration $\tau$ and total number of requests $L_t$ of type $t$, it is easy to see that the time average expected cache size $C$ is

$$C = \frac{\sum_{l=1}^{L} \theta_t(l)}{\tau} = \frac{\sum_{l=1}^{L} \theta_t(l)}{L_t} \times \frac{L_t}{\tau} = \frac{\sum_{l=1}^{L} \theta_t(l)}{L_t} \times \lambda_t.$$

From this we see that the normalized cache size $s_t$ is,

$$s_t = \frac{C}{\lambda_t} = \frac{\sum_{l=1}^{L} \theta_t(l)}{L_t}.$$

Since $s_t$ indicates the average time to eviction of all incoming requests in a cache of size $C$, it can be used by f-TTL to achieve a target cache size.

#### 4.1.2.3 TTL update rules

The goal of f-TTL is to adapt its TTL timers to achieve the dual targets of hit rate $h_t^*$ and normalized cache size $s_t^*$ for content type $t$. We associate with each cache

level a TTL value, $\theta_t^s(l)$ with the lower level cache $\mathcal{C}_s$ and $\theta_t(l)$ with both the higher level cache $\mathcal{C}$ and the shadow cache. When an object of type $t$ is requested, the TTL values are updated based on the caching states as follows.

*1) Cache miss.* The requested object is not found in $\mathcal{C}_s \cup \mathcal{C}$ and the shadow cache. This indicates that $\theta_t(l)$ needs to be increased to meet the target hit rate $h_t^*$. The update for $\theta_t(l)$ is the same as d-TTL, $\theta_t(l+1) = \theta_t(l) + \eta(l)(h_t^*)$.

In order to update $\theta_t^s(l)$, we need to estimate the normalized cache size at the $l^{th}$ request arrival for content type $t$. Since the requested object is in none of the caches, the normalized cache size estimate at the $l^{th}$ arrival is $\theta_t^s(l)$, the timer of lower level cache that this request gets into. Hence, the update rule for $\theta_t^s(l)$ is $\theta_t^s(l+1) = \theta_t^s(l) + \eta^s(l)(s_t^* - \theta_t^s(l))$ where $\eta^s(l)$ is a decaying step size of the form $1/l$.

*2) Cache virtual hit.* The requested object is not found in $\mathcal{C}_s \cup \mathcal{C}$ but its metadata is in the shadow cache. This indicates that $\theta_t(l)$ needs to be increased to meet the target hit rate $h_t^*$. The update rule for $\theta_t(l)$ is the same as the cache miss scenario. However, the update rule for $\theta_t^s(l)$ is different. During a cache virtual hit, f-TTL estimates the normalized cache size at the $l^{th}$ request arrival as the timer corresponding to the upper level cache $\mathcal{C}$, $\theta_t(l)$, that this request gets into. Hence, the update rule for $\theta_t(l)$ is $\theta_t^s(l+1) = \theta_t^s(l) + \eta^s(l)(s_t^* - \theta_t(l))$.

*3) Cache hit.* The requested object is found in $\mathcal{C}_s \cup \mathcal{C}$. This indicates that $\theta_t(l)$ needs to be decreased similar to d-TTL. The update rule is $\theta_t(l+1) = \theta_t(l) - \eta(l)(1 - h_t^*)$. The normalized cache size estimate at the $l^{th}$ request arrival during a cache hit is $\theta(l) - \phi$ where $\theta(l)$ is the timer corresponding to upper level cache and $\phi$ is the remaining time for the object. In other words, $\theta(l) - \phi$ is the smallest TTL of the upper level cache to guarantee a cache hit on the $l^{th}$ arrival. Hence, the update rule for $\theta_t^s(l)$ is $\theta_t^s(l+1) = \theta_t^s(l) + \eta^s(l)(s_t^* - (\theta_s(l) - \phi))$.

While the above description provides an intuitive understanding of f-TTL, the simple update rules above do not take into account scenarios where neither the target

hit rate nor the target normalized cache size are achieved even if either one of them is feasible. The complete algorithm description with the necessary conditions for convergence is discussed in [7]. The proofs of convergence for both d-TTL and f-TTL algorithms are also presented in [7].

## 4.2 Implementation of d-TTL and f-TTL

One of the main practical challenges in implementing d-TTL and f-TTL is adapting $\theta$ and $\theta^s$ (TTL vectors corresponding to the higher and lower level caches for all content types $t \in [T]$) to achieve the desired hit rate and cache size in the presence of unpredictable non-stationary traffic. We observe the following major changes in practical settings. First, the arrival process in practice changes over time (e.g. day-night variations) whereas our model assumes the arrival process for stationary traffic is fixed. Second, in practice, the performance in finite time horizons is of particular interest such as flash crowds in specific time windows. We now discuss some modifications we make to translate theory to practice and evaluate these modifications in Section 4.3.

– *Fixing the maximum TTL:* The theoretical analyses of the TTL algorithms require choosing a suitable parameter $\boldsymbol{L}$ as the maximum TTL value [7]. However, in practice, we can choose an arbitrarily large value such that we let $\theta$ and $\theta^s$ explore a larger space to achieve the desired hit rate in both d-TTL and f-TTL.

– *Constant step sizes for $\theta$ and $\theta^s$ updates:* d-TTL and f-TTL algorithms use decaying step sizes $\eta(l)$ and $\eta^s(l)$ while adapting $\theta$ and $\theta^s$. This is not ideal in practice where the traffic composition is constantly changing, and we need $\theta$ and $\theta^s$ to capture those variations. Therefore, we choose constant step sizes that capture the variability in our production traces while evaluating the algorithms.

– *Tuning cache size targets:* In practice, f-TTL may not be able to achieve small cache size targets in the presence of time varying and non-negligible non stationary traffic.

In such cases, f-TTL uses the target cache size to induce filtering. For instance, when there is a sudden surge of non-stationary content, $\theta^s$ can be aggressively reduced by setting small cache size targets. This in turn filters out a lot of non-stationary content while an appropriate increase in $\theta$ maintains the target hit rate. Hence, the target cache size can be used as a tunable knob in CDNs to adaptively filter out unpredictable non-stationary content. It should be noted that a target cache size of 0 while most aggressive, is not necessarily the best target. This is because, a target cache size of 0, sets $\theta^s$ to 0 and essentially increases the $\theta$ to attain the target hit rate. This may lead to an increase in the average cache size when compared to an f-TTL implementation with non-zero target cache size. For example, in our experiments, we observe that when we set a target hit rate to 40%, a non-zero target cache size leads to an expected cache size that is nearly 15% less than the expected cache size obtained when the target cache size is zero.

## 4.3 Experimental evaluation

We use an extensive data set containing access logs for content requested by users that we collected from a production server serving predominantly web content in Akamai's CDN, over a period of 9 days. The content requests traces contain 504 million requests (resp., 165 TB) for 25 million distinct objects (resp., 15 TB). We observe that the content popularity distribution exhibits a "long tail" with nearly 70% of the objects accessed only once. Further, we also see that 80% of the requests are for 1% of the objects. This indicates the presence of a significant amount of non-stationary traffic in the form of "one-hit-wonders" and rarely accessed content. We use constant step sizes, $\eta$=1e-2 and $\eta^s$=1e-9, while adapting the values of $\theta$ and $\theta^s$ respectively, in all our simulations. For f-TTL, we use a target cache size that is half of what is required by d-TTL to achieve a certain hit rate.

### 4.3.1 Hit rate performance of d-TTL and f-TTL

The performance of a caching algorithm is often measured by its hit rate curve (HRC) that relates the cache size to the hit rate it achieves. We compare the HRCs of d-TTL and f-TTL for object hit rates and byte hit rates and show that f-TTL significantly outperforms d-TTL by filtering out the rarely-accessed non-stationary content. The HRCs for object hit rates and byte hit rates are shown in Figures 4.1 and 4.2 respectively. Note that the y-axis is presented in log scale for clarity.

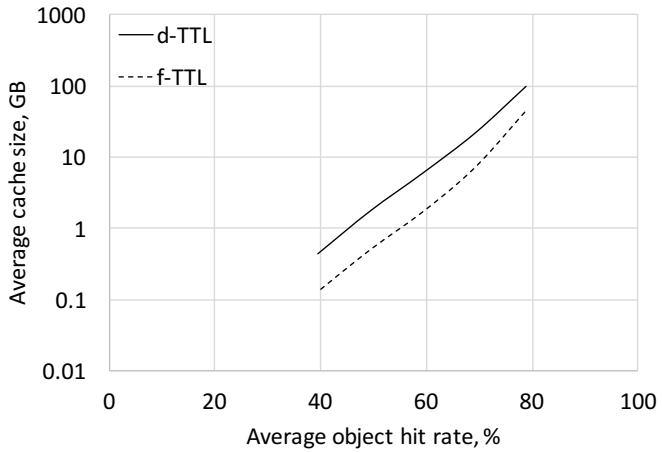


Figure 4.1: Hit rate curve for object hit rates.

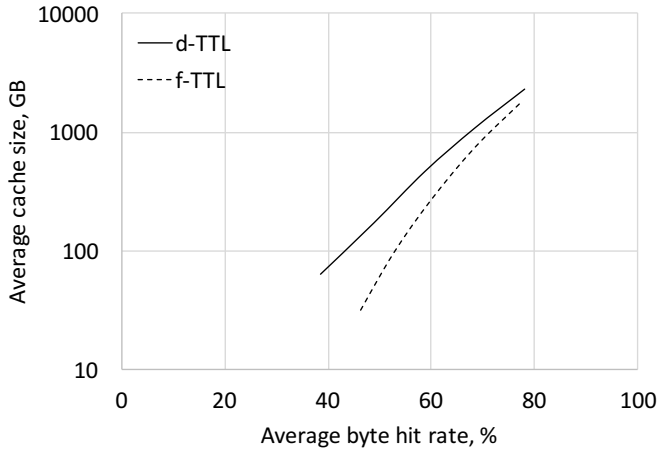


Figure 4.2: Hit rate curve for byte hit rates.

From Figure 4.1 we see that f-TTL always performs better than d-TTL. We find that on average, f-TTL requires a cache that is 49% smaller than d-TTL to achieve the same object hit rate. From 4.2, we see that, for a given byte hit rate, f-TTL requires lesser cache space than d-TTL. On average, f-TTL requires a cache that is 39% smaller than d-TTL to achieve the same byte hit rate. Further note that achieving a specific byte hit rate value requires more cache size than achieving the same value for object hit rate. For instance, the d-TTL algorithm requires a cache size of 469 GB to achieve a 60% byte hit rate, whereas a 60% object hit rate is achievable with a smaller cache size of 6 GB (Figure 4.1). This discrepancy is due to the fact that popular objects in production traces tend to be small (10's of KB) when compared to unpopular objects that tend to be larger (100's to 1000's of MB).

### 4.3.2 Convergence of d-TTL and f-TTL

For the dynamic TTL algorithms to be useful in practice, they need to converge to the target hit rate with low error. From Figures 4.3 and 4.4, we see that the 2 hour averaged object hit rates achieved by both d-TTL and f-TTL have a cumulative error of less than 1.3% while achieving the target object hit rate, on average. We also see that both d-TTL and f-TTL converge to the target hit rate quickly, which illustrates that both d-TTL and f-TTL are able to adapt well to the dynamics of the non-stationary traffic.

From Figures 4.5 and 4.6, we see that d-TTL has a cumulative error of less than 2.3% on average while achieving the target byte hit rate and f-TTL has a cumulative error of less than 0.3%. Moreover, we see that both d-TTL and f-TTL tend to converge to the target hit rate, which illustrates that both d-TTL and f-TTL are able to adapt well to the dynamics of the input traffic. We also observe that the average byte hit rates for both d-TTL and f-TTL have higher variability compared to object hit rates (Figures 4.3 and 4.4), due to the the fact that unpopular content in our traces have

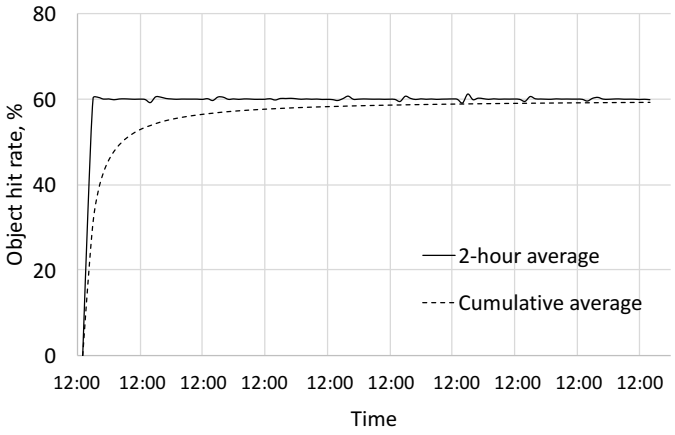


Figure 4.3: Object hit rate convergence over time for d-TTL; target hit rate=60%.

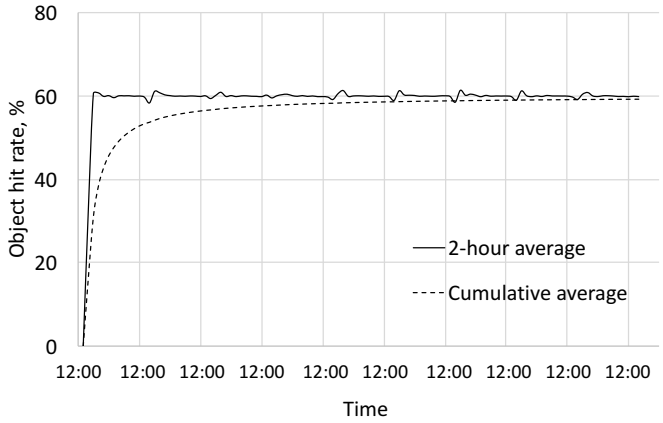


Figure 4.4: Object hit rate convergence over time for f-TTL; target hit rate=60%.

larger sizes, and the occurrence of non-stationary traffic can cause high variability in the dynamics of the algorithm.

In general, we also see that d-TTL has lower variability for both object hit rate and byte hit rate compared to f-TTL due to the fact that d-TTL does not have any bound on the normalized cache size while achieving the target hit rate, while f-TTL is constantly filtering out non-stationary objects to meet the target normalized cache size while also achieving the target hit rate.

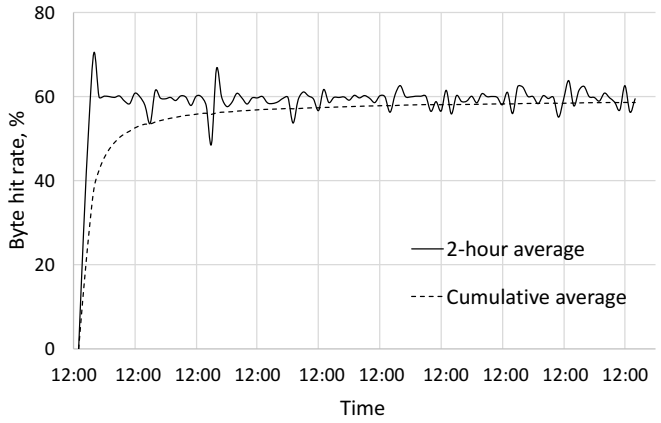


Figure 4.5: Byte hit rate convergence over time for d-TTL; target hit rate=60%.

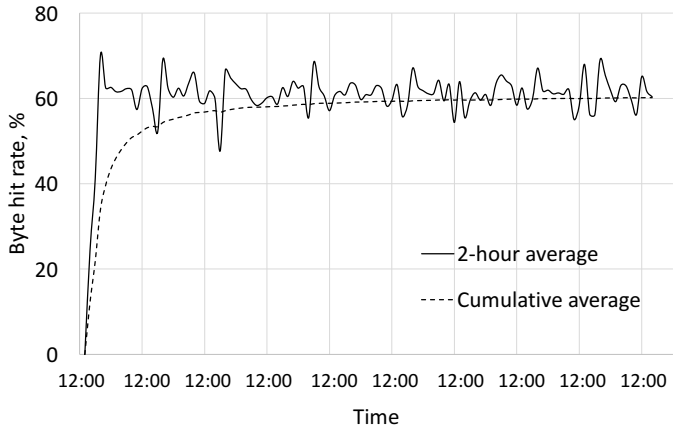


Figure 4.6: Byte hit rate convergence over time for f-TTL; target hit rate=60%.

