Dissertation

Design and Analysis of Adaptive Caching Techniques for Internet Content Delivery

Thesis approved by the
Department of Computer Science
of the University of Kaiserslautern (TU Kaiserslautern)

for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Daniel S. Berger

Date of the viva	:	June 8, 2018
Dean	:	Prof. Dr. Stefan Deßloch
PhD committee		
Chair	:	Prof. Dr. Pascal Schweitzer
Reviewers	:	Prof. Dr. Jens B. Schmitt
		Prof. Dr. Florin Ciucu (University of Warwick)
		Prof. Dr. Mor Harchol-Balter (Carnegie Mellon University)

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

D 386

Abstract

Fast Internet content delivery relies on two layers of caches on the request path. Firstly, content delivery networks (CDNs) seek to answer user requests before they traverse slow Internet paths. Secondly, aggregation caches in data centers seek to answer user requests before they traverse slow backend systems. The key challenge in managing these caches is the high variability of object sizes, request patterns, and retrieval latencies. Unfortunately, most existing literature focuses on caching with low (or no) variability in object sizes and ignores the intricacies of data center subsystems.

This thesis seeks to fill this gap with three contributions. First, we design a new caching system, called AdaptSize, that is robust under high object size variability. Second, we derive a method (called Flow-Offline Optimum or FOO) to predict the optimal cache hit ratio under variable object sizes. Third, we design a new caching system, called RobinHood, that exploits variances in retrieval latencies to deliver faster responses to user requests in data centers.

The techniques proposed in this thesis significantly improve the performance of CDN and data center caches. On two production traces from one of the world's largest CDN AdaptSize achieves 30-91% higher hit ratios than widely-used production systems, and 33-46% higher hit ratios than state-of-the-art research systems. Further, AdaptSize reduces the latency by more than 30% at the median, 90-percentile and 99-percentile.

We evaluate the accuracy of our FOO analysis technique on eight different production traces spanning four major Internet companies. We find that FOO's error is at most 0.3%. Further, FOO reveals that the gap between online policies and OPT is much larger than previously thought: 27% on average, and up to 43% on web application traces.

We evaluate RobinHood with production traces from a major Internet company on a 50-server cluster. We find that RobinHood improves the 99-percentile latency by more than 50% over existing caching systems. As load imbalances grow, RobinHood's latency improvement can be more than 2x. Further, we show that RobinHood is robust against server failures and adapts to automatic scaling of backend systems.

The results of this thesis demonstrate the power of guiding the design of practical caching policies using mathematical performance models and analysis. These models are general enough to find application in other areas of caching design and future challenges in Internet content delivery.

Acknowledgements

This dissertation would not have been possible without the ongoing support, advice, and encouragement of my mentors Jens Schmitt (TU Kaiserslautern), Florin Ciucu (University of Warwick), and Mor Harchol-Balter (Carnegie Mellon University).
Thank you, Jens, for believing and guiding me across five years and six countries (Germany, Canada, Switzerland, Italy, UK, and USA). Thank you, Florin, for challenging and nurturing my mathematical side and your advice on a life in academia. Thank you, Mor, for encouraging my practical side with industry collaborations and showing me the fun side of academic everyday life.

I am grateful to all of those with whom I have had the pleasure to work during this and other related projects. My early collaborators: Martin Karsten (University of Waterloo), Ivan Martinovic (University of Oxford), and Francesco Gringoli (University of Brescia). All members of the distributed systems lab: Steffen Bondorf, Michael Beck, Paul Nikolaus, Carolina Nogueira, and Matthias Schäfer. My current collaborators: Ramesh Sitaraman (University of Massachusetts and Akamai Technologies) and Nathan Beckmann (Carnegie Mellon University).

Nobody has been more important to me in the pursuit of my dissertation than the members of my family and my circle of friends. I would like to thank my parents, whose love and guidance are with me in whatever I pursue and wherever my journey takes me. My parents are my ultimate role models.
A sincere thank you also to all my travel companions and friends: Tobias (my brother), Tanja and Sabrina Stier, Hiroaki and Aya Shioi, Helen Xu and Peter Goodman, and Simon Birnbach. Without our journeys and vacations my source of creativity would have dried up years ago.

Contents

List of Figures

List of Tables

List of Abbreviations

AGS . . . . . Aggregation servers in data centers compile webpages from many subcomponents by querying backend systems.

Akamai . . . . . Akamai Technologies Inc. – one of the earliest and largest content delivery networks (CDN).

BHR . . . . . The Byte Hit Ratio is the sum of bytes that are served by the cache divided by the total number of requested bytes.

BMR . . . . . The Byte Miss Ratio is (1-BHR), i.e., the fraction of missed bytes.

CCP . . . . . The coupon collector's problem in probability theory describes the "collect all coupons and win" contests.

CDF . . . . . The cumulative distribution function.

CDN . . . . . A Content Delivery Network consists of a geographically distributed network of caching servers that quickly answer client requests.)

Che approx. . . The Che approximation is a performance heuristic that allows deriving the cache hit ratio under the IRM.

DC . . . . . The Disk Cache is a large second-level cache in CDN servers.

FastCGI . . . . The Fast-Common-Gateway-Interface is a binary protocol for interfacing interactive programs with a web server.

Flash Crowd . . Flash crowds describe exponential spikes in website usage, e.g, when a website is mentioned in the news.

FOO . . . . . The Flow-Offline-Optimum algorithm, proposed in this thesis, calculates the optimal cache hit ratio on a given request trace.

GCCP . . . . . In the generalized CCP, the underlying probability function of coupons is unconstrained, whereas in the classical CCP, all coupons occur with equal probability.

Hill Climbing . The hill climbing technique is a popular local-search optimization technique, in which the search algorithm follows the gradient "uphill" towards an optimization objective.

HOC . . . . . The Hot Object Cache is the first-level cache in a CDN, it is a small but very fast in-memory cache.

ILP . . . . . Integer linear program.

iostat . . . . . Iostat monitors key parameters of system input/output operations such as device utilization and queue lengths.

IRM . . . . . The Independent Reference Model is a popular modeling assumption where caching requests are modeled as independent sampling from a universe of objects with a static probability distribution defined over the universe.

libcurl . . . . . The curl library is an open-source client-side HTTP requestor library.

LRU . . . . . The Least Recently-Used eviction algorithm assumes that recently-requested objects are likely to get requested again soon, and thus evict the object which has been requested the longest time in the past.

MRU . . . . . The Most Recently-Used eviction algorithm is the opposite of the LRU algorithm. MRU also denotes the head of the LRU queue, i.e., the most-recently-requested object.

Mutex . . . . . Mutual exclusion.

Nginx . . . . . The Nginx web server is a production system used to serve and cache HTTP traffic used by large websites and CDNs.)

OHR . . . . . The Object hit ratio is the number of requests that are served by the cache divided by the total number of requests.

OMR . . . . . The Object Miss Ratio is (1 - OHR), i.e., the fraction of cache misses.

OPT . . . . . An optimal solution (e.g., the optimal cache hit ratio, or an optimally-tuned parameter of an adaptive caching system.

Padé approximant The Padé approximant is the 'best' approximation of a function by a rational function of given order.

shadow caches . Micro simulation of caches to evaluate different parameters.

SIZE-OPT . . . SIZE-OPT is an offline caching system that continuously optimizes OHR with knowledge of future requests.

SSE/AVX . . . SIMD optimizations in modern Intel and AMD processors that can speed up mathematical computations..

tc-netem . . . . The traffic control network emulator is an enhancement of the Linux traffic control facilities to simulate network properties (such as loss).

Threshold . . . There are many thresholds in caching system, e.g., objects are only admitted with a size less than a certain threshold, or after they've been requested more often than a certain threshold.

TTL . . . . . A Time-To-Live mechanism limits the lifespan of an object in a cache.

Varnish . . . . . The Varnish caching system is a production HTTP caching system used by large websites and CDNs.

1

Caching and Internet content delivery

Contents

Being fast really matters... half a second delay caused a 20% drop in traffic.
— Marissa Mayer, Google employee #20, in 2008

As observed by Marissa Mayer at Google, and as repeatedly stressed by several other companies, achieving low latency is a key challenge in Internet content delivery [4–7]. Along the path that requests travel between a user and the server holding the content, there are two principal sources of high latency. First, latency grows very quickly with distance [8]. For example, the latency between a user in Australia and the servers of a content provider in the US (such as Facebook) lies in the hundreds of milliseconds [9]. Second, queries to the servers of content providers are frequently slowed down by excessive queueing of user requests in data centers and backend systems [7].

A key component in fighting this latency has been the deployment of many layers of caching along the path taken by user requests. Ideally, we can quickly respond to a user request by *delivering the requested data from a cache early on the request path* instead of traversing high-latency parts further down the path.

Figure 1.1 outlines a typical path of a user request before entering a data center. On this path, the user request typically passes through several caching layers operated by a Content delivery network (CDN) [10]. CDNs operate caching servers around the globe such that most users can be served by a nearby CDN server. For example, a large CDN such as Akamai [11, 12] operates 240,000 servers located in 1,700+ networks in 130 countries around the world

As most users can connect to a nearby CDN server, this part of the path takes only a few tens of milliseconds. Each CDN server employs two levels of caching: a small but fast in-memory cache called the Hot Object Cache (HOC) and a large second-level Disk Cache (DC).

Each requested web object is first looked up in the HOC. If the object is found in the HOC (cache hit), it can be immediately returned to the user (green path in the figure). A HOC cache hit leads to the lowest-possible request latency. The first goal of this dissertation is to *maximize the fraction of user requests served from the HOC.*

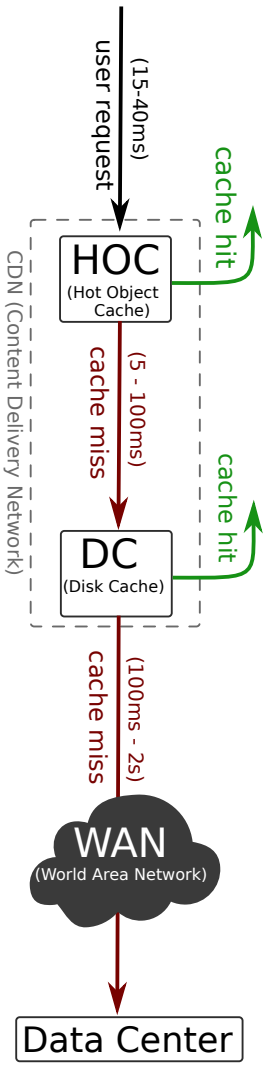


Figure 1.1: User request path (part I).

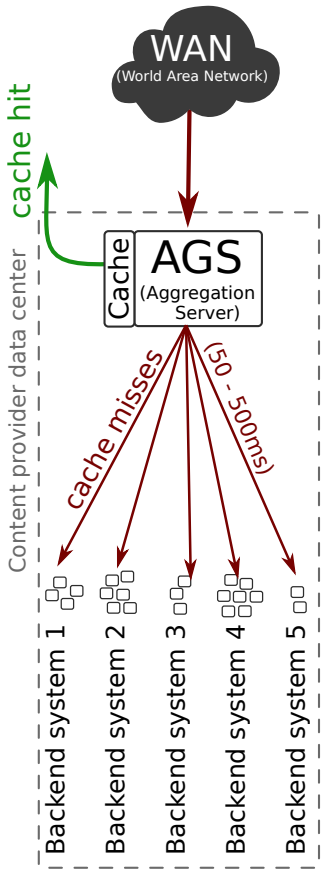


Figure 1.2: User request path (part II).

If the object is absent from the HOC (cache miss), then it is looked up in the DC. As the DC is much larger than the HOC, it is more likely to find an object there. However, the DC is also much slower than the HOC and can add a variable amount of several tens of milliseconds to the request path.

If the object is also absent in the DC, then the object is fetched over the world-area network (WAN) from the content provider's data center. Traversing the WAN and reaching the data center increases the latency by the hundreds of milliseconds.

Figure 1.2 outlines a typical path of a user request within a content provider's data center. As the request enters the data center, it is first routed to a so-called aggregation server. Most modern websites consist of many subcomponents and often include personalized content such as recommendations or advertisement. The aggregation server compiles these subcomponents into the final website and delivers it back to the user.

For each subcomponent, the aggregation server first queries its local cache (the aggregation cache). If all subcomponents can be found in the aggregation cache, the aggregation server can answer a user request within a few milliseconds.

If any subcomponent is not found in the aggregation cache, the subcomponent has to be fetched from a backend system such as a machine learning system for recommendations, or such as a database for user data. Querying a backend system is often compute-intensive and IO-intensive; this step can take several hundreds of milliseconds. The aggregation server must wait for the slowest backend query, as assembling the website requires all subcomponents to be present at the aggregation server.

Figure 1.3 shows a typical timeline for a request from the aggregation server's perspective. After the request arrives, the aggregation server processes the request and looks up all components it needs in the aggregation cache. In this case, there are three backend systems. Of two queries to the first backend system (green), one is found in the aggregation cache, the other query has to be retrieved from the backend. For the

second backend system (blue), all three queries must be retrieved from the backend; and for the third (red), two out of three queries have to be retrieved from the backend. Only once all queries to the backend systems have returned (the slowest is the third blue query), can the aggregation server continue processing the request and reply to the user. In the production systems we have analyzed, the total response time is dominated by the waiting time for queries to the backends. As the aggregation server waits for the slowest query, queries that return earlier essentially waste resources (green, red). The second goal of this dissertation is therefore to *balance the latency of backend queries* to minimize the overall response time of user requests.

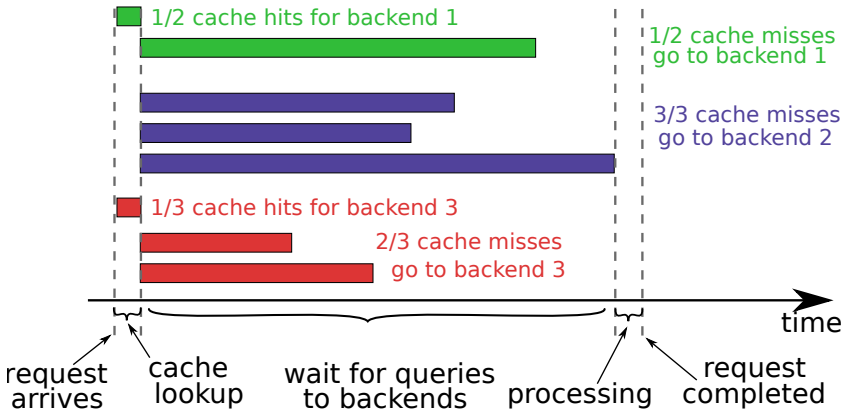


Figure 1.3: Timeline of a request from the perspective of an aggregation server in a data center at Microsoft.

High-level research question of this thesis. User requests pass through[1] three distinct caching layers: the HOC, the DC, and the aggregation server cache. As every layer adds significant latency to the request path, we seek to respond to user requests at the earliest possible layer and as quickly as possible. Unfortunately, we will see that achieving this goal is very challenging. The central research question of this thesis is: *how we can use these caches most effectively to minimize Internet request latency?*

We next introduce two key challenges that complicate the design and operation of CDN and aggregation caches.

[1]Note that not all user requests pass through a CDN and not all websites require a large data center. However, all major websites follow a variant of this design. CDNs already carry the majority of today's Internet traffic and are expected to carry almost two thirds by 2020 [13].

1.1 Challenges and Motivation

The research presented in this thesis is motivated by two classes of variability. Firstly, variability in object sizes and request patterns in CDN request streams. Seconds, variability in the retrieval latency, cacheability and request rate of aggregation cache request streams. We discuss these two types in turn.

1.1.1 At the CDN level: variability in object sizes and request patterns

CDNs serve multiple traffic classes using a *shared* server infrastructure. Such classes include web sites, videos, and interactive applications from thousands of content providers, each class with its own distinctive object size distributions and request patterns [11]. Figure 1.4 shows the object size distribution of requests served by two Akamai production servers (one in the US, the other in Hong Kong).

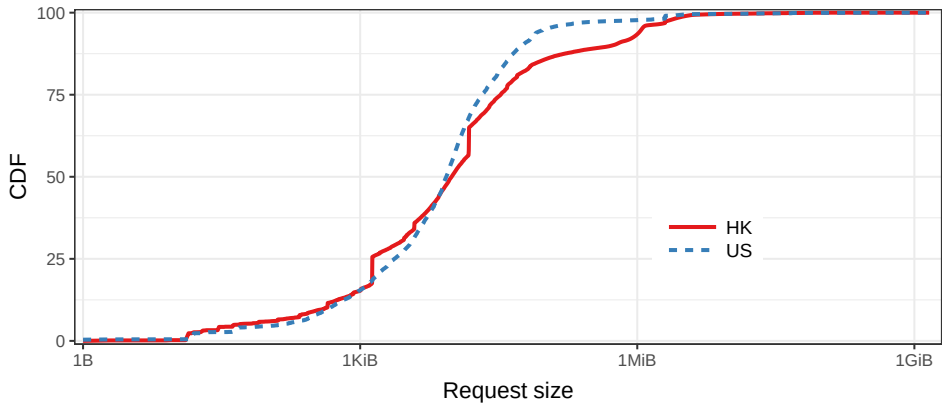


Figure 1.4: The cumulative distribution for object sizes in two Akamai production traces from Hong Kong and the US. Sizes vary by more than nine orders of magnitude.

We find that object sizes span more than nine orders of magnitude. This is particularly challenging for HOCs as their size is very small, e.g., a few GBs on the production servers we analyze in Chapter 3. In fact, HOCs are tiny when compared to the extent of size variability: CDNs have consistently observed that the largest objects are often of the same order of magnitude as the HOC size itself, even as HOC sizes and web objects have grown over the last decade.

To explain why this is challenging for a HOC, let us consider the limited choices that are open to a HOC. First, the cache can decide whether or not to admit an object (cache

admission). Second, the cache can decide which object to evict from the cache (cache eviction) if there is no space for a newly admitted object. Existing caching systems, both in academia and industry, focus on advanced *eviction policies* and typically *admit all objects* into the cache.

We use a toy example to illustrate the effect of size variability. Imagine that there are only two types of objects: 9999 small objects of size 100 KiB (say, web pages) and 1 large object of size 500 MiB (say, a software download). Further, assume that all objects are equally popular and requested forever in round-robin order. Suppose that our HOC has a capacity of 1 GiB.

A caching system which admits all objects cannot achieve an OHR above 0.5 – independently of the eviction policy. Every time the large object is requested, it pushes out ≈5000 small objects. It does not matter which objects are evicted: when the evicted objects are requested, they cannot contribute any cache hits.

This toy example is illustrative of what happens under real production traffic. We observe from Figure 1.4 that approximately 5% of objects have a size bigger than 1 MiB. Every time a cache admits a 1 MiB object, it needs to evict space equivalent to one thousand 1 KiB objects, which make up about 15% of requests. Again, those evicted objects will not be able to contribute any future cache hits.

While the academic literature on caching policies is extensive, it focuses on situations where objects are of the same size and eviction policies are sufficient (see Chapter 2).

This thesis will argue that cache admission policies warrant a much greater focus. For example, the toy example problem can be resolved using a simple size threshold. If the HOC admits only objects with a size at most 100 KiB, then it can achieve an OHR of 0.9999 as all small objects stay in the cache. Unfortunately, we will see that request patterns and size distribution in CDNs change significantly over the course of minutes, and thus static thresholds perform suboptimally. This problem is further complicated as we find that simple strategies for dynamically tuning thresholds do not work well (see Chapter 3).

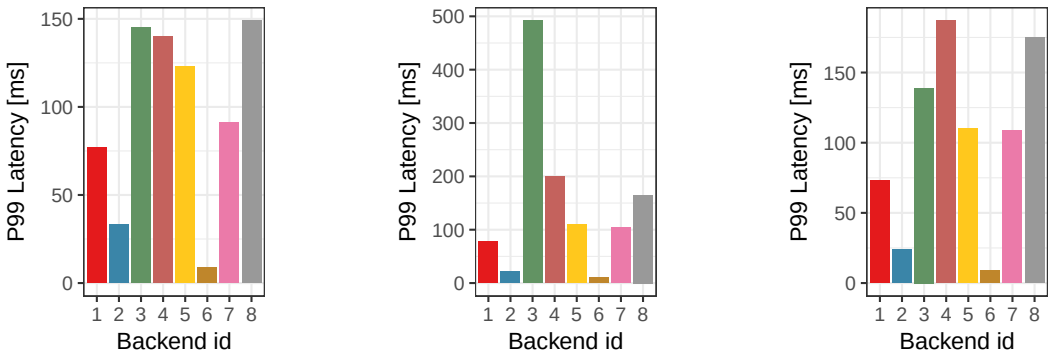
In summary, *we need a new caching system that handles high variability in object sizes and is robust to changes in the traffic mix* as they occur in daily operation, e.g., due to the CDN's global load balancer.

1.1.2 In the data center: variability in retrieval latency

Large commercial websites rely on a variety of backend systems such as advertising systems, recommender systems, databases for transactional data, and key-value stores

for product listings. To answer a user request, all these backend systems are queried for data, which is then assembled in the complete webpage. This is the architecture shown in Figure 1.2 and matches the architecture at Microsoft (see Chapter 5 for more details). Amazon uses a similar architecture [4].

A major goal in optimizing these data center architectures is to minimize the 99-th percentile of the request latency [4–7]. As the request latency is defined by the slowest query to a backend system, minimizing the 99-th percentile requires that the backend systems' latencies are approximately equal[2]. If queries to a backend service A take much longer than to the other systems, then user requests will always be bottlenecked by A.



(a) Latency at 8am (b) Latency at 11am (c) Latency at 2pm

Figure 1.5: Per-backend tail latency in a Microsoft data center at different times of a day (in February 2018). We observe that P99 latencies across different backend systems are highly variable, and the slowest backend system changes over the course of a few hours.

Figure 1.5 shows backend-query latencies from a production data center at Microsoft. The figure reveals two import properties. First, P99 latencies between different backend systems differ by an order of magnitude. For example, in Figure 1.5a, the latency of backends 3,4, and 8 are at almost 150ms whereas backend 2 is just above 50ms and backend 6 is at about 10ms. Second, the slowest backend system changes throughout the day. For example, backend 8 is the slowest system at 8am, backend 3 is the slowest system at 11am, and backend 4 is the slowest system at 2pm.

While there is much work on balancing load and on automatically scaling backend systems [15], these approaches do not resolve latency imbalance in practice. The reason

[2] As not all requests query all backends, and requests can send multiple queries to certain backends the actual goal significantly more involved, as shown in Chapter 5. We remark that in practice, some systems do not strictly wait for the slowest request, but instead present the user with lower-quality data [14]. In such a system, our goal would be to maximize the data quality of user requests.

is that it is often impossible to balance load across *different* backend systems, which use specialized software stacks and even sometimes specialized hardware. Additionally, most backend system are stateful, which makes scaling them very hard. In fact, Microsoft uses all these approaches in their data centers, and latency imbalance remains a challenge, as shown in Figure 1.5.

In summary, *we need a new way to balance latency across different backend systems* to minimize the request-level latency.

1.2 Research questions and contributions of this thesis

This thesis addresses the following three research questions motivated by the two types of variability observed in Internet content delivery systems.

1.2.1 How do we build caching systems that optimize hit ratios despite the size variability and changes in request patterns seen in CDNs?

To answer this question, we propose AdaptSize: a high-performance HOC system that is robust under highly-variable object sizes. AdaptSize's key idea is to use size-aware admission (instead of eviction) and to continuously adapt this admission policy to the request traffic. Our adaption policy is based on a stochastic approximation of the cache using a Markov model. We present a lock-free implementation of AdaptSize that integrates our model into a CDN production caching system, without limiting the inherent parallelism of such systems. We also show that AdaptSize significantly improves the hit ratio and reduces the latency of requests to the overall CDN server.

Contributions to the Internet content delivery research community. AdaptSize has raised an understanding in CDN operators that admission policies matter and need to be tuned. We have discussed AdaptSize in talks at Google, Microsoft, Facebook, and several universities. All of AdaptSize's source code is available online and has been widely studied. The CDN of the seventh-largest website in the world, wikipedia.org, has adapted a variant of AdaptSize into production use in late 2017.

1.2.2 What is the optimal hit ratio under size variability and how much further can hit ratios be improved in CDNs?

To answer this question, we propose Flow-offline optimal (FOO): a new analysis technique to derive the optimal hit ratio on a given request trace. Deriving the optimal hit ratio is NP hard, and existing approximation algorithms are both slow and highly inaccurate. FOO overcomes these limitations thanks to a new representation of optimal caching as a graph-theoretic flow problem. We prove that, under simple independence assumptions, FOO's bounds become asymptotically tight as the number of objects goes to infinity. We evaluate FOO on eight production traces from CDNs, storage systems, and data center caches and confirm that FOO's error is negligible in practice. FOO thus reveals, for the first time, the limits of caching with variable object sizes.

Contributions to the Internet content delivery research community. Several recent caching systems (following up on AdaptSize), have further improved hit ratios under variable object sizes. Many in the system community believe that these recent gains exhaust the potential for further improvement. This is motivated by the best prior bounds on optimality, which suggest that there is essentially no room for improvement. In contrast, our FOO analysis shows that current caching systems are in fact still far from optimal, suffering 11–43% more cache misses than the optimal policy. Therefore, we conclude that there is still significant room for improving hit ratios in Internet content delivery systems.

1.2.3 Can we build aggregation caches that balance latency across different backend systems to minimize request latency in data centers?

To answer this question, we propose RobinHood: a new aggregation server caching system that minimizes the request tail latency. The key observation of RobinHood is that existing cache-level metrics such as hit ratio, request rate, or latency are insufficient to decide about the allocation of cache space to backend systems. RobinHood develops a new decision metric that accurately captures the impact that each backend system has on the overall tail latency. We implement RobinHood in a cluster with 50 servers in the public cloud and show that RobinHood's metric can be retrieved online without overhead from a production system. Our evaluation with production traces from Microsoft shows

that RobinHood significantly outperforms existing resource allocation strategies and that RobinHood is robust under load changes across backend systems.

Contributions to the Internet content delivery research community. RobinHood introduces a new concept to caching research: caches cannot only be used to get fast responses for hits, but they can also be used to improve the latency of misses (by balancing load). This concept opens a new research direction in performance modeling: the intersection between queueing theory and caching analysis. A future quantitative understanding of this intersection promises a new set of tools to fight latency in data centers and beyond.

1.3 Thesis outline

The remainder of this thesis is structured as follows:

- **Chapter 2** introduces the constraints and goals of caching optimization and discusses the state of the art in caching systems and performance modeling.
- **Chapter 3** introduces AdaptSize and its tuning model, which has also been developed in several papers [16–18]. This chapter furthermore discusses our implementation and evaluation setup, and the empirical performance of AdaptSize on CDN production traces.
- **Chapter 4** introduces the FOO representation of optimal caching and proves that FOO is asymptotically correct. The FOO analysis has been published in two papers [19, 20]. This chapter furthermore discusses empirical results on the optimality of FOO and the gap between existing caching systems and the optimal cache hit ratio.
- **Chapter 5** proposes RobinHood caching as a mean to balance load across the backends of an aggregation server. The RobinHood idea and prototype have been published in an extended abstract [21] and a paper [22]. This chapter furthermore discusses our implementation and evaluation setup, and the empirical performance of RobinHood on data center production traces.
- **Chapter 6** concludes this thesis with a review and discussion of future and ongoing work.

2

Caching system design and analysis

Contents

This chapter introduces terminology and design constraints for Internet content caching systems and discusses the state of the art. Specifically, Section 2.1 discusses the design goal of building caching systems for CDNs and data centers. Section 2.2 discusses the state of the art in academic caching systems. Section 2.3 discusses the state of the art in cache performance modeling and evaluation. And, Section 2.4 discusses the state of the art in deriving optimal caching decisions and performance bounds.

2.1 Caching System Design Goals

Caches are deployed in many places throughout the Internet and in various places in computers themselves. These different deployment scenarios lead to a multitude of performance metrics and design constraints. This section introduces key notation, metrics and constraints that apply to CDN and aggregation caches.

At a high level there are three key design goals: we seek to maximize the hit ratio or minimize the tail latency, while maintaining a robust and scalable system, and avoiding adverse side-effects on second-level caches.

2.1.1 Maximizing hit ratios, minimizing miss ratios and latencies.

The classical performance metric in caching system design is the *hit ratio*, i.e., the number of requests that are served by the cache divided by the number of total requests. Conversely, the *miss ratio* is $1 - hit\ ratio$. As object sizes are variable in CDNs and aggregation caches, the hit ratio can be measured as either assigning an equal weight to every request, or as assigning a weight proportional to the size of each requested object. This leads to four metrics: OHR and OMR (for equal weight) and BHR and BMR (for weight proportional to size).

$$\textsf{Object Hit Ratio (OHR)} = \frac{\#\text{cache hits}}{\#\text{requests}}$$

$$\textsf{Object Miss Ratio (OMR)} = \frac{\#cachemisses}{\#requests}$$

$$\textsf{Byte Hit Ratio (BHR)} = \frac{\text{sum of bytes of with cache hits}}{\text{sum of bytes}}$$

$$\textsf{Byte Miss Ratio (BMR)} = \frac{\text{sum of bytes of with cache misses}}{\text{sum of bytes}}$$

CDN caches. The HOC's primary design objective is user performance, which it optimizes by providing fast responses for as many requests as possible. A natural way to measure this objective is the OHR, which gives equal weight to all user requests. Another important reason why CDNs focus on OHR in their HOCs is that HOCs are good at serving small objects, whereas, small objects are a problem for the DC. Specifically, every HOC cache miss typically requires a random read from the DC (an I/O operation), which is very slow on the spinning disks typically found in CDN deployments. Thus, improving the OHR/OMR typically leads to faster responses from the CDN server as the disk is less busy: if a HOC miss occurs, the DC's work queue is shorter. In summary, HOCs in production deployments at Akamai, Fastly [23] and Wikipedia [24], all seek to maximize the OHR, or minimize the OMR, of the HOC.

The BHR and BMR metric is less relevant to the HOC. While the much larger DC focuses on the BHR [25], the HOC has little impact on the BHR as it is typically three orders of magnitude smaller than the DC.

Aggregation caches. While hit ratio variants are an important metric for CDN performance, hit ratio is less well defined for the caches in aggregation servers in data centers. Specifically, as each request requires the results from many subqueries, full requests hits are rare as query results to all backends need to be in the cache. The key performance metric instead is the *tail latency*, which is the request latency at a high percentile such as the 99-th percentile.

2.1.2 Robustness against changing request patterns.

All caches on the Internet request path are subjected to a variety of traffic changes each day.

CDN caches. For HOCs, web content popularity changes during the day (e.g., news in the morning vs. video at night), which includes rapid changes due to flash crowds. Another source of traffic changes is the sharing of the server infrastructure between traffic classes. Such classes include web sites, videos, software downloads, and interactive applications from thousands of content providers [11]. As a shared infrastructure is more cost effective, a CDN server typically serves a mix of traffic classes. Due to load balancing decisions, this mix can change abruptly. This poses a particular challenge as each traffic class has its own distinctive request and object size distribution statistics: large objects can be unpopular within one hour and popular during the next. A HOC admission

policy must be able to rapidly adapt to all these changing request patterns to achieve consistently high OHRs.

Aggregation caches. In data centers there are many sources of variability besides retrieval latency (Subsection 1.1.2). As in CDNs, content popularity changes significantly during the day. Additionally, request rates and request composition are in constant flux. For example, at Microsoft, news see very high request rates in the morning, and involve image, text, and advertising backend systems, but typically do not involve the store catalog backend system. In the evening, xbox.com becomes very popular, which often involves the store catalog backend system and various recommender systems. Aggregation server caches must be able to rapidly adapt to all these changing request patterns to achieve consistently low tail latencies of user requests.

2.1.3 Low overhead and high concurrency.

Caches are deployed on the request path to answer user requests very quickly. Thus, HOCs and aggregation-server caches needs to both respond quickly to requests and deliver high throughput. The main bottleneck of a caching system is object lookup, the admission, and the cache eviction policies. To maintain high throughput, all three operations must have a small processing overhead, i.e., a constant time complexity per request. Additionally, almost all caching systems use multiple cores and thus caching systems must support concurrent implementations. This means that caching operations must involve as few concurrency locks (e.g., mutexes) as possible, and often aim to be lock free [26–28]. To maintain high throughput, any changes to the caching system must not interfere with this design.

CDN caches. In addition to the low overhead and high concurrency requirements, the HOC must also not impede the performance of the overall CDN server. Specifically, changes to the HOC must not negatively affect the BHR and disk utilization of the DC.

Aggregation caches. In addition to the low overhead and high concurrency requirements, aggregation caches must also not impede the performance of aggregation servers. Specifically, changes to the aggregation cache must not add significant CPU or network load.

2.2 State of the art in caching systems

This section discusses the most widely used types of caching systems and surveys the academic literature on caching systems.

2.2.1 Production caching systems.

Almost all production systems (both in CDNs and data centers) use a variant of a simple caching system. Lookups are performed using a hash map or hash tree, which can be implemented concurrently [29]. There is no admission policy (all objects get admitted into the cache). The cache evicts the least-recently-used (LRU) object.

The intuition behind the common LRU policy is, that a recently-requested object is much more likely to get requested that an object from several minutes or hours ago. LRU is also widely considered to be robust against changes in the request traffic, as it makes few assumptions on the request pattern. LRU is also easy to implement: a linked list keeps track of the recency order, where the most-recently-used (MRU) object is kept at the head, and the LRU object at the tail of the list. Whenever an object is requested, it's position is reset to the head. Whenever an object needs to be evicted, LRU picks the lists' tail.

In practice, the straightforward LRU implementation is actually very rare. The most common reason is that list-based implementations of LRU have inherent concurrent limits due to lock-competition for the head of the list [26–28]. Typical strategies include not always resetting objects to the LRU head (e.g., if they are not far from the head). Another strategy is to use a less-fine granular notion of recency which can be kept in a single lock-free ring buffer [28].

2.2.2 Academic caching systems.

The extensive body on related work on caching is surveyed in Table 2.1. We survey 33 major caching systems that have been proposed in the research literature between 1993 and 2016. We classify these systems in terms of the per-request time complexity, the eviction and admission policies used, the support for a concurrent implementation, and the evaluation method.

Not all of the 33 caching systems fulfill the low overhead design goal. Specifically, the complexity column in Table 2.1 shows that some proposals before 2002 have a computational overhead that scales logarithmically in the number of objects in the cache, which

is impractical. The caching systems discussed in this thesis differ from these systems because they have a constant complexity, and a low synchronization overhead, which we demonstrated by incorporating our proposals into production caching systems.

Of those caching systems that have a low overhead, almost none (except LRU-S and Threshold) incorporate object sizes. In particular, these systems admit and evict objects based only on recency, frequency, or a combination thereof. Our first proposal, AdaptSize, differs from these systems because it is size aware, which improves the OHR by 33-46% (as shown in Section 3.6.2).