### 4.3.3 Accuracy of d-TTL and f-TTL

A key goal of the dynamic TTL algorithms (d-TTL and f-TTL) is to achieve a target hit rate, even in the presence of bursty and non-stationary requests. We evaluate the performance of both these algorithms by fixing the target hit rate and comparing the hit rates achieved by d-TTL and f-TTL with caching algorithms such as Fixed TTL (TTL-based caching algorithm that uses a constant TTL value) and LRU (constant cache size), provisioned using Che's approximation [24]. We only present the results for object hit rates (OHR) in Table 4.1. Similar behavior is observed for byte hit rates.

| Target OHR (%) | Fixed TTL (Che's) | | | LRU (Che's) | | d-TTL | | f-TTL | |
|---|---|---|---|---|---|---|---|---|---|
| | TTL (s) | OHR (%) | Size (GB) | OHR (%) | Size (GB) | OHR (%) | Size (GB) | OHR (%) | Size (GB) |
| 80 | 2784 | 83.29 | 217.11 | 84.65 | 316.81 | 78.72 | 97.67 | 78.55 | 55.08 |
| 70 | 554 | 75.81 | 51.88 | 78.37 | 77.78 | 69.21 | 21.89 | 69.14 | 11.07 |
| 60 | 161 | 68.23 | 16.79 | 71.64 | 25.79 | 59.36 | 6.00 | 59.36 | 2.96 |
| 50 | 51 | 60.23 | 5.82 | 64.18 | 9.2 | 49.46 | 1.76 | 49.47 | 0.86 |
| 40 | 12 | 50.28 | 1.68 | 54.29 | 2.68 | 39.56 | 0.44 | 39.66 | 0.20 |

Table 4.1: Comparison of target hit rate and average cache size achieved by d-TTL and f-TTL with Fixed-TTL and LRU.

For this evaluation, we fix the target hit rates (column 1) and analytically compute the TTL (characteristic time) and cache size using Che's approximation (columns 2 and 6) on the request traces assuming Poisson traffic. We then measure the hit rate and cache size of Fixed TTL (columns 3 and 4) using the TTL computed in column 2, and the hit rate of LRU (column 5) using the cache size computed in column 6. Finally, we compute the hit rate and cache size achieved by d-TTL and f-TTL (columns 7-10) to achieve the target hit rates in column 1 and a target normalized cache size that is 50% of that of d-TTL. We make the following conclusions from Table 4.1.

1) The d-TTL and f-TTL algorithms meet the target hit rates with a small error of 1.2% on average. This is in contrast to the Fixed TTL algorithm which has a high error of 14.4% on average and LRU which has even higher error of 20.2% on average. This shows that existing algorithms such as Fixed TTL and LRU are unable to meet the target hit rates while using heuristics such as Che's approximation, which cannot account for non-stationary content.

2) The cache size required by d-TTL and f-TTL is 23.5% and 12% respectively, of the cache size estimated by Che's approximation and 35.8% and 18.3% respectively, of the cache size achieved by the Fixed TTL algorithm, on average. This indicates

that both LRU and the Fixed TTL algorithm, provisioned using Che's approximation, grossly overestimate the cache size requirements.

### 4.3.4 Sensitivity and robustness of d-TTL and f-TTL

We use constant step sizes while adapting the values of $\theta$ and $\theta^s$ in practical settings for reasons discussed in Section 4.2. In this section, we evaluate the robustness and sensitivity of d-TTL and f-TTL to the chosen step sizes. The robustness captures the change in performance due to large changes in step size, whereas the sensitivity captures the change due to small perturbations around a specific step size. For ease of explanation, we only focus on two target object hit rates, 60% and 80% corresponding to medium and high hit rates. The observations are similar for other target hit rates and for byte hit rates.

| Target OHR (%) | Average OHR (%) | | | Average cache size (GB) | | | 5% outage fraction | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\eta$ = 1e-1 | $\eta$ = 1e-2 | $\eta$ = 1e-3 | $\eta$ = 1e-1 | $\eta$ = 1e-2 | $\eta$ = 1e-3 | $\eta$ = 1e-1 | $\eta$ = 1e-2 | $\eta$ = 1e-3 |
| 60 | 59.35 | 59.36 | 59.17 | 9.03 | 6.00 | 5.41 | 0.01 | 0.01 | 0.05 |
| 80 | 79.13 | 78.72 | 77.69 | 150.56 | 97.67 | 75.27 | 0.07 | 0.11 | 0.23 |

Table 4.2: Impact of exponential changes in constant step size $\eta$ on the performance of d-TTL (robustness analysis of d-TTL).

| Target OHR (%) | Average OHR (%) | | | Average cache size (GB) | | | 5% outage fraction | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\eta\times$ 1.05 | $\eta$ | $\eta\times$ 0.95 | $\eta\times$ 1.05 | $\eta$ | $\eta\times$ 0.95 | $\eta\times$ 1.05 | $\eta$ | $\eta\times$ 0.95 |
| 60 | 59.36 | 59.36 | 59.36 | 5.98 | 6.00 | 6.02 | 0.01 | 0.01 | 0.01 |
| 80 | 78.73 | 78.72 | 78.71 | 98.21 | 97.67 | 97.1 | 0.11 | 0.11 | 0.11 |

Table 4.3: Impact of linear changes in constant step size $\eta = 0.01$ on the performance of d-TTL (sensitivity analysis of d-TTL).

Table 4.2 illustrates the robustness of d-TTL to exponential changes in the step size $\eta$. For each target hit rate, we measure the average hit rate achieved by d-TTL, the average cache size and the 5% outage fraction, for each value of step size. The

| Target OHR (%) | Average OHR (%) | | | Average cache size (GB) | | | 5% outage fraction | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\eta^s$ = 1e-8 | $\eta^s$ = 1e-9 | $\eta^s$ = 1e-10 | $\eta^s$ = 1e-8 | $\eta^s$ = 1e-9 | $\eta^s$ = 1e-10 | $\eta^s$ = 1e-8 | $\eta^s$ = 1e-9 | $\eta^s$ = 1e-10 |
| 60 | 59.36 | 59.36 | 59.36 | 5.46 | 2.96 | 1.88 | 0.01 | 0.01 | 0.02 |
| 80 | 78.65 | 78.55 | 78.47 | 89.52 | 55.08 | 43.34 | 0.12 | 0.14 | 0.17 |

Table 4.4: Impact of exponential changes in constant step size $\eta^s$ on the performance of f-TTL (robustness analysis of f-TTL).

| Target OHR (%) | Average OHR (%) | | | Average cache size (GB) | | | 5% outage fraction | | |
|---|---|---|---|---|---|---|---|---|---|
| | $\eta^s\times$ 1.05 | $\eta^s$ | $\eta^s\times$ 0.95 | $\eta^s\times$ 1.05 | $\eta^s$ | $\eta^s\times$ 0.95 | $\eta^s\times$ 1.05 | $\eta^s$ | $\eta^s\times$ 0.95 |
| 60 | 59.36 | 59.36 | 59.36 | 3.01 | 2.96 | 2.91 | 0.01 | 0.01 | 0.01 |
| 80 | 78.55 | 78.55 | 78.54 | 55.65 | 55.08 | 54.27 | 0.14 | 0.14 | 0.14 |

Table 4.5: Impact of linear changes in constant step size $\eta^s$ = 1e-9 on the performance of f-TTL (sensitivity analysis of f-TTL).

5% outage fraction is defined as the fraction of time the hit rate achieved by d-TTL differs from the target hit rate by more than 5%.

From this table, we see that a step size of 0.01 offers the best trade-off among the three parameters, namely average hit rate, average cache size and 5% outage fraction. Table 4.3 illustrates the sensitivity of d-TTL to small changes in the step size. We evaluate d-TTL at step sizes $\eta = 0.01 \times (1 \pm 0.05)$. We see that d-TTL is insensitive to small changes in step size.

To evaluate the robustness and sensitivity of f-TTL, we fix the step size $\eta = 0.01$ to update $\theta$ and evaluate the performance of f-TTL at different step sizes, $\eta^s$, to update $\theta^s$. The results for robustness and sensitivity are shown in Tables 4.4 and 4.5 respectively. For f-TTL, we see that a step size of $\eta^s$=1e-9 offers the best tradeoff among the different parameters namely average hit rate, average cache size and 5% outage fraction. Like, d-TTL, f-TTL is insensitive to small changes in step size parameter $\eta^s$.

In Table 4.2 and Table 4.4, a large step size makes the d-TTL and f-TTL algorithms more adaptive to the changes in traffic statistics. This results in reduced error in the average OHR and reduced 5% outage fraction. However, during periods of high burstiness, a large step size can lead to a rapid increase in the cache size required to maintain the target hit rate. The opposite happens for small step sizes.

### 4.3.5 Effect of target normalized size on f-TTL

In the previous evaluations, f-TTL is implemented by setting a normalized cache size target that is 50% of the normalized cache size of d-TTL. This helps f-TTL achieve the same target hit rate as d-TTL but at half the expected cache size. In this section, we evaluate the performance of f-TTL when we change the normalized cache size target. Specifically, we measure the average hit rate and cache size achieved by f-TTL when we set the target object hit rates to 60% and 80% and the target normalized cache size to 45%, 50% (as in Section 4.3) and 55% of the normalized cache size of d-TTL.

| Target OHR (%) | Normalized cache size target (%) | OHR achieved (%) | Cache size achieved (GB) |
|---|---|---|---|
| 60 | 45 | 59.36 | 2.94 |
| | 50 | 59.36 | 2.96 |
| | 55 | 59.36 | 2.98 |
| 80 | 45 | 78.55 | 54.81 |
| | 50 | 78.55 | 55.08 |
| | 55 | 78.55 | 55.32 |

Table 4.6: Impact of normalized cache size target on the performance of f-TTL.

From Table 4.6, we see that with a target hit rate of 60%, f-TTL is able to achieve the target hit rate with a small error of 0.64% in all three target normalized cache size scenarios. Similarly, f-TTL is also able to achieve the target hit rate of 80% in all three target normalized cache size scenarios with a slightly larger error of 1.45%.

Both these scenarios show that f-TTL is able to the target hit rate at different target normalized cache sizes with high accuracy.

We also measure the average cache size achieved by f-TTL in all three scenarios. In the case of the 60% target hit rate, we see that f-TTL achieves a hit rate of 59.36% with the smallest average cache size when the target normalized cache size of f-TTL is 45% of that of d-TTL and the largest average cache size when the target normalized cache size of f-TTL is 55% of that of d-TTL. This shows that f-TTL more aggressively filters out non-stationary content to achieve the target hit rate at smaller normalized cache size targets. The opposite happens at higher target normalized cache sizes. Similar behavior is observed when setting the target hit rate to 80%.

As discussed in Section 4.2, when a target normalized cache size is unachievable, the target can instead be used to control the aggressiveness of f-TTL in filtering out non-stationary content.

## 4.4 Related Work

Caching algorithms have been studied for decades in different contexts such as CPU caches, memory caches, CDN caches and so on. We briefly review some related prior work.

*TTL-based caching:* TTL caches have found their place in theory as a tool for analyzing capacity based caches [36, 47, 52], starting from Fagin's work [42] followed by Che's Approximation of LRU caches [24]. Recently, its generalizations [13,93] have commented on its wider applicability. However, the generalizations hint towards the need for more tailored approximations and show that the vanilla Che's approximation can be inaccurate [93]. On the applications side, recent works have demonstrated the use of TTL caches in utility maximization [36] and hit ratio maximization [11]. Specifically, in [36] the authors provide an online TTL adaptation algorithm highlighting the need for adaptive algorithms. *However, unlike prior work, we propose*

*the first adaptive TTL-based caching algorithms that provably achieve cache hit rate and expected cache size targets in the presence of non-stationary traffic such as one-hit wonders and traffic bursts.*

*Cache tuning and adaptation:* Most existing adaptive caching algorithms require careful parameter tuning to work in practice. There have been two main cache tuning methods: (1) global search over parameters based on prediction models, e.g. [29,103], and (2) simulation and parameter optimization based on shadow caches, e.g. [30]. The first method often fails in the presence of cache admission policies; whereas, the second method typically assumes stationary arrival processes to work well. However, with real traffic, static parameters are not desirable [87] and an adaptive/self-tuning cache is necessary. The self-tuning heuristics include, e.g., ARC [87], CAR [6], PB-LRU [126], which try to adapt cache partitions based on system dynamics. *While these tuning methods are meant to deal with non-stationary traffic, they lack theoretical guarantees unlike our work, where we provably achieve a target hit rate and a feasible cache size by dynamically changing the TTLs of cached content.*

## 4.5 Conclusion

In this work we propose TTL-based caching algorithms that can automatically learn and adapt to the request traffic and provably achieve any feasible hit rate and cache size. We present a theoretical justification for the use of two-level caches in CDN settings where large amounts of non-stationary traffic can be filtered out to conserve cache space while also achieving target hit rates. On the practical side, we evaluate our TTL caching algorithms using traffic traces from a production Akamai CDN server. The evaluation results show that our adaptive TTL algorithms can achieve the target hit rate with high accuracy; further, the two-level TTL algorithm can achieve the same target hit rate at a much smaller cache size.

# CHAPTER 5

# ENERGY-EFFICIENT CACHING USING DISK SHUTDOWN

CDNs deploy hundreds of thousands of servers around the world to cache and serve web pages, videos, and other content to billions of users around the world at low latency. While providing better performance in the form of fast downloads is the primary goal of a CDN, energy minimization has become critical in the past few years for two reasons. Deployed servers in data centers now account for more than 1.5% of the global power consumption [77], consuming more than mid-sized countries such as Argentina, and growing at a rapid pace commensurate with the growth of the Internet. With greater awareness of climate change, the CDN industry is increasingly focused on making their systems more sustainable. For CDNs, reducing the energy usage of server deployments is a major part of their sustainability goals.

A second motivator for the reduction of the energy usage of CDN servers is the rising cost of energy. The operational expenditure (OPEX) of a CDN can be divided into three broad categories: bandwidth cost, colocation cost and energy cost. Colocation cost is the cost of the datacenter space and racks to host the servers. While bandwidth costs have been the dominant factor in a CDN's OPEX in the past, bandwidth prices have fallen sharply each year in the past decades. The bandwidth cost of delivering 1 MByte has fallen from \$0.15 in 1998 to \$0.00005 per MB today, a drop of $1.8x$ per year. In stark contrast, the cost of energy has been rising over the past decade [8]. Due to these long-term price dynamics, energy cost is now comparable to the bandwidth cost. The cost structures at most data centers are such that the

energy cost is nearly 30% of the OPEX and is expected to only rise further in this decade.

– *Disk shutdown in CDNs:* In each server, there are several components that consume energy. Energy consumed by the CPU, disks, fans, memory chips and motherboard chipset account for most of the consumed energy. CDN server models vary widely from each other. The number of disk drives per server is a model-specific parameter and can vary from 2 to as high as 64. Across most production server models that CDNs use, the average fraction of energy consumption attributed to the spinning disks is estimated to range from 40% to 55%. Therefore, disk energy represents a sizable chunk of the CDN's energy consumption, making it the main focus of our study.