There are only three low-overhead caching systems that are size aware. Threshold [55] uses a static size threshold, which has to be determined in advance. The corresponding Static policy in Section 3.6.3 performs poorly in our experiments. LRU-S [46] uses size-aware admission, where it admits objects with probability $1/size$. Unfortunately, this static probability is too low[3]. AdaptSize achieves a 61-78% OHR improvement over LRU-S (Figures 3.12 and 3.11). The third system [58] also uses a static parameter and was developed in parallel to AdaptSize. AdaptSize differs from these caching systems by automatically adapting the size-aware admission parameter over time.

While tuning for size-based admission is entirely new, tuning has been used in other caching contexts such as tuning for the optimal balance between recency and frequency [18, 39–41, 43, 50, 56] and for the allocation of capacity to cache partitions [30, 33, 34, 59]. In these other contexts, the most common tuning approach is hill climbing with shadow caches [30, 39–41, 43, 50, 56]. Section 3.1.3 discusses why this approach often performs poorly when tuning size-aware admission, and Section 3.6 provides corresponding experimental evidence.

Another method involves a prediction model together with a global search algorithm. The most widely used prediction model is the calculation of stack distances [60–63], which has been recently used as an alternative to shadow caches [34, 59, 59]. Unfortunately, the stack distance model is not suited to optimizing the parameters of an admission policy like in AdaptSize, since each admission parameter leads to a different request sequence and thus a different stack distance distribution that needs to be recalculated. The first caching systems proposed in this thesis, AdaptSize, introduces a new tuning model based on a Markov chain that is very different from existing tuning models.

While most of these caching systems share our goal of improving the OHR, an orthogonal line of research seeks to achieve superior throughput using concurrent cache

[3]We also tested several variants of LRU-S. We were either confronted with a cache tuning problem

implementations (compare the concurrent implementation column in Table 2.1). AdaptSize also uses a concurrent implementation and achieves throughput comparable to production systems (Section 3.6.1). AdaptSize differs from these systems by improving the OHR – without sacrificing cache throughput.

Our second caching system, RobinHood, differs from all these works, as RobinHood targets the tail latency of requests that are composed of many subqueries. We are not aware of prior literature studying this goal in the context of caching systems.

The last column in Table 2.1 shows that most recent caching systems are evaluated using prototype implementations. Likewise, we evaluate an actual implementation of AdaptSize of RobinHood through experiments in dedicated and shared data centers. We additionally use trace-driven simulations to compare to some of those systems that have only been used in simulations.

2.3 State of the art in cache performance modeling

Online policies such as LRU and its variants are studied extensively in the literature [17, 18, 32, 64–85]. A common theme in the literature is that all these models assume *unit-sized objects* and focus on the eviction policy. AdaptSize's Markov model focuses on size-aware admission and the performance under variable-sized objects.

Within the class of unit-size-object cache models, there are two major branches.

In the first branch, people have modeled the entire state of the cache, tracking all objects in the cache and their ordering in the LRU list [64–69, 72, 73, 75, 76]. Classical works have compared LRU and FIFO (First-in-first-out) and have shown convergence between FIFO and RND (random eviction): [64] uses a Markov chain of the entire cache state to model LRU and FIFO, and [65] uses an automaton model of the entire cache state [65] While these models 100% accurate, subsequent works found the solutions to be impractical when the number of objects is high, because of a combinatorial state space explosion. Subsequent work in this branch has thus derived numerical approximation methods [70, 72, 73] or relaxed the problem to asymptotic distributions [74–78, 86].

In the second branch, people start with a model that is already an approximation and do not consider the entire state space. A popular method is due to Che et al. [87] and thus often called the *Che approximation*. The essential idea in this model is to collapse the state space to two states per object: either an object is cached (IN), or it is not

with no obvious solution (Section 3.1.3), or (by removing the admission component) with an OHR similar to LRU.

(OUT). If an object in the IN state does not receive a request for a certain time, typically called the *characteristic time*, then the object is assumed to transition to OUT. If an OUT state receives a request, is transition to IN.

The intuition behind the Che approximation is that LRU works essentially as a frequency filter: objects, for which two consecutive requests are farther apart that the characteristic time, never receive a cache hit. This intuition has been supported using simulations [87], using mean-field theory analysis [88], an in asymptotic fluid-limit analysis [89]. Recent work has extended this approximation concept to LRU variants [85, 88, 90]. We remark that none of these models considers variable object sizes and size-aware admission.

2.4 State of the art in optimal caching

We define OPT as the optimal caching policy for a given cache size and a given trace, free of algorithm constraints such as the information available to the caching policy. Specifically, OPT is the offline optimal policy, which is has knowledge of the future and maximizes OHR (or minimizes OMR, equivalently).

Very little is known about how to efficiently compute OPT with variable object sizes. On the theory side, the best known approximation algorithms give weak approximation guarantees and are computationally expensive. On the practical side, system builders use offline heuristics that are much cheaper to compute, but give no guarantee that they are close to OPT. This section surveys theoretical results on OPT and offline heuristics used in practice.

2.4.1 OPT with variable object sizes is hard

While OPT is simple to compute for equal-sized objects [91, 92], computing OPT with variable object sizes is significantly harder. In fact, this problem has been recently shown to be strongly NP-complete [93], which means that no fully polynomial-time approximation scheme (FPTAS) can exist.[4]

Though caching may seem similar to Bin-Packing or Knapsack, it is quite different because the trace imposes an order on requests that constrains OPT's choices in ways that are not captured by these problems or their variants. In fact, the proof in [93] is by reduction from Vertex Cover, not Knapsack. Furthermore, unlike Bin-Packing

[4]The observation that no FPTAS can exist follows from Corollary 8.6 in [94] because OPT meets the assumptions of Theorem 8.5.

and Knapsack variants which can be approximated well for limited (small) object sizes and costs, computing OPT remains strongly NP-complete even with just three object sizes [95], and heuristics that work well on Knapsack perform badly in caching (see below).

2.4.2 Theoretical bounds on OPT

Prior work gives only three polynomial time bounds on OPT [1–3], which vary in time complexity and approximation guarantee. Table 2.2 summarizes these bounds by comparing their asymptotic run-time, how many requests can be calculated in practice (e.g., within 24 hrs), and their approximation guarantee.

Albers et al. [1] propose an LP relaxation of OPT and a rounding scheme. Unfortunately, the LP requires N^2 variables, which leads to a high $\Omega(N^{5.6})$-time complexity [96]. Not only is this running time high, but the approximation factor is logarithmic in the ratio of largest to smallest object (e.g., around 30 on production traces), making this approach impractical.

Bar et al. [2] propose a general approximation framework (which we call *LocalRatio*), which can be applied to the offline caching problem. This algorithm gives the best-known approximation guarantee, a factor of 4. Unfortunately, this is still a weak guarantee, as for miss ratio of 0.4, the offline optimal may lie anywhere between 0.1 and 0.4. Additionally, LocalRatio is a purely theoretical algorithm, with a high running time of $O(N^3)$, and which we believe had not been implemented prior to our work. Our implementation of LocalRatio can calculate up to 500 K requests in 24 hrs, which is only a small fraction of the length of production traces.

Irani proposes the OFMA approximation algorithm [3], which has $O(N^2)$ running time. This running time is small enough for our implementation of OFMA to run on small traces. Unfortunately, OFMA achieves a weak approximation guarantee, logarithmic in the cache capacity C, and in fact OFMA does badly on our traces, giving much weaker bounds than simple Belady-inspired heuristics.

Hence, prior work that considers adversarial assumptions yields only weak approximation guarantees. We therefore turn to stochastic assumptions to obtain tight bounds on the kinds of traces actually seen in practice. Under independence assumptions, FOO achieves a tight approximation guarantee on OPT, unlike prior approximation algorithms, and has asymptotically better runtime, specifically $O\big(N^{3/2}\big)$.

We are not aware of any prior stochastic analysis of offline optimal caching.

2.4.3 Heuristics used in practice to bound OPT

Since the running times of prior approximation algorithms are too high for production traces, practitioners have been forced to rely on heuristics that can be calculated more quickly. However, these heuristics only give upper bounds on OPT and there is no guarantee on how close to OPT they are.

The simplest offline upper bound is Belady's algorithm, which evicts the object whose next use lies furthest in the future. Belady is optimal in caching variants with equal-sized objects [91,92,97,98]. Even though it has no approximation guarantees for variable object sizes, it is still widely used in the systems community [36, 99–101]. However, Belady performs very badly with variable object sizes and is easily outperformed by state-of-the-art online policies.

A straightforward size-aware extension of Belady is to evict the object with the highest cost = object size × next-use distance. We call this variant *Belady-Size*. Among practitioners, Belady-Size is widely believed to perform near-optimally, but it has no guarantees. It falls short on simple examples: e.g., imagine that A is 4 MB and is referenced 10 requests hence and never referenced again, and B is 5 MB and is referenced 9 and 12 requests hence. With 5 MB of cache space, the best choice between these objects is to keep B, getting two hits. But A has cost = $4 \times 10 = 40$, and B has cost = $5 \times 9 = 45$, so Belady-Size keeps A and gets only one hit.

Alternatively, one could use Knapsack heuristics as size-aware offline upper bounds, such as the density-ordered Knapsack heuristic, which is known to perform well on Knapsack in practice [102]. We call this heuristic *Freq/Size*, as Freq/Size evicts the object with the lowest utility = frequency / size, where frequency is the number of requests to the object. Unfortunately, Freq/Size also falls short on simple examples: e.g., imagine that A is 1 MB and is referenced 10 requests hence, and B is (as before) 5 MB and is referenced 9 and 12 requests hence. With 5 MB of cache space, the best choice between these objects is to keep B, getting two hits. But A has utility = $1 \div 1 = 1$, and B has utility = $2 \div 5 = 0.4$, so Freq/Size keeps A and gets only one hit.

Though these heuristics are easy to compute and intuitive, they give no approximation guarantees. We will show that they are in fact far from OPT on real traces, and PFOO is a much better bound.

Name	Year	Overhead	Admission Policy	Eviction Policy	Concurrent	Evaluation
Cliffhanger [30]	2016	$O(1)$	none	recency	no	implementation
Billion [26]	2015	$O(1)$	none	recency	yes	implementation
BloomFilter [31]	2015	$O(1)$	frequency	recency	no	implementation
SLRU [32]	2015	$O(1)$	none	recency+frequency	no	analysis
Lama [33]	2015	$O(1)$	none	recency	no	implementation
DynaCache [34]	2015	$O(1)$	none	recency	no	implementation
MICA [27]	2014	$O(1)$	none	recency	yes	implementation
TLRU [35]	2014	$O(1)$	frequency	recency	no	simulation
MemC3 [28]	2013	$O(1)$	none	recency	yes	implementation
S4LRU [36]	2013	$O(1)$	none	recency+frequency	no	simulation
CFLRU [37]	2006	$O(1)$	none	recency+cost	no	simulation
Clock-Pro [38]	2005	$O(1)$	none	recency+frequency	yes	simulation
CAR [39]	2004	$O(1)$	none	recency+frequency	yes	simulation
ARC [40]	2003	$O(1)$	none	recency+frequency	no	simulation
LIRS [41]	2002	$O(1)$	none	recency+frequency	no	simulation
LUV [42]	2002	$O(\log n)$	none	recency+size	no	simulation
MQ [43]	2001	$O(1)$	none	recency+frequency	no	simulation
PGDS [44]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
GD* [45]	2001	$O(\log n)$	none	recency+frequency+size	no	simulation
LRU-S [46]	2001	$O(1)$	size	recency+size	no	simulation
LRV [47]	2000	$O(\log n)$	none	frequency+recency+size	no	simulation
LFU-DA [48, 49]	2000	$O(1)$	none	frequency	no	simulation
LRFU [50]	1999	$O(\log n)$	none	recency+frequency	no	simulation
PSS [51]	1999	$O(\log n)$	frequency	frequency+size	no	simulation
GDS [52]	1997	$O(\log n)$	none	recency+size	no	simulation
Hybrid [53]	1997	$O(\log n)$	none	recency+frequency+size	no	simulation
SIZE [54]	1996	$O(\log n)$	none	size	no	simulation
Hyper [54]	1996	$O(\log n)$	none	frequency+recency	no	simulation
Log2(SIZE) [55]	1995	$O(\log n)$	none	recency+size	no	simulation
LRU-MIN [55]	1995	$O(n)$	none	recency+size	no	simulation
Threshold [55]	1995	$O(1)$	size	recency	no	simulation
2Q [56]	1994	$O(1)$	frequency	recency+frequency	no	simulation
LRU-K [57]	1993	$O(\log n)$	none	recency+frequency	no	implementation

Table 2.1: Historical overview of web caching systems. While many sophisticated eviction policies combine different properties (indicated by a "+" in the eviction policy column), there are only two systems (other than AdaptSize) that use size-aware admission. The complexity column shows that systems after 2002 have a constant per-request time complexity, whereas the complexity of some older systems depends on the number (n) of cached objects. More recently, several systems introduced concurrent caching systems. The last column distinguishes between evaluation through research-based prototypes (implementation) and simulation experiments.

Technique	**Time**	**Requests / 24hrs**	**Approximation**
OPT	NP-hard [93]	<1 K	1
LP rounding	$\Omega(N^{5.6})$	50 K	$O\left(\log \frac{\max_i\{s_i\}}{\min_i\{s_i\}}\right)$
LocalRatio [2]	$O(N^3)$	500 K	4
OFMA [3]	$O(N^2)$	28 M	$O(\log C)$
FOO[a]	$O(N^{3/2})$	28 M	1
PFOO[b]	$O(N \log N)$	250 M	≈1.06

Notation: N is the trace length, C is the cache capacity, and s_i is the size of object i.

[a] FOO's approximation guarantee holds under independence assumptions.

[b] PFOO does not have an approximation guarantee but its upper and lower bounds are within 6% on average on production traces.

Table 2.2: Comparison of FOO and PFOO to prior bounds on OPT with variable object sizes. Computing OPT is NP-hard. Prior bounds [1–3] provide only weak approximation guarantees, whereas FOO's bounds are tight. PFOO performs well empirically and can be calculated for hundreds of millions of requests.

3

AdaptSize: a size-aware hot object memory cache

Contents

This chapter introduces a new caching system for the Hot Object Cache (HOC) in a CDN. The main goal of our system, AdaptSize, is to improve the OHR under the huge amounts of object size variability seen in CDNs (see Section 1.1). AdaptSize is a new size-aware admission policy that is dynamically adapted to the request traffic. The key idea of AdaptSize is to use a new Markov model that allows us to continuously pick the parameters, that optimize the OHR.

This chapter is structured as follows. Section 3.1 discusses the rationale for AdaptSize, i.e., why HOCs need size-aware admission policies, why they need to be tune the parameters of such an admission policy, and why existing tuning methods are inadequate. Section 3.2 introduces the high-level design of AdaptSize. Section 3.3 describes AdaptSize's tuning model and the underlying approximation ideas. Section 3.4 shows how AdaptSize is integrated into a production caching system. Section 3.5 discusses our experimental setup. Section 3.6 discusses the results from our empirical evaluation with production traces. And, Section 3.7 summarizes the results and ideas introduced in this chapter.

3.1 Rationale for AdaptSize

The goal of this section is to answer why the HOC needs size-aware admission, why such an admission policy needs to be adaptively tuned, and why a new approach to parameter tuning is needed.

3.1.1 Why HOCs need size-aware admission

We recall the toy example from Subsection 1.1.1. There are only two types of objects: 9999 small objects of size 100 KiB (say, web pages) and 1 large object of size 500 MiB (say, a software download). Further, assume that all objects are equally popular and requested forever in round-robin order. Suppose that our HOC has a capacity of 1 GiB.

As discussed before, a HOC that does not use admission control cannot achieve an OHR above 0.5. Every time the large object is requested, it pushes out ≈5000 small objects. It does not matter which objects are evicted: when the evicted objects are requested, they cannot contribute to the OHR.

An obvious solution for this toy example is to control admissions via a size threshold. If the HOC admits only objects with a size at most 100 KiB, then it can achieve an OHR of 0.9999 as all small objects stay in the cache.

System	Eviction Volume	
	HK trace	US trace
LRU w/o admission policy	3.4 TB	2.8 TB
Any eviction policy w/o admission policy	1.4 TB	0.9 TB
AdaptSize(Threshold)	27 GB	82 GB

Table 3.1: The eviction volume of caching systems without admission policy is very high, independent of the eviction policy. The table shows the result for LRU and a lower bound for any eviction policy (Any) based purely on the unique bytes requested in the trace. AdaptSize requires much fewer evictions, which are a magnitude lower than that of any eviction policy.

We experimentally verify the insights taken from our toy example under production traffic. Using a cache simulator (Section 3.5.2), we record the volume of evictions caused by LRU on the first 20 million requests of both of our production traces. Additionally, we record the sum of unique object sizes in the same trace section. This sum serves as a lower bound for *any* eviction policy that uses no admission policy: every unique object is admitted at least once. The results in Table 3.1 shows that on both traces, LRU evicts several TBs of objects. In comparison, the result for any eviction policy is about 3x lower, but still very large.

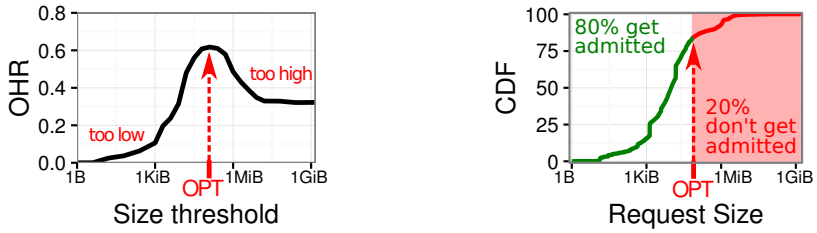
We also record the volume of evictions under a size threshold. Table 3.1 shows that on both traces, the eviction volume of a size threshold is an order of magnitude below that of any system that uses no admission policy.

Note that size-aware cache admission boosts the OHR by shielding already admitted objects from We therefore want to implement size-aware admission for production HOCs. Because high eviction numbers are triggered mainly by large objects, it is not sufficient to use a purely frequency-based admission policy.

A small cache such as the HOC therefore needs to implement size-aware admission. AdaptSize is motivated by the high variability of object sizes in CDN request traffic.

3.1.2 Why size-aware admission needs to be dynamically tuned

In contrast to much of the academic research, production systems recognize the fact that *not all* objects can be admitted into the HOC. A common approach is to define a static size threshold and to only admit objects with size below this threshold. Unfortunately, the static admission policies used in production systems perform sub-optimally for CDN workloads.



(a) OHR versus threshold. (b) Request size distribution.

Figure 3.1: Experimental results with different size thresholds. (a) A OHR-vs-threshold curve shows that the Object Hit Ratio (OHR) is highly sensitive to the size threshold, and that the optimal threshold (red arrow) can significantly improve the OHR. (b) The optimal threshold admits the requested object for only 80% of the requests.

(a) Morning: web traffic. (b) Evening: web/video mix.

Figure 3.2: The optimal size threshold changes significantly over time. (a) In the morning hours, small objects (e.g., news items) are more popular, which requires a small size threshold of a few tens of KiBs. (b) In the evening hours, web traffic gets mixed with video traffic, which requires a size threshold of a few MiBs.

Figure 3.1a shows how OHR is affected by the size threshold for a production CDN workload. While the optimal threshold (OPT) almost doubles the OHR compared to admitting all objects, conservative thresholds that are too high lead to marginal gains, and the OHR quickly drops to zero for aggressive thresholds that are too low.

Unfortunately, the "best" threshold changes significantly over time. Figures 3.2a and 3.2b show the OHR as a function of the size threshold at two different times of the day. Note that the optimal thresholds can vary by as much as two orders of magnitude during a day. Since no prior method exists for dynamically tuning such a threshold, companies have resorted to either setting the size admission threshold conservatively high, or (more commonly) not using size-aware admission at all [23, 24, 103].

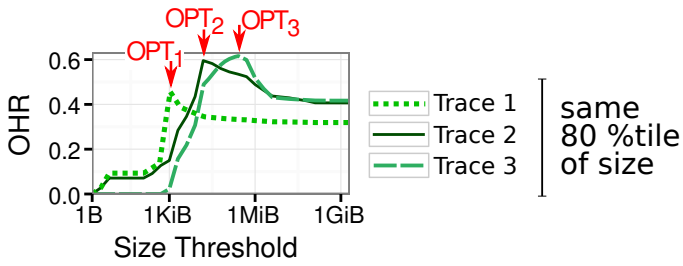


Figure 3.3: Experimental results showing that setting the size threshold to a fixed function does not work. All three traces shown here have the same 80-th size percentile, but their optimal thresholds differ by two orders of magnitude.

3.1.3 Why we need a new tuning method

We have seen that the parameter of a size-aware admission policy needs to be adapted over time. The key question when implementing size-aware admission is picking its parameters.

We explore three canonical approaches for tuning a size threshold. These approaches are well-known in prior literature and have been applied in other contexts (unrelated to the tuning of size thresholds). However, we show that these known approaches are deficient in our context, motivating the need for AdaptSize's new tuning mechanism.

Tuning based on request size percentiles. A common approach used in many contexts (e.g., capacity provisioning) is to derive the required parameter as some function of the request size distribution and arrival rate. A simple way of using this approach in our context is to set the size threshold for cache admission to be a fixed percentile of the object size distribution. However, for production CDN traces, there is no fixed relationship between the percentiles of the object size distribution and optimal size threshold that maximizes the OHR. In Figure 3.1, the optimal size threshold lands on the 80-th percentile request size. However, in Figure 3.3, note that all three traces have the same 80-th percentile but very different optimal thresholds. In fact, we found many examples of multiple traces that agree on *all* size percentiles and yet have *different* optimal size thresholds. The reason is that for maximizing OHR it matters whether the number of requests seen for a specific object size come from one (very popular) object or from many (unpopular) objects. This information is not captured by the request size distribution.

Tuning via hill climbing and shadow caches. A common tool for the tuning of caching parameters is the use of shadow caches. For example, in the seminal paper on ARC [40], the authors tune their eviction policy to have the optimal balance between

recency and frequency by using a shadow cache (see Section 2.2). A shadow cache is a simulation which is run in real time simultaneously with the main (implemented) cache, but using a different parameter value than the main cache. Hill climbing then adapts the parameter by comparing the hit ratio achieved by the shadow cache to that of the main cache (or another shadow cache). In theory, we could exploit the same idea to set our size-aware admission threshold. Unfortunately, when we tried this, we found that the OHR-vs-threshold curves are not concave and that they can have several local optima, in which the hill climbing gets frequently stuck. Figure 3.2b shows such an example, in which the local optima result from mixed traffic (web and video). Consequently, we will demonstrate experimentally in Section 3.6.3 that hill climbing is suboptimal. AdaptSize achieves an OHR that is 29% higher than hill climbing on average and 75% higher in some cases. We tried adding more shadow caches, and randomizing the evaluated parameters, but could not find a robust variant that consistently optimized the OHR across multiple traces[5].

In conclusion, our extensive experiments show that tuning methods like shadow caches with hill climbing are simply not robust enough for the problem of size-aware admission with CDN traffic.

Avoiding tuning by using probabilistic admission. One might imagine that the difficulty in tuning the size threshold lies in the fact that we are limited to a single strict threshold. The vast literature on randomized algorithm suggests that probabilistic parameters are more robust than deterministic ones [104]. We attempted to apply this idea to size-aware tuning by considering probabilistic admission policies, which "favor the smalls" by admitting them with high probability, whereas large objects are admitted with low probability. We chose a probabilistic function that is exponentially decreasing in the object size ($e^{-size/c}$). Unfortunately, the parameterization of the exponential curve (the c) matters a lot – and it's just as hard to find this c parameter as it is to find the optimal size threshold. Furthermore, the best exponential curve (the best c) changes over time. In addition to exponentially decreasing probabilities, we also tried inversely proportional (admission probability $c/size$), linear (admission probability $c - size/\max(size)$), and log-linear (admission probability $c - \log(size)/\log(\max(size))$), and several other variants. Unfortunately, none of these variants resolves the problem that there is at least one parameter without an obvious way how to choose it.

In conclusion, even randomized admission control requires the tuning of some parameter.

[5]While there are many complicated variants of shadow-cache search algorithms, they all rely on a

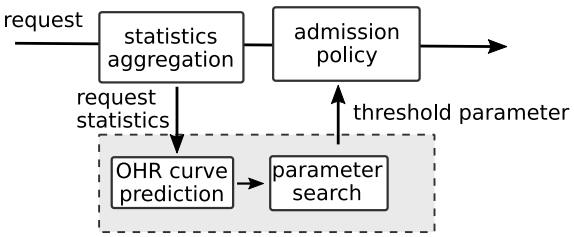


Figure 3.4: AdaptSize system overview.

3.2 High-level description of AdaptSize

AdaptSize is a HOC caching system based on a lightweight and near-optimal tuning method for size-aware cache admission.

Figure 3.4 shows a high-level description of AdaptSize. When requests enter the HOC, AdaptSize gathers aggregate statistics on the request popularity and size distribution. These statistics are fed into our model, which predicts the OHR of different parameter choices. We derive the optimal parameter from this model and enforce the parameter as an admission policy.

AdaptSize admits objects with probability $e^{-size/c}$ and evicts objects using a concurrent variant of LRU [105]. Observe that the function $e^{-size/c}$ is biased in favor of admitting small sizes with higher *probability*.

Why a probabilistic admission function? The simplest size-based admission policy is a deterministic threshold c where only objects with a size $< c$ are admitted. However, a probabilistic admission function, like $e^{-size/c}$, is more flexible: objects greater than c retain a low but non-zero admission probability, which results in eventual admission for popular objects (but not for unpopular ones). In this way, probabilistic admission functions incorporate both object size and popularity — without the need for additional data structures that keep track of object requests counts [31]. In our experiments $e^{-size/c}$ consistently achieves a 10% higher OHR than the best deterministic threshold.

What parameter c does AdaptSize use in the $e^{-size/c}$ function? AdaptSize's tuning policy recomputes the optimal c every Δ requests. A natural approach is to use hill-climbing with shadow caches to determine the optimal c parameter. Unfortunately, that leads to a myopic view in that only a *local* neighborhood of the current c can be searched. This leads to sub-optimal results, given the non-convexities present in the

fundamental assumption of stationarity, which does not need to apply to web traffic.

OHR-vs-c curve (Figure 3.2). By contrast, we derive a full Markov chain model of the cache. This model allows AdaptSize to view the entire OHR-vs-c curve and perform a *global* search for the optimal c. The challenge of the Markov model approach is in devising an algorithm for finding the solution quickly and in incorporating that algorithm into a production system.

In the following, we describe the derivation of AdaptSize's Markov model (Section 3.3), and how we incorporate AdaptSize into a production system (Section 3.4).

3.3 AdaptSize's Markov chain tuning model

To find the optimal c, AdaptSize uses a novel Markov chain model, which differs significantly from existing cache models. Traditionally, people have modeled the entire state of the cache, tracking all objects in the cache and their ordering in the LRU list [64–69, 72, 73, 75, 76]. While this is 100% accurate, it also becomes completely infeasible when the number of objects is high, because of a combinatorial state space explosion.

AdaptSize instead creates a *separate* Markov chain for each object (cf. Figure 3.5). Each object's chain tracks its position in the LRU list (if the object is in the cache), as well as a state for the possibility that the object is out of the cache. Using an individual Markov chain greatly reduces the model complexity, which now scales *linearly* with the number of objects, rather than *exponentially* in the number of objects.

3.3.1 The Markov chain approximation model.

Figure 3.5 shows the Markov chain for the i^{th} object. The chain has two important parameters. The first is the rate at which object i is moved *up* to the head of the LRU list, due to accesses to the object. We get the "move up" rate, r_i, by collecting aggregate statistics for object i during the previous Δ time interval. The second parameter is the average rate at which object i is pushed *down* the LRU list. The "pushdown" rate, μ_c, depends on the rate with which any object is moved to the top of the LRU list (due to a hit, or after cache admission). As it does not matter which object is moved to the top, μ_c is approximately the same for all objects [88]. So, we consider a single "pushdown" rate for all objects. We calculate μ_c by solving an equation that takes all objects into account, and thus captures the interactions between all the objects[6]. Specifically, we

[6]Mean-field theory [106] provides analytical justification for why it is reasonable to assume a single average pushdown rate, when there are thousands of objects (as in our case).

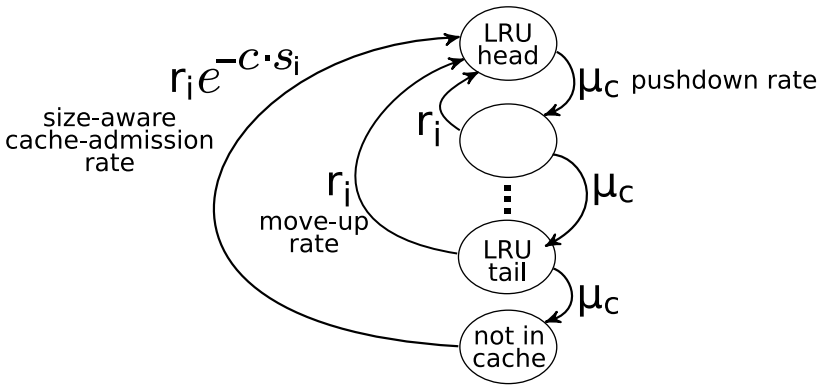


Figure 3.5: AdaptSize's Markov chain model for object i represents i's position in the LRU list and the possibility that the object is out of the cache. Each object is represented by a separate Markov chain, but all Markov chains are connected by the common "pushdown" rate μ_c. Solving these models yields the OHR as a function of c.

find μ_c by solving an equation that says that the expected size of all cached objects can't exceed the capacity K that is actually available to the cache:

$$\sum_{i=1}^{N} \mathbb{P}\left[\text{object } i \text{ in cache}\right] s_i = K \ . \tag{3.1}$$

Here, N is the number of all objects observed over the previous Δ interval, and s_i is the size of object i. Note that $\mathbb{P}\left[\text{object } i \text{ in cache}\right]$ is a monotonic function in terms of μ_c, which leads to a unique solution.

3.3.2 Deriving the OHR from the Markov chain

We seek to find $\mathbb{P}\left[\text{object } i \text{ in cache}\right]$ as a function of c by solving for the limiting probabilities of all "in" states in Figure 3.5. We first derive these limiting probabilities and then obtain the OHR as a function of c in closed form.

Limiting probabilities of "in" states.

The key challenge when solving this chain is that the length of the LRU list changes over time. We solve this by using a mathematical convergence result [89].

We consider a fixed object i, and a fixed size-aware admission parameter c. Let ℓ denote the length of the LRU list. Now the Markov chain has $\ell + 1$ states: one for each position in the list and one to represent the object is out of the cache, as shown below:

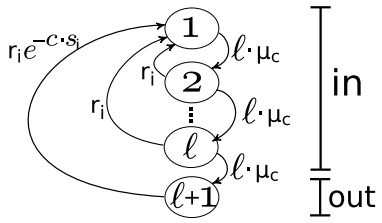


Figure 3.6: The modified Markov chain for AdaptSize with a length of the LRU list, ℓ.

Over time, ℓ changes as either larger or small objects populate the cache. However, what remains constant is the expected time for an object to get evicted (if it is not requested again) as this time only depends on the overall admission rate (i.e. the size-aware admission parameter c), which is independent of ℓ. Using this insight, we modify the Markov chain to increase the push-down rate μ_c by a factor of ℓ: now, the expected time to traverse from position 1 to $\ell + 1$ (without new requests) is constant at $1/\mu_c$.

We now solve the Markov chain for a fixed ℓ and obtain the limiting probability π_i of each position $i \in \{0, \ldots, \ell, \ell{+}1\}$. Using the π_i, we can now derive the limiting probability (as time $\to \infty$) of being "in" the cache, $\pi_{in} = \sum_{i=0}^{\ell} \pi_i$, which can be algebraically simplified to:

$$\pi_{in} = 1 - \frac{\left(\frac{\ell}{\ell + r_i/\mu_c}\right)^{\ell}}{e^{-s_i/c} + \left(\frac{\ell}{\ell + r_i/\mu_c}\right)^{\ell} - e^{-s_i/c}\left(\frac{\ell}{\ell + r_i/\mu_c}\right)^{\ell}}$$

We observe that the π_{in} quickly converges in ℓ; numerically, convergence happens around $\ell > 100$. In our simulations, the cache typically holds many more objects than 100, simultaneously. Therefore, it is reasonable to always use the converged result $\ell \to \infty$. We formally solve this limit for π_{in} and obtain the closed-form solution of the long-term probability that object i is present in the cache, as stated in Theorem 3.3.1.

The convergence (which we only verified numerically) can be formally proven in the fluid limit as objects are further and further divided into smaller parts (or, equivalently, the cache size becomes large) [89]. Specifically, [89] shows that the time it takes an object to get from position 1 to $\ell + 1$ (if there are no further requests to it) converges to a constant in a LRU cache. As AdaptSize's only difference is a thinning-out of the LRU request stream (through size-aware admission), our final result only slightly differs from the model in [89].

We obtain the following equation for the limiting probabilities.

Theorem 3.3.1

$$\mathbb{P}\left[\textit{object } i \textit{ in cache}\right] = \frac{(e^{r_i/\mu_c} - 1) \cdot e^{-c \cdot s_i}}{1 + (e^{r_i/\mu_c} - 1) \cdot e^{-c \cdot s_i}}$$

Note that the size admission parameter c affects both the admission probability ($e^{-s_i/c}$) and the pushdown rate (μ_c). For example, a lower c results in fewer admissions, which results in fewer evictions, and in a smaller pushdown rate.

The OHR as a function of c. Theorem 3.3.1 and Equation (3.1) yield the OHR by observing that the expected number of hits of object i equals r_i (i's average request rate) times the long-term probability that i is in the cache. The OHR predicted for the threshold parameter c is then simply the ratio of expected hits to requests:

$$OHR(c) = \frac{\sum_{i=1}^{N} r_i \, \mathbb{P}\left[\text{object } i \text{ in cache}\right]}{\sum_{i=1}^{N} r_i} \quad .$$

If we consider a discretized range of c values, we can now compute the OHR for each c in the range which gives us a "curve" of OHR-vs-c (similar to the curves in Figure 3.7).

3.3.3 Accuracy of AdaptSize's model.

Our Markov chain relies on several simplifying assumptions that can potentially impact the accuracy of the OHR predictions. Figure 3.7 shows that AdaptSize's OHR equation matches experimental results across the whole range of the threshold parameter c on two typical traces of length Δ. In addition, we continuously compared AdaptSize's model to measurements during our experiments (Section 3.6). AdaptSize is very accurate with an average error of about 1%.