The primary mechanism to save energy that we explore in our work is disk shutdown. For hard disks, the two natural options for energy reduction are shutting down disks entirely or reducing their rotational speed, the former providing more drastic energy reduction than the latter. In most IT systems, disks store *original* data that could be accessed at any time and shutting down disks means that the data is completely unavailable, an unacceptable outcome that must be avoided at all costs. In fact, most server software would need to be re-designed significantly to handle the unavailability of data from disk shutdowns. Therefore, disk shutdown has seldom been explored or implemented in industry. However, disk shutdown is a viable energy saving mechanism for a CDN because the disk cache of a CDN server only stores a *copy* of the content that is stored persistently at the content provider's origin servers. Thus, the unavailability of a cached copy is easily rectified by retrieving it from a peer server or origin. While this causes performance degradation for the user, it is less severe than content unavailability. Thus, if disk shutdown provides significant energy saving in exchange for a small performance degradation, that is an interesting possibility from the standpoint of a CDN operator. Our work is focused on understanding

this energy-performance tradeoff, allowing a CDN operator to choose an acceptable operating regime in that tradeoff.

A complementary approach studied in the literature is turning off servers entirely [23, 78, 80, 84, 111]. However, turning off servers has the disadvantage of complicating network management in a global CDN. If servers are unreachable for extended periods of time, they miss real-time reporting, software updates and control messages for that duration. This may upend network management guarantees and operational practices of the CDN platform. Thus, shutting down disks as proposed here, while the servers are live and serving content, represents an attractive complementary option worth exploring.

– *Challenges:* An important determinant of user-perceived performance is the cache hit rate[1]. When disks are shut down, the content stored on them become unavailable, leading to a decrease in the cache hit rate. To scope out the impact of disk shutdown on performance, we first implemented a simple baseline scheme that shuts down disks in proportion to the disk load, e.g., when the disks are loaded at x% of its I/O capacity, we turn off roughly (100 - x)% of the disks chosen at random. The baseline scheme was tested on a simulated CDN server with traffic logs from a live server in the Akamai CDN. Figure 5.1 shows the hit rate for a server with the baseline disk shutdown algorithm (labeled `Load/Random/LRU`) in comparison with hit rate for the same server when all disks are active (labeled `NOOFF`). The observed 15-20% drop in cache hit rate with disk shutdown would be unpalatable to the CDN operator for performance reasons, even if the energy savings were significant. *Our main challenge is to propose algorithms that are smarter than the baseline scheme, so that the performance penalty incurred to obtain energy savings is not steep.*

– *Contributions:* We make the following contributions in this work.

[1]In this work, we consider the byte hit rate as we want to reduce the amount of cache miss traffic sent to the origin server.

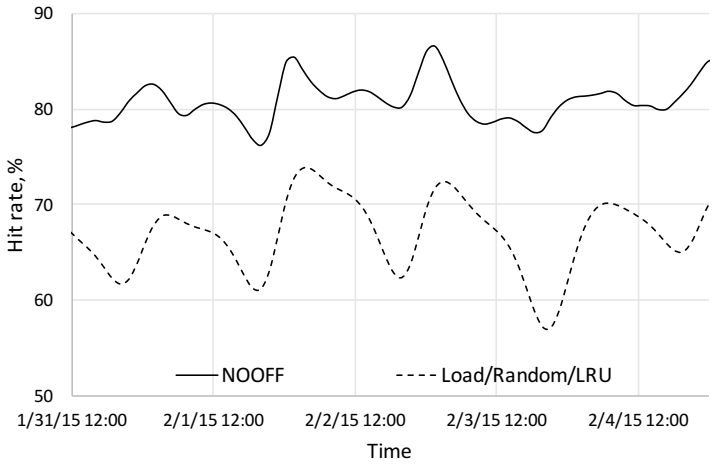


Figure 5.1: Hit rate decreases significantly when disks are shut down using a simple baseline scheme.

**1)** We show that unlike most enterprise class servers, CDN servers are able to save energy through disk shutdown without major software redesign or major performance degradation. We develop energy-efficient cache management schemes to address three key questions.

- *Cache sizing.* How large a disk cache does a server need to hold the "working set" of the content that is being accessed by users?
- *Disk shutdown.* Which disks must be shut down (or woken up) to realize the cache size that is required?
- *Content placement & eviction.* Where should content be placed and what is to be evicted if the cache is full?

**2)** We explore simple and implementable algorithms for cache sizing, disk shutdown and content placement & eviction, to understand their impact on both energy savings and cache hit rates. Using extensive traces from Akamai's CDN servers, we derive the energy-performance tradeoff for our algorithms. We show that our algorithms achieve a 30% energy reduction in a single server with only 6.5% reduction in the normalized server hit rate.

**3)** The proposed algorithms provide a much better energy-performance in a cluster as multiple servers might have copies of the same content for redundancy. Using production traces, we show that our algorithms achieve a 30% energy reduction with only a 3% reduction in the normalized cluster hit rate.

– *Note:* A key reason behind the effectiveness of our algorithms lies in the very nature of how content on the Internet is accessed by users. As shown in Figure 5.2, of the 25 million objects accessed on an Akamai CDN server over a period of 9 days, over 16 million were "one-hit-wonders" accessed only once! In fact, only 6.6% of the objects were accessed more than 10 times over the 9-day span. Further, as shown in Figure 5.4, 80% of the requests are for 1% of the objects. Our algorithms migrate the more popular content to a subset of disks within the server, allowing the other disks to be shut down. As long as our algorithms place at least one copy of the small fraction of popular objects on an active disk, the loss of the remaining "long tail" of less popular content due to disk shutdown has only a modest impact on hit rates.

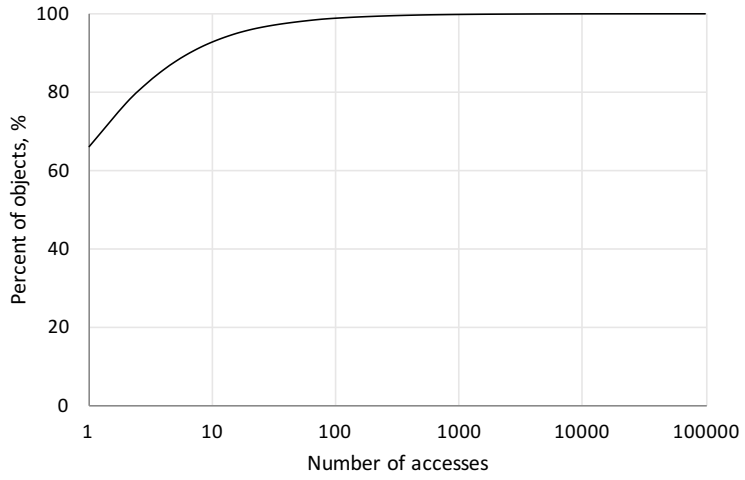


Figure 5.2: Popularity of content accessed by users on a CDN server.

In summary, our work is the first to establish disk shutdown as a key mechanism for energy savings in CDNs, paving the way for its future roll-out on the production network.

– *Roadmap:* The rest of this chapter is organized as follows. In Section 5.1, we describe the typical cache management function of the edge server, and how we modify it to be energy efficient. We also describe our evaluation methodology in this section. The three main components of energy efficient cache management are cache sizing, disk shutdown, and content placement & eviction. We describe and evaluate the performance of these components in Sections 5.2, 5.3, and 5.4 respectively. The evaluation is performed in the context of the hit rate of a single server. In Section 5.5, we evaluate the different cache management schemes in the context of a cluster of servers and show how the hit rates for an entire cluster differ from that of individual servers. Finally, we review related work in Section 5.6 before concluding in Section 5.7.

## 5.1 Cache management schemes

We describe the cache management schemes that we propose, implement, and study in our work.

### 5.1.1 A typical algorithm without disk shutdown

We describe a typical cache management scheme that is often used by CDNs that we call `NOOFF`. `NOOFF` does not shut down disks and hence performs only content placement and eviction. Each server has multiple disks. Each requested object that is not already in cache is placed on a randomly-selected disk so that load and space utilization are uniform across all disks. The entire cache space is part of one single Least-Recently-Used (LRU) stack. Eviction occurs when the cache is more than 95% full, at which time ∼5% of the least recently used bytes are evicted from cache.

We implement `NOOFF` to assess the hit rates that current CDN servers achieve when no energy savings are in place and no disks are shut down. The hit rates of the algorithms we propose for disk energy reduction must be viewed in relation to

<table>
<tr><th>Cache Sizing</th><th>Disk Shutdown</th><th>Content Placement & Eviction</th></tr>
<tr><td rowspan="3"><code>Hybrid</code> (Load* & storage based)</td><td><code>Random*</code></td><td><code>LRU*</code> (Random placement & LRU)</td></tr>
<tr><td><code>Fixed</code></td><td><code>SLRU</code> (Segmented placement & LRU)</td></tr>
<tr><td colspan="2"><code>LRU-DS</code> LRU-ordered disk shutdown</td></tr>
</table>

Table 5.1: Algorithms for energy-efficient cache management. The starred algorithms are simple options that we use as a baseline that we improve upon.

the hit rate of `NOOFF`. The decrease in the hit rate of an energy-efficient algorithm in comparison to `NOOFF` is the performance penalty that is paid in exchange for the energy savings. In particular, for our energy-efficient algorithms, we compute the *normalized* hit rate which is simply the ratio of the algorithm's hit rate and that of `NOOFF`.

### 5.1.2 Energy-efficient cache management

Besides content placement and eviction, our energy-efficient cache management schemes incorporate two other components (see Table 5.1). A *cache sizing algorithm* determines the number of active disks required for storing and serving the current working set of content that is being accessed by users. We describe our cache sizing algorithm, `Hybrid`, in Section 5.2. Once cache sizing sets a target number of active disks, a *disk shutdown algorithm* chooses the precise set of disks that must be shut down or woken up. We study two algorithms, `Random` and `Fixed`, that we describe in Section 5.3. Finally, we study content placement & eviction algorithms and compare the baseline scheme of `LRU` with a sophisticated scheme of segmented placement and LRU (`SLRU`) as shown in Table 5.1.

In the rest of the chapter, we represent our energy-efficient cache management solutions as a triple, specifying what algorithm was used for cache sizing, disk shutdown,

and content placement & eviction. We start out with the simple baseline solution of `Load/Random/LRU` whose poor hit rate performance we outlined earlier Figure 5.1. We progressively improve each of the three algorithms to show that `Hybrid/Fixed/SLRU` outperforms other combinations and provides the best energy-performance tradeoffs.

### 5.1.3 An ideal energy-efficient variant of LRU

LRU is known to be an effective technique for content eviction, variants of which are implemented in most real-world CDNs. We propose a simple extension of LRU to incorporate disk shutdown which we call LRU-ordered disk shutdown (`LRU-DS`). `LRU-DS` keeps content on disk as per the LRU ordering, i.e., the content on disk $i$ is less recently used than the content on disk $i+1$ for all $1 \leq i < n$. When the cache sizing algorithm requires $k$ disks to be shut down, `LRU-DS` marks the $k$ lowered numbered disks as being inactive, i.e., the ones with content that was least recently used are marked inactive. Likewise, when $k$ disks need to be woken up, the higher numbered disks are marked active, i.e., the disks that have the more recently accessed content are marked active. If a requested object is present on a disk that is marked active, that request is considered a cache hit. If a requested object is not present or if it is present only on an inactive disk, that request is deemed a cache miss.

Note that `LRU-DS` is *not* an implementable algorithm as it requires the content on *both* active and inactive disks to be *always* ordered in an LRU fashion. It is not possible to maintain that property since inactive disks cannot be read or written into. However, `LRU-DS` in combination with a cache sizing algorithm such as `Hybrid` provides an idealized upper bound on hit rates that our algorithms can attempt to reach. For this reason, we plot the energy-performance tradeoff of `Hybrid/LRU-DS` as a point of comparison to the tradeoff achieved by our algorithms, though the former may not be achievable by any implementable algorithm.

### 5.1.4 Evaluation methodology

#### 5.1.4.1 Content request traces

The extensive data set used in this work was collected from one of the server clusters in Akamai's CDN. The data set contains anonymized access logs for content requested by users. Each log line corresponds to a single request and contains a timestamp, the requested URL (anonymized), object size, and bytes served for the request. The access logs were collected over a period of 9 days from a cluster containing 5 CDN servers. Each server has a disk configuration that is typical for deployed CDN servers: 8 spinning disks with a capacity of 600GB each with a content cache. We chose a busier cluster for our analysis to provide a conservative bound on the energy savings. *Disk load* [2] is a function of the number and the amount of read and write I/O operations that are being performed on the disk and is expressed as a fraction of its I/O capacity[3]. The average disk load of the servers in our cluster was 39.4% of capacity, this average includes both peak and off-peak periods. The average disk load of a typical CDN server cluster tends to be lower than 20%, and is likely to provide an even greater opportunity for energy savings from disk shutdown. The traffic served in Gbps captured in our data set is shown in Figure 5.3. We can see that the cluster has a short off-peak duration of about 6 hours each day.

| | |
|---|---|
| Total requests | 3 billion |
| Total bytes served | 429 TB |
| Total distinct objects | 162 million |
| Total distinct bytes served | 67 TB |

Table 5.2: Characteristics of content request traces from the Akamai cluster.

[2]See linux utility iostat for a description of how disk load is computed [2].

[3]We conservatively use 80% of the hardware-rated I/O capacity as our available I/O capacity in our experiments.

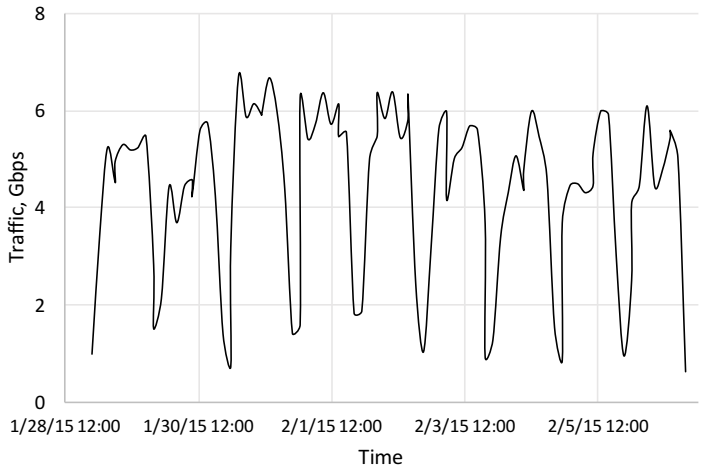


Figure 5.3: Content traffic served to users from our cluster (in Gbps), averaged hourly.

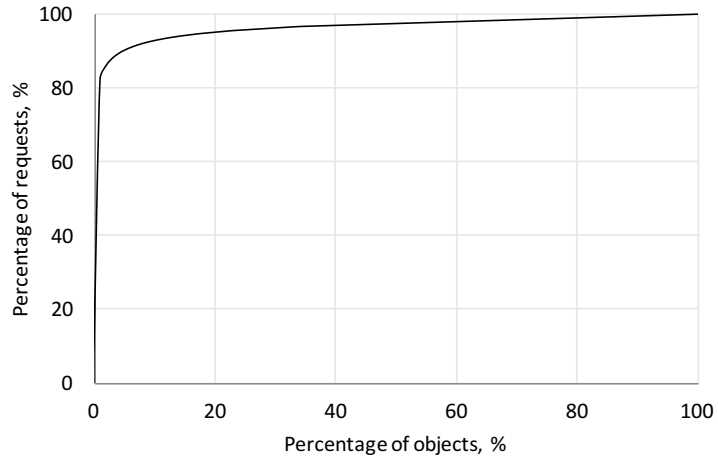


Figure 5.4: A large fraction of the requests are for a small fraction of the objects.

Table 5.2 lists some of the characteristics of the content request traces used in this work. As shown in Figure 5.2, the content popularity distribution exhibits a "long tail" with nearly 70% of the objects accessed only once. Further, as in Figure 5.4, 80% of the requests are for 1% of the objects.