3.4 Implementation and integration with a production system

We implemented AdaptSize on top of Varnish [107, 108], a production caching system, by modifying the miss request path. On a cache miss, Varnish accesses the second-level cache to retrieve the object, and places it in its HOC. With AdaptSize, the probabilistic admission decision is executed, which is evaluated independently for all cache threads

(a) HK trace (b) US trace

Figure 3.7: AdaptSize's Markov model predicts the OHR sensitivity curve (red solid line). This is very accurate when compared to the actual OHR (black dots) that results when that threshold is chosen. Each experiment involves a portion of the production trace of length $\Delta = 250K$.

and adds a constant number of instructions to the request path. If the object is not admitted, it is served from Varnish's transient memory.

Our implementation uses a parameter Δ which is the size of the window of requests over which our Markov model for tuning is computed. In addition to statistics from the current window, we also incorporate the statistical history from prior windows via exponential smoothing, which makes AdaptSize more robust and largely insensitive to Δ on both of our production traces. In our experiments, we choose Δ=250K requests (about 5-10 mins on average), which allows AdaptSize to react quickly to changes in the request traffic.

We describe three key implementation challenges: how to gather statistics in a concurrent implementation, how to efficiently implement our math model, and how to use our math model to find the optimal c.

3.4.1 Lock-free statistics collection.

A key problem in implementing AdaptSize lies in efficient statistics collection for the tuning model. Gathering request statistics can add significant overhead to concurrent caching designs [59]. Varnish and AdaptSize use thousands of threads in our experiments, so centralized request counters would cause high lock contention. In fact, we find that Varnish's throughput bottleneck is lock contention for the few remaining synchronization points (e.g., [105]).

Instead of a central request counter, AdaptSize hooks into the internal data structure of the cache threads. Each cache thread keeps debugging information in a concurrent ring buffer, to which all events are simply appended (overwriting old events after some time). AdaptSize's statistics collection frequently scans this ring buffer (read only) and does not require any synchronization.

3.4.2 Robust and efficient model evaluation.

The OHR prediction in our statistical model involves two more implementation challenges. The first challenge lies in efficiently solving equation (3.1). We achieve a constant time overhead by using a fixed-point solver [109]. The second challenge is due to the exponential function in the Theorem 3.3.1. The value of the exponential function outgrows even 128-bit float number representations. We solve this problem by using an accurate and efficient approximation for the exponential function using a Padé approximant [110] that only uses simple float operations which are compatible with SSE/AVX vectorization, speeding up the model evaluation by about 10-50× in our experiments.

3.4.3 Global search for the optimal c.

Once we have the input data and the model, we are capable of producing an OHR-vs-c plot within each Δ interval. To search for the optimal c, we use a systematic sampling of the search space combined with a local search method (as suggested in [111]). The systematic sampling uses logarithmic step sizes (1B-2B, 2B-4B, etc.) and starts in parallel from the smallest (c =1B) and largest threshold parameter (c =cache capacity). The local search method is a text book approach, golden section search, with the default parameters [112].

At the end of the parameter search step, we have found the threshold parameter that maximizes the OHR for each Δ interval.

3.5 Evaluation Methodology

We evaluate AdaptSize using *both* trace-based simulations (Section 3.5.2) and a Varnish-based implementation (Section 3.5.3) running on our experimental testbed. For both these approaches, the request load is derived from traces from Akamai's production CDN servers (Section 3.5.1).

3.5.1 Production CDN request traces

We collected request traces from two production CDN servers in Akamai's global network. Table 3.2 summarizes the main characteristics of the two traces. Our first trace is from urban Hong Kong (**HK trace**). Our second trace is from rural Tennessee, in the US, (**US trace**). Both span multiple consecutive days, with over 440 million requests

	HK trace	US trace
Total Requests	450 million	440 million
Total Bytes	157.5 TiB	152.3 TiB
Unique Objects	25 million	55 million
Unique Bytes	14.7 TiB	8.9 TiB
Start Date	Jan 29, 2015	Jul 15, 2015
End Date	Feb 06, 2015	Jul 20, 2015

Table 3.2: Basic information about our web traces.

per trace during the months of February and July 2015. Both production servers use a HOC of size 1.2 GiB and several hard disks as second-level caches. They serve a traffic mix of several thousand popular web sites, which represents a typical cross section of the web (news, social networks, downloads, ecommerce, etc.) with highly variable object sizes. Some content providers split very large objects (e.g., videos) into smaller (e.g., 2 MiB) chunks. The chunking approach is accurately represented in our request traces. For example, the cumulative distribution function shown in Figure 1.4 shows a noticeable jump around the popular 2 MiB chunk size.

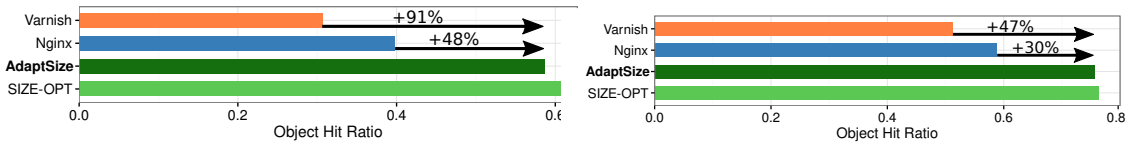
3.5.2 Trace-based simulator

We implemented a cache simulator in C++ that incorporates AdaptSize and several state-of-the-art research caching policies. The simulator is a single-threaded implementation of the admission and eviction policies and performs the appropriate cache actions when it is fed the CDN request traces. Objects are only stored via their ids and the HOC size is enforced by a simple check on the sum of bytes currently stored. While actual caching systems (such as Varnish [105,113]) use multi-threaded concurrent implementations, our single-threaded simulator provides a good approximation of the OHR when compared with our prototype implementations that we describe next.

3.5.3 Prototype Evaluation Testbed

Our implementation testbed is a dedicated (university) data center consisting of a client server, an origin server, and a CDN server that incorporates the HOC. We use FUJITSU CX250 HPC servers, which run RHEL 6.5, kernel 2.6.32 and gcc 4.4.7 on two Intel E5-2670 CPUs with 32 GiB RAM and an IB QDR networking interface.

[7] We refer to AdaptSize incorporated into Varnish as "AdaptSize" and Varnish without modifications as "Varnish".



(a) HK. (b) HK.

Figure 3.8: Comparison of AdaptSize's implementation to the Varnish and Nginx production systems and SIZE-OPT. (a) On the HK trace, AdaptSize improves the OHR by 48-91% over the production systems and achieves 95% of the OHR of SIZE-OPT. (b) On the US trace, AdaptSize improves the OHR by 30-47% over the production systems and achieves 99% of the OHR of SIZE-OPT.

In our evaluation, the HOC on our CDN server is either running Nginx, Varnish, or AdaptSize. Recall that we implemented AdaptSize by adding it to Varnish[7] as described in Section 3.4. We use Nginx 1.9.12 (February 2016) with its build-in frequency-based admission policy. This policy relies on one parameter: how many requests need to be seen for an object before being admitted to the cache. We use an optimized version of Nginx, since we have tuned its parameter offline for both traces. We use Varnish 4.1.2 (March 2016) with its default configuration that does not use an admission policy.

The experiments in Section 3.6.1, 3.6.2, and 3.6.3 focus on the HOC and do not use a DC. The DC in Section 3.6.1 uses Varnish in a configuration similar to that of the Wikimedia Foundation's CDN [24]. We use four equal dedicated 1 TB WD-RE3 7200 RPM 32 MiB-Cache hard disks attached via a Dell 6 Gb/s SAS Host Bus Adapter Card in raw mode (RAID disabled).

The client fetches content specified in the request trace from the CDN server using libcurl. The request trace is continuously read into a global queue, which is distributed to worker threads (client threads). Each client thread continually requests objects in a closed-loop fashion. We use up to 200 such threads and verified that the number of client threads has a negligible impact on the OHR.

If the CDN server does not have the requested content, it is fetched from the origin server. Our origin server is implemented in FastCGI. As it is infeasible to store all trace objects (23 TB total) on the origin server, our implementation creates objects with the correct size on the fly before sending them over the network. In order to stress test our caching implementation, the origin server is highly multi-threaded and intentionally never the bottleneck.

3.6 Empirical Evaluation

This section presents our empirical evaluation of AdaptSize. We divide our evaluation into three parts. In Section 3.6.1, we compare AdaptSize with production caching systems, as well as with an *offline* caching system called SIZE-OPT that continuously optimizes OHR with knowledge of future requests. While SIZE-OPT is not implementable in practice. it provides an upper bound on the achievable OHR to which AdaptSize can be compared. In Section 3.6.2, we compare AdaptSize with research caching systems that use more elaborate eviction and admission policies. In Section 3.6.3, we evaluate the robustness of AdaptSize by emulating both randomized and adversarial traffic mix changes. In Section 3.6.4, we evaluate the side-effects of AdaptSize on the overall CDN server.

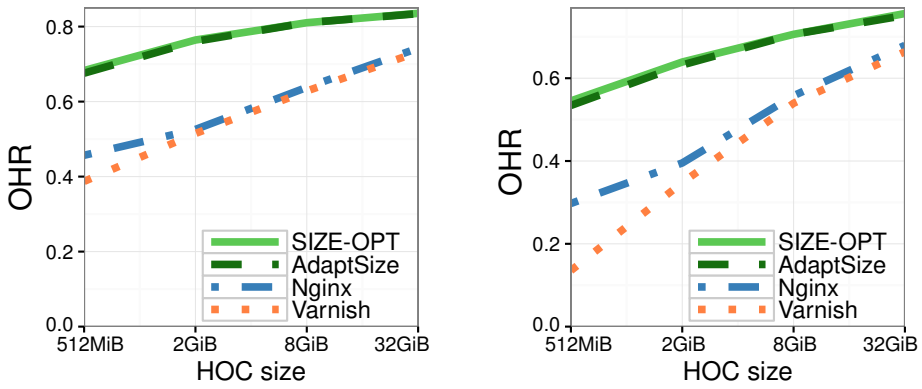
3.6.1 Comparison with production systems

We use our experimental testbed outlined in Section 3.5.3 and answer four basic questions about AdaptSize.

What is AdaptSize's OHR improvement over production systems? Quick answer: *AdaptSize improves the OHR by 47-91% over Varnish and by 30-48% over Nginx*. We compare the OHR of AdaptSize to Nginx and Varnish using the 1.2 GiB HOC configuration from the corresponding Akamai production servers (Section 3.5.1). For the HK trace (Figure 3.8b), we find that AdaptSize improves over Nginx by 30% and over Varnish by 47%. For the US trace (Figure 3.8a), the improvement increases to 48% over Nginx and 91% over Varnish.

The difference in the improvement over the two traces stems from the fact that the US trace contains 55 million unique objects as compared to only 25 million unique objects in the HK trace. We further find that AdaptSize improves the OHR variability (the coefficient of variation) by 1.9× on the HK trace and by 3.8× on the US trace (compared to Nginx and Varnish).

How does AdaptSize compare with SIZE-OPT? Quick answer: *for the typical HOC size, AdaptSize achieves an OHR within 95% of SIZE-OPT*. We benchmark AdaptSize against the SIZE-OPT policy, which tunes the threshold parameter c using a priori knowledge of the next one million requests. Figures 3.8a and 3.8b show that AdaptSize is within 95% of SIZE-OPT on the US trace, and within 99% of SIZE-OPT on the HK trace, respectively.



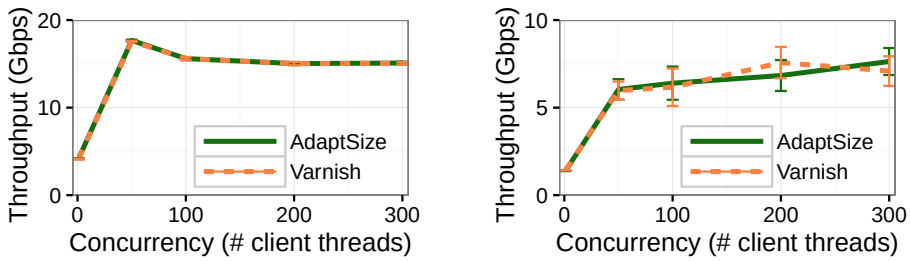
(a) HK trace (b) US trace

Figure 3.9: Comparison of AdaptSize to SIZE-OPT, Varnish, and Nginx when scaling the HOC size under the production server traffic of two 1.2 GiB HOCs. AdaptSize always stays close to SIZE-OPT and significantly improves the OHR for all HOC sizes.

How much is AdaptSize's performance affected by the HOC size? Quick answer: *AdaptSize's improvement over production caching systems becomes greater for smaller HOC sizes and decreases for larger HOC sizes.* We consider the OHR when scaling the HOC size between 512 MiB and 32 GiB under the production server traffic of a 1.2 GiB HOC. Figures 3.9a and 3.9b shows that the performance of AdaptSize is close to SIZE-OPT for all HOC sizes. The improvement of AdaptSize over Nginx and Varnish is most pronounced for HOC sizes close to the original configuration. As the HOC size increases, the OHR of all caching systems improves, since the HOC can store more objects. This leads to a smaller relative improvement of AdaptSize for a HOC size of 32 GiB: 10-12% over Nginx and 13-14% over Varnish.

How much is AdaptSize's performance affected when jointly scaling up HOC size and traffic rate? Quick answer: *AdaptSize's improvement over production caching systems remains constant for larger HOC sizes.* We consider the OHR when jointly scaling the HOC size and the traffic rate by up 128x (153 GiB HOC size). This is done by splitting a production trace into 128 non-overlapping segments and replaying all 128 segments concurrently. We find that the OHR remains approximately constant as we scale up the system, and that AdaptSize achieves similar OHR improvements as under the original 1.2 GiB HOC configuration.

What about AdaptSize's overhead? Quick answer: *AdaptSize's throughput is comparable to existing production systems and AdaptSize's memory overhead is reasonably small.* AdaptSize is built on top of Varnish, which focuses on high concurrency and



(a) 100% OHR scenario.

(b) 0% OHR scenario.

Figure 3.10: Comparison of the throughput of AdaptSize and Varnish in micro experiments with (a) 100% OHR and (b) 0% OHR. Scenario (a) stress tests the hit request path and shows that there is no difference between AdaptSize and Varnish. Scenario (b) stress tests the miss request path (every request requires an admission decision) and shows that the throughput of AdaptSize and Varnish is very close (within confidence intervals).

simplicity. In Figure 3.10, we compare the throughput (bytes per second of satisfied requests) of AdaptSize to an unmodified Varnish system. We use two micro experiments. The first benchmarks the hit request path (100% OHR scenario), to verify that there is indeed no overhead for cache hits (see section 3.4). The second benchmarks the miss request path (0% OHR scenario), to assess the worst-case overhead due to the admission decision.

We replay one million requests and configure different concurrency levels via the number of client threads. Note that a client thread does not represent an individual user (Section 3.5.3). The results are based on 50 repetitions.

Figure 3.10a shows that the application throughput of AdaptSize and Varnish are indistinguishable in the 100% OHR scenario. Both systems achieve a peak throughput of 17.5 Gb/s for 50 clients' threads. Due to lock contention, the throughput of both systems decreases to around 15 Gb/s for 100-300 clients' threads. Figure 3.10b shows that the application throughput of both systems in the 0% OHR scenario is very close, and always within the 95% confidence interval.

The memory overhead of AdaptSize is small. The memory overhead comes from the request statistics needed for AdaptSize's tuning model. Each entry in this list describes one object (size, request count, hash), which requires less than 40 bytes. The maximum length of this list, across all experiments, is 1.5 million objects (58 MiB), which also agrees with the memory high water mark (VmHWM) reported by the Kernel for AdaptSize's tuning process.

3.6.2 Comparison with research systems

We have seen that AdaptSize performs very well against production systems. We now ask the following.

How does AdaptSize compare with research caching systems, which involve more sophisticated admission and eviction policies? Quick answer: *AdaptSize improves by 33-46% over state-of-the-art research caching system.* We use the simulation evaluation setup explained in Section 3.5.2 with eight systems from Table 2.1, which are selected with the criteria of having an efficient constant-time implementation. Four of the eight systems use a recency and frequency trade-off with fixed weights between recency and frequency. Another three systems (ending with "++") use sophisticated recency and frequency trade-offs with variable weights, which we hand-tuned to our traces to create optimistic variants[8]. The remaining system is LRU-S [46], which uses size-aware eviction and admission with static parameters.

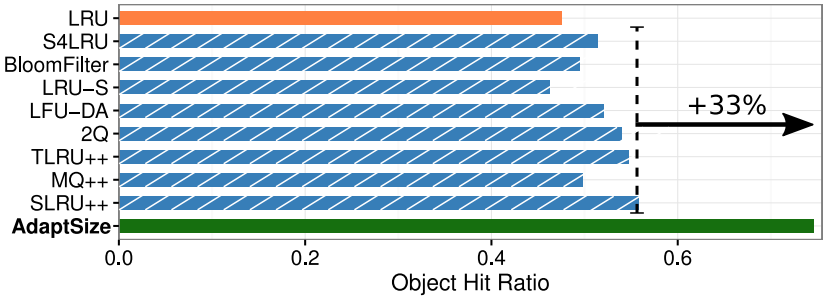


Figure 3.11: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these are sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). LRU-S is the only system – besides AdaptSize – that incorporates size. AdaptSize improves the OHR by 33% over the next best system. Policies annotated by "++" are optimistic, because we offline-tuned their parameters to the trace. These results are for the HK trace; corresponding results for the US trace are shown in Figure 3.12.

Figure 3.11 shows the simulation results for a HOC of size 1.2 GiB on the HK trace. We find that AdaptSize achieves a 33% higher OHR than the second best system, which is SLRU++. Figure 3.12 shows the simulation results for the US trace. AdaptSize achieves a 46% higher OHR than the second best system, which is again SLRU++. Note that SLRU's performance heavily relies on offline parameters as can be seen by

[8]There are self-tuning variants of recency-frequency trade-offs such as ARC [40]. Unfortunately, we could not test ARC itself, because its learning rule relies on the assumption of unit-sized object sizes.

Figure 3.12: Comparison of AdaptSize to state-of-the-art research caching systems. Most of these use sophisticated admission and eviction policies that combine recency and frequency (striped blue bars). AdaptSize improves the OHR by 46% over the next best system. Policies annotated by "++" are optimistic, because we offline-tuned their parameters to the trace. These results are for the US trace and a HOC size 1.2 GiB.

the much smaller OHR of S4LRU, which is a static-parameter variant of SLRU. In contrast, AdaptSize achieves its superior performance without needing offline parameter optimization. In conclusion, we find that AdaptSize's policies outperform sophisticated eviction and admission policies, which do not depend on the object size.

3.6.3 Robustness of alternative tuning methods

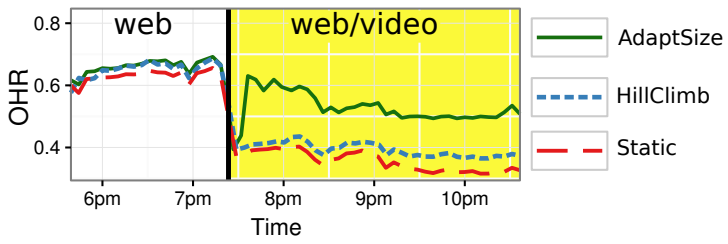


Figure 3.13: Comparison of AdaptSize, threshold tuning via hill climbing and shadow caches (HillClimb), and a static size threshold (Static) under a traffic mix change from only web to mixed web/video traffic. While AdaptSize quickly adapts to the new traffic mix, HillClimb gets stuck in a suboptimal configuration, and Static (by definition) does not adapt. AdaptSize improves the OHR by 20% over HillClimb and by 25% over Static on this trace.

So far we have seen that AdaptSize significantly improves the OHR over caching systems without size-aware admission, including production caching systems (Section 3.6.1) and research caching systems (Section 3.6.2). We now focus on different cache tuning methods for the size-aware admission parameter c (see the beginning of Section 3.2).

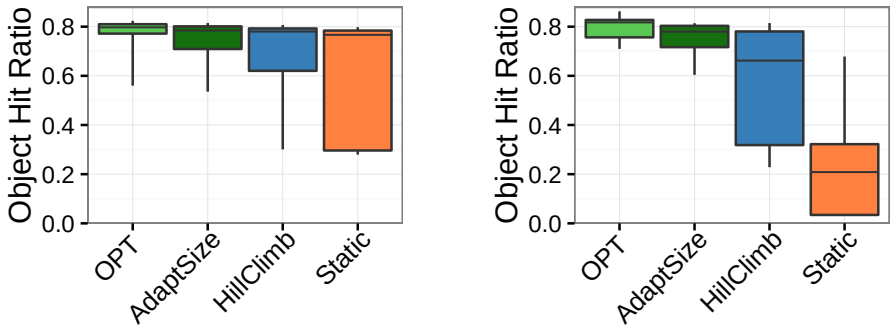
Specifically, we compare AdaptSize with hill climbing (**HillClimb**), based on shadow caches (cf. Section 3.1). HillClimb uses two shadow caches and we hand-optimized its parameters (interval of climbing steps, step size) on our production traces. We also compare to a static size threshold (**Static**), where the value of this static threshold is offline optimized on our production traces. We also compare to SIZE-OPT, which tunes c based on offline knowledge of the next one million requests. All four policies are implemented on Varnish using the setup explained in Section 3.5.3.

We consider two scenarios: 1) *randomized traffic mix changes* and 2) *adversarial traffic mix changes*. A randomized traffic mix change involves a random selection of objects which abruptly become very popular (similar to a flash crowd event). An adversarial traffic mix change involves frequently changing the traffic mix between classes that require vastly different size-aware admission parameters (e.g., web, video, or download traffic). An example of an adversarial change is the case where objects larger than the previously-optimal threshold suddenly become very popular, as shown in Figure 3.13.

Is AdaptSize robust against randomized traffic mix changes? Quick answer: *AdaptSize performs within 95% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 47-54% of SIZE-OPT's OHR.* We create 50 different randomized traffic mix changes. Each experiment consists of two parts. The first part is five million requests long and allows each tuning method to converge to a stable configuration. The second part is ten million requests long and consists of 50% production-trace requests and 50% of very popular objects. The very popular objects consist of a random number of objects (between 200 and 1000), which are randomly sampled from the trace.

Figure 3.14a shows a boxplot of the OHR for each caching tuning method across the 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 9% on average, and by more than 75% in five of the 50 experiments. AdaptSize improves the OHR over Static across every percentile, by 30% on average, and by more than 100% in five of the 50 experiments. Compared to SIZE-OPT, AdaptSize achieves 95% of the OHR for all percentiles.

Is AdaptSize robust against adversarial traffic mix changes? Quick answer: *AdaptSize performs within 81% of SIZE-OPT's OHR even for the worst 5% of experiments, whereas HillClimb and Static achieve only 5-15% of SIZE-OPT's OHR.* Our experiment consists of 25 traffic mix changes. Each traffic mix is three million requests



(a) Randomized. (b) Adversarial.

Figure 3.14: Comparison of cache tuning methods under traffic mix changes. We performed 50 randomized traffic mix changes (a), and 25 adversarial traffic mix changes (b). The boxes show the range of OHR from the 25-th to the 75-th percentile among the 25-50 experiments. The whiskers show the 5-th to the 95-th percentile.

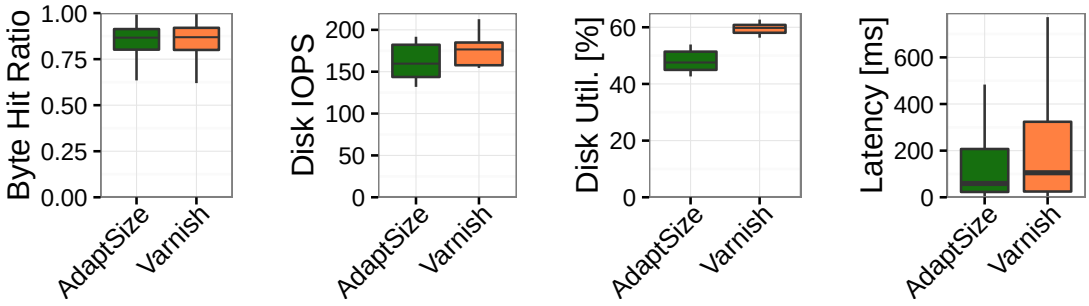
long, and the optimal c parameter changes from 32-256 KiB to 1-2 MiB, then to 16-32 MiB, and back again.

Figure 3.14b shows a boxplot of the OHR for each caching tuning method across all 50 experiments. The boxes indicate the 25-th and 75-th percentile, the whiskers indicate the 5-th and 95-th percentile. AdaptSize improves the OHR over HillClimb across every percentile, by 29% on average, and by more than 75% in seven of the 25 experiments. AdaptSize improves the OHR over Static across every percentile, by almost 3x on average, and by more than 10x in eleven of the 25 experiments. Compared to SIZE-OPT, AdaptSize achieves 81% of the OHR for all percentiles.

3.6.4 Side effects of Size-Aware Admission

So far, our evaluation has focused on AdaptSize's improvement with regard to the OHR. We evaluate AdaptSize's side-effects on the DC and on the client's request latency (cf. Section 2.1). Specifically, we compare AdaptSize to an unmodified Varnish system using the setup explained in Section 3.5.3. Network latencies are emulated using the Linux kernel (tc-netem). We set a 30ms round-trip latency between client and CDN server, and 100ms round-trip latency between CDN server and origin server. We answer the following three questions on the CDN server's performance.

How much does AdaptSize affect the BHR of the DC? Quick answer: *AdaptSize has a neutral effect on the BHR of the DC.* The DC's goal is to maximize the BHR,



(a) BHR. (b) IOPS. (c) Utilization. (d) Latency.

Figure 3.15: Evaluation of AdaptSize's side effects across ten different sections of the US trace. AdaptSize has a neutral impact on the byte hit ratio and leads to a 10% reduction in the median number of I/O operations going to the disk, and a 20% reduction in disk utilization.

which is achieved by a very large DC capacity [31]. In fact, compared to the DC the HOC has less than on thousandth the capacity. Therefore, changes to the HOC have little effect on the DC's BHR.

In our experiment, we measure the DC's byte hit ratio (BHR) from the origin server. Figure 3.15a shows that there is no noticeable difference between the BHR under AdaptSize and under an unmodified Varnish.

Does AdaptSize increase the load of the DC's hard disks? Quick answer: *No. In fact, AdaptSize reduces the average disk utilization by 20%.* With AdaptSize, the HOC admits fewer large objects, but caches many more small objects. The DC's request traffic therefore consists of more requests to large objects, and significantly fewer requests to small objects.

We measure the request size distribution at the DC and report the corresponding histogram in Figure 3.16. We observe that AdaptSize decreases the number of cache misses significantly for all object sizes below 256 KiB. For object sizes above 256 KiB, we observe a slight increase in the number of cache misses. Overall, we find that the DC has to serve 60% fewer requests with AdaptSize, but that the disks have to transfer a 30% higher byte volume. The average request size is also 4x larger with AdaptSize, which improves the sequentiality of disk access and thus makes the DC's disks more efficient.

To quantify the performance impact on the DC's hard disks we use iostat [114]. Figure 3.15b shows that the average rate of I/O operations per second decreases by about 10%. Moreover, Figure 3.15c shows that AdaptSize reduces the disk's utilization (the fraction of time with busy periods) by more than 20%. We conclude that the increase in

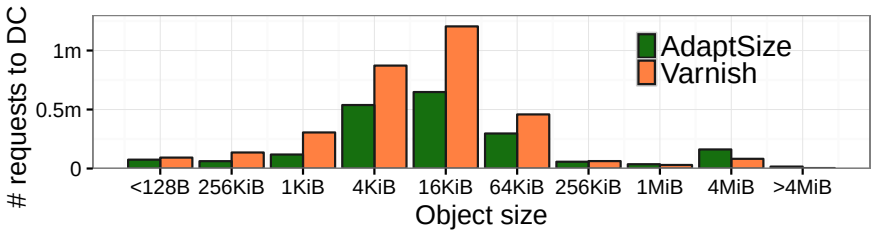


Figure 3.16: Comparison of the distribution of request sizes to the disk cache under a HOC running AdaptSize versus unmodified Varnish. All object sizes below 256 KiB are significantly less frequent under AdaptSize, whereas larger objects are slightly more frequent.

byte volume is more than offset by the fact that AdaptSize shields the DC from many small requests and improves the sequentiality of requests served by the DC.

How much does AdaptSize reduce the request latency? Quick answer: *AdaptSize reduces the request latency across all percentiles by at least 30%.*

We measure the end-to-end request latency (time until completion of a request) from the client server. Figure 3.15d shows that AdaptSize reduces the median request latency by 43%, which is mostly achieved by the fast HOC answering a higher fraction of requests. The figure also shows significant reduction of tail latency, e.g., the 90-th and 99-th latency percentiles are reduced by more than 30%. This reduction in the tail latency is due to the DC's improved utilization factor, which leads to a much smaller number of outstanding requests, which makes it easier to absorb traffic bursts.

3.7 Summary

AdaptSize is a new caching system for the hot object cache in CDN servers. The power of AdaptSize stems from a size-aware admission policy that is continuously optimized using a new Markov model of the HOC. In experiments with Akamai production traces, we show that AdaptSize vastly improves the OHR over both state-of-the-art production systems and research systems. We also show that our implementation of AdaptSize is robust and scalable and improves the DC's disk utilization.

As more diverse applications with richer content migrate onto the Internet, future CDNs will experience even greater variability in request patterns and object sizes. We believe that AdaptSize and its underlying mathematical model will be valuable in addressing this challenge.

To summarize, AdaptSize addresses the concerns in [28] in making the point that it is possible to maximize the OHR with a sophisticated caching policy in a high throughput caching system.

4

FOO: Analysis of optimal caching under variable object sizes

Contents

In Chapter 3 we have seen that the OHR of CDN caching system can be significantly improved in practice. This chapter asks: *how much further can we improve hit ratios? Should the systems community continue trying to improve miss ratios, or have all achievable gains been exhausted?*

To answer this question, one would like to know the best achievable hit ratio, free of constraints—i.e., the offline optimal (OPT). In computer science theory community, there are a few works that study OPT's miss ratio (the OMR, see Section 2.1). Consequently, our performance metric throughout this chapter is the OMR.

Unfortunately, very little is known about OPT with variable object sizes (see Section 2.4). For equal-sized objects, computing OPT is simple (i.e., Belady [91, 92]), and it is widely used in the systems community to bound miss ratios. But object sizes often vary widely in practice, from a few bytes (e.g., metadata [115]) to several gigabytes (e.g., videos [16, 36]). We need a way to compute OPT for variable object sizes, but unfortunately this is known to be NP-hard [93]. The best known approximation algorithm [1–3] is only provably within a factor of 4 of OPT. Hence, when this algorithm estimates a miss ratio of 0.4, OPT may lie anywhere between 0.1 and 0.4. This is a big range—in practice, a difference of 0.05 in miss ratio is significant—, so bounds from prior theory are of limited practical value.

Since the theoretical bounds are incomputable, practitioners have been forced to use conservative lower bounds or pessimistic upper bounds on OPT. The only prior lower bound is an infinitely large cache [30, 36, 55], which is very conservative and gives no sense of how OPT changes at different cache sizes. Belady variants (e.g., Belady-Size in Subsection 2.4.3) are widely used as an upper bound [36, 99–101], despite offering no guarantees of optimality.

While these offline bounds are easy to compute, we will show that they are in fact far from OPT. They have thus given practitioners a false sense of complacency, since existing online algorithms often achieve similar miss ratios to these weak offline upper bounds.

This chapter proposes a new approach to compute bounds on OPT with variable object sizes, which we call the *flow-based offline optimal (FOO)*. The key insight behind FOO is to represent caching as a min-cost flow problem. This formulation yields a lower bound on OPT by allowing non-integer decisions, i.e., letting the cache retain fractions of objects for a proportionally smaller reward. It also yields an upper bound on OPT by ignoring all non-integer decisions. Under simple independence assumptions, we prove

that the non-integer decisions become negligible as the number of objects goes to infinity, and thus the bounds are asymptotically tight.

Our proof is based on the observation that an optimal policy will strictly prefer some requests over others, forcing integer decisions. We show such preferences apply to almost all requests by relating such preferences to the well-known coupon collector problem.

While FOO is very accurate, it is too computationally expensive to apply directly to production traces containing hundreds of millions of requests. To extend our analysis to such traces, we develop more efficient upper and lower bounds on OPT, which we call *practical flow-based offline optimal (PFOO)*. PFOO enables the first analysis of optimal caching on traces with hundreds of millions of requests and reveals that there is still significant room to improve current caching systems.

In the following, we first give a high-level overview of FOO and our proof of FOO's optimality (Section 4.1). We then formally define FOO (Section 4.2) and state our proof of correctness (Section 4.3). We proceed by presenting the PFOO upper and lower bounds (Section 4.4). Finally, we evaluate FOO and PFOO on eight production traces (Section 4.5 and Section 4.6). We summarize the chapter's results and ideas in Section 4.7.

4.1 Flow-based Offline Optimal

This section gives a conceptual roadmap for our construction of FOO and our proof of FOO's optimality, which we present formally in Sections 4.2 and 4.3. Throughout this section we use a small request trace shown in Figure 4.1 as a running example. This trace contains four objects, **a**, **b**, **c**, and **d**, with sizes 3, 1, 1, and 2, respectively.

Object	**a**	**b**	**c**	**b**	**d**	**a**	**c**	**d**	**a**	**b**	**b**	**a**
Size	3	1	1	1	2	3	1	2	3	1	1	3

Figure 4.1: Example trace of requests to objects a, b, c, and d, of sizes 3, 1, 1, and 2, respectively.

First, we introduce a new integer linear program to represent OPT (Subsection 4.1.1). After relaxing integrality constraints, we derive FOO's min-cost flow representation, which can be solved efficiently (Subsection 4.1.2). We then observe how FOO yields tight upper and lower bounds on OPT (Subsection 4.1.3). To prove that FOO's bounds are tight on real-world traces, we relate the gap between FOO's upper and lower bounds

to the occurrence of a partial order on intervals, and then reduce the partial order's occurrence to an instance of the generalized coupon collector problem (Subsection 4.1.4).

4.1.1 Our new interval representation of OPT

We start by introducing a novel representation of OPT. Our integer linear program (ILP) minimizes the number of cache misses, while having full knowledge of the request trace.

We exploit a unique property of offline optimal caching: OPT never changes its decision to cache object k in between two requests to k (see Section 4.2). This naturally leads to an interval representation of OPT as shown in Figure 4.2. While the classical representation of OPT uses decision variables to track the state of every object at every time step [1], our ILP only keeps track of interval-level decisions. Specifically, we use decision variables x_i to indicate whether OPT caches the object requested at time i, or not.

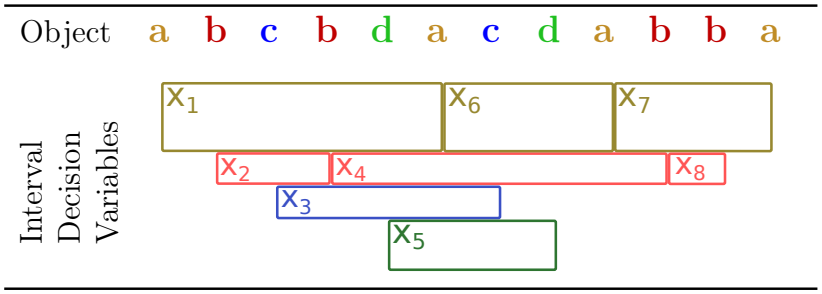


Figure 4.2: Interval ILP representation of OPT.

4.1.2 FOO's min-cost flow representation

This interval representation leads naturally to FOO's flow-based representation, shown in Figure 4.3. The key idea is to use flow to represent the interval decision variables. Each request is represented by a node. Each object's first request is a source of flow equal to the object's size, and its last request is a sink of flow in the same amount. This flow must be routed along intervening edges, and hence min-cost flow must decide whether to cache the object throughout the trace.

For cached objects, there is a central path of black edges connecting all requests. These edges have capacity equal to the cache capacity and cost zero (since cached objects lead to zero misses). Min-cost flow will thus route as much flow as possible through this central path to avoid costly misses elsewhere [116]. Table 4.1 visualizes our min cost flow notation for the example of a two-node graph.

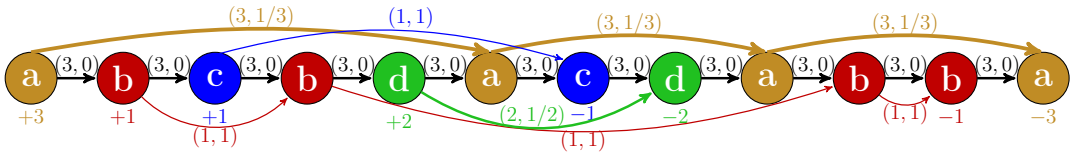


Figure 4.3: FOO's min-cost flow problem for the short trace in Figure 4.1. Nodes represent requests, and cost measures cache misses. Requests are connected by central edges with capacity equal to the cache capacity and cost zero—flow routed along this path represents cached objects (hits). Outer edges connect requests to the same object, with capacity equal to the object's size—flow routed along this path represents misses. The first request for each object is a source of flow equal to the object's size, and the last request is a sink of flow of the same amount. Outer edges' costs are inversely proportional to object size so that they cost 1 miss when an entire object is not cached. The minimum-cost flow achieves the fewest misses.

cap	Capacity of edge (i, j)
cost	Cost per unit flow on edge (i, j)
β_i	Flow surplus at node i, if $\beta_i > 0$, flow demand if $\beta_i < 0$

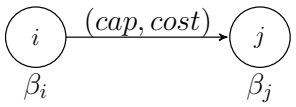


Table 4.1: Notation for FOO's min cost flow graph.

To represent cache misses, FOO adds outer edges between subsequent requests to the same object. For example, there are three edges along the top of Figure 4.3 connecting the requests to **a**. These edges have capacity equal to the object's size s and cost inversely proportional to the object's size $1/s$. Hence, if an object of size s is not cached (i.e., its flow s is routed along this outer edge), it will incur a cost of $s \times (1/s) = 1$ miss.

The routing of flow through this graph implies which objects are cached and when. When no flow is routed along an outer edge, this implies that the object is cached, and the subsequent request is a hit. All other requests, i.e., those with any flow routed along an outer edge, are misses. The min-cost flow gives the decisions that minimize total misses.

4.1.3 FOO yields upper and lower bounds on OPT

FOO can deviate from OPT as there is no guarantee that an object's flow will be entirely routed along its outer edge. Thus, FOO allows the cache to keep fractions of objects, accounting for only a fractional miss on the next request to that object. In a real system,

each fractional miss would be a full miss. This error is the price FOO pays for making the offline optimal computable.

To deal with fractional (non-integer) solutions, we consider two variants of FOO. FOO-L keeps all non-integer solutions and is therefore a lower bound on OPT. FOO-U considers all non-integer decisions as uncached, "rounding up" flow along outer edges, and is therefore an upper bound on OPT. We will prove this in Section 4.2.

Object	a	b	c	b	d	a	c	d	a	b	b	a
OPT decision	✗	✓	✓	✓	✗	✗	✓	✗	✗	✓	✗	✗
FOO-L decision	0	1	1	1	$\frac{1}{2}$	0	1	$\frac{1}{2}$	0	1	0	0
FOO-U decision	0	1	1	1	0	0	1	0	0	1	0	0

Figure 4.4: Caching decisions made by OPT, FOO-L, and FOO-U with a cache capacity of $C = 3$.

Figure 4.4 shows the caching decisions made by OPT, FOO-L, and FOO-U assuming a cache of size 3. A "✓" indicates that OPT caches the object until its next request, and a "✗" indicates it is not cached. OPT suffers five misses on this trace by caching object **b** and either **c** or **d**. OPT caches **b** because it is referenced thrice and is small. This leaves space to cache the two references to either **c** or **d**, but not both. (OPT in Figure 4.4 chooses to cache **c** since it requires less space.) OPT does not cache **a** because it takes the full cache, forcing misses on all other requests.

The solutions found by FOO-L are very similar to OPT. FOO-L decides to cache objects **b** and **c**, matching OPT, and also caches *half* of **d**. FOO-L thus underestimates the misses by one, counting **d**'s misses fractionally. FOO-U gives an upper bound for OPT by counting **d**'s misses fully. In this example, FOO-U matches OPT exactly.

4.1.4 Overview of our proof of FOO's optimality

We show both theoretically and empirically that FOO-U and FOO-L yield tight bounds. Specifically, we prove that FOO-L's solutions are almost always integer when there are many objects (as in production traces). Thus, FOO-U and FOO-L coincide with OPT.

Our proof is based on a natural precedence relation between intervals such that an optimal policy strictly prefers some intervals over others. For example, in Figure 4.2, FOO will always prefer x_2 over x_1 and x_8 over x_7. This can be seen in the figure, as interval x_2 fits entirely within x_1, and likewise x_8 fits within x_7. In contrast, no such precedence relation exists between x_6 and x_4 because **a** is larger than **b**, and so x_6 does not fit within x_4. Similarly, no precedence relation exists between x_2 and x_5 because,

although x_5 is longer and larger, their intervals do not overlap, and so x_2 does not fit within x_5.

This precedence relation means that if FOO caches any part of x_1, then it must have cached all of x_2. Likewise, if FOO caches any part of x_7, then it must have cached all of x_8. The precedence relation thus forces integer solutions in FOO. Although this relation is sparse in the small trace from Figure 4.2, as one scales up the number of objects the precedence relation becomes dense. Our challenge is to prove that this holds on traces seen in practice.

At the highest level, our proof distinguishes between "typical" and "atypical" objects. Atypical objects are those that are exceptionally unpopular or exceptionally large; typical objects are everything else. While the precedence relation may not hold for atypical objects, intervals from atypical objects are rare enough that they can be safely ignored. We then show that for all the typical objects, the precedence relation is dense. In fact, one only needs to consider precedence relations among *cached* objects, as all other interval have zero decision variables. The basic intuition behind our proof is that a popular cached object almost always takes precedence over another object. Specifically, it will take precedence over one of the exceptionally large objects, since the only way it could not is if *all* of the exceptionally large objects were requested before it was requested again. There are enough large objects to make this vanishingly unlikely.

This is an instance of the generalized *coupon collector problem* (CCP). In the CCP, one collects coupons (with replacement) from an urn with k distinct types of coupons, stopping once all k types have been collected. The classical CCP (where coupons are equally likely) is a well-studied problem [117]. The generalized CCP, where coupons have non-uniform probabilities, is very challenging and the focus of recent work in probability theory [118–121].

Applying these recent results, we show that it is extremely unlikely that a popular object does not take precedence over any others. Therefore, there are very few non-integer solutions among popular objects, which make up nearly all hits, and the gap between FOO-U and FOO-L vanishes as the number of objects grows large.

4.2 Formal Definition of FOO

This section shows how to construct FOO and that FOO yields upper and lower bounds on OPT. Section 4.2.1 introduces our notation. Section 4.2.2 defines our new interval

representation of OPT. Section 4.2.4 relaxes the integer constraints and proves that our min-cost flow representation yields upper and lower bounds on OPT.

4.2.1 Notation and definitions

The trace σ consists of N requests to M distinct objects. The i-th request σ_i contains the corresponding object id, for all $i \in \{1 \dots N\}$. We use s_i to reference the size of the object σ_i referenced in the i-th request. We denote the i-th interval (e.g., in Figure 4.2) by $[i, \ell_i)$, where

$$\ell_i = \begin{cases} \infty \text{ , if } \sigma_i \text{ is not requested again;} \\ \text{the time of the next request to object } \sigma_i \text{ after time } i \text{ , otherwise.} \end{cases} \tag{4.1}$$

OPT minimizes the number of cache misses, while having full knowledge of the request trace. OPT is constrained to only use cache capacity C (bytes), and is *not allowed to prefetch* objects as this would lead to trivial solutions (no misses) [1]. Formally,

Assumption 4.2.1 *An object $k \in \{1 \dots M\}$ can only enter the cache at times $i \in \{1 \dots N\}$ with $\sigma_i = k$.*

4.2.2 New ILP representation of OPT

We start by formally stating our ILP formulation of OPT, based on intervals as illustrated in Figure 4.2. First, we define the set I of all requests i where σ_i is requested again, i.e., $I = \{i : \ell_i < \infty\}$. I is the times when OPT must decide whether to cache an object. For all $i \in I$, we associate a decision variable x_i. This decision variable denotes whether object σ_i is cached during the interval $[i, \ell_i)$. Our ILP formulation needs only $N - M$ variables, vs. $N \times M$ for prior approaches [1], and leads directly to our flow-based approximation.

Definition 4.2.1 (Definition of OPT) *The interval representation of OPT for a trace of length N with M objects is as follows.*

$$OPT = \min \sum_{i \in I} (1 - x_i) \tag{4.2}$$

subject to:

$$\sum_{j \in I \,:\, j < i < \ell_j} s_j x_j \leq C \qquad \forall i \in I \tag{4.3}$$

$$x_i \in \{0, 1\} \qquad \forall i \in I \tag{4.4}$$

To represent the capacity constraint at every time step i, our representation needs to find all intervals $[j, \ell_j)$ that intersect with i, i.e., where $j < i < \ell_j$. Eq. (4.3) enforces the capacity constraint by bounding the size of cached intervals to be less than the cache size C. Eq. (4.4) ensures that decisions are integral, i.e., that each interval is cached either fully or not at all.

4.2.3 Proof of equivalence of interval and classic ILP representations of OPT

We next prove that our new interval ILP is equivalent to classic ILP formulations of OPT from prior work [1]. Figure 4.5 shows this classic ILP representation on the example trace from Section 4.1. The ILP uses decision variables $x_{i,k}$ to track at each time i whether object k is cached or not. The constraint on the cache capacity is naturally represented: the sum of the sizes for all cached objects must be less than the cache capacity for ever time i. Additional constraints enforce that OPT is not allowed to prefetch objects (decision variables must not increase if the corresponding object is not requested) and that the cache starts empty.

Object	a	b	c	b	d	a	c	d	a	b	a...
Decision Variables	$x_{1,a}$	$x_{2,a}$	$x_{3,a}$	$x_{4,a}$	$x_{5,a}$	$x_{6,a}$	$x_{7,a}$	$x_{8,a}$	$x_{9,a}$	$x_{10,a}$	$x_{11,a}$
	$x_{1,b}$	$x_{2,b}$	$x_{3,b}$	$x_{4,b}$	$x_{5,b}$	$x_{6,b}$	$x_{7,b}$	$x_{8,b}$	$x_{9,b}$	$x_{10,b}$	$x_{11,b}$
	$x_{1,c}$	$x_{2,c}$	$x_{3,c}$	$x_{4,c}$	$x_{5,c}$	$x_{6,c}$	$x_{7,c}$	$x_{8,c}$	$x_{9,c}$	$x_{10,c}$	$x_{11,c}$
	$x_{1,d}$	$x_{2,d}$	$x_{3,d}$	$x_{4,d}$	$x_{5,d}$	$x_{6,d}$	$x_{7,d}$	$x_{8,d}$	$x_{9,d}$	$x_{10,d}$	$x_{11,d}$

Figure 4.5: Classic ILP representation of OPT.

Lemma 4.2.1 *Under Assumption 4.2.1, our ILP in Definition 4.2.1 is equivalent to the classical ILP from [1].*

Proof sketch Under Assumption 4.2.1, OPT changes the caching decision of object k only at times i when $\sigma_i = k$. To see why this is true, let us consider the two cases of changing a decision variable $x_{k,j}$ for $i < j < \ell_i$. If $x_{k,i} = 0$, then OPT cannot set $x_{k,j} = 1$ because this would violate Assumption 4.2.1. Similarly, if $x_{k,j} = 0$, then setting $x_{k,i} = 1$ does not yield any fewer misses, so we can safely assume that $x_{k,i} = 0$. Hence, decisions do not change within an interval in the classic ILP formulation.

To obtain the decision variables $x'_{p,i}$ of the classical ILP formulation of OPT from a given solution x_i for the interval ILP, set $x'_{\sigma_i,j} = x_i$ for all $i \leq j < \ell_i$, and for all i. This leads to an equivalent solution because the capacity constraint is enforced at every time step.

□

Having formulated OPT with fewer decision variables, we could try to solve the LP relaxation of this specific ILP. However, the capacity constraint, Eq. (4.3), still poses a practical problem since finding the intersecting intervals is computationally expensive. Additionally, the LP formulation does not exploit the underlying problem structure, which we need to bound the number of integer solutions. We instead reformulate the problem as min-cost flow.

4.2.4 FOO's min-cost flow representation of OPT

This section presents the relaxed version of OPT as an instance of min-cost flow (MCF) in a graph G. We denote a surplus of flow at a node i with $\beta_i > 0$, and a demand for flow with $\beta_i < 0$. Each edge (i, j) in G has a cost per unit flow $\gamma_{(i,j)}$ and a capacity for flow $\mu_{(i,j)}$ (see right-hand side of Figure 4.3).

As discussed in Section 4.1, the key idea in our construction of an MCF instance is that each interval introduces an amount of flow equal to the object's size. The graph G is constructed such that this flow competes for a single sequence of edges (the "inner edges") with zero cost. These "inner edges" represent the cache's capacity: if an object is stored in the cache, we incur zero cost (no misses). As not all objects will fit into the cache, we introduce "outer edges", which allow MCF to satisfy the flow constraints. However, these outer edges come at a cost: when the full flow of an object uses an outer edge we incur cost 1 (i.e., a miss). Non-integer decision variables arise if part of an object is in the cache (flow along inner edges) and part is out of the cache (flow along outer edges).

Formally, we construct our MCF instance of OPT as follows:

Definition 4.2.2 (FOO's representation of OPT) *Given a trace with N requests and M objects, the MCF graph G consists of N nodes. For each request $i \in \{1 \dots N\}$ there is a node with supply/demand*

$$\beta_i = \begin{cases} s_i & \text{if } i \text{ is the first request to } \sigma_i \\ -s_i & \text{if } i \text{ is the last request to } \sigma_i \\ 0 & \text{otherwise.} \end{cases} \tag{4.5}$$

*An **inner edge** connects nodes i and $i+1$. Inner edges have capacity $\mu_{(i,i+1)} = C$ and cost $\gamma_{(i,i+1)} = 0$, for $i \in \{1 \dots N-1\}$.*

*For all $i \in I$, an **outer edge** connects nodes i and ℓ_i. Outer edges have capacity $\mu_{(i,\ell_i)} = s_i$ and cost $\gamma_{(i,\ell_i)} = 1/s_i$. We denote the flow through outer edge (i, ℓ_i) as f_i.*
***FOO-L** denotes the cost of an optimal feasible solution to the MCF graph G. **FOO-U** denotes the cost if all non-zero flows through outer edges f_i are rounded up to the edge's capacity s_i.*

This representation yields a min-cost flow instance with $2N - M - 1$ edges, which is solvable in $O(N^{3/2})$ [122–124]. Note that while this chapter focuses on optimizing miss ratio (i.e., the fault model [1], where all misses have the same cost), Definition 4.2.2 easily supports non-uniform miss costs by setting outer edge costs to $\gamma_{(i,\ell_i)} = \text{cost}_i/s_i$. We next show how to derive upper and lower bounds from this min-cost flow representation.

Lemma 4.2.2 (FOO bounds OPT) *For FOO-L and FOO-U from Definition 4.2.2,*

$$\boxed{\textit{FOO-L} \leq \textit{OPT} \leq \textit{FOO-U}} \tag{4.6}$$

Proof: We observe that f_i as defined in Definition 4.2.2, defines the number of bytes "not stored" in the cache. f_i corresponds to the i-th decision variable x_i from Definition 4.2.1 as $x_i = (1 - f_i/s_i)$.
(FOO-L $\leq$ OPT): FOO-L is a feasible solution for the LP relaxation of Definition 4.2.1, because a total amount of flow s_i needs to flow from node i to node ℓ_i (by definition of β_i). At most $\mu_{(i,i+1)} = C$ flows uses an inner edge which enforces constraint Eq. (4.3). FOO-L is an optimal solution because it minimizes the total cost of flow along outer edges. Each outer edge's cost is $\gamma_{(i,\ell_i)} = 1/s_i$, so $\gamma_{(i,\ell_i)} f_i = (1 - x_i)$, and thus

$$\text{FOO-L} = \min\Big\{\sum_{i \in I} \gamma_{(i,\ell_i)} f_i\Big\} = \min\Big\{\sum_{i \in I} (1 - x_i)\Big\} \leq \text{OPT} \tag{4.7}$$

(OPT $\leq$ FOO-U): After rounding, each outer edge (i, ℓ_i) has flow $f_i \in \{0, s_i\}$, so the corresponding decision variable $x_i \in \{0, 1\}$. FOO-U thus yields a feasible integer solution, and OPT yields no more misses than any feasible solution. □

4.3 FOO is Asymptotically Optimal

This section proves that FOO is asymptotically optimal, namely that the gap between FOO-U and FOO-L vanishes as the number of objects grows large. Subsection 4.3.1 formally states this result and our assumptions, and Sections 4.3.2–4.3.5 present the proof.

4.3.1 Main result and assumptions

Our proof of FOO's optimality relies on two assumptions: *(i)* that the trace is created by stochastically independent request processes and *(ii)* that the popularity distribution is not concentrated on a finite set of objects as the number of objects grows.

Assumption 4.3.1 (Independence) *The request sequence is generated by independently sampling from a popularity distribution $\mathcal{P}^M$. Object sizes are sampled from an arbitrary continuous size distribution $\mathcal{S}$, which is independent of M and has a finite* $\max_i s_i$.

We assume that object sizes are unique to break ties when making caching decisions. If the object sizes are not unique, one can simply add small amounts of noise to make them so. We assume a maximum object size to show the existence of a scaling regime, i.e., that the number of cached objects grows large as the cache grows large. For the same reason, we exclude trivial cases where a finite set of objects dominates the request sequence even as the total universe of objects grows large:

Assumption 4.3.2 (Diverging popularity distribution) *For any number $M > 0$ of objects, the popularity distribution $\mathcal{P}^M$ is defined via an infinite sequence ψ_k. At any time $1 \leq i \leq N$,*

$$\mathbb{P}\left[\textit{object } k \textit{ is requested} \mid M \textit{ objects overall}\right] = \frac{\psi_k}{\sum_{k=1}^{M} \psi_k} \tag{4.8}$$

The sequence ψ_k must be positive and diverging such that cache size $C \to \infty$ is required to achieve a constant miss ratio as $M \to \infty$.

Our assumptions on $\mathcal{P}^M$ allow for many common distributions, such as uniform popularities ($\psi_k = 1$) or heavy-tailed Zipfian probabilities ($\psi_k = 1/k^\alpha$ for $\alpha \leq 1$, as is common in practice [16, 25, 31, 36, 125]). Moreover, with some change to notation, our proofs can be extended to require only that ψ_k remains constant over short timeframes. With these assumptions in place, we are now ready to state our main result on FOO's asymptotic optimality.

Theorem 4.3.1 (FOO is Asymptotically Optimal) *Under Assumptions 4.3.1 and 4.3.2, for any error ε and violation probability κ, there exists an M^* such that for any trace with $M > M^*$ objects*

$$\mathbb{P}\left[\textit{FOO-U} - \textit{FOO-L} \geq \varepsilon\, N\right] \leq \kappa \tag{4.9}$$

where the trace length $N \geq M \log^2 M$ and the cache capacity C is scaled with M such that FOO-L's miss ratio remains constant.

Theorem 4.3.1 states that, as $M \to \infty$, FOO's *miss ratio error* is almost surely less than ε for any $\varepsilon > 0$. Since FOO-L and FOO-U bound OPT (Lemma 4.2.2), FOO-L = OPT = FOO-U.

The rest of this section is dedicated to the proof Theorem 4.3.1. The key idea in our proof is to bound the number of non-integer solutions in FOO-L via a precedence relation that forces FOO-L to strictly prefer some decision variables over others, which forces them to be integer. Subsection 4.3.2 introduces this precedence relation. Subsection 4.3.3 maps this relation to a representation that can be stochastically analyzed (as a variant of the coupon collector problem). Subsection 4.3.4 then shows that almost all decision variables are part of a precedence relation and thus integer, and Subsection 4.3.5 brings all these parts together in the proof of Theorem 4.3.1.

4.3.2 Bounding the number of non-integer solutions using a precedence graph

This section introduces the precedence relation $\prec$ between caching intervals. The intuition behind $\prec$ is that if an interval i is nested entirely within interval j, then min-cost flow must prefer i over j. We first formally define $\prec$, and then state the property about optimal policies in Theorem 4.3.2.

Definition 4.3.1 (Precedence relation) *For two caching intervals $[i, \ell_i)$ and $[j, \ell_j)$, let the relation $\prec$ be such that $i \prec j$ ("i takes precedence over j") if*

1. $j < i$,
2. $\ell_j > \ell_i$, *and*
3. $s_i < s_j$.

The key property of $\prec$ is that it forces integer decision variables.

Theorem 4.3.2 (Precedence forces integer decisions) *If $i \prec j$, then $x_j > 0$ in FOO-L's min-cost flow solution implies $x_i = 1$.*

In other words, if interval i is strictly preferable to interval j, then FOO-L will take all of i before taking any of j. The proof of this result relies on the notion of a residual MCF graph [116, p.304 ff], where for any edge $(i, j) \in G$ with positive flow, we add a backwards edge (j, i) with cost $\gamma_{j,i} = -\gamma_{i,j}$.

Proof: By contradiction. Let G' be the residual MCF graph induced by a given MCF solution. Figure 4.6 sketches the MCF graph in the neighborhood of $j, \ldots, i, \ldots, \ell_i, \ldots, \ell_j$.

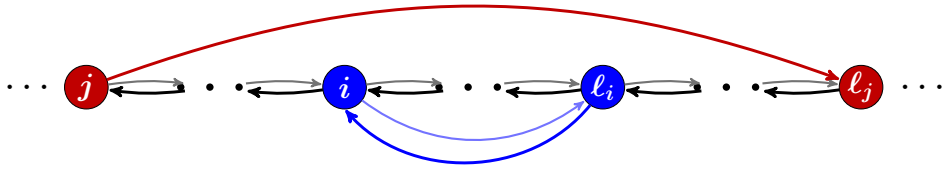


Figure 4.6: The precedence relation $i \prec j$ from Definition 4.3.1 forces integer decisions on interval i. In any min-cost flow solution, we can reroute flow such that if $x_j > 0$ then $x_i = 1$.

Assume that $x_j > 0$ and that $x_i < 1$, as otherwise the statement is trivially true. Because $x_j > 0$ there exist backwards inner edges all the way between ℓ_j and j. Because $x_i < 1$, the MCF solution must include some flow on the outer edge (i, ℓ_i), and there exists a backwards outer edge $(\ell_i, i) \in G'$ with cost $\gamma_{\ell_i,i} = -1/s_i$.

We can use the backwards edges to create a cycle in G', starting at j, then following edge (j, ℓ_j), backwards inner edges to ℓ_i, the backwards outer edge (ℓ_i, i), and finally backwards inner edges to return to j. Figure 4.6 highlights this clockwise path in darker colored edges.

This cycle has cost $= 1/s_j - 1/s_i$, which is negative because $s_i < s_j$ (by definition of $\prec$ and since $i \prec j$). As negative-cost cycles cannot exist in a MCF solution [116, Theorem 9.1, p.307], this leads to a contradiction.

□

To quantify how many integer solutions there are, we need to know the general structure of the precedence graph. Intuitively, for large production traces with many overlapping intervals, the graph will be very dense. We have empirically verified this for our production traces.

Unfortunately, the combinatorial nature of caching traces made it difficult for us to characterize the general structure of the precedence graph under stochastic assumptions. For example, we considered classical results on random graphs [126] and the concentration of measure in random partial orders [127]. None of these paths yielded sufficiently tight bounds on FOO. Instead, we bound the number of non-integer solutions via the generalized coupon collector problem.

4.3.3 Relating the precedence graph to the coupon collector problem

We now translate the problem of intervals without child in the precedence graph (i.e., intervals i for which there exists no $i \prec j$) into a tractable stochastic representation.

We first describe the intuition for equal object sizes, and then consider variable object sizes.

Definition 4.3.2 (Cached objects) *Let H_i denote the set of cached intervals that overlap time i, excluding i.*

$$H_i = \left\{ j \neq i : \;\; x_j > 0 \text{ and } i \in [j, \ell_j) \right\} \qquad \text{and} \qquad h_i = |H_i| \tag{4.10}$$

We observe that interval $[i, \ell_i)$ is without child if and only if *all other objects* $x_j \in H_i$ are *requested at least once* in $[i, \ell_i)$. Figure 4.7 shows an example where H_i consists of five objects (intervals $x_a, \ldots, x_e$). As all five objects are requested before ℓ_i, all five intervals

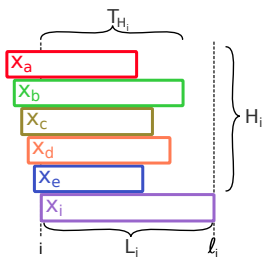


Figure 4.7: Simplified notation for the coupon collector representation of offline caching with equal object sizes. We translate the precedence relation from Theorem 4.3.2 into the relation between two random variables. L_i denotes the length of interval i. T_{H_i} is the coupon collector time, where we wait until all objects that are cached at the beginning of L_i (H_i denotes these objects) are requested at least once.

end before x_i ends, and so $[i, \ell_i)$ cannot fit in any of them. To formalize this observation, we introduce the following random variables, also illustrated in Figure 4.7. L_i is the length of the i-th interval, i.e., $L_i = \ell_i - i$. T_{H_i} is the time after i when all intervals in H_i end. We observe that T_{H_i} is the stopping time in a *coupon-collector problem* (CCP) where we associate a coupon type with every object in H_i. With equal object sizes, the event $\{i \text{ has a child}\}$ is equivalent to the event $\{T_{H_i} > L_i\}$.

We now extend our intuition to the case of variable object sizes. We now need to consider that objects in H_i can be smaller than s_i and thus may not be i's children for a new reason: the precedence relation (Definition 4.3.1) requires i's children to have size larger than or equal to s_i. Figure 4.8 shows an example where L_i is without child because *(i)* x_b, which ends after ℓ_i, is smaller than s_i, and *(ii)* all larger objects (x_a, x_c, x_d) are requested before ℓ_i. The important conclusion is that, by ignoring the smaller objects, we can reduce the problem back to the CCP.

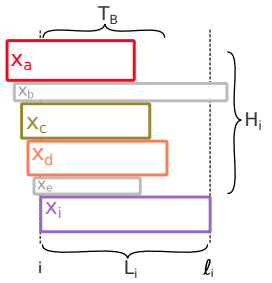


Figure 4.8: Full notation for the coupon collector representation of offline caching. With variable object sizes, we need to ignore all objects with a smaller size than s_i (grayed out intervals x_b and x_e). We then define the coupon collector time T_B among a subset $B \subset H_i$ of cached objects with a size larger than or equal to s_i. Using this notation, the event $T_B > L_i$ implies that x_i has a child, which forces x_i to be integer by Theorem 4.3.2.

To formalize our observation about the relation to the CCP, we introduce the following random variables, also illustrated in Figure 4.8. We define B, which is a subset of the cached objects H_i with a size equal to or larger than s_i, and the coupon collector time T_B for B-objects. These definitions are useful as the event $\{T_B > L_i\}$ implies that i has a child and thus x_i is integer, as we now show.

Theorem 4.3.3 (Stochastic bound on non-integer variables) *For decision variable x_i, $i \in \{1 \dots N\}$, assume that $B \subseteq H_i$ is a subset of cached objects where $s_j \geq s_i$ for all $j \in B$. Further, let the random variable T_B denote the time until all intervals in B end, i.e., $T_B = \max_{j \in B} \ell_j - i$.*

If B is non-empty, then the probability that x_i is non-integer is upper bounded by the probability interval i ends after all intervals in B, i.e.,

$$\mathbb{P}\left[0 < x_i < 1\right] \leq \mathbb{P}\left[L_i > T_B\right] \quad . \tag{4.11}$$

The proof works backwards by assuming that $T_B > L_i$. We then show that this implies that there exists an interval j with $i \prec j$ and then apply Theorem 4.3.2 to conclude that x_i is integer. Finally, we use this implication to bound the probability.

Proof: Consider an arbitrary $i \in \{1 \dots N\}$ with $T_B > L_i$. Let $[j, \ell_j)$ denote the interval in B that is last requested, i.e., $\ell_j = \max_{k \in B} \ell_k$ (j exists because B is non-empty). To show that $i \prec j$, we check the three conditions of Definition 4.3.1.

1. $j < i$, because $j \in H_i$ (i.e., j is cached at time i);
2. $\ell_j > \ell_i$, because $\ell_j = \max_{k \in B} \ell_k = i + T_B > i + L_i = \ell_i$; and
3. $s_i < s_j$, because $j \in B$ (i.e., s_j is bigger than s_i by assumption).

Having shown that $i \prec j$, we can apply Theorem 4.3.2, so that $x_j > 0$ implies $x_i = 1$. Because $j \in H_i$, j is cached $x_j > 0$ and thus $x_i = 1$. Finally, we observe that $L_i \neq T_B$ and conclude the theorem's statement by translating the above implications, $x_i = 1 \Leftarrow i \prec j \Leftarrow T_B > L_i$, into probability.

$$\mathbb{P}\left[0 < x_i < 1\right] = 1 - \mathbb{P}\left[x_i \in \{0, 1\}\right] \leq 1 - \mathbb{P}\left[i \prec j\right] \leq 1 - \mathbb{P}\left[T_B > L_i\right] = \mathbb{P}\left[L_i > T_B\right] \quad . \tag{4.12}$$

□

Theorem 4.3.3 simplifies the analysis of non-integer x_i to the relation of two random variables, L_i and T_B. While L_i is geometrically distributed, T_B's distribution is more involved.

We map T_B to the stopping time $T_{b,p}$ of a generalized CCP with $b = |B|$ different coupon types. The coupon probabilities p follow from the object popularity distribution $\mathcal{P}^M$ by conditioning on objects in B. As the object popularities p are not equal in general, characterizing the stopping time $T_{b,p}$ is much more challenging than in the classical CCP, where the coupon probabilities are assumed to be equal. We solve this problem by observing that collecting b coupons under equal probabilities stops faster than under p. This fact may appear obvious, but it was only recently shown by Anceaume et al. [121, Theorem 4, p. 415] (the proof is non-trivial). Thus, we can use a classical CCP to bound the generalized CCP's stopping time and T_B.

Lemma 4.3.1 (Connection to classical coupon connector problem) *For any object popularity distribution $\mathcal{P}^M$, and for $q = (1/b, \dots, 1/b)$, using the notation from Theorem 4.3.3*

$$\mathbb{P}\left[T_B < l\right] \leq \mathbb{P}\left[T_{b,q} < l\right] \qquad \textit{for any} \qquad l \geq 0 \quad . \tag{4.13}$$

The proof of this Lemma simply extends the following result by Anceaume et al., which is proven as Theorem 4, in [121, p. 415].

Theorem 4.3.4 *For $b \geq 0$ coupons, any probability vector p, and the equal-probability vector $q = (1/b, \ldots, 1/b)$, it holds that $\mathbb{P}[T_{b,p} < l] \leq \mathbb{P}[T_{b,q} < l]$ for any $l \geq 0$.*

The proof bounds $T_{b,q}$ first using a GCCP and then a CCCP.

Proof: We first bound T_B via a generalized CCP with stopping time $T_{b,p}$ and $p = (p_1, \ldots, p_b)$ with

$$p_i = \mathbb{P}[\text{object } k \text{ is requested} \mid k \in B] \quad \text{under } \mathcal{P}^M \quad . \tag{4.14}$$

As T_B contains requests to other objects $j \notin B$, it always holds that $T_B \geq T_{b,p}$. Figure 4.9 shows such a case, where T_B is extended because of requests to uncached objects and large cached objects. $T_{b,p}$, on the other hand, does not include these other requests, and is thus always shorter or equal to T_B.

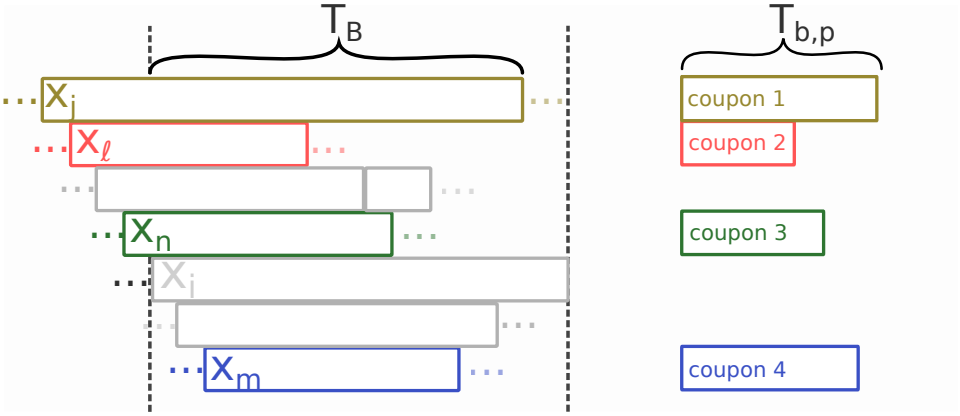


Figure 4.9: Translation of the time until all B objects are requested once, T_B into a coupon-collector problem (CCP), $T_{b,p}$. As the CCP is based on fewer coupons (only objects $\in B$), the CCP serves as a lower bound on T_B.