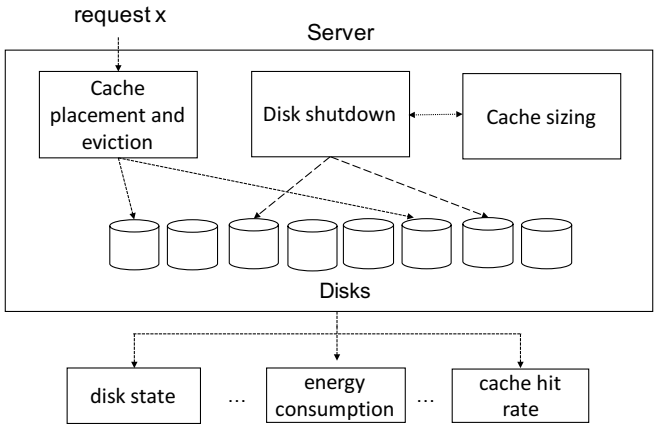


Figure 5.5: The architectural components of a content cache that uses disk shutdown.

#### 5.1.4.2 Cache simulator for disk shutdown

We evaluate our algorithms using a custom event-driven simulator. This simulator simulates all the necessary hardware and functional details of a CDN server as shown in Figure 5.5. The simulator mimics the content placement and eviction algorithms used by the live CDN servers. Every incoming request is placed on one of the disks and existing content is evicted from disks when necessary. In addition to the above fundamental functionality, the simulator implements new architectural components, the cache sizing and disk shutdown components. These components do not exist presently in the live production CDN servers. The cache sizing component estimates the number of active disks needed to store the working set of content that is currently being accessed by users. The disk shutdown component chooses the precise set of disks to be shut down or woken up. For each of these components, we implement multiple algorithms described in Table 5.1. The simulator also carefully tracks the I/O request rates that every disk receives. It plugs these I/O request statistics into the disk power model outlined in Appendix B to compute per-disk, per-server and cluster-wide energy consumption.

From the configuration provided, the simulator constructs the multi-layer cache hierarchy within each server: a number of simulated disks of the given size, the filesys-

tem buffer cache, and the web-server software's hot-object memory cache. In addition to a single-server environment, the simulator is also able to create a cluster of servers. In this mode, it mimics the Internet Cache Protocol (ICP)-based [116] intra-cluster content sharing among all the servers of a cluster, same as the mechanism used in production servers. As its input, it accepts content access logs from live CDN servers in the production network. It periodically outputs a rich set of metrics such as traffic volumes, cache hit rates (broken down into the hit-rate seen at every cache hierarchy layer within the server and cluster hit rates when ICP is used). The simulator was validated by replaying 9 days of a production cluster's logs and matching the simulator output metrics with the production cluster's traffic and hit rate statistics.

## 5.2 Cache sizing algorithms

A cache sizing algorithm determines how many active disks are required for storing and serving the current working set of content that is being accessed by users. The number of disks that need to be active depends on the current incoming requests for content, i.e., potentially more active disks are required when large volumes of content are being accessed than during periods when access volumes are smaller.

A cache sizing algorithm must consider two different types of resource constraints. First, each disk has an I/O capacity that determines the number of input/output operations (IOPS) that it can sustain. We define *disk load demand* $L$ as the amount of disk IOPS required to serve the incoming content requests, expressed in the units of the maximum IOPS that a single disk can support. For instance, if $L = 6.5$, then the number of IOPS that need to be supported is 6.5 times that of the IOPS of a single disk. Thus, we need to have at least $\lceil L \rceil$ active disks to serve the content requests, since otherwise the load of some active disk will exceed 100%, resulting in very slow response times for retrieving content from disk. A second constraint is the disk storage capacity that dictates how much content you can store in the disk.

Shutting down too many disks results in a decrease in the active disk storage capacity, resulting in an increase in cache misses, leading to poor performance. The goal of the cache sizing algorithm is to determine the number of active disks, taking into consideration both disk I/O and storage constraints.

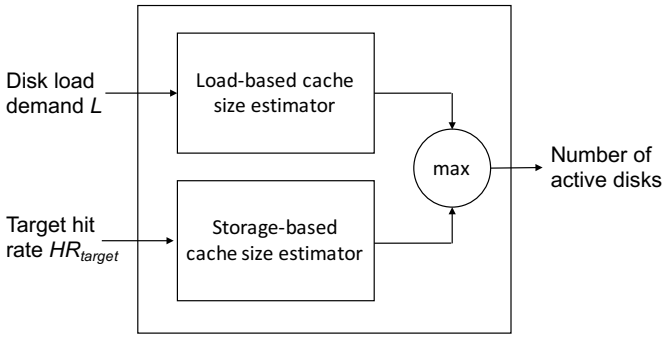


Figure 5.6: The `Hybrid` cache sizing algorithm.

As shown in Figure 5.6, our cache-sizing algorithm takes as input the disk load demand $L$ that needs to be supported by the active disks and a target hit rate $HR_{target}$ that must be sustained and computes the number of disks that must stay active. We call our algorithm "`Hybrid`" as it uses two different estimators, a load-based cache size estimator and a storage-based cache size estimator, and takes the maximum of these two estimates. We describe each estimator below.

*Load-based cache size estimator:* Load-based sizing determines the number of active disks using the disk load demand $L$ that must be supported to serve the incoming content requests. Specifically, it estimates that $\lceil L \rceil$ disks need to be active. In our trace-based simulations, at time $t$, we record the average disk load $x_t\%$ of a CDN server with $n$ (active) disks and estimate the disk load demand $L$ to equal $\frac{x_t}{100} \cdot n$. To provide more stability to the algorithm, the average disk load $x_t$ is computed as the average of the instantaneous disk load values for all the disks during the time interval $[t - \tau_l, t]$. Note that the choice of $\tau_l$ presents a recency-stability tradeoff, smaller values of $\tau_l$ could lead to more recent but spiky estimates, and larger $\tau_l$ could lead to less recent but more stable estimates.

*Storage-based cache size estimator:* Storage-based cache sizing uses the recent content access sequence to predict how much cache storage is required to achieve a target cache hit rate $HR_{target}$. It performs the following two steps.

*i) Compute the hit rate curve (HRC).* The HRC relates cache size with hit rate. Figure 5.7 shows an example HRC computed for a segment of our CDN content access trace. Given a request sequence $R = \langle r_1, r_2, \cdots, r_n \rangle$, we first compute the stack distance $s_i$ [85] for each request $r_i$, $1 \leq i \leq n$. If the object requested by $r_i$ was never requested before, its stack distance $s_i$ is infinite. Otherwise, let $j$ be the largest integer such that $j < i$ and $r_j$ is a request for the same object as $r_i$, i.e., $r_j$ is the previous request for the same object as $r_i$. The stack distance of $r_i$ is simply the number of *unique* bytes accessed in the request sequence $\langle r_j, \cdots r_i \rangle$. The hit rate for request sequence $R$ and cache size $C$ is computed by assuming that every request $r \in R$ that has stack distance less than or equal to $C$ is a cache hit and every request $r' \in R$ that has stack distance greater than $C$ is a cache miss. This computation is repeated for different values of $C$ to obtain the HRC. A keen reader will note that HRC is the hit rate achieved on request sequence $R$ by an idealized cache of size $C$ that is maintained in LRU order.

*ii) Given a target hit rate $HR_{target}$, estimate the number of active disks required to achieve that target using the HRC.* The timeline is divided into segments of length of $\tau_{hrc}$ hours. At the end of each time segment, a new HRC is computed by setting the request sequence $R$ to be all the requests received in that segment. The choice of $\tau_{hrc}$ presents a tradeoff. Larger $\tau_{hrc}$ records more history but varies slowly with time and smaller $\tau_{hrc}$ records lesser history but is more sensitive to time-varying traffic. The variability in input traffic can be used to decide a suitable value for $\tau_{hrc}$.

Once the HRC is computed, the "ideal" cache size estimate $C_{ideal}$ is the value that corresponds to $HR_{target}$ in the HRC (cf. Figure 5.7). One key aspect of our algorithms not modeled by the idealized LRU cache is that *multiple* copies of the

same object may be stored on a server, albeit the copies must appear on different disks within the server. Such object replication occurs when an object stored in a currently inactive disk is accessed by a user, resulting in a new copy being created on an active disk. When the first disk becomes active again, we may end up with multiple copies of an object in the active disks. The degree of object replication depends on the algorithms being used for disk shutdown. To account for this replication, we compute a replication factor $\rho$ which is simply the actual total bytes currently in cache divided by the total unique bytes. The required cache size $C$ is set to be equal to $\rho \times C_{ideal}$. Thus, the number of active disks is $\min\{\lceil C/C_s \rceil, n\}$ disks, where $C_s$ is the storage capacity of a single disk and $n$ is the total number of disks in the server.

*Example:* Suppose that $HR_{target} = 75\%$ and the HRC is as shown in Figure 5.7. $C_{ideal}$ is 3000 GB requiring 5 active disks of 600 GB each. If $\rho = 1.15$, then $C = 3450$ GB, requiring an additional disk to account for the replication. Our storage-based sizing algorithm recommends 6 active disks.

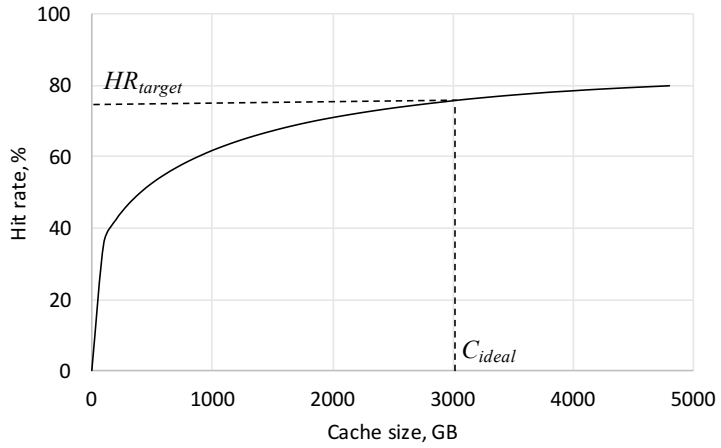


Figure 5.7: Hit rate curve (HRC) shows the relation between cache size and hit rate.

After the load-based and storage-based cache size estimates have been determined, our `Hybrid` algorithm computes the maximum of the two estimates to satisfy both the load and storage constraints.

### 5.2.1 Experimental evaluation

We evaluate the performance of our cache sizing algorithm `Hybrid`, with two other algorithms `Load` that uses only a load-based cache size estimator and `Storage` that uses only a storage-based cache size estimator. We use the baseline algorithm of `Random` for disk shutdown and `LRU` for content placement & eviction (cf., Table 5.1), i.e., we pick the disks to be shut down (or woken up) randomly and we place new objects on a random disk and use LRU eviction when the cache is full. We evaluate all three sizing algorithms and `NOOFF` using our Akamai content access traces for a CDN server that has $n = 8$ disks, each disk having capacity $C_s = 600GB$. We use $\tau_l = 60s$ in the simulations. We set $\tau_{hrc} = 6\ hrs$ to account for the day/night variations in the traffic and we set $HR_{target}$ to equal the current hit rate of `NOOFF`.

In Figure 5.8, we see that `Load/Random/LRU` has an average hit rate that is nearly 15% lower than `NOOFF`, since the load-based scheme underestimates the need for cache space and turns off more disks than it should during off-peak hours when the incoming content request volume is low. The load-based scheme keeps only ∼2 disks (out of the 8) active during off-peak hours, leading to a higher rate of eviction. We also observe that the eviction age for the load-based cache sizing scheme is ∼2.5 times lower than that of `NOOFF`. Thus, a load-based scheme alone does not provide good hit rate performance, since cache sizing also depends on the actual content access patterns that cause a given disk load. For instance, if the disk load was caused by very few popular objects in cache, that implies that a smaller cache size with fewer active disks is sufficient. However, if the same disk load was distributed over a large number of less popular objects, more active disks are required to hold the working set. Thus, we must consider the actual content access patterns to determine the number of active disks.

In Figure 5.8, note that the `Hybrid/Random/LRU` has a higher cache hit rate when compared to `Load/Random/LRU`, since `Hybrid` also accounts for the storage

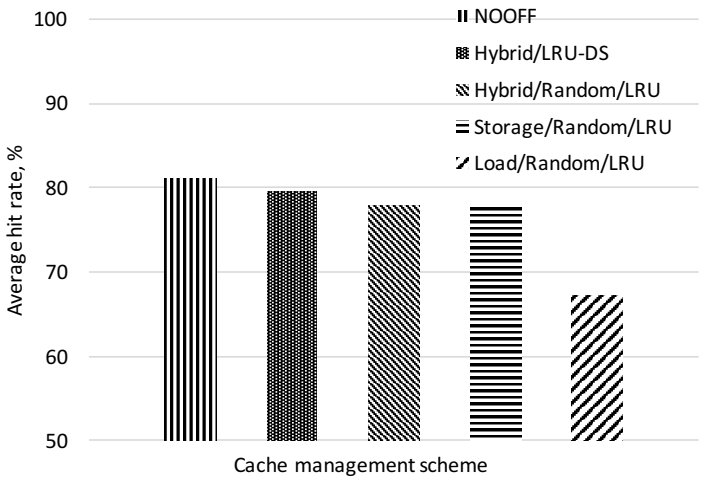


Figure 5.8: Comparing the hit rate performance of the different cache sizing algorithms with the hit rate of `NOOFF` that does not shut down disks.

constraints. But the cache hit rate of the `Hybrid/Random/LRU` is still ∼3% lower than the ideal target hit rate that was set to equal `NOOFF`. This discrepancy is in part due to the inefficiency of `random` disk shutdown that could make frequently accessed objects in the randomly-selected disks inactive. To remove this effect, we compare the performance of `Hybrid/LRU-DS` with `NOOFF`. We see that `Hybrid/LRU-DS` has a hit rate performance that is ∼1.5% less than `NOOFF`. This small difference indicates that `Hybrid` is a good cache sizing algorithm that is able to closely match the target hit rate of `NOOFF`.

What about storage-based *only* cache sizing? The hit rate of storage-based cache sizing, `Storage/Random/LRU`, is nearly the same as using `Hybrid/Random/LRU` as shown in Figure 5.8. However, the former has the drawback of occasionally overloading the disks. The disk load occasionally goes over the 100% mark as highlighted in Figure 5.9. To avoid such overloading, the hybrid cache sizing algorithm proposed above should be used to account for both the load and storage constraints of the server disks.

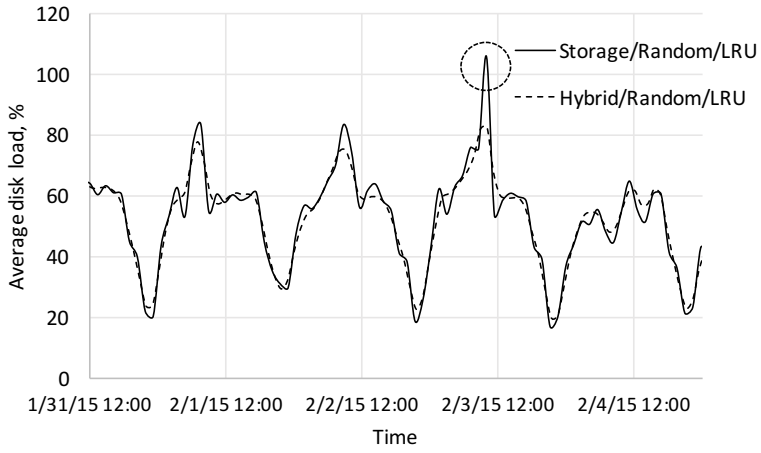


Figure 5.9: The storage-based cache sizing algorithm occasionally overloads the disks, since it does not factor in disk load. This deficiency can be corrected with a hybrid scheme.

*Concluding remark:* The `Hybrid` algorithm works the best for cache sizing and we use this algorithm as the default option in all our future experiments where we investigate disk shutdown and content placement & eviction algorithms.