This inequality bounds the probabilities for all $l \geq 0$.

$$\mathbb{P}[T_B < l] \leq \mathbb{P}[T_{b,p} < l] \tag{4.15}$$

We then apply Theorem 4.3.4.

$$\leq \mathbb{P}[T_{b,q} < l] \tag{4.16}$$

□

4.3.4 Typical objects almost always lead to integer decision variables

We now use the connection to the coupon collector problem to show that almost all of FOO's decision variables are integer. Specifically, we exclude a small number of very large and unpopular objects, and show that the remaining objects are almost always part of a precedence relation, which forces the corresponding decision variables to become integer. In Subsection 4.3.5, we show that the excluded fraction is diminishingly small.

We start with a definition of the sets of large objects and the set of popular objects.

Definition 4.3.3 (Large objects and popular objects) *Let N^* be the time after which the cache needs to evict at least one object. For time $i \in \{N^* \dots N\}$, we define the sets of large objects, B_i, and the set of popular objects, F_i.*

The set $B_i \subseteq H_i$ consists of the requests to the fraction δ largest objects of H_i ($0 < \delta < 1$). We also define $b_i = |B_i|$, and we write "$s_i < B_i$" if $s_i < s_j$ for all $j \in B_i$ and "$s_i \not< B_i$" otherwise.

The set F_i consists of those objects k with a request probability ρ_k which lies above the following threshold.

$$F_i = \left\{ k : \rho_k \geq \frac{1}{b_i \log \log b_i} \right\} \tag{4.17}$$

Using the above definitions, we prove that "typical" objects (i.e., popular objects that are not too large) rarely lead to non-integer decision variables as the number of objects M grows large.

Theorem 4.3.5 (Typical objects are rarely non-integer) *For $i \in \{N^* \dots N\}$, B_i, and F_i from Definition 4.3.3,*

$$\mathbb{P}\left[0 < x_i < 1 \;\middle|\; s_i < B_i, \sigma_i \in F_i\right] \to 0 \text{ as } M \to \infty \quad . \tag{4.18}$$

The intuition is that, as the number of cached objects grows large, it is vanishingly unlikely that *all* objects in B_i will be requested before a *single* object is requested again. That is, though there are not many large objects in B_i, there are enough that, following Theorem 4.3.3, x_i is vanishingly unlikely to be non-integer. The proof of Theorem 4.3.5 uses elementary probability but relies on several very technical proofs.

We first given an overview, then state two auxiliary results, and then state the proof.

Overview and main ideas in the proof of Lemma 4.3.5

Following Theorem 4.3.3, it suffices to consider the event $\{L_i > T_{B_i}\}$.

- ($\mathbb{P}[L_i > T_{B_i}] \to 0$ as $h_i \to \infty$): We first condition on $L_i = l$, so that L_i and T_{B_i} become stochastically independent and we can bound $\mathbb{P}[L_i > T_{B_i}]$ by bounding either $\mathbb{P}[L_i > l]$ or $\mathbb{P}[l > T_{B_i}]$. Specifically, we split l carefully into "small l" and "large l", and then show that L_i is concentrated at small l, and T_{B_i} is concentrated at large l. Hence, $\mathbb{P}[L_i > T_{B_i}]$ is negligible.
  - (Small l:) For small l, we show that it is unlikely for all objects in B_i to have been requested after l requests. We upper bound the distribution of T_{B_i} with $T_{b_i,u}$ (4.3.1). We then show that the distribution of $T_{b_i,u}$ decays exponentially at values below its expectation (4.3.2). Hence, for l far below the expectation of $T_{b_i,u}$, the probability vanishes $\mathbb{P}[T_{b_i,u} < l] \to 0$, so long as $b_i = \delta h_i$ grows large, which it does because h_i grows large.
  - (Large l:) For large l, $\mathbb{P}[L_i > l] \to 0$ because we only consider popular objects $\sigma_i \in F_i$ by assumption, and it is highly unlikely that a popular object is not requested after many requests.
- ($h_i \to \infty$): What remains to be shown is that the number of cached objects h_i actually grows large. Since the cache size $C \to \infty$ as $M \to \infty$ by Assumption 4.3.2, this may seem obvious. Nevertheless, it must be demonstrated (4.3.3). The basic intuition is that h_i is almost never much less than $h^* = C/\max_k s_k$, the fewest number of objects that could fit in the cache, and $h^* \to \infty$ as $C \to \infty$.

  To see why h_i is almost never much less than h^*, consider the probability that $h_i < x$, where x is constant with respect to M. For any x, take large enough M such that $x < h^*$.

  Now, in order for $h_i < x$, almost all requests must go to distinct objects. Any object that is requested twice between u (the last time where $h_u \geq h^*$) and v (the next time where $h_v \geq h^*$) produces an interval (see Figure 4.10). This interval is guaranteed to fit in the cache, since $h_i < x < h^*$ means there is space for an object of any size. As h^* and M grow further, the amount of cache resources that must lay unused for $h_i < x$ grows further and further, and the probability that no interval fits within these resources becomes negligible.

Before we state the proof of Lemma 4.3.5, we introduce two auxiliary results.

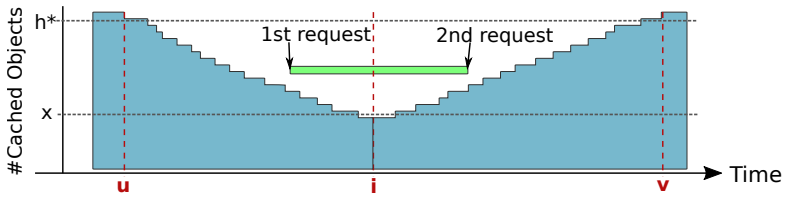


Figure 4.10: The number of cached objects at time i, h_i, is unlikely to be far below $h^* = C/\max_k s_k$, the fewest number of objects that fit in the cache. In order for $h_i < h^*$ to happen, no other interval must fit in the white triangular space centered at i (otherwise FOO-L would cache the interval).

Exponential coupon collector bound

Our first auxiliary result derives an exponential bound on the lower tail of the distribution of the coupon collector stopping time as it gets further from its mean (roughly $h \log h$).

Lemma 4.3.2 *The time $T_{b,q}$ to collect $b > 1$ coupons, which have equal probabilities $q = (1/b, \ldots, 1/b)$, is lower bounded by*

$$\mathbb{P}\left[T_{b,q} \leq b \log b - c\, b\right] < e^{-c} \quad \textit{for all } c > 0 \quad . \tag{4.19}$$

Proof: We consider the time $T_{b,q}$ to collect $b > 1$ coupons, which have equal probabilities $q = (1/b, \ldots, 1/b)$. To simplify notation, we set $T = T_{b,q}$ throughput this proof.

We first transform our term using the exponential function, which is strictly monotonic.

$$\mathbb{P}\left[T \leq b \log b - c\, b\right] = \mathbb{P}\left[e^{-sT} \leq e^{-s(b \log b - c\, b)}\right] \quad \text{for all } s > 0 \tag{4.20}$$

We next apply the Chernoff bound.

$$\mathbb{P}\left[e^{-sT} \leq e^{-s(b \log b - c\, b)}\right] \leq \mathbb{E}\left[e^{-sT}\right] e^{s(b \log b - c\, b)} \tag{4.21}$$

To derive $\mathbb{E}\left[e^{-sT}\right]$, we observe that $T = \sum_{i=1}^{b} T_i$, where T_i is the time between collecting the $(i-1)$-th unique coupon and the i-th unique coupon. As all T_i are independent, we obtain a product of Laplace-Stieltjes transforms.

$$\mathbb{E}\left[e^{-sT}\right] = \prod_{i=1}^{b} \mathbb{E}\left[e^{-sT_i}\right] \tag{4.22}$$

We derive the individual transforms.

$$\mathbb{E}\left[e^{-sT_i}\right] = \sum_{k=1}^{\infty} e^{-s\,k} p_i (1-p_i)^{k-1} \tag{4.23}$$

$$= \frac{p_i}{e^s + p_i - 1} \tag{4.24}$$

We plug the coupon probabilities $p_i = 1 - \frac{i-1}{b} = \frac{b-i+1}{b}$ into Eq. (4.22), and simplify by reversing the product order.

$$\prod_{i=1}^{b} \mathbb{E}\left[e^{-sT_i}\right] = \prod_{i=1}^{b} \frac{(b-i+1)/b}{e^s + (b-i+1)/b - 1} = \prod_{j=1}^{b} \frac{j/b}{e^s + j/b - 1} \tag{4.25}$$

Finally, we choose $s = \frac{1}{b}$, which yields $e^s = e^{1/b} \geq 1 + 1/b$ and simplifies the product.

$$\prod_{j=1}^{b} \frac{j/b}{e^s + j/b - 1} \leq \prod_{j=1}^{b} \frac{j/b}{1/b + j/b} = \frac{1}{b+1} \tag{4.26}$$

This gives the statement of the lemma.

$$\mathbb{P}\left[T \leq b \log b - c\, b\right] \leq \frac{1}{b+1}\, e^{\frac{b \log b - c\, b}{b}} < e^{-c} \tag{4.27}$$

□

The number of cached objects grows to infinity

Our second auxiliary result shows that the number of cached objects, h_i, goes to infinity as the cache capacity, C, and the number of objects, M, go to infinity.

Lemma 4.3.3 *For $i > N^*$ from Definition 4.3.3, $\mathbb{P}\left[h_i \to \infty\right] = 1$ as $M \to \infty$.*

We assume throughout the proof that $i > N^*$, the time after which the cache needs to evict at least one object. Throughout the proof, let $\overline{E}$ denote the complementary event of an event E.

Intuition of the proof. The proof exploits the fact that at least $h^* = C/\max_k s_k$ distinct objects fit into the cache at any time, and that FOO finds an optimal solution (4.2.2). Due to Assumption 4.3.2, $M \to \infty$ implies that $C \to \infty$, and thus $h^* \to \infty$. So, the case where FOO caches only a finite number of objects requires that $h_i < h^*$. Whenever $h_i < h^*$ occurs, there cannot exist intervals that FOO could put into the

cache. If any intervals could be put into the cache, FOO would cache them, due to its optimality.

Our proof is by induction. We first show that the case of no intervals that could be put into the cache has zero probability, and then prove this successively for larger thresholds.

Proof: We assume that $0 < h^* < M$ because $h^* \in \{0, M\}$ leads to trivial hit ratios $\in \{0, 1\}$.

For arbitrary $i > N^*$ and any x that is constant in M, we consider the event $X = \{h_i \leq x\}$. Furthermore, let $Z = \{h_i \leq z\}$ for any $0 \leq z \leq x$. We prove that $\mathbb{P}[X]$ vanishes as M grows by induction over z and corresponding $\mathbb{P}[Z]$.

Figure 4.11 sketches the number of objects over a time interval including i. Note that, for any $z \leq x$, we can take a large enough M such that $z < h^*$, because $h^* \to \infty$. So the figure shows $h^* > z$. The figure also defines the time interval $[u, v]$, where u is the last time before i when FOO cached h^* objects, and v is the next time after i when FOO caches h^* objects. So, for times $j \in (u, v)$, it holds that $h_j < h^*$.

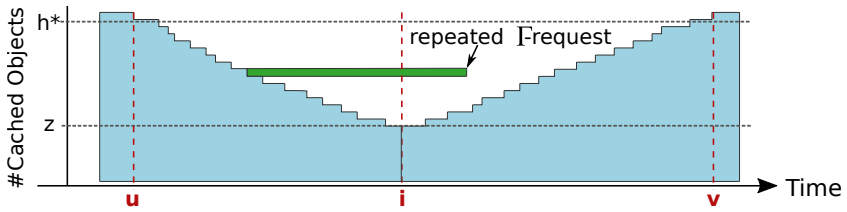


Figure 4.11: Sketch of the event $Z = \{h_i \leq z\}$, which happens with vanishing probability if z is a constant with respect to M. The times u and v denote the beginning and the end of the current period where the number of cached objects is less than $h^* = C/\max_k s_k$, the fewest number of objects that fit in the cache. We define the set Γ of objects that are requested in $(u, i]$. If any object in Γ is requested in $[i, v)$, then FOO must cache this object (green interval). If such an interval exists, $h_i > z$ and thus Z cannot happen.

Induction base: $z = 0$ *and event* $Z = \{h_i \leq 0\}$. In other words, Z means the cache is empty. Let Γ denote the set of distinct objects requested in the interval $(u, i]$. Note that the event Z requires that, during $(u, i]$, FOO stopped caching all h^* objects. Because FOO only changes caching decisions at interval boundaries, Γ must at least contain h^* objects. Using the same argument, we observe that there happen at least h^* requests to distinct objects in $[i, v)$.

A request to any Γ object in $[i, v)$ makes Z impossible. Formally, let A denote the event that any object in Γ is requested again in $[i, v)$. We observe that $A \Rightarrow \overline{Z}$ because any

Γ-object that is requested in $[i, v)$ must be cached by FOO due to FOO-L's optimality (4.2.2). By inverting the implication we obtain $Z \Rightarrow \overline{A}$ and thus $\mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right]$.

We next upper bound $\mathbb{P}\left[\overline{A}\right]$. We start by observing that $\mathbb{P}[A]$ is minimized (and thus $\mathbb{P}\left[\overline{A}\right]$ is maximized) if all objects are requested with equal popularities. This follows because, if popularities are not equal, popular objects are more likely to be in Γ than unpopular objects due to the popular object's higher sampling probability (similar to the inspection paradox). When Γ contains more popular objects, it is more likely that we repeat a request in $[i, v)$, and thus $\mathbb{P}[A]$ increases ($\mathbb{P}\left[\overline{A}\right]$ decreases).

We upper bound $\mathbb{P}\left[\overline{A}\right]$ by assuming that objects are requested with equal probability $\rho_k = 1/M$ for $1 \leq k \leq M$. As the number of Γ-objects is at least h^*, the probability of requesting any Γ-object is at least h^*/M. Further, we know that $v - i \geq h^*$ and so

$$\mathbb{P}\left[\overline{A}\right] \leq \left(1 - \frac{h^*}{M}\right)^{h^*} \quad .$$

We arrive at the following bound.

$$\mathbb{P}[\{h_i \leq z\}] = \mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right] \leq \left(1 - \frac{h^*}{M}\right)^{h^*} \longrightarrow 0 \quad \text{as } M \to \infty \tag{4.28}$$

Induction step: $z - 1 \to z$ for $z \leq x$. We assume that the probability of caching only $z - 1$ objects goes to zero as $M \to \infty$. We prove the same statement for z objects.

As for the induction base, let Γ denote the set of distinct objects requested in the interval $(u, i]$, excluding objects in H_i. We observe that $|\Gamma| \geq h^* - z$, following a similar argument.

We define $\mathbb{P}[A]$ as above and use the induction assumption. As the probability of less than $z - 1$ is vanishingly small, it must be that $h_i \geq z$. Thus, a request to any Γ object in $[i, v)$ makes $h_i = z$ impossible. Consequently, $Z \Rightarrow \overline{A}$ and thus $\mathbb{P}[Z] \leq \mathbb{P}\left[\overline{A}\right]$.

To bound $\mathbb{P}[A]$, we focus on the requests in $[i, v)$ that do not go H_i-objects. There are at least $h^* - x$ such requests. $\mathbb{P}[A]$ is minimized if all objects, ignoring objects in H_i, are requested with equal popularities. We thus upper bound $\mathbb{P}\left[\overline{A}\right]$ by assuming the condition requests happen to objects with equal probability $\rho_k = 1/(M - z)$ for $1 \leq k \leq M - z$. As before, we conclude that the probability of requesting any Γ-object

is at least $(h^* - z)/(M - z)$ and we use the fact that there are at least $h^* - x$ to them in $[i, v)$.

$$\mathbb{P}\left[\{h_i \leq z\}\right] = \mathbb{P}\left[Z\right] \leq \mathbb{P}\left[\overline{A}\right] \tag{4.29}$$

$$\leq \left(1 - \frac{h^* - z}{M - z}\right)^{h^* - z} \tag{4.30}$$

We then use that $z \leq x$ and that x is constant in M.

$$\leq \left(1 - \frac{h^* - x}{M}\right)^{h^* - x} \longrightarrow 0 \quad \text{as } M \to \infty \tag{4.31}$$

In summary, the number of objects cached by FOO at an arbitrary time i remains constant only with vanishingly small probability. Consequently, this number grows to infinity with probability one.

□

Proof of Lemma 4.3.5

With these results in place, we are ready to prove Lemma 4.3.5. This proof proceeds by using elementary probability theory and exploits our previous definitions of L_i, T_{H_i}, and $T_{h_i,u}$ from Lemma 4.3.1.

Proof: We know from Theorem 4.3.3 that the probability of non-integer decision variables can be upper bounded using the random variables L_i and T_{B_i}.

$$\mathbb{P}\left[0 < x_i < 1\right] \leq \mathbb{P}\left[L_i > T_{B_i}\right] \tag{4.32}$$

We expand this expression by conditioning on $L_i = l$.

$$= \sum_{l=1}^{\infty} \mathbb{P}\left[T_{B_i} < l | L_i = l\right] \mathbb{P}\left[L_i = l\right] \tag{4.33}$$

We observe that $\mathbb{P}\left[T_{B_i} < l\right] = 0$ for $l \leq b_i$ because requesting b_i distinct objects takes at least b_i time steps.

$$= \sum_{l=b_i+1}^{\infty} \mathbb{P}\left[T_{B_i} < l | L_i = l\right] \mathbb{P}\left[L_i = l\right] \tag{4.34}$$

We use the fact that conditioned on $L_i = l$, events $\{L_i = l\}$ and $\{T_{B_i} < l\}$ are stochastically independent.

$$= \sum_{l=b_i+1}^{\infty} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{4.35}$$

We split this sum into two parts, $l \leq \Lambda$ and $l > \Lambda$, where $\Lambda = \frac{1}{2} b_i \log b_i$ is chosen such that Λ scales slower than the expectation of the underlying coupon collector problem with $b_i = \delta h_i$ coupons. (Recall that $\delta = |B_i|/|H_i|$ is the largest fraction of objects in H_i, defined in Definition 4.3.3.)

$$\leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{4.36}$$

$$+ \sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[T_{B_i} < l\right] \mathbb{P}\left[L_i = l\right] \tag{4.37}$$

$$\leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] + \sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[L_i = l\right] \tag{4.38}$$

We now bound the two terms in Eq. (4.38), separately. For the first term, we start by applying 4.3.1.

$$\sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{B_i} < l\right] \leq \sum_{l=b_i}^{\Lambda} \mathbb{P}\left[T_{b_i,q} \leq l\right] \tag{4.39}$$

We rearrange the sum (replacing l by c).

$$= \sum_{c=\frac{1}{2}\log b_i}^{1+\log b_i} \mathbb{P}\left[T_{b_i,q} \leq b_i \log b_i - c\, b_i\right] \tag{4.40}$$

We apply 4.3.2.

$$< \sum_{c=\frac{1}{2}\log b_i}^{1+\log b_i} e^{-c} \tag{4.41}$$

We solve the finite exponential sum.

$$= \frac{e^2}{e^2 - e} \frac{1}{\sqrt{b_i}} \tag{4.42}$$

For the second term in Eq. (4.38), we use the fact that L_i's distribution is Geometric(ρ_{σ_i}) due to Assumption 4.3.1.

$$\sum_{l=\Lambda+1}^{\infty} \mathbb{P}\left[L_i = l\right] = \sum_{l=\Lambda+1}^{\infty} \left(1 - \rho_{\sigma_i}\right)^{l-1} \rho_{\sigma_i} \tag{4.43}$$

We solve the finite sum.

$$= \left(1 - \rho_{\sigma_i}\right)^{\Lambda} \tag{4.44}$$

We apply Definition 4.3.3, i.e., $\rho_{\sigma_i} \geq \frac{1}{b_i \log\log b_i}$.

$$\leq \left(1 - \frac{1}{b_i \log\log b_i}\right)^{\frac{1}{2} b_i \log b_i} \tag{4.45}$$

Finally, combining Eqs. (4.42) and (4.45) yields the following.

$$\mathbb{P}\left[L_i \geq T_{b_i,q}\right] < \frac{e^2}{e^2 - e} \frac{1}{\sqrt{b_i}} + \left(1 - \frac{1}{b_i \log\log b_i}\right)^{b_i \log b_i} \tag{4.46}$$

As $b_i \log b_i$ grows faster than $b_i \log\log b_i$, this proves the statement $\mathbb{P}\left[L_i \geq T_{b_i,q}\right] \to 0$ as $h_i \to \infty$ (implying that $b_i = \delta h_i \to \infty$) due to 4.3.3.

□

4.3.5 Bringing it all together: Proof of Theorem 4.3.1

This section combines our results so far and shows how to obtain the exact statement on the violation probability of Theorem 4.3.1.

There exists M^* such that for any $M > M^*$ and for any error ε and violation probability κ,

$$\mathbb{P}\left[\text{FOO-U} - \text{FOO-L} \geq \varepsilon\, N\right] \leq \kappa \tag{4.47}$$

Proof of Theorem 4.3.1 We start by bounding the cost of non-integer solutions by the number of non-integer solutions, Ω.

$$\sum_{\{i:\, 0<x_i<1\}} x_i \leq \sum_{i \in I} \mathbb{1}_{\{i:\, 0<x_i<1\}} = \Omega \tag{4.48}$$

It follows that (FOO-U $-$ FOO-L) $\leq \Omega$.

$$\mathbb{P}\left[\text{FOO-U} - \text{FOO-L} \geq \varepsilon\, N\right] \leq \mathbb{P}\left[\Omega \geq \varepsilon\, N\right] \tag{4.49}$$

We apply the Markov inequality.

$$\leq \frac{\mathbb{E}\left[\Omega\right]}{N\, \varepsilon} = \frac{1}{N\, \varepsilon} \sum_{i \in I} \mathbb{P}\left[0 < x_i < 1\right] \tag{4.50}$$

There are at most N terms in the sum.

$$\leq \frac{\mathbb{P}\left[0 < x_i < 1\right]}{\varepsilon} \tag{4.51}$$

To complete Eq. (4.47), we upper bound $\mathbb{P}\left[0 < x_i < 1\right]$ to be less than $\varepsilon\, \kappa$. We first condition on $s_i < B_i$ and $\sigma_i \in F_i$, double counting those i where $s_i \not< B_i$ and $\sigma_i \notin F_i$.

$$\begin{aligned}
\mathbb{P}\left[0 < x_i < 1\right] \leq & \mathbb{P}\left[0 < x_i < 1 \;\middle|\; s_i < B_i, \sigma_i \in F_i\right] \mathbb{P}\left[s_i < B_i, \sigma_i \in F_i\right] && (4.52)\\
& + \mathbb{P}\left[0 < x_i < 1 \;\middle|\; s_i \not< B_i\right] \mathbb{P}\left[s_i \not< B_i\right] && (4.53)\\
& + \mathbb{P}\left[0 < x_i < 1 \;\middle|\; \sigma_i \notin F_i\right] \mathbb{P}\left[\sigma_i \notin F_i\right] && (4.54)
\end{aligned}$$

Drop ≤ 1 terms.

$$\leq \mathbb{P}\left[0 < x_i < 1 \;\middle|\; s_i < B_i, \sigma_i \in F_i\right] + \mathbb{P}\left[s_i \not< B_i\right] + \mathbb{P}\left[i \notin F_i\right] \tag{4.55}$$

To bound $\mathbb{P}\left[0 < x_i < 1\right] \leq \varepsilon\, \kappa$, we choose parameters such that each term in Eq. (4.55) is less than $\varepsilon\, \kappa/3$. The first term vanishes by Theorem 4.3.5. The second term is satisfied by choosing $\delta = \varepsilon\, \kappa/3$ (Definition 4.3.3). For the third term, the probability that any cached object is unpopular vanishes as h_i grows large.

$$\mathbb{P}\left[i \notin F_i\right] \leq \frac{h_i}{b_i \log\log b_i} = \frac{3}{\varepsilon\, \kappa \log\log \varepsilon\, \kappa\, h_i/3} \to 0 \text{ as } h_i \to \infty \tag{4.56}$$

Finally, we choose M^* large enough that the first and third terms in Eq. (4.55) are each less than $\varepsilon\,\kappa/3$.

□

This concludes our theoretical proof of FOO's optimality.

4.4 Practical Flow-based Offline Optimal for Real Traces

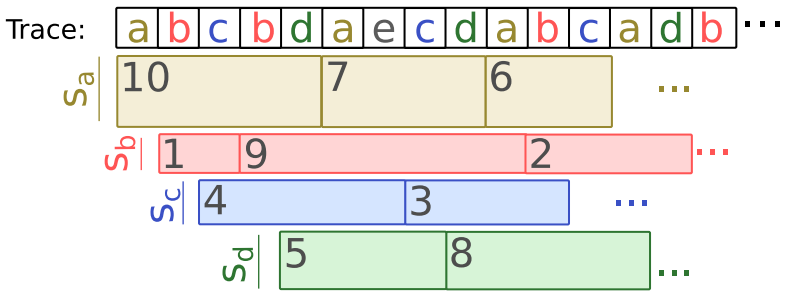
While FOO is asymptotically optimal and very accurate in practice, as well as faster than prior approximation algorithms, it is still not fast enough to process production traces with hundreds of millions of requests in a reasonable timeframe. We now use the insights gained from FOO's graph-theoretic formulation to design new upper and lower bounds on OPT, which we call *practical flow-based offline optimal (PFOO)*. We provide the first practically useful lower bound, PFOO-L, and an upper bound that is much tighter than prior practical offline upper bounds, PFOO-U:

$$\boxed{\text{PFOO-L} \leq \text{FOO-L} \leq \text{OPT} \leq \text{FOO-U} \leq \text{PFOO-U}}$$

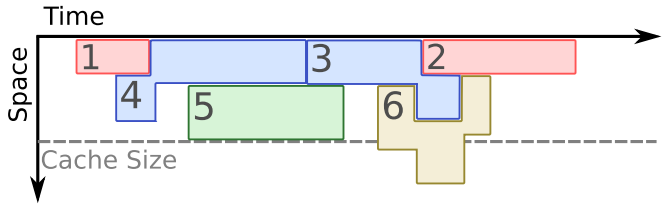
4.4.1 Practical lower bound: PFOO-L

PFOO-L considers the *total resources* consumed by OPT. As Figure 4.12a illustrates, cache resources are limited in both space and time [128]: measured in resources, the cost to cache an object is the product of *(i)* its size and *(ii)* its reuse distance (i.e., the number of accesses until it is next requested). On a trace of length N, a cache of size C has total resources $N \times C$. The objects cached by OPT, or any other policy, cannot cost more total resources than this.

Definition of PFOO-L PFOO-L sorts all intervals by their resource cost and caches the smallest-cost intervals up a total resource usage of $N \times C$. Figure 4.12b shows PFOO-L on a short request trace. By considering only the total resource usage, PFOO-L ignores other constraints that are faced by caching policies. In particular, PFOO-L does not guarantee that cached intervals take less than C space at all times, as shown by interval 6 for object **a** in Figure 4.12b, which exceeds the cache capacity during part of its interval.



(a) Intervals sorted by resource cost = reuse distance × object size.



(b) PFOO-L greedily claims the smallest intervals.

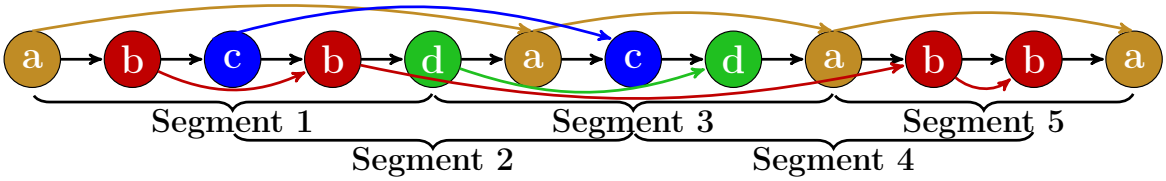
Figure 4.12: PFOO's lower bound, PFOO-L, constrains the total resources used over the full trace (i.e., size × time). PFOO-L claims the hits that require fewest resources, allowing cached objects to temporarily exceed the cache capacity.

Why PFOO-L works PFOO-L is a lower bound because no policy, including OPT, can get fewer misses using $N \times C$ total resources. It gives a reasonably tight bound because, on large caches with many objects, the distribution of interval costs is similar throughout the request trace. Hence, for a given cache capacity, the "marginal interval" (i.e., the one barely does not fit in the cache under OPT) is also of similar cost throughout the trace. Informally, PFOO-L caches intervals up to this marginal cost, and so rarely exceeds cache capacity by very much. This intuition holds particularly when requests are largely independent, as in our proof assumptions and in traces from CDNs or other Internet services. However, as we will see, PFOO-L introduces modest error even on other workloads where these assumptions do not hold.

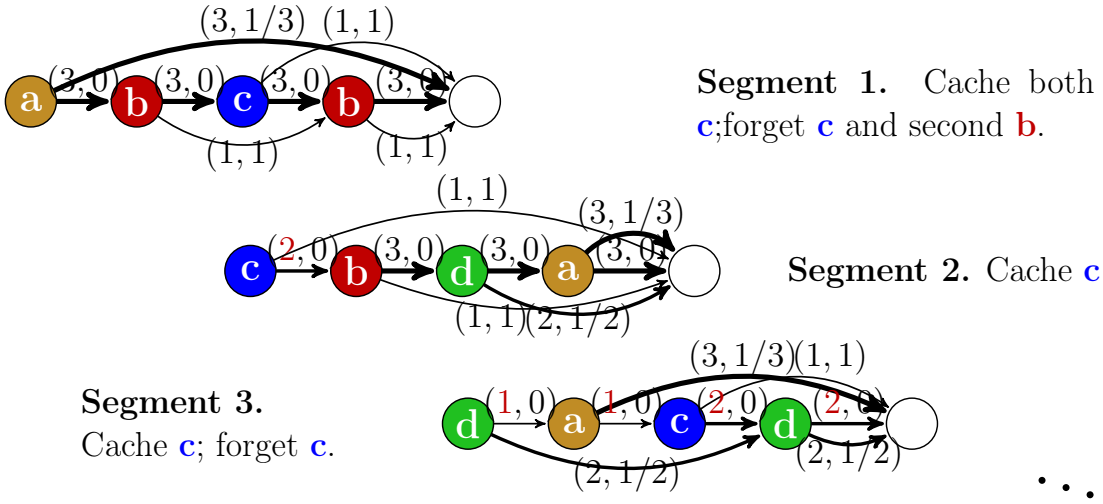
PFOO-L uses a similar notion of "cost" as Belady-Size but provides two key advantages. First, PFOO-L is closer to OPT. Relaxing the capacity constraint lets PFOO-L avoid the pathologies discussed in Subsection 2.4.3, since PFOO-L can temporarily exceed the cache capacity to retain valuable objects that Belady-Size is forced to evict. Second, relaxing the capacity constraint makes PFOO-L a *lower* bound, giving the first reasonably tight lower bound on long traces.

4.4.2 Practical upper bound: PFOO-U

Definition of PFOO-U PFOO-U breaks FOO's min-cost flow graph into smaller segments of constant size, and then solves each segment using min-cost flow incrementally. By keeping track of the resource usage of already solved segments, PFOO-U yields a globally-feasible solution, which is an upper bound on FOO-U. Furthermore, since each segment takes constant time to solve, PFOO-U completes in linear time on the trace length.



(a) PFOO-U breaks the trace into small segments . . .



(b) . . . and solves min-cost flow for each segment.

Figure 4.13: Starting from FOO's full formulation, PFOO-U breaks the min-cost flow problem into overlapping segments. Going left-to-right through the trace, PFOO-U optimally solves MCF on each segment, and updates link capacities in subsequent segments to maintain feasibility for all cached objects. The segments overlap to capture interactions across segment boundaries.

Example of PFOO-U Figure 4.13 shows our approach on the trace from Figure 4.1 for a cache capacity of 3. At the top is FOO's full min-cost flow problem; for large

traces, this MCF problem is too expensive to solve directly. Instead, PFOO-U breaks the trace into segments and constructs a min-cost flow problem for each segment.

PFOO-U begins by solving the min-cost flow for the first segment. In this case, the solution is to cache both **b**s, **c**, and one-third of **a**, since these decisions incur the minimum cost of two-thirds, i.e., less than one cache miss. As in FOO-U, PFOO-U rounds down the non-integer decision for **a** and all following non-integer decisions. Furthermore, PFOO-U only fixes decisions for objects in the first half of this segment. This is done to capture interactions between intervals that cross segment boundaries. Hence, PFOO-U "forgets" the decision to cache **c** and the second **b**, and its final decision for this segment is only to cache the first **b** interval.

PFOO-U then updates the second segment to account for its previous decisions. That is, since **b** is cached until the second request to **b**, capacity must be removed from the min-cost flow to reflect this allocation. Hence, the capacity along the inner edge **c** $\rightarrow$ **b** is reduced from 3 to 2 in the second segment (**b** is size 1). Solving the second segment, PFOO-U decides to cache **c** and **b** (as well as half of **d**, which is ignored). Since these are in the first half of the segment, we fix both decisions, and move onto the third segment, updating the capacity of edges to reflect these decisions as before.

PFOO-U continues to solve the following segments in this manner until the full trace is processed. On the trace from Section 4.1, it decides to cache all requests to **b** and **c**, yielding 5 misses on the requests to **a** and **d**. These are the same decisions as taken by FOO-U and OPT. We generally find that PFOO-U yields nearly identical miss ratios as FOO-U, as we next demonstrate on real traces.

4.4.3 Summary

Putting it all together, PFOO provides efficient lower and upper bounds on OPT with variable object sizes. PFOO-L runs in $O(N \log N)$ time, as required to sort the intervals; and PFOO-U runs in $O(N)$ because it divides the min-cost flow into segments of constant size. In practice, PFOO-L is faster than PFOO-U at realistic trace lengths, despite its worse asymptotic runtime, due to the large constant factor in solving each segment in PFOO-U.

4.5 Experimental Methodology

We evaluate FOO and PFOO against prior offline bounds and online caching policies on eight different production traces.

4.5.1 Trace Characterization

Traces We use production traces from three global content-distribution networks (CDNs), two web-applications from different anonymous large Internet companies, and storage workloads from Microsoft [129]. We summarize the trace characteristics in Table 4.2. Figure 4.14 shows four key distributions of these workloads.