## 5.3 Disk shutdown algorithms

Once the cache sizing algorithm outputs a target number of disks that need to be active, the disk shutdown algorithm decides precisely *which* disks should be shut down (or woken up) to meet that target. Let $dcount_t$ be the number of active disks at time $t$ and suppose that the cache sizing algorithm produces a target $dtarget_t$ of active disks. Then, if $dtarget_t > dcount_t$, the disk shutdown algorithm wakes up $|D| = dtarget_t - dcount_t$ disks, where the set $D$ is the set of all disks that need to be woken up, and if $dtarget_t < dcount_t$, the disk shutdown algorithm shuts down $|D| = dcount_t - dtarget_t$ disks, where the set $D$ is the set of all disks to be shut down. There are a number of ways in which the set of disks $D$ can be chosen and we review two shutdown algorithms below.

*1) Random disk shutdown.* Algorithm `Random` is a simple baseline scheme where the required number of disks to be shut down (resp. woken up) are randomly selected from among the active (resp. inactive) disks.

*2) Fixed disk shutdown.* The disks are ordered in sequence from 1 through $n$. Algorithm `Fixed` shuts down disks in the increasing order starting from 1, i.e., if $k$ disks are to be made inactive the disks 1 to $k$ are shut down. When the disks are made active, they are woken up in the decreasing order, i.e., disk $k$ is woken up, followed by $k-1$, and so on till disk 1.

The objects may get replicated on two or more disks within the same server in both shutdown schemes. If an object that is currently accessed is on an inactive disk, a new copy is made on an active disk. When both disks are active at a later time, we have more than one copy of the object.

### 5.3.1 Experimental evaluation

We empirically evaluate the algorithms `Random` and `Fixed` by simulating them on the CDN content request traces. In particular, we use `Hybrid` for cache sizing and the baseline content placement & eviction algorithm of `LRU` with our two shutdown algorithms. We also compare the energy-performance tradeoff of these two algorithms with that of the idealized algorithm `LRU-DS` described in Section 5.1.3.

To explore different ranges for the energy-performance tradeoff, we use an internal disk shutdown aggressiveness knob. The knob lowers the hit rate target $HR_{target}$ given to the cache sizing algorithm with respect to `NOOFF`, and controls how aggressively that algorithm turns disks off to save energy. In order to plot the tradeoff curve between energy savings and the corresponding hit rates, we run the simulation five times, each time with a higher aggressiveness than the previous run. In each run, the hit rate target $HR_{target}$ is lowered in steps of 5%. This creates five scenarios of gradually increasing energy savings.

In Figure 5.10, we plot the energy-performance tradeoff for algorithms `Random` and `Fixed`. For comparison, we also plot the tradeoff for the idealized algorithm `LRU-DS`. Note that an algorithm that provides a larger hit rate for a given reduction in energy can be deemed to be better. We see that `Fixed` offers a better energy-performance tradeoff when compared to `Random`. For instance, for a 30% energy reduction, `Random` has a normalized hit rate of 88%, while `Fixed` has a larger normalized hit rate of 91.5% (recall that normalized hit rate is the actual hit rate divided by the hit rate of `NOOFF`). The reasons for the superior hit rate performance of `Fixed` in comparison with `Random` are two-fold.

*1)* As noted in Figure 5.4, 80% of requests are for a small fraction of 1% of popular objects. In the case of `Fixed`, the popular objects that get accessed throughout the day tend to get replicated on higher disks that are seldom ever shut down. Thus, popular objects that account for almost all of the user requests are eventually always available on an active higher disk in cache, even when the lower disks are shut down. However, in the case of `Random`, there is a probability that copies of popular objects are made inactive, since the disks are shut down randomly, generating cache misses for future requests for them.

*2)* `Random` also has a greater replication factor $\rho$ than `Fixed`. The reason is that when `Fixed` replicates an object to a higher disk, more copies are not likely to be needed since that copy is likely to be available at all times. However, `Random` could continue to make more copies with some probability, since the existing copies could be made inactive by the random choice of disks for shutdown. Figure 5.11, shows the higher replication factor for `Random` in comparison with `Fixed`, for all five settings of the disk shutdown aggressiveness knob. Higher replication factor means that fewer unique objects are stored for a given cache size, resulting in a less efficient use of the cache space.

Figure 5.10 also plots the tradeoff for `LRU-DS` that shows a tradeoff that is better than `Fixed`, for instance, a 30% energy savings can be had with a normalized hit rate of 96%. This suggests that further improvements are possible, motivating our quest for better content placement & eviction algorithms in Section 5.4.

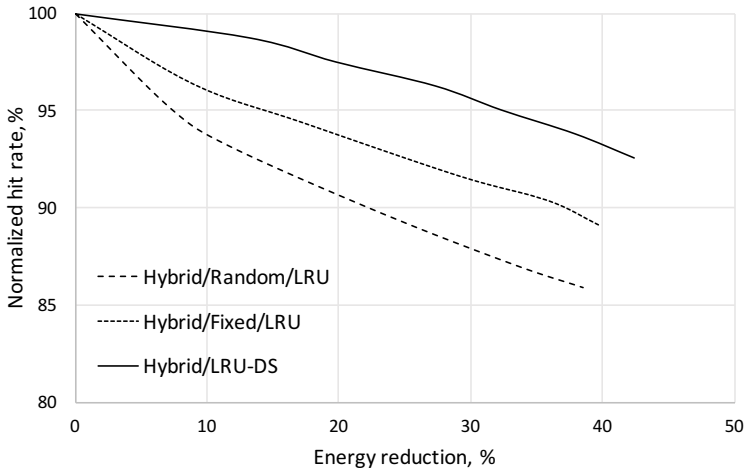


Figure 5.10: `Fixed` disk shutdown provides a better energy-performance tradeoff than `Random` disk shutdown.

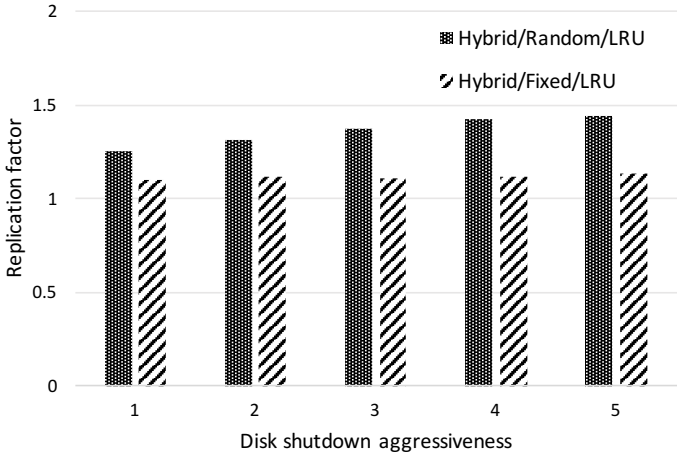


Figure 5.11: `Fixed` replicates content less than `Random` resulting in a more efficient use of the cache space.

*Concluding remark:* `Fixed` disk shutdown algorithm should be used to select disks that need to be switched on and off since it offers a better energy-performance tradeoff. We use `Fixed` as our default disk shutdown algorithm in our experimental results in the future sections.

## 5.4 Content placement & eviction algorithms

Cache management schemes comprise content placement algorithms & eviction algorithms that work together to manage the objects in cache. The cache sizing algorithms and the disk shutdown algorithms described in Sections 5.2 and 5.3 control the number of disks, and the actual disks that are shut down; but they have no control over the placement of objects on those disks. Content placement algorithms choose one among all the active disks in a server, to place the requested object on. Content eviction algorithms select the objects to be evicted from active disks to make space for new ones. In this section, we describe two algorithms for content placement & eviction. We assume that `Hybrid` sizing and `Fixed` shutdown are used with the two algorithms studied in this section.

*1) Random object placement with LRU eviction* (`LRU`). `LRU` is a baseline scheme and all the algorithms evaluated in Section 5.3 used `LRU`. Each requested object is placed on a randomly selected active disk. All objects are part of one LRU list. During eviction, the least recently accessed objects on active disks are evicted until enough cache space is reclaimed.

*2) Segmented placement and LRU* (`SLRU`). `LRU` is oblivious to the fixed order in which disks are shut down. `SLRU` avoids the drawbacks of `LRU` by placing more popular objects on higher numbered disks that are less likely to be shut down. Specifically, `SLRU` divides the cache space into $k$ equally sized segments, where segment 1 contains the least popular objects and segment $k$ contains the most popular objects. Every incoming object is first placed in segment 1. Each subsequent request for that object

migrates it one segment up, until it reaches segment $k$. Objects in each segment are evicted independently using the LRU eviction policy. Hence, we have $k$ LRU lists, one for each segment. In this work, we assume that that the maximum number of segments in a cache is the number of disks in the server, $k \leq n$. Since the available cache space in the server is reduced or increased at the granularity of a disk, $k > n$ provides no benefit from the perspective of disk energy savings.

### 5.4.1 Experimental evaluation

We empirically evaluate the hit rate performance of `LRU` and `SLRU` by performing simulations using the CDN content request traces. In particular, as we experiment with `LRU` and `SLRU`, we use `Hybrid` and `Fixed` as the cache sizing and disk shutdown algorithms respectively. To explore different ranges for the energy-performance trade-off, we use 5 different settings for the disk shutdown aggressiveness knob as before. Recall that for higher values of the knob, `Hybrid` cache sizing will lower the target hit rate $HR_{target}$, resulting in fewer disks being active, saving more energy. As in prior experiments, we set $\tau_l = 60s$, $\tau_{hrc} = 6\ hrs$. Further, we simulate the simplest form of `SLRU` that has $k = 2$ segments, i.e., there is a segment with the 4 higher-ordered disks and a segment with the 4 lower-ordered disks,

From Figure 5.12, we see that `SLRU` has higher hit rate than `LRU` for any given value of the energy reduction. For instance, for a 30% reduction in energy, `SLRU` has a normalized hit rate of 93.5%, while `LRU` has a normalized hit rate of 91.5%. This is due to the fact that `SLRU` is cognizant of the manner in which algorithm `Fixed` shuts down disks and places popular objects accessed more than once in the higher segment. Since the higher segment resides in the higher numbered disks, it is unlikely to be shut down.

### 5.4.2 Disk power cycles and impact on lifetimes

One of the major impacts on disk lifetime is the number of disk power cycles, in other words the number of times the disk is switched off and switched on. Disks used in current CDN servers are limited to anywhere from 7 to 35 in the number of disk power cycles per day, given that servers are upgraded every 4-5 years. We measure the average number of disk power cycles per day for all 5 simulations and see that both `LRU` and `SLRU` have ∼2-4 disk transitions per day. This is within the bounds of manufacturer specifications and hence disk shutdown is feasible without sacrificing disk lifetimes.

*Concluding remark:* The `Hybrid/Fixed/SLRU` scheme provides the best energy-performance tradeoff from the standpoint of a single server. In addition, the significant energy savings are obtainable without significant impact on disk lifetimes.

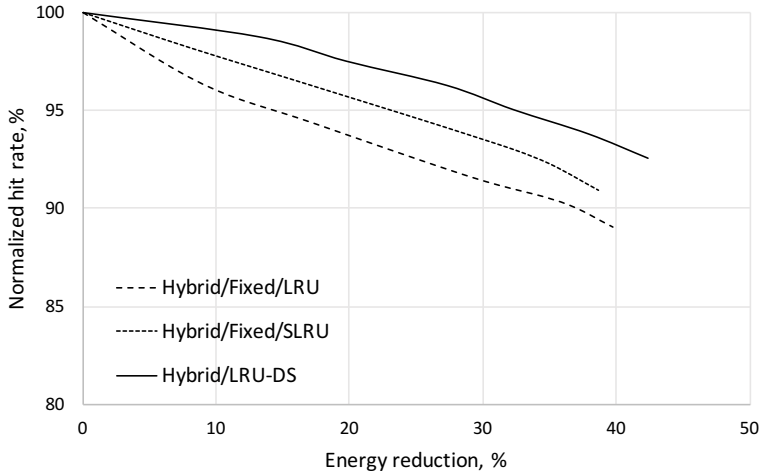


Figure 5.12: Energy-performance tradeoff of content placement & eviction algorithms.

## 5.5 Understanding cluster hit rates

In a typical CDN cluster, popular objects are often stored in more than one server. Algorithms such as consistent hashing [69, 81] that replicate content within a cluster were originally designed to better balance the load within the cluster. Here we study

how disk shutdown interacts with load balancing and content replication within a cluster by simulating an entire CDN server cluster consisting of 5 servers.

When the requested object is unavailable on the chosen server, due to disk shutdown or otherwise, the server attempts to fetch a copy from a peer server within the same cluster via ICP [116]. If no copy of the object is found within the cluster, the server fetches it from the origin server over the WAN. While server cache hits are the most desirable scenario, cluster hits also often provide adequate performance, since latencies between servers in the same datacenter are quite small. Further, ICP transfers within the same cluster incur no cost for the CDN, since such ICP traffic does not exit the datacenter. The least desirable scenario is when the object experiences a cluster miss, and the object will have to be fetched from a distant origin server over the WAN. Such WAN traffic incurs additional bandwidth costs for the CDN. Thus, the cluster hit rate that we study in this section is important both from a performance and cost perspective.

### 5.5.1 Experimental evaluation

In this work, thus far, we have looked at cache hit rates within a server. This section evaluates the energy-performance tradeoff from the standpoint of cluster hit rates. To perform the evaluation, we ran cluster-wide simulations where every server independently shuts down disks using one of the proposed cache management schemes: `Hybrid/Random/LRU`, `Hybrid/Fixed/LRU` and `Hybrid/Fixed/SLRU`. For each cache management scheme, we ran the simulation five times, each time with a higher disk shutdown aggressiveness knob than the previous run. The normalized cluster hit rates for the three cache management schemes are shown in Figure 5.13. We see that the difference in the normalized cluster hit rates for the three cache management schemes are not as significant as the differences in the (server) hit rates observed for these schemes in Figures 5.10 and 5.12. From a cluster hit rate perspective, simple disk

shutdown algorithms such as `Hybrid/Random/LRU` and `Hybrid/Fixed/LRU` provide energy-performance tradeoffs that are comparable to `Hybrid/Fixed/SLRU`. This is due to the following reasons: *1)* since popular objects are replicated across servers, a cluster miss occurs only when all the copies of an object within the cluster are inaccessible, and *2)* content placement and eviction are performed independently on each server within a cluster. Hence, the probability that all the copies of an object will be on inactive disks is now much smaller for all three cache management schemes.

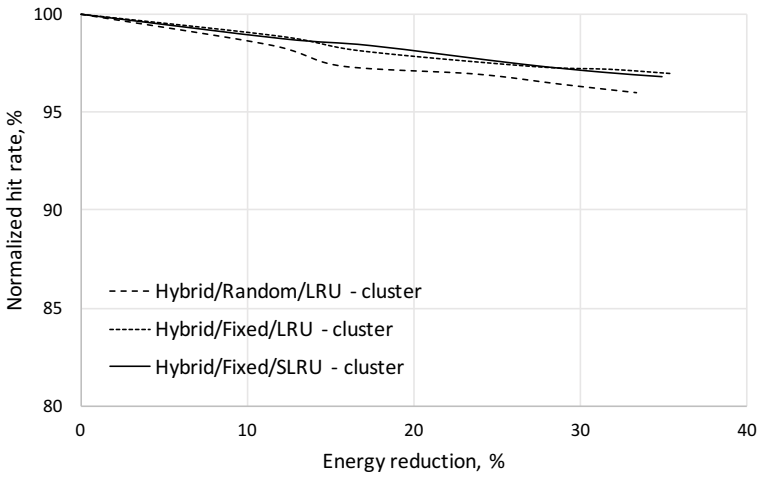


Figure 5.13: Energy-performance tradeoff of disk shutdown within a cluster.