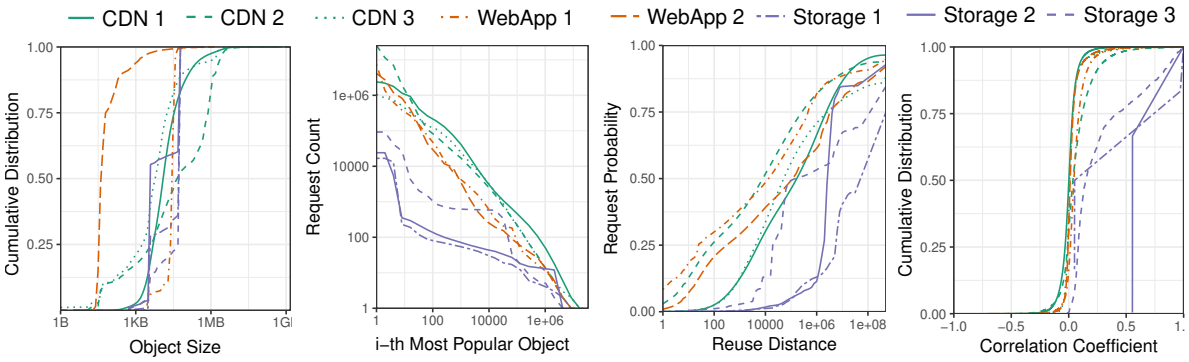
Trace	Year	# Requests	# Objects	Object sizes
CDN 1	2016	500 M	18 M	10 B – 616 MB
CDN 2	2015	440 M	19 M	1 B – 1.5 GB
CDN 3	2015	420 M	43 M	1 B – 2.3 GB
WebApp 1	2017	104 M	10 M	3 B – 1.9 MB
WebApp 2	2016	100 M	14 M	5 B – 977 KB
Storage 1	2008	29 M	16 M	501 B – 780 KB
Storage 2	2008	37 M	6 M	501 B – 78 KB
Storage 3	2008	45 M	14 M	501 B – 489 KB

Table 4.2: Length and object sizes for evaluated traces.

The object size distribution (Figure 4.14a) shows that object sizes are variable in all traces. However, while they span almost ten orders of magnitude in CDNs, object sizes vary only by six orders of magnitude in web applications, and only by three orders of magnitude in storage systems. WebApp 1 also has noticeably smaller object sizes throughout, as is representative for application-cache workloads.

The popularity distribution (Figure 4.14b) shows that CDN workloads and WebApp workloads all follow approximately a Zipf distribution with a Zipf alpha parameter between 0.85 and 1. In contrast, the popularity distribution of storage traces is much more irregular with a set of disproportionally popular objects, an approximately log-linear middle part, and an exponential cutoff for the least popular objects.

The reuse distance distribution (Figure 4.14c) — i.e., the distribution of the number of requests between requests to the same object — further distinguishes CDN and WebApp traces from storage workloads. CDNs and WebApps serve millions of different customers and so exhibit largely independent requests with smoothly diminishing object popular-



(a) Object sizes. (b) Popularities. (c) Reuse distances. (d) Correlations.

Figure 4.14: The production traces used in our evaluation come from three different domains (CDNs, WebApps, and storage) and thus exhibit starkly different request patterns in terms of object sizes (a), object popularities (b), reuse distances (c), and correlations between the request streams of different objects.

ities, which matches our proof assumptions. Thus, the CDN and WebApp traces lead to a smooth reuse distance, as shown in the figure. In contrast, storage workloads serve requests from one or a few applications, and so often exhibit highly correlated requests (producing spikes in the reuse distance distribution). For example, scans are common in storage (e.g., traces like: ABCD ABCD ...), but never seen in CDNs. This is evident from the figure, where the storage traces exhibit several steps in their cumulative request probability, as correlated objects (e.g., due to scans) have the same reuse distance.

Finally, we measure the correlation across different objects (Figure 4.14d). Ideally, we could directly test our independence assumption (Assumption 4.3.1). Unfortunately, quantifying independence on real traces is challenging. For example, classical methods such as Hoeffding's independence test [130] only apply to continuous distributions, whereas we consider cache requests in discrete time (our traces include only second-accuracy timestamps).

We therefore turn to correlation coefficients. Specifically, we use the Pearson correlation coefficient as it is computable in linear time (as opposed to Spearman's rank and Kendall's tau [131]). We define the coefficient based on the number of requests each object receives in a time bucket that spans 4000 requests (we verified that the results do not change significantly for time bucket sizes in the range 400 - 40k requests). In order to capture pair-wise correlations, we chose the top 10k objects in each trace, calculated

the request counts for all time buckets, and then calculated the Pearson coefficient for all possible combinations of object pairs.

Figure 4.14d shows the distribution of coefficients for all object pair combinations. We find that both CDN and WebApps do not show a significant correlation; over 95% of the object pair have a coefficient coefficient between -0.25 and 0.25. In contrast, we find strong positive correlations in the storage traces. For the first storage trace, we measure a correlation coefficient greater than 0.5 for all 10k-most popular objects. For the second storage trace, we measure a correlation coefficient greater than 0.5 for more than 20% of the object pairs. And, for the third storage trace, we measure a correlation coefficient greater than 0.5 for more than 14% of the object pairs. We conclude that the simple Pearson correlation coefficient is sufficiently powerful to quantify the position linear correlation inherent to storage traces (such as loops and scans).

4.5.2 Caching policies.

We evaluate three classes of policies: theoretical bounds on OPT, practical offline heuristics, and online caching policies. Besides FOO, there exist three other theoretical bounds on OPT with approximation guarantees (Subsection 2.4.2): OFMA, LocalRatio, and LP. Besides PFOO-U, we consider three other practical upper bounds (Subsection 2.4.3: Belady, Belady-Size, and Freq/Size. Besides PFOO-L, there is only one other practical lower bound: a cache with infinite capacity (Infinite-Cap). Finally, for online policies, we evaluated GDSF [132], GD-Wheel [133], AdaptSize Chapter 3, and Hyperbolic [134]. We also evaluated several other older policies which perform much worse on our traces (including LRU-K [57], TLFU [35], SLRU [36], and LRU).

Our implementations are in C++ and use the COIN-OR::LEMON library [135], GNU parallel [136], OpenMP [137], and CPLEX 12.6.1.0. OFMA runs in $O(N^2)$, LocalRatio runs in $O(N^3)$, Belady in $O(N \log C)$. We rely on sampling [138] to run Belady-Size on large traces, which gives us an $O(N)$ implementation.

4.6 Empirical Evaluation

We evaluate FOO and PFOO to demonstrate the following: *(i)* PFOO is fast enough to process real traces, whereas FOO and prior theoretical bounds are not; *(ii)* FOO yields nearly tight bounds on OPT, even when our proof assumptions do not hold; *(iii)* PFOO is highly accurate on full production traces; and *(iv)* PFOO reveals that

there is significantly more room for improving current caching systems than implied by prior offline bounds.

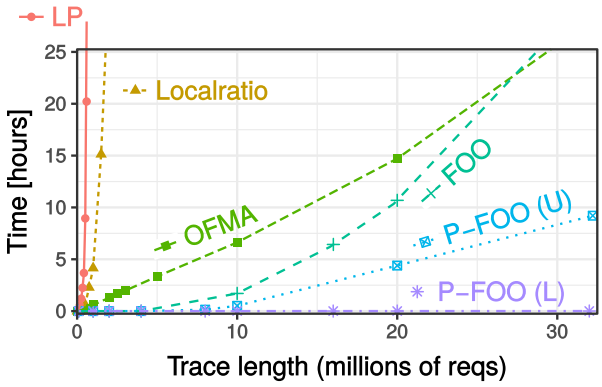


Figure 4.15: Execution time of FOO, PFOO, and prior theoretical offline bounds at different trace lengths. Most prior bounds are unusable above 500 K requests. Only PFOO can process real traces with many millions of requests.

4.6.1 PFOO is necessary to process real traces

Figure 4.15 shows the execution time of FOO, PFOO, and prior theoretical offline bounds at different trace lengths. Specifically, we run each policy on the first N requests of the CDN 1 trace, and vary N from a few thousand to over 30 million. Each policy ran alone on a 2016 SuperMicro server with 44 Intel Xeon E5-2699 cores and 500 GB of memory.

These results show that LP and LocalRatio are unusable: they can process only a few hundred thousand requests in a 24-hour period, and their execution time increases rapidly as traces lengthen. While FOO and OFMA are faster, they both take more than 24 hours to process more than 30 million requests, and their execution times increase super-linearly.

Finally, PFOO is much faster and scales well, allowing us to process traces with hundreds of millions of requests. PFOO's lower bound completes in a few minutes, and while PFOO's upper bound is slower, it scales linearly with trace length. *PFOO is thus the only bound that completes in reasonable time on real traces.*

[9]While we have tried downsampling the traces to run LP and LocalRatio (as suggested in [139–141] for equal-sized objects), we were unable to achieve meaningful results. Under variable object sizes, scaling down the system (including the cache capacity), makes large objects disproportionately disruptive.

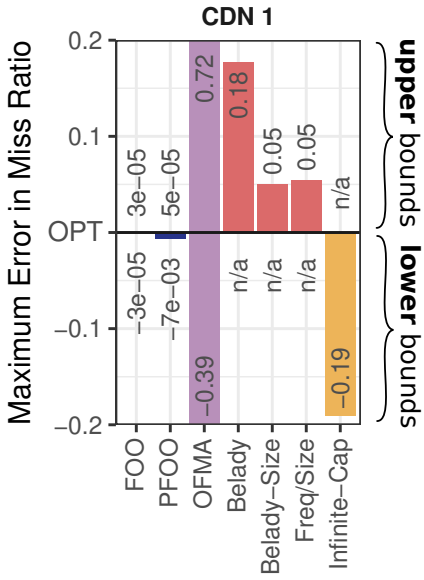


Figure 4.16: Comparison of the maximum approximation error of FOO, PFOO, and prior offline bounds across five cache sizes on a CDN production trace. FOO's upper and lower bounds are nearly identical and PFOO introduces small error, whereas all prior policies have error several orders-of-magnitude larger. (OPT is assumed to be halfway between FOO-U and FOO-L, which introduces negligible error due to FOO's high accuracy.)

4.6.2 FOO is nearly exact on short traces

We compare FOO, PFOO, and prior theoretical upper bounds on the first 10 million requests of each trace. Of the prior theoretical upper bounds, only OFMA runs in a reasonable time at this trace length,[9] so we compare FOO, PFOO, OFMA, the Belady variants, Freq/Size, and Infinite-Cap.

Our first finding is that *FOO-U and FOO-L are nearly identical*, as predicted by our analysis. The largest difference between FOO-U's and FOO-L's miss ratio on CDN and WebApp traces is 0.0005—a relative error of 0.15%. Even on the storage traces, where requests are highly correlated and hence our proof assumptions do not hold, the largest difference is 0.0014—a relative error of 0.27%. Compared to the other offline bounds, FOO is at least an order of magnitude and often several orders of magnitude more accurate.

Given FOO's high accuracy, we use FOO to estimate the error of the other offline bounds. Specifically, we assume that OPT lies in the middle between FOO-U and FOO-L. Since the difference between FOO-U and FOO-L is so small, this adds negligible error (less than 0.14%) to all other results.

Figure 4.16 shows the maximum error from OPT across five cache sizes on our first CDN production trace. All upper bounds are shown with a bar extending above OPT, and all lower bounds are shown with a bar extending below OPT. Note that the practical offline upper bounds (e.g., Belady) do not have corresponding lower bounds. Likewise,

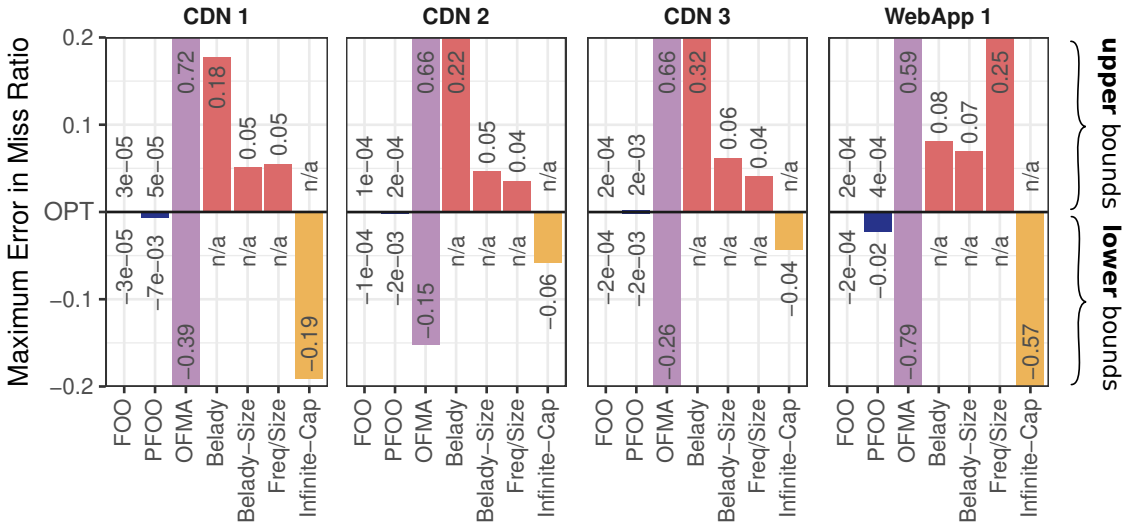


Figure 4.17: Approximation error of FOO, PFOO, and several prior offline bounds on four of our eight production traces (Figure 4.18 shows the other four). FOO and PFOO's lower and upper bounds are orders of magnitude better than any other offline bound. (See Figure 4.16.)

there is no upper bound corresponding to Infinite-Cap. Also note that OFMA's bars are so large than they extend above and below the figure. We have therefore annotated each bar with its absolute error from OPT.

The figure shows that FOO-U and FOO-L nearly coincide, with error of 0.00003 (=3e-5) on this trace. PFOO-U is 0.00005 (=5e-5) above OPT, nearly matching FOO-U, and PFOO-L is 0.007 below OPT, which is very accurate though worse than FOO-L.

All prior techniques yield error several orders of magnitude larger. OFMA has very high error: its bounds are 0.72 above and 0.39 below OPT. The practical upper bounds are more accurate than OFMA: Belady is 0.18 above OPT, Belady-Size 0.05, and Freq/Size 0.05. Finally, Infinite-Cap is 0.19 below OPT. Prior to FOO and PFOO, the best bounds for OPT give a broad range of up to 0.24. *PFOO and FOO reduce error by 34× and 4000×, respectively.*

Figures 4.17 and 4.18 show the approximation error on all eight production traces. The prior upper bounds are much worse than PFOO-U, except on one trace (Storage 1), where Belady-Size and Freq/Size are somewhat accurate. Averaging across all traces, PFOO-U is 0.0014 above OPT. PFOO-U reduces mean error by 37× over Belady-Size, the best prior upper bound. Prior work gives even weaker lower bounds. PFOO-L is on average 0.004 below OPT on the CDN traces, 0.02 below OPT on the WebApp traces,

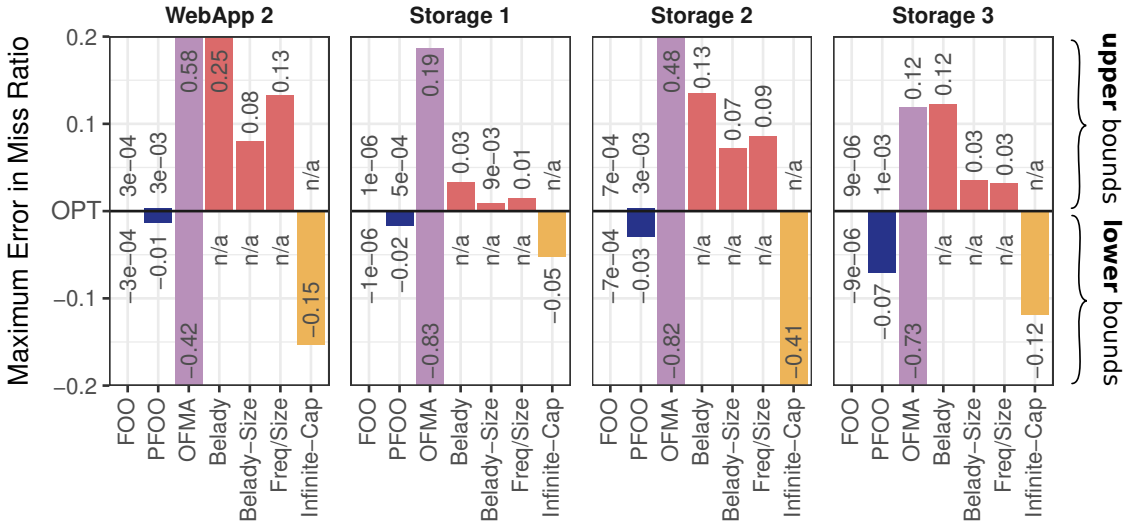


Figure 4.18: Approximation error of FOO, PFOO, and several prior offline bounds on four of our eight production traces (Figure 4.17 shows the other four). FOO and PFOO's lower and upper bounds are orders of magnitude better than any other offline bound. (See Figure 4.16.)

and 0.04 below OPT on the storage traces. PFOO-L reduces mean error by 9.8× over Infinite-Cap and 27× over OFMA. Hence, across a wide range of workloads, PFOO is by far the best practical bound on OPT.

4.6.3 PFOO is accurate on real traces

Now that we have seen that FOO is accurate on short traces, we next show that PFOO is accurate on long traces. Figures 4.19 and 4.20 show the miss ratio over the full traces achieved by PFOO, the best prior practical upper bounds (one of Belady, Belady-Size, and Freq/Size), the Infinite-Cap lower bound, and the best online policy (see Section 4.5).

On average, PFOO-U and PFOO-L bound the optimal miss ratio within a narrow range of 4.2%. PFOO's bounds are tighter on the CDN and WebApp traces than the storage traces: PFOO gives an average bound of just 1.4% on CDN 1-3 and 1.3% on WebApp 1-2 but 5.7% on Storage 1-3. This is likely due to error in PFOO-L when requests are highly correlated, as they are in the storage traces (see our trace characterization in Subsection 4.5.1).

Nevertheless, PFOO gives much tighter bounds than prior techniques on every trace. The prior offline upper bounds are noticeably higher than PFOO-U. On average, com-

pared to PFOO-U, Belady-Size is 19% higher, Freq/Size is 22% higher, and Belady is fully 72% higher. These prior upper bounds are not, therefore, good proxies for the offline optimal. Moreover, the best upper bound varies across traces: Freq/Size is lower on CDN 1 and CDN 3, but Belady-Size is lower on the others. Unmodified Belady gives a very poor upper bound, showing that caching policies must account for object size. The only lower bound in prior work is an infinitely large cache, whose miss ratio is much lower than PFOO-L. *PFOO thus gives the first reasonably tight bounds on the offline miss ratio for real traces.*

4.6.4 PFOO shows that there is significant room for improvement in online policies

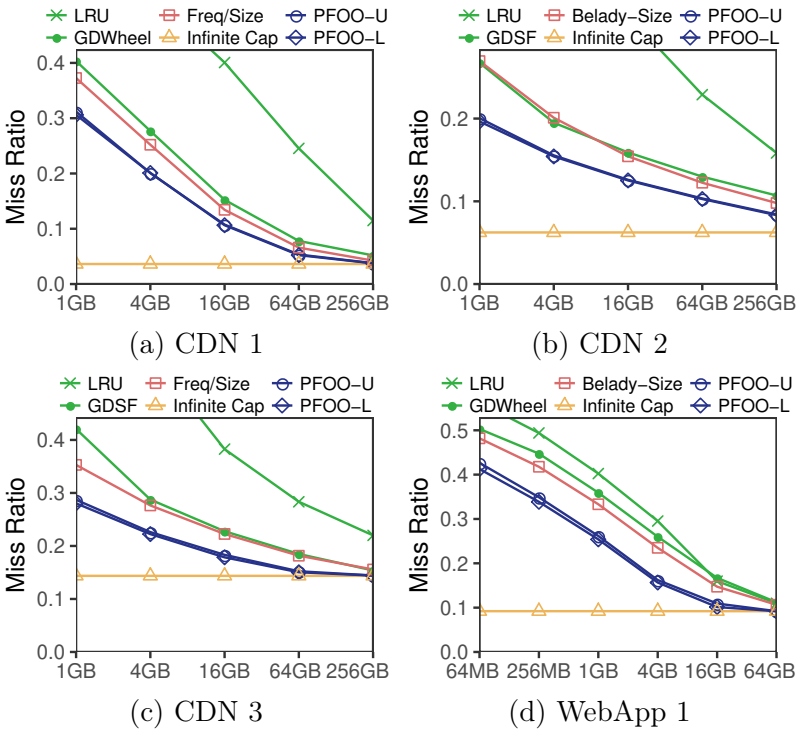


Figure 4.19: Miss ratio curves for PFOO vs. LRU, Infinite-Cap, the best prior offline upper bound, and the best online policy for the first four traces (see Figure 4.20 for the other four).

Finally, we compare with online caching policies. Figures 4.19 and 4.20 show the best online policy (by average miss ratio) and the offline bounds for each trace. We also show LRU for reference on all traces.

On all traces at most cache capacities, there is a large gap between the best online policy and PFOO-U, showing that there remains significant room for improvement in online caching policies. Moreover, *this gap is much larger than prior offline bounds would suggest.* On average, PFOO-U achieves 27% fewer misses than the best online policy, whereas the best prior offline policy achieves only 7.2% fewer misses; the miss ratio gap between online policies and offline optimal is thus 3.75× as large as implied by prior bounds. The storage traces are the only ones where PFOO does not consistently increase this gap vs. prior offline bounds, but even on these traces there is a large difference at some sizes (e.g., at 64 GB in Figure 4.20c). On CDN and WebApp traces, the gap is much larger.

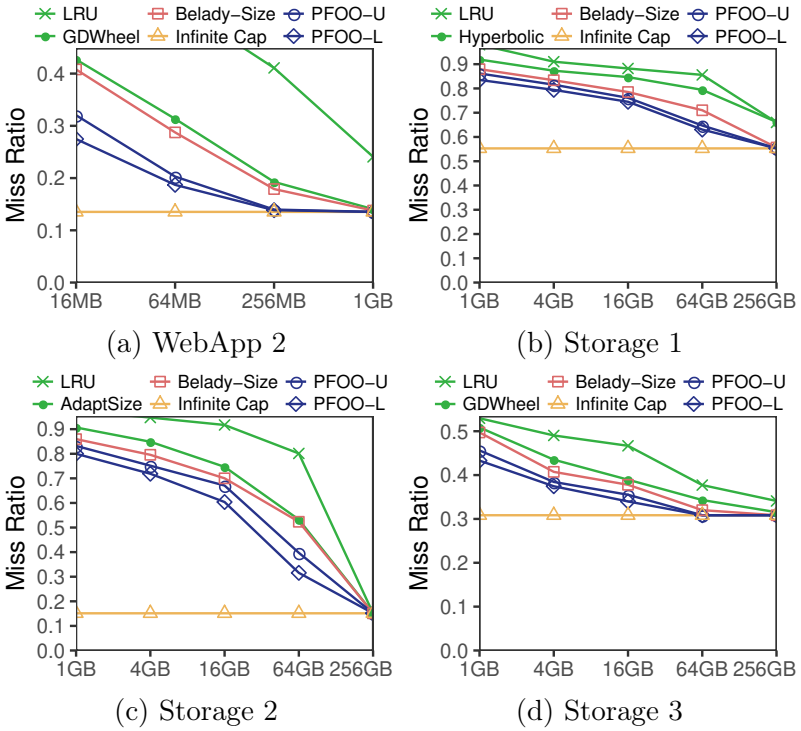
For example, on CDN 2, GDSF (the best online policy) matches Belady-Size (the best prior offline upper bound) at most cache capacities. One would therefore conclude that existing online policies are nearly optimal, but PFOO-U reveals that there is in fact a large gap between GDSF and OPT on this trace, as it is 21% lower on average.

These miss ratio reductions make a large difference in real systems. For example, on CDN 2, CDN 3, and WebApp 1, OPT requires just 16 GB to match the miss ratio of the best prior offline bound at 64 GB (recall that the x-axis in these figures is shown in log-scale). Prior bounds thus suggest that online policies require 4× as much cache space as is necessary.

4.7 Summary

We began this chapter by asking: *Should the systems community continue trying to improve miss ratios, or have all achievable gains been exhausted?* We have answered this question by developing new techniques, FOO and PFOO, to accurately and quickly estimate OPT with variable object sizes. Our techniques reveal that prior bounds for OPT lead to qualitatively wrong conclusions about the potential for improving current caching systems. Prior bounds indicate that current systems are nearly optimal, whereas PFOO reveals that misses can be reduced by up to 43%.

This chapter introduces the first principled way to evaluate caching policies with variable object sizes: FOO gives the first asymptotically exact, polynomial-time bounds on OPT, and PFOO gives the first practical and accurate bounds for long traces. Furthermore, our results are verified on eight production traces from several large Internet companies, including CDN, web application, and storage workloads, where FOO reduces



(a) WebApp 2 (b) Storage 1

(c) Storage 2 (d) Storage 3

Figure 4.20: Miss ratio curves for PFOO vs. LRU, Infinite-Cap, the best prior offline upper bound, and the best online policy for the first four traces (see Figure 4.19 for the other four).

approximation error by 4000×. We anticipate that FOO and PFOO will prove important tools in the design of future caching systems.

5

RobinHood: a tail latency aware cache partitioning system

Contents

In previous chapters we have seen that adaptive caching systems can significantly improve cache hit ratios, and we have studied optimal cache hit ratios. This chapter extends the idea of adaptive caching to a different performance metric: *request tail latency*. Specifically, we are interested in the 99th-percentile request latency (P99) in systems where requests are composed of many subqueries.

We analyze the example of a modern webservice at Microsoft. The OneRF page rendering framework serves a wide range of content including news (microsoft.com) and an online retail software store (xbox.com). This system relies on more than 20 backend systems, such as product catalog databases and recommender systems. As shown in Figure 5.1, each user *request* is received by an aggregation server, which then sends *queries* to the necessary backends, waits for all queries to complete, and then packages the results for delivery back to the user. Other large webservices like Wikipedia [24], Amazon [142], and Facebook [143] have all followed a similar design. Note that a request in these systems is not considered complete until all queries have completed.

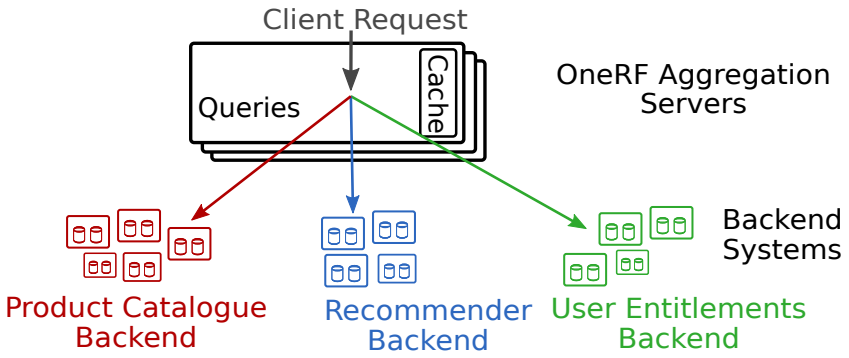


Figure 5.1: In the OneRF aggregation system at Microsoft, a user *request* requires aggregating data from various *backend services* by issuing a series of *queries*. The request is only considered to be completed when all subqueries have finished.

In this chapter we ask the question whether the aggregation caches (colocated with aggregation servers) can help to reduce the P99 request latency. Traditionally, aggregation caches are very simple. At Microsoft, they are unmanaged caches with a single LRU eviction queue. In other systems, such as at Wikipedia and Facebook, the aggregation caches are statically partitioned into separate caches, each tasked with caching queries for a specific backend.

We propose *RobinHood*, which is an adaptive caching system that *dynamically partitions* an aggregation cache with the goal of *minimizing request tail latency*. Section 5.1 motivates the problem by showing that existing caching systems are not effective in

reducing tail latency. Section 5.2 then introduces our proposed RobinHood caching system, and discusses some of our experiences that have affected its design. Section 5.3 describes the implementation of RobinHood and of our evaluation testbed. Our empirical results are presented in Section 5.4. Section 5.5 summarizes the results from this chapter.

5.1 Background and Motivation

It is common for large Internet companies to support a wide variety of webservices which all make use of a common set of backend services. Several Microsoft storefront properties[10] share more than 20 backend services. These storefronts use a common aggregation service, the *OneRF* system (see Figure 5.1), which translates a user request into backend queries and aggregates the results. Such aggregation services are common in *multi-tiered systems* [142].

In such a system, a request first arrives at an *aggregation server*, where it is broken into its component queries. Queries are first looked up in a *local cache*, which is collocated with the aggregation server. The aggregation server then dispatches the remaining queries (cache misses) to the appropriate *backend*. A request is considered complete when each query has either been found in the cache or retrieved from a backend. Hence, latency of a request is defined as the maximum of its query latencies.

The goal of RobinHood. RobinHood aims to minimize the tail latency of requests in a multi-tiered system by dynamically allocating cache space to the backends which cause high-latency requests. While RobinHood can optimize any latency percentile, we will focus on minimizing the 99-percentile request latency (P99) throughout this chapter.

We first discuss the intuition behind RobinHood (Subsection 5.1.1) and then the challenges in achieving RobinHood's goal (Subsection 5.1.2).

5.1.1 How does Caching Address Tail Latency?

Caching is widely used to shield backends from overload [144] and to improve average-case performance [128, 145]. However, once a backend is within its capacity region [146] further *improving cache hit ratios is widely considered to not help tail latency latency* [147]. This perspective is often explained with a simple example. Let's say that a cache responds within 1ms and has hit ratio 70%, while its backend responds to cache misses

[10] microsoft.com, xbox.com, onestore.com

within 50ms. In this case, the P99 is 50ms. From this perspective, the P99 will be 50ms for any cache with a hit ratio below 99%, which is typical for web and datacenter cache hit ratios [31, 144]. Thus, it seems that in practice caches cannot improve the P99.

RobinHood uses a cache to significantly improve the P99 in systems like OneRF, where requests depend on queries to many backends. The intuition behind this counter-intuitive result relies on three observations.

Observation 1: backend latency is not constant and not correlated with individual queries. Unlike the example above, backend query latency is highly variable in practice, typically spanning more than an order of magnitude. In OneRF's backends and in other large web backends [146], this variance in query latency is not correlated to particular "slow queries", but rather reflects a more holistic state of the backend system at some point in time. Figure 5.2 shows latency scatterplots ordered by the popularity of the underlying query. We observe that high latency is neither correlated with a specific query nor with the query's popularity. We thus think of each backend query latency as a sample from some distribution which reflects the state of the backend system.

Even small increases in the hit ratio (10-20%) will result in fewer queries to the backend system and consequently fewer samples from the query latency distribution. This reduces the probability of sampling at least one high-latency query[11]. Hence, by increasing the cache hit ratio of a backend, we are able to decrease the total number of high latency queries, which may improve request tail latency.

Observation 2: higher cache hit ratios reduce backend load. Most production systems are run at low load to maintain low tail latency since queueing effects often cause latency to increases quickly for loads above 50% [146–148]. Higher cache hit ratios also decrease load, which typically improves latency across a wide range of percentiles. In many backend systems such as OneRF, backends can become temporarily overloaded (see Figure 1.5). It is at these moments that small reductions in backend load can have an outsized impact on the tail latency. Caching can provide a flexible mechanism for providing this temporary relief.

Observation 3: in multi-tiered architectures with many backend systems, caches can act as a load balancer. Tail request latency in the OneRF system would benefit greatly from the ability to balance load between backend systems. While load would have to be balanced carefully to account for request structure, load balancing

[11] Consider the following toy example. A backend query takes 10ms 95% of the time, and 50ms 5% of the time. Then, increasing the hit ratio from 70% to 85% means the probability of a query taking 50ms goes down from 1.5% to 0.75% (so the P99 goes down from 50ms to 10ms).

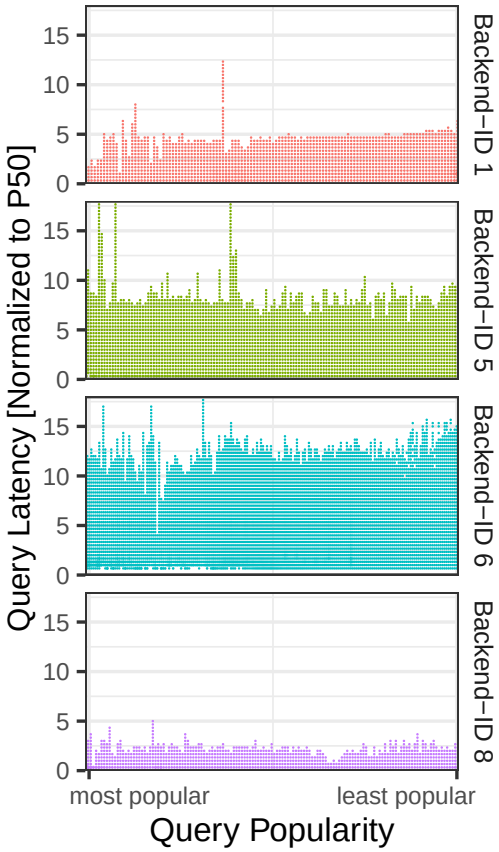


Figure 5.2: Scatterplot of query latency and popularity. We find that query latency is neither correlated with popularity or a particular query.

could be used to make the slowest backends in the system faster without increasing the latency of the fast, lightly-loaded backends too much. Unfortunately, traditional load balancing cannot balance load between backends, since every query must be sent to its correct, corresponding backend. The caching layer, however, can shift cache capacity from backends that do not affect the P99 request latency to those that do. According to Observation 2, this will raise the cache hit ratio and lower the load of the beneficiary backends and have the inverse effect on their benefactors. RobinHood exploits this phenomenon and tries to balance load across dissimilar backend systems by continuously reallocating cache space.

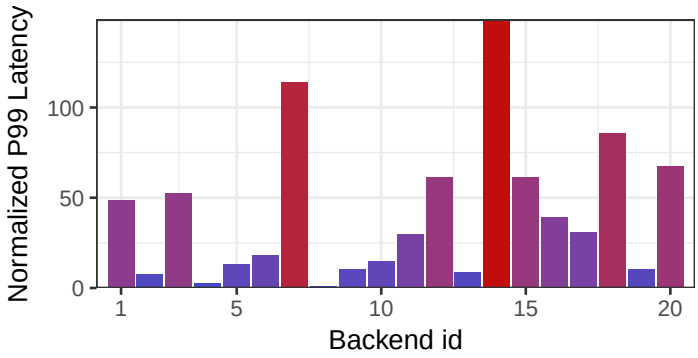


Figure 5.3: Backend latencies in OneRF production backend systems.

5.1.2 Key Challenges of Caching for Tail Latency

We analyze traces from a OneRF production cluster to diagnose why achieving low tail latency in a multi-tiered system is difficult. We identified the following three key challenges.

Latency is imbalanced and the imbalance changes over time

As noted previously, it is common for the latencies of different backends to vary widely.

In OneRF, Figure 5.3 shows that latency across the 20 most used backends varies by more than two orders of magnitude. The fundamental reason for this latency imbalance is that several of these backend systems are complex multi-tiered distributed systems in their own right. They serve multiple customers within the company, not just OneRF. In addition to high latency imbalance, backend latencies also change over time (see Figure 5.4) independently of the request stream seen by applications servers.