We also compare the normalized cluster hit rate for `Hybrid/Fixed/LRU` with its normalized (server) hit rate in Figure 5.14. We see that cluster hit rates have a better energy-performance tradeoff when compared to single server hit rates. For instance, for an energy reduction of 30%, the normalized cluster hit rate reduces by a mere 3%, in comparison with the normalized server hit rate that reduces by 6.5%. In this case, the *absolute* hit rate reduction for the cluster hit rate and the server hit rate were 2.5% and 5% respectively. Thus, shutting down disks to save energy has a much smaller impact on cluster hit rates than server hit rates.

*Concluding remark:* For CDNs primarily concerned with maximizing cluster hit rates to minimize bandwidth costs, our simple scheme `Hybrid/Fixed/LRU` is an attrac-

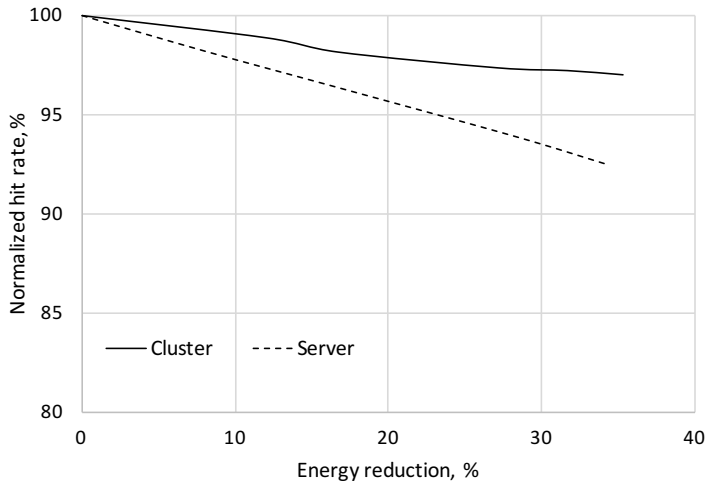


Figure 5.14: Cluster hit rate provide a better energy-performance tradeoff because of content replication across servers.

tive option, as it saves significant amounts of energy with only a modest performance loss and a small implementation overhead. However, CDNs interested in maximizing both server and cluster hit rates must invest in more sophisticated algorithms such as `Hybrid/Fixed/SLRU`.

## 5.6 Related work

First we review work that reduce disk energy consumption by the use of multi-speed disks that consume less energy by rotating at lower speeds at periods of low load. The authors in [58] propose the use of multi-speed disks to reduce disk power consumption, instead of shutting down disks. The rotation speed is chosen proportional to the disk load. The work in [20] is focused on reducing the energy consumption of disks in network servers that serve web traffic. The authors in this work, also use multi-speed disks, to conserve energy while maintaining the server's throughput. Hibernator [125], is another work that uses multi-speed disks to reduce energy consumption. This work is targeted at conserving disk energy in disk arrays that serve transaction workloads. Our work differs from these approaches in that we consider

typical CDN servers that are commodity hardware with fixed speed disks, and typical CDN workloads. Further, our approach saves additional energy by shutting down disks entirely, an approach that is feasible for CDNs but not for other enterprise networks that store original content.

Other work that attempt to switch disks to a lower power mode include [97] where popular objects are migrated to a subset of disks so that the other disks can be switched to a low-power mode without affecting the performance of the server. The decision to switch disks to low-power modes is based on the incoming request rate. In [16], the size of the front-end cache is optimized to reduce the energy consumption of back-end disks in a storage system, while the front-end cache consumes power. The cache size is estimated as being proportional to the disk load. However, in our CDN context, we identify that the disk load alone is not a good indicator of the cache space requirement. Complementary to our work, there is significant work on saving energy in other components of the server such as CPU, including dynamic power scaling and dynamic component deactivation [41, 51, 115, 117].

Prior works also propose turning off servers entirely in the context of a data center [23, 78, 80, 111] and in the context of a CDN [84]. However, turning off servers makes network management difficult in a global CDN. If servers are unreachable for extended periods of time, they may miss real-time reporting, software updates and control messages for the duration. This may upend some of the network management guarantees and operational practices of the CDN platform. Therefore, shutting down disks as proposed in this work, while the servers are still live and serving content, represents an attractive alternative worth exploring.

## 5.7 Conclusion

Reducing the energy consumption of CDNs is an important problem, both from the standpoint of sustainability and OPEX cost reduction. The energy consumed by

spinning disks constitute a significant portion of a CDN's energy usage. Our work explores the possibility of reducing the disk energy usage by shutting down disks, a possibility that is particularly well-suited for CDNs since these disks do not store original copies of the content. Our main contribution is developing and evaluating algorithms for cache sizing, disk shutdown, and content placement & eviction that allow disks to be shut down without significantly impacting cache hit rates and user-perceived performance. We empirically evaluate the energy-performance tradeoff for our algorithms using extensive request traces from the world's largest CDN. We show that it is feasible to obtain 30% disk energy savings with only a 6.5% reduction in the normalized server hit rate and a mere 3% reduction in the normalized cluster hit rate. This work establishes disk shutdown as a key mechanism for energy savings in CDNs, making it a prime candidate for implementation in the production network.

# CHAPTER 6

# SUMMARY AND FUTURE WORK

In this dissertation, we propose algorithms for traffic provisioning and cache management that enhance the performance and reduce the cost of content delivery. We show that the proposed algorithms are easy to implement and evaluate them using production traces from Akamai's CDN. Some of the proposed algorithms have been implemented in a production setting and others are being evaluated for future deployments. We summarize our contributions below.

– *Traffic provisioning:* Traffic provisioning is the process of deciding which traffic classes are hosted in which set of servers across the CDN to maximize the cache hit rate of the network. We propose footprint descriptors that are a succinct representation of request traces that belong to different traffic classes. We show that footprint descriptors accurately predict the effects of traffic mixing which is crucial for traffic provisioning. Footprint descriptors are currently deployed in Akamai's production network to provision traffic across the network. We also propose optimization models for traffic provisioning that minimize the midgress traffic. We show that midgress-aware traffic provisioning can be easily deployed in production settings.

– *Cache management:* Given a stream of user requests sent to a cache server (or cluster of servers), cache management is the process of deciding which objects enter cache and which objects are evicted from cache, to maximize the cache hit rate. We propose two adaptive TTL-based cache management algorithms, d-TTL and f-TTL that adapt to non-stationary traffic and provably achieve a desired cache hit rate and expected cache size. We also propose energy-efficient cache management

algorithms using disk shutdown. We show that disk shutdown provides a good energy-performance tradeoff and is well suited for CDN servers.

## 6.1 Future work

The algorithms proposed in this dissertation advance the state-of-the-art in traffic provisioning and cache management to improve the performance of CDNs. But these techniques merely scratch the surface of what needs to be done and there are several directions that can be taken to further maximize performance. We list a few ideas for future work.

### 6.1.1 Traffic provisioning of hierarchical caches

In this dissertation, we have focused on improving the performance of single level caches either within a cluster or within an entire metro area. While some of our proposed algorithms consider inter-server communication using ICP, we have not considered a hierarchical network of servers which is common in many production settings. Accurately modeling hierarchical cache networks that serve non-stationary traffic can be extremely useful in understanding how traffic needs to be provisioned across the entire deployed network to maximize cache hit rates. However, the main roadblock is the fact that in a multi-level cache hierarchy, inferring the cache miss traffic from the lower layer to the next higher layer is a hard problem. Our work on footprint descriptors (Chapter 2) efficiently models the traffic in the lower most layer. As future work, the footprint descriptor calculus could be extended to include a "filter" operation where given the footprint descriptor of a request trace and some cache of size $C$, the filter operator computes the footprint descriptor of the cache miss traffic resulting from the cache of size $C$. Such a calculus could be used to compute the "filtered" footprint descriptor for different hierarchical settings and could also be

incorporated into the midgress-aware traffic provisioning model proposed in Chapter 3 to reduce midgress.

### 6.1.2 Modeling server provisioning

Traffic provisioning is typically constrained by the available resources such as server storage capacities, network bandwidth, disk and CPU throughputs and so on. In Chapter 3, we show how traffic provisioning can be modeled and evaluated to minimize midgress. The "dual" of traffic provisioning is *server provisioning*. Server provisioning is the problem of deciding how many and what configuration of servers should be deployed to meet the performance requirements of traffic classes served by a CDN while also minimizing the capital and operational expenditures (CAPEX and OPEX). For example, one instance of server provisioning could be as follows. Given $N$ traffic classes with each traffic class having a maximum cache miss rate threshold $MR_j, 1 \leq j \leq N$, determine the minimum number of servers $M$ with cache size $C$ GB and capacity $T$ Gbps each, that are required to minimize the CAPEX + OPEX of the CDN. The optimization model proposed in Chapter 3 could be easily extended to model server provisioning. More complex server provisioning models could take into account the types of hardware being used where different hardware models for a given resource would present a cost-performance tradeoff. For example, while SSDs provide fast content access when compared to spinning disks, they are much more expensive. Hence a hybrid deployment of SSDs and spinning disks, that accounts for this tradeoff, might make more sense. A good understanding of server provisioning helps CDNs deploy and update infrastructure to sustain future traffic growth of existing traffic classes and incorporate new classes of traffic.

### 6.1.3 Machine learning based caching

With the rapid rise in the amount of traffic being served to end users, the variety of content types and the amount of content footprint being accessed, there is an

increasing need to develop smart and adaptive cache management algorithms that require little to no human intervention in setting different parameter values such as the minimum number of content accesses required for cache admission [81], minimum object size required to cache the object [12] and so on. Traditional cache management algorithms were not designed to be adaptive to such diverse workloads and hence do not work well in such settings. Our work on TTL-based caching (Chapter 4) is a step towards addressing this challenge. We show how to achieve feasible cache hit rate and cache size targets for caches serving non-stationary and bursty traffic. More recently, a promising technique to address this problem is to use machine learning based approaches to adaptively cache content to maximize hit rates [9, 32, 43, 72, 90, 106]. For instance, we show how cache admission could be modeled as a reinforcement learning problem [75]. The proposed algorithm, RL-Cache, uses multiple object features such as the object size, the frequency of access, the recency of access, and multiple combinations of these features to train a reinforcement learning model to maximize object hit rates. RL-Cache achieves high object hit rates when compared to other state-of-the-art admission algorithms. But there is a lot more to be done. For example: *1)* How do we jointly optimize cache admission and cache eviction to maximize hit rates, *2)* How do we implement and deploy on-line learning models in the edge server without impacting throughput, and *3)* How to auto-tune machine learning model parameters in dynamic settings such as CDNs with little or no human intervention.

# APPENDIX A

# SOLVING THE OPTIMIZATION MODEL FOR MIDGRESS-AWARE TRAFFIC PROVISIONING

The optimization model for midgress-aware traffic provisioning described in Section 3.1.2 is non-convex and hence an NP-hard problem to solve. We make a few assumptions to transform the problem to a mixed integer linear program (MILP) that can be solved using CPLEX.

## A.1 Midgress-aware traffic provisioning as a MILP

At a high level, our goal is to transform every biconvex term (product of two convex functions) in the original optimization model to bilinear terms that are a product of one continuous term and one binary term. The bilinear terms so formed can then be linearized to yield a MILP. The transformations are described in the following subsections as we linearize all terms in the original non-convex problem. We present the original non-convex terms first followed by the reformulations.

### A.1.1 Objective function

The objective of midgress-aware traffic provisioning is to assign $N$ traffic classes to $M$ sites such that the midgress traffic from all the sites is minimized.

*Original formulation:* The objective function in the original formulation is

$$\text{min.} \sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij} \lambda_j \mathcal{M}_j(c_{ij}).$$

In the objective function, the load fraction $x_{ij}$ is a linear term, and the MRC of each traffic class $\mathcal{M}_j(c_{ij})$ where $c_{ij}$ is the cache size occupied by traffic class $j$ in site $i$, is a convex function. The load of traffic class $j$, $\lambda_j$ is a constant term.

*MILP reformulation:* We take the following steps to linearize the objective and in the process add more linear constraints.

1) Load fraction $x_{ij}$ can only take a discrete set of values. For some integer $F$, $x_{ij} \in \{0, \frac{1}{F}, \frac{2}{F}, \ldots, \frac{F-1}{F}, 1\}$. Hence, $x_{ij}$ can be rewritten as $x_{ij} = \frac{\sum_{f=1}^{F} x_{ijf}^b}{F}$, where $x_{ijf}^b$ are binary variables.

2) The MRC of traffic class $j$, $\mathcal{M}_j(c_{ij})$ can be represented by a piece-wise linear approximation with $K$ linear functions. Let $a_{jk}$ and $b_{jk}$ represent the coefficients of the $K$ linear functions that approximate the MRC of traffic class $j$. Let $m_{ij}$ be a non-zero variable representing the miss rate of traffic class $j$ in site $i$. Then, we replace $\mathcal{M}_j(c_{ij})$ with $m_{ij}$ in the objective and introduce $K$ affine constraints $a_{jk}c_{ij} + b_{jk} \leq m_{ij}, \quad \forall\, j(i)(k) = 1 \ldots N(M)(K)$.

After these two transformations we have the intermediate objective function

$$\text{min.} \sum_{i=1}^{M} \sum_{j=1}^{N} \left( \frac{\sum_{f=1}^{F} x_{ijf}^b}{F} \right) m_{ij}$$

The final step is to linearize every bilinear product term in the objective, $x_{ijf}^b m_{ij}$, which is a product of a binary variable and a continuous variable. We use standard McCormick envelope based linearization for this transformation by introducing a new continuous variable $z_{ijf} = x_{ijf}^b m_{ij}, \ \forall\ i(j)(f) = 1 \ldots M(N)(F)$. Putting it all together, the linearized objective, the piecewise linear MRCs and the McCormick envelope constraints are

$$\text{min.} \sum_{i=1}^{M} \sum_{j=1}^{N} \left( \frac{\sum_{f=1}^{F} z_{ijf}}{F} \right) \tag{A.1}$$

$$a_{jk}c_{ij} + b_{jk} \leq m_{ij} \tag{A.2}$$

$$z_{ijf} \leq x^b_{ijf} * m^{UB}_{ij} \tag{A.3}$$

$$z_{ijf} \leq m_{ij} \tag{A.4}$$

$$z_{ijf} \geq m_{ij} - (1 - x^b_{ijf}) m^{UB}_{ij} \tag{A.5}$$

$$z_{ijf} \geq 0 \tag{A.6}$$

where, $m^{UB}_{ij}$ and $m^{LB}_{ij}$ are the upper and lower bounds of the cache miss rates of each traffic class in each site.

After the transformations, Equation A.1 is the new linearized objective function. In the process of linearizing the objective, we have added additional linear constraints, Equation A.2 for the MRCs and Equations A.3-A.6 for the McCormick envelopes.

### A.1.2 Resource constraints

*Original formulation:* The cache size and the capacity constraints from the original model are as follows.

$$\sum_{j=1}^{N} c_{ij} \leq C_i \quad \forall i = 1 \ldots M$$

$$\sum_{j=1}^{N} x_{ij} \lambda_j \leq T_i \quad \forall i = 1 \ldots M$$

*MILP reformulation:* The cache size constraint remains unchanged in the MILP. The capacity constraint is updated to incorporate the discrete set of values that the load fraction $x_{ij}$ is allowed to take.

$$\sum_{j=1}^{N} c_{ij} \leq C_i \quad \forall i = 1 \ldots M \tag{A.7}$$

$$\sum_{j=1}^{N} \left( \frac{\sum_{f=1}^{F} x^b_{ijf}}{F} \right) \lambda_j \leq T_i \quad \forall i = 1 \ldots M \tag{A.8}$$

### A.1.3 Eviction age equality constraint

*Original formulation:* The eviction age equality constraint in the original model is as follows.

$$\mathcal{T}_j(c_{ij}, \lambda_j) = \rho_i x_{ij} \quad \forall j(i) = 1 \ldots N(M)$$

The eviction age function $\mathcal{T}_j(c_{ij}, \lambda_j)$, where $c_{ij}$ is the cache size occupied by traffic class $j$ in site $i$ is convex, the eviction age of site $i$, $\rho_i$ is linear and so is the load fraction $x_{ij}$.