Why latency imbalance poses a challenge for existing systems. Existing caching systems take a myopic view of performance in that they focus on optimizing cache-centric metrics (e.g., hit ratio) instead of latency. For example, a common approach is to divide fairly the cache space between partitions [149]. Another common approach is to allocate cache space to balance hit ratios or to maximize the marginal gain in hit ratio [145]. These approaches fail to explicitly account for latency, and hence would only work if backends have identical latencies.

Some production systems use static cache allocations, e.g., the "arenas" in Facebook's TAO [143]. Manually deriving the optimal allocation is challenging and mentioned as an open problem [143]. To actually minimize request tail latency, any solution must not

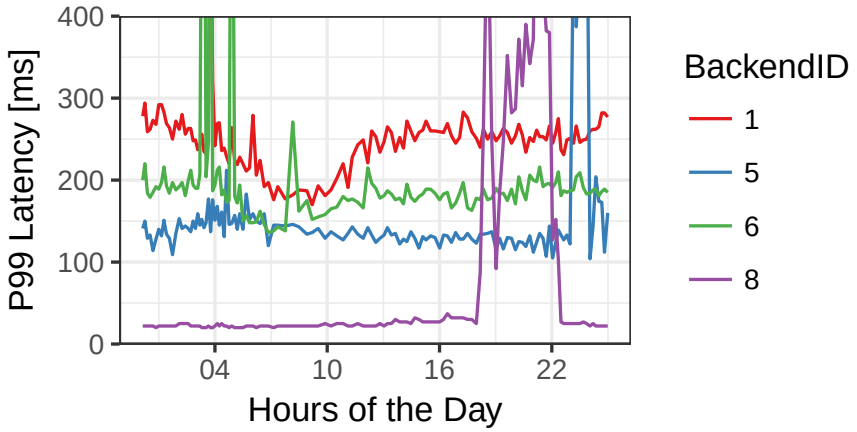


Figure 5.4: Per-backend system P99 latency over the course of a typical day in the OneRF production system.

only consider which backends will tend to be the slowest, but it must be able to adapt as this latency imbalance changes.

Latency is not correlated with query rate

Many caching schemes (including the OneRF production system) share the cache space among the backends and use a common eviction policy (such as LRU). Shared caching systems effectively give more cache space to backends that have a higher query rate. Intuitively, this occurs because backends that have a higher query rate have more opportunities to their objects admitted into the cache. Unfortunately, query rate is not necessarily correlated with latency. Figure 5.5 shows the query rate per backend (BackendIDs are ordered by rate), and Figure 5.3 shows the latency per backend. We find that query rate is typically not correlated with P99 query latency. For example, the seventh most popular backend receives only about 0.04x as many queries as the most popular backend but has 2x the latency.

Why uncorrelated latency poses a challenge for existing systems. Many caching systems allocate cache space proportionally to query rate, which is a known problem in practice [150]. This applies to LRU, as used by OneRF, and many caching systems [16]. Since latency is not correlated with query rate, such caching systems will not be effective in reducing request tail latency.

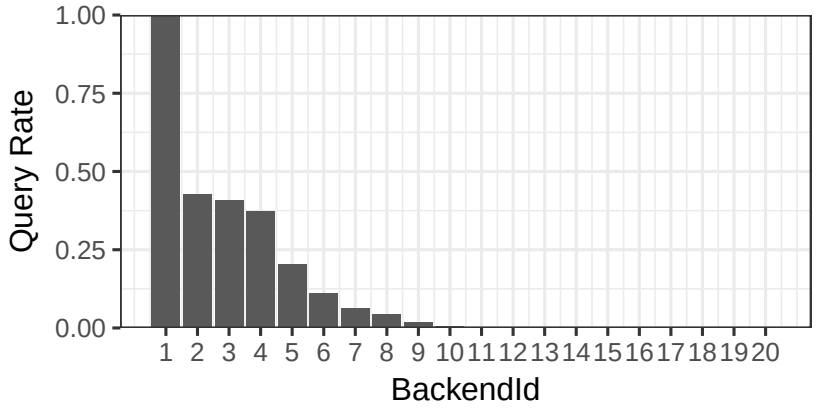


Figure 5.5: The query rate per backend in the OneRF production system.

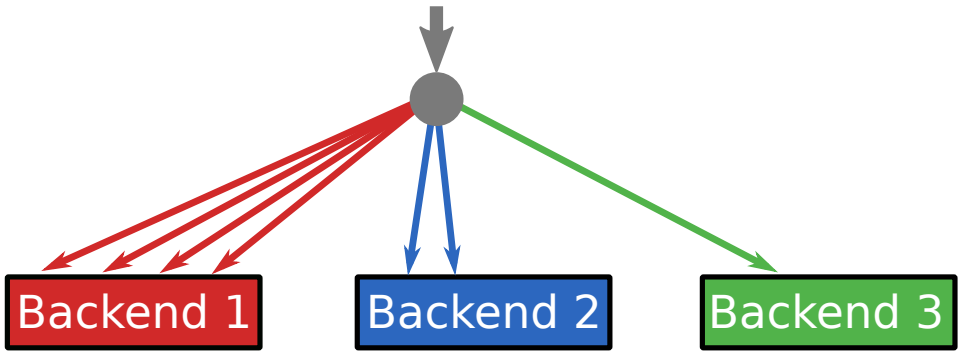


Figure 5.6: Request structure example. This request has 7 queries and fanout 3 (queries three distinct backends). The backend's batch sizes are 4 (Backend 1), 2 (Backend 2), and 1 (Backend 3).

Latency Depends on Request Structure, Which Varies Greatly

The manner in which an incoming request is split into backend queries by the application varies between requests. We call the mapping of a request to its component backend queries the *request structure* (e.g., Figure 5.6).

To characterize the request structure, we define the number of parallel queries to a single backend as the backend's *batch size*. For example, Backend 1 in Figure 5.6 has batch size 4. We define the number of distinct backends queried by a request as its *fanout*. For example, the request in Figure 5.6 has fanout 3. For a given backend, we measure the average batch size and fanout of requests which reference this backend.

Table 5.1 summarizes how the query traffic of different backends is affected by the request structure. We list the percentage of the overall number of queries that go to each backend, and the percentage of requests which reference each backend. Furthermore, we

ID	Query %	Request %	Mean Batch Size	Mean Fanout Width
1	37.7%	14.7%	15	5.6
2	16.0%	4.5%	32	7.4
3	15.3%	4.5%	26	7.4
4	14.0%	20.0%	2	4.8
5	7.7%	19.0%	2	4.9
6	4.2%	4.7%	15	7.3
7	2.4%	10.8%	2	5.3
8	1.6%	15.5%	1	5.3
9	0.7%	3.4%	2	7.5
10	0.2%	0.7%	3	9.1

Table 5.1: Four key metrics describing the 10 most popular OneRF backends. Query % describes the percentage of the total number of queries directed to a given backend. Request % denotes the percentage of requests with at least one query to the given backend. Batch size describes the average number of simultaneous queries made to the given backend across requests with at least one query to that backend. Fanout width describes the average number of backends queried across requests with at least one query to the given backend.

list the batch size and fanout by backend. We can see that all of these metrics vary across the different backends and are not strongly correlated with each other.

Why request structure poses a challenge for existing systems. There are few caching systems that incorporate latency into their decisions, and they consider the average query latency as opposed to the tail request latency [33, 134]. We find that even after changing these *latency aware* systems to measure the P99 query latency, they remain ineffective.

These systems fail because a backend with high query latency does not always cause high request latency. A simple example would be high query latency in backend 10. As backend 10 occurs in less than 0.7% of all requests, its impact on the P99 request latency is limited – even if backend 10 was arbitrarily slow, it could not be responsible for *all* of the requests above the P99 request latency. A scheme that incorporates query rate and latency might decide to allocate most of the cache space towards backend 10, while not improving the P99 request latency. While this specific case might be simple to detect, differences in batch sizes and fanout-widths give rise to complicated scenarios[12]. As a consequence, minimizing request tail latency is difficult unless request structure is explicitly considered.

[12]For example, in Table 5.1, backend 3's query latency is often high (due to large batch sizes). In comparison, backend 4 often has lower query latency, but occurs in 4.5× more requests, which makes it more likely to affect the P99 request latency. In addition, backend 4 occurs in requests with a 55% smaller fanout width, which makes it more likely to be actually the slowest backend, whereas backend 3's query latency is frequently hidden by queries to other backends.

5.2 The RobinHood Caching System

The purpose of this section is to describe the design of the RobinHood caching system, whereas implementation details are discussed in Section 5.3. We describe the basic RobinHood algorithm (Section 5.2.1), how we accommodate real-world constraints (Section 5.2.2), and the high-level architecture of RobinHood (Section 5.2.3).

5.2.1 Basic RobinHood algorithm

We first describe the high-level idea behind RobinHood's adaption algorithm. Given some target percentile, e.g., the P99, RobinHood reallocates cache space towards the backends that are responsible for high P99 request latency. We call such backends "resource poor". RobinHood repeatedly identifies the resource poor backends, taxes every backend with 1% of its cache space, and redistributes the pooled tax to resource poor backends.

Now, we discuss how RobinHood identifies resource-poor backends. First, we identify the set S of requests whose latency exceeds the P99. For each request in S, we then determine the query that took the longest, blocking the completion of this request. Then, we tally the number of times each backend contributed a blocking query in S. A backend's tally is called the request-blocking-count (RBC).

A backend's RBC is a measure of how poor a resource is. RobinHood thus distributes the pooled taxed to all backends in proportion to the RBC.

As request structures and backend loads change over time (see Section 5.1), RobinHood continuously repeats this algorithm every Δ seconds We currently use $\Delta = 5$ seconds.

5.2.2 Accommodating Real-World Constraints in RobinHood

Next, we discuss three key constraints that have shaped the design and the implementation of RobinHood.

Backends appreciate the loot differently. The basic RobinHood algorithm assumes that redistributed cache space is used immediately by each backend's queries. In reality, some backend's queries do not use the additional cache space because their working set already fits into the cache. RobinHood detects this phenomenon by monitoring for each backend the gap between the assigned and the used cache capacity. If this gap is more

than a safety margin of 30%, RobinHood temporarily ignores RBC of this backend to avoids wasting cache space. Note that such a backend may continue to affect the request tail latency. However, RobinHood instead focuses on backends that are currently more receptive to additional cache space.

Local decision making and distributed controllers. The basic RobinHood algorithm assumes an abstraction of a single cache with one partition per backend. In reality, e.g., in the OneRF system, incoming requests are load balanced across a cluster of aggregation servers, each of which has its own local cache (see Section 5.1). Due to randomness[13], each server may see a slightly different view of the request and query streams. For example, the speed at which different backend queries claim new cache space slightly differs across aggregation servers. So, instead of making a global decision of how to reallocate cache space (which may be suboptimal on some aggregation servers), we make cache allocation decisions locally on each aggregation server. Thus, RobinHood runs as a distributed controller as shown in Section 5.2.3.

One might think that the choice of distributed controllers could lead to diverging allocations and cache space fragmentation across application servers over time. However, we can show that over time, the difference in allocation is not larger than the difference in working sets. Specifically, given $\Delta = 5$ seconds, any application server (e.g., a newly started one) will converge to the average allocation within 30 minutes for all partitions that see sufficient traffic to fill the caches.[14]

Honing the definition of the P99. The basic RobinHood algorithm assumes that the slowest 1% of requests (which includes the P99.9 and the P99.99) are representative of the P99. In reality, it is well known that the highest percentiles, e.g., the P99.99, include non-representative outliers [142]. Rather than catering to these outliers, RobinHood focuses on the region around the P99, specifically the P98.5 to the P99.5, which better reflects the requests affecting the P99 in the next round.

[13]The load balancer in aggregation systems is typically only aware of requests, and not aware of the queries that are triggered by the request. Therefore, differences in application server query streams are common.

[14]This assumes stationary RBCs and query rates, and this holds for an application server starting with any initial allocation and for the time until it is within 5% of the mean allocation across all application servers.

5.2.3 RobinHood Architecture

Figure 5.7 shows the RobinHood architecture. It consists of the aggregation servers and their caches, the backend services, and a statistics collection server.

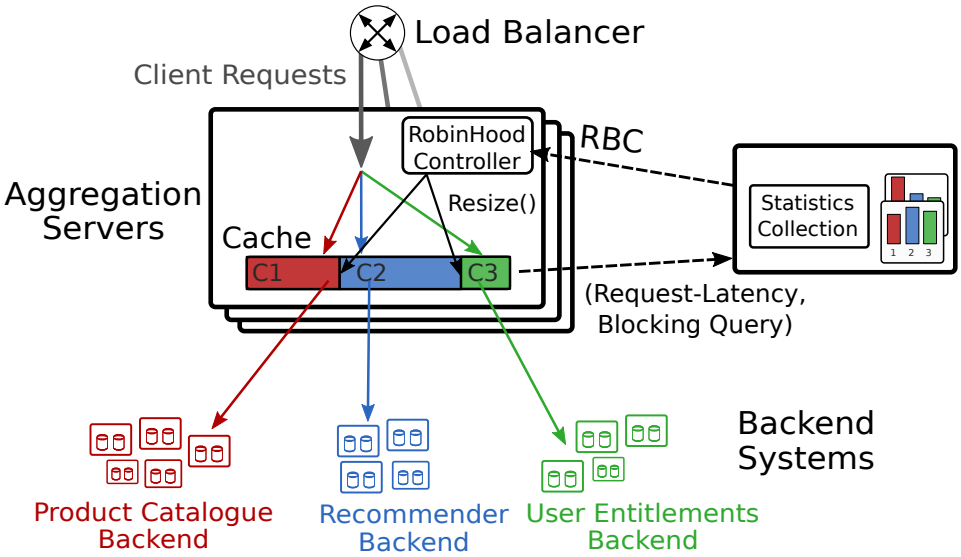


Figure 5.7: Sketch of RobinHood.

RobinHood requires an application caching system that can be dynamically resized. For example, we use an unmodified memcached instance per aggregation server in our testbed (see Section 5.3). Compared to a production aggregation system, such as OneRF, we add two components. First, we add a lightweight controller to each aggregation server. The controller implements the RobinHood algorithm (Sections 5.2.1 and 5.2.2) and issues resize requests to the local cache's partitions. Second, we add a statistics collection server (or extend an existing real-time statistics framework). The aggregation library in the aggregation server sends the latency of each request, and the blocking query's backend-ID to the statistics server. In our implementation this happens in batches, every second. The statistics server calculates the RBC based on these measurements. The controller then pulls the RBC from the statistics server and runs the RobinHood algorithm.

Notice that both RobinHood components (controller and statistics server) are not on the critical path of requests and queries, and thus do not impose any latency overhead. Moreover, both are stateless and the RobinHood architecture can tolerate faults and restarts.

5.3 System Implementation and Challenges

In designing an experimental testbed for the RobinHood Algorithm, our goal was to recreate the dynamics of the OneRF system as accurately as possible. To accomplish this, we made an effort to scale our testbed to a sufficient size that we were likely to encounter challenges similar to those faced by the OneRF system. The end result was a deployment across 50 servers that consisted of 20 backends (where several backends have replicas), and 16 aggregation servers. Indeed, scaling to this degree revealed several challenges not found in smaller scale experiments (see Section 5.3.3).

5.3.1 Generating Experimental Data

The first challenge in replicating the OneRF system is generating a realistic trace of requests to drive our experiments. This requires knowing the distribution of request structures – the relative frequency of each different way a request can be split into queries. Furthermore, for each query to a given backend, we must know the popularity distribution for this backend – the relative frequency with which each object in this backend is queried. Finally, we must know the distribution of sizes for objects in each backend.

Microsoft shared the above data with us for one of their OneRF cluster deployments in their "east-us" data center for one week in 2017 and one week in 2018. We use the 2018 data in our evaluation (Section 5.4). The OneRF production cluster aggregated queries from more than 40 distinct backend systems. We focus on the top 20 backend systems, which make up more than 99% of all queries. The technologies used for each backend include distributed key-value stores, multitiered content management systems, replicated databases, and distributed machine learning systems. In addition to the data necessary for generating the above distributions, we received per-query latency information. This allowed additional analysis of performance trends across backends (see Section 5.1).

To create the experimental request trace, we generate a sequence of 50 million requests. To generate each request, we first sample i.i.d. from the request structure distribution, revealing which backends will be queried (and how many times each will be queried) by this request. Then, for each query, we sample i.i.d. from the appropriate backend popularity distribution, defining exactly which objects will be retrieved by each query. Object sizes are generated from a size distribution independently by each backend and are not explicitly encoded in the request trace. Note that, although we draw i.i.d. from each distribution, the request structure and popularity distributions encode correlations

between queries. Hence, our generated trace will preserve correlations from the production request stream. Furthermore, the popularity distribution also preserves the unique locality and cacheability characteristics of each stream of backend queries.

5.3.2 Our Experimental Deployment

Our experimental deployment captures all of the key elements of the OneRF system described in Section 5.1. Our deployment consists of 20 distinct backend services. These services include several I/O intensive databases which store objects of the appropriate size, a key-value store which serves objects of various sizes from memory, and a CPU intensive machine-learning emulation which multiplies matrices of various sizes. Each of these backends corresponds to the databases, key-value stores, and machine learning services used by OneRF. Each backend is replicated according to its popularity relative to the other backends. We additionally deploy a requestor, which reads the experimental trace and generates requests, aggregation servers to respond to these requests, and the RobinHood statistics server which aggregates system statistics and makes them available system wide.

For database backends, MySQL was used. We wrote our own key-value store and matrix-multiply backend in Go. Similarly, we wrote the aggregation servers and the RobinHood statistics in Go. The caches on each aggregation server are memcached instances. We developed the RobinHood controller in Python, as the controller is very lightweight and not on the critical path of requests or queries.

All of these components are deployed as individual docker containers on a Microsoft Azure virtual machine scale set composed of 50 instances joined by a virtual network. Each server has 60 GB of memory; on aggregation servers 32 GB can be used by the caching system. Container management services, persistent container storage, and inter-container networking (via an overlay network) are provided by Docker Swarm. Swarm leverages IPVS advanced layer-4 load balancing to route requests between replicated container instances. On top of this framework, we built an extensive real-time monitoring and visualization framework to track various metrics such as latency, RBC, CPU load, memory usage, and disk and network I/O for each container. This extensive setup allowed us to reliably scale to production traffic speeds of several hundred thousand queries per second.

5.3.3 Implementation Challenges

The biggest challenges in implementing this experimental setup was associated with the high number (20) of backend services we had incorporated. An astute reader may ask why we eschew the use of a remote, distributed caching layer, which is the design choice in some production systems. We began by building a system which used a distributed caching layer to support only 4 backends. When we increased the number of backends, we began to see some of the issues that have been described in the literature on distributed caching, such as hotspots and the distribution of correlated data across cache instances [143, 146]. While we acknowledge that solutions to these problems exist, they are non-trivial and outside the scope of this work. We thus reverted to our design of placing local, redundant caches on each aggregation server. This had the additional benefit of removing network latencies from our measurement of cache hit times. This matches, in fact, the design used in the OneRF production system.

Another challenge we encountered was the contention caused by reallocating cache space in memcached instances. While we were able to implement the aggregation server cache on top of unmodified memcached, we find that reallocating partition sizes while under load can be expensive. For example, a deallocation of space requires acquiring several locks to evict safely a potentially high number of pages to free up space. Hence, we built our controllers to tolerate significant contention on these resources. The controller makes a best-effort to enforce their desired allocation but will defer enforcement to a future iteration of the algorithm if there is contention on cache resources is too high.

Finally, we expended significant effort creating a reproducible experimental setup. Given the complexity of the deployment, and the variable nature of the cloud platform on which we run, this required extensive monitoring. We also developed complex procedures for resetting the state of the system between experimental runs to clear all system buffers and prevent any carried over optimizations attempted by the underlying hardware. This included a full restart of every instance between runs.

5.4 Empirical Evaluation

This section presents empirical results on the performance of RobinHood and existing state of the art caching systems. The experiments use 50 servers, production traces from the OneRF system, and the experimental testbed described in Section 5.3. We first describe the competing caching systems (Section 5.4.1). Next, we present our results from microexperiments on the impact of latency imbalance on these systems

(Section 5.4.2). Finally, we discuss the results from scaling up the experiments to 20 backend systems (Section 5.4.3).

5.4.1 Competing Caching Systems

We compare RobinHood to the state-of-the-art caching systems described in Section 5.1. In total our experiments contain up to:

RobinHood: Our proposed dynamic partitioning system, using the design discussed in Section 5.2, and the implementation discussed in Section 5.3.

Shared-Cache: A single shared cache partition with LRU eviction. This resembles most closely the OneRF production configuration.

Static-Partition: Static per-backend partitions, where the partition sizes are optimized once and then kept static (e.g., optimized for one part of the trace). This is an optimistic (due to the recent optimization) representation of the Facebook TAO configuration [143]

Offline-Opt: An offline-optimized partitioning scheme that has knowledge about future requests and load patterns. We created this policy's allocation by brute-force searching over the space of allocation over the course of several days. This is an impractical policy in general, and even with future knowledge we were only able to implement this scheme for up to four partitions.

FairSpace (Static): Another static per-backend partitioning scheme, where each partition is assigned the same share of the cache. Fair sharing has been proposed (in the different context of sharing cluster computing caches) in FairRide [149].

By-Latency: A dynamic partitioning system that allocates in proportion to the P99 latency of each backend. This has not previously been proposed in the literature but is similar in spirit to the recently-proposed Hyperbolic system [134]. As in RobinHood, By-Latency taxes all backends every $Delta = 5$ with 1%. In our first implementation of ByLatency, we just allocated to the slowest P99, which worked for 4 backends but not for 20 (too slow). To fix this problem, ByLatency now allocates to all backends that are above the average P99 across the backends, in proportion to how much they exceed the average.

[15]We also tested a scheme that allocates with the goal of maximize the overall hit ratio, but that this

By-HitRatio: A dynamic partitioning system that allocates with the goal of equalizing hit ratios[15]. This is similar to Cliffhanger [145].

By-QueryRate: A dynamic partitioning system that allocates in proportion to a backend's query rate

The following two sections use a subset of these policies, where we selected the policies by feasibility.

5.4.2 Latency-Imbalance Microexperiments

A key feature of production aggregation systems, like OneRF, is a severe degree of latency imbalance (see Section 5.1). In this section, we evaluate how different caching systems react to latency imbalance. We induce this latency imbalance by limiting the resource of a backend system[16]. To isolate the effect of load imbalance, we consider only four backends and induce latency imbalance into only one of the four backends. Specifically, we randomly picked BackendIDs 1,4,7,9 and induce latency imbalance into BackendID 7 to reproduce a scenario similar to Figure 5.3, where latency is uncorrelated with query rate (see Section 5.1).

We evaluate three levels of latency imbalance: "mild", "moderate", and "severe". Under mild latency imbalance, almost all backends have the same latency (there is some imbalance due to randomness in the backend architectures). Under moderate latency imbalance, BackendId 7 has 10x the latency as BackendsIds 1,4, and 9. Under severe latency imbalance, BackendId 7 has 100x the latency as BackendsIds 1,4, and 9. An experiment is performed by gradually increasing the imbalance to the target value over the course of 15 minutes and is then kept stable over the course of 45 minutes. This is repeated at all three levels, for RobinHood, OfflineOpt, By-Latency, FairSpace, By-HitRatio, Shared, and By-QueryRate.

Figure 5.8 shows a box plot of the P99 request latency in this experiment, for each caching policy, at all three levels of latency imbalance. We find that OfflineOpt, RobinHood and By-Latency are the only policies whose P99 latency remains below 100ms across all three levels of latency imbalance. All other policies are not robust under latency imbalance.

alternative scheme performed always worse than Shared-Cache and is thus represented by Shared-Cache.

[16]Using Linux Control Groups, we limit how many IOPS and how much CPU time a set of backend servers is allowed to use.

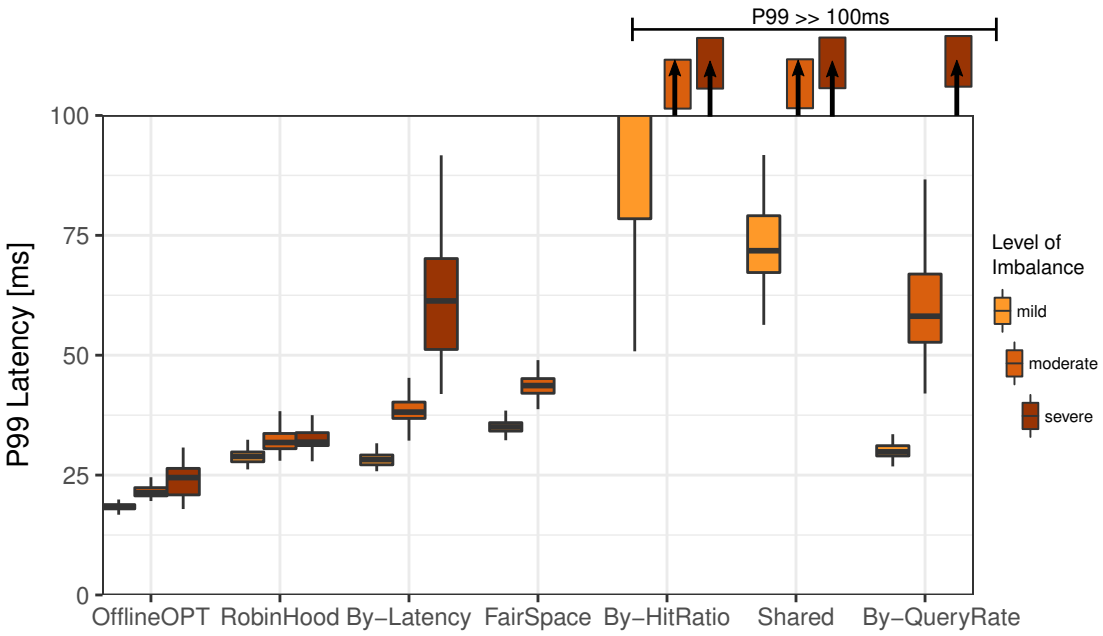


Figure 5.8: P99 request latency in the latency imbalance microexperiment as a boxplot: bold black line indicates median, box indicates 25/75-percentiles, whiskers indicate 10/90-percentiles.

When comparing RobinHood and By-Latency, we find that their P99 latency is very similar under mild latency imbalance. However, under moderate latency imbalance the P99 latency of By-Latency grows significantly, whereas RobinHood's P99 remains low. Under several latency imbalances, the median P99 latency of By-Latency is more than 70% higher than RobinHood's P99. The P99 of By-Latency is also much more variable, whereas RobinHood's P99 is tightly concentrated.

When comparing RobinHood and OfflineOpt, we find that OfflineOpt's P99 is 30-50% better than RobinHood. This happens because OfflineOpt has prior knowledge of the increase in load imbalance and does not need to adjust its partitions, and is thus not subjected to fluctuations in the partition sizes. This allows OfflineOpt to more efficiently use its cache capacity. In contrast to OfflineOpt, RobinHood is a practical policy. However, we find that there is significant potential for future improvements in the RobinHood design.

5.4.3 Scaled-Up Experiments

In this experiment, we scale up the number of backends from 4 to 20 different backend systems, which matches the OneRF production system. We also limit the resources on

several groups of backends in order to create a scenario with dynamic latency imbalances similar to the period between 19:00 and 24:00 on the day of our production trace (see Figure 5.4 in Section 5.1). Our experiment are about four hours long, and we discard the 25 minutes of each experiment (so, we are showing a total of 13K seconds or 216 minutes of experiment time). The experiment includes the following policies: RobinHood, By-Latency, Static, Shared, By-HitRatio, and By-QueryRate. We excluded FairSpace, as it leads to an unstable system, and we excluded OfflineOpt as it was impossible to brute-force search a 20-dimensional space of partition sizes. Static was optimized for the first hour of the experiment, starting with the configuration found by RobinHood.

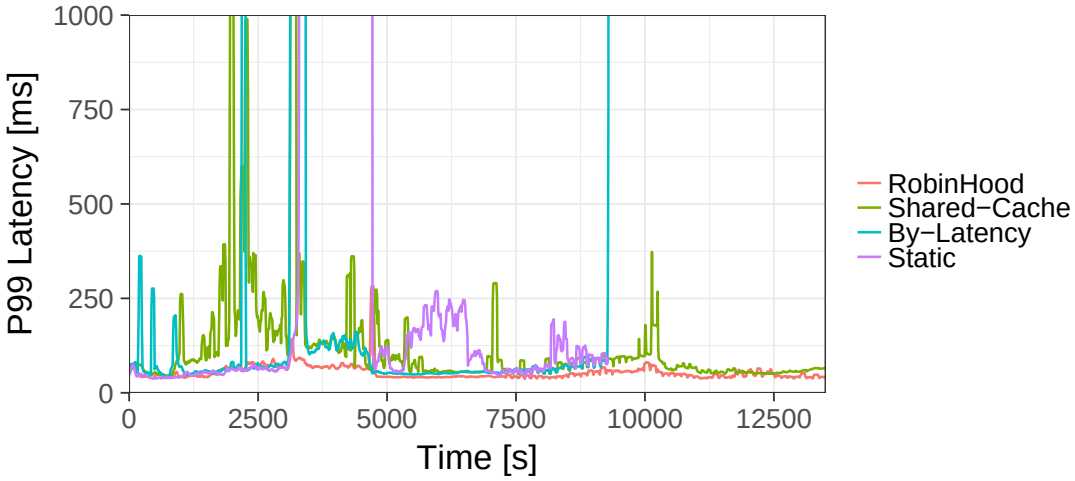


Figure 5.9: Comparison of the P99 request latency of RobinHood, Shared-Cache, and By-Latency allocation.

Figure 5.9 shows the P99 request latency over time for RobinHood, Shared, By-Latency and Static. We find that RobinHood is the only policy whose P99 stays below 100ms throughout the experiment.

Shared shows high latency early in the experiment, between 0 and 5000 seconds. Shared peaks again around 10000 seconds.

By-Latency generally works well in the first half of the experiment. There are several small P99 peaks at the beginning, and a prolonged period of around 125ms P99 latency between 2500 and 5000 seconds. In the second half of the experiment, By-Latency becomes unstable around 9000 seconds, and does not recover.

Static does very well in the first hour of the experiment (as expected), where it's P99 is 5-10% better than RobinHood's P99. However, Static is unstable between 3000 and 5000 seconds, and after 9000 seconds.

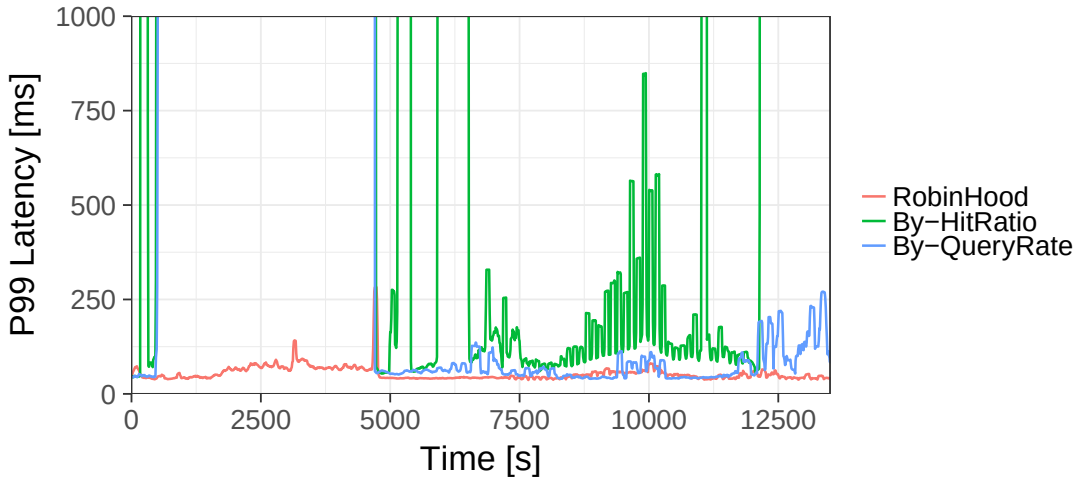


Figure 5.10: Comparison of the P99 request latency of RobinHood, By-HitRatio, and By-QueryRate allocation.

Figure 5.10 shows the P99 request latency over time for RobinHood, By-HitRatio, and By-QueryRate. We find that, of these three, RobinHood is the only policy that is stable between 0 and 5000 seconds. By-Hit-Ratio also leads to very high P99 latency in the second half of the experiment. By-QueryRate performs better in the second half of the experiment, but still occasionally peaks above 200ms.

We summarize the performance of all six policies throughout the whole experiment as a boxplot in Figure 5.11.

Compared to Shared, RobinHood improves the P99 by 57% in the median. RobinHood also leads to a much more stable performance, as it improves by more than 2.7x the 90-percentile of the P99 over the course of the experiment.

Compared to By-Latency, RobinHood improves the P99 by 58% in the median. Due to By-Latency leading to unstable performance in part of the trace, RobinHood's improvement is more than 10x for higher percentiles. Compared to Static, RobinHood improves the P99 by 56% in the median. Again, the improvement is even more significant at higher percentiles. Compared to By-HitRatio, RobinHood's improvement is more than 2x at all percentiles. Compared to By-QueryRate, RobinHood's improvement is 60% in the median, and more than 3x at the 75-th and 90-th percentiles.

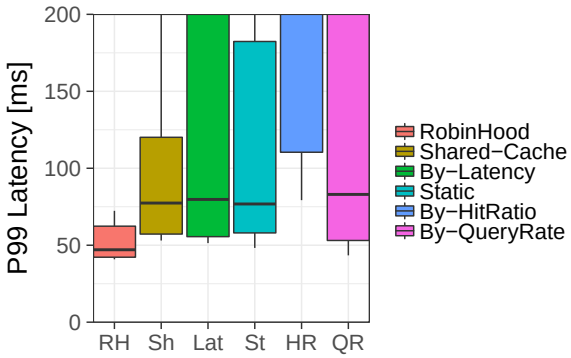


Figure 5.11: P99 request latency in the experiment in Figure 5.9 as a boxplot: bold black line indicates median, box indicates 25/75-percentiles, whiskers indicate 10/90-percentiles.

We finally consider how RobinHood adjusts the partition sizes. Figure 5.12 shows the cache allocation chosen by RobinHood across all aggregation servers for all partitions at 5-second granularity, over the course of the experiment.

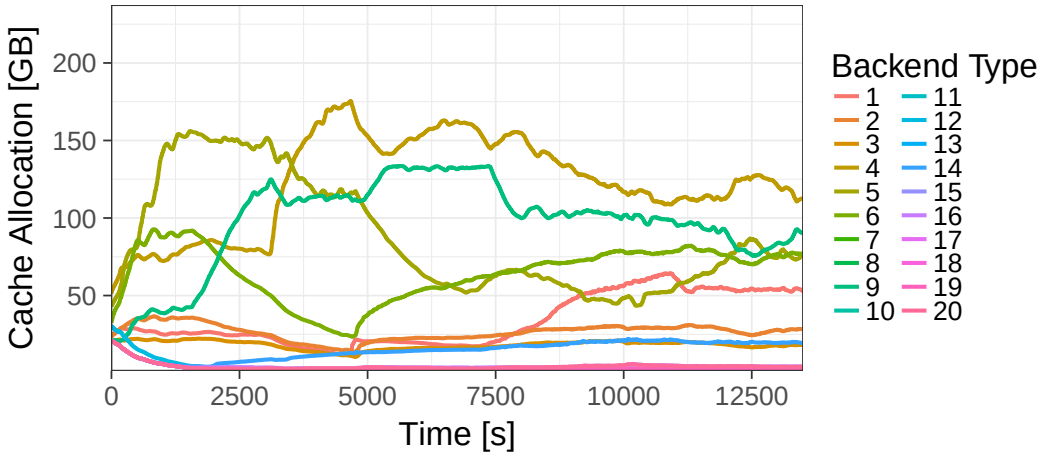


Figure 5.12: Cache allocation of RobinHood during the experiment in Figure 5.9. This is the total allocation across all 16 aggregation servers.

We find that RobinHood ramps up the cache space of BackendID 5 (which is the bottleneck is the first quarter of the experiment) from 50 GB to 150 GB and back within the first 6000 seconds of the experiment. BackendID 4 starts at 75 GB, grows up 175 GB around 5000 seconds, and then slowly decreases over time. The partition

sizes of BackendIDs 6, 9, and 1 also significantly change during the experiment. Overall, RobinHood dynamically reallocates almost 75% of its 512 GB of total cache space[17] throughout the experiment. Without the multiplexing gain (of being able to ramp-up and ramp-down partition sizes), a cache would have wasted more than 380 GB of cache space. RobinHood thus allows us aggregation server to use their cache space much more efficiently.

5.5 Summary

RobinHood is a new caching system for aggregation servers in data centers. The goal of RobinHood is to reduce tail request latency by dynamically allocating the cache resources towards backends that slow down requests. In experiments with Microsoft production traces, we show that RobinHood significantly reduces tail latency and, additionally, that tail latencies are more stable under RobinHood than in existing state-of-the-art caching systems.

The concept and results presented in this chapter disprove a widely held opinion in the literature, which is that the *caching layer does not directly address tail latency* [147]. Our design and evaluation of RobinHood has, for the first time, connected the previously separate communities on tail latency reduction and caching system optimization.

[17] As described in Section 5.3, each aggregation server allocates 32 GB of cache space. With 16 aggregation servers, there are 512 GB of cache space in total.

6

Summary & Future Work

Contents

The research presented in this thesis is motivated by the large scale of variability observed in production systems at Akamai and Microsoft. The robustness of our proposals AdaptSize (Chapter 3) and RobinHood (Chapter 5) relies on being aware of this variability and on continuously adapting the systems' parameters over time. Variability affects many other systems in Internet content delivery and beyond. In fact, several large Internet companies, such as Akamai, have reported to us that the amount of variability has been steadily increasing over the past years, as content delivery architectures are shared by an increasing number and variety of applications [13].

This chapter discusses future work in this area, as well as in the analysis of optimal caching (Chapter 4), and puts into a larger context our proposals of adaptive caching systems.

6.1 Future Directions

Designing and analyzing second-level caching system for CDNs. In our work on AdaptSize, we have focused on maximizing the OHR of the first-level cache, the HOC. CDNs are interested in the OHR, because many Internet objects are small and lead to an isolated (random) I/O operation on the second-level cache, the DC. In our evaluation of AdaptSize, we have seen that improving the OHR had a highly positive effect on the DC, which served fewer files or a larger average size, which lead to more sequential access patterns and lower latency.

Explicitly optimizing the DC is more complex that optimizing the HOC in a CDN. The main optimization goal of DCs is cost (whereas the HOC optimizes performance). Cost is inherently harder to determine than a simple metric such as OHR. A common metric is the byte miss ratio (BMR), as every byte not served by the DC needs to be send over the public Internet, where bandwidth is costly. Optimizing the BMR requires a significantly different approach than OHR as the trade-off between small and large object is less clearly defined. The direction of designing caching systems for BMR is wide open for further study. Furthermore, analyzing the optimal BMR is also a new question. While our FOO analysis (the min cost flow representation) can capture any cost metric, our analysis of the asymptotic correctness of this approach relies heavily on an equal cost per miss. Preliminary experiments with FOO for BMR also suggest that FOO's upper and lower bounds are further apart under this metric.

A further challenge in optimizing the DC is that there are secondary constraints dictated by the hardware. For example, the speed of serving requests from a spinning

disk depends heavily on the position of where data is stored on the disk (outer vs inner ring). On the other hand, flash-based disks are highly sensitive to writes. In fact, most flash disks have a small write budget per day – exceeding this write budget increases the probability of failure for this type of disk. This creates a dual-optimization goal or highly-constraint optimization problem.

Adaptive caching systems promise significant improvements in this area, but we need to further develop our toolset in order to be able to incorporate these additional constraints.

Cross-layer caching optimization in Internet content delivery. Many CDN servers include two to three levels of caching. Furthermore, CDN servers are typically orchestrated in several layers of caching servers, called the edge, parent, and origin layers. Each layer consists of a pool of CDN servers in the same geographic location. A miss in the edge layer, is looked up in the parent layer, and so on.

In AdaptSize we use a Markov-chain model of a single cache to find the optimal size-threshold parameter. In general, we will need to apply our tuning method across multiple levels of caches. Thus, we need to a) find a way to expand the model to capture a vector of states (i.e., the cache state across multiple caches as in [17]), and b) ensure that the model's complexity is still simple enough to have an efficient tuning system. The challenge is that state vectors require multi-dimensional Markov chains, which are inherently hard to solve. We plan to exploit recent mathematical developments such as the Recursive Renewal Reward technique [151, 152] and to pursue a higher-order modeling approaches to these caching systems.

Cross-layer optimization also requires tuning and modeling more complex caching rules. Specifically, the interaction between edge, parent, and origin caches frequently requires splitting the misses of an edge server between multiple parent servers. Additionally, servers at different levels have different optimization goals and see very different traffic patterns. To jointly optimize caching rules across several levels of CDN servers, we need a significantly more powerful tuning approach.

This scenario may well be beyond the reach of detailed performance modeling techniques (such as Markov chains). Some of the ideas from RobinHood may be applied in this case, e.g., deriving a feedback metric for the critical latency path, and combining this with a simple controller. However, this problem remains wide open and future work in this direction relies a strong industrial partner to allow prototyping analysis and design technique in this space.

6.2 Final Thoughts

While variability occurs as a motivating problem throughout this thesis, variability can also be seen as an opportunity. Without size variability, AdaptSize would miss a key predictor that distinguishes the effect of objects on the cache. If all objects were equal, it would be much harder to decide whether to admit and object or not – less information is known about an object that which is new or for which we do not track meta data. In this sense AdaptSize *exploits the variability* inherent to CDN traffic. Similarly, RobinHood relies on the occurrence of temporarily over-provisioned backends so that it can allocate their cache space towards backends that are currently overloaded. If all backends were equally highly loaded, RobinHood cannot do anything to improve the request latency.

In practice, variability can come from different sources. Variability that is inherent in the system and that has a known and static magnitude over time can be seen as an opportunity for systems like AdaptSize and RobinHood. However, Internet system also sometimes face *adversarial variability*, e.g., where outside actors seek to bring down a part of the content delivery architecture to deny service to a set of users. While we tested AdaptSize under randomized and adversarial traffic changes, such synthetics are not enough to certify robustness against any type of variability. Recent examples in the context of neural networks, which can easily be tricked [153, 154] to show unintended behavior, prove that highly-complex systems are not robust against adversarial input and variability. While AdaptSize and RobinHood are conceptually simple, and likely to be more robust than a static size threshold or static cache allocation, we will need to incorporate adversarial variability into our increasingly complex infrastructure.

Bibliography

[1] S. Albers, S. Arora, and S. Khanna, "Page replacement for general caching problems," in *SODA*, vol. 99, 1999, pp. 31–40.

[2] A. Bar-Noy, R. Bar-Yehuda, A. Freund, J. Naor, and B. Schieber, "A unified approach to approximating resource allocation and scheduling," *Journal of the ACM*, vol. 48, no. 5, pp. 1069–1090, 2001.

[3] S. Irani, "Page replacement with multi-size pages and applications to web caching," in *ACM STOC*, 1997, pp. 701–710.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SOSP*, 2007, pp. 205–220.

[5] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding long tails in the cloud," in *USENIX NSDI*, 2013, pp. 329–342.

[6] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013.

[7] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *USENIX NSDI*, 2015.

[8] I. N. Bozkurt, A. Aguirre, B. Chandrasekaran, P. B. Godfrey, G. Laughlin, B. Maggs, and A. Singla, "Why is the internet so slow?!" in *International Conference on Passive and Active Network Measurement*, 2017, pp. 173–187.

[9] S. Sundaresan, N. Magharei, N. Feamster, R. Teixeira, and S. Crawford, "Web performance bottlenecks in broadband access networks," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, 2013, pp. 383–384.

[10] J. Dilley, B. M. Maggs, J. Parikh, H. Prokop, R. K. Sitaraman, and W. E. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.

[11] E. Nygren, R. K. Sitaraman, and J. Sun, "The Akamai Network: A platform for high-performance Internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[12] "Akamai facts and figures," April 2018, available at https://www.akamai.com/us/en/about/facts-figures.jsp, accessed 04/13/18.

[13] "CISCO VNI global IP traffic forecast: The zettabyte era—trends and analysis," May 2015, available at http://goo.gl/wxuvVk, accessed 09/12/16.

[14] M. Chow, K. Veeraraghavan, M. J. Cafarella, and J. Flinn, "Dqbarge: Improving data-quality tradeoffs in large-scale internet services." in *USENIX OSDI*, 2016, pp. 771–786.

[15] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter *et al.*, "Slicer: Auto-sharding for datacenter applications." in *OSDI*, 2016, pp. 739–753.

[16] D. S. Berger, R. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a cdn," in *USENIX NSDI*, March 2017.

[17] D. S. Berger, P. Gland, S. Singla, and F. Ciucu, "Exact analysis of TTL cache networks," *Perform. Eval.*, vol. 79, pp. 2 – 23, 2014, special Issue: Performance 2014.

[18] D. S. Berger, S. Henningsen, F. Ciucu, and J. B. Schmitt, "Maximizing cache hit ratios by variance reduction," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 2, pp. 57–59, 2015.

[19] D. S. Berger, N. Beckmann, and M. Harchol-Balter, "Practical bounds on optimal caching with variable object sizes," *ACM POMACS*, vol. 2, no. 2, p. 32, 2018.

[20] ——, "Practical bounds on optimal caching with variable object sizes," in *ACM SIGMETRICS*, June 2018.

[21] D. S. Berger, B. Berg, T. Zhu, and M. Harchol-Balter, "The case for dynamic cache partitioning for tail latency," in *USENIX NSDI (Extended Abstract)*, March 2017.

[22] D. S. Berger, B. Berg, T. Zhu, S. Sen, and M. Harchol-Balter, "Robinhood: Tail latency aware caching—dynamic reallocation from cache-rich to cache-poor," in *USENIX OSDI*, October 2018.

[23] "Modern network design," November 2016, available at https://www.fastly.com/products/modern-network-design, accessed 02/17/17.

[24] E. Rocca, "Running Wikipedia.org," June 2016, available at https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf, accessed 09/12/16.

[25] R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain, "Overlay networks: An Akamai perspective," in *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[26] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. Dubey, "Architecting to achieve a billion requests per second throughput on a single key-value store server platform," in *ACM ISCA*, 2015, pp. 476–488.

[27] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *USENIX NSDI*, 2014, pp. 429–444.

[28] B. Fan, D. G. Andersen, and M. Kaminsky, "MemC3: Compact and concurrent memcache with dumber caching and smarter hashing," in *USENIX NSDI*, 2013, pp. 371–384.

[29] P.-H. Kamp, "You're doing it wrong," *Communications of the ACM*, vol. 53, no. 7, pp. 55–59, 2010.

[30] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *USENIX NSDI*, 2016, pp. 379–392.

[31] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM CCR*, vol. 45, pp. 52–66, 2015.

[32] N. Gast and B. Van Houdt, "Transient and steady-state regime of a family of list-based cache replacement algorithms," in *ACM SIGMETRICS*, 2015, pp. 123–136.

[33] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *USENIX ATC*, 2015, pp. 57–69.

[34] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: dynamic cloud caching," in *USENIX HotCloud*, 2015.

[35] G. Einziger and R. Friedman, "Tinylfu: A highly efficient cache admission policy," in *IEE Euromicro PDP*, 2014, pp. 146–153.

[36] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of Facebook photo caching," in *ACM SOSP*, 2013, pp. 167–181.

[37] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in *ACM/IEEE CASES*, 2006, pp. 234–241.

[38] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the clock replacement." in *USENIX ATC*, 2005, pp. 323–336.

[39] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement." in *USENIX FAST*, vol. 4, 2004, pp. 187–200.

[40] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache." in *USENIX FAST*, vol. 3, 2003, pp. 115–130.

[41] S. Jiang and X. Zhang, "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS*, vol. 30, no. 1, pp. 31–42, 2002.

[42] H. Bahn, K. Koh, S. H. Noh, and S. Lyul, "Efficient replacement of nonuniform objects in web caches," *IEEE Computer*, vol. 35, no. 6, pp. 65–73, 2002.

[43] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches." in *USENIX ATC*, 2001, pp. 91–104.

[44] L. Cherkasova and G. Ciardo, "Role of aging, frequency, and size in web cache replacement policies," in *High-Performance Computing and Networking*, 2001, pp. 114–123.

[45] S. Jin and A. Bestavros, "GreedyDual* web caching algorithm: exploiting the two sources of temporal locality in web request streams," *Computer Communications*, vol. 24, pp. 174–183, 2001.

[46] D. Starobinski and D. Tse, "Probabilistic methods for web caching," *Perform. Eval.*, vol. 46, pp. 125–137, 2001.

[47] L. Rizzo and L. Vicisano, "Replacement policies for a proxy cache," *IEEE/ACM TON*, vol. 8, pp. 158–170, 2000.

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

[49] K. Shah, A. Mitra, and D. Matani, "An O(1) algorithm for implementing the LFU cache eviction scheme," Stony Brook University, Tech. Rep., 2010.

[50] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies," in *ACM SIGMETRICS*, vol. 27, 1999, pp. 134–143.

[51] C. Aggarwal, J. L. Wolf, and P. S. Yu, "Caching on the world wide web," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 94–107, 1999.

[52] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms." in *USENIX symposium on Internet technologies and systems*, vol. 12, 1997, pp. 193–206.

[53] R. P. Wooster and M. Abrams, "Proxy caching that estimates page load delays," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 977–986, 1997.

[54] M. Abrams, C. R. Standridge, G. Abdulla, E. A. Fox, and S. Williams, "Removal policies in network caches for World-Wide Web documents," in *ACM SIGCOMM*, 1996, pp. 293–305.

[55] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox, "Caching Proxies: Limitations and Potentials," Virginia Polytechnic Institute & State University Blacksburgh, VA, Tech. Rep., 1995.

[56] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *VLDB*, 1994, pp. 439–450.

[57] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," *ACM SIGMOD*, vol. 22, no. 2, pp. 297–306, 1993.

[58] G. Neglia, D. Carra, M. Feng, V. Janardhan, P. Michiardi, and D. Tsigkari, "Access-time aware cache algorithms," in *IEEE ITC*, vol. 1, 2016, pp. 148–156.

[59] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *ACM SoCC*, 2014, pp. 1–14.

[60] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Systems journal*, vol. 9, no. 2, pp. 78–117, 1970.

[61] G. Almasi, C. Caşcaval, and D. A. Padua, "Calculating stack distances efficiently," in *ACM SIGPLAN Notices*, vol. 38, 2002, pp. 37–43.

[62] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield, "Characterizing storage workloads with counter stacks," in *USENIX OSDI*, 2014, pp. 335–349.

[63] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *USENIX FAST*, 2015, pp. 95–110.

[64] W. F. King, "Analysis of demand paging algorithms," in *IFIP Congress (1)*, 1971, pp. 485–490.

[65] E. Gelenbe, "A unified approach to the evaluation of a class of replacement algorithms," *IEEE Transactions on Computers*, vol. 100, pp. 611–618, 1973.

[66] E. G. Coffman and P. J. Denning, *Operating systems theory*. Prentice-Hall, 1973.

[67] J. McCabe, "On serial files with relocatable records," *Operations Research*, vol. 13, pp. 609–618, 1965.

[68] P. Burville and J. Kingman, "On a model for storage and search," *Journal of Applied Probability*, pp. 697–701, 1973.

[69] W. Hendricks, "The stationary distribution of an interesting Markov chain," *Journal of Applied Probability*, pp. 231–233, 1972.

[70] A. Dan and D. Towsley, "An approximate analysis of the LRU and FIFO buffer replacement schemes," in *ACM SIGMETRICS*, 1990, pp. 143–152.

[71] N. Tsukada, R. Hirade, and N. Miyoshi, "Fluid limit analysis of FIFO and RR caching for independent reference model," *Perform. Eval.*, vol. 69, pp. 403–412, Sep. 2012.

[72] P. Flajolet, D. Gardy, and L. Thimonier, "Birthday paradox, coupon collectors, caching algorithms and self-organizing search," *Discrete Applied Mathematics*, vol. 39, pp. 207–229, 1992.

[73] J. A. Fill and L. Holst, "On the distribution of search cost for the move-to-front rule," *Random Structures & Algorithms*, vol. 8, pp. 179–186, 1996.

[74] P. R. Jelenković, "Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities," *The Annals of Applied Probability*, vol. 9, pp. 430–464, 1999.

[75] R. P. Dobrow and J. A. Fill, "The move-to-front rule for self-organizing lists with Markov dependent requests," in *Discrete Probability and Algorithms*. Springer, 1995, pp. 57–80.

[76] E. R. Rodrigues, "The performance of the move-to-front scheme under some particular forms of Markov requests," *Journal of applied probability*, pp. 1089–1102, 1995.

[77] E. G. Coffman and P. Jelenković, "Performance of the move-to-front algorithm with Markov-modulated request sequences," *Operations Research Letters*, vol. 25, pp. 109–118, 1999.

[78] P. R. Jelenković and A. Radovanović, "Least-recently-used caching with dependent requests," *Theoretical computer science*, vol. 326, pp. 293–327, 2004.

[79] A. Panagakis, A. Vaios, and I. Stavrakakis, "Approximate analysis of LRU in the case of short term correlations," *Computer Networks*, vol. 52, pp. 1142–1152, 2008.

[80] K. Psounis, A. Zhu, B. Prabhakar, and R. Motwani, "Modeling correlations in web traces and implications for designing replacement policies," *Computer Networks*, vol. 45, pp. 379–398, 2004.

[81] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance evaluation of the random replacement policy for networks of caches," in *ACM SIGMETRICS/ PERFORMANCE*, 2012, pp. 395–396.

[82] N. E. Young, "Online paging against adversarially biased random inputs," *Journal of Algorithms*, vol. 37, pp. 218–235, 2000.

[83] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-K page replacement algorithm," *JACM*, vol. 46, pp. 92–112, 1999.

[84] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for LRU cache performance," in *ITC*, 2012, p. 8.

[85] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems." in *IEEE INFOCOM*, 2014.

[86] P. Olivier and A. Simonian, "Performance of a cache with random replacement and zipf document popularity," in *VALUETOOLS*, 2013.

[87] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE JSAC*, vol. 20, pp. 1305–1314, 2002.

[88] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for LRU cache performance," in *International Teletraffic Congress*, 2012, pp. 8:1–8:8.

[89] T. Osogami, "A fluid limit for a cache algorithm with general request processes," *Advances in Applied Probability*, vol. 42, no. 3, pp. 816–833, 2010.

[90] G. Bianchi, A. Detti, A. Caponi, and N. Blefari Melazzi, "Check before storing: what is the performance price of content integrity verification in LRU caching?" *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, pp. 59–67, 2013.

[91] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer." in *IBM Systems journal*, vol. 5, 1996, pp. 78–101.

[92] e. a. Mattson, Richard L., "Evaluation techniques for storage hierarchies." in *IBM Systems journal*, vol. 9, 1970, pp. 78–117.

[93] M. Chrobak, G. J. Woeginger, K. Makino, and H. Xu, "Caching is hard—even in the fault model," in *Algorithmica*, vol. 63, 2012, pp. 781–794.

[94] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[95] L. Folwarczný and J. Sgall, "General caching is hard: Even with small pages," *Algorithmica*, vol. 79, no. 2, pp. 319–339, 2017.

[96] C. Koufogiannakis and N. E. Young, "A nearly linear-time PTAS for explicit fractional packing and covering linear programs," *Algorithmica*, vol. 70, no. 4, pp. 648–674, 2014.

[97] B. S. Gill, "On multi-level exclusive caching: offline optimality and why promotions are better than demotions," in *USENIX FAST*, 2008, p. 4.

[98] M. Kallahalla and P. J. Varman, "Pc-opt: optimal offline prefetching and caching for parallel i/o systems," *IEEE Transactions on Computers*, vol. 51, no. 11, pp. 1333–1344, 2002.

[99] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *IEEE INFOCOM*, 2016, pp. 1–9.

[100] P. Hillmann, T. Uhlig, G. D. Rodosek, and O. Rose, "Simulation and optimization of content delivery networks considering user profiles and preferences of internet service providers," in *IEEE Winter Simulation Conference*, 2016, pp. 3143–3154.

[101] S. Shukla and A. Abouzeid, "Optimal device aware caching," *IEEE Transactions on Mobile Computing*, vol. PP, no. 99, pp. 1–1, September 2016.

[102] D.-Z. Du and P. M. Pardalos, *Handbook of combinatorial optimization*, 2nd ed. Springer, 2013.

[103] H. ElAarag, S. Romano, and J. Cobb, *Web Proxy Cache Replacement Strategies: Simulation, Implementation, and Performance Evaluation*, ser. Springer Briefs in Computer Science. Springer London, 2013.

[104] R. Motwani and P. Raghavan, *Randomized algorithms*. Chapman & Hall/CRC, 2010.

[105] P.-H. Kamp, "Varnish LRU architecture," June 2007, available at https://www.varnish-cache.org/trac/wiki/ArchitectureLRU, accessed 09/12/16.

[106] J.-Y. Le Boudec, D. McDonald, and J. Mundinger, "A generic mean field convergence result for systems of interacting objects," in *Quantitative Evaluation of Systems*. IEEE, 2007, pp. 3–18.

[107] F. Velázquez, K. Lyngstøl, T. Fog Heen, and J. Renard, *The Varnish Book for Varnish 4.0*. Varnish Software AS, March 2016.

[108] P. Graziano, "Speed up your web site with Varnish," *Linux Journal*, vol. 2013, no. 227, p. 4, 2013.

[109] N. C. Fofack, M. Dehghan, D. Towsley, M. Badov, and D. L. Goeckel, "On the performance of general cache networks," in *VALUETOOLS*, 2014, pp. 106–113.

[110] P. P. Petrushev and V. A. Popov, *Rational approximation of real functions*. Cambridge University Press, 2011, vol. 28.

[111] P. Pošík, W. Huyer, and L. Pál, "A comparison of global search algorithms for continuous black box optimization," *Evolutionary computation*, vol. 20, no. 4, pp. 509–541, 2012.

[112] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[113] P.-H. Kamp, "Varnish notes from the architect," 2006, available at https://www.varnish-cache.org/docs/trunk/phk/notes.html, accessed 09/12/16.

[114] S. Godard, "Iostat," 2015, available at http://goo.gl/JZmbUp, accessed 09/12/16.

[115] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab *et al.*, "Scaling memcache at facebook," in *USENIX NSDI*, 2013, pp. 385–398.

[116] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[117] W. Feller, *An introduction to probability theory and its applications*. John Wiley & Sons, 2008, vol. 2.

[118] A. V. Doumas and V. G. Papanicolaou, "The coupon collector's problem revisited: asymptotics of the variance," *Advances in Applied Probability*, vol. 44, no. 1, pp. 166–195, 2012.

[119] J. Moriarty and P. Neal, "The generalized coupon collector problem," *Journal of Applied Probability*, vol. 45, pp. 621–29, 2008.

[120] W. Xu and A. K. Tang, "A generalized coupon collector problem," *Journal of Applied Probability*, vol. 48, no. 4, pp. 1081–1094, 2011.

[121] E. Anceaume, Y. Busnel, and B. Sericola, "New results on a generalized coupon collector problem using markov chains," *Journal of Applied Probability*, vol. 52, no. 2, pp. 405–418, 2015.

[122] S. I. Daitch and D. A. Spielman, "Faster approximate lossy generalized flow via interior point algorithms," in *ACM STOC*, 2008, pp. 451–460.

[123] R. Becker and A. Karrenbauer, "Adaptive caching networks with optimality guarantees," in *ISAAC*, 2014, pp. 753–765.

[124] ——, "A combinatorial o(m 3/2)-time algorithm for the min-cost flow problem," *arXiv preprint arXiv:1312.3905*, 2013.

[125] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *IEEE INFOCOM*, 1999, pp. 126–134.

[126] B. Karrer and M. E. Newman, "Random graph models for directed acyclic networks," *Physical Review E*, vol. 80, no. 4, p. 046110, 2009.

[127] B. Bollobás and G. Brightwell, "The height of a random partial order: concentration of measure," *The Annals of Applied Probability*, pp. 1009–1018, 1992.

[128] N. Beckmann, H. Chen, and A. Cidon, "Lhd: Improving hit rate by maximizing hit density," in *USENIX NSDI.*, 2018.

[129] SNIA, "MSR Cambridge Traces," http://iotta.snia.org/traces/388, 2008.

[130] W. Hoeffding, "A non-parametric test of independence," *The annals of mathematical statistics*, pp. 546–557, 1948.

[131] W. W. Daniel *et al.*, *Applied nonparametric statistics*. Houghton Mifflin, 1978.

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

[133] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *EUROSYS*, 2015, p. 5.

[134] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *USENIX ATC*, 2017.

[135] E. R. G. on Combinatorial Optimization, "Coin-or::lemon library," 2015, available at http://lemon.cs.elte.hu/trac/lemon, accessed 10/21/17.

[136] O. Tange, "Gnu parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, Feb 2011. [Online]. Available: http://www.gnu.org/s/parallel

[137] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[138] K. Psounis and B. Prabhakar, "A randomized web-cache replacement scheme," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3. IEEE, 2001, pp. 1407–1415.

[139] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *IEEE Transactions on Computers*, 1994.

[140] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *IEEE HPCA.*, 2015.

[141] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017.

[142] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SOSP*, vol. 41, no. 6, 2007, pp. 205–220.

[143] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li *et al.*, "Tao: Facebook's distributed data store for the social graph." in *USENIX ATC*, 2013, pp. 49–60.

[144] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An analysis of Facebook photo caching," in *SOSP*, 2013.

[145] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *USENIX NSDI*, 2016.

[146] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song, "Kraken: leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *USENIX OSDI*, 2016, pp. 635–650.

[147] J. Dean and L. A. Barroso, "The tail at scale," *CACM*, vol. 56, no. 2, pp. 74–80, 2013.

[148] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," in *ACM SIGPLAN Notices*, vol. 49, no. 4, 2014, pp. 729–742.

[149] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "Fairride: Near-optimal, fair cache sharing," in *NSDI*, 2016, pp. 393–406.

[150] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS*, 2012, pp. 53–64.

[151] T. Osogami, M. Harchol-Balter, and A. Scheller-Wolf, "Analysis of cycle stealing with switching times and thresholds," in *ACM SIGMETRICS*, San Diego, CA, June 2003, pp. 184–195.

[152] A. Gandhi, S. Doroudi, M. Harchol-Balter, and A. Scheller-Wolf, "Exact analysis of the M/M/k/setup class of markov chains via recursive renewal reward," in *ACM SIGMETRICS*, 2013.

[153] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE EuroS&P*, 2016, pp. 372–387.

[154] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 427–436.

Curriculum Vitae

Daniel S. Berger

Education

2014 - 2018	**PhD** in *Computer Science* Technische Universität Kaiserslautern, Germany Thesis: "*Design and Analysis of Adaptive Caching Techniques for Internet Content Delivery*" Committee: Jens Schmitt (Kaiserslautern), Mor Harchol-Balter (CMU), Florin Ciucu (U. Warwick)
2012 - 2014	**Master of Science** in *Computer Science* Technische Universität Kaiserslautern, Germany Specialization: information and communication systems Thesis: "*Towards Analytical Cache Models for Feedforward Networks*" Committee: Jens Schmitt (Kaiserslautern), Florin Ciucu (U. Warwick)
2009 - 2012	**Bachelor of Science** in *Computer Science* Technische Universität Kaiserslautern, Germany Specialization: information and communication systems Thesis: "*Effects and Factors of Network Instability:*" Committee: Jens Schmitt (Kaiserslautern), Martin Karsten (U. Waterloo)

Professional Experience

since 2011	**Teaching Assistant and Tutor** at *Distributed Computer Systems Lab* Technische Universität Kaiserslautern, Germany

	Lectures: Performance Evaluation of Distributed Systems (89-4245), Mobile Computing (89-4271), Distributed and Networked Systems (89-4111), Kommunikationssysteme/Computer Networks (89-0013)
2/2013–10/2013	**Internship** at *Telekom Innovation Laboratories+T* Berlin, Germany
7/2009–10/2012	**Research Assistant** at *German Cancer Research Center* Heidelberg, Germany Department of Radiology

Honors & Awards

2016	**M.Sc. Research Price**, German Informatics Society. Technical committee on Measurements, Modeling and Evaluation of Computer Systems.
2015	**Invited to the 3rd Heidelberg Laureate Forum**, selected as one of 200 students to meet Turing and Abel price laureates.
2014	**Best Paper Award at IFIP Performance**, for "Exact Analysis of TTL Cache Networks", International Federation for Information Processing. Working group on Computer System Modeling
2014	**Best Student Paper Award at ACM WiSec**, for "Gaining Insight on Friendly Jamming in a Real-World IEEE 802.11 Network", ACM Special Interest Group on Security, Audit and Control.
2015	**Mobility Grant**, Kaiserslautern Network for the Promotion of Young Scientists.
2012–2017	**Informatik Promotionsprogram**, Department of Computer Science, University of Kaiserslautern.
2008–2013	**Full Scholarship Holder**, Deutsche Studien Stiftung.

Publications

Peer-Reviewed

- *RobinHood: Tail Latency Aware Caching - Dynamic Reallocation from Cache-Rich to Cache-Poor.*
  (Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, Mor Harchol-Balter), In Proceedings of USENIX OSDI (Symposium on Operating Systems Design and Implementation), October 2018.

- *Practical Bounds on Optimal Caching with Variable Object Sizes.*
  (Daniel S. Berger, Nathan Beckmann, Mor Harchol-Balter), In Proceedings of ACM SIGMETRICS (International Conference on Measurement and Modeling of Computer Systems), June 2018.

- *AdaptSize: Orchestrating the Hot Object Memory Cache in a CDN.*
  (Daniel S. Berger, Ramesh Sitaraman, Mor Harchol-Balter), In Proceedings of USENIX NSDI (Symposium on Networked Systems Design and Implementation), March 2017.

- *The Case for Dynamic Cache Partitioning.*
  (Daniel S. Berger, Benjamin Berg, Timothy Zhu, Mor Harchol-Balter), Extended Abstract, In Proceedings of USENIX NSDI (Symposium on Networked Systems Design and Implementation), March 2017.

- *Maximizing Cache Hit Ratios by Variance Reduction.*
  (Daniel S. Berger, Sebastian A. Henningsen, Florin Ciucu, Jens B. Schmitt), Workshop on Mathematical Performance Modeling and Analysis (ACM SIGMETRICS Performance Evaluation Review, volume 43, 2015), June 2015.

- *Exact analysis of TTL cache networks.*
  (Daniel S. Berger, Philipp Gland, Sahil Singla, Florin Ciucu), In IFIP Performance (International Symposium on Computer Performance, Modeling, Measurements and Evaluation), October 2014.

- *Stochastic Bounds on Inter-Miss Times from TTL Caches.*
  (Daniel S. Berger, Florin Ciucu), Praxis der Informationsverarbeitung und Kommunikation 37(2): 109-120 (2014).

- *Exact analysis of TTL cache networks: the case of caching policies driven by stopping times.*
  (Daniel S. Berger, Philipp Gland, Sahil Singla, Florin Ciucu), Extended Abstract, In Proceedings of ACM SIGMETRICS (International Conference on Measurement and Modeling of Computer Systems), June 2014.

Technical Reports

- *Practical Bounds on Optimal Caching with Variable Object Sizes* (Daniel S. Berger, N. Beckmann, M. Harchol-Balter), CoRR abs/1711.03709, December 2017.

- *Achieving High Cache Hit Ratios for CDN Memory Caches with Size-aware Admission* (Daniel S. Berger, RK. Sitaraman, M. Harchol-Balter), Technical Report University of Kaiserslautern / CMU-CS-16-120, June 2016.

- *Exact Analysis of TTL Cache Networks: The Case of Caching Policies driven by Stopping Times* (Daniel S. Berger, P. Gland, S. Singla, F. Ciucu), Tech Report University of Kaiserslautern / arXiv:1402.5987, February 2014.

Invited Talks

- *Towards Practical Bounds on Optimal Caching with Variable Object Sizes.*
  CMU CS Theory Lunch, Pittsburgh, December 2017.

- *Can caching be used to resolve data center load imbalances?*
  Microsoft Research Lab, New York City, September 2017.

- *Can caching be used to resolve data center load imbalances?*
  The New York Times Tech Lab, New York City, April 2017.

- *Adaptive Caching Techniques for CDN Memory Caches.*
  Google CDN Research Lab, Boston, March 2017.

- *Adaptive Caching Techniques for CDN Memory Caches.*
  Facebook HQ, Menlo Park, October 2016.

- *Maximizing Cache Hit Ratios of CDN Memory Caches with Adaptive Size-Aware Admission Control.*
  Intel Science and Technology Center, Pittsburgh, September 2016.

- *Towards Analytical Cache Models for Feedforward Networks.*
  GI/ITG Conference on Measurement, Modeling and Evaluation of Computing Systems, Muenster, April 2016.

- *New mathematical techniques for the analysis of TTL caches.*
  Probability seminar, Division of Applied Mathematics, Brown University, Providence, May 2015.

- *New mathematical techniques for the analysis of TTL caches.*
  SQUALL seminar series, School of Computer Science, Carnegie Mellon University, Pittsburgh, USA. April 2015.