*MILP reformulation:* To simplify the linearization process, we rewrite the eviction age constraint in the original formulation by introducing an intermediate variable $t_{ij}$. The modified constraints are

$$t_{ij} = \rho_i x_{ij} \quad \forall j(i) = 1 \ldots N(M) \tag{A.9}$$

$$t_{ij} = \mathcal{T}_j(c_{ij}, \lambda_j) \quad \forall j(i) = 1 \ldots N(M) \tag{A.10}$$

We first linearize Equation A.9. We discretize $x_{ij}$ to take on only $F$ discrete values as before. This gives us

$$t_{ij} = \rho_i \left( \frac{\sum_{f=1}^{F} x_{ijf}^b}{F} \right).$$

Following the same steps as in Section A.1.1, we introduce a new variable $t'_{ijf} = \rho_i x_{ijf}^b$, and linearize the product using McCormick envelopes.

$$t_{ij} = \frac{\sum_{f=1}^{F} t'_{ijf}}{F} \tag{A.11}$$

$$t'_{ijf} \leq x_{ijf}^b * \rho_i^{UB} \tag{A.12}$$

$$t'_{ijf} \leq \rho_i \tag{A.13}$$

$$t'_{ijf} \geq \rho_{ij} - (1 - x_{ijf}^b)\rho_i^{UB} \tag{A.14}$$

$$t'_{ijf} \geq 0 \tag{A.15}$$

where $\rho_i^{UB}$ and $\rho_i^{LB}$ are the upper and lower bounds of the eviction ages of site $i$ respectively.

We now linearize Equation A.10. We approximate the eviction age function $\mathcal{T}_j(c_{ij}, \lambda_j)$ by a piece-wise linear function with $L$ pieces. We model the piece-wise linear function using variables that satisfy the special order set (SOS) of type 2 constraints. SOS type-2 variables are a set of consecutive variables in which no more that two adjacent variables are non-zero. Let $\Gamma_{ijl}$ be SOS type-2 variables for each piece $l$ and every combination of traffic class $j$ and site $i$. Then Equation A.10 can be linearized as follows.

$$t_{ij} = \sum_{l=1}^{L} \Gamma_{ijl} \mathcal{T}_{jl}^{y} \tag{A.16}$$

$$c_{ij} = \sum_{l=1}^{L} \Gamma_{ijl} \mathcal{T}_{jl}^{x} \tag{A.17}$$

$$\sum_{l=1}^{L} \Gamma_{ijl} = 1 \tag{A.18}$$

where $\mathcal{T}_{jl}^{y}$ are the Y coordinates of the eviction age function $\mathcal{T}_j(c_{ij})$ at the $L$ breakpoints with X coordinates $\mathcal{T}_{jl}^{x}$.

Putting it all together, Equations A.11-A.18 are the linearized eviction age equality constraints in the MILP.

### A.1.4 Load assignment constraint

*Original formulation:* The load assignment in the original model is

$$\sum_{i=1}^{M} x_{ij} = 1 \quad \forall j = 1 \ldots N$$

*MILP reformulation:* In the MILP reformulation, the load fraction $x_{ij}$ is transformed to take on a discrete set of values.

$$\sum_{i=1}^{M} \left( \frac{\sum_{f=1}^{F} x_{ijf}^{b}}{F} \right) = 1 \quad \forall j = 1 \ldots N \tag{A.19}$$

### A.1.5 Non-negativity constraints

*Original formulation:* In the original formulation, the non-negativity constraints apply to the set of output variables $\rho_i$, $c_{ij}$ and $x_{ij}$ as follows.

$$\rho_i > 0 \quad \forall i = 1 \ldots M$$

$$c_{ij} \geq 0 \quad \forall j = 1 \ldots N$$

$$x_{ij} \in [0, 1] \quad \forall j(i) = 1 \ldots N(M)$$

*MILP reformulation:* In the transformed MILP, the non-negativity constraints apply to the output variables and the intermediate variables introduced as part of the transformations.

$$\rho_i > 0 \quad \forall i = 1 \ldots M \tag{A.20}$$

$$c_{ij} \geq 0 \quad \forall j = 1 \ldots N \tag{A.21}$$

$$x_{ijf}^{b} \in \{0, 1\} \quad \forall\, j(i) = 1 \ldots N(M) \tag{A.22}$$

$$m_{ij} \geq 0 \quad \forall\, j(i) = 1 \ldots N(M) \tag{A.23}$$

$$z_{ijf} \geq 0 \quad \forall\, j(i)(f) = 1 \ldots N(M)(F) \tag{A.24}$$

$$t_{ij} \geq 0 \quad \forall j(i) = 1 \ldots N(M) \tag{A.25}$$

$$t'_{ijf} \geq 0 \quad \forall\, j(i)(f) = 1 \ldots N(M)(F) \tag{A.26}$$

$$\Gamma_{ijl} \geq 0 \quad \forall\, j(i)(l) = 1 \ldots N(M)(L) \tag{A.27}$$

Now that we have linearized the objective and all the constraints in the original problem, Equations A.1-A.8 and A.11-A.27 represent the transformed MILP that can be solved using CPLEX.

# APPENDIX B

# DISK POWER MODEL

In the simulator, we used a power model similar to the 2-parameter disk-power model described in Dempsey [123]. In this model, the energy consumed by the disk is modeled as $E_{total} = E_{idle} + E_{active}$. The values of the two components of the total energy were empirically determined as follows.

– *Measurement of $E_{idle}$:* We used a typical CDN server that comes equipped with a power-supply unit that has a PMBus interface that allows us to query the server's power consumption at any time. We used the CDN server in the lab for disk power measurement in which four of the disks had no files or directories. A script running on this server queried the PMBus interface every 10 seconds to record a time-series of server power. All the processes other than the bare minimum needed to keep the server up were stopped. Then, the four disks were accessed for a duration of time, then left idle for a duration of time, and then spun down using the SCSI stop command. The time series of server power measurement collected during the time allows us to identify $P_{rotation}$ and $P_{electronics}$. Since the disks in CDN servers never get a chance to shut their electronics down, we use $P_{rotation} + P_{electronics}$ as $P_{idle}$, which is used to compute $E_{idle}$. Further, in the manufacturer's detailed data sheet, we located the maximum observed power consumption of the disk model. We call this $P_{max}$. $P_{max} - P_{idle}$ gives $P_{iomax}$, the maximum power that the disk's I/O activity can consume. These observations are listed below in Table B.1.

– *Model for $E_{active}$:* $E_{active}$ is the sum of the products of $T_{active}$ and $P_{active}$ for all the I/O operations, where $T_{active}$ is the time consumed by an I/O operation, and $P_{active}$ is

| Component | Power consumption (W) |
|---|---|
| $P_{iomax}$ | 2.25 |
| $P_{electronics}$ | 0.75 |
| $P_{rotation}$ | 3 |
| $P_{max}$ | 6 |

Table B.1: Disk power consumption.

the power consumption of that operation. In the typical CDN servers that we studied, the disk I/O pattern caused by serving web traffic has a very narrow range of bytes transferred per request. Over 85% of the I/O requests to the disks have transfer sizes of less than 100KB. Since most transfer sizes are clustered in such a narrow band, we do not create a generalized fine-grain model for $P_{active}$ covering a wide range of I/O sizes, but assume that $P_{active}$ is narrowly clustered around a mean. Due to the low variance property, $P_{active}$ can be said to scale linearly with $T_{active}$.

We model $T_{active}$ as a function of four disk activity parameters collected at the block layer. To create this model, we collected a large archive of disk statistics using the linux iostat command. The archive contains data points from each machine in a cluster of edge servers for 5 days. Each data point is of the form (read operations/sec, average read size, write operations/sec, average write size, $T_{active}$), each providing the average of a 30 second observation period. The value of $P_{active}$ for every data point in the archive was estimated as $P_{max} \times T_{active}/T_{total}$, where $T_{total}$ is the observation period. Using linear regression, the 172,800 data points so collected were converted to a power model, which expresses $T_{active}$ and $P_{active}$ as a piecewise linear function of the four disk activity parameters.

Ideally, the power model should also address the energy consumed by disk spin-up phase, which is easily obtained from measurement. But as we saw earlier, the number of power cycles per day per disk is low (approximately 3). Therefore, this component

has a minor impact on the total energy consumption, and is excluded from the power model.

# BIBLIOGRAPHY


[1] Amazon EMR, `https://aws.amazon.com/emr/`.

[2] iostat, `http://sebastien.godard.pagesperso-orange.fr/man_iostat.html`.

[3] Agarwal, A., Hennessy, J., and Horowitz, M. An analytical cache model. *ACM Trans. Comput. Syst. 7*, 2 (May 1989), 184–215.

[4] Almási, George, Caşcaval, Clin, and Padua, David A. Calculating stack distances efficiently. In *ACM SIGPLAN Notices* (2002), vol. 38, ACM, pp. 37–43.

[5] Arteaga, Dulcardo, Cabrera, Jorge, Xu, Jing, Sundararaman, Swaminathan, and Zhao, Ming. Cloudcache: On-demand flash cache management for cloud computing. In *FAST* (2016), pp. 355–369.

[6] Bansal, Sorav, and Modha, Dharmendra S. Car: Clock with adaptive replacement. In *FAST* (2004), vol. 4, pp. 187–200.

[7] Basu, Soumya, Sundarrajan, Aditya, Ghaderi, Javad, Shakkottai, Sanjay, and Sitaraman, Ramesh. Adaptive ttl-based caching for content delivery. *IEEE/ACM transactions on networking 26*, 3 (2018), 1063–1077.

[8] Belady, C.L. In the data center, power and cooling costs more than the IT equipment it supports. *Electronics Cooling 13*, 1 (2007), 24.

[9] Berger, D. Towards lightweight and robust machine learning for cdn caching. HotNets 2018.

[10] Berger, Daniel S, Gland, Philipp, Singla, Sahil, and Ciucu, Florin. Exact analysis of ttl cache networks. *Performance Evaluation 79* (2014), 2–23.

[11] Berger, Daniel S, Henningsen, Sebastian, Ciucu, Florin, and Schmitt, Jens B. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review 43*, 2 (2015), 57–59.

[12] Berger, Daniel S, Sitaraman, Ramesh K, and Harchol-Balter, Mor. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (2017), USENIX Association, pp. 483–498.

[13] 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? *ACM SIGCOMM Computer Communication Review 43*, 3 (2013), 59–67.

[14] Borst, Sem, Gupta, Varun, and Walid, Anwar. Distributed caching algorithms for content distribution networks. In *INFOCOM, 2010 Proceedings IEEE* (2010), Citeseer, pp. 1–9.

[15] Burville, PJ, and Kingman, JFC. On a model for storage and search. *Journal of Applied Probability* (1973), 697–701.

[16] Cai, Le, and Lu, Yung-Hsiang. Power reduction of multiple disks using dynamic cache resizing and speed control. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)* (2006).

[17] Cao, Pei, and Irani, Sandy. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems* (1997), vol. 12, pp. 193–206.

[18] Cardellini, Valeria, Colajanni, Michele, and Yu, Philip S. Request redirection algorithms for distributed web systems. *IEEE transactions on parallel and distributed systems 14*, 4 (2003), 355–368.

[19] Carra, Damiano, and Michiardi, Pietro. Memory partitioning and management in memcached. *IEEE Transactions on Services Computing* (2016).

[20] Carrera, Enrique V., Pinheiro, Eduardo, and Bianchini, Ricardo. Conserving disk energy in network servers. In *Proceedings of the 17th Annual International Conference on Supercomputing (ICS)* (2003).

[21] Carter, Robert L, and Crovella, Mark E. Server selection using dynamic path characterization in wide-area networks. In *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE* (1997), vol. 3, IEEE, pp. 1014–1021.

[22] Chandra, Dhruba, Guo, Fei, Kim, Seongbeom, and Solihin, Yan. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proc. 11th Int. Symp. High-Performance Computer Architecture* (Feb. 2005), pp. 340–351.

[23] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing energy and server resources in hosting centers. In *Proc. ACM SIGOPS* (2001).

[24] Che, Hao, Tung, Ye, and Wang, Zhijun. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications 20*, 7 (2002), 1305–1314.

[25] Chen, Chung-Min, Ling, Yibei, Pang, Marcus, Chen, Wai, Cai, Shengwei, Suwa, Yoshihisa, and Altintas, Onur. Scalable request routing with next-neighbor load sharing in multi-server environments. In *Advanced Information Networking and Applications, 2005. AINA 2005. 19th International Conference on* (2005), vol. 1, IEEE, pp. 441–446.

[26] Chen, Fangfei, Sitaraman, Ramesh K, and Torres, Marcelo. End-user mapping: Next generation request routing for content delivery. In *ACM SIGCOMM Computer Communication Review* (2015), vol. 45, ACM, pp. 167–181.

[27] Cherkasova, Ludmila. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy.* Hewlett-Packard Laboratories, 1998.

[28] Chu, Weibo, Dehghan, Mostafa, Lui, John CS, Towsley, Don, and Zhang, Zhi-Li. Joint cache resource allocation and request routing for in-network caching services. *Computer Networks 131* (2018), 1–14.

[29] Cidon, Asaf, Eisenman, Assaf, Alizadeh, Mohammad, and Katti, Sachin. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (2015).

[30] Cidon, Asaf, Eisenman, Assaf, Alizadeh, Mohammad, and Katti, Sachin. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (2016), pp. 379–392.

[31] Cidon, Asaf, Rushton, Daniel, Rumble, Stephen M, and Stutsman, Ryan. Memshare: a dynamic multi-tenant key-value cache. In *Usenix ATC* (2017).

[32] Cobb, J., and ElAarag, H. Web proxy cache replacement scheme based on backpropagation neural network. vol. 81, pp. 1539–1558.

[33] Coffman, Edward G., and Jelenković, Predrag. Performance of the move-to-front algorithm with Markov-modulated request sequences. *Operations Research Letters 25* (1999), 109–118.

[34] Coffman, Edward Grady, and Denning, Peter J. *Operating systems theory*. Prentice-Hall, 1973.

[35] Dan, Asit, and Towsley, Don. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *ACM SIGMETRICS* (1990), pp. 143–152.

[36] Dehghan, M., Massouli, L., Towsley, D., Menasch, D. S., and Tay, Y. C. A utility optimization approach to network cache design. *IEEE/ACM Transactions on Networking 27*, 3 (June 2019), 1013–1027.

[37] Dehghan, Mostafa, Chu, Weibo, Nain, Philippe, Towsley, Don, and Zhang, Zhi-Li. Sharing cache resources among content providers: A utility-based approach. *IEEE/ACM Transactions on Networking (TON) 27*, 2 (2019), 477–490.

[38] Ding, Chen, and Chilimbi, Trishul. A composable model for analyzing locality of multi-threaded programs. Tech. rep., Technical Report MSR-TR-2009-107, Microsoft Research, 2009.

[39] Dobrow, Robert P, and Fill, James Allen. The move-to-front rule for self-organizing lists with Markov dependent requests. In *Discrete Probability and Algorithms*. Springer, 1995, pp. 57–80.

[40] Einziger, Gil, Friedman, Roy, and Manes, Ben. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS) 13*, 4 (2017), 35.

[41] Elnozahy, E., Kistler, M., and Rajamony, R. Energy-efficient server clusters. *Power-Aware Computer Systems* (2003), 179–197.

[42] Fagin, Ronald. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences 14*, 2 (1977), 222–250.

[43] Fedchenko, Vladyslav, Neglia, Giovanni, and Ribeiro, Bruno. Feedforward neural networks for caching: Enough or too much? *SIGMETRICS Perform. Eval. Rev. 46*, 3 (Jan. 2019), 139–142.

[44] Feldman, Michal, and Chuang, John. Service differentiation in web caching and content distribution. In *Proceedings of the IASTED International Conference on Communications and Computer Networks* (2002).

[45] Fill, James Allen, and Holst, Lars. On the distribution of search cost for the move-to-front rule. *Random Structures & Algorithms 8* (1996), 179–186.

[46] Flajolet, Philippe, Gardy, Daniele, and Thimonier, Loÿs. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics 39* (1992), 207–229.

[47] Fofack, N Choungmo, Nain, Philippe, Neglia, Giovanni, and Towsley, Don. Analysis of ttl-based cache networks. In *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on* (2012), IEEE, pp. 1–10.

[48] Fricker, Christine, Robert, Philippe, and Roberts, James. A versatile and accurate approximation for lru cache performance. In *Proceedings of the 24th International Teletraffic Congress* (2012), International Teletraffic Congress, p. 8.

[49] Fricker, Christine, Robert, Philippe, Roberts, James, and Sbihi, Nada. Impact of traffic mix on caching performance in a content-centric network. In *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on* (2012), IEEE, pp. 310–315.

[50] Gallo, Massimo, Kauffmann, Bruno, Muscariello, Luca, Simonian, Alain, and Tanguy, Christian. Performance evaluation of the random replacement policy for networks of caches. In *ACM SIGMETRICS/ PERFORMANCE* (2012), pp. 395–396.

[51] Gandhi, A., Harchol-Balter, M., Das, R., and Lefurgy, C. Optimal power allocation in server farms. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems* (2009), ACM, pp. 157–168.

[52] Garetto, Michele, Leonardi, Emilio, and Martina, Valentina. A unified approach to the performance analysis of caching systems. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems 1*, 3 (2016), 12.

[53] Gast, Nicolas, and Van Houdt, Benny. Transient and steady-state regime of a family of list-based cache replacement algorithms. *ACM SIGMETRICS Performance Evaluation Review 43*, 1 (2015), 123–136.

[54] Gast, Nicolas, and Van Houdt, Benny. Asymptotically exact ttl-approximations of the cache replacement algorithms lru (m) and h-lru. In *Teletraffic Congress (ITC 28), 2016 28th International* (2016), vol. 1, IEEE, pp. 157–165.

[55] Gelenbe, E. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers C-22*, 6 (June 1973), 611–618.

[56] Goemans, Michel. Load balancing in content deliverynetworks. *MA Annual Program Year Workshop:Network Management and Design* (April 2003).

[57] Guillemin, Fabrice, Kauffmann, Bruno, Moteau, Stephanie, and Simonian, Alain. Experimental analysis of caching efficiency for youtube traffic in an isp network. In *Teletraffic Congress (ITC), 2013 25th International* (2013), IEEE, pp. 1–9.

[58] Gurumurthi, Sudhanva, Sivasubramaniam, Anand, Kandemir, Mahmut, and Franke, Hubertus. Drpm: Dynamic speed control for power management in server class disks. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)* (2003).

[59] Hendricks, WJ. The stationary distribution of an interesting Markov chain. *Journal of Applied Probability* (1972), 231–233.

[60] Hu, Xiameng, Wang, Xiaolin, Zhou, Lan, Luo, Yingwei, Ding, Chen, and Wang, Zhenlin. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (2016), USENIX Association, pp. 351–364.

[61] Huang, Qi, Birman, Ken, van Renesse, Robbert, Lloyd, Wyatt, Kumar, Sanjeev, and Li, Harry C. An analysis of Facebook photo caching. In *ACM SOSP* (2013), pp. 167–181.

[62] Index, Cisco Visual Networking. The zettabyte era: Trends and analysis, `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html`.

[63] Ioannidis, Stratis, and Yeh, Edmund. Jointly optimal routing and caching for arbitrary network topologies. *IEEE Journal on Selected Areas in Communications* (2018).

[64] Jelenković, Predrag R. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *The Annals of Applied Probability 9* (1999), 430–464.

[65] Jelenković, Predrag R, and Radovanović, Ana. Least-recently-used caching with dependent requests. *Theoretical computer science 326* (2004), 293–327.

[66] Jung, Jaeyeon, Berger, Arthur W, and Balakrishnan, Hari. Modeling ttl-based internet caches. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies* (2003), vol. 1, IEEE, pp. 417–426.

[67] Kamp, Poul-Henning. Varnish LRU architecture, June 2007. Available at `https://www.varnish-cache.org/trac/wiki/ArchitectureLRU`, accessed 09/12/16.

[68] Karedla, Ramakrishna, Love, J Spencer, and Wherry, Bradley G. Caching strategies to improve disk system performance. *Computer*, 3 (1994), 38–46.

[69] Karger, David, Lehman, Eric, Leighton, Tom, Panigrahy, Rina, Levine, Matthew, and Lewin, Daniel. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), ACM, pp. 654–663.

[70] Karger, David, Sherman, Alex, Berkheimer, Andy, Bogstad, Bill, Dhanidina, Rizwan, Iwamoto, Ken, Kim, Brian, Matkins, Luke, and Yerushalmi, Yoav. Web caching with consistent hashing. *Computer Networks 31*, 11 (1999), 1203–1213.

[71] Kelly, Terence, Chan, Yee Man, Jamin, Sugih, and MacKie-Mason, Jeffrey. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value.

[72] Khalid, H., and Obaidat, M. S. Kora-2: A new cache replacement policy and its performance. ICECS 1999.

[73] Kim, Seongbeom, Chandra, Dhruba, and Solihin, Yan. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques* (2004), IEEE Computer Society, pp. 111–122.

[74] King, W. Frank. Analysis of demand paging algorithms. In *IFIP Congress (1)* (1971), pp. 485–490.

[75] Kirilin, Vadim, Sundarrajan, Aditya, Gorinsky, Sergey, and Sitaraman, Ramesh K. Rl-cache: Learning-based cache admission for content delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML* (2019), ACM, pp. 57–63.

[76] Ko, Bong-Jun, Lee, Kang-Won, Amiri, Khalil, and Calo, Seraphin. Scalable service differentiation in a shared storage cache. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on* (2003), IEEE, pp. 184–193.

[77] Koomey, J.G. Worldwide electricity used in data centers. *Environmental Research Letters 3* (Sept 2008).

[78] Krioukov, A., Mohan, P., Alspaugh, S., Keys, L., Culler, D., and Katz, R. Napsac: Design and implementation of a power-proportional web cluster. In *Proc. of ACM Sigcomm workshop on Green Networking* (August 2010).

[79] Leonardi, Emilio, and Torrisi, Giovanni Luca. Least recently used caches under the shot noise model. In *Computer Communications (INFOCOM), 2015 IEEE Conference on* (2015), IEEE, pp. 2281–2289.

[80] Lin, M., Wierman, A., Andrew, L., and Thereska, E. Dynamic right-sizing for power-proportional data centers. In *Proceedings of the 30th IEEE International Conference on Computer Communications (INFOCOM)* (2011).

[81] Maggs, Bruce M, and Sitaraman, Ramesh K. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review 45*, 3 (2015), 52–66.

[82] Manfredi, Sabato, Oliviero, Francesco, and Romano, Simon Pietro. A distributed control law for load balancing in content delivery networks. *IEEE/ACM Transactions on Networking (TON) 21*, 1 (2013), 55–68.

[83] Martina, Valentina, Garetto, Michele, and Leonardi, Emilio. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM* (2014).

[84] Mathew, Vimal, Sitaraman, Ramesh K, and Shenoy, Prashant. Energy-aware load balancing in content delivery networks. In *INFOCOM, 2012 Proceedings IEEE* (2012), IEEE, pp. 954–962.

[85] Mattson, Richard L., Gecsei, Jan, Slutz, Donald R., and Traiger, Irving L. Evaluation techniques for storage hierarchies. *IBM Systems journal 9*, 2 (1970), 78–117.

[86] McCabe, John. On serial files with relocatable records. *Operations Research 13* (1965), 609–618.

[87] Megiddo, Nimrod, and Modha, Dharmendra S. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST* (2003), vol. 3, pp. 115–130.

[88] Mirrokni, Vahab, Thorup, Mikkel, and Zadimoghaddam, Morteza. Consistent hashing with bounded loads. *arXiv preprint arXiv:1608.01350* (2016).

[89] Mitzenmacher, Michael. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[90] Narayanan, Arvind, Verma, Saurabh, Ramadan, Eman, Babaie, Pariya, and Zhang, Zhi-Li. Deepcache: A deep learning based framework for content caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML* (2018), ACM, pp. 48–53.

[91] Niu, Q., Dinan, J., Lu, Q., and Sadayappan, P. Parda: A fast parallel reuse distance analysis algorithm. In *Proc. IEEE 26th Int. Parallel and Distributed Processing Symp* (May 2012), pp. 1284–1294.

[92] Nygren, Erik, Sitaraman, Ramesh K, and Sun, Jennifer. The akamai network: A platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review 44*, 3 (2010), 2–19.

[93] Olmos, Felipe, Kauffmann, Bruno, Simonian, Alain, and Carlinet, Yannick. Catalog dynamics: Impact of content publishing and perishing on the performance of a lru cache. In *Teletraffic Congress (ITC), 2014 26th International* (2014), IEEE, pp. 1–9.

[94] O'neil, Elizabeth J, O'neil, Patrick E, and Weikum, Gerhard. The lru-k page replacement algorithm for database disk buffering. *ACM SIGMOD Record 22*, 2 (1993), 297–306.

[95] Panagakis, Antonis, Vaios, Athanasios, and Stavrakakis, Ioannis. Approximate analysis of LRU in the case of short term correlations. *Computer Networks 52* (2008), 1142–1152.

[96] Park, Seon-yeong, Jung, Dawoon, Kang, Jeong-uk, Kim, Jin-soo, and Lee, Joonwon. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES* (2006), pp. 234–241.

[97] Pinheiro, Eduardo, and Bianchini, Ricardo. Energy conservation techniques for disk array-based servers. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)* (2004).

[98] Psounis, Konstantinos, Zhu, An, Prabhakar, Balaji, and Motwani, Rajeev. Modeling correlations in web traces and implications for designing replacement policies. *Computer Networks 45* (2004), 379–398.

[99] Quan, Guocong, Tan, Jian, Eryilmaz, Atilla, and Shroff, Ness. A new flexible multi-flow lru cache management paradigm for minimizing misses. *Proceedings of the ACM on Measurement and Analysis of Computing Systems 3*, 2 (2019), 39.

[100] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. 39th Annual IEEE/ACM Int. Symp. Microarchitecture (MICRO'06)* (Dec. 2006), pp. 423–432.

[101] Reese, Will. Nginx: the high-performance web server and reverse proxy. *Linux Journal 2008*, 173 (2008), 2.

[102] Rodrigues, Eliane R. The performance of the move-to-front scheme under some particular forms of Markov requests. *Journal of applied probability* (1995), 1089–1102.

[103] Saemundsson, Trausti, Bjornsson, Hjortur, Chockler, Gregory, and Vigfusson, Ymir. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.

[104] Starobinski, David, and Tse, David. Probabilistic methods for web caching. *Performance evaluation 46*, 2 (2001), 125–137.

[105] Suh, G. Edward, Devadas, Srinivas, and Rudolph, Larry. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th International Conference on Supercomputing* (New York, NY, USA, 2001), ICS '01, ACM, pp. 1–12.

[106] Suksomboon, K., et al. PopCache: Cache more or less based on content popularity for information-centric networking. LCN 2013.

[107] Sundarrajan, Aditya, Feng, Mingdong, Kasbekar, Mangesh, and Sitaraman, Ramesh. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies* (2017), ACM, pp. 55–67.

[108] Sundarrajan, Aditya, Kasbekar, Mangesh, and Sitaraman, Ramesh K. Energy-efficient disk caching for content delivery. In *Proceedings of the Seventh International Conference on Future Energy Systems* (2016), ACM, pp. 20:1–20:12.

[109] Thiebaut, Dominique, and Stone, Harold S. Footprints in the cache. *ACM Trans. Comput. Syst. 5*, 4 (Oct. 1987), 305–329.

[110] Thiébaut, Dominique, Stone, Harold S., and Wolf, Joel L. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers 41*, 6 (1992), 665–676.

[111] Tolia, Niraj, Wang, Zhikui, Marwah, Manish, Bash, Cullen, Ranganathan, Parthasarathy, and Zhu, Xiaoyun. Delivering energy proportionality with non energy-proportional systems-optimizing the ensemble. *HotPower 8* (2008), 2–2.

[112] Tsukada, Naoki, Hirade, Ryo, and Miyoshi, Naoto. Fluid limit analysis of fifo and rr caching for independent reference models. *Performance Evaluation 69*, 9 (2012), 403–412.

[113] Waldspurger, Carl A, Park, Nohhyun, Garthwaite, Alexander T, and Ahmad, Irfan. Efficient MRC construction with SHARDS. In *FAST* (2015), pp. 95–110.

[114] Wang, Xiaolin, Li, Yechen, Luo, Yingwei, Hu, Xiameng, Brock, Jacob, Ding, Chen, and Wang, Zhenlin. Optimal footprint symbiosis in shared cache. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on* (2015), IEEE, pp. 412–422.

[115] Weiser, M., Welch, B., Demers, A., and Shenker, S. Scheduling for reduced cpu energy. *Mobile Computing* (1996), 449–471.

[116] Wessels, D., and claffy, k. IETF RFC 2186: Internet Cache Protocol (ICP), version 2, 1997.

[117] Wierman, A., Andrew, L.L.H., and Tang, A. Power-aware speed scaling in processor sharing systems. In *Proceedings of the 28th IEEE International Conference on Computer Communications (INFOCOM)* (2009), IEEE, pp. 2007–2015.

[118] Wires, Jake, Ingram, Stephen, Drudi, Zachary, Harvey, Nicholas JA, Warfield, Andrew, and Data, Coho. Characterizing storage workloads with counter stacks. In *OSDI* (2014), pp. 335–349.

[119] Xiang, Xiaoya, Bao, Bin, Bai, Tongxin, Ding, Chen, and Chilimbi, Trishul. All-window profiling and composable models of cache sharing. In *ACM SIGPLAN Notices* (2011), vol. 46, ACM, pp. 91–102.

[120] Xiang, Xiaoya, Ding, Chen, Luo, Hao, and Bao, Bin. HOTL: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News* (2013), vol. 41, ACM, pp. 343–356.

[121] Ye, Ying, West, Richard, Cheng, Zhuoqun, and Li, Ye. Coloris: a dynamic cache partitioning system using page coloring. In *Parallel Architecture and Compilation Techniques (PACT), 2014 23rd International Conference on* (2014), IEEE, pp. 381–392.

[122] Young, Neal E. Online paging against adversarially biased random inputs. *Journal of Algorithms 37* (2000), 218–235.

[123] Zedlewski, John, Sobti, Sumeet, Garg, Nitin, Zheng, Fengzhou, Krishnamurthy, Arvind, and Wang, Randolph. Modeling hard-disk power consumption. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)* (2003).

[124] Zeng, Zeng, and Veeravalli, Bharadwaj. Design and performance evaluation of queue-and-rate-adjustment dynamic load balancing policies for distributed networks. *IEEE Transactions on Computers 55*, 11 (2006), 1410–1422.

[125] Zhu, Qingbo, Chen, Zhifeng, Tan, Lin, Zhou, Yuanyuan, Keeton, Kimberly, and Wilkes, John. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[126] Zhu, Qingbo, Shankar, Asim, and Zhou, Yuanyuan. Pb-lru: a self-tuning power aware storage cache replacement algorithm for conserving disk energy. In *Proceedings of the 18th annual international conference on Supercomputing* (2004), ACM, pp. 79–88.